Николай Прохоренок

Основы Java

2-е издание

- Базовый синтаксис языка Java
- Объектно-ориентированное программирование
- Работа с файлами и каталогами
- Stream API
- Функциональные интерфейсы
- Лямбда-выражения
- Работа с базой данных MySQL
- Получение данных из Интернета
- Интерактивная оболочка JShell

Материалы на www.bhv.ru

bhv®

Николай Прохоренок

Основы Java

2-е издание

Санкт-Петербург
«БХВ-Петербург»
2019

УДК 004.438 Java
ББК 32.973.26-018.1
П84

Прохоренок Н. А.

П84 Основы Java. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2019. — 768 с.: ил.

ISBN 978-5-9775-4012-4

Описан базовый синтаксис языка Java: типы данных, операторы, условия, циклы, регулярные выражения, лямбда-выражения, ссылки на методы, объектно-ориентированное программирование. Рассмотрены основные классы стандартной библиотеки, получение данных из сети Интернет, работа с базой данных MySQL. Книга содержит большое количество практических примеров, помогающих начать программировать на языке Java самостоятельно. Весь материал тщательно подобран, хорошо структурирован и компактно изложен, что позволяет использовать книгу как удобный справочник. Во втором издании добавлена глава по Java 11 и описано большинство нововведений: модули, интерактивная оболочка JShell, инструкция var и др. Электронный архив с примерами находится на сайте издательства.

Для программистов

УДК 004.438 Java
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Сависте*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн обложки	*Елизаветы Романовой*

Подписано в печать 02.07.19.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 61,92.
Тираж 1500 экз. Заказ № 9429.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-5-9775-4012-4

Оглавление

Введение

Добро пожаловать в мир Java!

Java — это объектно-ориентированный язык программирования высокого уровня, предназначенный для самого широкого круга задач. С его помощью можно обрабатывать различные данные, создавать изображения, работать с базами данных, разрабатывать Web-сайты, мобильные приложения и приложения с графическим интерфейсом. Java — язык кроссплатформенный, позволяющий создавать программы, которые будут работать во всех операционных системах. В этой книге мы рассмотрим основы языка Java SE (*SE* — Standard Edition) применительно к операционной системе Windows.

Язык Java часто путают с языком JavaScript. Помните — это абсолютно разные языки и по синтаксису, и по назначению. JavaScript работает в основном в Web-браузере, тогда как Java является универсальным языком, хотя он также может работать в Web-браузере (в виде апплета или приложения JavaFX).

Синтаксис языка Java очень похож на синтаксис языков C++ и C#. Если вы знакомы с этими языками, то изучить язык Java не составит вам особого труда. Однако если вы этих языков не знаете или вообще никогда не программировали, то у вас могут возникнуть некоторые сложности. Дело в том, что язык Java, как уже отмечалось, представляет собой язык объектно-ориентированный, и, в отличие от C++, объектно-ориентированный стиль программирования (ООП-стиль) является для него обязательным. Это означает, что с самой первой программы вы погрузитесь в мир ООП, а мне, как автору книги, придется много раз решать проблему, что было раньше: курица или яйцо. Ведь объяснить начинающему программисту преимущества ООП-стиля программирования, когда он не знает, что такое «переменная», очень сложно. И с самой первой программы нам необходимо будет разъяснить, что такое «класс» и «модификатор доступа», что такое «метод» и чем отличается обычный метод от статического, что такое «переменная» и «массив», что такое «поток вывода» и др. Поэтому, если вы не знакомы с языками C++ и C#, то в обязательном порядке вам следует прочитать книгу несколько раз, — только в этом случае вы сможете изучить язык Java.

Если вам ранее доводилось читать мои книги по PHP, JavaScript, Perl, Python или C++, то изучить язык Java по этой книге вам будет гораздо проще, т. к. вы уже

знаете структуру моих книг. Она в большинстве случаев совпадает, но не на все сто процентов, т. к. языки эти все-таки разные и обеспечивают выполнение различных задач. Если же структура моих книг вам пока не знакома, то ничего страшного, сейчас я ее вкратце представлю.

- *Глава 1* является вводной. Мы установим необходимое программное обеспечение, настроим среду разработки, скомпилируем и запустим первую программу — как из командной строки, так и из редактора Eclipse. Кроме того, мы вкратце разберемся со структурой программы, а также научимся выводить результат работы программы и получать данные от пользователя.
- В *главе 2* мы познакомимся с переменными и типами данных в языке Java, а в *главе 3* рассмотрим различные операторы, предназначенные для выполнения определенных действий с данными. Кроме того, в этой главе мы изучим операторы ветвления и циклы, позволяющие изменить порядок выполнения программы.
- *Глава 4* полностью посвящена работе с числами. Вы узнаете, какие числовые типы данных поддерживает язык Java, научитесь применять различные методы, генерировать случайные числа и др.
- *Глава 5* познакомит вас с массивами в языке Java. Вы научитесь создавать как одномерные массивы, так и многомерные, перебирать элементы массива, сортировать, выполнять поиск значений, преобразовывать массив в строку и др.
- *Глава 6* полностью посвящена работе с символами и строками. В этой главе вы также научитесь работать с различными кодировками, настраивать локаль, выполнять форматирование строки и осуществлять поиск внутри строки. А в *главе 7* мы рассмотрим регулярные выражения, которые позволят осуществить сложный поиск в строке в соответствии с шаблоном.
- *Главы 8* и *9* познакомят вас с двумя способами работы с датой и временем. Первый способ доступен с самой первой версии языка Java, а второй был добавлен в 8-й его версии.
- В *главе 10* вы научитесь создавать пользовательские методы, а в *главе 11* познакомитесь с объектно-ориентированным программированием, позволяющим использовать код многократно.
- *Глава 12* познакомит вас с интерфейсами, а *глава 13* — с обобщенными типами данных.
- *Главы 14* и *15* посвящены описанию каркаса коллекций. В этот каркас входят обобщенные классы, реализующие различные структуры данных: динамические списки, очереди, множества и словари.
- В *главе 16* вы познакомитесь со способом организации больших программ в виде пакетов и модулей, а также научитесь создавать JAR-архивы. Если вас очень интересует способ запуска Java-программ двойным щелчком на значке файла, то именно в этой главе вы найдете ответ на свой вопрос.
- В *главе 17* мы рассмотрим способы поиска ошибок в программе и научимся отлаживать программу в редакторе Eclipse.

- *Главы 18*, *19* и *20* научат вас работать с файлами и каталогами, читать и писать файлы в различных форматах.
- *Глава 21* познакомит с потоками данных (Stream API), введенных в 8-й версии языка.
- В *главе 22* мы рассмотрим способы получения данных из сети Интернет, а в *главе 23* научимся работать с базой данных MySQL.
- *Глава 24* познакомит вас с многопоточными приложениями, позволяющими значительно повысить производительность программы за счет параллельного выполнения задач несколькими потоками управления.
- И, наконец, в *главе 25* мы рассмотрим отличия Java 11 от Java 10, а также возможности, добавленные в новой версии языка.

Все листинги из этой книги вы найдете в файле Listings.doc, электронный архив с которым можно загрузить с FTP-сервера издательства «БХВ-Петербург» по ссылке: **ftp://ftp.bhv.ru/9785977540124.zip** или со страницы книги на сайте **www.bhv.ru** (см. *приложение*).

Желаю приятного чтения и надеюсь, что эта книга выведет вас на верный путь в мире профессионального программирования.

ГЛАВА 1

Первые шаги

Прежде чем мы начнем рассматривать синтаксис языка, необходимо сделать два замечания. Во-первых, не забывайте, что книги по программированию нужно не только читать, но и выполнять все приведенные в них примеры, а также экспериментировать, что-либо в этих примерах изменяя. Поэтому, если вы удобно устроились на диване и настроились просто читать, у вас практически нет шансов изучить язык! Во-вторых, помните, что прочитать эту книгу один раз недостаточно — ее вы должны выучить наизусть! Это же основы языка! Сколько на это уйдет времени, зависит от ваших способностей и желания. Как минимум, вы должны знать структуру книги.

Чем больше вы будете делать самостоятельно, тем большему научитесь. Обычно после первого прочтения многое непонятно, после второго прочтения — некоторые вещи становятся понятнее, после третьего — еще лучше, ну, а после *N*-го прочтения — не понимаешь, как могло быть что-то непонятно после первого прочтения. Повторение — мать учения. Наберитесь терпения, и вперед!

Итак, приступим к изучению языка Java и начнем с установки необходимых программ.

> ***НА ЗАМЕТКУ***
>
> Нововведения, появившиеся в версиях 9 и 10, помечены в книге метками *Java9* и *Java10*. Найти страницы, содержащие эти метки, можно с помощью предметного указателя, расположенного в конце книги.

1.1. Установка Java SE Development Kit (JDK)

Для изучения языка Java необходимо установить на компьютер комплект разработчика Java Development Kit (сокращенно JDK), который включает компилятор, стандартные библиотеки классов, различные утилиты и исполнительную среду Java Runtime Environment (сокращенно JRE). Для этого переходим на страницу:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

и скачиваем дистрибутив, соответствующий операционной системе вашего компьютера. Поскольку в этой книге мы станем рассматривать основы языка примени-

тельно к 64-битной операционной системе Windows, скачать вам следует файл jdk-10_windows-x64_bin.exe. Соглашаемся с лицензионным соглашением, скачиваем файл и запускаем его. Процесс установки полностью автоматизирован и не нуждается в особых комментариях. Во всех случаях соглашаемся с настройками по умолчанию.

После установки в каталоге C:\Program Files\Java будут созданы две папки:

- jre-10 — содержит файлы исполнительной среды Java Runtime Environment (JRE). Соответственно, в папке C:\Program Files\Java\jre-10\bin расположены исполняемые файлы и библиотеки, которых достаточно для выполнения программ, написанных на языке Java;
- jdk-10 — содержит файлы комплекта разработчика Java Development Kit (JDK). Соответственно, в папке C:\Program Files\Java\jdk-10\bin расположен компилятор (javac) и различные утилиты.

Чтобы проверить правильность установки, запускаем приложение **Командная строка** (рис. 1.1) и выполняем следующую команду:

```
java -version
```

Результат должен быть примерно следующим:

```
C:\Users\Unicross>java -version
java version "10" 2018-03-20
Java(TM) SE Runtime Environment 18.3 (build 10+46)
Java HotSpot(TM) 64-Bit Server VM 18.3 (build 10+46, mixed mode)
```

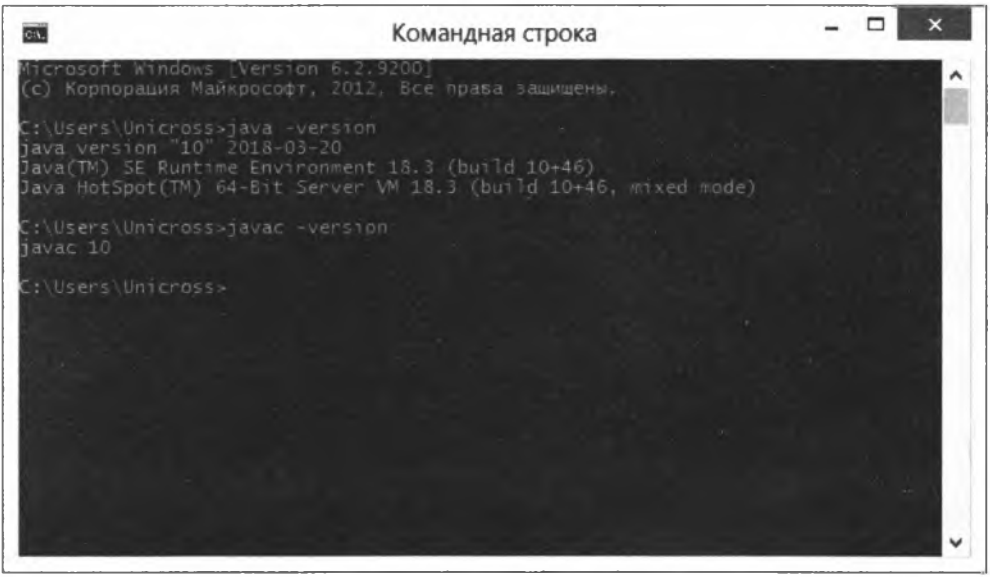

Рис. 1.1. Приложение Командная строка

Вполне возможно, что вы никогда не пользовались командной строкой и не знаете, как запустить это приложение. Давайте рассмотрим некоторые способы его запуска в Windows:

- через поиск находим приложение **Командная строка**;
- нажимаем комбинацию клавиш <Windows>+<R>. В открывшемся окне вводим cmd и нажимаем кнопку **ОК**;
- находим файл cmd.exe в папке C:\Windows\System32;
- в Проводнике щелкаем правой кнопкой мыши на свободном месте списка файлов, удерживая при этом нажатой клавишу <Shift>, и из контекстного меню выбираем пункт **Открыть окно команд**.

Если результат выглядит так:

```
C:\Users\Unicross>java -version
"java" не является внутренней или внешней
командой, исполняемой программой или пакетным файлом.
```

то необходимо добавить путь к папке C:\Program Files\Java\jre-10\bin в системную переменную PATH. Программа установки обычно прописывает путь к файлу java.exe автоматически, но вполне возможно, что вы изменили настройки в процессе установки и в результате получили эту ошибку.

Чтобы изменить системную переменную в Windows, переходим в **Параметры | Панель управления | Система и безопасность | Система | Дополнительные параметры системы**. В результате откроется окно **Свойства системы** (рис. 1.2). На вкладке **Дополнительно** нажимаем кнопку **Переменные среды**. В открывшемся

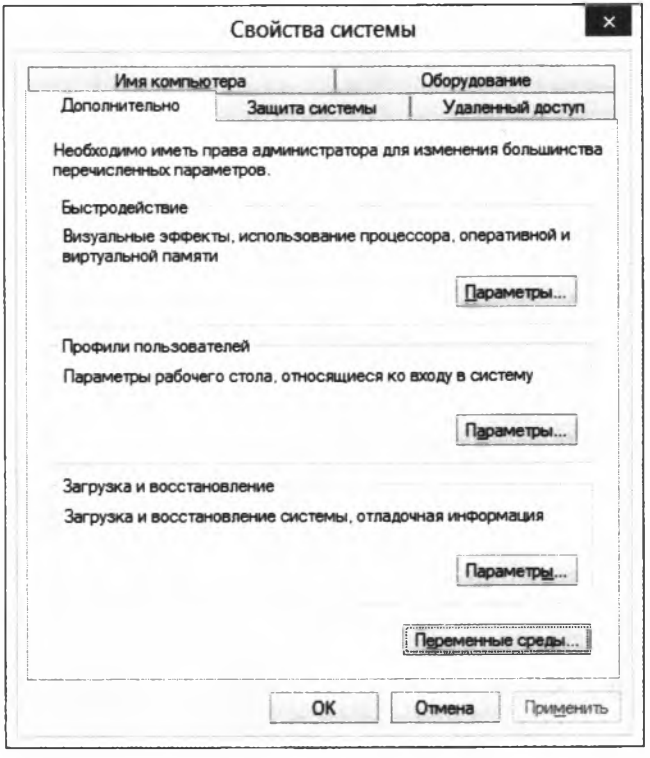

Рис. 1.2. Окно **Свойства системы**

окне (рис. 1.3) в списке **Системные переменные** выделяем строку с переменной **Path** и нажимаем кнопку **Изменить**. В открывшемся окне (рис. 1.4) изменяем значение в поле **Значение переменной** — для этого переходим в конец существующей строки, ставим точку с запятой, а затем вводим путь к папке C:\Program Files\Java\jre-10\bin:

```
<Текущее значение>;C:\Program Files\Java\jre-10\bin
```

Сохраняем изменения и повторно проверяем установку.

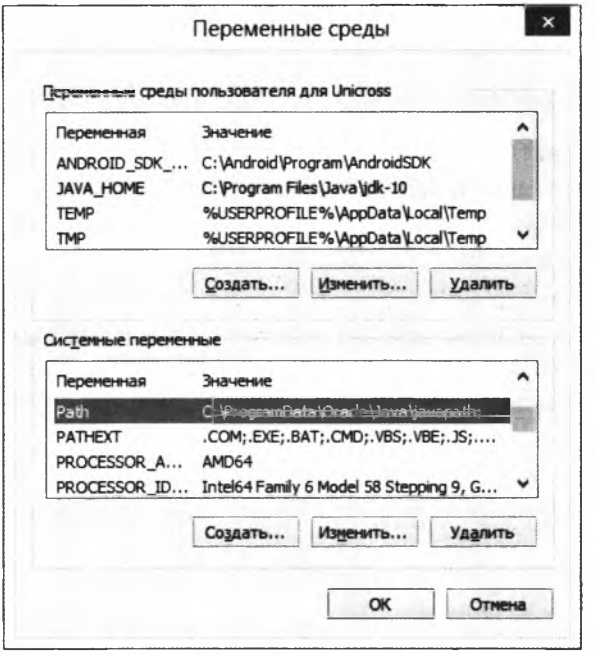

Рис. 1.3. Окно **Переменные среды**

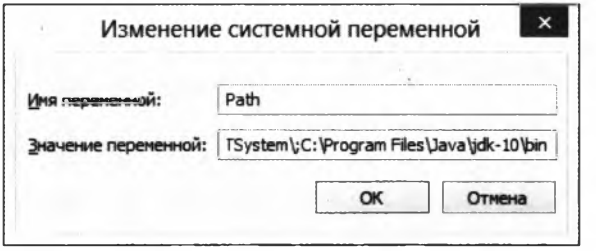

Рис. 1.4. Окно **Изменение системной переменной**

Внимание!

Случайно не удалите существующее значение переменной `PATH`, **иначе другие приложения перестанут запускаться.**

Помимо добавления пути к папке C:\Program Files\Java\jre-10\bin в переменную `PATH`, во избежание проблем с настройками различных редакторов необходимо прописать

переменную окружения JAVA_HOME и присвоить ей путь к папке с установленным JDK. Делается это также в окне **Переменные среды** (см. рис. 1.3), но в списке переменных для пользователя, хотя вы можете добавить ее и в список системных переменных. Нажимаем в этом окне кнопку **Создать**, в поле **Имя переменной** вводим значение JAVA_HOME, а в поле **Значение переменной** — C:\Program Files\Java\jdk-10. Нажимаем кнопку **OK**. Запускаем командную строку и проверяем установку переменной JAVA_HOME:

```
C:\Users\Unicross>set JAVA_HOME
JAVA_HOME=C:\Program Files\Java\jdk-10
```

1.2. Первая программа

При изучении языков программирования принято начинать с программы, выводящей надпись **Hello, world!** в окно консоли. Не будем нарушать традицию и продемонстрируем, как это выглядит на языке Java (листинг 1.1).

Листинг 1.1. Первая программа

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, world!");
   }
}
```

> ***ЭЛЕКТРОННЫЙ АРХИВ***
>
> Напомним, что все листинги из этой книги вы найдете в файле *Listings.doc*, электронный архив с которым можно загрузить с FTP-сервера издательства «БХВ-Петербург» по ссылке: **ftp://ftp.bhv.ru/9785977540124.zip** или со страницы книги на сайте **www.bhv.ru** (см. *приложение*).

На этом этапе мы пока не станем рассматривать структуру приведенной программы. Просто поверьте мне на слово, что эта программа выводит надпись **Hello, world!** Сейчас мы попробуем скомпилировать программу и запустить ее на исполнение. Но прежде создадим следующую структуру каталогов:

```
book
JavaSE
    eclipse
    projects
```

Папки book и JavaSE лучше разместить в корне какого-либо диска. В моем случае это будет диск C:, следовательно, пути к содержимому папок: C:\book и C:\JavaSE. Можно создать папку и в любом другом месте, но в пути не должно быть русских букв и пробелов — только английские буквы, цифры, тире и подчеркивание. Остальных символов лучше избегать, если не хотите проблем с компиляцией и запуском программ.

В папке C:\book мы станем размещать наши тестовые программы и тренироваться при изучении работы с файлами и каталогами. Некоторые методы без проблем могут при неправильном действии удалить все дерево каталогов, поэтому для экспериментов мы воспользуемся отдельной папкой, а не папкой C:\JavaSE, в которой у нас будет находиться все сокровенное, — обидно, если по случайности мы удалим все установленные программы и проекты.

Кстати, рекомендация не использовать русские буквы относится и к имени пользователя. Многие программы сохраняют настройки в каталоге C:\Users\<Имя пользователя>. В частности, по умолчанию настройки виртуальных устройств в Android SDK сохраняются в папке C:\Users\<Имя пользователя>\.android. В результате в пути окажутся русские буквы, которые могут стать причиной множества проблем.

Внутри папки JavaSE у нас созданы две вложенные папки:

- eclipse — в эту папку мы установим редактор Eclipse;
- projects — в этой папке мы станем сохранять проекты из различных редакторов, например, из того же Eclipse.

Для создания файла с программой можно воспользоваться любым текстовым редактором. Нам больше всего нравится редактор Notepad++. Итак, создаем текстовый файл с названием HelloWorld и расширением java. Обратите внимание на регистр символов в названии файла. Только так! Кроме того, это название в точности совпадает с именем класса, указанным после ключевого слова `class` внутри программы. Теперь несколько слов о расширении. Буквы в расширении набираются строчными. При сохранении файла обратите внимание, чтобы в конец не добавилось расширение txt. Программа Блокнот по умолчанию делает именно так, что приводит к ошибкам. Название файла должно быть HelloWorld.java, а не — HelloWorld.java.txt.

Набираем текст программы из листинга 1.1 и сохраняем файл в папке C:\book. Теперь необходимо скомпилировать программу. Для этого открываем командную строку, делаем текущей папку C:\book:

```
C:\Users\Unicross>cd C:\book
```

Предварительно проверим доступную версию компилятора, выполнив команду:

```
javac -version
```

Результат должен быть таким:

```
C:\book>javac -version
javac 10
```

Скорее всего, вы получите сообщение:

```
"javac" не является внутренней или внешней
командой, исполняемой программой или пакетным файлом.
```

Чтобы избежать этой проблемы, необходимо добавить путь к папке C:\Program Files\Java\jdk-10\bin в конец текущего значения системной переменной `PATH` (через точку с запятой). Порядок изменения значения системной переменной `PATH` мы уже

рассматривали в предыдущем разделе. Добавляем путь и заново выполняем команду, предварительно перезапустив командную строку, чтобы новое значение системной переменной стало доступным.

Запускаем процесс компиляции с помощью следующего кода:

```
C:\book>javac HelloWorld.java
```

Вполне возможно, что вы получите следующее сообщение об ошибке:

```
C:\book>javac helloworld.java
helloworld.java:1: error: class HelloWorld is public, should be
declared in a file named HelloWorld.java
public class HelloWorld {
       ^
1 error
```

Как можно видеть, регистр в набранном вами имени файла не совпадает с регистром названия класса. Примерно такое же сообщение вы получите, если имя файла не будет совпадать с названием класса.

Если имя файла будет иметь расширение txt (например, HelloWorld.java.txt), то компилятор не сможет найти файл HelloWorld.java и выведет соответствующее сообщение:

```
C:\book>javac HelloWorld.java
javac: file not found: HelloWorld.java
Usage: javac <options> <source files>
use -help for a list of possible options
```

Возможны и другие сообщения об ошибках — например, если при наборе текста программы была допущена опечатка или вы забыли добавить какой-либо символ. Так, если вместо слова `public` набрать `publik`, мы получим следующее сообщение об ошибке:

```
C:\book>javac HelloWorld.java
HelloWorld.java:1: error: class, interface, or enum expected
publik class HelloWorld {
^
1 error
```

В любом случае ошибку нужно найти и исправить.

Если компиляция прошла успешно, то никаких сообщений не появится, а отобразится приглашение для ввода следующей команды:

```
C:\book>javac HelloWorld.java

C:\book>
```

При этом в папке C:\book создастся одноименный файл с расширением class, который содержит специальный байт-код для виртуальной машины Java. Для запуска программы в командной строке набираем команду:

```
C:\book>java HelloWorld
```

Если все выполнено правильно, то получим приветствие:

```
Hello, world!
```

Обратите внимание, программе java.exe мы передаем имя файла с байт-кодом без расширения class. Если расширение указать, то получим следующее сообщение об ошибке:

```
C:\book>java HelloWorld.class
Error: Could not find or load main class HelloWorld.class
Caused by: java.lang.ClassNotFoundException: HelloWorld.class
```

Итак, вначале мы создаем файл с расширением java, затем запускаем процесс компиляции с помощью программы javac.exe и получаем файл с расширением class, который можно запустить на выполнение с помощью программы java.exe. Вот так упрощенно выглядит процесс создания программ на языке Java. После компиляции мы получаем не EXE-файл, а всего лишь файл с байт-кодом. Зато этот байт-код может быть запущен на любой виртуальной машине Java вне зависимости от архитектуры компьютера — тем самым и обеспечивается кроссплатформенность программ на языке Java.

Прежде чем мы продолжим, необходимо сказать несколько слов о кодировке файла с программой. По умолчанию предполагается, что файл с программой сохранен в кодировке, используемой в системе по умолчанию. В русской версии Windows этой кодировкой является windows-1251. Не следует полагаться на кодировку по умолчанию. Вместо этого при компиляции нужно указать кодировку явным образом с помощью флага `-encoding`:

```
C:\book>javac -encoding utf-8 HelloWorld.java
```

В этой инструкции мы указали компилятору, что файл с программой был сохранен в кодировке UTF-8. Мы будем пользоваться именно этой кодировкой, т. к. она позволяет хранить любые символы. При использовании редактора Notepad++ убедитесь, что в меню **Кодировки** установлен флажок у пункта **Кодировать в UTF-8 (без BOM)**. Если это не так, то в меню **Кодировки** следует выбрать пункт **Преобразовать в UTF-8 без BOM**.

1.3. Установка и настройка редактора Eclipse

При описании большинства языков программирования (таких как PHP, Python и др.) я обычно рекомендовал читателям использовать редактор Notepad++. Он очень удобен и хорошо подсвечивает код, написанный практически на всех языках, включая Java. Тем не менее, для создания программ на языке Java возможностей этого редактора недостаточно. Посмотрите на код листинга 1.1. Такое большое количество кода выводит в окно консоли всего лишь одну надпись. Теперь представьте, сколько придется написать текста, чтобы сделать что-то более сложное. Инструкции и названия методов в языке Java весьма длинные, и набирать их вручную очень долго. Если вы не владеете методом слепого десятипальцевого набора текста,

а еще хуже — не владеете английским языком, то программирование на языке Java при использовании редактора Notepad++ может стать настоящим мучением.

К счастью, существуют специализированные редакторы, которые не только подсвечивают код программы, но и позволяют вывести список всех методов и свойств объекта, автоматически закончить слово, подчеркнуть код с ошибкой, а также скомпилировать проект всего лишь нажатием одной кнопки без необходимости использования командной строки. В этой книге мы для создания программ воспользуемся редактором Eclipse.

Для загрузки редактора Eclipse переходим на страницу:

http://www.eclipse.org/downloads/eclipse-packages/

и скачиваем архив с программой из раздела **Eclipse IDE for Java Developers**. Распаковываем скачанный архив в папку C:\JavaSE. Редактор не нуждается в установке, поэтому просто переходим в папку C:\JavaSE\eclipse и запускаем файл eclipse.exe.

При запуске редактор попросит указать папку с рабочим пространством (рис. 1.5). Указываем **C:\JavaSE\projects** и нажимаем кнопку **Launch**. Для создания проекта в меню **File** редактора выбираем пункт **New | Java Project**. В открывшемся окне (рис. 1.6) в поле **Project name** вводим `HelloWorld`, выбираем версию JRE и нажимаем кнопку **Finish**. Теперь добавим в проект файл с классом. Для этого в меню **File** выбираем пункт **New | Class**. В открывшемся окне (рис. 1.7) в поле **Name** вводим `HelloWorld` и нажимаем кнопку **Finish**. Редактор создаст файл HelloWorld.java и откроет его для редактирования. Причем внутри файла уже будет вставлен код. Вводим сюда код из листинга 1.1 и сохраняем проект. Для этого в меню **File** выбираем пункт **Save**.

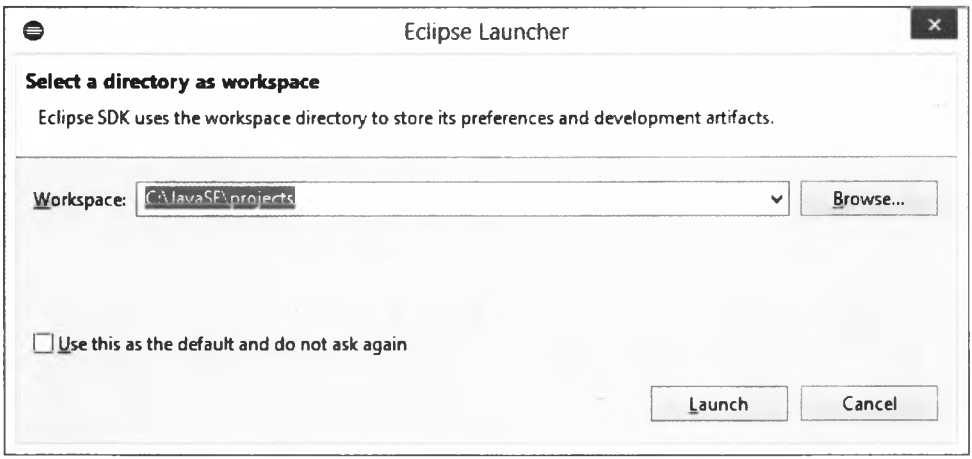

Рис. 1.5. Окно **Eclipse Launcher**

Давайте теперь откроем папку C:\JavaSE\projects в Проводнике и посмотрим ее содержимое. В результате наших действий редактор создал папку HelloWorld и две папки внутри нее: src и bin. Внутри папки C:\JavaSE\projects\HelloWorld\src мы можем найти файл HelloWorld.java, который содержит код из листинга 1.1. Помимо этого

Рис. 1.6. Создание проекта

Рис. 1.7. Создание класса

редактор создал вспомогательные папки и файлы, названия которых начинаются с точки. В этих папках и файлах редактор сохраняет различные настройки. Не изменяйте и не удаляйте эти файлы.

Теперь попробуем скомпилировать программу и запустить ее на выполнение. Для этого командная строка нам больше не понадобится. Переходим в редактор и в меню **Run** выбираем пункт **Run** или нажимаем комбинацию клавиш <Ctrl>+<F11>. В окне **Run As** (если оно отобразилось) выбираем пункт **Java Application** и нажимаем кнопку **OK**.

В итоге в папке C:\JavaSE\projects\HelloWorld\bin будет создан файл HelloWorld.class с байт-кодом, а результат выполнения программы отобразится в окне **Console** редактора Eclipse (рис. 1.8). Все очень просто, быстро и удобно.

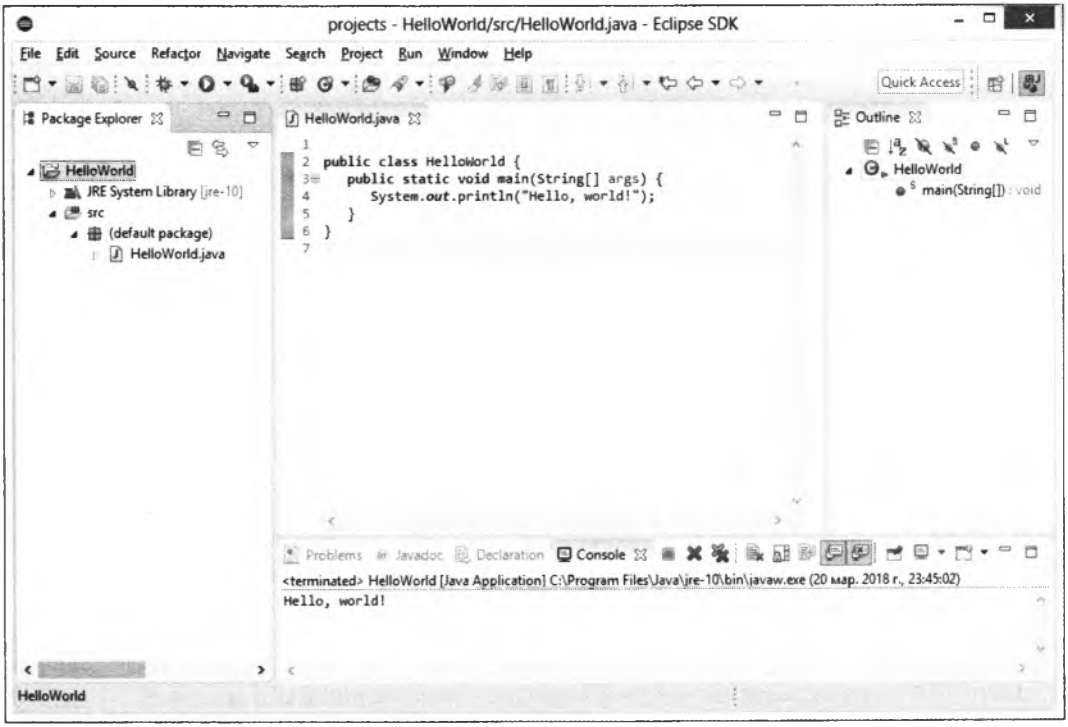

Рис. 1.8. Главное окно редактора Eclipse

Редактор Eclipse использует встроенный компилятор, а не внешний компилятор из папки с установленным JDK. Причем доступны компиляторы практически всех версий. Для выбора версии компилятора в меню **Window** выбираем пункт **Preferences**. В открывшемся окне переходим на вкладку **Java | Compiler** (рис. 1.9) и выбираем нужную версию из списка **Compiler compliance level**. Мы будем изучать Java 10, поэтому в этом списке должен быть выбран пункт **10**.

Как можно видеть, редактор не только подсвечивает код программы и выделяет фрагменты с ошибками, но и позволяет получить справку при наведении указателя мыши на какое-либо название. Например, наведите указатель на метод `println()`,

Рис. 1.9. Окно Preferences: вкладка Compiler

и вы получите краткую справку по этому методу. Но сейчас это будет работать только при активном подключении к Интернету. Если хотите, чтобы справка отображалась без подключения к Интернету, то нужно выполнить следующие действия:

1. Переходим на страницу:

   http://www.oracle.com/technetwork/java/javase/downloads/index.html

   и скачиваем архив с документацией. В нашем случае архив называется jdk-10_doc-all.zip.

2. Распаковываем архив и копируем папку docs в каталог C:\Program Files\Java\jdk-10.

3. В меню **Window** редактора Eclipse выбираем пункт **Preferences**. В открывшемся окне переходим на вкладку **Java | Installed JREs** (рис. 1.10). Выделяем строку с JRE и нажимаем кнопку **Edit**. В списке библиотек (рис. 1.11) находим строку **C:\Program Files\Java\jre-10\lib\jrt-fs.jar**, выделяем ее и нажимаем кнопку **Javadoc Location**. В поле **Javadoc location path** (рис. 1.12) вводим значение `file:/C:/Program%20Files/Java/jdk-10/docs/api/`. Если редактор не признает введенное значение, то находим папку docs/api/, нажав кнопку **Browse**. Сохраняем все изменения.

Рис. 1.10. Окно **Preferences**: вкладка **Installed JREs**

Примечание

В Java 10 изменилась структура файлов документации. Поэтому, если возникла проблема с подключением новой документации, просто скачайте документацию от Java 9 и подключите ее.

Если после названия объекта вставить точку, то редактор отобразит список методов и свойств. Например, попробуйте поставить точку после объекта `System.out`, и вы получите список методов этого объекта (рис. 1.13). Если при этом набирать первые буквы, то содержимое списка сократится. Достаточно выбрать метод из списка, и его название будет вставлено в код. Каким бы длинным ни было название метода, мы можем его вставить в код без ошибок в его названии, и очень быстро.

Редактор позволяет также закончить частично набранное слово. Для этого достаточно нажать комбинацию клавиш <Ctrl>+<Пробел>. В результате редактор автоматически закончит слово или предложит список с возможными вариантами. Причем этот способ отобразит еще и названия *шаблонов кода*. При выборе шаблона будет вставлено не просто слово, а целый фрагмент кода. Например, введите слово `sysout` и нажмите комбинацию клавиш <Ctrl>+<Пробел>. В результате будет вставлен следующий код:

```
System.out.println();
```

Рис. 1.11. Окно **Edit JRE**

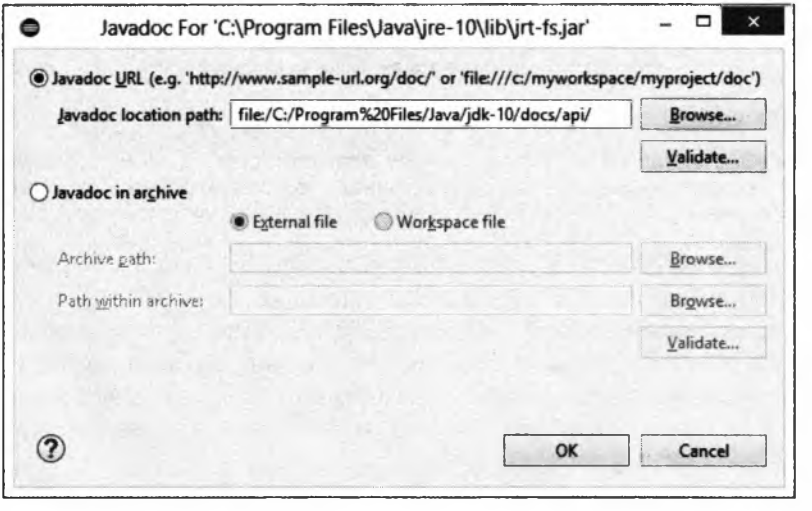

Рис. 1.12. Указание пути к документации

Это очень удобно и экономит много времени. Чтобы увидеть все доступные шаблоны, в меню **Window** выбираем пункт **Preferences**. В открывшемся окне переходим на вкладку **Java | Editor | Templates** (рис. 1.14). С помощью этой вкладки можно не только посмотреть все шаблоны, но и отредактировать их, а также создать свой собственный шаблон.

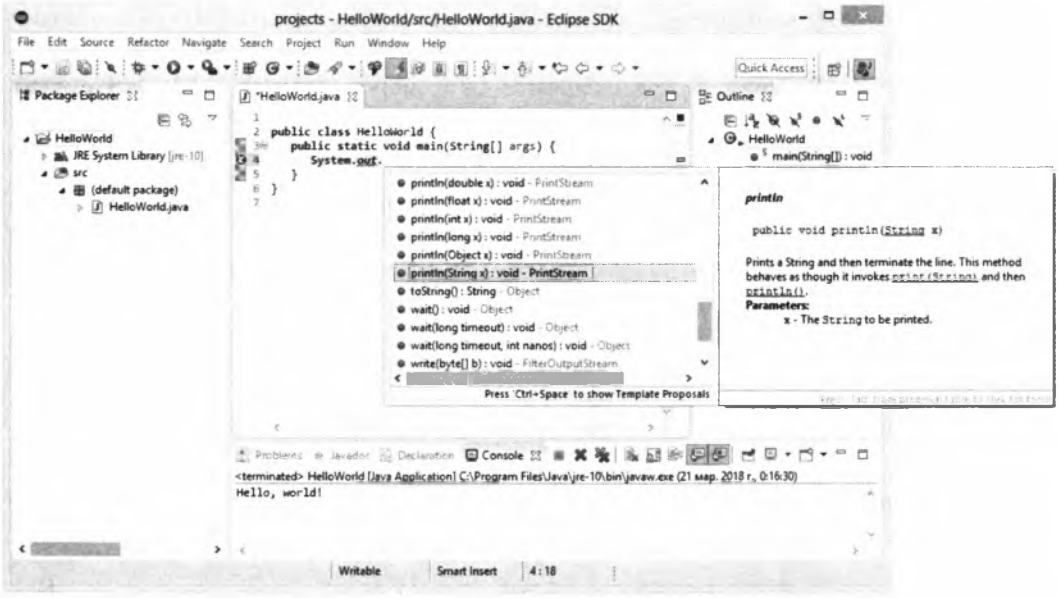

Рис. 1.13. Отображение списка методов и документации к методу

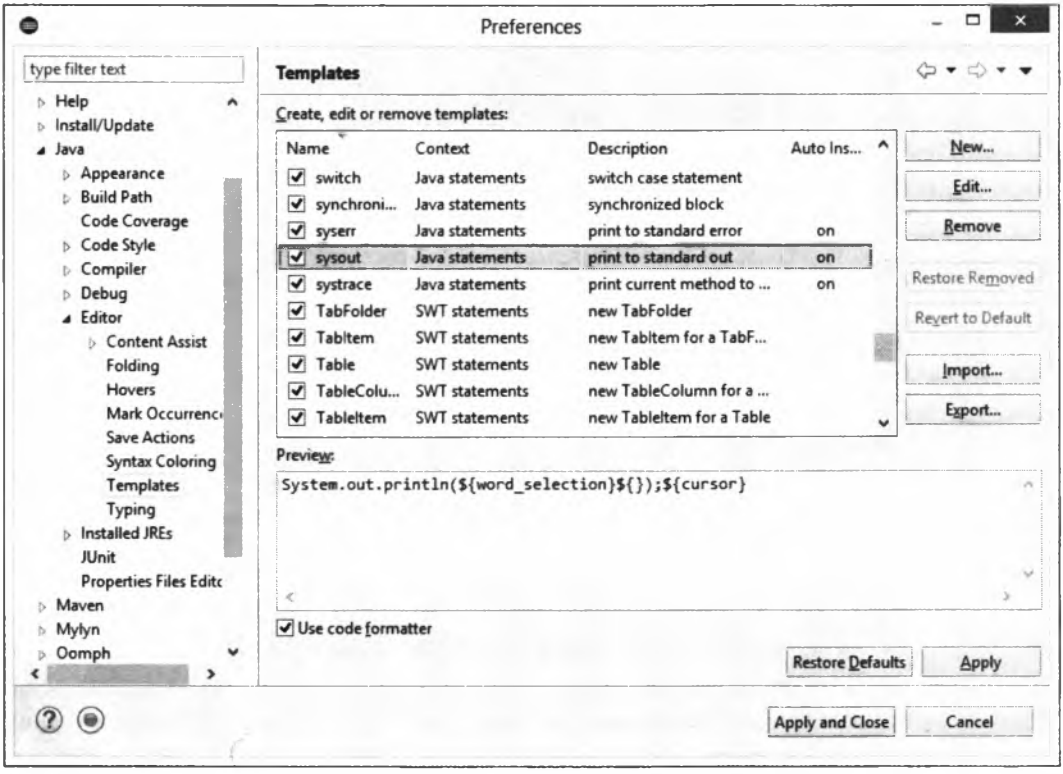

Рис. 1.14. Окно **Preferences**: вкладка **Templates**

Как уже было отмечено ранее, работать мы будем с кодировкой UTF-8, поэтому давайте настроим кодировку файлов по умолчанию. Для этого в меню **Window** выбираем пункт **Preferences**. В открывшемся окне переходим на вкладку **General | Workspace** (рис. 1.15). В группе **Text file encoding** устанавливаем флажок **Other** и из списка выбираем кодировку **UTF-8**. Сохраняем изменения. Если необходимо изменить кодировку уже открытого файла, в меню **Edit** выбираем пункт **Set Encoding**.

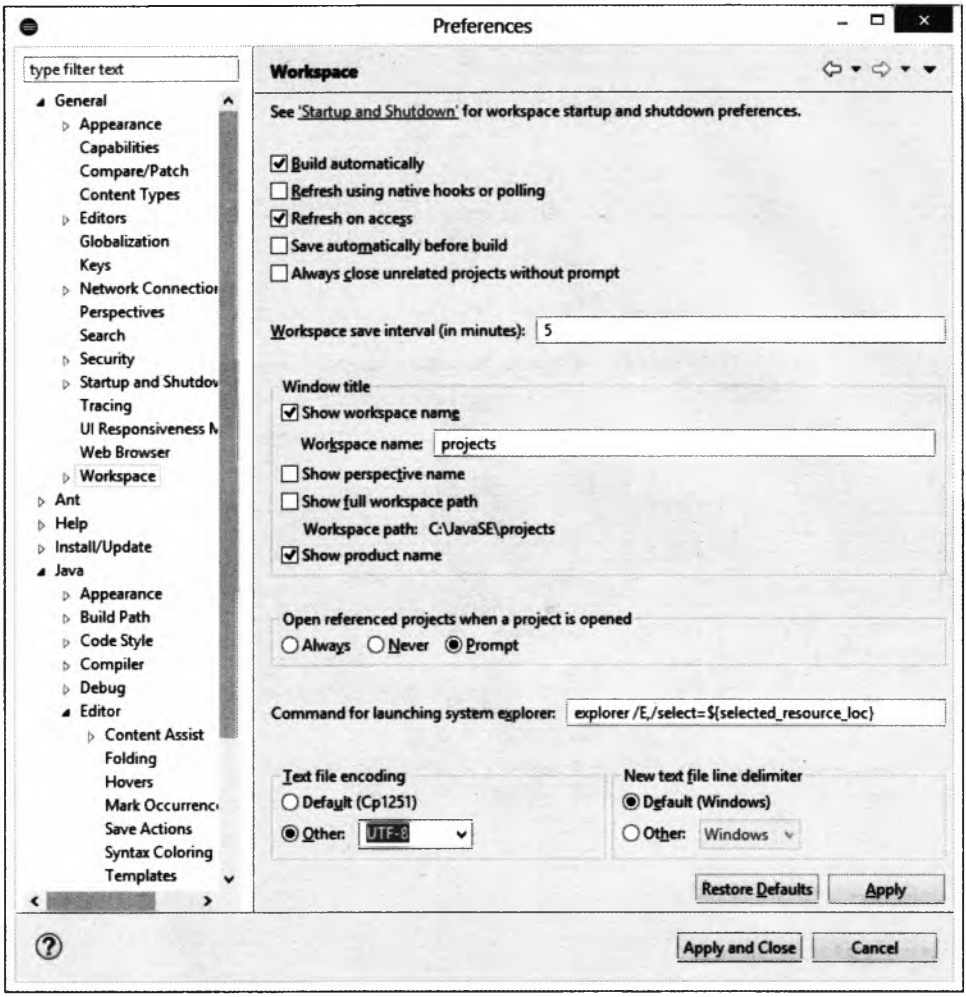

Рис. 1.15. Указание кодировки файлов

По умолчанию редактор вместо пробелов вставляет символы табуляции. Нас это не устраивает. Давайте изменим настройку форматирования кода. Для этого в меню **Window** выбираем пункт **Preferences**. В открывшемся окне переходим на вкладку **Java | Code Style | Formatter** (рис. 1.16). Нажимаем кнопку **New**. В открывшемся окне (рис. 1.17) в поле **Profile name** вводим название стиля, например: `MyStyle`, а из списка выбираем пункт **Eclipse [built-in]**. Нажимаем кнопку **OK**. Откроется окно,

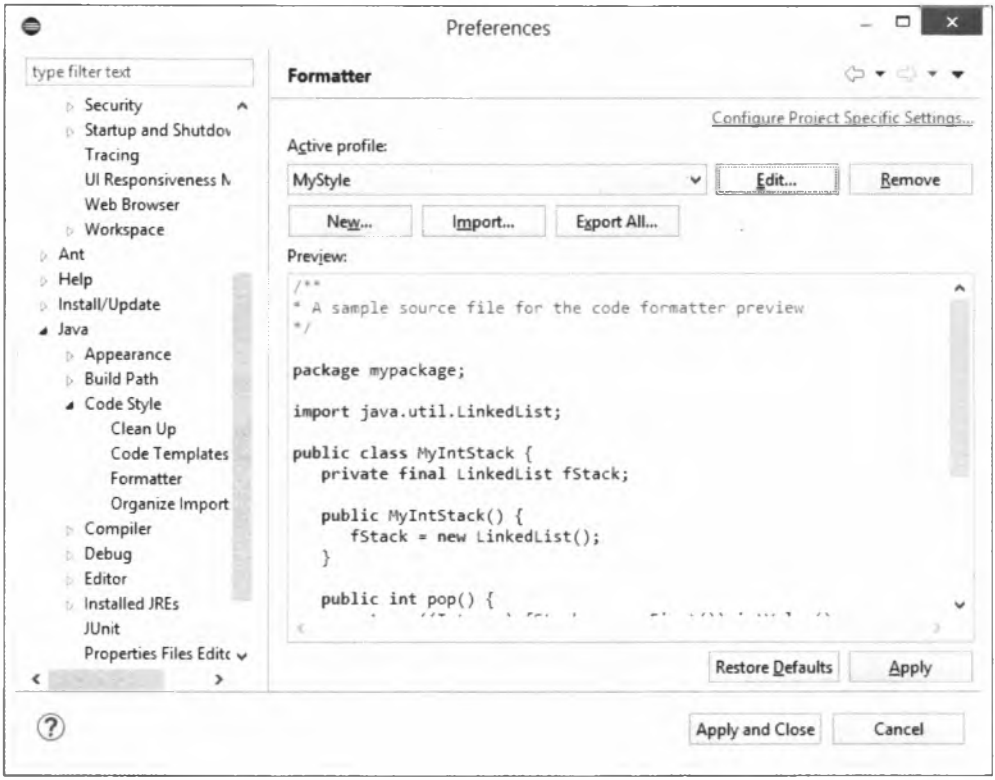

Рис. 1.16. Окно **Preferences**: вкладка **Formatter**

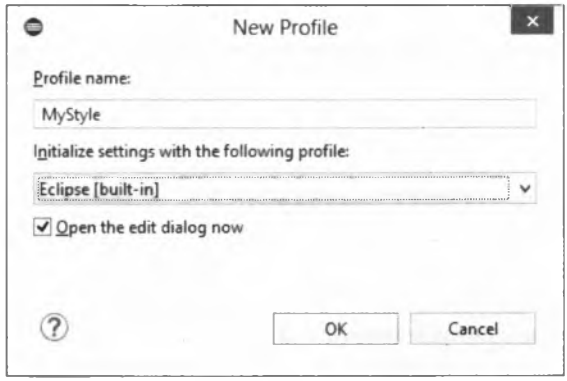

Рис. 1.17. Окно **New Profile**

в котором можно изменить настройки нашего стиля. На вкладке **Indentation** из списка **Tab policy** выбираем пункт **Spaces only**, а в поля **Indentation size** и **Tab size** вводим число 3. Сохраняем все изменения.

Если необходимо изменить размер шрифта, то в меню **Window** выбираем пункт **Preferences**. В открывшемся окне переходим на вкладку **General | Appearance | Colors and Fonts** (рис. 1.18). Из списка выбираем пункт **Java | Java Editor Text Font** и нажимаем кнопку **Edit**.

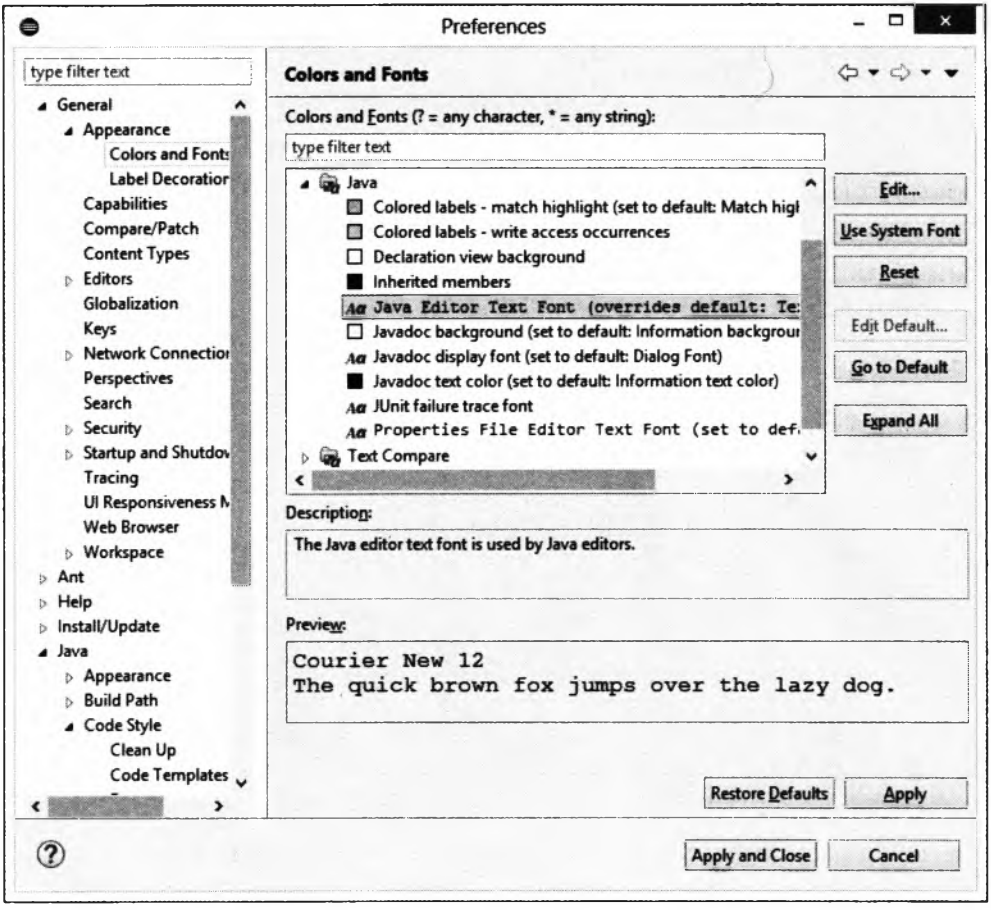

Рис. 1.18. Окно **Preferences**: вкладка **Colors and Fonts**

1.4. Структура программы

Что ж, программная среда установлена, и редактор настроен. Теперь можно приступать к изучению языка. Начнем с рассмотрения кода из листинга 1.1, который выводит приветствие в окно консоли. Весь код состоит из трех инструкций:

- описание класса `HelloWorld`;
- описание метода `main()`;
- вывод приветствия с помощью метода `println()`.

Поскольку язык Java является объектно-ориентированным языком программирования, текст программы всегда находится внутри описания какого-либо класса. Причем название файла, содержащего описание класса, в точности совпадает с названием класса. Вплоть до регистра символов! В нашем случае класс `HelloWorld` находится внутри файла с названием HelloWorld.java.

Описание класса является составной инструкцией:

```
public class HelloWorld {
}
```

Модификатор доступа `public` означает, что класс `HelloWorld` является общедоступным, и любой другой объект может обратиться к этому классу, создать экземпляр этого класса, получить доступ к общедоступным методам и полям. Ключевое слово `class` означает, что описывается класс с названием, указанным после слова `class`. Далее между фигурными скобками вставляется описание методов и полей класса. В нашем случае описание метода `main()`.

Фигурные скобки заключают блок, который ограничивает область видимости идентификаторов. Все идентификаторы, описанные внутри блока, видны только внутри этого блока. Извне блока обратиться к идентификаторам можно только через точку после имени класса. В нашем случае к методу `main()` можно обратиться так:

```
HelloWorld.main(<Значение>)
```

Причем не ко всем идентификаторам можно получить доступ таким образом. Давайте взглянем на описание метода `main()`:

```
public static void main(String[] args) {
}
```

Модификатор доступа `public` означает, что метод `main()` является общедоступным. Если метод был бы описан с помощью ключевого слова `private`, то обратиться к методу с помощью оператора «точка» было бы нельзя.

Ключевое слово `static` означает, что метод является *статическим*. В этом случае мы можем обратиться к методу через оператор «точка» без необходимости создания экземпляра класса. Что же такое класс, а что *экземпляр класса*?

В языке Java все — либо класс, либо объект (экземпляр класса). Предположим, у нас есть класс `Телевизор`. Внутри этого класса описан чертеж, электрическая схема и т. д. То есть приведено всего лишь описание телевизора и принципа его работы. Далее на основании этого класса мы производим несколько экземпляров телевизоров (создаем несколько объектов класса `Телевизор`). Все они построены на основе одного класса, но могут иметь разные свойства. Один черного цвета, второй серого, третий белого. Один включен, второй в режиме ожидания, третий вообще выключен. Таким образом, говоря простым языком, класс — это схема, а экземпляр класса — объект, созданный по этой схеме. Класс один, а экземпляров может быть множество, и каждый экземпляр может обладать своими собственными свойствами, не зависящими от свойств других экземпляров.

Класс может быть не только схемой, но и пространством имен, внутри которого описываются какие-либо методы. В этом случае методы помечаются с помощью ключевого слова `static`. Доступ к статическому методу осуществляется без создания экземпляра класса с помощью оператора «точка» по следующей схеме:

```
<Название класса>.<Метод>([<Параметры>])
```

В описании метода `main()` после ключевого слова `static` указан тип возвращаемого методом значения. Ключевое слово `void` означает, что метод вообще ничего не возвращает. Что такое метод? *Метод* — это фрагмент кода внутри блока класса, который может быть вызван из какого-либо места программы сколько угодно раз. При этом код может возвращать какое-либо значение в место вызова или вообще ничего не возвращать, а просто выполнять какую-либо операцию. В нашем случае метод `main()` выводит приветствие в окно консоли и ничего не возвращает. Если вы программировали на других языках, то знакомы с понятием *функция*. Можно сказать, что метод — это функция, объявленная внутри класса. В языке Java нет процедурного стиля программирования, поэтому просто функций самих по себе не существует. Только методы внутри класса.

Название метода `main()` является предопределенным. Именно этот метод выполняется, когда мы запускаем программу из командной строки. Регистр в названии метода имеет значение. Если программа не содержит метода `main()`, то при запуске из командной строки мы получим следующее сообщение об ошибке:

```
C:\book>java HelloWorld
Error: Main method not found in class HelloWorld, please define
the main method
as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

Метод может принимать какие-либо параметры, указываемые внутри круглых скобок через запятую. Язык Java является строго типизированным языком, поэтому в описании метода указывается тип значения принимаемого параметра. В нашем случае внутри круглых скобок указано следующее выражение:

```
String[] args
```

Ключевое слово `String` означает, что метод принимает объект класса `String` (проще говоря, строку из символов). Квадратные скобки являются признаком массива (проще говоря, строк может быть много). Идентификатор `args` — всего лишь имя переменной, через которое мы можем получить доступ к массиву строк внутри метода. При запуске программы в метод передаются параметры, указанные после названия программы в командной строке. Например, передадим два параметра:

```
C:\book>java HelloWorld string1 string2
```

Описание метода также является блочной инструкцией, поэтому после описания параметров указывается открывающая фигурная скобка. Закрывающая фигурная скобка является признаком конца блока. Описание метода `main()` вложено в блок класса, поэтому метод принадлежит классу. Чтобы наглядно видеть вложенность блоков, в коде перед описанием метода `main()` указывается одинаковое количество пробелов — обычно три или четыре. Вместо пробелов можно использовать символ табуляции. Выберите для себя какой-либо один способ и используйте его постоянно. В этой книге мы будем использовать три пробела. Компилятору не важно, сколько пробелов вы поставите. Можно вообще написать так:

```
public class HelloWorld {
public static void main(String[] args) {
System.out.println("Hello, world!");
}
}
```

Такой код становится трудно читать, прежде всего, самому программисту. Ведь непонятно, где начало блока, а где конец. Поэтому возьмите за правило выделять блоки в программе одинаковым количеством пробелов или символов табуляции. Для вложенных блоков количество пробелов умножают на уровень вложенности: если для блока первого уровня вложенности использовались три пробела, то для блока второго уровня вложенности следует вставить шесть пробелов, для третьего уровня — девять и т. д. В одной программе в качестве отступа не следует использовать и пробелы, и табуляцию, — необходимо выбрать что-то одно и пользоваться этим во всей программе.

Фигурные скобки также можно расставлять по-разному. Опять-таки, компилятору это не важно. Обычно применяются два стиля. Первый стиль мы использовали в листинге 1.1: открывающая фигурная скобка на одной строке с описанием класса или метода, а закрывающая — на отдельной строке. Второй стиль выглядит следующим образом:

```
public class HelloWorld
{
   public static void main(String[] args)
   {
      System.out.println("Hello, world!");
   }
}
```

Во втором стиле обе фигурные скобки расположены на отдельных строках. Многие программисты считают такой стиль наиболее приемлемым, т. к. открывающая и закрывающая скобки расположены друг под другом. На мой же взгляд здесь образуются лишние пустые строки. А так как размеры экрана ограничены, при наличии пустых строк на экране помещается меньше кода, и приходится чаще пользоваться полосой прокрутки. Если же размещать инструкции с равным отступом, то блок кода выделяется визуально и следить за положением фигурных скобок просто излишне. Тем более, что редактор позволяет подсветить парные скобки. Какой стиль использовать, зависит от личного предпочтения программиста или от правил оформления кода, принятых в той или иной фирме. Главное, чтобы стиль оформления внутри одной программы был одинаковым. В этой книге мы будем использовать первый стиль.

Внутри метода `main()` с помощью инструкции:

```
System.out.println("Hello, world!");
```

выводится приветствие **Hello, world!** в окно консоли. Идентификатор `System` является системным классом, входящим в состав библиотеки языка Java. Внутри этого класса находится свойство `out`, содержащее ссылку на объект класса `PrintStream`,

который связан с окном консоли. Так как `out` находится внутри класса `System`, то получить к нему доступ можно через оператор «точка». Теперь у нас есть объект класса `PrintStream`, который имеет несколько методов, позволяющих вывести данные. Чтобы получить доступ к этим методам, опять применяется оператор «точка». В нашем случае используется метод `println()` из этого класса. Внутри круглых скобок мы передаем строку, которая и будет выведена в окне консоли.

После закрывающей круглой скобки ставится точка с запятой, которая говорит компилятору о конце инструкции. Это все равно, что поставить точку в конце предложения. Если точку с запятой не указать, то компилятор выведет сообщение об ошибке. Обратите внимание на то, что в конце составных инструкций после закрывающей фигурной скобки точка с запятой не ставится. Именно скобки указывают компилятору начало и конец блока. На самом деле, если поставить точку с запятой после закрывающей фигурной скобки, ошибки не будет. Просто компилятор посчитает пустое пространство между скобкой и точкой с запятой как пустую инструкцию.

Начинающему программисту сложно понять все, что мы рассмотрели в этом разделе. Слишком много информации... Тут и классы, и методы, и инструкции. Да, *объектно-ориентированный стиль программирования* (сокращенно *ООП-стиль*) даже опытные программисты не могут сразу понять. Но ничего не поделаешь, в языке Java возможен только ООП-стиль. Не беспокойтесь, в дальнейшем классы и методы мы рассмотрим подробно и последовательно. Чтобы сократить количество кода, в дальнейших примерах код создания класса и метода `main()` мы будем опускать. В результате шаблон для тестирования примеров должен выглядеть так:

```
public class MyClass {
   public static void main(String[] args) {
<Здесь набираем код из примеров>
   }
}
```

1.5. Комментарии в программе

Комментарии предназначены для вставки пояснений в текст программы, и компилятор полностью их игнорирует. Внутри комментария может располагаться любой текст, включая инструкции, которые выполнять не следует. Помните, комментарии нужны программисту, а не компилятору. Вставка комментариев в код позволит через некоторое время быстро вспомнить предназначение фрагмента кода.

В языке Java присутствуют два типа комментариев: *однострочный* и *многострочный*. Однострочный комментарий начинается с символов `//` и заканчивается в конце строки. Вставлять однострочный комментарий можно как в начале строки, так и после инструкции:

```
// Это комментарий
System.out.println("Hello, world!"); // Это комментарий
```

Если символ комментария разместить перед инструкцией, то она не будет выполнена:

```
// System.out.println("Hello, world!"); // Инструкция выполнена не будет
```

Если символы // расположены внутри кавычек, то они не являются признаком начала комментария:

```
System.out.println("// Это НЕ комментарий!!!");
```

Многострочный комментарий начинается с символов /* и заканчивается символами */. Комментарий может быть расположен как на одной строке, так и на нескольких. Кроме того, многострочный комментарий можно размещать внутри выражения, хотя это и нежелательно. Следует иметь в виду, что многострочные комментарии не могут быть вложенными, поэтому при комментировании больших блоков следует проверять, что в них не встречается закрывающая комментарий комбинация символов */. Вот примеры многострочных комментариев:

```
/*
Многострочный комментарий
*/

System.out.println("Hello, world!"); /* Это комментарий */

/* System.out.println("Hello, world!"); // Инструкция выполнена не будет
*/

int x;
x = 10 /* Комментарий */ + 50 /* внутри выражения */;

System.out.println("/* Это НЕ комментарий!!! */");
```

Редактор Eclipse позволяет быстро добавить символы комментариев. Чтобы вставить однострочный комментарий в начале строки, нужно сделать текущей строку с инструкцией и нажать комбинацию клавиш <Ctrl>+</>. Если предварительно выделить сразу несколько строк, то перед всеми выделенными строками будет вставлен однострочный комментарий. Если все выделенные строки были закомментированы ранее, то комбинация клавиш <Ctrl>+</> удалит все однострочные комментарии. Для вставки многострочного комментария необходимо выделить строки и нажать комбинацию клавиш <Shift>+<Ctrl>+</>. Для удаления многострочного комментария предназначена комбинация клавиш <Shift>+<Ctrl>+<\>.

У начинающих программистов может возникнуть вопрос: зачем комментировать инструкции? Проблема заключается в том, что часто в логике работы программы возникают проблемы. Именно по вине программиста. Например, программа выдает результат, который является неверным. Чтобы найти ошибку в алгоритме работы программы, приходится отключать часть кода с помощью комментариев, вставлять инструкции вывода промежуточных результатов и анализировать их. Как говорится: разделяй и властвуй. Таким «дедовским» способом мы обычно ищем ошибки в коде. А «дедовским» мы назвали способ потому, что сейчас все редакторы пре-

доставляют методы отладки, которые позволяют выполнять код построчно и сразу видеть промежуточные результаты. Раньше такого не было. Хотя способ и устарел, но все равно им часто пользуются.

Язык Java поддерживает также *комментарии документирования*. Комментарий документирования является многострочным и начинается с символов `/**` и заканчивается символами `*/`. Чтобы автоматически сгенерировать такой комментарий в редакторе Eclipse, необходимо предварительно установить курсор, например, на название класса, и в меню **Source** выбрать пункт **Generate Element Comment** или нажать комбинацию клавиш <Shift>+<Alt>+<J>. Попробуйте создать комментарии документирования для класса и для метода `main()`. В результате код будет выглядеть примерно следующим образом:

```
/**
 * Описание класса
 *
 * @author Unicross
 *
 */
public class MyClass {
   /**
    * @param args
    */
   public static void main(String[] args) {
      System.out.println("Hello, world!");
   }
}
```

Внутри комментария документирования может присутствовать как обычный текст (допускается использование HTML-тегов для форматирования), так и специальные дескрипторы, начинающиеся с символа `@`. В нашем примере присутствуют два дескриптора: `@author` и `@param`. Перечислим основные дескрипторы и их предназначение:

- `@author` — задает имя автора;
- `@version` — номер версии;
- `@since` — начальная версия, с которой стал доступен код;
- `@param` — описание параметра метода;
- `@return` — описание возвращаемого методом значения;
- `@throws` и `@exception` — описывают генерируемое внутри метода исключение;
- `{@inheritDoc}` — вставляет комментарий из базового класса;
- `{@code}` — позволяет добавить код примера:

```
* <pre> {@code
* List<Integer> list = new ArrayList<Integer>();
* }</pre>
```

- `{@link}` — создает ссылку на дополнительную информацию (ссылка размещается внутри текста):

```
* {@link MyClass#MyClass() Конструктор класса}
```

- `@see` — создает ссылку на дополнительную информацию (ссылка размещается в блоке `See Also`):

```
@see MyClass#MyClass() Конструктор класса
```

Существуют и другие дескрипторы. Лучший способ изучать комментарии документирования — смотреть исходный код классов из стандартной библиотеки. Такой исходный код можно найти в архиве C:\Program Files\Java\jdk-10\lib\src.zip.

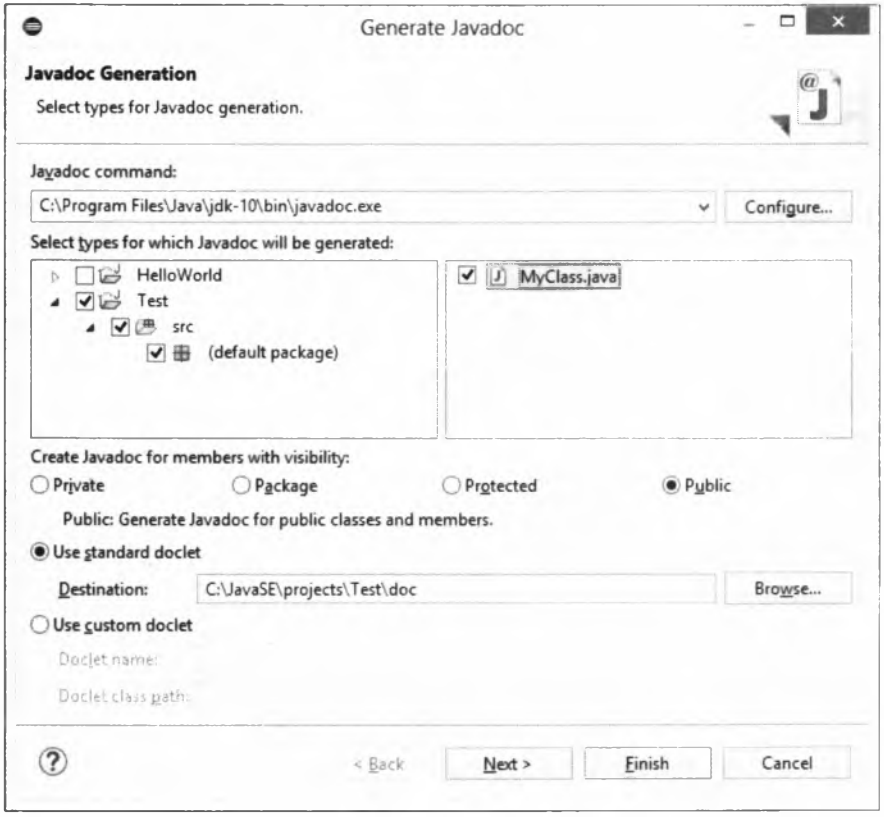

Рис. 1.19. Окно **Generate Javadoc**

Теперь попробуем сгенерировать документацию в формате HTML на основе комментариев документирования. Для этого в редакторе Eclipse в меню **Project** выбираем пункт **Generate Javadoc**. В открывшемся окне (рис. 1.19) нажимаем кнопку **Configure** и находим программу **javadoc.exe (C:\Program Files\Java\jdk-10\bin\javadoc.exe)**. В результате путь к программе должен отображаться в поле **Javadoc command**. Выбираем наш класс и нажимаем кнопку **Next**. И еще раз нажимаем кнопку **Next**. Чтобы русские буквы нормально отображались в документации,

необходимо дополнительно указать кодировку файла с программой, файла с документацией и кодировку для тега `<meta>`:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

Для этого в поле **Extra Javadoc options** (рис. 1.20) вводим следующую команду:

```
-encoding utf-8 -docencoding utf-8 -charset utf-8 -html5
```

Опция `-html5` указывает, что документация должна быть создана в формате HTML 5. Чтобы получить полный список доступных опций в командной строке, наберите команду:

```
javadoc -help
```

Нажимаем кнопку **Finish** для запуска генерации документации. Если все сделано правильно, то в папке C:\JavaSE\projects\Test\doc будет создано множество различных файлов. Для просмотра документации открываем файл index.html с помощью любого Web-браузера. Результат показан на рис. 1.21.

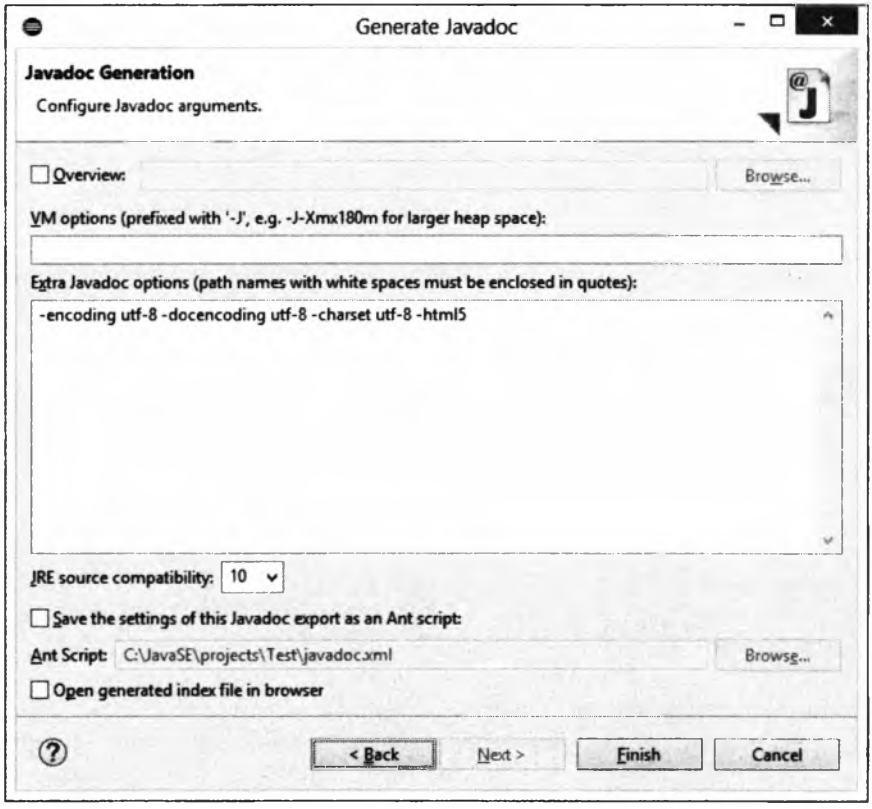

Рис. 1.20. Указание кодировки файлов документации

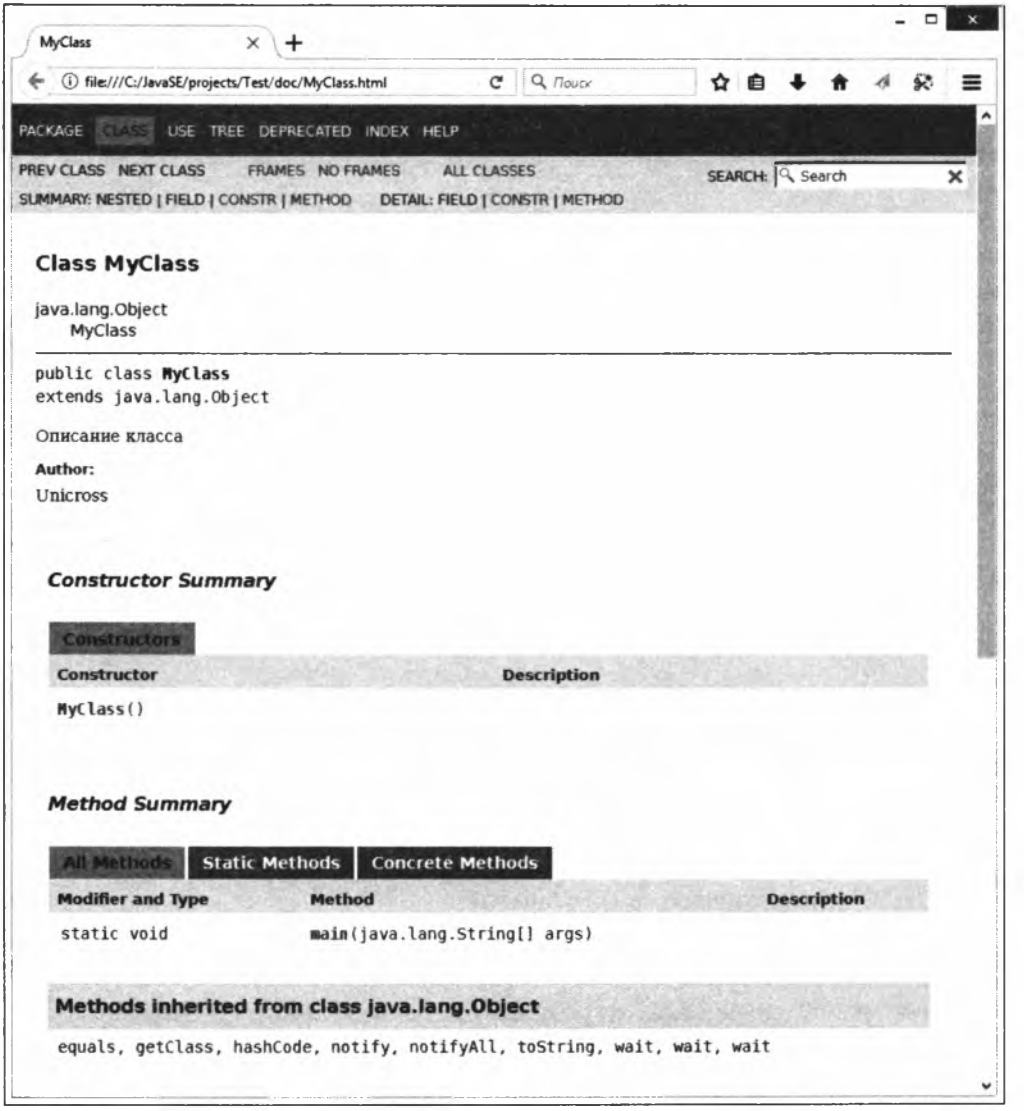

Рис. 1.21. Документация в окне Web-браузера

1.6. Вывод данных

Для вывода данных в языке Java предназначены объекты `System.out` и `System.err`. Объект `System.out` используется для вывода обычных сообщений в окно консоли, а объект `System.err` — для вывода сообщений об ошибках. Оба объекта первоначально связаны с окном консоли, однако возможно перенаправить поток на другое устройство, например, в файл.

Объекты `System.out` и `System.err` возвращают экземпляр класса `PrintStream`, через который доступны следующие основные методы:

- `print()` — отправляет данные в стандартный поток вывода. Формат метода:

```
public void print(<Данные>)
```

В параметре `<Данные>` можно указать данные типов `boolean`, `int`, `long`, `float`, `double`, `char`, `char[]`, `String` и `Object`:

```
System.out.print(10);
System.out.print("Hello, world!");
// Результат: 10Hello, world!
System.err.print("Сообщение об ошибке");
// Результат: Сообщение об ошибке
```

- `println()` — отправляет данные в стандартный поток вывода и завершает текущую строку. Формат метода:

```
public void println([<Данные>])
```

В параметре `<Данные>` можно указать данные типов `boolean`, `int`, `long`, `float`, `double`, `char`, `char[]`, `String` и `Object`. Если параметр не указан, то метод просто завершает текущую строку:

```
System.out.println(10);
System.out.println("Hello, world!");
/*
Результат:
10
Hello, world!
*/
System.err.println("Сообщение об ошибке");
```

- `printf()` — предназначен для форматированного вывода данных. Форматы метода:

```
public PrintStream printf(String format, Object... args)
public PrintStream printf(Locale locale, String format,
                                         Object... args)
```

В параметре `format` указывается строка специального формата, внутри которой с помощью спецификаторов задаются правила форматирования. Какие спецификаторы используются, мы рассмотрим немного позже при изучении форматирования строк. В параметре `args` через запятую указываются различные значения. Параметр `locale` позволяет задать *локаль*. Настройки локали для разных стран различаются — например, в одной стране принято десятичный разделитель вещественных чисел выводить в виде точки, в другой — в виде запятой. В первом формате метода используются настройки локали по умолчанию:

```
System.out.println(10.5125484);          // 10.5125484
System.out.printf("%.2f", 10.5125484);   // 10,51
```

- `flush()` — сбрасывает данные из буфера. Формат метода:

```
public void flush()
```

- `checkError()` — сбрасывает данные из буфера и возвращает значение `true`, если произошла ошибка, или `false`, если ошибка не возникла. Формат метода:

```
public boolean checkError()
```

С помощью стандартного вывода можно создать индикатор выполнения процесса в окне консоли. Чтобы реализовать такой индикатор, нужно вспомнить, что символ перевода строки в Windows состоит из двух символов `\r` (перевод каретки) и `\n` (перевод строки). Таким образом, используя только символ перевода каретки `\r`, можно перемещаться в начало строки и перезаписывать ранее выведенную информацию. Пример индикатора выполнения процесса показан в листинге 1.2.

Листинг 1.2. Индикатор выполнения процесса

```
public class MyClass {
   public static void main(String[] args) throws InterruptedException {
      System.out.print("... 0%");
      for (int i = 5; i < 101; i += 5) {
         Thread.sleep(1000); // Имитация процесса
         System.out.print("\r... " + i + "%");
      }
      System.out.println();
   }
}
```

Открываем командную строку (в редакторе Eclipse эффекта не будет видно), компилируем программу и запускаем:

```
C:\Users\Unicross>cd C:\book

C:\book>javac -encoding utf-8 MyClass.java

C:\book>java MyClass
```

Эта программа немного сложнее, чем простое приветствие из листинга 1.1. Здесь присутствует обработка исключений (выражение `throws InterruptedException`), имитация процесса с помощью метода `sleep()` из класса `Thread` (при этом программа «засыпает» на указанное количество миллисекунд).

Кроме того, в программе использован цикл `for`, который позволяет изменить порядок обработки инструкций. Обычно программа выполняется сверху вниз и слева направо. Инструкция за инструкцией. Цикл `for` меняет эту последовательность выполнения: инструкции, расположенные внутри блока, выполняются несколько раз. Количество повторений зависит от выражений внутри круглых скобок. Этих выражений три, и разделены они точками с запятой. Первое выражение объявляет целочисленную переменную `i` и присваивает ей значение `5`. Второе выражение является условием продолжения повторений. Пока значение переменной `i` меньше значения `101`, инструкции внутри блока будут повторяться. Это условие проверяется на каж-

дой итерации цикла. Третье выражение на каждой итерации цикла прибавляет значение 5 к текущему значению переменной i.

1.7. Ввод данных

Для ввода данных в языке Java предназначен объект System.in. Работать с этим объектом напрямую не очень удобно, поэтому обычно используется вспомогательный класс Scanner, который связывают с входным потоком следующим образом:

```
Scanner in = new Scanner(System.in);
```

Класс Scanner объявлен в пакете java.util, поэтому, прежде чем использовать этот класс, необходимо подключить его, добавив в начале программы такую инструкцию:

```
import java.util.Scanner;
```

Для получения данных предназначены следующие основные методы из класса Scanner (полный список методов мы будем рассматривать при изучении потоков):

- nextInt() — используется для ввода целых чисел (тип int). Формат метода:

  ```
  public int nextInt()
  ```

  Пример:

  ```
  Scanner in = new Scanner(System.in);
  int x = 0;
  x = in.nextInt();                    // Вводим:  10
  System.out.println("Вы ввели " + x); // Выведет: Вы ввели 10
  ```

- nextLong() — предназначен для ввода целых чисел (тип long). Формат метода:

  ```
  public long nextLong()
  ```

  Пример:

  ```
  Scanner in = new Scanner(System.in);
  long x;
  x = in.nextLong();                   // Вводим:  20
  System.out.println("Вы ввели " + x); // Выведет: Вы ввели 20
  ```

- nextShort() — используется для ввода целых чисел (тип short). Формат метода:

  ```
  public short nextShort()
  ```

  Пример:

  ```
  Scanner in = new Scanner(System.in);
  short x;
  x = in.nextShort();                  // Вводим:  30
  System.out.println("Вы ввели " + x); // Выведет: Вы ввели 30
  ```

- nextFloat() — предназначен для ввода вещественных чисел (тип float). Формат метода:

  ```
  public float nextFloat()
  ```

Пример:

```
Scanner in = new Scanner(System.in);
float x;
x = in.nextFloat();                    // Вводим:  10,5
System.out.println("Вы ввели " + x); // Выведет: Вы ввели 10.5
```

- `nextDouble()` — используется для ввода вещественных чисел (тип `double`). Формат метода:

```
public double nextDouble()
```

Пример:

```
Scanner in = new Scanner(System.in);
double x;
x = in.nextDouble();                   // Вводим:  10,5
System.out.println("Вы ввели " + x); // Выведет: Вы ввели 10.5
```

- `nextLine()` — получает строку (тип `String`). Формат метода:

```
public String nextLine()
```

Пример:

```
Scanner in = new Scanner(System.in);
String s;
s = in.nextLine();                     // Вводим:  Сергей
System.out.println("Вы ввели " + s); // Выведет: Вы ввели Сергей
```

В качестве примера произведем суммирование двух целых чисел, введенных пользователем в окне консоли (листинг 1.3).

Листинг 1.3. Суммирование двух введенных чисел

```
import java.util.Scanner;

public class MyClass {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      int x = 0, y = 0;
      System.out.print("x = ");
      x = in.nextInt();
      System.out.print("y = ");
      y = in.nextInt();
      System.out.println("Сумма = " + (x + y));
   }
}
```

Язык Java, как уже указывалось, является строго типизированным. Это означает, что переменная может содержать значения того типа, который указан при объявле-

нии переменной. В нашем примере мы объявляем две целочисленные переменные x и y:

```
int x = 0, y = 0;
```

Эти переменные могут хранить только значения, имеющие тип int. То есть целые числа в определенном диапазоне значений.

Говоря простым языком: *переменная* — это коробка, в которую мы можем что-то положить и из которой потом вытащить. Поскольку таких коробок может быть много, то каждая коробка подписывается (каждая переменная имеет уникальное имя внутри программы). Коробки могут быть разного размера. Например, необходимо хранить яблоко и арбуз. Согласитесь, что размеры яблока и арбуза различаются, и чтобы поместить яблоко и арбуз, мы должны взять для них соответствующего размера коробки. Таким образом, тип данных при объявлении переменной задает, какого размера коробку подготовить и что мы туда будем класть. Кроме того, в одну коробку мы можем положить только один предмет. Если нам нужно положить несколько яблок, то мы должны взять уже ящик (который в языке программирования называется *массивом*) и складывать туда коробки с яблоками.

Что будет, если вместо целого числа мы введем в окне консоли, например, строку, не содержащую число? В этом случае метод nextInt() не сможет преобразовать строку в число, и программа аварийно завершится. Обработку таких ошибок мы будем рассматривать далее в этой книге. А пока набирайте только целые числа!

1.8. Получение данных из командной строки

Передать данные можно в командной строке после названия файла. Получить эти данные в программе позволяет параметр args в методе main(). Квадратные скобки после типа String означают, что доступен массив строк (массив объектов типа String). Чтобы получить количество аргументов, переданных в командной строке, следует воспользоваться свойством length. Рассмотрим получение данных из командной строки на примере (листинг 1.4).

Листинг 1.4. Получение данных из командной строки. Вариант 1

```
public class MyClass {
   public static void main(String[] args) {
      for (int i = 0; i < args.length; i++) {
         System.out.println(args[i]);
      }
   }
}
```

Компилируем программу и запускаем из командной строки. Для запуска программы вводим команду:

```
C:\book>java MyClass string1 string2
```

В этой команде мы передаем программе `MyClass` некоторые данные (`string1 string2`). Результат выполнения программы будет выглядеть так:

```
string1
string2
```

Для перебора элементов массива можно также использовать другой синтаксис цикла `for` (листинг 1.5).

Листинг 1.5. Получение данных из командной строки. Вариант 2

```
public class MyClass {
   public static void main(String[] args) {
      for (String s: args) {
         System.out.println(s);
      }
   }
}
```

В этом случае внутри круглых скобок объявляется переменная `s`, имеющая тип `String` (текстовая строка), а после двоеточия указывается массив `args` (это переменная, которая объявлена в методе `main()`). На каждой итерации цикла из массива строк берется одна строка и присваивается переменной `s`. И так до тех пор, пока не будут перебраны все элементы массива.

1.9. Преждевременное завершение выполнения программы

В некоторых случаях может возникнуть условие, при котором дальнейшее выполнение программы лишено смысла. Тогда лучше вывести сообщение об ошибке и прервать выполнение программы досрочно. Для этого предназначен метод `exit()` из класса `System`. Формат метода:

```
public static void exit(int status)
```

В качестве параметра метод принимает число, которое является статусом завершения. Число `0` означает нормальное завершение программы, а любое другое число — некорректное завершение. Это число передается операционной системе.

В качестве примера произведем деление двух целых чисел, введенных пользователем, и выведем результат. При этом обработаем деление на нуль (листинг 1.6).

Листинг 1.6. Преждевременное завершение выполнения программы

```
import java.util.Scanner;

public class MyClass {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
```

```
        int x = 0, y = 0;
        System.out.print("x = ");
        x = in.nextInt();
        System.out.print("y = ");
        y = in.nextInt();
        if (y == 0) {
           System.out.println("Нельзя делить на 0");
           System.exit(1); // Завершаем выполнение программы
        }
        System.out.println("Результат деления = " + (x / y));
    }
}
```

В этом примере вместо метода `exit()` можно было воспользоваться инструкцией `return`, т. к. завершение программы выполнялось внутри метода `main()`. Однако в больших программах основная часть кода расположена вне метода `main()`, и в этом случае инструкцией `return` для завершения всей программы уже не обойтись. Попробуйте заменить инструкцию:

```
System.exit(1); // Завершаем выполнение программы
```

следующей:

```
return; // Завершаем выполнение программы
```

Код из листинга 1.6 содержит новую для нас инструкцию — условный оператор `if`. С помощью этого оператора мы можем проверить значение и, если оно соответствует условию, выполнить инструкции внутри блока (внутри фигурных скобок). Условие указывается внутри круглых скобок. Здесь мы проверяем значение переменной `y` равенству значению `0` с помощью оператора сравнения `==` (равно). Операторы сравнения возвращают либо значение `true` (истина), либо значение `false` (ложь). Если сравнение вернуло значение `true`, то будут выполнены инструкции, расположенные внутри фигурных скобок, а если `false` — то инструкции пропускаются.

1.10. Интерактивная оболочка JShell

Начиная с Java 9, в состав JDK входит консольная программа jshell.exe (расположена в папке jdk-10\bin), которая позволяет увидеть результаты выполнения команд сразу после ввода. Интерактивная оболочка JShell дает возможности объявлять переменные, создавать методы, описывать классы, импортировать классы и многое другое. Благодаря этой программе, начинающие программисты могут изучать конструкции языка без необходимости размещать их внутри метода какого-либо класса, а опытные программисты — демонстрировать работу различных инструкций другим программистам.

Давайте рассмотрим основные возможности JShell. Для этого запустим командную строку и введем команду `jshell`:

```
C:\Users\Unicross>jshell
|  Welcome to JShell -- Version 10
|  For an introduction type: /help intro

jshell>
```

В результате выведено приветствие и приглашение для ввода команды.

Например, выведем какую-либо строку (после ввода команды нажимаем клавишу <Enter>. Точку с запятой после команды можно не указывать):

```
jshell> System.out.println("Hello");
Hello

jshell>
```

На следующей строке сразу видим результат выполнения команды, и программа выводит приглашение для ввода следующей команды.

Если при выводе русских букв возникают проблемы, то перед запуском JShell нужно сменить кодировку windows-866 на windows-1251 с помощью команды `chcp 1251`:

```
C:\Users\Unicross>chcp 1251
Текущая кодовая страница: 1251
```

Учитывая возможность получить результат сразу после ввода команды, программу JShell можно использовать в качестве многофункционального калькулятора:

```
jshell> 12 * 32 + 54
$2 ==> 438
```

Результат выполнения выражения автоматически сохраняется в переменной с названием `$<Номер>` (в нашем примере — в переменной `$2`). Это позволяет производить дальнейшие расчеты без ввода предыдущего результата — достаточно указать название переменной:

```
jshell> $2 + 10
$3 ==> 448
```

При необходимости мы можем объявить переменную явным образом и использовать ее в расчетах:

```
jshell> int x = $2 + 10;
x ==> 448

jshell> x - 5
$5 ==> 443
```

Все команды, выполненные в текущем сеансе, сохраняются в списке команд. Посмотреть этот список позволяет команда `/list`:

```
jshell> /list

   1 : System.out.println("Hello");
   2 : 12 * 32 + 54
```

```
   3 : $2 + 10
   4 : int x = $2 + 10;
   5 : x - 5
```

Слева от команды отображается ее порядковый номер, который можно использовать, например, для повторного выполнения команды. Для этого используется следующий синтаксис: `/<Номер>`. Выполним повторно первую команду:

```
jshell> /1
System.out.println("Hello");
Hello
```

С помощью команды `/drop <Номер>` можно удалить команду из списка:

```
jshell> /drop 5
|  dropped variable $5

jshell> /list

   1 : System.out.println("Hello");
   2 : 12 * 32 + 54
   3 : $2 + 10
   4 : int x = $2 + 10;
   6 : System.out.println("Hello");
```

Команда `/save` позволяет сохранить список в файл, а команда `/open` — восстановить список из файла:

```
jshell> /save C:\book\list.txt

jshell> /reset
|  Resetting state.

jshell> /list

jshell> /open C:\book\list.txt
Hello
Hello

jshell> /list

   1 : System.out.println("Hello");
   2 : 12 * 32 + 54
   3 : $2 + 10
   4 : int x = $2 + 10;
   5 : System.out.println("Hello");
```

В этом примере мы воспользовались командой `/reset`, которая выполняет очистку всей истории текущего сеанса.

Команда /vars позволяет вывести все доступные переменные:

```
jshell> /vars
|    int $2 = 438
|    int $3 = 448
|    int x = 448
```

Команда /methods выведет названия всех созданных методов:

```
jshell> void printMessage(String msg) {
   ...> System.out.println(msg);
   ...> }
|  created method printMessage(String)

jshell> /methods
|    void printMessage(String)
```

Вывести названия всех созданных классов, интерфейсов и перечислений позволяет команда /types:

```
jshell> class Test {}
|  created class Test

jshell> /types
|    class Test
```

Используя команду /imports можно увидеть инструкции импорта:

```
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*
```

Программа JShell позволяет также автоматически закончить слово или вывести список возможных вариантов. Для этого вводим известные буквы, а затем нажимаем клавишу <Tab>:

```
jshell> System.out.print
print(     printf(    println(
```

Нажатие клавиши <Tab> после открывающей круглой скобки позволяет вывести список форматов метода, а также документацию по каждому формату:

```
jshell> System.out.print(
Signatures:
void PrintStream.print(boolean b)
```

```
void PrintStream.print(char c)
void PrintStream.print(int i)
void PrintStream.print(long l)
void PrintStream.print(float f)
void PrintStream.print(double d)
void PrintStream.print(char[] s)
void PrintStream.print(String s)
void PrintStream.print(Object obj)

<press tab again to see documentation>
```

Получить описание всех команд можно с помощью команды `/help`.

Для завершения текущего сеанса и выхода из JShell предназначена команда `/exit`:

```
jshell> /exit
|  Goodbye
```

ГЛАВА 2

Переменные и типы данных

Переменные — это участки памяти, используемые программой для хранения данных. Как уже говорилось в предыдущей главе, переменная похожа на коробку, в которую можно что-то положить, а через некоторое время достать. Прежде чем использовать переменную, ее необходимо предварительно *объявить*.

2.1. Объявление переменной внутри метода

Для объявления переменной внутри метода используется следующий формат:

```
<Тип данных> <Переменная1>[=<Значение1>]
     [, ..., <ПеременнаяN>[=<ЗначениеN>]];
```

Пример объявления целочисленной переменной x:

```
int x;
```

Можно указать несколько переменных через запятую:

```
int x, y, z;
```

При объявлении допускается сразу задать значения переменным, используя оператор присваивания =:

```
int x = 20;
int y, z = 10, n;
```

Обратите внимание: перед оператором = и после него вставлены пробелы. Количество пробелов может быть произвольным или пробелов может не быть вовсе. Кроме того, допускается вместо пробелов использовать символ перевода строки или символ табуляции. Например, эти инструкции вполне допустимы:

```
int x=25;
int y=                  85;
int z
=
56;
```

Тем не менее, следует придерживаться единообразия в коде и обрамлять операторы одним пробелом. Впрочем, это не строгое правило, а лишь рекомендация по оформлению кода.

Как видно из предыдущего примера, инструкция может быть расположена на нескольких строках. Концом инструкции является точка с запятой, а не конец строки. Например, на одной строке может быть несколько инструкций:

```
int x = 25; int y = 85; int z = 56;
```

Тем не менее, старайтесь избегать размещения нескольких инструкций на одной строке. Придерживайтесь правила — каждая инструкция на отдельной строке.

Объявление переменной можно размещать в любом месте метода. Главное, чтобы переменная была объявлена и инициализирована до ее первого использования. А вот объявить две одноименные переменные в одном блоке нельзя:

```
int x = 25;
System.out.println(x); // 25
int y = 89;
System.out.println(y); // 89
int y;                 // Ошибка. Переменная y уже объявлена!
```

Обратите внимание на то, что сейчас мы говорим об объявлении переменной внутри метода. Переменную можно объявить и в блоке класса вне метода, но в этом случае существуют некоторые дополнительные правила, которые мы рассмотрим немного позже. А вот объявить переменную вне класса нельзя. В языке Java нет переменных уровня файла. Объявление переменной должно быть либо в блоке класса, либо в блоке метода.

2.2. Именование переменных

Каждая переменная должна иметь уникальное имя, состоящее из латинских букв, цифр, знака подчеркивания, символа `$` и (или) любого символа в кодировке Unicode, соответствующего букве. Последнее утверждение говорит, что буквы русского языка также можно использовать в составе имени переменной, но старайтесь этого не делать. Лучше использовать только латинские буквы. Обратите внимание на то, что имя переменной не может начинаться с цифры.

Начиная с Java 9, имя переменной не может состоять только из одного символа подчеркивания:

```
jshell> int _ = 10;
|  Error:
|  as of release 9, '_' is a keyword, and may not be used as an
   identifier
|  int _ = 10;
|      ^
```

С помощью методов `isJavaIdentifierStart()` и `isJavaIdentifierPart()` из класса `Character` можно проверить символ на допустимость использования в имени пере-

менной. Если символ допустим, то методы возвращают значение `true`, в противном случае — `false`. Пример использования:

```
// Допустим ли символ в начале имени?
System.out.println(Character.isJavaIdentifierStart('1')); // false
// Допустим ли символ в любом другом месте?
System.out.println(Character.isJavaIdentifierPart('$'));  // true
```

При указании имени переменной важно учитывать регистр букв: `x` и `X` — разные переменные!

В качестве имени переменной нельзя использовать ключевые слова языка Java:

```
abstract, assert, boolean, break, byte, case, catch, char, class, const,
continue, default, do, double, else, enum, extends, false, final, finally,
float, for, goto, if, implements, import, instanceof, int, interface, long,
native, new, null, package, private, protected, public, return, short, static,
strictfp, super, switch, this, throw, throws, transient, true, try, void,
volatile, while
```

Запоминать все ключевые слова нет необходимости, т. к. в редакторе эти слова подсвечиваются. Любая попытка использования ключевого слова вместо названия переменной приведет к ошибке при компиляции. Помимо ключевых слов следует избегать совпадений со встроенными идентификаторами.

2.3. Типы данных

В языке Java определены восемь простых (элементарных) типов данных:

- `boolean` — логический тип данных. Может содержать значения `true` (истина) или `false` (ложь). Пример объявления переменной:

```
boolean isInt;
isInt = true;
```

- `char` — символ в кодировке UTF-16. Пример объявления переменной:

```
char ch;
ch = 's';
```

- `byte` — целое число в диапазоне от `-128` до `127`. Занимает 1 байт. Пример объявления переменной:

```
byte x;
x = 10;
System.out.println(Byte.MIN_VALUE); // -128
System.out.println(Byte.MAX_VALUE); // 127
```

- `short` — целое число в диапазоне от `-32 768` до `32 767`. Занимает 2 байта. Пример объявления переменной:

```
short x;
x = 20;
System.out.println(Short.MIN_VALUE); // -32768
System.out.println(Short.MAX_VALUE); // 32767
```

- `int` — целое число в диапазоне от `-2 147 483 648` до `2 147 483 647`. Занимает 4 байта. В большинстве случаев этот целочисленный тип наиболее удобен. Пример объявления переменной:

```
int x;
x = 30;
System.out.println(Integer.MIN_VALUE); // -2147483648
System.out.println(Integer.MAX_VALUE); // 2147483647
```

- `long` — целое число в диапазоне от `-9 223 372 036 854 775 808` до `9 223 372 036 854 775 807`. Занимает 8 байтов. Пример объявления переменной:

```
long x;
x = 2147483648L;
System.out.println(Long.MIN_VALUE); // -9223372036854775808
System.out.println(Long.MAX_VALUE); // 9223372036854775807
```

- `float` — вещественное число. Занимает 4 байта. Пример объявления переменной:

```
float x;
x = 127.5F;
System.out.println(Float.MIN_VALUE); // 1.4E-45
System.out.println(Float.MAX_VALUE); // 3.4028235E38
```

- `double` — вещественное число двойной точности. Занимает 8 байтов. В большинстве случаев этот вещественный тип наиболее удобен. Пример объявления переменной:

```
double x;
x = 127.5;
System.out.println(Double.MIN_VALUE); // 4.9E-324
System.out.println(Double.MAX_VALUE); // 1.7976931348623157E308
```

В одной инструкции можно объявить сразу несколько переменных, указав их через запятую после названия типа данных:

```
int x, y, z;
```

После объявления переменной под нее выделяется определенная память, размер которой зависит от используемого типа данных. Обратите внимание на то, что типы данных в языке Java не зависят от разрядности операционной системы. Код на языке Java является машинонезависимым.

2.4. Инициализация переменных

При объявлении переменной ей можно сразу присвоить начальное значение, указав его после оператора =. Эта операция называется *инициализацией переменных*. Пример указания значения:

```
int x, y = 10, z = 30, k;
```

Переменная становится видимой сразу после объявления, поэтому на одной строке с объявлением (после запятой) эту переменную уже можно использовать для инициализации других переменных:

```
int x = 5, y = 10, z = x + y; // z равно 15
```

Присвоить можно значение и уже объявленной переменной, указав его после оператора =. Эта операция называется *присваиванием*. Пример присваивания:

```
int x;
x = 10;
```

Обратите внимание на то, что использовать переменную, которой не присвоено никакого начального значения, нельзя. Например, следующий код приведет к ошибке:

```
int x;
System.out.println(x); // Ошибка. Переменная x не инициализирована
```

Переменной, имеющей тип `boolean`, можно присвоить значения `true` или `false`:

```
boolean a, b;
a = true;
b = false;
```

Переменной, имеющей тип `char`, можно присвоить числовое значение (код символа) или указать символ внутри апострофов. Обратите внимание на то, что использовать кавычки нельзя:

```
char ch1, ch2;
ch1 = 119;
ch2 = 'w';
System.out.println(ch1 + " " + ch2); // w w
```

Внутри апострофов можно указать и *специальные символы* — комбинации знаков, обозначающих служебные или непечатаемые символы. Вот специальные символы, доступные в языке Java:

- `\n` — перевод строки;
- `\r` — возврат каретки;
- `\t` — горизонтальная табуляция;
- `\b` — возврат на один символ;
- `\f` — перевод формата;
- `\'` — апостроф;
- `\"` — кавычка;
- `\\` — обратная косая черта;
- `\uN` — Unicode-код символа в шестнадцатеричном виде (значение `N` от `0000` до `FFFF`).

Пример указания значений:

```
char ch1, ch2;
ch1 = '\u005B'; // Символ [
ch2 = '\u005D'; // Символ ]
System.out.println(ch1 + " " + ch2); // [ ]
```

Целочисленное значение задается в десятичной, двоичной, восьмеричной или шестнадцатеричной форме. Двоичные числа начинаются с комбинации символов `0b` (или `0B`) и могут содержать числа `0` или `1`. Восьмеричные числа начинаются с нуля и содержат цифры от `0` до `7`. Шестнадцатеричные числа начинаются с комбинации символов `0x` (или `0X`) и могут содержать числа от `0` до `9` и буквы от `A` до `F` (регистр букв не имеет значения). Двоичные, восьмеричные и шестнадцатеричные значения преобразуются в десятичное значение:

```
int x, y, z, k;
x = 119;         // Десятичное значение
y = 0167;        // Восьмеричное значение
z = 0x77;        // Шестнадцатеричное значение
k = 0B01110111; // Двоичное значение
System.out.println(x + " " + y + " " + z + " " + k); // 119 119 119 119
```

По умолчанию целочисленные константы, которые мы вводим в программе, имеют тип `int`. Вводимое значение автоматически приводится к типам `byte` и `short`, если оно входит в диапазон значений этих типов. Если значение не входит в диапазон, то компилятор выведет сообщение об ошибке:

```
byte x;  // Диапазон от -128 до 127
x = 100; // OK
x = 300; // Ошибка
```

Диапазон значений у типа `long` гораздо больше, чем диапазон значений у типа `int`. Чтобы задать значение для типа `long`, необходимо после константы указать букву `L` (или `l`):

```
long x;
x = 2147483648L; // OK
x = 2147483648;  // Ошибка. int до 2 147 483 647
```

Вещественное число может содержать точку и (или) экспоненту, начинающуюся с буквы `E` (регистр не имеет значения):

```
double x, y, z, k;
x = 20.0;
y = 12.1e5;
z = .123;     // Эквивалентно z = 0.123;
k = 47.E-5;
```

По умолчанию вещественные константы имеют тип `double`. Чтобы задать значение для переменной, имеющей тип `float`, необходимо после константы указать букву `F` (или `f`):

```
float x, y;
x = 20.0F;
y = 12.1e5f;
```

2.5. Константы

Константы — это переменные, значения в которых не должны изменяться во время работы программы. В более широком смысле под константой понимают любое значение, которое нельзя изменить, — например: `10`, `12.5`, `'W'`, `"string"`.

При объявлении константы перед типом данных указывается ключевое слово `final`:

```
final int MY_CONST = 10;
```

Обратите внимание на то, что в названии константы принято использовать буквы только в верхнем регистре. Если название константы состоит из нескольких слов, то между словами вставляется символ подчеркивания. Это позволяет отличить внутри программы константу от обычной переменной. После объявления константы ее можно использовать в выражениях:

```
final int MY_CONST = 10;
int y;
y = MY_CONST + 20;
```

Присвоить значение константе можно только один раз. Обычно значение присваивается при объявлении константы, хотя можно присвоить значение и позже. Любая попытка повторного присваивания константе значения приведет к ошибке при компиляции:

```
final int MY_CONST;
MY_CONST = 10;      // OK
MY_CONST = 20;      // Ошибка!
```

2.6. Статические переменные и константы класса

Если необходимо получить доступ к переменной из разных методов, то следует объявить переменную внутри блока класса, вне блоков методов. Чтобы переменную можно было использовать без создания экземпляра класса, перед типом данных следует указать ключевое слово `static`. Пример объявления статических переменных:

```
static int x, y = 10;
```

Как видно из примера, статической переменной можно присвоить значение при объявлении. В отличие от обычных переменных, где мы должны инициализировать переменную до первого использования, статические переменные класса инициализируются автоматически. Целочисленным переменным присваивается значение `0`, вещественным — `0.0`, логическим — `false`, объектным — `null`. Иными словами, переменную `x` можно использовать без присваивания какого-либо значения.

Если необходимо, чтобы значение переменной нельзя было менять внутри класса, то ее следует объявить как константу. Для этого после ключевого слова `static` указывается слово `final`. При объявлении статической константы класса необходимо присвоить ей некоторое значение. Пример объявления и инициализации константы класса:

```
static final int MY_CONST = 50;
```

Получить доступ к статической переменной и константе можно либо как обычно, либо указав перед именем переменной название класса через оператор «точка»:

```
x = 10;
MyClass.x = 88;
```

Пример использования статической переменной и константы класса приведен в листинге 2.1.

Листинг 2.1. Статические переменные и константы класса

```
public class MyClass {

   static int x;                    // Объявление статической переменной
   static final int MY_CONST = 50; // Объявление статической константы

   public static void main(String[] args) {
      System.out.println(x);                // 0
      x = 10;
      System.out.println(x);                // 10
      MyClass.x = 88;
      System.out.println(x);                // 88
      System.out.println(MyClass.x);        // 88
      System.out.println(MY_CONST);         // 50
      System.out.println(MyClass.MY_CONST); // 50
   }
}
```

2.7. Области видимости переменных

Прежде чем использовать переменную, ее необходимо предварительно объявить. До объявления переменная не видна в программе. Объявить переменную можно глобально (вне методов) или локально (внутри методов или блока).

- *Глобальные переменные* — это статические переменные, объявленные в программе вне методов в блоке класса. Глобальные переменные видны в любой части класса, включая методы. Если при объявлении переменной не было присвоено начальное значение, то производится ее автоматическая инициализация. Целочисленным переменным присваивается значение `0`, вещественным — `0.0`, логическим — `false`, объектным — `null`.

- *Локальные переменные* — это переменные, которые объявлены внутри метода. Локальные переменные видны только внутри метода или вложенного блока. Инициализация таких переменных производится при каждом вызове метода. После выхода из метода локальная переменная уничтожается. Локальные переменные обязательно должны быть инициализированы до первого использования.
- *Локальные переменные внутри блока* — это переменные, которые объявлены внутри метода в неименованном блоке (в области, ограниченной фигурными скобками). Такие переменные видны только внутри блока и во вложенных блоках. Инициализация таких переменных производится при каждом входе в блок. После выхода из блока переменная уничтожается. Внутри блока нельзя объявить переменную, если одноименная локальная переменная была объявлена ранее, но можно объявить переменную, совпадающую с именем глобальной переменной.

Если имя локальной переменной совпадает с именем глобальной переменной, то все операции внутри метода осуществляются с локальной переменной, а значение глобальной не изменяется. Чтобы в этом случае получить доступ к глобальной переменной, необходимо перед названием переменной указать название класса через оператор «точка», например, так: `MyClass.x`.

Область видимости глобальных и локальных переменных показана в листинге 2.2.

Листинг 2.2. Область видимости переменных

```
public class MyClass {

   static int x;                        // Глобальная переменная

   public static void main(String[] args) {
      x = 10;
      System.out.println(x);            // 10
      func();                           // Вызов метода func()
      System.out.println(x);            // 88
   }
   public static void func() {
      int x = 30;                       // Локальная переменная
      System.out.println(x);            // 30
      System.out.println(MyClass.x);    // 10
      MyClass.x = 88;
   }
}
```

Очень важно учитывать, что переменная, объявленная внутри блока, видна только в пределах блока (внутри фигурных скобок). Например, присвоим значение переменной в зависимости от некоторого условия:

```
// Неправильно!!!
if (x == 10) { // Какое-то условие
   int y;
   y = 5;
   System.out.println(y);
}
else {
   int y;
   y = 25;
   System.out.println(y);
}
System.out.println(y); // Ошибка! Переменная y здесь не видна!
```

В этом примере переменная `y` видна только внутри блока. После условного оператора `if` переменной не существует. Чтобы переменная была видна внутри блока и после выхода из него, необходимо поместить объявление переменной перед блоком:

```
// Правильно
int y = 0;
if (x == 10) { // Какое-то условие
   y = 5;
   System.out.println(y);
}
else {
   y = 25;
   System.out.println(y);
}
System.out.println(y);
```

Язык Java поддерживает также неименованные блоки. Такие блоки не привязаны ни к какой инструкции — просто фигурные скобки сами по себе. Если переменная объявлена внутри такого блока, то после блока она уже не будет видна:

```
{ // Блок
   int y;
   y = 5;
   System.out.println(y);
}
System.out.println(y); // Ошибка! Переменная y здесь не видна!
```

Следует учитывать, что внутри неименованного блока нельзя объявить переменную, если одноименная локальная переменная была объявлена ранее:

```
int x;        // Локальная переменная
{
   int x;     // Ошибка! Одноименная локальная переменная существует
   int y;     // Локальная переменная внутри блока
   {
      int y; // Ошибка! Одноименная локальная переменная существует
   }
}
```

В цикле `for` переменная, объявленная внутри круглых скобок, видна только внутри блока:

```
for (int i = 0; i < 10; i++) {
   System.out.println(i);
}
System.out.println(i); // Ошибка! Переменная i здесь не видна!
```

Если обращение к переменной внутри метода или блока производится до объявления одноименной локальной переменной, то до объявления будет использоваться глобальная переменная, а после объявления — локальная переменная (листинг 2.3). Если глобальной переменной с таким названием не существует, то при компиляции произойдет ошибка.

Листинг 2.3. Обращение к переменной до объявления внутри метода

```
public class MyClass {

   static int x;                        // Глобальная переменная

   public static void main(String[] args) {
      x = 10;
      System.out.println(x);            // 10
      int x = 88;                       // Локальная переменная
      System.out.println(x);            // 88
      System.out.println(MyClass.x); // 10
   }
}
```

2.8. Преобразование и приведение типов

Как вы уже знаете, при объявлении переменной необходимо указать определенный тип данных. Далее над переменной можно производить операции, предназначенные для этого типа данных. Если в выражении используются числовые переменные, имеющие разные типы данных, то тип результата выражения будет соответствовать наиболее сложному типу. Например, если производится сложение переменной, имеющей тип `int`, с переменной, имеющей тип `double`, то целое число будет автоматически преобразовано в вещественное. Результатом этого выражения станет значение, имеющее тип `double`. Правила автоматического преобразования следующие:

- если один из операндов типа `double`, то второй операнд преобразуется в тип `double`;
- в противном случае, если один из операндов типа `float`, то второй операнд преобразуется в тип `float`;
- в противном случае, если один из операндов типа `long`, то второй операнд преобразуется в тип `long`;
- в противном случае оба операнда преобразуются в тип `int`.

Последнее правило означает, что результатом операций с типами byte и short будет как минимум тип int:

```
byte y1 = 1, y2 = 2;
y1 = y1 + y2;                    // Ошибка! Тип int
short z1 = 1, z2 = 2;
z1 = z1 + z2;                    // Ошибка! Тип int
```

Чтобы результат выполнения этих выражений сохранить в переменных, необходимо выполнить операцию *приведения типов*. Формат операции:

```
(<Тип результата>)<Выражение или переменная>
```

Пример приведения типов:

```
byte y1 = 1, y2 = 2;
y1 = (byte)(y1 + y2);            // OK
short z1 = 1, z2 = 2;
z1 = (short)(z1 + z2);           // OK
```

В этом случае компилятор считает, что мы знаем, что делаем, и осведомлены о возможном несоответствии значения указанному типу данных. Если значение выражения будет выходить за диапазон значений типа, то произойдет усечение результата, но никакого сообщения об ошибке не появится. Давайте рассмотрим это на примере:

```
byte y1 = 127, y2 = 0;
y1 = (byte)(y1 + y2);            // OK. 127
byte z1 = 127, z2 = 10;
z1 = (byte)(z1 + z2);            // Усечение. -119
```

Результатом первого выражения будет число 127, которое входит в диапазон значений типа byte, а вот результат второго выражения (число 137) — в диапазон не входит, и происходит усечение. В итоге мы получили число –119. Согласитесь — это совсем не то, что мы хотели получить. Поэтому приведением типов нужно пользоваться осторожно. Хотя в некоторых случаях такое усечение очень даже полезно. Например, вещественное число можно преобразовать в целое. В этом случае дробная часть просто отбрасывается:

```
float x = 1.2f;
double y = 2.5;
System.out.println(x);          // 1.2
System.out.println( (int)x ); // 1
System.out.println(y);          // 2.5
System.out.println( (int)y ); // 2
```

Преобразование без потерь данных происходит в следующих случаях:

- тип byte без потерь преобразуется в типы short, int, long, double;
- тип short — в типы int, long, double;
- тип int — в типы long, double;
- тип char — в тип int, long, double.

Рассмотрим пример, который демонстрирует частый способ применения приведения типов. В языке Java деление целых чисел всегда возвращает целое число. Дробная часть при этом просто отбрасывается. Чтобы деление целых чисел возвращало вещественное число, необходимо преобразовать одно из целых чисел в вещественное (второе число преобразуется автоматически):

```
int x = 10, y = 3;
System.out.println( x / y );          // 3
System.out.println( (double)x / y ); // 3.3333333333333335
```

2.9. Инструкция *var*

Начиная с Java 10, при объявлении локальной переменной вместо конкретного типа данных можно указать слово `var`. В этом случае тип переменной будет определяться компилятором динамически на основе контекста при инициализации:

```
var a = true;              // Переменная имеет тип boolean
System.out.println( a );   // true
var ch = 'w';              // Переменная имеет тип char
System.out.println( ch ); // w
var x = 10;                // Переменная имеет тип int
System.out.println( x );   // 10
var y = 10.5;              // Переменная имеет тип double
System.out.println( y );   // 10.5
var s = "строка";          // Переменная имеет тип String
System.out.println( s );   // строка
```

Пример использования инструкции `var` совместно с циклом `for`:

```
for (var i = 0; i < 10; i++) {
   System.out.println(i);
}
```

При использовании обобщенных типов преимущества инструкции `var` становятся особенно заметны. Например, вместо:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
```

мы можем написать так:

```
var arr = new ArrayList<Integer>();
```

Тип определяется при инициализации, поэтому так написать нельзя:

```
var x;                        // Ошибка! Так нельзя!
x = 10;
```

Кроме того, нельзя объявлять сразу несколько переменных в одной инструкции:

```
var x = 10, y = 20;           // Ошибка! Так нельзя!
```

Нельзя использовать слово `var` при объявлении глобальных переменных, параметров методов и вместо типа возвращаемого методом значения.

Слово `var` не является зарезервированным ключевым словом, поэтому мы можем его использовать, например, в качестве имени переменной:

```
var var = 10;                // OK, но лучше так не делать!
System.out.println( var );   // 10
```

2.10. Перечисления

Перечисление — это совокупность констант, описывающих все допустимые значения переменной. Если переменной присвоить значение, не совпадающее с перечисленными при объявлении константами, то компилятор выведет сообщение об ошибке. Объявление перечисления имеет следующий формат:

```
[<Модификатор доступа>] enum <Название перечисления> {
   <Список констант через запятую>
}
```

Пример объявления перечисления:

```
enum Color { RED, BLUE, GREEN, BLACK }
```

Название перечисления становится новым типом данных, который указывается при объявлении переменной. Вот пример объявления двух переменных перечисления `Color`:

```
Color color1, color2;
```

В дальнейшем можно присвоить переменной одну из констант или значение `null`, означающее отсутствие значения:

```
color1 = Color.RED;
color2 = Color.BLACK;
color1 = null;
```

Перечисления допускают операцию сравнения значений с помощью операторов `==` и `!=`:

```
color1 == Color.RED
color1 != color2
```

Пример использования перечисления приведен в листинге 2.4.

Листинг 2.4. Перечисления

```
public class MyClass {
   public static void main(String[] args) {
      Color color1, color2;       // Объявление переменной
      color1 = Color.RED;
      color2 = Color.BLACK;
      if (color1 == Color.RED) {  // Проверка значения
         System.out.println("color1 == RED");
      }
```

```
      if (color1 != color2) {     // Проверка значения
         System.out.println("color1 != color2");
      }
      System.out.println(color1); // Выведет: RED
   }
}

// Объявление перечисления
enum Color { RED, BLUE, GREEN, BLACK }
```

Здесь мы поместили объявление перечисления в одном файле с классом. Обратите внимание: оно расположено вне блока класса. Однако чаще всего объявление перечисления размещают в отдельном файле. Чтобы создать такой файл, в меню **File** выбираем пункт **New | Enum**. В открывшемся окне в поле **Name** вводим название перечисления и нажимаем кнопку **Finish**. В этом случае перед ключевым словом `enum` можно указать модификатор доступа, например, `public`:

```
// Файл Color.java
public enum Color {
   RED, BLUE, GREEN, BLACK
}
```

ГЛАВА 3

Операторы и циклы

Операторы позволяют выполнить с данными определенные действия. Например, операторы *присваивания* служат для сохранения данных в переменной, *математические* операторы предназначены для арифметических вычислений, а *условные* операторы позволяют в зависимости от значения логического выражения выполнить отдельный участок программы или, наоборот, не выполнять его.

3.1. Математические операторы

Производить арифметические вычисления позволяют следующие операторы:

- `+` — сложение:

```
System.out.println( 10 + 15 );          // 25
```

- `-` — вычитание:

```
System.out.println( 35 - 15 );          // 20
```

- `-` — унарный минус:

```
int x = 10;
System.out.println( -x );              // -10
```

- `*` — умножение:

```
System.out.println( 25 * 2 );          // 50
```

- `/` — деление. Если производится деление целых чисел, то остаток отбрасывается и возвращается целое число. Деление вещественных чисел производится классическим способом. Если в выражении участвуют вещественное и целое числа, то целое число автоматически преобразуется в вещественное:

```
System.out.println( 10 / 3 );          // 3
System.out.println( 10.0 / 3.0 );      // 3.3333333333333335
System.out.println( 10.0 / 3 );        // 3.3333333333333335
System.out.println( (double)10 / 3 ); // 3.3333333333333335
```

Целочисленное деление на `0` приведет к ошибке при выполнении программы. Деление вещественного числа на `0` приведет к значению плюс или минус

`Infinity` (бесконечность), а деление вещественного числа `0.0` на `0` — к значению `NaN` (нет числа):

```
System.out.println( 10.0 / 0 );       // Infinity
System.out.println( -10.0 / 0 );      // -Infinity
System.out.println( 0.0 / 0 );        // NaN
```

- `%` — остаток от деления:

```
System.out.println( 10 % 2 );         // 0 (10 - 10 / 2 * 2)
System.out.println( 10 % 3 );         // 1 (10 - 10 / 3 * 3)
System.out.println( 10 % 4 );         // 2 (10 - 10 / 4 * 4)
System.out.println( 10 % 6 );         // 4 (10 - 10 / 6 * 6)
```

- `++` — оператор инкремента. Увеличивает значение переменной на `1`:

```
int x = 10;
x++;                          // Эквивалентно x = x + 1;
System.out.println( x );      // 11
```

- `--` — оператор декремента. Уменьшает значение переменной на `1`:

```
int x = 10;
x--;                          // Эквивалентно x = x - 1;
System.out.println( x );      // 9
```

Операторы инкремента и декремента могут использоваться в постфиксной или префиксной формах:

```
x++; x--; // Постфиксная форма
++x; --x; // Префиксная форма
```

При постфиксной форме (`x++`) возвращается значение переменной перед операцией, а при префиксной форме (`++x`) — вначале производится операция и только потом возвращается значение. Продемонстрируем это на примере (листинг 3.1).

Листинг 3.1. Постфиксная и префиксная формы

```
public class MyClass {
   public static void main(String[] args) {
      int x = 0, y = 0;
      x = 5;
      y = x++; // y = 5, x = 6
      System.out.println("Постфиксная форма (y = x++;):");
      System.out.println("y = " + y);
      System.out.println("x = " + x);
      x = 5;
      y = ++x; // y = 6, x = 6
      System.out.println("Префиксная форма (y = ++x;):");
      System.out.println("y = " + y);
      System.out.println("x = " + x);
   }
}
```

В итоге получим следующий результат:

```
Постфиксная форма (y = x++;):
y = 5
x = 6
Префиксная форма (y = ++x;):
y = 6
x = 6
```

Если операторы инкремента и декремента используются в сложных выражениях, то понять, каким будет результат выполнения выражения, становится сложно. Например, каким будет значение переменной `y` после выполнения этих инструкций?

```
int x = 5, y = 0;
y = ++x + ++x + ++y;
```

Чтобы облегчить жизнь себе и всем другим программистам, которые будут разбираться в программе, операторы инкремента и декремента лучше использовать отдельно от других операторов.

3.2. Побитовые операторы

Побитовые операторы предназначены для манипуляции отдельными битами. Язык Java поддерживает следующие побитовые операторы:

- `~` — двоичная инверсия. Значение каждого бита заменяется противоположным:

```
int x = 100;
System.out.printf("%32s\n", Integer.toBinaryString(x));
//                           1100100
x = ~x;
System.out.printf("%32s\n", Integer.toBinaryString(x));
// 11111111111111111111111110011011
```

- `&` — двоичное И:

```
int x = 100, y = 75;
int z = x & y;
System.out.printf("%32s\n", Integer.toBinaryString(x));
//                           1100100
System.out.printf("%32s\n", Integer.toBinaryString(y));
//                           1001011
System.out.printf("%32s\n", Integer.toBinaryString(z));
//                           1000000
```

- `|` — двоичное ИЛИ:

```
int x = 100, y = 75;
int z = x | y;
System.out.printf("%32s\n", Integer.toBinaryString(x));
//                           1100100
```

```
System.out.printf("%32s\n", Integer.toBinaryString(y));
//                           1001011
System.out.printf("%32s\n", Integer.toBinaryString(z));
//                           1101111
```

- ^ — двоичное исключающее ИЛИ:

```
int x = 100, y = 250;
int z = x ^ y;
System.out.printf("%32s\n", Integer.toBinaryString(x));
//                           1100100
System.out.printf("%32s\n", Integer.toBinaryString(y));
//                          11111010
System.out.printf("%32s\n", Integer.toBinaryString(z));
//                          10011110
```

- << — сдвиг влево. Сдвигает двоичное представление числа влево на один или более разрядов и заполняет разряды справа нулями:

```
int x = 100;
System.out.printf("%32s\n", Integer.toBinaryString(x));
//                           1100100
x = x << 1;
System.out.printf("%32s\n", Integer.toBinaryString(x));
//                          11001000
x = x << 1;
System.out.printf("%32s\n", Integer.toBinaryString(x));
//                         110010000
x = x << 2;
System.out.printf("%32s\n", Integer.toBinaryString(x));
//                       11001000000
```

- >> — сдвиг вправо. Сдвигает двоичное представление числа вправо на один или более разрядов и заполняет разряды слева нулями, если число положительное:

```
int x = 100;
System.out.printf("%32s\n", Integer.toBinaryString(x));
//                           1100100
x = x >> 1;
System.out.printf("%32s\n", Integer.toBinaryString(x));
//                            110010
x = x >> 1;
System.out.printf("%32s\n", Integer.toBinaryString(x));
//                              11001
x = x >> 2;
System.out.printf("%32s\n", Integer.toBinaryString(x));
//                                 110
```

Если число отрицательное, то разряды слева заполняются единицами:

```
int x = -127;
System.out.printf("%32s\n", Integer.toBinaryString(x));
// 11111111111111111111111110000001
```

```
x = x >> 1;
System.out.printf("%32s\n", Integer.toBinaryString(x));
// 11111111111111111111111111000000
x = x >> 1;
System.out.printf("%32s\n", Integer.toBinaryString(x));
// 11111111111111111111111111100000
x = x >> 2;
System.out.printf("%32s\n", Integer.toBinaryString(x));
// 11111111111111111111111111111000
```

- >>> — сдвиг вправо. Сдвигает двоичное представление числа вправо на один или более разрядов и заполняет разряды слева нулями, даже если число отрицательное:

```
int x = -127;
System.out.printf("%32s\n", Integer.toBinaryString(x));
// 11111111111111111111111110000001
x = x >>> 1;
System.out.printf("%32s\n", Integer.toBinaryString(x));
//  01111111111111111111111111000000
x = x >>> 8;
System.out.printf("%32s\n", Integer.toBinaryString(x));
//          00000000111111111111111111111111
```

При использовании типов `byte` и `short` надо учитывать, что результат выражения будет соответствовать типу `int`. Поэтому следует выполнить приведение типов:

```
byte x = 100, y = 75, z;
z = (byte)(x & y);
```

Наиболее часто двоичное представление числа используется для хранения различных флагов (`0` — флаг сброшен, `1` — флаг установлен). Примеры установки, снятия и проверки установки флага приведены в листинге 3.2.

Листинг 3.2. Работа с флагами

```
public class MyClass {
   public static void main(String[] args) {
      final byte FLAG1 = 1,  FLAG2 = 2,  FLAG3 = 4, FLAG4 = 8,
                 FLAG5 = 16, FLAG6 = 32, FLAG7 = 64;
      byte x = 0; // Все флаги сброшены
      // Устанавливаем флаги  FLAG1 и FLAG7
      x = (byte)(x  | FLAG1 | FLAG7);
      System.out.println(Integer.toBinaryString(x)); // 1000001
      // Устанавливаем флаги  FLAG4 и FLAG5
      x = (byte)(x  | FLAG4 | FLAG5);
      System.out.println(Integer.toBinaryString(x)); // 1011001
      // Снимаем флаги  FLAG4 и FLAG5
      x = (byte)(x  ^ FLAG4 ^ FLAG5);
      System.out.println(Integer.toBinaryString(x)); // 1000001
```

```
        // Проверка установки флага FLAG1
        if ((x & FLAG1) != 0) {
            System.out.println("FLAG1 установлен");
        }
    }
}
```

В качестве еще одного примера выведем статус всех флагов (листинг 3.3).

Листинг 3.3. Вывод статуса флагов

```
public class MyClass {
    public static void main(String[] args) {
        int x = 0b1011001;
        int n = 0; // Индекс бита
        while (x != 0) {
            if ((x & 1) != 0) { // Проверка статуса последнего бита
                System.out.println(n + " установлен");
            }
            else {
                System.out.println(n + " сброшен");
            }
            x = x >>> 1; // Сдвиг на один разряд вправо
            n++;
        }
    }
}
```

3.3. Операторы присваивания

Операторы присваивания предназначены для сохранения значения в переменной. Язык Java поддерживает следующие операторы присваивания:

- = — присваивает переменной значение. Обратите внимание на то, что хотя оператор похож на математический знак равенства, смысл у него совершенно другой. Справа от оператора присваивания может располагаться константа или сложное выражение. Слева от оператора присваивания может располагаться только переменная. Пример присваивания значения:

```
int x;
x = 10;
x = 12 * 10 + 45 / 5;
12 + 45 = x; // Так нельзя!!!
```

  В одной инструкции можно присвоить значение сразу нескольким переменным:

```
int x, y, z;
x = y = z = 2;
```

- `+=` — увеличивает значение переменной на указанную величину:

```
x += 10;      // Эквивалентно x = x + 10;
```

- `-=` — уменьшает значение переменной на указанную величину:

```
x -= 10;      // Эквивалентно x = x - 10;
```

- `*=` — умножает значение переменной на указанную величину:

```
x *= 10 + 5; // Эквивалентно x = x * (10 + 5);
```

- `/=` — делит значение переменной на указанную величину:

```
x /= 2;       // Эквивалентно x = x / 2;
```

- `%=` — делит значение переменной на указанную величину и возвращает остаток:

```
x %= 2;       // Эквивалентно x = x % 2;
```

- `&=`, `|=`, `^=`, `<<=` , `>>=` и `>>>=` — побитовые операторы с присваиванием.

3.4. Операторы сравнения

Операторы сравнения используются в логических выражениях. Язык Java поддерживает следующие операторы сравнения:

- `==` — равно;
- `!=` — не равно;
- `<` — меньше;
- `>` — больше;
- `<=` — меньше или равно;
- `>=` — больше или равно.

Логические выражения возвращают только два значения: `true` (истина) или `false` (ложь). Пример вывода значения логического выражения:

```
System.out.println( 10 == 10 ); // true
System.out.println( 10 == 5 );  // false
System.out.println( 10 != 5 );  // true
System.out.println( 10 < 5 );   // false
System.out.println( 10 > 5 );   // true
System.out.println( 10 <= 5 );  // false
System.out.println( 10 >= 5 );  // true
```

Следует учитывать, что оператор проверки на равенство содержит два символа `=`. Указание одного символа `=` является ошибкой, т. к. этот оператор служит для присваивания значения переменной, а не для проверки условия.

Значение логического выражения можно инвертировать с помощью оператора `!`. Если логическое выражение возвращает `false`, то `!false` вернет значение `true`:

```
System.out.println(   10 == 5  );  // false
System.out.println( !(10 == 5) );  // true
```

Несколько логических выражений можно объединить в одно большое с помощью следующих операторов:

- `&&` — логическое И. Логическое выражение вернет `true` только в случае, если оба подвыражения вернут `true`:

```
System.out.println( (10 == 10) && (5 != 3)  ); // true
System.out.println( (10 == 10) && (5 == 3) );  // false
```

- `||` — логическое ИЛИ. Логическое выражение вернет `true`, если хотя бы одно из подвыражений вернет `true`. Пример использования оператора:

```
System.out.println( (10 == 10) || (5 != 3) ); // true
System.out.println( (10 == 10) || (5 == 3) ); // true
```

Если первое выражение вернет значение `true`, то второе выражение даже не будет вычисляться. Например, в этом выражении деление на `0` никогда не будет выполнено (следовательно, ошибки не будет):

```
int x = 0;
System.out.println( (10 == 10) || ((10 / x) > 0) ); // true
```

Результаты выполнения операторов `&&` и `||` показаны в табл. 3.1.

***Таблица 3.1.** Операторы `&&` и `||`*

a	b	a && b	a || b
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

3.5. Приоритет выполнения операторов

Все операторы выполняются в порядке *приоритета* (табл. 3.2). Вначале вычисляется выражение, в котором оператор имеет наивысший приоритет, а затем выражение с меньшим приоритетом. Например, выражение с оператором умножения будет выполнено раньше выражения с оператором сложения, т. к. приоритет оператора умножения выше. Если приоритет операторов одинаковый, то используется порядок вычисления, определенный для конкретного оператора. Операторы присваивания и унарные операторы выполняются справа налево. Математические, побитовые и операторы сравнения выполняются слева направо. Изменить последовательность вычисления выражения можно с помощью круглых скобок:

```
int x = 0;
x = 5 + 10 * 3 / 2;       // Умножение -> деление -> сложение
```

```
System.out.println( x ); // 20
x = (5 + 10) * 3 / 2;    // Сложение -> умножение -> деление
System.out.println( x ); // 22
```

Таблица 3.2. Приоритеты операторов

Приоритет	Операторы
1 (высший)	`()` (вызов метода) `[] .`
2	`++ -- ~ !` () (приведение) – (унарный минус) `new`
3	`* / %`
4	`+ –` (минус)
5	`<< >> >>>`
6	`< > <= >= instanceof`
7	`== !=`
8	`&` (побитовое И)
9	`^`
10	`|`
11	`&&`
12	`||`
13	`?:`
14 (низший)	`= *= /= %= += -= >>= >>>= <<= &= ^= |=`

3.6. Оператор ветвления *if*

Оператор ветвления `if` позволяет в зависимости от значения логического выражения выполнить отдельный блок программы или, наоборот, не выполнять его. Оператор имеет следующий формат:

```
if (<Логическое выражение>) {
   <Блок, выполняемый, если условие истинно>
}
[else {
   <Блок, выполняемый, если условие ложно>
}]
```

Если логическое выражение возвращает значение `true` (истина), то выполняются инструкции, расположенные внутри фигурных скобок сразу после оператора `if`. Если логическое выражение возвращает значение `false` (ложь), то выполняются инструкции после ключевого слова `else`. Блок `else` не является обязательным. Допускается не указывать фигурные скобки, если блоки состоят из одной инструкции. В качестве примера проверим целое число, введенное пользователем, на четность и выведем соответствующее сообщение (листинг 3.4).

Листинг 3.4. Проверка числа на четность

```
import java.util.Scanner;

public class MyClass {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      int x = 0;
      System.out.print("Введите целое число: ");
      x = in.nextInt();
      if (x % 2 == 0)
         System.out.print(x + " - четное число");
      else
         System.out.print(x + " - нечетное число");
      System.out.println();
   }
}
```

Что будет, если вместо целого числа мы введем в окне консоли, например, строку, не содержащую число? В этом случае метод `nextInt()` не сможет преобразовать строку в число, и программа аварийно завершится. Способы обработки таких ошибок мы рассмотрим в этой книге позже. А пока набирайте только целые числа!

В зависимости от условия `x % 2 == 0` выводится соответствующее сообщение. Если число делится на `2` без остатка, то оператор `%` вернет значение `0`, в противном случае — число `1`. Обратите внимание на то, что оператор ветвления не содержит фигурных скобок:

```
if (x % 2 == 0)
   System.out.print(x + " - четное число");
else
   System.out.print(x + " - нечетное число");
System.out.println();
```

В этом случае считается, что внутри блока содержится только одна инструкция. Поэтому последняя инструкция к блоку `else` не относится. Она будет выполнена в любом случае, вне зависимости от условия. Чтобы это сделать наглядным, перед инструкциями, расположенными внутри блока, добавлено одинаковое количество пробелов. Впрочем, ничего не изменится, если записать приведенный код и следующим образом:

```
if (x % 2 == 0)
System.out.print(x + " - четное число");
else
System.out.print(x + " - нечетное число");
System.out.println();
```

Однако в дальнейшем разбираться в таком коде будет неудобно самому программисту. Поэтому перед инструкциями внутри блока всегда следует размещать оди-

наковые отступы. В качестве отступа, как уже отмечалось в *главе 1*, можно использовать пробелы или символы табуляции. При использовании пробелов размер отступа для блока первого уровня равняется трем или четырем пробелам. Для вложенных блоков количество пробелов умножают на уровень вложенности: если для блока первого уровня вложенности вставлялись три пробела, то для блока второго уровня вложенности следует вставить шесть пробелов, для третьего уровня — девять и т. д. В одной программе в качестве отступа не следует использовать и пробелы, и табуляцию, — необходимо выбрать что-то одно и пользоваться этим во всей программе.

При отсутствии пробелов или фигурных скобок чтение программы даже у опытных программистов может вызывать затруднения. Даже если вы знаете приоритет выполнения операторов, всегда закрадывается сомнение, и чтение программы приостанавливается. Например, к какому оператору `if` принадлежит блок `else` в этом примере?

```
if (x >= 0) if (x == 0) y = 0; else y = 1;
```

Задумались? А зачем задумываться об этом, если можно сразу расставить фигурные скобки? Ведь тогда никаких сомнений вообще не будет:

```
if (x >= 0) { if (x == 0) y = 0; else y = 1; }
```

А если сделать так, то чтение и понимание программы станет мгновенным:

```
if (x >= 0) {
   if (x == 0) y = 0;
   else y = 1;
}
```

Если блок состоит из нескольких инструкций, то следует указать фигурные скобки. Существует несколько стилей размещения скобок в операторе `if`:

```
// Стиль 1
if (<Логическое выражение>) {
   // Инструкции
}
else {
   // Инструкции
}
// Стиль 2
if (<Логическое выражение>) {
   // Инструкции
} else {
   // Инструкции
}
// Стиль 3
if (<Логическое выражение>)
{
   // Инструкции
}
```

```
else
{
   // Инструкции
}
```

Многие программисты считают Стиль 3 наиболее приемлемым, т. к. открывающая и закрывающая скобки расположены друг под другом. На мой же взгляд здесь образуются лишние пустые строки. А поскольку размеры экрана ограничены, при наличии пустых строк на экране помещается меньше кода, и приходится чаще пользоваться полосой прокрутки. Если же размещать инструкции с равными отступами, то блок кода выделяется визуально, и следить за положением фигурных скобок просто излишне. Тем более, что редактор кода позволяет подсветить парные скобки. Какой стиль использовать, зависит от личного предпочтения программиста или от правил оформления кода, принятых в той или иной фирме. Главное, чтобы стиль оформления внутри одной программы был одинаковым. В этой книге мы будем использовать Стиль 1.

Один условный оператор можно вложить в другой. Например, так:

```
if (<Условие 1>) {
   // Инструкции
}
else {
   if (<Условие 2>) {
      // Инструкции
   }
   else {
      // Инструкции
   }
}
```

Чтобы проверить несколько условий, эту схему можно изменить так:

```
if (<Условие 1>) {
   // <Блок 1>
}
else if (<Условие 2>) {
   // <Блок 2>
}
// ... Фрагмент опущен ...
else if (<Условие N>) {
   // <Блок N>
}
else {
   // <Блок else>
}
```

Если <Условие 1> истинно, то выполняется <Блок 1>, а все остальные условия пропускаются. Если <Условие 1> ложно, то проверяется <Условие 2>. Если <Условие 2>

истинно, то выполняется <Блок 2>, а все остальные условия пропускаются. Если <Условие 2> ложно, то точно так же проверяются остальные условия. Если все условия ложны, то выполняется <Блок else>. В качестве примера определим, какое число от 0 до 2 ввел пользователь (листинг 3.5).

Листинг 3.5. Проверка нескольких условий

```
import java.util.Scanner;

public class MyClass {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      int x = 0;
      System.out.print("Введите число от 0 до 2: ");
      x = in.nextInt();
      if (x == 0)
         System.out.println("Вы ввели число 0");
      else if (x == 1)
         System.out.println("Вы ввели число 1");
      else if (x == 2)
         System.out.println("Вы ввели число 2");
      else {
         System.out.println("Вы ввели другое число");
         System.out.println("x = " + x);
      }
   }
}
```

3.7. Оператор ?:

Для проверки условия вместо оператора `if` можно использовать оператор `?:`. Оператор имеет следующий формат:

```
<Переменная> = <Логическое выражение> ? <Выражение если Истина> :
                                        <Выражение если Ложь>;
```

Если логическое выражение возвращает значение `true`, то выполняется выражение, расположенное после вопросительного знака. Если логическое выражение возвращает значение `false`, то выполняется выражение, расположенное после двоеточия. Результат выполнения выражений становится результатом выполнения оператора. Пример проверки числа на четность и вывода результата:

```
String s;
int x = 10;
System.out.print(x);
s = x % 2 == 0 ? " - четное число" : " - нечетное число";
System.out.println(s);
```

Обратите внимание на то, что в качестве операндов указываются именно выражения, а не инструкции, заканчивающиеся точкой с запятой. Кроме того, выражения обязательно должны возвращать какое-либо значение, причем одинакового типа. Так как оператор возвращает значение, его можно использовать внутри выражений:

```
int x, y;
x = 0;
y = 30 + 10 / (x==0 ? 1 : x);    // 30 + 10 / 1
System.out.println(y);           // 40
x = 2;
y = 30 + 10 / (x==0 ? 1 : x);    // 30 + 10 / 2
System.out.println(y);           // 35
```

Если необходимо выполнить несколько инструкций, то в качестве операнда можно указать метод, который возвращает значение:

```
public static String func1() {
   // Инструкции
   return " - четное число"; // Возвращаемое значение
}
public static String func2() {
   return " - нечетное число";
}
// ... Фрагмент опущен ...
String s;
int x = 10;
s = (x % 2 == 0) ? func1() : func2();
System.out.println(x + s);
```

3.8. Оператор выбора *switch*

Оператор выбора `switch` имеет следующий формат:

```
switch (<Выражение>) {
   case <Константа 1>:
      <Инструкции>
      break;
[...
   case <Константа N>:
      <Инструкции>
      break;]
[   default:
      <Инструкции>]
}
```

Вместо условия оператор `switch` принимает выражение. В зависимости от значения выражения выполняется один из блоков `case`, в котором указано это значение. Значением выражения должно быть целое число (тип `int`, `byte` или `short`, но не `long`), символ или перечислимый тип. Если ни одно из значений не описано в блоках `case`,

то выполняется блок `default` (если он указан). Обратите внимание на то, что значения в блоках `case` не могут иметь одинаковые константы. Пример использования оператора `switch` приведен в листинге 3.6.

Листинг 3.6. Использование оператора `switch`

```
import java.util.Scanner;

public class MyClass {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      int os = 0;
      System.out.print(
         "Какой операционной системой вы пользуетесь?\n\n"
         + "1 - Windows XP\n"
         + "2 - Windows 8\n"
         + "3 - Windows 10\n"
         + "4 - Другая\n\n"
         + "Введите число, соответствующее ответу: "
      );
      os = in.nextInt();
      switch (os) {
         case 1:
            System.out.println("Вы выбрали - Windows XP");
            break;
         case 2:
            System.out.println("Вы выбрали - Windows 8");
            break;
         case 3:
            System.out.println("Вы выбрали - Windows 10");
            break;
         case 4:
            System.out.println("Вы выбрали - Другая");
            break;
         default:
            System.out.println(
               "Мы не смогли определить систему");
      }
   }
}
```

Как видно из примера, в конце каждого блока `case` указан оператор `break`. Этот оператор позволяет досрочно выйти из оператора выбора `switch`. Если не указать оператор `break`, то будет выполняться следующий блок `case` вне зависимости от указанного значения. В большинстве случаев такая запись приводит к ошибкам, однако в некоторых случаях это может быть полезным. Например, можно выпол-

нить одни и те же инструкции при разных значениях, поместив инструкции в конце диапазона значений:

```
char ch = 'b';
switch (ch) {
   case 'a':
   case 'b':
   case 'c':
      System.out.println("a, b или c");
      break;
   case 'd':
      System.out.println("Только d");
}
```

В операторе `case` можно указать одно из значений перечисления. Причем предварять это значение именем перечисления через оператор «точка» не нужно. Пример проверки выбранного значения перечисления приведен в листинге 3.7.

Листинг 3.7. Использование оператора `switch` с перечислениями

```
public class MyClass {
   public static void main(String[] args) {
      Color color = Color.BLUE;
      switch (color) {
         case RED: // Не Color.RED !
            System.out.println("RED");
            break;
         case BLUE:
            System.out.println("BLUE");
            break;
         case GREEN:
            System.out.println("GREEN");
      }
   }
}
// Объявление перечисления
enum Color { RED, BLUE, GREEN }
```

Начиная с Java 7, помимо целого числа, символа или перечислимого типа, можно использовать и строки. Обратите внимание, что при сравнении регистр символов имеет значение:

```
String color = "RED";
switch (color) {
   case "red":
      System.out.println("red");
      break;
```

```
    case "RED":
        System.out.println("RED"); // RED
        break;
    case "BLUE":
        System.out.println("BLUE");
        break;
    case "GREEN":
        System.out.println("GREEN");
}
```

3.9. Цикл *for*

Операторы циклов позволяют выполнить одни и те же инструкции многократно. Предположим, нужно вывести все числа от `1` до `100` по одному на строке. Обычным способом пришлось бы писать 100 строк кода:

```
System.out.println(1);
System.out.println(2);
// ...
System.out.println(100);
```

С помощью циклов то же действие можно выполнить одной строкой кода:

```
for (int i = 1; i < 101; i++) System.out.println(i);
```

Цикл `for` используется для выполнения выражений определенное число раз. Цикл имеет следующий формат:

```
for (<Начальное значение>; <Условие>; <Приращение>) {
    <Инструкции>
}
```

Параметры имеют следующие значения:

- `<Начальное значение>` — присваивает переменной-счетчику начальное значение;
- `<Условие>` — содержит логическое выражение. Пока логическое выражение возвращает значение `true`, выполняются инструкции внутри цикла;
- `<Приращение>` — задает изменение переменной-счетчика на каждой итерации.

Последовательность работы цикла `for`:

1. Переменной-счетчику присваивается начальное значение.
2. Проверяется условие, и если оно истинно, то выполняются выражения внутри цикла, а в противном случае выполнение цикла завершается.
3. Переменная-счетчик изменяется на величину, указанную в параметре `<Приращение>`.
4. Переход к п. 2.

Переменная-счетчик может быть объявлена как вне цикла `for`, так и в параметре `<Начальное значение>`. Если переменная объявлена в параметре, то она будет видна

только внутри цикла. Кроме того, допускается объявить переменную вне цикла и сразу присвоить ей начальное значение. В этом случае параметр <Начальное значение> можно оставить пустым:

```
int i; // Объявление вне цикла
for (i = 1; i <= 20; i++) {
   System.out.print (i + " ");
}
System.out.println();
// Переменная i видна вне цикла
System.out.println(i); // 21
// Объявление внутри цикла
for (int j = 1; j <= 20; j++) {
   System.out.print(j + " ");
}
System.out.println();
// Переменная j НЕ видна вне цикла
// System.out.println(j); // Ошибка!
int k = 1; // Инициализация вне цикла
for (; k <= 20; k++) {
   System.out.print(k + " ");
}
```

Цикл выполняется до тех пор, пока <Условие> не вернет `false`. Если это не произойдет, то цикл станет *бесконечным*. Логическое выражение, указанное в параметре <Условие>, вычисляется на каждой итерации. Поэтому, если внутри логического выражения производятся какие-либо вычисления, и значение не изменяется внутри цикла, то вычисление следует вынести в параметр <Начальное значение>. В этом случае вычисление указывается после присваивания значения переменной-счетчику через запятую:

```
for (int i=1, j=10+30; i <= j; i++) {
   System.out.print(i + " ");
}
```

Начиная с Java 10, вместо типа переменной-счетчика можно указать слово `var` — тогда тип будет определен автоматически по контексту. При этом использовать оператор «запятая» нельзя:

```
for (var i = 1; i <= 10; i++) {
   System.out.print(i + " ");
} // 1 2 3 4 5 6 7 8 9 10
```

Выражение, указанное в параметре <Приращение>, может не только увеличивать значение переменной-счетчика, но и уменьшать его. Кроме того, значение может изменяться на любую величину:

```
// Выводим числа от 100 до 1
for (int i = 100; i > 0; i--) {
   System.out.println(i);
}
```

```
// Выводим четные числа от 2 до 100
for (int j = 2; j <= 100; j += 2) {
   System.out.println(j);
}
```

Если переменная-счетчик изменяется внутри цикла, то выражение в параметре <Приращение> можно вообще не указывать:

```
for (int i = 1; i <= 10; ) {
   System.out.println(i);
   i++; // Приращение
}
```

Все параметры цикла `for` не являются обязательными. Но, хотя параметры можно не указывать, точки с запятой обязательно должны присутствовать. Если все параметры не указаны, то цикл окажется бесконечным. Чтобы выйти из бесконечного цикла, следует использовать оператор `break`:

```
int i = 1;              // <Начальное значение>
for ( ; ; ) {           // Бесконечный цикл
   if (i <= 10) {       // <Условие>
      System.out.println(i);
      i++;              // <Приращение>
   }
   else {
      break;            // Выходим из цикла
   }
}
```

3.10. Цикл *for each*

В языке Java существует еще один вариант цикла `for`, который позволяет перебирать элементы массива или любого другого набора данных. Его формат:

```
for (<Объявление переменной>: <Массив или набор>) {
   <Инструкции>
}
```

Пример вывода всех элементов массива:

```
int[] arr = { 1, 2, 3, 4, 5 }; // Массив
for (int v: arr) {
   System.out.println(v);
}
```

На каждой итерации цикла переменной `v` присваивается один элемент массива `arr`. Цикл завершится, когда будут перебраны все элементы массива.

Пример использования слова `var` вместо явного типа данных переменной:

```
int[] arr = { 1, 2, 3, 4, 5 }; // Массив
for (var v: arr) {
   System.out.println(v);
}
```

3.11. Цикл *while*

Выполнение инструкций в цикле `while` продолжается до тех пор, пока логическое выражение истинно. Цикл имеет следующий формат:

```
<Начальное значение>
while (<Условие>) {
   <Инструкции>
   <Приращение>
}
```

Последовательность работы цикла `while`:

1. Переменной-счетчику присваивается начальное значение.
2. Проверяется условие, и если оно истинно, то выполняются инструкции внутри цикла, иначе выполнение цикла завершается.
3. Переменная-счетчик изменяется на величину, указанную в параметре `<Приращение>`.
4. Переход к п. 2.

Выведем все числа от `1` до `100`, используя цикл `while`:

```
int i = 1;                    // <Начальное значение>
while (i <= 100) {            // <Условие>
   System.out.println(i);     // <Инструкции>
   i++;                       // <Приращение>
}
```

> ***Внимание!***
>
> Если `<Приращение>` не указано, то цикл будет бесконечным.

3.12. Цикл *do...while*

Выполнение инструкций в цикле `do...while` продолжается до тех пор, пока логическое выражение истинно. В отличие от цикла `while`, условие проверяется не в начале цикла, а в конце. По этой причине инструкции внутри цикла `do...while` выполнятся минимум один раз. Цикл имеет следующий формат:

```
<Начальное значение>
do {
   <Инструкции>
   <Приращение>
} while (<Условие>);
```

Последовательность работы цикла `do...while`:

1. Переменной-счетчику присваивается начальное значение.
2. Выполняются инструкции внутри цикла.

3. Переменная-счетчик изменяется на величину, указанную в параметре <Приращение>.
4. Проверяется условие, и если оно истинно, то происходит переход к п. 2, а если нет — выполнение цикла завершается.

Выведем все числа от 1 до 100, используя цикл do...while:

```
int i = 1;                    // <Начальное значение>
do {
   System.out.println(i);     // <Инструкции>
   i++;                       // <Приращение>
} while (i <= 100);           // <Условие>
```

> ***Внимание!***
>
> Если <Приращение> не указано, то цикл будет бесконечным.

3.13. Оператор *continue*: переход на следующую итерацию цикла

Оператор continue позволяет перейти к следующей итерации цикла до завершения выполнения всех инструкций внутри цикла. В качестве примера выведем все числа от 1 до 100, кроме чисел от 5 до 10 включительно:

```
for (int i = 1; i <= 100; i++) {
   if (i > 4 && i < 11) continue;
   System.out.println(i);
}
```

3.14. Оператор *break*: прерывание цикла

Оператор break позволяет прервать выполнение цикла досрочно. Для примера выведем все числа от 1 до 100 еще одним способом:

```
int i = 1;
while (true) {
   if (i > 100) break;
   System.out.println(i);
   i++;
}
```

Здесь мы в условии указали значение true. В этом случае инструкции внутри цикла будут выполняться бесконечно. Однако использование оператора break прерывает его выполнение, как только 100 строк уже напечатано.

> ***Внимание!***
>
> Оператор break прерывает выполнение цикла, а не программы, т. е. далее будет выполнена инструкция, следующая сразу за циклом.

Бесконечный цикл совместно с оператором `break` удобно использовать для получения от пользователя не известного заранее количества данных. В качестве примера просуммируем не известное заранее количество целых чисел (листинг 3.8).

Листинг 3.8. Суммирование не известного заранее количества чисел

```
import java.util.Scanner;

public class MyClass {
   public static void main(String[] args) {
      int x = 0, result = 0;
      Scanner in = new Scanner(System.in);
      System.out.println(
         "Введите число 0 для получения результата\n");
      for ( ; ; ) {
         System.out.print("Введите число: ");
         x = in.nextInt();
         if (x == 0) break;
         result += x;
      }
      System.out.println("Сумма чисел равна: " + result);
   }
}
```

Оператор `break` позволяет выйти сразу из нескольких вложенных циклов. Для этого применяется следующий формат оператора `break`:

```
break <Метка>
```

Значение в параметре `<Метка>` должно быть допустимым идентификатором. Место в программе помечается одноименной меткой, после которой указывается двоеточие. Метку можно указать перед циклом, оператором `if` или другим блоком. После вызова оператора `break` управление передается инструкции, расположенной сразу после закрывающей фигурной скобки блока, к которому привязана метка:

```
BLOCK1:
while (true) {
   System.out.println("Начало цикла 1");
   BLOCK2:
   for (int i = 0; i < 5; i++) {
      System.out.println("---- Начало цикла 2");
      if (i == 1) {
         System.out.println("---- break");
         break BLOCK1; // Выход из блоков
      }
      System.out.println("---- Внутри цикла 2");
   }
```

```
    // Инструкция не выполняется
    System.out.println("После цикла 2");
}
System.out.println("После цикла 1");
```

Последовательность выполнения будет выглядеть следующим образом:

```
Начало цикла 1
---- Начало цикла 2
---- Внутри цикла 2
---- Начало цикла 2
---- break
После цикла 1
```

ГЛАВА 4

Числа

В языке Java почти все элементарные типы данных: `byte`, `short`, `int`, `long`, `float` и `double` — являются числовыми. Исключением является тип `boolean`, предназначенный для хранения только логических значений, которые нельзя преобразовать в числовые. Тип `char` содержит код символа в кодировке UTF-16, который может быть автоматически без потерь преобразован в тип `int`. Поэтому значения этого типа можно использовать в одном выражении вместе с числовыми значениями:

```
char ch = 'w';                // 119
System.out.println(ch + 10); // 129
```

Для хранения целых чисел предназначены типы `byte`, `short`, `int` и `long`. Выведем диапазоны значений этих типов и размер в байтах:

```
System.out.println(Byte.MIN_VALUE);    // -128
System.out.println(Byte.MAX_VALUE);    // 127
System.out.println(Byte.BYTES);        // 1
System.out.println(Short.MIN_VALUE);   // -32768
System.out.println(Short.MAX_VALUE);   // 32767
System.out.println(Short.BYTES);       // 2
System.out.println(Integer.MIN_VALUE); // -2147483648
System.out.println(Integer.MAX_VALUE); // 2147483647
System.out.println(Integer.BYTES);     // 4
System.out.println(Long.MIN_VALUE);    // -9223372036854775808
System.out.println(Long.MAX_VALUE);    // 9223372036854775807
System.out.println(Long.BYTES);        // 8
System.out.println((int)Character.MIN_VALUE);    // 0
System.out.println((int)Character.MAX_VALUE);    // 65535
System.out.println(Character.BYTES);             // 2
```

Обратите внимание на то, что типы данных в языке Java не зависят от разрядности операционной системы. Код на языке Java является машинонезависимым.

Как уже отмечалось в *главе 2*, целочисленное значение задается в десятичной, двоичной, восьмеричной или шестнадцатеричной форме. Двоичные числа начинаются с комбинации символов `0b` (или `0B`) и могут содержать числа `0` или `1`. Восьмеричные

числа начинаются с нуля и содержат цифры от 0 до 7. Шестнадцатеричные числа начинаются с комбинации символов 0x (или 0X) и могут содержать числа от 0 до 9 и буквы от A до F (регистр букв не имеет значения). Двоичные, восьмеричные и шестнадцатеричные значения преобразуются в десятичное значение:

```
int x, y, z, k;
x = 119;         // Десятичное значение
y = 0167;        // Восьмеричное значение
z = 0x77;        // Шестнадцатеричное значение
k = 0B01110111; // Двоичное значение
System.out.println(x + " " + y + " " + z + " " + k); // 119 119 119 119
```

Начиная с Java 7, в составе числового литерала можно использовать символ подчеркивания:

```
int x = 1_000_000_000;
System.out.println(x); // 1000000000
```

Согласитесь, что так нагляднее виден порядок числа. Символ подчеркивания можно использовать не только для типа int, но и для других числовых типов.

По умолчанию целочисленные константы, которые мы вводим в программе, имеют тип int. Вводимое значение автоматически приводится к типам byte и short, если оно входит в диапазон значений этих типов. Если значение не входит в диапазон, то компилятор выведет сообщение об ошибке:

```
byte x;  // Диапазон от -128 до 127
x = 100; // OK
x = 300; // Ошибка
```

Диапазон значений у типа long гораздо больше, чем диапазон значений у типа int. Чтобы задать значение для типа long, необходимо после константы указать букву L (или l):

```
long x;
x = 2147483648L; // OK
x = 2147483648;  // Ошибка. int до 2 147 483 647
```

Для хранения вещественных чисел предназначены типы float и double. Выведем диапазоны значений этих типов и их размеры в байтах:

```
System.out.println(Float.MIN_VALUE);     // 1.4E-45
System.out.println(Float.MAX_VALUE);     // 3.4028235E38
System.out.println(Float.BYTES);         // 4
System.out.println(Double.MIN_VALUE);    // 4.9E-324
System.out.println(Double.MAX_VALUE);    // 1.7976931348623157E308
System.out.println(Double.BYTES);        // 8
```

Вещественное число может содержать точку и (или) экспоненту, начинающуюся с буквы E (регистр не имеет значения):

```
double x, y, z, k;
x = 20.0;
```

```
y = 12.1e5;
z = .123;    // Эквивалентно z = 0.123;
k = 47.E-5;
```

По умолчанию вещественные константы имеют тип `double`. Чтобы задать значение для переменной, имеющей тип `float`, необходимо после константы указать букву `F` (или `f`):

```
float x, y;
x = 20.0F;
y = 12.1e5f;
```

При выполнении операций над вещественными числами следует учитывать ограничения точности вычислений. Например, результат следующей инструкции может показаться странным:

```
System.out.println(0.3 - 0.1 - 0.1 - 0.1); // -2.7755575615628914E-17
```

Ожидаемым был бы результат `0.0`, но, как видно из примера, мы получили совсем другой результат (`–2.7755575615628914E–17`). Он очень близок к нулю, но не равен нулю. Учитывайте это при указании вещественных чисел в качестве значения счетчика внутри цикла, т. к. попытка проверить это значение на равенство может привести к бесконечному циклу.

Если необходимо производить операции с фиксированной точностью, то следует использовать класс `BigDecimal`:

```
// import java.math.BigDecimal;
BigDecimal x = new BigDecimal("0.3");  // Строка, а не тип double!
BigDecimal y = new BigDecimal("0.1");
x = x.subtract(y);                     // Вычитание x = x - y
x = x.subtract(y);
x = x.subtract(y);
System.out.println(x);                 // 0.0
```

Если в выражении используются числовые переменные, имеющие разный тип данных, то:

- если один из операндов типа `double`, то второй операнд преобразуется в тип `double`;
- в противном случае, если один из операндов типа `float`, то второй операнд преобразуется в тип `float`;
- в противном случае, если один из операндов типа `long`, то второй операнд преобразуется в тип `long`;
- в противном случае оба операнда преобразуются в тип `int`.

Последнее правило означает, что результатом операций с типами `byte` и `short` будет как минимум тип `int`:

```
byte y1 = 1, y2 = 2;
y1 = y1 + y2;                  // Ошибка! Тип int
```

```
short z1 = 1, z2 = 2;
z1 = z1 + z2;                    // Ошибка! Тип int
```

Чтобы результат выполнения этих выражений сохранить в переменных, необходимо выполнить операцию приведения типов. Пример приведения типов:

```
byte y1 = 1, y2 = 2;
y1 = (byte)(y1 + y2);            // OK
short z1 = 1, z2 = 2;
z1 = (short)(z1 + z2);           // OK
```

Преобразование без потерь данных происходит в следующих случаях:

- тип `byte` без потерь преобразуется в типы `short`, `int`, `long`, `double`;
- тип `short` — в типы `int`, `long`, `double`;
- тип `int` — в типы `long`, `double`;
- тип `char` — в типы `int`, `long`, `double`.

4.1. Математические константы

Класс `Math` содержит следующие стандартные константы:

- `PI` — возвращает число ПИ:

  ```
  public static final double PI
  ```

  Пример:

  ```
  System.out.println(Math.PI);   // 3.141592653589793
  ```

- `E` — возвращает значение константы `e`:

  ```
  public static final double E
  ```

  Пример:

  ```
  System.out.println(Math.E);    // 2.718281828459045
  ```

4.2. Основные методы для работы с числами

Для работы с числами предназначены следующие методы из класса `Math`:

- `abs()` — возвращает абсолютное значение. Форматы метода:

  ```
  public static int abs(int a)
  public static long abs(long a)
  public static float abs(float a)
  public static double abs(double a)
  ```

  Пример:

  ```
  System.out.println(Math.abs(-1));        // 1
  ```

- `pow()` — возводит число `a` в степень `b`. Формат метода:

  ```
  public static double pow(double a, double b)
  ```

Пример:

```
System.out.println(Math.pow(10, 2));     // 100.0
System.out.println(Math.pow(3.0, 3.0)); // 27.0
```

- `sqrt()` — **квадратный корень. Формат метода:**

```
public static double sqrt(double a)
```

Пример:

```
System.out.println(Math.sqrt(100.0));    // 10.0
```

- `exp()` — **экспонента. Формат метода:**

```
public static double exp(double a)
```

- `log()` — **натуральный логарифм. Формат метода:**

```
public static double log(double a)
```

- `log10()` — **десятичный логарифм. Формат метода:**

```
public static double log10(double a)
```

- `IEEEremainder()` — **остаток от деления согласно стандарту IEEE 754. Обратите внимание на то, что результат метода в некоторых случаях будет отличаться от результата оператора `%`, т. к. используется другой принцип вычисления. Формат метода:**

```
public static double IEEEremainder(double f1, double f2)
```

Пример:

```
System.out.println(Math.IEEEremainder(13.5, 2.0)); // -0.5
System.out.println(
              13.5 - 2.0 * Math.round(13.5/2.0)); // -0.5
// Оператор % дает другой результат
System.out.println(13.5 % 2.0);                   //  1.5
```

- `max()` — **максимальное значение. Форматы метода:**

```
public static int max(int a, int b)
public static long max(long a, long b)
public static float max(float a, float b)
public static double max(double a, double b)
```

Пример:

```
System.out.println(Math.max(10, 3));  // 10
```

- `min()` — **минимальное значение. Форматы метода:**

```
public static int min(int a, int b)
public static long min(long a, long b)
public static float min(float a, float b)
public static double min(double a, double b)
```

Пример:

```
System.out.println(Math.min(10, 3));  // 3
```

4.3. Округление чисел

Для округления чисел предназначены следующие методы из класса `Math`:

- `ceil()` — возвращает значение, округленное до ближайшего большего значения. Формат метода:

  ```
  public static double ceil(double a)
  ```

  Пример:

  ```
  System.out.println(Math.ceil(1.49));  // 2.0
  System.out.println(Math.ceil(1.5));   // 2.0
  System.out.println(Math.ceil(1.51));  // 2.0
  ```

- `floor()` — значение, округленное до ближайшего меньшего значения. Формат метода:

  ```
  public static double floor(double a)
  ```

  Пример:

  ```
  System.out.println(Math.floor(1.49));  // 1.0
  System.out.println(Math.floor(1.5));   // 1.0
  System.out.println(Math.floor(1.51));  // 1.0
  ```

- `round()` — возвращает число, округленное до ближайшего меньшего целого — для чисел с дробной частью меньше `0.5`, или значение, округленное до ближайшего большего целого — для чисел с дробной частью больше или равной `0.5`. Форматы метода:

  ```
  public static int round(float a)
  public static long round(double a)
  ```

  Пример:

  ```
  System.out.println(Math.round(1.49));  // 1
  System.out.println(Math.round(1.5));   // 2
  System.out.println(Math.round(1.51));  // 2
  ```

4.4. Тригонометрические методы

В языке Java доступны следующие основные тригонометрические методы:

- `sin()`, `cos()`, `tan()` — стандартные тригонометрические функции (синус, косинус, тангенс). Угол задается в радианах. Форматы методов:

  ```
  public static double sin(double a)
  public static double cos(double a)
  public static double tan(double a)
  ```

Пример:

```
System.out.println(Math.sin(Math.toRadians(90.0))); // 1.0
```

- `asin()`, `acos()`, `atan()` — обратные тригонометрические функции (арксинус, арккосинус, арктангенс). Форматы методов:

```
public static double asin(double a)
public static double acos(double a)
public static double atan(double a)
```

- `toRadians()` — преобразует градусы в радианы. Формат метода:

```
public static double toRadians(double a)
```

Пример:

```
System.out.println(Math.toRadians(180.0)); // 3.141592653589793
```

- `toDegrees()` — преобразует радианы в градусы. Формат метода:

```
public static double toDegrees(double a)
```

Пример:

```
System.out.println(Math.toDegrees(Math.PI)); // 180.0
```

4.5. Преобразование строки в число

Числа, вводимые пользователем, например, в окне консоли, представлены в виде строки. Чтобы в дальнейшем использовать эти числа, необходимо выполнить преобразование строки в число. Для этого в языке Java предназначены следующие методы:

- `parseByte()` — возвращает целое число, имеющее тип `byte`. Форматы метода:

```
public static byte parseByte(String s)
public static byte parseByte(String s, int r)
```

Если второй параметр не указан, то используется десятичная система:

```
byte x = 0;
x = Byte.parseByte("10");
System.out.println(x);    // 10
```

Во втором параметре можно указать систему счисления:

```
System.out.println(Byte.parseByte("119", 10));     // 119
System.out.println(Byte.parseByte("167", 8));      // 119
System.out.println(Byte.parseByte("77", 16));      // 119
System.out.println(Byte.parseByte("01110111", 2)); // 119
```

- `parseShort()` — возвращает целое число, имеющее тип `short`. Форматы метода:

```
public static short parseShort(String s)
public static short parseShort(String s, int r)
```

Пример:

```
System.out.println(Short.parseShort("119"));          // 119
System.out.println(Short.parseShort("119", 10));      // 119
System.out.println(Short.parseShort("167", 8));       // 119
System.out.println(Short.parseShort("77", 16));       // 119
System.out.println(Short.parseShort("01110111", 2)); // 119
```

- `parseInt()` — возвращает целое число, имеющее тип `int`. Форматы метода:

```
public static int parseInt(String s)
public static int parseInt(String s, int r)
```

Пример:

```
System.out.println(Integer.parseInt("119"));          // 119
System.out.println(Integer.parseInt("119", 10));      // 119
System.out.println(Integer.parseInt("167", 8));       // 119
System.out.println(Integer.parseInt("77", 16));       // 119
System.out.println(Integer.parseInt("01110111", 2)); // 119
```

- `parseLong()` — возвращает целое число, имеющее тип `long`. Форматы метода:

```
public static long parseLong(String s)
public static long parseLong(String s, int r)
```

Пример:

```
System.out.println(Long.parseLong("119"));          // 119
System.out.println(Long.parseLong("119", 10));      // 119
System.out.println(Long.parseLong("167", 8));       // 119
System.out.println(Long.parseLong("77", 16));       // 119
System.out.println(Long.parseLong("01110111", 2)); // 119
```

- `parseFloat()` — возвращает вещественное число, имеющее тип `float`. Формат метода:

```
public static float parseFloat(String s)
```

Пример:

```
System.out.println(Float.parseFloat("119.5"));      // 119.5
System.out.println(Float.parseFloat("119.5e2"));    // 11950.0
```

- `parseDouble()` — возвращает вещественное число, имеющее тип `double`. Формат метода:

```
public static double parseDouble(String s)
```

Пример:

```
System.out.println(Double.parseDouble("119.5"));    // 119.5
System.out.println(Double.parseDouble("119.5e2"));  // 11950.0
```

Если строку невозможно преобразовать в число, все эти методы генерируют исключение `NumberFormatException`, которое приводит к остановке работы программы и выводу сообщения об ошибке. Чтобы избежать этого, необходимо использовать

инструкцию try...catch. Если внутри блока try возникнет исключение, то управление передается блоку catch, внутри которого можно обработать ошибку:

```
int x = 0;
try {
   x = Integer.parseInt("Строка");
   System.out.println(x); // Инструкция не будет выполнена!
}
catch (NumberFormatException e) {
   // Обработка ошибки
   System.out.println("Не удалось преобразовать");
}
```

4.6. Преобразование числа в строку

Чтобы преобразовать число в строку, достаточно выполнить операцию конкатенации:

```
int x = 100;
double y = 1.8;
String s = "x = " + x + " y = " + y;
System.out.println(s);                    // x = 100 y = 1.8
```

Оператор + перегружен в классе String и применяется для соединения строк. Если один из операндов не является строкой, то производится его автоматическое преобразование в строку. Однако учитывайте, что конкатенация выполняется очень медленно. Способ простой, но не эффективный. Вместо конкатенации лучше использовать метод valueOf() из класса String:

```
String s = String.valueOf(10);
System.out.println(s);          // 10
```

Для преобразования числа в строку можно также воспользоваться следующими методами:

- toString() — этот метод есть у всех классов в языке Java:

```
byte a = 1;        short b = 20;
int x = 100;       long y = 200L;
float k = 10.3f;   double n = 1.8;
System.out.println(Byte.toString(a));    // 1
System.out.println(Short.toString(b));   // 20
System.out.println(Integer.toString(x)); // 100
System.out.println(Long.toString(y));    // 200
System.out.println(Float.toString(k));   // 10.3
System.out.println(Double.toString(n));  // 1.8
```

  В классах Integer и Long есть также метод toString() с двумя параметрами. Форматы метода:

```
public static String toString(int i, int r)
public static String toString(long i, int r)
```

Во втором параметре можно указать систему счисления:

```
System.out.println(Integer.toString(119, 10)); // 119
System.out.println(Integer.toString(119, 2));  // 1110111
System.out.println(Integer.toString(-119, 2)); // -1110111
System.out.println(Integer.toString(119, 8));  // 167
System.out.println(Integer.toString(119, 16)); // 77
System.out.println(Long.toString(119L, 2));    // 1110111
```

- `toBinaryString()` — возвращает строковое представление целого числа в двоичной системе. Форматы метода:

```
public static String toBinaryString(int i)
public static String toBinaryString(long i)
```

Пример:

```
System.out.println(Integer.toBinaryString(119));  // 1110111
System.out.println(Long.toBinaryString(119L));    // 1110111
```

Следует учитывать, что при использовании отрицательных чисел результат метода `toBinaryString()` будет отличаться от результата метода `toString()`:

```
System.out.println(Integer.toString(-119, 2));
// -1110111
System.out.println(Integer.toBinaryString(-119));
// 11111111111111111111111110001001
```

- `toOctalString()` — возвращает строковое представление целого числа в восьмеричной системе. Форматы метода:

```
public static String toOctalString(int i)
public static String toOctalString(long i)
```

Пример:

```
System.out.println(Integer.toOctalString(119));   // 167
System.out.println(Long.toOctalString(119L));     // 167
```

- `toHexString()` — возвращает строковое представление целого числа в шестнадцатеричной системе. Форматы метода:

```
public static String toHexString(int i)
public static String toHexString(long i)
```

Пример:

```
System.out.println(Integer.toHexString(119)); // 77
System.out.println(Long.toHexString(119L));   // 77
```

Для форматированного вывода чисел предназначен метод `printf()` из класса `PrintStream`. Экземпляр этого класса доступен через объект `System.out`. Форматы метода:

```
public PrintStream printf(String format, Object... args)
public PrintStream printf(Locale locale, String format, Object... args)
```

В параметре `format` указывается строка специального формата, внутри которой с помощью спецификаторов задаются правила форматирования. Какие спецификаторы используются, мы рассмотрим немного позже при изучении форматирования строк. В параметре `args` через запятую указываются различные значения. Параметр `locale` позволяет задать *локаль*. Настройки локали для разных стран различаются — например, в одной стране принято десятичный разделитель вещественных чисел выводить в виде точки, в другой — в виде запятой. В первом формате метода используются настройки локали по умолчанию. Прежде чем настраивать локаль, необходимо импортировать класс `Locale` с помощью инструкции:

```
import java.util.Locale;
```

Пример:

```
System.out.println(10.5125484);              // 10.5125484
System.out.printf("%.2f\n", 10.5125484);     // 10,51
System.out.printf(
   new Locale("ru"), "%.2f\n", 10.5125484); // 10,51
System.out.printf(
   new Locale("en"), "%.2f\n", 10.5125484); // 10.51
```

Если необходимо не выводить, а сохранить результат в виде строки, то следует воспользоваться методом `format()` из класса `String`. Форматы метода:

```
public static String format(String format, Object... args)
public static String format(Locale locale, String format, Object... args)
```

Смысл всех параметров полностью соответствует параметрам метода `printf()`:

```
String s = String.valueOf(10.5125484);
System.out.println(s);                        // 10.5125484
s = String.format("%.2f", 10.5125484);
System.out.println(s);                        // 10,51
s = String.format(
    new Locale("ru"), "%.2f", 10.5125484);
System.out.println(s);                        // 10,51
s = String.format(
    new Locale("en"), "%.2f", 10.5125484);
System.out.println(s);                        // 10.51
```

4.7. Генерация псевдослучайных чисел

Для генерации случайных чисел в языке Java предназначен метод `random()` из класса `Math`. Метод возвращает псевдослучайное число, которое больше или равно `0.0` и меньше `1.0`. Формат метода:

```
public static double random()
```

Пример генерации псевдослучайных чисел:

```
System.out.println(Math.random()); // 0.9716780632629719
System.out.println(Math.random()); // 0.32302953237853427
System.out.println(Math.random()); // 0.779069650193378
```

Чтобы получить псевдослучайное целое число от `0` до `9` включительно, нужно возвращаемое методом значение умножить на `10`, а затем выполнить приведение к типу `int`:

```
System.out.println((int)(Math.random() * 10)); // 9
System.out.println((int)(Math.random() * 10)); // 0
System.out.println((int)(Math.random() * 10)); // 1
```

Для генерации псевдослучайных чисел можно также воспользоваться классом `Random`. Прежде чем применить этот класс, его необходимо импортировать с помощью инструкции:

```
import java.util.Random;
```

Класс содержит два конструктора:

```
Random()
Random(long a)
```

Первый конструктор (без параметра) создает объект с настройками по умолчанию:

```
Random r1 = new Random();
Random r2 = new Random();
System.out.println(r1.nextDouble()); // 0.21271435020874052
System.out.println(r2.nextDouble()); // 0.6758420841211599
```

Второй конструктор позволяет указать начальное значение. Если это значение будет одинаковым у нескольких объектов, то мы получим одно и то же псевдослучайное число:

```
Random r1 = new Random(1);
Random r2 = new Random(1);
System.out.println(r1.nextDouble()); // 0.7308781907032909
System.out.println(r2.nextDouble()); // 0.7308781907032909
```

Класс `Random` содержит следующие основные методы:

- `setSeed()` — настраивает генератор псевдослучайных чисел на новую последовательность. В качестве параметра обычно указывается время — число, возвращаемое методом `getTime()` из класса `Date` (класс `Date` нужно предварительно подключить с помощью инструкции `import java.util.Date;`). Формат метода:

  ```
  public void setSeed(long a)
  ```

  Пример:

  ```
  Random r = new Random();
  r.setSeed(1);
  System.out.println(r.nextDouble()); // 0.7308781907032909
  ```

```
r.setSeed(1);
System.out.println(r.nextDouble()); // 0.7308781907032909
r.setSeed((new Date()).getTime());
System.out.println(r.nextDouble()); // 0.4257746867955643
```

- `nextBoolean()` — возвращает псевдослучайное логическое значение. Формат метода:

```
public boolean nextBoolean()
```

Пример:

```
Random r = new Random();
System.out.println(r.nextBoolean()); // true
System.out.println(r.nextBoolean()); // false
```

- `nextInt()` — возвращает псевдослучайное целое число. Форматы метода:

```
public int nextInt()
public int nextInt(int n)
```

Если параметр не указан, то возвращается значение в диапазоне для типа `int`:

```
Random r = new Random();
System.out.println(r.nextInt()); // -457749303
System.out.println(r.nextInt()); // 288354376
```

Если параметр указан, то число будет больше или равно `0` и меньше `n`:

```
Random r = new Random();
System.out.println(r.nextInt(5)); // 4
System.out.println(r.nextInt(5)); // 0
```

- `nextLong()` — возвращает значение в диапазоне для типа `long`. Формат метода:

```
public long nextLong()
```

Пример:

```
Random r = new Random();
System.out.println(r.nextLong()); // 2078068389855764962
System.out.println(r.nextLong()); // -4497372885969498850
```

- `nextFloat()` — возвращает псевдослучайное число, которое больше или равно `0.0` и меньше `1.0`. Формат метода:

```
public float nextFloat()
```

Пример:

```
Random r = new Random();
System.out.println(r.nextFloat()); // 0.2868402
System.out.println(r.nextFloat()); // 0.83520055
```

- `nextDouble()` — возвращает псевдослучайное число, которое больше или равно `0.0` и меньше `1.0`. Формат метода:

```
public double nextDouble()
```

Пример:

```
Random r = new Random();
System.out.println(r.nextDouble()); // 0.6145411976182029
System.out.println(r.nextDouble()); // 0.3770044403409455
```

- `nextBytes()` — **заполняет массив псевдослучайными числами в диапазоне для типа** `byte`. **Формат метода:**

```
public void nextBytes(byte[] a)
```

Пример:

```
byte[] arr = new byte[10]; // Массив
Random r = new Random();
r.nextBytes(arr);
for (byte x: arr) {
   System.out.print(x + " ");
   // 67 51 -73 72 -6 -69 -109 -126 73 -121
}
```

В качестве примера создадим генератор паролей произвольной длины (листинг 4.1). Для этого добавляем в массив `arr` **все разрешенные символы, а далее в цикле получаем случайный элемент из массива и добавляем его в набор символов, реализованный классом** `StringBuilder`. **После чего преобразуем объект класса** `StringBuilder` **в строку (с помощью метода** `toString()`**) и возвращаем ее.**

Листинг 4.1. Генератор паролей

```
import java.util.Random;

public class MyClass {
   public static void main(String[] args) {
      System.out.println(passwGen(6)); // qbycyG
      System.out.println(passwGen(6)); // VPfU5c
      System.out.println(passwGen(8)); // pzayi9JU
   }

   public static String passwGen(int count_char) {
      if (count_char < 1) return "";
      StringBuilder s = new StringBuilder();
      Random r = new Random();
      char[] arr = {'a','b','c','d','e','f','g','h','i','j',
         'k','l','m','n','p','q','r','s','t','u','v','w','x','y','z',
         'A','B','C','D','E','F','G','H','I','J','K','L',
         'M','N','P','Q','R','S','T','U','V','W',
         'X','Y','Z','1','2','3','4','5','6','7','8','9','0'};
      for (int i = 0; i < count_char; i++) {
         s.append(arr[r.nextInt(arr.length)]);
      }
```

```
        return s.toString();
    }
}
```

4.8. Бесконечность и значение *NaN*

Целочисленное деление на `0` приведет к ошибке при выполнении программы, а вот с вещественными числами все обстоит несколько иначе. Деление вещественного числа на `0.0` приведет к значению плюс или минус `Infinity` (бесконечность), а деление вещественного числа `0.0` на `0.0` — к значению `NaN` (нет числа):

```
System.out.println( 10.0 / 0 );              // Infinity
System.out.println( -10.0 / 0 );             // -Infinity
System.out.println( 0.0 / 0 );               // NaN
```

В классах `Float` и `Double` для этих значений существуют константы `POSITIVE_INFINITY`, `NEGATIVE_INFINITY` и `NaN`:

```
System.out.println(Double.POSITIVE_INFINITY); // Infinity
System.out.println(Double.NEGATIVE_INFINITY); // -Infinity
System.out.println(Double.NaN);               // NaN
```

Для проверки соответствия этим значениям нельзя применять логические операторы. Чтобы проверить значения, надо воспользоваться следующими методами из классов `Float` и `Double`:

- `isInfinite()` — возвращает `true`, если значение равно плюс или минус бесконечность, и `false` — в противном случае. Форматы метода:

  ```
  public static boolean isInfinite(float f)
  public static boolean isInfinite(double d)
  ```

  Пример:

  ```
  System.out.println(Float.isInfinite(10.0f / 0.0f));   // true
  System.out.println(Double.isInfinite(10.0 / 1.0));    // false
  System.out.println(Double.isInfinite(10.0 / 0.0));    // true
  System.out.println(Double.isInfinite(-10.0 / 0.0));   // true
  System.out.println(Double.isInfinite(0.0 / 0.0));     // false
  ```

- `isFinite()` — возвращает `true`, если значение не равно плюс или минус бесконечность или значению `NaN`, и `false` — в противном случае. Форматы метода:

  ```
  public static boolean isFinite(float f)
  public static boolean isFinite(double d)
  ```

  Пример:

  ```
  System.out.println(Float.isFinite(10.0f / 1.0f)); // true
  System.out.println(Double.isFinite(10.0 / 1.0));  // true
  System.out.println(Double.isFinite(10.0 / 0.0));  // false
  System.out.println(Double.isFinite(-10.0 / 0.0)); // false
  System.out.println(Double.isFinite(0.0 / 0.0));   // false
  ```

- `isNaN()` — возвращает `true`, если значение равно `NaN`, и `false` — в противном случае. Форматы метода:

```
public static boolean isNaN(float f)
public static boolean isNaN(double d)
```

Пример:

```
System.out.println(Float.isNaN(0.0f / 0.0f));     // true
System.out.println(Double.isNaN(10.0 / 1.0));     // false
System.out.println(Double.isNaN(10.0 / 0.0));     // false
System.out.println(Double.isNaN(-10.0 / 0.0));    // false
System.out.println(Double.isNaN(0.0 / 0.0));      // true
```

ГЛАВА 5

Массивы

Массив — это нумерованный набор переменных одного типа. Переменная в массиве называется *элементом*, а ее позиция в массиве задается *индексом*. Обратите внимание на то, что нумерация элементов начинается с `0`, а не с `1`. Следовательно, индекс первого элемента будет равен `0`, а последнего — длина массива минус `1` (так, если массив состоит из `10` элементов, то последний элемент будет иметь индекс `9`). Попытка обратиться к элементу, индекс которого не существует в массиве, приведет к ошибке во время выполнения.

5.1. Объявление и инициализация массива

Объявить массив можно двумя способами:

```
<Тип>[] <Переменная>;
<Тип> <Переменная>[];
```

Пример:

```
int[] arr1;
double arr2[];
```

Как видно из примера, в первом способе квадратные скобки указываются сразу после типа данных. Это наиболее часто используемый способ объявления массива в языке Java, и в этой книге мы будем придерживаться именно его. Во втором способе квадратные скобки указываются после названия переменной.

Объявить массив — это всего лишь полдела. Далее необходимо выполнить *инициализацию* массива с помощью оператора `new` по следующей схеме:

```
<Переменная> = new <Тип>[<Количество элементов>];
```

Пример объявления и инициализации массива из `5` элементов типа `int`:

```
int[] arr;
arr = new int[5];
```

Можно объявление и инициализацию указать на одной строке:

```
int[] arr = new int[5];
```

В результате инициализации в динамической памяти компьютера будет создан массив из указанного количества элементов. Каждый элемент массива получит значение по умолчанию. Для числовых типов значение равно `0`, для типа `boolean` — `false`, для объектов — `null`:

```
// import java.util.Arrays;
int[] arr1 = new int[5];
boolean[] arr2 = new boolean[2];
String[] arr3 = new String[3];
System.out.println(Arrays.toString(arr1)); // [0, 0, 0, 0, 0]
System.out.println(Arrays.toString(arr2)); // [false, false]
System.out.println(Arrays.toString(arr3)); // [null, null, null]
```

Можно также создать массив нулевого размера, который полезен при возврате пустого массива из какого-либо метода. Пример создания массива нулевого размера:

```
int[] arr = new int[0];
System.out.println(Arrays.toString(arr)); // []
System.out.println(arr.length);           // 0
```

При объявлении элементам массива можно присвоить начальные значения. Для этого после объявления указывается оператор =, а далее значения через запятую внутри фигурных скобок. После закрывающей фигурной скобки обязательно ставится точка с запятой. Обратите внимание на то, что не нужно указывать количество элементов массива явным образом, — размер массива будет соответствовать количеству значений внутри фигурных скобок. Пример инициализации массива из трех элементов:

```
int[] arr = {10, 20, 30};
System.out.println(Arrays.toString(arr)); // [10, 20, 30]
System.out.println(arr.length);           // 3
```

Обратите внимание на то, что в этом случае использовать оператор `new` не нужно. Хотя можно и указать его следующим образом:

```
int[] arr = new int[] {10, 20, 30};
```

Такую форму инициализации удобно использовать для создания анонимных массивов. Например, мы хотим передать массив в какой-либо метод и не хотим для этого использовать переменную. В качестве примера создадим метод, в котором выведем все элементы массива по одному на строке. При вызове метода укажем *анонимный массив* (листинг 5.1).

Листинг 5.1. Передача анонимного массива в качестве параметра

```
public class MyClass {
   public static void main(String[] args) {
      int[] arr = {10, 20, 30};
      // Передаем обычный массив
      printArr(arr);
```

```
        // Передаем анонимный массив
        printArr(new int[] {40, 50, 60});
    }

    public static void printArr(int[] a) {
        for (int i: a) {
            System.out.println(i);
        }
    }
}
```

Анонимные массивы можно также использовать для повторной инициализации массива:

```
int[] arr = {10, 20, 30};
System.out.println(Arrays.toString(arr)); // [10, 20, 30]
// Повторная инициализация
arr = new int[] {40, 50, 60};
System.out.println(Arrays.toString(arr)); // [40, 50, 60]
```

Если вы программировали на других языках и работали с динамической памятью, то после этого примера у вас может возникнуть вопрос: а что будет с предыдущим массивом? Не произойдет ли утечка памяти? Нет, не произойдет. Управление динамической памятью полностью контролируется виртуальной машиной Java. Если на объект нет ссылок, то объект будет удален автоматически. Только не стоит рассчитывать, что это произойдет немедленно, прямо сейчас, — удаление объекта может быть осуществлено с задержкой.

Пример использования слова `var` вместо явного типа данных переменной:

```
var arr1 = new int[3];              // Тип int[]
var arr2 = new int[] {10, 20, 30}; // Тип int[]
// var arr3 = {40, 50, 60};         // Ошибка!
System.out.println(Arrays.toString(arr1)); // [0, 0, 0]
System.out.println(Arrays.toString(arr2)); // [10, 20, 30]
```

5.2. Определение размера массива

Количество элементов массива задается при инициализации и не может быть изменено позже. Впрочем, можно присвоить переменной ссылку на другой массив, имеющий другую длину:

```
int[] arr1 = {10, 20, 30};
int[] arr2 = {40, 50, 60, 70};
System.out.println(Arrays.toString(arr1)); // [10, 20, 30]
arr1 = arr2;
System.out.println(Arrays.toString(arr1)); // [40, 50, 60, 70]
```

В переменной сохраняется ссылка на массив, а не сам массив. Таким образом, длина массива, на который ссылается переменная, может динамически изменяться.

Чтобы получить длину массива, следует воспользоваться свойством `length`, которое указывается через оператор «точка» после имени переменной:

```
int[] arr = {1, 2, 3};
System.out.println(arr.length); // 3
```

Так как в переменной сохраняется лишь ссылка на массив, это обстоятельство следует учитывать при использовании оператора присваивания. Рассмотрим пример:

```
int[] arr1 = {10, 20, 30};
int[] arr2 = {40, 50, 60, 70};
arr1 = arr2; // Якобы создали копию массива
System.out.println(Arrays.toString(arr1)); // [40, 50, 60, 70]
System.out.println(Arrays.toString(arr2)); // [40, 50, 60, 70]
```

Теперь попробуем изменить первый элемент массива в переменной `arr1`:

```
arr1[0] = 33;
System.out.println(Arrays.toString(arr1)); // [33, 50, 60, 70]
System.out.println(Arrays.toString(arr2)); // [33, 50, 60, 70]
```

Как видно из примера, изменение значения в переменной `arr1` привело также к изменению значения в переменной `arr2`. Таким образом, обе переменные ссылаются на один и тот же массив, а не на два разных массива. Помните, что оператор = не копирует исходный массив.

5.3. Получение и изменение значения элемента массива

Обращение к элементам массива осуществляется с помощью квадратных скобок, в которых указывается индекс элемента. Обратите внимание на то, что нумерация элементов массива начинается с `0`, а не с `1`, поэтому первый элемент имеет индекс `0`. С помощью индексов можно присвоить начальные значения элементам массива уже после инициализации:

```
int[] arr = new int[3];
arr[0] = 10; // Первый элемент имеет индекс 0 !
arr[1] = 20; // Второй элемент
arr[2] = 30; // Третий элемент
System.out.println(Arrays.toString(arr)); // [10, 20, 30]
```

Следует учитывать, что проверка выхода указанного индекса за пределы диапазона на этапе компиляции не производится. Если указать индекс, которого нет в массиве, то на этапе выполнения возникнет ошибка, и выполнение программы будет остановлено. Часто подобная ошибка возникает при обращении к последнему элементу массива. Помните, что индекс последнего элемента на единицу меньше длины массива, т. к. нумерация индексов начинается с `0`:

```
int[] arr = {10, 20, 30};
System.out.println(arr[arr.length - 1]); // 30
```

С элементами массива можно производить такие же операции, что и с обычными переменными:

```
int[] arr = {10, 20, 30};
int x = 0;
x = arr[1] + 12;
arr[2] = x - arr[2];
System.out.println(x);       // 32
System.out.println(arr[2]); // 2
```

5.4. Перебор элементов массива

Для перебора элементов массива удобно использовать цикл `for`. В первом параметре переменной-счетчику присваивается значение `0` (элементы массива нумеруются с нуля), условием продолжения является значение переменной-счетчика меньше количества элементов массива. В третьем параметре указывается приращение на единицу на каждой итерации цикла. Внутри цикла доступ к элементу массива осуществляется с помощью квадратных скобок, в которых указывается переменная-счетчик. Пронумеруем все элементы массива, а затем выведем все значения в прямом и обратном порядке:

```
int[] arr = new int[5];
// Нумеруем все элементы массива
for (int i = 0, j = 1; i < arr.length; i++, j++) {
   arr[i] = j;
}
// Выводим значения в прямом порядке
for (int i = 0; i < arr.length; i++) {
   System.out.print(arr[i] + " ");
}
System.out.println();
System.out.println("--------------------");
// Выводим значения в обратном порядке
for (int i = arr.length - 1; i >= 0; i--) {
   System.out.print(arr[i] + " ");
}
```

Цикл `for` всегда можно заменить циклом `while`. В качестве примера пронумеруем элементы в обратном порядке, а затем выведем все значения:

```
int[] arr = new int[5];
// Нумеруем все элементы массива
int i = 0, j = arr.length;
while (i < arr.length) {
   arr[i] = j;
   i++;
   j--;
}
```

```
// Выводим значения массива
i = 0;
while (i < arr.length) {
   System.out.print(arr[i] + " ");
   i++;
} // 5 4 3 2 1
```

Если нужно перебрать все элементы массива, то можно воспользоваться циклом for each:

```
int[] arr = {1, 2, 3, 4, 5};
for (int v: arr) {
   System.out.print(v + " ");
} // 1 2 3 4 5
```

Следует заметить, что переменную v внутри цикла можно изменить, но это не отразится на исходном массиве, т. к. переменная является локальной и при использовании элементарных типов содержит лишь копию значения элемента массива:

```
int[] arr = {1, 2, 3, 4, 5};
for (int v: arr) {
   v += 10; // Переменная v локальная
}
for (int v: arr) {
   System.out.print(v + " ");
}
// Массив не изменился
// 1 2 3 4 5
```

5.5. Многомерные массивы

Многомерными массивами в языке Java считаются одномерные массивы, элементы которых содержат ссылки на вложенные массивы. На практике наиболее часто используются двумерные массивы, которые можно сравнить с таблицей, содержащей определенное количество строк и столбцов. Объявление и инициализация двумерного массива имеет следующий формат:

```
<Тип>[][] <Переменная> =
          new <Тип>[<Количество элементов1>][<Количество элементов2>];
```

Пример создания двумерного массива, содержащего две строки и четыре столбца:

```
int[][] arr = new int[2][4];
System.out.println(Arrays.deepToString(arr));
// [[0, 0, 0, 0], [0, 0, 0, 0]]
```

Для инициализации двумерного массива можно также воспользоваться фигурными скобками:

```
int[][] arr =
   {
      {1, 2, 3, 4},
```

```
        {5, 6, 7, 8}
    };
System.out.println(Arrays.deepToString(arr));
// [[1, 2, 3, 4], [5, 6, 7, 8]]
```

Получить или задать значение элемента можно, указав два индекса (не забывайте, что нумерация начинается с нуля):

```
System.out.println(arr[0][0]); // 1
System.out.println(arr[1][0]); // 5
System.out.println(arr[1][3]); // 8
arr[1][3] = 10;
System.out.println(Arrays.deepToString(arr));
// [[1, 2, 3, 4], [5, 6, 7, 10]]
```

Чтобы вывести все значения массива, необходимо использовать вложенные циклы. В качестве примера пронумеруем все элементы массива, а затем выведем все значения:

```
int[][] arr = new int[2][4];
int n = 1;
// Нумеруем все элементы массива
for (int i = 0; i < arr.length; i++) {
    for (int j = 0; j < arr[i].length; j++) {
        arr[i][j] = n;
        n++;
    }
}
// Выводим значения
for (int i = 0; i < arr.length; i++) {
    for (int j = 0; j < arr[i].length; j++) {
        System.out.printf("%3s", arr[i][j]);
    }
    System.out.println();
}
for (int[] i: arr) {
    for (int j: i) {
        System.out.printf("%3s", j);
    }
    System.out.println();
}
// Можно вывести так
System.out.println(Arrays.deepToString(arr));
```

Как уже говорилось ранее, все массивы в языке Java на самом деле одномерные. Каждый элемент многомерного массива содержит лишь ссылку на вложенный массив. Следовательно, вложенные массивы могут иметь разное количество элементов, что позволяет создавать так называемые «зубчатые» многомерные массивы:

```
int[][] arr =
   {
      {1},
      {2, 3},
      {4, 5, 6},
      {7, 8, 9, 10}
   };
// Выводим значения
for (int i = 0; i < arr.length; i++) {
   for (int j = 0; j < arr[i].length; j++) {
      System.out.printf("%3s", arr[i][j]);
   }
   System.out.println();
}
```

Результат:

```
  1
  2  3
  4  5  6
  7  8  9 10
```

Создать двумерный зубчатый массив можно также с помощью оператора `new`:

```
int[][] arr = new int[4][];
arr[0] = new int[] {1};
arr[1] = new int[] {2, 3};
arr[2] = new int[] {4, 5, 6};
arr[3] = new int[] {7, 8, 9, 10};
```

Или так:

```
int[][] arr = new int[4][];
arr[0] = new int[1];
arr[1] = new int[2];
arr[2] = new int[3];
arr[3] = new int[4];
int n = 1;
for (int i = 0; i < arr.length; i++) {
   for (int j = 0; j < arr[i].length; j++) {
      arr[i][j] = n;
      n++;
   }
}
```

5.6. Поиск минимального и максимального значений

Чтобы в массиве найти минимальное значение, следует присвоить переменной `min` **значение первого элемента массива, а затем произвести ее сравнение с остальными элементами. Если значение текущего элемента меньше значения переменной** `min`,

то присваиваем переменной `min` значение текущего элемента. Аналогично, чтобы найти максимальное значение, следует присвоить переменной `max` значение первого элемента массива, а затем произвести ее сравнение с остальными элементами. Если значение текущего элемента больше значения переменной `max`, то присваиваем переменной `max` значение текущего элемента. Пример поиска минимального и максимального значений приведен в листинге 5.2.

Листинг 5.2. Поиск минимального и максимального значений

```
public class MyClass {
   public static void main(String[] args) {
      int[] arr = {2, 5, 6, 1, 3};
      System.out.println("min = " + min(arr));
      System.out.println("max = " + max(arr));
   }
   public static int max(int[] arr) {
      int x = arr[0];
      for (int i = 0; i < arr.length; i++) {
         if (x < arr[i]) x = arr[i];
      }
      return x;
   }
   public static int min(int[] arr) {
      int x = arr[0];
      for (int i = 0; i < arr.length; i++) {
         if (x > arr[i]) x = arr[i];
      }
      return x;
   }
}
```

Используя Stream API (см. *главу 21*), аналогичную операцию можно выполнить следующим образом:

```
// import java.util.Arrays;
int[] arr = {2, 5, 6, 1, 3};
int min = Arrays.stream(arr).min().getAsInt();
int max = Arrays.stream(arr).max().getAsInt();
System.out.println("min = " + min);
System.out.println("max = " + max);
```

5.7. Заполнение массива значениями

В результате инициализации массива он будет создан из указанного количества элементов. Каждый элемент массива получит значение по умолчанию. Для числовых типов значение равно `0`, для типа `boolean` — `false`, для объектов — `null`. Если необходимо заполнить массив другими значениями, то можно задействовать цик-

лы, как мы это делали в предыдущих примерах, либо применить метод `fill()` из класса `Arrays`. Прежде чем использовать класс `Arrays`, необходимо импортировать его с помощью инструкции:

```
import java.util.Arrays;
```

Форматы метода:

```
public static void fill(<Массив>, <Значение>)
public static void fill(<Массив>, int fromIndex, int toIndex, <Значение>)
```

Например, заполним все элементы массива значением `50`:

```
int[] arr = new int[10];
System.out.println(Arrays.toString(arr));
// [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Arrays.fill(arr, 50);
System.out.println(Arrays.toString(arr));
// [50, 50, 50, 50, 50, 50, 50, 50, 50, 50]
```

При создании массива строк все элементы массива по умолчанию получат значение `null`. Метод `fill()` поможет исправить ситуацию и заполнить массив пустыми строками:

```
String[] arr = new String[5];
System.out.println(Arrays.toString(arr));
// [null, null, null, null, null]
Arrays.fill(arr, "");
System.out.println(Arrays.toString(arr));
// [, , , , ]
```

Второй формат метода `fill()` позволяет указать начальный и конечный индексы элементов, подлежащих изменению. Элемент с конечным индексом изменен не будет:

```
int[] arr = new int[10];
Arrays.fill(arr, 2, 7, 50);
System.out.println(Arrays.toString(arr));
// [0, 0, 50, 50, 50, 50, 50, 0, 0, 0]
```

5.8. Сортировка массива

Сортировка массива — это упорядочивание его элементов по возрастанию или убыванию значений. Сортировка применяется при выводе значений, а также при подготовке массива к частому поиску значений. Поиск по отсортированному массиву производится гораздо быстрее, т. к. не приходится каждый раз просматривать все значения массива.

Для сортировки массивов в языке Java предназначены статические методы `sort()` (обычная сортировка) и `parallelSort()` (параллельная сортировка) из класса `Arrays`.

Прежде чем использовать класс `Arrays`, необходимо импортировать его с помощью инструкции:

```
import java.util.Arrays;
```

Форматы метода `sort()`:

```
public static void sort(<Массив>)
public static void sort(<Массив>, int fromIndex, int toIndex)
public static <T> void sort(T[] a, Comparator<? super T> c)
public static <T> void sort(T[] a, int fromIndex, int toIndex,
                            Comparator<? super T> c)
```

Пример сортировки массива из целых чисел:

```
int[] arr = {10, 5, 6, 1, 3};
System.out.println(Arrays.toString(arr));
// [10, 5, 6, 1, 3]
Arrays.sort(arr);
System.out.println(Arrays.toString(arr));
// [1, 3, 5, 6, 10]
```

Второй формат метода `sort()` позволяет отсортировать только часть массива. Для этого нужно указать начальный и конечный индексы элементов. Элемент с конечным индексом не входит в диапазон:

```
int[] arr = {10, 5, 6, 1, 3};
Arrays.sort(arr, 0, 3);
System.out.println(Arrays.toString(arr));
// [5, 6, 10, 1, 3]
```

Если необходимо отсортировать массив в обратном порядке, то можно использовать следующий код:

```
Integer[] arr = {10, 5, 6, 1, 3};
Arrays.sort(arr, Collections.reverseOrder());
System.out.println(Arrays.toString(arr));
// [10, 6, 5, 3, 1]
```

Прежде чем использовать класс `Collections`, необходимо импортировать его с помощью инструкции:

```
import java.util.Collections;
```

Обратите внимание на то, что этот код не работает с элементарными типами — вам придется использовать классы-«обертки» над элементарными типами, такие как `Integer` для типа `int`.

Рассмотрим метод упорядочивания элементов массива, называемый *пузырьковой сортировкой*. При этом методе наименьшее значение как бы «всплывает» в начало массива, а наибольшее значение «погружается» в его конец. Сортировка выполняется в несколько проходов. При каждом проходе последовательно сравниваются значения двух элементов, которые расположены рядом. Если значение первого

элемента больше второго, то значения элементов меняются местами. Для сортировки массива из пяти элементов необходимо максимум четыре прохода и десять сравнений. Если после прохода не было ни одной перестановки, то сортировку можно прервать. В этом случае для сортировки ранее уже отсортированного массива нужен всего один проход. Последовательность сравнения элементов массива при пузырьковой сортировке по возрастанию показана в табл. 5.1, а код соответствующей программы приведен в листинге 5.3.

***Таблица 5.1.** Последовательность сравнения элементов при пузырьковой сортировке*

Проход	Описание	Значения элементов массива				
	Элементы массива	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
	Начальные значения	10	5	6	1	3
1	Сравниваются arr[3] и arr[4]	10	5	6	1	3
	Сравниваются arr[2] и arr[3]	10	5	1	6	3
	Сравниваются arr[1] и arr[2]	10	1	5	6	3
	Сравниваются arr[0] и arr[1]	1	10	5	6	3
2	Сравниваются arr[3] и arr[4]	1	10	5	3	6
	Сравниваются arr[2] и arr[3]	1	10	3	5	6
	Сравниваются arr[1] и arr[2]	1	3	10	5	6
3	Сравниваются arr[3] и arr[4]	1	3	10	5	6
	Сравниваются arr[2] и arr[3]	1	3	5	10	6
4	Сравниваются arr[3] и arr[4]	1	3	5	6	10

Листинг 5.3. Пузырьковая сортировка по возрастанию

```
import java.util.Arrays;

public class MyClass {
   public static void main(String[] args) {
      int[] arr = {10, 5, 6, 1, 3};
      sort(arr);
      System.out.println(Arrays.toString(arr));
   }
   public static void sort(int[] arr) {
      int tmp = 0, k = arr.length - 2;
      boolean is_swap = false;
      for (int i = 0; i <= k; i++) {
         is_swap = false;
         for (int j = k; j >= i; j--) {
            if (arr[j] > arr[j + 1]) {
               tmp = arr[j + 1];
```

```
                arr[j + 1] = arr[j];
                arr[j] = tmp;
                is_swap = true;
            }
        }
        // Если перестановок не было, то выходим
        if (is_swap == false) break;
    }
  }
}
```

В качестве еще одного примера произведем сортировку по убыванию (листинг 5.4). Чтобы пример был более полезным, изменим направление проходов.

Листинг 5.4. Пузырьковая сортировка по убыванию

```
import java.util.Arrays;

public class MyClass {
   public static void main(String[] args) {
      int[] arr = {10, 5, 6, 1, 3};
      sortReverse(arr);
      System.out.println(Arrays.toString(arr));
   }
   public static void sortReverse(int[] arr) {
      int tmp = 0;
      boolean is_swap = false;
      for (int i = arr.length - 1; i >= 1; i--) {
         is_swap = false;
         for (int j = 0; j < i; j++) {
            if (arr[j] < arr[j + 1]) {
               tmp = arr[j + 1];
               arr[j + 1] = arr[j];
               arr[j] = tmp;
               is_swap = true;
            }
         }
         // Если перестановок не было, то выходим
         if (is_swap == false) break;
      }
   }
}
```

Следует заметить, что метод пузырьковой сортировки в настоящее время используется большей частью только для обучения, поскольку при сортировке больших массивов он работает весьма медленно. Сейчас существуют более эффективные

алгоритмы сортировки, поэтому для сортировки лучше использовать метод `sort()` из класса `Arrays`, который мы рассмотрели в начале этого раздела.

5.9. Проверка наличия значения в массиве

Если массив не отсортирован, то проверка наличия значения в массиве сводится к перебору всех элементов от начала до конца. При наличии первого вхождения можно прервать поиск. Пример поиска первого вхождения приведен в листинге 5.5.

Листинг 5.5. Поиск элемента в неотсортированном массиве

```
public class MyClass {
   public static void main(String[] args) {
      int[] arr = {10, 5, 6, 1, 3};
      System.out.println(search(arr, 5));
   }
   public static int search(int[] arr, int key) {
      int index = -1;
      for (int i = 0; i < arr.length; i++) {
         if (arr[i] == key) {
            index = i;
            break;
         }
      }
      return index;
   }
}
```

Если значение находится в начале массива, то поиск будет произведен достаточно быстро, но если значение расположено в конце массива, то придется просматривать весь массив. Если массив состоит из 100 000 элементов, то нужно будет сделать 100 000 сравнений. Поиск по отсортированному массиву производится гораздо быстрее, т. к. не понадобится каждый раз просматривать все значения массива.

Для бинарного поиска значения в отсортированном массиве предназначен метод `binarySearch()` из класса `Arrays`. Прежде чем использовать класс `Arrays`, необходимо импортировать его с помощью инструкции:

```
import java.util.Arrays;
```

Форматы метода:

```
public static int binarySearch(<Массив>, <Значение>)
public static int binarySearch(<Массив>,
                               int fromIndex, int toIndex, <Значение>)
public static <T> int binarySearch(T[] a, T key,
                                   Comparator<? super T> c)
public static <T> int binarySearch(T[] a, int fromIndex, int toIndex,
                                   T key, Comparator<? super T> c)
```

Обратите внимание на то, что массив обязательно должен быть отсортирован. Если значение найдено в массиве, то метод возвращает индекс позиции в массиве, в противном случае — отрицательное значение, задающее позицию, в которую должно быть вставлено значение, чтобы сохранился порядок сортировки (индекс вычисляется по формуле: `-(возвращенное значение) - 1`):

```
int[] arr = {10, 5, 6, 1, 3};
Arrays.sort(arr);
System.out.println(Arrays.toString(arr));
// [1, 3, 5, 6, 10]
System.out.println(Arrays.binarySearch(arr, 10)); // 4
System.out.println(Arrays.binarySearch(arr, 8));  // -5
System.out.println(-(-5) - 1);                    // Индекс 4
```

5.10. Переворачивание и перемешивание массива

В языке Java нет методов, позволяющих изменить порядок следования элементов массива, а в некоторых случаях это может понадобиться. Например, при сортировке массива с помощью метода `sort()` нет возможности выполнить сортировку массива по убыванию при использовании элементарных типов. Но ведь мы можем отсортировать массив в прямом порядке, а затем просто изменить порядок следования элементов на противоположный, — так сказать, перевернуть массив. Давайте потренируемся и напишем метод `reverse()`, позволяющий перевернуть массив, состоящий из целых чисел (листинг 5.6).

Листинг 5.6. Изменение порядка следования элементов в массиве на противоположный

```
import java.util.Arrays;

public class MyClass {
   public static void main(String[] args) {
      int[] arr = {1, 2, 3, 4, 5};
      reverse(arr);
      System.out.println(Arrays.toString(arr)); // [5, 4, 3, 2, 1]
      reverse(arr);
      System.out.println(Arrays.toString(arr)); // [1, 2, 3, 4, 5]
   }
   public static void reverse(int[] arr) {
      int tmp = 0;
      for (int i = 0, j = arr.length - 1; i < j; i++, j--) {
         tmp = arr[i];
         arr[i] = arr[j];
         arr[j] = tmp;
      }
   }
}
```

Вначале переменной i присваиваем индекс первого элемента, а переменной j — индекс последнего элемента. На каждой итерации цикла будем увеличивать значение переменной i и уменьшать значение переменной j. Цикл будет выполняться, пока i меньше j. На каждой итерации цикла мы просто меняем местами значения двух элементов массива, предварительно сохраняя значение в промежуточной переменной tmp.

Теперь реализуем метод shuffle(), позволяющий перемешать элементы массива случайным образом (листинг 5.7). При этом значение переменной i будет меняться от 0 и до конца массива, а значение переменной j мы сгенерируем случайным образом в пределах индексов массива, используя метод random() из класса Math. Затем, как и в прошлый раз, просто поменяем местами значения двух элементов массива.

Листинг 5.7. Перемешивание элементов массива случайным образом

```
import java.util.Arrays;

public class MyClass {
   public static void main(String[] args) {
      int[] arr = {1, 2, 3, 4, 5};
      shuffle(arr);
      System.out.println(Arrays.toString(arr)); // [2, 4, 1, 5, 3]
      shuffle(arr);
      System.out.println(Arrays.toString(arr)); // [3, 1, 5, 2, 4]
   }
   public static void shuffle(int[] arr) {
      int tmp = 0, j = 0;
      for (int i = 0; i < arr.length; i++) {
         j = (int)(Math.random() * arr.length);
         if (i == j) continue;
         tmp = arr[i];
         arr[i] = arr[j];
         arr[j] = tmp;
      }
   }
}
```

5.11. Заполнение массива случайными числами

Для заполнения массива псевдослучайными числами типа byte можно воспользоваться методом nextBytes() из класса Random. Прежде чем использовать этот класс, его необходимо импортировать с помощью инструкции:

```
import java.util.Random;
```

Формат метода:

```
public void nextBytes(byte[] a)
```

Пример:

```
byte[] arr = new byte[10];
Random r = new Random();
r.nextBytes(arr);
for (byte x: arr) {
   System.out.print(x + " ");
   // 67 51 -73 72 -6 -69 -109 -126 73 -121
}
```

Если нужно заполнить массив числами от 0 до какого-либо значения n–1, то можно воспользоваться кодом из листинга 5.8.

Листинг 5.8. Заполнение массива случайными числами от 0 до n–1

```
import java.util.Arrays;
import java.util.Random;

public class MyClass {
   public static void main(String[] args) {
      int[] arr = new int[5];
      fillRandom(arr, 101);
      System.out.println(Arrays.toString(arr)); // [30, 82, 29, 35, 37]
      fillRandom(arr, 11);
      System.out.println(Arrays.toString(arr)); // [10, 9, 0, 5, 1]
   }
   public static void fillRandom(int[] arr, int n) {
      Random r = new Random();
      for (int i = 0; i < arr.length; i++) {
         arr[i] = r.nextInt(n);
      }
   }
}
```

5.12. Копирование элементов из одного массива в другой

Так как в переменной сохраняется лишь ссылка на массив, это обстоятельство следует учитывать при использовании оператора присваивания. Рассмотрим пример:

```
int[] arr1 = {1, 2, 3, 4, 5}, arr2;
arr2 = arr1; // Якобы создали копию массива
System.out.println(Arrays.toString(arr1)); // [1, 2, 3, 4, 5]
System.out.println(Arrays.toString(arr2)); // [1, 2, 3, 4, 5]
```

Теперь попробуем изменить первый элемент массива в переменной arr2:

```
arr2[0] = 33;
System.out.println(Arrays.toString(arr1)); // [33, 2, 3, 4, 5]
System.out.println(Arrays.toString(arr2)); // [33, 2, 3, 4, 5]
```

Как видно из примера, изменение значения в переменной arr2 привело также к изменению значения в переменной arr1. А это значит, что обе переменные ссылаются на один и тот же массив, а не на два разных массива. Помните, что оператор = не копирует исходный массив.

Чтобы создать копию массива или скопировать элементы из одного массива в другой, применяется метод copyOf() из класса Arrays. Прежде чем использовать класс Arrays, необходимо импортировать его с помощью инструкции:

```
import java.util.Arrays;
```

Формат метода:

```
public static <Тип>[] copyOf(<Тип>[] original, int newLength)
```

Пример создания копии массива:

```
int[] arr1 = {1, 2, 3, 4, 5}, arr2;
arr2 = Arrays.copyOf(arr1, arr1.length);
System.out.println(Arrays.toString(arr1)); // [1, 2, 3, 4, 5]
System.out.println(Arrays.toString(arr2)); // [1, 2, 3, 4, 5]
```

Попробуем изменить первый элемент массива в переменной arr2:

```
arr2[0] = 33;
System.out.println(Arrays.toString(arr1)); // [1, 2, 3, 4, 5]
System.out.println(Arrays.toString(arr2)); // [33, 2, 3, 4, 5]
```

Теперь изменения коснулись только одного массива.

Если во втором параметре указать не количество элементов исходного массива, а любое другое значение, то тем самым мы укажем длину нового массива. Если длина нового массива больше длины исходного массива, то дополнительные элементы получат значения по умолчанию для используемого типа. Если длина меньше, то будут скопированы только начальные элементы:

```
int[] arr1 = {1, 2, 3, 4, 5}, arr2;
arr2 = Arrays.copyOf(arr1, 7);
System.out.println(Arrays.toString(arr2)); // [1, 2, 3, 4, 5, 0, 0]
arr2 = Arrays.copyOf(arr1, 3);
System.out.println(Arrays.toString(arr2)); // [1, 2, 3]
```

Для копирования всех элементов массива или только какого-либо диапазона можно воспользоваться методом copyOfRange() из класса Arrays. Формат метода:

```
public static <Тип>[] copyOfRange(<Тип>[] original, int from, int to)
```

В параметре from указывается начальный индекс, а в параметре to — конечный индекс (элемент с конечным индексом скопирован не будет). Создадим копию массива с помощью метода copyOfRange():

```
int[] arr1 = {1, 2, 3, 4, 5}, arr2;
arr2 = Arrays.copyOfRange(arr1, 0, arr1.length);
System.out.println(Arrays.toString(arr1)); // [1, 2, 3, 4, 5]
System.out.println(Arrays.toString(arr2)); // [1, 2, 3, 4, 5]
arr2[0] = 33;
System.out.println(Arrays.toString(arr1)); // [1, 2, 3, 4, 5]
System.out.println(Arrays.toString(arr2)); // [33, 2, 3, 4, 5]
```

А теперь скопируем только элементы от индекса `3` до конца массива:

```
int[] arr1 = {1, 2, 3, 4, 5}, arr2;
arr2 = Arrays.copyOfRange(arr1, 3, arr1.length);
System.out.println(Arrays.toString(arr1)); // [1, 2, 3, 4, 5]
System.out.println(Arrays.toString(arr2)); // [4, 5]
```

Если значение в параметре `to` окажется больше количества элементов в исходном массиве, то будут добавлены элементы, и они получат значения по умолчанию для используемого типа:

```
int[] arr1 = {1, 2, 3, 4, 5}, arr2;
arr2 = Arrays.copyOfRange(arr1, 3, 7);
System.out.println(Arrays.toString(arr1)); // [1, 2, 3, 4, 5]
System.out.println(Arrays.toString(arr2)); // [4, 5, 0, 0]
```

Для копирования массива можно также воспользоваться методом `arraycopy()` из класса `System`. Формат метода:

```
public static void arraycopy(Object src, int srcPos,
                             Object dest, int destPos, int length)
```

В первом параметре указывается исходный массив, из которого копируются элементы, а во втором параметре — начальный индекс в этом массиве. В третьем параметре задается массив, в который будут вставлены элементы, а в четвертом параметре — индекс, с которого начнется вставка. Параметр `length` задает количество копируемых элементов. Обратите внимание на то, что метод не возвращает новый массив, а изменяет существующий массив из третьего параметра, поэтому длина этого массива должна соответствовать указанным значениям. Указываемые индексы в массивах должны существовать, иначе будет выведено сообщение об ошибке, и выполнение программы остановится. В качестве примера создадим копию массива с помощью метода `arraycopy()`:

```
int[] arr1 = {1, 2, 3, 4, 5}, arr2 = new int[5];
System.arraycopy(arr1, 0, arr2, 0, arr1.length);
System.out.println(Arrays.toString(arr1)); // [1, 2, 3, 4, 5]
System.out.println(Arrays.toString(arr2)); // [1, 2, 3, 4, 5]
arr2[0] = 33;
System.out.println(Arrays.toString(arr1)); // [1, 2, 3, 4, 5]
System.out.println(Arrays.toString(arr2)); // [33, 2, 3, 4, 5]
```

А теперь скопируем только три элемента из массива `arr1` от индекса `2` и вставим их, начиная с позиции, имеющей индекс `1`, в массив `arr2`:

```
int[] arr1 = {1, 2, 3, 4, 5}, arr2 = new int[6];
System.arraycopy(arr1, 2, arr2, 1, 3);
System.out.println(Arrays.toString(arr1)); // [1, 2, 3, 4, 5]
System.out.println(Arrays.toString(arr2)); // [0, 3, 4, 5, 0, 0]
```

Итак, мы научились создавать копии массивов, а не просто копировать ссылку на массив, но тут скрывается еще одна проблема. Массивы в языке Java могут содержать объекты и могут быть многомерными. В этом случае элементы опять содержат только ссылку на объект или вложенный массив. Рассмотрим проблему на примере копирования многомерного массива:

```
int[][] arr1 =
{
   {1, 2, 3, 4},
   {5, 6, 7, 8}
};
int[][] arr2 = new int[2][4];
// Якобы создали копию массива
System.arraycopy(arr1, 0, arr2, 0, arr1.length);
System.out.println(Arrays.deepToString(arr1));
// [[1, 2, 3, 4], [5, 6, 7, 8]]
System.out.println(Arrays.deepToString(arr2));
// [[1, 2, 3, 4], [5, 6, 7, 8]]
arr2[0][0] = 33;
// Изменились оба вложенных массива!!!
System.out.println(Arrays.deepToString(arr1));
// [[33, 2, 3, 4], [5, 6, 7, 8]]
System.out.println(Arrays.deepToString(arr2));
// [[33, 2, 3, 4], [5, 6, 7, 8]]
// Якобы создали копию массива
arr2 = Arrays.copyOf(arr1, arr1.length);
arr2[0][0] = 0;
// Изменились оба вложенных массива!!!
System.out.println(Arrays.deepToString(arr1));
// [[0, 2, 3, 4], [5, 6, 7, 8]]
System.out.println(Arrays.deepToString(arr2));
// [[0, 2, 3, 4], [5, 6, 7, 8]]
```

Помните, что реальную копию массива мы получим только при использовании элементарных типов. При использовании объектов и многомерных массивов всегда будут копироваться именно ссылки.

5.13. Сравнение массивов

При использовании логического оператора == применительно к массивам производится сравнение ссылок, а не элементов массива. Операция сравнения вернет значение `true` только в том случае, если обе переменные ссылаются на один массив:

```
int[] arr1 = {1, 2, 3, 4, 5};
int[] arr2 = {1, 2, 3, 4, 5};
int[] arr3 = arr1;
System.out.println(arr1 == arr2); // false
System.out.println(arr1 == arr3); // true
```

Чтобы сравнить именно элементы массивов, необходимо задействовать метод `equals()` из класса `Arrays`. Прежде чем использовать класс `Arrays`, необходимо импортировать его с помощью инструкции:

```
import java.util.Arrays;
```

Форматы метода:

```
public static boolean equals(<Массив1>, <Массив2>)
public static boolean equals(<Массив1>, int aFromIndex, int aToIndex,
                             <Массив2>, int bFromIndex, int bToIndex)
public static <T> boolean equals(T[] a, T[] a2,
                                 Comparator<? super T> cmp)
public static <T> boolean equals(T[] a, int aFromIndex, int aToIndex,
                                 T[] b, int bFromIndex, int bToIndex,
                                 Comparator<? super T> cmp)
```

Первый формат метода `equals()` вернет значение `true`, если длина массивов одинаковая, а также совпадают значения всех элементов:

```
int[] arr1 = {1, 2, 3, 4, 5};
int[] arr2 = {1, 2, 3, 4, 5};
int[] arr3 = arr1;
int[] arr4 = {1, 0, 3, 4, 5};
System.out.println(Arrays.equals(arr1, arr2)); // true
System.out.println(Arrays.equals(arr1, arr3)); // true
System.out.println(Arrays.equals(arr1, arr4)); // false
```

Остальные форматы метода `equals()` доступны, начиная с Java 9:

```
int[] arr1 = {1, 2, 3, 4, 5};
int[] arr2 = {1, 2, 3, 7, 8};
System.out.println(Arrays.equals(arr1, 0, 3, arr2, 0, 3)); // true
```

При сравнении многомерных массивов возникнет точно такая же проблема, как и с копированием многомерных массивов. Речь опять идет о ссылках:

```
int[][] arr1 = { {1, 2, 3, 4}, {5, 6, 7, 8} };
int[][] arr2 = { {1, 2, 3, 4}, {5, 6, 7, 8} };
int[][] arr3 = arr1;
System.out.println(Arrays.equals(arr1, arr2)); // false
System.out.println(Arrays.equals(arr1, arr3)); // true
```

Для сравнения многомерных массивов вместо метода `equals()` следует использовать метод `deepEquals()` из класса `Arrays`. Формат метода:

```
public static boolean deepEquals(<Массив1>, <Массив2>)
```

Пример:

```
int[][] arr1 = { {1, 2, 3, 4}, {5, 6, 7, 8} };
int[][] arr2 = { {1, 2, 3, 4}, {5, 6, 7, 8} };
int[][] arr3 = arr1;
int[][] arr4 = { {1, 0, 3, 4}, {5, 6, 7, 8} };
System.out.println(Arrays.deepEquals(arr1, arr2)); // true
System.out.println(Arrays.deepEquals(arr1, arr3)); // true
System.out.println(Arrays.deepEquals(arr1, arr4)); // false
```

5.14. Преобразование массива в строку

Преобразовать массив в строку позволяют следующие методы из класса `Arrays`:

- `toString()` — преобразует массив в строку специального формата. Формат метода:

  ```
  public static String toString(<Массив>)
  ```

  Пример:

  ```
  int[] arr1 = {1, 2, 3, 4};
  int[][] arr2 = { {1, 2, 3, 4}, {5, 6, 7, 8} };
  System.out.println(Arrays.toString(arr1)); // [1, 2, 3, 4]
  System.out.println(Arrays.toString(arr2));
  // [[I@15db9742, [I@6d06d69c]
  ```

  Как видно из последней инструкции, при указании многомерного массива в качестве значения элементов мы получили адреса вложенных массивов;

- `deepToString()` — преобразует многомерный массив в строку специального формата. С одномерными массивами метод не работает. Формат метода:

  ```
  public static String deepToString(Object[] a)
  ```

  Пример:

  ```
  int[][] arr1 = { {1, 2, 3, 4}, {5, 6, 7, 8} };
  int[][][] arr2 = {
     { {1, 2, 3, 4},  {5, 6, 7, 8} },
     { {1, 2, 3, 4},  {5, 6, 7, 8} }
  };
  System.out.println(Arrays.deepToString(arr1));
  // [[1, 2, 3, 4], [5, 6, 7, 8]]
  System.out.println(Arrays.deepToString(arr2));
  // [[[1, 2, 3, 4], [5, 6, 7, 8]], [[1, 2, 3, 4], [5, 6, 7, 8]]]
  ```

ГЛАВА 6

Символы и строки

Строки в языке Java представляют собой последовательности символов в кодировке UTF-16. Длина строки ограничена лишь объемом оперативной памяти компьютера. Строки являются неизменяемыми, поэтому практически все строковые методы в качестве значения возвращают новую строку. Иными словами, можно получить отдельный символ по индексу, но изменить его нельзя.

В некоторых языках программирования концом строки является нулевой символ. В языке Java нулевой символ может быть расположен внутри строки:

```
String s = "string" + (char)0 + "string";
System.out.println(s); // Нулевой символ - НЕ конец строки
System.out.println((int)s.charAt(6)); // 0
```

6.1. Объявление и инициализация отдельного символа

Для хранения символа используется тип `char`. Переменной, имеющей тип `char`, можно присвоить числовое значение (код символа) или указать символ внутри апострофов. Обратите внимание на то, что использовать вместо апострофов кавычки нельзя:

```
char ch1, ch2;
ch1 = 119;
ch2 = 'w';
System.out.println(ch1 + " " + ch2); // w w
```

Внутри апострофов можно указать *специальные символы* — комбинации знаков, обозначающие служебные или непечатаемые символы. В языке Java доступны следующие специальные символы:

- `\n` — перевод строки;
- `\r` — возврат каретки;
- `\t` — горизонтальная табуляция;

- \b — возврат на один символ;
- \f — перевод формата;
- \' — апостроф;
- \" — кавычка;
- \\ — обратная косая черта;
- \uN — Unicode-код символа в шестнадцатеричном виде (значение N от 0000 до FFFF).

Пример указания значений:

```
char ch1, ch2;
ch1 = '\u005B'; // Символ [
ch2 = '\u005D'; // Символ ]
System.out.println(ch1 + " " + ch2); // [ ]
```

Чтобы получить код символа, достаточно выполнить приведение к типу int:

```
char ch = 'n';
System.out.println((int)ch);          // 110
```

6.2. Создание строки

Строки в языке Java являются экземплярами класса String. Объявить и инициализировать переменную можно следующим образом:

```
String s = "";
```

Класс String содержит несколько конструкторов, позволяющих создать пустую строку или строку на основе массивов типа char[] и byte[]. В этом случае для создания строки нужно использовать оператор new. В качестве примера создадим пустую строку и строку из массивов:

```
String s1 = new String();         // Пустая строка
System.out.println(s1.length()); // 0
char[] arr1 = {'с', 'т', 'р', 'о', 'к', 'а'};
String s2 = new String(arr1);     // Строка из массива типа char[]
System.out.println(s2);           // строка
byte[] arr2 = {115, 116, 114, 105, 110, 103};
String s3 = new String(arr2);     // Строка из массива типа byte[]
System.out.println(s3);           // string
```

Строку можно создать на основе других данных. Для этого следует воспользоваться статическим методом valueOf() из класса String:

```
String s1 = String.valueOf(10);
System.out.println(s1);             // 10
String s2 = String.valueOf(15.35);
System.out.println(s2);            // 15.35
char[] arr1 = {'с', 'т', 'р', 'о', 'к', 'а'};
```

```
String s3 = String.valueOf(arr1);
System.out.println(s3);              // строка
```

Каждая строка внутри кавычек является экземпляром класса `String`. Поэтому, даже если мы просто выводим строку, мы имеем дело с экземпляром класса `String`:

```
System.out.println("строка"); // строка
```

Если сразу после закрывающей кавычки вставить точку, то мы получим доступ ко всем методам класса `String`. Например, выведем только третий символ:

```
System.out.println("строка".charAt(2)); // р
```

Строки являются последовательностями символов, и так же, как и к элементу массива, к отдельному символу строки можно обратиться по индексу, начинающемуся с `0`. Поэтому для доступа к третьему символу мы указали индекс `2`. В отличие от массива, мы не можем обратиться к символу с помощью квадратных скобок. Для этого в классе `String` предназначен метод `charAt()`. Кроме того, нельзя изменить символ внутри строки. Строковые методы, изменяющие отдельные символы, всегда возвращают новую строку.

Каждый отдельный символ в строке является символом типа `char`. Чтобы вставить специальный символ в строку, можно воспользоваться теми же самыми последовательностями, что и у типа `char`:

```
System.out.println("\n \r \t \b \f ' \" \\ \u005B");
```

В этом примере мы не добавили перед апострофом символ слэша, т. к. апостроф внутри строки можно не экранировать. Если перед апострофом добавить символ слэша, то эта комбинация будет расценена как специальный символ, и в строку будет добавлен только апостроф:

```
System.out.println("\'"); // '
```

Символ кавычки необходимо обязательно экранировать внутри строки с помощью слэша, иначе кавычка станет концом строки. Точно так же обязательно нужно экранировать сам символ слэша, чтобы вставить его в строку:

```
System.out.println("строка \"); // Ошибка!
```

Правильно будет так:

```
System.out.println("строка \\"); // строка \
```

Следует также заметить, что поместить объект на нескольких строках нельзя. Переход на новую строку вызовет синтаксическую ошибку:

```
s = "Строка1
Строка2";                            // Так нельзя!
```

В этом случае следует использовать конкатенацию строк (между объектами указывается символ `+`):

```
s = "Строка1"
  + "Строка2";                       // Правильно
```

При использовании строковых переменных уровня класса следует учитывать, что если переменная не была инициализирована при объявлении, то она получит значение по умолчанию, и это значение будет равно `null`, а не пустой строке. Любая попытка обращения к строковым методам через эту переменную приведет к ошибке во время выполнения программы. Чтобы этого избежать, статические строковые переменные класса следует инициализировать при объявлении хотя бы пустой строкой:

```
public static String s = "";
```

Точно такая же проблема возникнет при создании массива строк:

```
String[] arr = new String[3];
System.out.println(Arrays.toString(arr)); // [null, null, null]
```

В этом случае сразу после объявления массива следует заполнить массив пустыми строками с помощью метода `fill()` из класса `Arrays`:

```
String[] arr = new String[3];
Arrays.fill(arr, "");
System.out.println(Arrays.toString(arr)); // [, , ]
```

6.3. Определение длины строки

Получить длину строки позволяет метод `length()` класса `String`. Формат метода:

```
public int length()
```

Пример:

```
System.out.println("строка".length()); // 6
```

Как вы уже знаете, строки в языке Java являются последовательностями символов в кодировке UTF-16. На сегодняшний день 16-ти битов недостаточно для кодирования символов всех языков мира. Поэтому некоторые символы могут кодироваться с помощью 32 битов. Иными словами, один символ может кодироваться в строке двумя символами: первый символ содержит специальное значение от `\uD800` до `\uDBFF`, а второй — значения от `\uDC00` до `\uDFFF`. Если мы будем использовать метод `length()`, то получим длину строки. Чтобы получить количество *кодовых точек*, следует воспользоваться методом `codePointCount()`:

```
String s = "\uD835\uDFFF";
System.out.println(s.length());                        // 2
System.out.println(s.codePointCount(0, s.length())); // 1
s = "строка";
System.out.println(s.length());                        // 6
System.out.println(s.codePointCount(0, s.length())); // 6
```

Символы русского языка не попадают в диапазон этих значений, поэтому мы не станем рассматривать методы, предназначенные для работы с кодовыми точками. Однако знать об этом нужно. В случае необходимости вы всегда сможете найти описание этих методов в документации.

6.4. Доступ к отдельным символам

Получить отдельный символ из строки позволяет метод `charAt()` из класса `String`. Формат метода:

```
public char charAt(int index)
```

В качестве параметра указывается индекс символа в строке. Нумерация начинается с `0`. Если указать несуществующий индекс, то во время выполнения программы возникнет ошибка. Пример получения первого и пятого символов:

```
String s = "строка";
char ch;
ch = s.charAt(0);
System.out.println(ch); // с
ch = s.charAt(4);
System.out.println(ch); // к
```

6.5. Получение фрагмента строки

Получить фрагмент строки позволяет метод `substring()` из класса `String`. Форматы метода:

```
public String substring(int beginIndex)
public String substring(int beginIndex, int endIndex)
```

Первый формат метода `substring()` позволяет указать начальный индекс. Нумерация начинается с нуля. Метод возвращает фрагмент от указанного индекса и до конца строки:

```
String s = "0123456789";
System.out.println(s.substring(0)); // 0123456789
System.out.println(s.substring(5)); // 56789
```

Второй формат позволяет указать начальный и конечный индексы. Символ, совпадающий с конечным индексом, не будет включен в результат. Если индексы не существуют в строке, или начальный индекс меньше конечного, то при выполнении программы возникнет ошибка:

```
String s = "0123456789";
System.out.println(s.substring(0, 10)); // 0123456789
System.out.println(s.substring(5, 8));  // 567
```

6.6. Конкатенация строк

Для объединения двух строк (*конкатенации*) предназначен оператор `+`:

```
System.out.println("строка1" + "строка2"); // строка1строка2
```

Помимо оператора `+` доступен оператор `+=`, который производит конкатенацию с присваиванием:

```
String s = "строка1";
s += "строка2";
System.out.println(s);                    // строка1строка2
```

Если слева или справа от оператора + расположено значение другого типа (например, число), то это значение будет автоматически преобразовано в строку. Этим способом мы очень много раз пользовались для вывода значения различных переменных:

```
int x = 5;
System.out.println("x = " + x);           // x = 5
System.out.println(x + " - целое число"); // 5 - целое число
```

Вместо оператора + для объединения строк можно воспользоваться методом `concat()` из класса `String`. Формат метода:

```
public String concat(String str)
```

Пример:

```
String s = "строка1";
System.out.println(s.concat("строка2")); // строка1строка2
```

Оператор + очень удобен, но ценой каждой операции объединения станет новая строка. И если объединение строк выполняется достаточно часто — например, строка формируется внутри цикла, — то код станет неэффективным и очень медленным. В этом случае следует воспользоваться классом `StringBuilder`. Сначала создаем экземпляр класса, а затем (например, внутри цикла) добавляем в набор значения с помощью метода `append()`. После добавления всех элементов вызываем метод `toString()`, который возвращает строку, состоящую из этих элементов в порядке добавления:

```
StringBuilder sb = new StringBuilder();
sb.append("строка1");
sb.append("строка2");
sb.append(10);
sb.append(8.5);
String s = sb.toString();
System.out.println(s); // строка1строка2108.5
```

6.7. Настройка локали

Локалью называют совокупность локальных настроек системы. Настройки локали, как уже неоднократно отмечалось ранее, для разных стран различаются. Например, в одной стране принято десятичный разделитель вещественных чисел выводить в виде точки, а в другой — в виде запятой. Настройки локали влияют также на работу с датой и со строками.

Прежде чем настраивать локаль, необходимо импортировать класс `Locale` с помощью инструкции:

```
import java.util.Locale;
```

Настройки локали по умолчанию берутся из операционной системы. Получить текущую локаль можно с помощью метода `getDefault()`. Формат метода:

```
public static Locale getDefault()
```

Выведем локаль по умолчанию:

```
System.out.println(Locale.getDefault()); // ru_RU
```

Первый фрагмент до символа подчеркивания означает название языка, а второй фрагмент — задает страну. В нашем примере оба фрагмента совпадают. В других странах название языка может отличаться от названия страны — например, `de_DE` — для Германии и `de_AT` — для Австрии. Давайте выведем все доступные локали с помощью метода `getAvailableLocales()`:

```
System.out.println(Arrays.toString(Locale.getAvailableLocales()));
```

Список получится довольно внушительный, поэтому в книге его публиковать не вполне уместно, — запустите команду и посмотрите результат самостоятельно.

Для установки локали по умолчанию предназначен метод `setDefault()`. Формат метода:

```
public static void setDefault(Locale newLocale)
```

В качестве параметра метод `setDefault()` принимает экземпляр класса `Locale`. Для создания экземпляра в основном используются два конструктора:

```
Locale(String language)
Locale(String language, String country)
```

Первый параметр задает сокращенное название языка (например, `ru` для русского языка), а второй — сокращенное название страны (например, `RU` для России). Пример создания экземпляра класса `Locale`:

```
Locale locale = new Locale("en", "US");
```

Теперь этот экземпляр можно передать в метод `setDefault()` для установки локали по умолчанию или в другие методы для временного изменения локали. Пример установки локали по умолчанию:

```
Locale locale = new Locale("en", "US");
Locale.setDefault(locale);
System.out.println(Locale.getDefault());  // en_US
Locale.setDefault(new Locale("ru", "RU"));
System.out.println(Locale.getDefault());  // ru_RU
```

6.8. Изменение регистра символов

Для изменения регистра символов в строке предназначены следующие методы класса `String`:

- `toLowerCase()` — заменяет все символы строки соответствующими строчными буквами. Форматы метода:

```
public String toLowerCase()
public String toLowerCase(Locale locale)
```

Первый формат использует настройки локали по умолчанию, а второй — позволяет указать локаль, не меняя локаль по умолчанию:

```
System.out.println("СТРОКА".toLowerCase()); // строка
System.out.println("СТРОКА".toLowerCase(
                            new Locale("ru", "RU")));
// строка
```

- `toUpperCase()` — заменяет все символы строки соответствующими прописными буквами. Форматы метода:

```
public String toUpperCase()
public String toUpperCase(Locale locale)
```

Первый формат использует настройки локали по умолчанию, а второй — позволяет указать локаль, не меняя локаль по умолчанию:

```
System.out.println("строка".toUpperCase()); // СТРОКА
System.out.println("строка".toUpperCase(
                            new Locale("ru", "RU")));
// СТРОКА
```

6.9. Сравнение строк

Для сравнения строк нельзя использовать логические операторы == и !=, т. к. в этом случае будут сравниваться ссылки, а не строки. Давайте рассмотрим следующий пример:

```
String s1 = "строка", s2 = "строка";
System.out.println(s1 == s2);                    // true
System.out.println(
   s1.substring(0, 2) == s2.substring(0, 2));    // false
```

В первом сравнении мы получили вроде бы эквивалентность строк. Нет! Мы получили равенство ссылок. Вы помните, что строки нельзя изменять? Следовательно, их можно кэшировать в целях повышения эффективности. В нашем случае был создан один объект "строка", и каждая переменная получила ссылку на этот объект. Строки, получаемые динамически, не кэшируются, поэтому второе сравнение вернуло значение `false`, хотя строки полностью эквивалентны. Рассчитывать, что такие строки всегда не будут кэшироваться, нельзя, иначе вы получите ошибку, которая то будет появляться, то не будет. Время от времени появляющаяся ошибка — это настоящий кошмар для программиста! Бессонные ночи вам обеспечены! Никогда не используйте логические операторы для сравнения строк.

Для сравнения строк следует использовать следующие методы из класса `String`:

- `equals()` — возвращает значение `true`, если строки эквивалентны, и `false` — в противном случае. Формат метода:

```
public boolean equals(Object ob)
```

Пример:

```
String s1 = "строка", s2 = "строка";
System.out.println(s1.equals(s2));                    // true
System.out.println(
   s1.substring(0, 2).equals(s2.substring(0, 2)));  // true
System.out.println(s1.equals("СТРОКА"));              // false
```

- `equalsIgnoreCase()` — возвращает значение `true`, если строки эквивалентны, и `false` — в противном случае. Сравнение производится без учета регистра символов. Формат метода:

```
public boolean equalsIgnoreCase(String str)
```

Пример:

```
String s1 = "строка", s2 = "строка";
System.out.println(s1.equalsIgnoreCase(s2));        // true
System.out.println(s1.equalsIgnoreCase("СТРОКА")); // true
```

- `compareTo()` — сравнивает строку `str1` со строкой `str2` и возвращает одно из значений:
  - `отрицательное число` — если `str1` меньше `str2`;
  - `0` — если строки равны;
  - `положительное число` — если `str1` больше `str2`.

  Сравнение производится с учетом регистра символов. Формат метода:

```
public int compareTo(String str2)
```

Пример:

```
System.out.println("абв".compareTo("абв"));       // 0
System.out.println("ббб".compareTo("абв"));       // -1
System.out.println("абг".compareTo("абв"));       // 1
System.out.println("абв".compareTo("абВ"));       // 32
```

- `compareToIgnoreCase()` — метод аналогичен методу `compareTo()`, но сравнение выполняется без учета регистра символов. Формат метода:

```
public int compareToIgnoreCase(String str2)
```

Пример:

```
System.out.println("абв".compareToIgnoreCase("абВ")); // 0
System.out.println("ббб".compareToIgnoreCase("абВ")); // -1
System.out.println("абг".compareToIgnoreCase("абВ")); // 1
```

- `isEmpty()` — возвращает значение `true`, если строка пустая, и `false` — в противном случае. Формат метода:

```
public boolean isEmpty()
```

Пример:

```
System.out.println("".isEmpty());    // true
System.out.println("абв".isEmpty()); // false
```

- `startsWith()` — возвращает значение `true`, если строка начинается с фрагмента `prefix`, и `false` — в противном случае. Форматы метода:

```
public boolean startsWith(String prefix)
public boolean startsWith(String prefix, int offset)
```

Пример:

```
System.out.println("абвгде".startsWith("абв")); // true
System.out.println("абвгде".startsWith("абг")); // false
```

Второй формат позволяет в параметре `offset` указать индекс символа, который будет считаться началом строки:

```
System.out.println("абвгде".startsWith("абв", 0)); // true
System.out.println("абвгде".startsWith("бвг", 1)); // true
System.out.println("абвгде".startsWith("бвг", 0)); // false
```

- `endsWith()` — возвращает значение `true`, если строка заканчивается фрагментом `suffix`, и `false` — в противном случае. Формат метода:

```
public boolean endsWith(String suffix)
```

Пример:

```
System.out.println("абвгде".endsWith("где")); // true
System.out.println("абвгде".endsWith("вгд")); // false
```

6.10. Поиск и замена в строке

Для поиска и замены в строке предназначены следующие методы из класса `String`:

- `indexOf()` — ищет подстроку в строке. Возвращает индекс позиции, с которой начинается вхождение подстроки в строку. Если подстрока в строку не входит, то возвращается значение `-1`. Метод зависит от регистра символов. Форматы метода:

```
public int indexOf(String str)
public int indexOf(String str, int fromIndex)
```

Если начальная позиция `fromIndex` не указана, то поиск будет производиться с начала строки, в противном случае — с индекса `fromIndex`:

```
String s = "пример пример Пример";
System.out.println(s.indexOf("при"));     // 0
System.out.println(s.indexOf("При"));     // 14
System.out.println(s.indexOf("тест"));    // -1
System.out.println(s.indexOf("при", 9)); // -1
System.out.println(s.indexOf("при", 0)); // 0
System.out.println(s.indexOf("при", 7)); // 7
```

- `lastIndexOf()` — ищет подстроку в строке. Возвращает позицию последнего вхождения подстроки в строку. Если подстрока в строку не входит, то возвращается значение `-1`. Метод зависит от регистра символов. Форматы метода:

```
public int lastIndexOf(String str)
public int lastIndexOf(String str, int fromIndex)
```

Если начальная позиция `fromIndex` не указана, то поиск будет производиться с конца строки, в противном случае — с индекса `fromIndex`:

```
String s = "пример пример Пример Пример";
System.out.println(s.lastIndexOf("при"));     // 7
System.out.println(s.lastIndexOf("При"));     // 21
System.out.println(s.lastIndexOf("тест"));    // -1
System.out.println(s.lastIndexOf("при", 0));  // 0
System.out.println(s.lastIndexOf("при", 20)); // 7
```

- `replace()` — производит замену всех вхождений подстроки (или символа) в строку на другую подстроку (или символ) и возвращает результат в виде новой строки. Метод зависит от регистра символов. Форматы метода:

```
public String replace(char oldChar, char newChar)
public String replace(CharSequence target,
                      CharSequence replacement)
```

Пример:

```
String s = "Привет, Петя";
System.out.println(s.replace("Петя", "Вася"));
// Привет, Вася
s = "strstrstrstrstrN";
System.out.println(s.replace("str", "")); // N
System.out.println(s.replace('s', 'S'));
// StrStrStrStrStrN
```

- `trim()` — удаляет пробельные символы в начале и конце строки. Формат метода:

```
public String trim()
```

Пример:

```
String s = "     str\n\r\f\t";
System.out.println("'" + s.trim() + "'"); // 'str'
```

6.11. Преобразование строки в массив и обратно

Преобразовать строку в массив и обратно позволяют следующие методы из класса `String`:

- `toCharArray()` — преобразует строку в массив типа `char[]`. Формат метода:

```
public char[] toCharArray()
```

Пример:

```
String s = "строка";
char[] arr = s.toCharArray();
```

```
System.out.println(Arrays.toString(arr));
// [с, т, р, о, к, а]
```

- `getBytes()` — **преобразует строку в массив типа** `byte[]`, **используя кодировку по умолчанию или указанную кодировку. Форматы метода:**

```
public byte[] getBytes()
public byte[] getBytes(String charsetName)
                        throws UnsupportedEncodingException
public byte[] getBytes(Charset charset)
```

Пример:

```
// import java.nio.charset.Charset;
String s = "тест";
byte[] arr = s.getBytes();
System.out.println(Arrays.toString(arr));
// [-47, -126, -48, -75, -47, -127, -47, -126]
System.out.println(Charset.defaultCharset()); // UTF-8
arr = s.getBytes(Charset.forName("cp1251"));
System.out.println(Arrays.toString(arr));
// [-14, -27, -15, -14]
```

- `split()` — **разбивает строку по шаблону регулярного выражения и возвращает массив подстрок (шаблоны регулярных выражений мы рассмотрим в следующей главе). Форматы метода:**

```
public String[] split(String regex)
public String[] split(String regex, int limit)
```

Пример:

```
String[] arr =
         "word1.word2.word3.word4.word5".split("[\\s,.]+");
System.out.println(Arrays.toString(arr));
// [word1, word2, word3, word4, word5]
arr = "word1.word2.word3.word4.word5".split("[\\s,.]+", 3);
System.out.println(Arrays.toString(arr));
// [word1, word2, word3.word4.word5]
```

- `join()` — **преобразует массив подстрок в строку. Элементы добавляются через указанный разделитель** `delimiter`. **Форматы метода:**

```
public static String join(CharSequence delimiter,
                          CharSequence... elements)
public static String join(CharSequence delimiter,
                    Iterable<? extends CharSequence> elements)
```

Пример:

```
String[] s = {"word1", "word2", "word3"};
System.out.println(String.join(" – ", s));
```

```
// word1 — word2 — word3
System.out.println(
   String.join(" — ", "word1", "word2", "word3"));
// word1 — word2 — word3
```

- преобразовать массив байтов и массив символов в строку позволяют следующие конструкторы класса `String`:

```
String(byte[] bytes)
String(byte[] bytes, String charsetName)
                     throws UnsupportedEncodingException
String(byte[] bytes, Charset charset)
String(byte[] bytes, int offset, int length)
String(byte[] bytes, int offset, int length, String charsetName)
                     throws UnsupportedEncodingException
String(byte[] bytes, int offset, int length, Charset charset)
String(char[] value)
String(char[] value, int offset, int count)
```

Пример:

```
// import java.nio.charset.Charset;
byte[] arr = {-47, -126, -48, -75, -47, -127, -47, -126};
String s = new String(arr, Charset.forName("utf-8"));
System.out.println(s);  // тест
char[] arr2 = {'с', 'т', 'р', 'о', 'к', 'а'};
String s2 = new String(arr2);
System.out.println(s2); // строка
```

- `valueOf()` — преобразует массив символов в строку. Форматы метода:

```
public static String valueOf(char[] data)
public static String valueOf(char[] data, int offset,
                             int count)
```

Пример:

```
char[] arr = {'с', 'т', 'р', 'о', 'к', 'а'};
String s = String.valueOf(arr);
System.out.println(s); // строка
s = String.valueOf(arr, 0, 3);
System.out.println(s); // строка
```

6.12. Преобразование кодировок

Строки в языке Java всегда хранятся в кодировке UTF-16. Слово «всегда» означает, что изменить кодировку внутри строки нельзя. Но мы можем столкнуться с тремя ситуациями, при которых необходимо помнить о других кодировках:

- *кодировка файла с программой* — это первая проблема, которая может возникнуть. Буквы мы указываем внутри кавычек, но они пока не являются строкой.

При компиляции производится преобразование кодировки файла с программой в модифицированную версию кодировки UTF-8, а при запуске файл класса преобразуется в кодировку UTF-16. Если кодировка будет отличаться от используемой по умолчанию, то русские буквы либо вызовут ошибку, либо будут искажены до неузнаваемости. Дело в том, что для русского языка используются целых пять однобайтовых кодировок, и во всех этих кодировках коды русских букв различны. Избежать проблемы с кодировкой файла с программой позволяет флаг `-encoding`, который указывается при компиляции программы. Пример указания кодировки UTF-8:

```
javac -encoding utf-8 MyClass.java
```

Пример указания кодировки windows-1251:

```
javac -encoding cp1251 MyClass.java
```

- *кодировка входных данных* — данные, получаемые извне (введенные пользователем в консоли, полученные при чтении из текстового файла и др.), могут быть в различных кодировках;
- *кодировка выходных данных* — данные, которые мы выводим из программы, например, в окно консоли, в файл и др.

Чтобы правильно обработать данные в двух последних ситуациях, необходимо уметь преобразовывать строку в массив байтов в определенной кодировке и, наоборот, преобразовывать массив байтов в строку. Для преобразования кодировок используются следующие методы:

- `getBytes()` — преобразует строку в массив типа `byte[]` в определенной кодировке. Форматы метода:

```
public byte[] getBytes()
public byte[] getBytes(String charsetName)
                        throws UnsupportedEncodingException
public byte[] getBytes(Charset charset)
```

Первый формат возвращает массив байтов в кодировке по умолчанию:

```
String s = "тест";
byte[] arr = s.getBytes();
System.out.println(Arrays.toString(arr));
// [-47, -126, -48, -75, -47, -127, -47, -126]
System.out.println(Charset.defaultCharset()); // UTF-8
```

Второй формат позволяет указать кодировку в виде строки, а третий — в виде объекта класса `Charset`. Например, для кодировки windows-1251 можно указать строку `"cp1251"`, для windows-866 — `"cp866"`, для KOI8-R — `"koi8-r"` и т. д. Полный список поддерживаемых кодировок позволяет получить статический метод `availableCharsets()` из класса `Charset`:

```
// import java.nio.charset.Charset;
System.out.println(Charset.availableCharsets());
```

Если указанная кодировка не поддерживается, то второй формат метода `getBytes()` генерирует исключение `UnsupportedEncodingException`. Поэтому код преобразования необходимо поместить внутри инструкции `try...catch`. Кроме того, необходимо импортировать класс исключения с помощью инструкции:

```
import java.io.UnsupportedEncodingException;
```

В качестве примера преобразуем строку в массив байтов в кодировке windows-1251:

```
String s = "тест";
byte[] arr = new byte[0];
try {
   arr = s.getBytes("cp1251");
}
catch (UnsupportedEncodingException e) {
   e.printStackTrace();
   System.exit(1);
}
System.out.println(Arrays.toString(arr));
// [-14, -27, -15, -14]
```

- чтобы преобразовать массив байтов в строку, следует использовать следующие конструкторы класса `String`:

```
String(byte[] bytes)
String(byte[] bytes, String charsetName)
                     throws UnsupportedEncodingException
String(byte[] bytes, Charset charset)
String(byte[] bytes, int offset, int length)
String(byte[] bytes, int offset, int length, String charsetName)
                     throws UnsupportedEncodingException
String(byte[] bytes, int offset, int length, Charset charset)
```

Первый конструктор использует кодировку по умолчанию, а второй — позволяет указать кодировку в виде строки. Если указанная кодировка не поддерживается, то генерируется исключение `UnsupportedEncodingException`. Поэтому код преобразования необходимо поместить внутри инструкции `try...catch`. Кроме того, необходимо импортировать класс исключения с помощью инструкции:

```
import java.io.UnsupportedEncodingException;
```

В качестве примера преобразуем массив байтов в кодировке windows-1251 в строку:

```
byte[] arr = { -14, -27, -15, -14 };
String s = "";
try {
   s = new String(arr, "cp1251");
}
catch (UnsupportedEncodingException e) {
   e.printStackTrace();
```

```
    System.exit(1);
}
System.out.println(s); // тест
```

Четвертый, пятый и шестой конструкторы позволяют преобразовать только часть массива. Преобразуем только второй и третий символы:

```
s = new String(arr, 1, 2, Charset.forName("cp1251")); // ес
```

Если мы хотим сразу вывести массив байтов в определенной кодировке в окно консоли, то можно воспользоваться методом `write()` объекта `System.out`. Формат метода:

```
public void write(byte[] buf, int off, int len)
```

Выведем весь массив:

```
byte[] arr = { -14, -27, -15, -14 };
System.out.write(arr, 0, arr.length);
```

В этом случае корректность кодировки лежит полностью на наших плечах. Сейчас у нас данные в кодировке windows-1251. Если мы попытаемся вывести эти данные в окно консоли Windows, то, скорее всего, получим искаженный текст, т. к. по умолчанию в консоли используется кодировка windows-866. Получить текущую кодировку в консоли позволяет команда `chcp`:

```
c:\book>chcp
Текущая кодовая страница: 866
```

С помощью команды `chcp 1251` можно сменить кодировку в консоли для текущего сеанса:

```
c:\book>chcp 1251
Текущая кодовая страница: 1251
```

Но, это только полдела. По умолчанию используются точечные шрифты, которые не поддерживают кодировку windows-1251. Чтобы буквы отображались правильно, щелкаем правой кнопкой мыши на заголовке окна консоли и из контекстного меню выбираем пункт **Свойства**. В открывшемся окне переходим на вкладку **Шрифт** и выбираем шрифт **Lucida Console**. Сохраняем изменения. Теперь русские буквы в кодировке windows-1251 должны отображаться корректно.

Вроде бы мы настроили кодировку в консоли Windows, но попробуем перенести наш код в Linux. В результате опять получим проблему, ведь в этой операционной системе в консоли совсем другая кодировка. Чтобы сделать код кроссплатформенным, следует отказаться от использования конкретной кодировки. Методы `println()` и `print()` объекта `System.out` в большинстве случаев правильно определяют кодировку консоли (хотя иногда и ошибаются) и автоматически преобразуют строку в нужную кодировку. Чтобы вообще не было проблем с кодировкой в консоли, следует использовать только английские буквы, цифры и различные символы (точка, запятая и т. д.). Коды английских букв во всех однобайтовых кодировках и в кодировке UTF-8 совпадают, поэтому всегда будут отображаться правильно.

В качестве примера выведем коды русских и английских букв в кодировках windows-1251, windows-866 и KOI8-R (листинг 6.1).

Листинг 6.1. Вывод кодов русских и английских букв в разных кодировках

```
import java.io.UnsupportedEncodingException;

public class MyClass {
   public static void main(String[] args) {
      String s =
           "абвгдеёжзийклмнопрстуфхцчшщъыьэюя"
         + "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ"
         + "0123456789"
         + "abcdefghijklmnopqrstuvwxyz"
         + "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
      byte[] cp1251 = new byte[1];
      byte[] cp866 = new byte[1];
      byte[] koi8_r = new byte[1];
      try {
         cp1251 = s.getBytes("cp1251");
         cp866 = s.getBytes("cp866");
         koi8_r = s.getBytes("koi8-r");
      }
      catch (UnsupportedEncodingException e) {
         e.printStackTrace();
         System.exit(1);
      }
      System.out.printf("%6s %7s %7s %7s\n",
                        "Символ", "cp1251", "cp866", "koi8-r");
      for (int i = 0; i < cp1251.length; i++) {
         System.out.printf("%6s %7d %7d %7d\n",
            s.charAt(i), cp1251[i], cp866[i], koi8_r[i]);
      }
   }
}
```

В листинге 6.1 мы использовали метод `printf()` для форматированного вывода результата и указали в первом параметре строку, состоящую из каких-то непонятных комбинаций символов. Давайте разберемся, что это за символы и что они означают.

6.13. Форматирование строки

Для форматированного вывода в языке Java предназначен метод `printf()` из класса `PrintStream`. Форматы метода:

```
public PrintStream printf(String format, Object... args)
public PrintStream printf(Locale l, String format, Object... args)
```

Пример:

```
// import java.util.Locale;
System.out.printf("'%10.2f'", 10.5);        // '     10,50'
System.out.printf(new Locale("en", "US"),
                  "'%10.2f'", 10.5);        // '     10.50'
```

Если мы не хотим сразу выводить результат, а хотим его сохранить в виде строки, то следует воспользоваться статическим методом `format()` из класса `String`. Форматы метода:

```
public static String format(String format, Object... args)
public static String format(Locale l, String format, Object... args)
```

Пример:

```
System.out.println(
   String.format("'%10.2f'", 10.5));        // '     10,50'
System.out.println(
   String.format(new Locale("en", "US"),
                 "'%10.2f'", 10.5));        // '     10.50'
```

Оба метода зависят от настройки локали. Если локаль не указана, то используется локаль по умолчанию. От настроек локали зависит отображение десятичного разделителя вещественных чисел, разделителя тысячных групп, отображение даты и времени, а также различные другие параметры.

В параметре `format` задается строка специального формата, внутри которой могут быть указаны спецификаторы, имеющие следующие форматы:

```
%[<Индекс>][<Флаги>][<Ширина>][.<Точность>]<Тип преобразования>
%[<Индекс>][<Флаги>][<Ширина>]t<Тип преобразования даты>
```

Первый формат используется для форматирования чисел и строк, а второй — для форматирования даты и времени. Второй формат мы рассмотрим немного позже при изучении работы с датой и временем (см. *главу 8*).

В параметре `format` можно вообще не указывать спецификаторы. В этом случае значения в параметре `args` не задаются:

```
System.out.printf("Просто строка"); // Просто строка
```

Если используется только один спецификатор, то параметр `args` может содержать одно значение, в противном случае необходимо привести значения через запятую. Если количество значений не совпадает с количеством спецификаторов, то генерируется исключение. Пример указания двух спецификаторов и двух значений:

```
System.out.printf("%d %s", 10, "руб."); // 10 руб.
```

В этом примере строка содержит сразу два спецификатора: `%d` и `%s`. Спецификатор `%d` предназначен для вывода целого числа, а спецификатор `%s` — для вывода строки. Вместо спецификатора `%d` будет подставлено число `10`, а вместо спецификатора `%s` — строка `"руб."`. Обратите внимание на то, что тип данных переданных значений должен совпадать с типом спецификатора. Если в качестве значения для спе-

цификатора %d указать строку, то это приведет к ошибке времени исполнения. Никакой проверки соответствия типа на этапе компиляции не производится.

По умолчанию первому спецификатору соответствует первое значение, второму спецификатору — второе значение и т. д. Параметр <Индекс> позволяет изменить этот порядок и указать индекс значения. Обратите внимание на то, что индексы нумеруются с 1, а не с нуля. После индекса указывается символ $:

```
System.out.printf("%2$s %1$d руб.", 10, "цена"); // цена 10 руб.
```

Кроме того, один индекс можно указать несколько раз:

```
System.out.printf("%1$d %1$d", 10); // 10 10
```

Вместо указания индекса и знака $ можно использовать символ <, который означает тот же индекс, что и предыдущий:

```
System.out.printf("%d %<d", 10);    // 10 10
```

В параметре <Тип преобразования> могут быть указаны следующие символы:

- b — логическое значение:

```
System.out.printf("%b %b", true, false); // true false
```

- c — символ:

```
System.out.printf("%c", 'w');          // w
```

- s — строка (или любой другой тип, который автоматически будет преобразован в строку):

```
System.out.printf(
   "%s %s %s", "строка", 10, 5.33);    // строка 10 5.33
```

- d — десятичное целое число:

```
System.out.printf("%d %d", 10, -5);    // 10 -5
```

- o — восьмеричное число:

```
System.out.printf("%o %o", 10, 077);   // 12 77
System.out.printf("%#o %#o", 10, 077); // 012 077
```

- x — шестнадцатеричное число в нижнем регистре:

```
System.out.printf("%x %x", 10, 0xff);   // a ff
System.out.printf("%#x %#x", 10, 0xff); // 0xa 0xff
```

- X — шестнадцатеричное число в верхнем регистре:

```
System.out.printf("%X %X", 10, 0xff);   // A FF
System.out.printf("%#X %#X", 10, 0xff); // 0XA 0XFF
```

- f — вещественное число в десятичном представлении:

```
System.out.printf(
        "%f %f", 18.65781452, 12.5);   // 18,657815 12,500000
System.out.printf("%f", -18.65781452); // -18,657815
System.out.printf("%#.0f %.0f", 100.0, 100.0); // 100, 100
```

- `e` — вещественное число в экспоненциальной форме (буква `e` в нижнем регистре):

```
System.out.printf("%e", 18657.81452);   // 1,865781e+04
System.out.printf("%e", 0.000081452);   // 8,145200e-05
```

- `E` — вещественное число в экспоненциальной форме (буква `E` в верхнем регистре):

```
System.out.printf("%E", 18657.81452);   // 1,865781E+04
```

- `g` — эквивалентно `f` или `e` (выбирается более короткая запись числа):

```
System.out.printf("%g %g %g", 0.086578, 0.000086578, 1.865E-005);
      // 0,0865780 8,65780e-05 1,86500e-05
```

- `G` — эквивалентно `f` или `E` (выбирается более короткая запись числа):

```
System.out.printf("%G %G %G", 0.086578, 0.000086578, 1.865E-005);
      // 0,0865780 8,65780E-05 1,86500E-05
```

- `n` — перевод строки в зависимости от платформы:

```
System.out.printf("Строка1%nСтрока2");
```

Результат:

```
Строка1
Строка2
```

- `%` — символ процента (`%`):

```
System.out.printf("10%%"); // 10%
```

Параметр <Ширина> задает минимальную ширину поля. Если строка меньше ширины поля, то она дополняется пробелами. Если строка не помещается в указанную ширину, то значение игнорируется, и строка выводится полностью:

```
System.out.printf("'%3s'", "string");  // 'string'
System.out.printf("'%10s'", "string"); // '    string'
```

Задать минимальную ширину можно не только для строк, но и для других типов:

```
System.out.printf("'%10d'", 25);   // '        25'
System.out.printf("'%10f'", 12.5); // ' 12,500000'
```

Параметр <Точность> задает количество знаков после точки для вещественных чисел. Перед этим параметром обязательно должна стоять точка:

```
System.out.printf("'%10.5f'", 3.14159265359); // '   3,14159'
System.out.printf("'%.3f'", 3.14159265359);   // '3,142'
```

Если параметр <Точность> используется применительно к строке, то он задает максимальное количество символов. Символы, которые не помещаются, будут отброшены:

```
System.out.printf("'%5.7s'", "Hello, world!");   // 'Hello, '
System.out.printf("'%15.20s'", "Hello, world!"); // '  Hello, world!'
```

В параметре <Флаги> могут быть указаны следующие символы или их комбинация:

- # — для восьмеричных значений добавляет в начало символ 0, для шестнадцатеричных значений добавляет комбинацию символов 0x (если используется тип "x") или 0X (если используется тип "X"), для вещественных чисел указывает всегда выводить дробную точку, даже если задано значение 0 в параметре <Точность>:

```
System.out.printf("%#o %#o", 10, 077);          // 012 077
System.out.printf("%#x %#x", 10, 0xff);        // 0xa 0xff
System.out.printf("%#X %#X", 10, 0xff);        // 0XA 0XFF
System.out.printf("%#.0f %.0f", 100.0, 100.0); // 100, 100
```

- 0 — задает наличие ведущих нулей для числового значения:

```
System.out.printf("'%7d'", 100);          // '    100'
System.out.printf("'%07d'", 100);         // '0000100'
```

- – — задает выравнивание по левой границе области. По умолчанию используется выравнивание по правой границе:

```
System.out.printf("'%5d' '%-5d'", 3, 3); // '    3' '3    '
System.out.printf("'%05d'", 3);          // '00003'
```

- пробел — вставляет пробел перед положительным числом. Перед отрицательным числом будет стоять минус:

```
System.out.printf("'% d' '% d'", -3, 3);     // '-3' ' 3'
```

- + — обязательный вывод знака, как для отрицательных, так и для положительных чисел:

```
System.out.printf("'%+d' '%+d'", -3, 3);     // '-3' '+3'
```

- , — задает вывод разделителя тысячных групп:

```
System.out.printf("%d", 1000000);        // 1000000
System.out.printf("%,d", 1000000);       // 1 000 000
System.out.printf("%.2f", 1000000.5);    // 1000000,50
System.out.printf("%,.2f", 1000000.5);   // 1 000 000,50
```

- (— отрицательные числа будут помещены в круглые скобки:

```
System.out.printf("%(d", 10);            // 10
System.out.printf("%(d", -10);           // (10)
System.out.printf("%(.2f", 10.5);        // 10,50
System.out.printf("%(.2f", -10.5);       // (10,50)
```

Все результаты, которые были показаны в примерах, справедливы для локали ru_RU, которая установлена по умолчанию на моем компьютере. Если изменить локаль, то и результат будет соответствовать настройкам указанной локали. В частности, от локали зависят символ десятичного разделителя и разделитель тысячных групп для чисел. Посмотрите, как сильно отличается результат в зависимости от локали:

```
System.out.printf(new Locale("ru", "RU"),
                  "%,.2f", 1000000.5);       // 1 000 000,50
System.out.printf(new Locale("en", "US"),
                  "%,.2f", 1000000.5);       // 1,000,000.50
System.out.printf(new Locale("de", "DE"),
                  "%,.2f", 1000000.5);       // 1.000.000,50
```

При правильно настроенной локали результат будет соответствовать стандартам, принятым в конкретной стране, поэтому не забывайте настраивать локаль перед форматированием.

ГЛАВА 7

Регулярные выражения

Регулярные выражения дают возможность осуществить сложный поиск или замену в строке. В языке Java использовать регулярные выражения позволяют классы `Pattern` и `Matcher`. Класс `Pattern` описывает шаблон регулярного выражения, а класс `Matcher` реализует методы поиска по шаблону. Прежде чем использовать эти классы, необходимо подключить их с помощью инструкций:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
```

7.1. Создание шаблона и проверка полного соответствия шаблону

Создать откомпилированный шаблон регулярного выражения позволяет статический метод `compile()` из класса `Pattern`. Форматы метода:

```
public static Pattern compile(String regex)
public static Pattern compile(String regex, int flags)
```

В первом параметре указывается строка специального формата, задающая шаблон регулярного выражения. Во втором параметре можно указать различные модификаторы, задающие дополнительные настройки. Пример создания шаблона:

```
Pattern p = Pattern.compile("^[0-9]+$");
```

После создания шаблона следует вызвать метод `matcher()` и передать ему строку, с которой нужно произвести сравнение. Формат метода:

```
public Matcher matcher(CharSequence input)
```

Метод возвращает объект класса `Matcher`, который содержит множество методов, позволяющих получить результат поиска. Чтобы проверить строку на полное соответствие шаблону, следует вызвать метод `matches()`. Формат метода:

```
public boolean matches()
```

Если строка полностью соответствует шаблону, то метод возвращает значение `true`, в противном случае — значение `false`. Пример проверки:

```
Pattern p = Pattern.compile("^[0-9]+$");
Matcher m = p.matcher("12");
if (m.matches()) {
   System.out.println("Строка соответствует шаблону");
}
else {
   System.out.println("Не соответствует");
}
```

Так как метод проверяет полное соответствие строки шаблону, символы привязки к началу и концу строки (`^` и `$`) в шаблоне можно не указывать, но так выглядит более наглядно.

Выполнить проверку на полное соответствие шаблону позволяет также статический метод `matches()` из класса `Pattern`. Формат метода:

```
public static boolean matches(String regex, CharSequence input)
```

В этом случае шаблон указывается в виде строки. Пример проверки:

```
if (Pattern.matches("^[0-9]+$", "12")) {
   System.out.println("Строка соответствует шаблону");
}
else {
   System.out.println("Не соответствует");
}
```

Существует еще один, более простой способ проверки полного соответствия шаблону, не требующий использования классов `Pattern` и `Matcher`. Класс `String` содержит метод `matches()`, который позволяет указать шаблон в виде строки. Формат метода:

```
public boolean matches(String regex)
```

Если строка полностью соответствует шаблону, то метод возвращает значение `true`, в противном случае — значение `false`. Пример проверки:

```
if ("12".matches("^[0-9]+$")) {
   System.out.println("Строка соответствует шаблону");
}
else {
   System.out.println("Не соответствует");
}
```

7.2. Модификаторы

В параметре `flags` метода `compile()` можно указать следующие основные модификаторы (флаги) или их комбинацию через оператор `|` (полный перечень модификаторов смотрите в документации):

- CASE_INSENSITIVE — поиск без учета регистра символов (только для английских букв. Для поиска русских букв необходимо дополнительно указать модификатор UNICODE_CASE, как показано далее):

```
Pattern p = Pattern.compile("^[a-z]+$",
   Pattern.CASE_INSENSITIVE);
Matcher m = p.matcher("ABCDFE");
System.out.println(m.matches());                    // true
```

- UNICODE_CASE — используется совместно с флагом CASE_INSENSITIVE для поиска без учета регистра символов в кодировке Unicode. Пример поиска без учета регистра символов для русских букв:

```
Pattern p = Pattern.compile("^[а-яё]+$",
   Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE);
Matcher m = p.matcher("АБВГДЕЁ");
System.out.println(m.matches());                    // true
```

Если модификаторы не установлены, то поиск зависит от регистра символов:

```
System.out.println("АБВГДЕЁ".matches("^[а-яё]+$")); // false
```

Во-первых, обратите внимание на то, что буква ё не входит в диапазон а–я, а во-вторых, чтобы не зависеть от этих модификаторов, нужно просто указать диапазоны для разных регистров символов. Пример проверки соответствия с любой буквой русского языка вне зависимости от регистра символов:

```
System.out.println("АБВГДЕЁ".matches("^[а-яА-ЯёЁ]+$")); // true
```

- MULTILINE — поиск в строке, состоящей из нескольких подстрок, разделенных символом перевода строки. Символ ^ соответствует привязке к началу каждой подстроки, а символ $ соответствует позиции перед символом перевода строки;

- DOTALL — метасимвол «точка» будет соответствовать любому символу, включая символ перевода строки (по умолчанию метасимвол «точка» не соответствует символу перевода строки):

```
Pattern p = Pattern.compile("^.$", Pattern.DOTALL);
Matcher m = p.matcher("\n");
System.out.println(m.matches());          // true
System.out.println("\n".matches("^.$")); // false
```

- UNICODE_CHARACTER_CLASS — классы \w, \W, \d, \D, \s и \S будут соответствовать Unicode-символам:

```
Pattern p = Pattern.compile("^\\w+$",
            Pattern.UNICODE_CHARACTER_CLASS);
Matcher m = p.matcher("абв");
System.out.println(m.matches());              // true
System.out.println("абв".matches("^\\w+$")); // false
```

Обратите внимание на то, что перед классом \w указан дополнительный слэш. Символ слэша в строке является специальным, поэтому его необходимо экранировать вторым слэшем;

- LITERAL — все специальные символы будут трактоваться как обычные:

```
Pattern p = Pattern.compile("[a-z]+", Pattern.LITERAL);
Matcher m = p.matcher("h");
System.out.println(m.matches()); // false
m = p.matcher("[a-z]+");
System.out.println(m.matches()); // true
```

- UNIX_LINES — указывает считать символом перевода строки только символ \n.

Модификаторы можно установить или сбросить внутри шаблона регулярного выражения с помощью следующих выражений:

- (?idmsuxU-idmsuxU) — позволяет установить или сбросить (если буква указана после знака -) модификаторы внутри регулярного выражения. Буквы i (CASE_INSENSITIVE), d (UNIX_LINES), m (MULTILINE), s (DOTALL), u (UNICODE_CASE), x (COMMENTS) и U (UNICODE_CHARACTER_CLASS) соответствуют модификаторам в методе compile(). Пример установки модификаторов:

```
System.out.println("ABCDFE".matches("^[a-z]+$"));          // false
System.out.println("ABCDFE".matches("(?i)^[a-z]+$"));      // true
System.out.println("АБВГДЕЁ".matches("^[а-яё]+$"));        // false
System.out.println("АБВГДЕЁ".matches("(?iu)^[а-яё]+$")); // true
System.out.println("\n".matches("^.$"));                   // false
System.out.println("\n".matches("(?s)^.$"));               // true
```

- (?idmsux-idmsux:шаблон) — позволяет установить или сбросить (если буква указана после знака -) модификаторы внутри регулярного выражения. В этом случае шаблон регулярного выражения располагается внутри круглых скобок после двоеточия, что позволяет установить или сбросить модификаторы только для фрагмента шаблона. Пример установки модификаторов:

```
System.out.println(
   "ABC DEF".matches("^[a-z]+ [a-z]+$"));              // false
System.out.println(
   "ABC DEF".matches("(?i:^[a-z]+) [a-z]+$"));         // false
System.out.println(
   "ABC DEF".matches("(?i:^[a-z]+) [a-zA-Z]+$"));      // true
System.out.println(
   "ABC DEF".matches("(?i:^[a-z]+) (?i:[a-z]+$)"));    // true
```

7.3. Синтаксис регулярных выражений

Обычные символы, не имеющие специального значения, могут присутствовать в строке шаблона, и они будут трактоваться как есть. Пример указания в шаблоне последовательности обычных символов:

```
Pattern p = Pattern.compile("просто строка");
Matcher m = p.matcher("строка");
System.out.println(m.matches()); // false
```

```
m = p.matcher("просто строка");
System.out.println(m.matches()); // true
```

Внутри регулярного выражения символы ., ^, *, +, ?, {, [,], \, |, (и) имеют специальное значение. Если эти символы должны трактоваться как есть, то их следует экранировать с помощью слэша. Некоторые специальные символы теряют свое специальное значение, если их разместить внутри квадратных скобок. В этом случае экранировать их не нужно. Например, метасимвол «точка» соответствует любому символу, кроме символа перевода строки. Если необходимо найти именно точку, то перед точкой следует указать символ \ или поместить точку внутри квадратных скобок ([.]). Продемонстрируем это на примере проверки правильности введенной даты (листинг 7.1).

Листинг 7.1. Проверка правильности ввода даты

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class MyClass {
   public static void main(String[] args) {
      String d = "29,07.2018"; // Вместо точки указана запятая
      // Символ "\" не указан перед точкой
      Pattern p = Pattern.compile(
                  "^[0-3][0-9].[01][0-9].[12][09][0-9][0-9]$");
      Matcher m = p.matcher(d);
      if (m.matches())
         System.out.println("Дата введена правильно");
      else
         System.out.println("Дата введена неправильно");
      // Так как точка означает любой символ, выведет:
      // Дата введена правильно

      // Символ "\" указан перед точкой
      p = Pattern.compile(
          "^[0-3][0-9]\\.[01][0-9]\\.[12][09][0-9][0-9]$");
      m = p.matcher(d);
      if (m.matches())
         System.out.println("Дата введена правильно");
      else
         System.out.println("Дата введена неправильно");
      // Так как перед точкой указан символ "\", выведет:
      // Дата введена неправильно

      // Точка внутри квадратных скобок
      p = Pattern.compile(
          "^[0-3][0-9][.][01][0-9][.][12][09][0-9][0-9]$");
      m = p.matcher(d);
```

```
      if (m.matches())
         System.out.println("Дата введена правильно");
      else
         System.out.println("Дата введена неправильно");
      // Выведет: Дата введена неправильно
   }
}
```

В этом примере мы осуществляли привязку к началу и концу строки с помощью следующих *метасимволов*:

- `^` — привязка к началу строки (назначение зависит от модификатора);
- `$` — привязка к концу строки (назначение зависит от модификатора);
- `\A` — привязка к началу строки (не зависит от модификатора);
- `\z` — привязка к концу строки (не зависит от модификатора).

Так как метод `matches()` проверяет полное соответствие строки шаблону, символы привязки к началу и концу строки (`^` и `$`) в шаблоне можно не указывать, но так выглядит нагляднее. Если мы используем другие методы, то символы становятся важными. В этом примере, не указав привязку к началу и концу, мы получим все три цифры:

```
Pattern p = Pattern.compile("[0-9]+");
Matcher m = p.matcher("10\n20\n30");
while (m.find()) {
   System.out.println(m.group());
}
```

Результат:

```
10
20
30
```

Если мы первую инструкцию изменим следующим образом

```
Pattern p = Pattern.compile("^[0-9]+$");
```

то не будет найдено ни одного соответствия, т. к. указана привязка к концу и началу строки. Однако если указан модификатор `MULTILINE`, то поиск производится в строке, состоящей из нескольких подстрок, разделенных символом перевода строки. В этом случае символ `^` соответствует привязке к началу каждой подстроки, а символ `$` соответствует позиции перед символом перевода строки:

```
Pattern p = Pattern.compile("^[0-9]+$", Pattern.MULTILINE);
Matcher m = p.matcher("10\n20\n30");
while (m.find()) {
   System.out.println(m.group());
}
```

Хотя мы и указали привязку, все равно получим все числа, т. к. привязка действует к началу и концу каждой отдельной подстроки:

```
10
20
30
```

Чтобы в режиме MULTILINE работала привязка к началу и концу всей строки, необходимо использовать метасимволы \A и \z:

```
Pattern p = Pattern.compile("\\A[0-9]+\\z", Pattern.MULTILINE);
```

Привязку к началу и концу строки следует использовать, если строка должна полностью соответствовать регулярному выражению. Если убрать привязку, то любая строка, содержащая хотя бы одну цифру, будет соответствовать шаблону "[0-9]+":

```
Pattern p = Pattern.compile("[0-9]+");
Matcher m = p.matcher("Строка245");
while (m.find()) {
   System.out.println(m.group());
} // 245
```

Можно указать привязку только к началу или только к концу строки:

```
Pattern p = Pattern.compile("^[0-9]+");
Matcher m = p.matcher("123Строка245");
while (m.find()) {
   System.out.println(m.group());
} // 123
```

В этом примере мы получили только первое число, которое расположено в начале строки, т. к. указана привязка к началу строки. Если указать привязку только к концу строки, то получим только последнее число:

```
Pattern p = Pattern.compile("[0-9]+$");
Matcher m = p.matcher("123Строка245");
while (m.find()) {
   System.out.println(m.group());
} // 245
```

В квадратных скобках [] можно указать символы, которые могут встречаться на этом месте в строке. Можно записать символы подряд или указать диапазон через тире:

- [09] — соответствует числу 0 или 9;
- [0-9] — соответствует любому числу от 0 до 9;
- [абв] — соответствует буквам а, б и в;
- [а-г] — соответствует буквам а, б, в и г;
- [а-яё] — соответствует любой букве от а до я;
- [АБВ] — соответствует буквам А, Б и В;
- [А-ЯЁ] — соответствует любой букве от А до Я;
- [а-яА-ЯёЁ] — соответствует любой русской букве в любом регистре;

- `[0-9a-яA-ЯёЁa-zA-Z]` — любая цифра и любая русская или английская буква независимо от регистра.

> ***Обратите внимание!***
> Буква ё не входит в диапазон `[а-я]`.

Значение можно инвертировать, если после первой скобки указать символ ^. Таким образом можно указать символы, которых не должно быть на этом месте в строке:

- `[^09]` — не цифра 0 или 9;
- `[^0-9]` — не цифра от 0 до 9;
- `[^a-яA-ЯёЁa-zA-Z]` — не русская или английская буква в любом регистре.

Как вы уже знаете, точка теряет свое специальное значение, если ее заключить в квадратные скобки. Кроме того, внутри квадратных скобок могут встретиться символы, которые имеют специальное значение (например, ^ и –). Символ ^ теряет свое специальное значение, если он не расположен сразу после открывающей квадратной скобки:

```
Pattern p = Pattern.compile("[09^]");   // 0, 9 или ^
```

Чтобы отменить специальное значение символа -, его необходимо указать после всех символов перед закрывающей квадратной скобкой:

```
Pattern p = Pattern.compile("[09-]");   // 0, 9 или -
```

Все специальные символы можно сделать обычными, если перед ними указать символ \:

```
Pattern p = Pattern.compile("[0\\-9]"); // 0, - или 9
```

Вместо указания символов можно использовать стандартные классы, но следует учитывать, что они зависят от значений модификаторов, в режиме `UNICODE_CHARACTER_CLASS` трактуются гораздо шире, чем по умолчанию, и результат может не оправдать ваших ожиданий:

- `\d` — соответствует любой цифре;
- `\w` — соответствует любой букве, цифре или символу подчеркивания. Зависит от модификатора `UNICODE_CHARACTER_CLASS`. По умолчанию эквивалентно `[a–zA–Z0–9_]`. Настроим класс на работу с русскими буквами:

```
Pattern p = Pattern.compile("^\\w+$",
            Pattern.UNICODE_CHARACTER_CLASS);
Matcher m = p.matcher("абв");
System.out.println(m.matches());              // true
System.out.println("абв".matches("^\\w+$")); // false
```

- `\s` — любой пробельный символ;
- `\D` — не цифра. Эквивалентно `[^\d]`;
- `\W` — эквивалентно `[^\w]`;
- `\S` — не пробельный символ. Эквивалентно `[^\s]`.

Количество вхождений символа в строку задается с помощью *квантификаторов*:

- `{n}` — `n` вхождений символа в строку. Например, шаблон `"^[0-9]{2}$"` соответствует двум вхождениям любой цифры;
- `{n,}` — `n` или более вхождений символа в строку. Например, шаблон `"^[0-9]{2,}$"` соответствует двум и более вхождениям любой цифры;
- `{n,m}` — не менее `n` и не более `m` вхождений символа в строку. Числа указываются через запятую без пробела. Например, шаблон `"^[0-9]{2,4}$"` соответствует от двух до четырех вхождений любой цифры;
- `*` — ноль или большее число вхождений символа в строку. Эквивалентно комбинации `{0,}`;
- `+` — одно или большее число вхождений символа в строку. Эквивалентно комбинации `{1,}`;
- `?` — ни одного или одно вхождение символа в строку. Эквивалентно комбинации `{0,1}`.

Все квантификаторы являются «жадными». Это означает, что при поиске соответствия ищется самая длинная подстрока, соответствующая шаблону, и не учитываются более короткие соответствия. Рассмотрим это на примере — получим содержимое всех тегов `<b>` вместе с тегами:

```
Pattern p = Pattern.compile("<b>.*</b>", Pattern.DOTALL);
Matcher m = p.matcher("<b>Text1</b>Text2<b>Text3</b>");
while (m.find()) {
   System.out.println(m.group());
} // <b>Text1</b>Text2<b>Text3</b>
```

Вместо желаемого результата мы получили полностью строку. Чтобы ограничить «жадность» квантификатора, необходимо после него указать символ `?`:

```
Pattern p = Pattern.compile("<b>.*?</b>", Pattern.DOTALL);
Matcher m = p.matcher("<b>Text1</b>Text2<b>Text3</b>");
while (m.find()) {
   System.out.println(m.group());
}
```

Этот код выведет то, что мы искали:

```
<b>Text1</b>
<b>Text3</b>
```

Если необходимо получить содержимое без тегов, то нужный фрагмент внутри шаблона следует разместить внутри круглых скобок:

```
Pattern p = Pattern.compile("<b>(.*?)</b>", Pattern.DOTALL);
Matcher m = p.matcher("<b>Text1</b>Text2<b>Text3</b>");
while (m.find()) {
   System.out.println(m.group(0));
   System.out.println(m.group(1));
}
```

Результат:

```
<b>Text1</b>
Text1
<b>Text3</b>
Text3
```

Обратите внимание на то, что мы указали индексы в методе `group()`. Если индекс не указан или равняется `0`, то получим строку, полностью соответствующую шаблону. Индекс от `1` и далее указывает на порядковый номер круглых скобок внутри шаблона. Указав индекс `1`, мы получили фрагмент, соответствующий `(.*?)`, т. е. текст внутри тегов <b>.

Круглые скобки часто используются для группировки фрагментов внутри шаблона. В этих случаях не требуется, чтобы фрагмент запоминался и был доступен в результатах поиска:

```
Pattern p = Pattern.compile("([a-z]+((st)|(xt)))", Pattern.DOTALL);
Matcher m = p.matcher("test text");
while (m.find()) {
   for (int i = 0; i < m.groupCount(); i++) {
      System.out.print(m.group(i + 1) + " ");
   }
   System.out.println();
}
```

Результат:

```
test st st null
text xt null xt
```

В этом примере мы получили список из четырех элементов для каждого совпадения. Все эти элементы соответствуют фрагментам, заключенным в шаблоне в круглые скобки. Первый элемент содержит фрагмент, расположенный в первых круглых скобках, второй — во вторых круглых скобках и т. д. Три последних элемента являются лишними. Чтобы избежать захвата фрагмента, после открывающей круглой скобки следует разместить символы `?:`:

```
Pattern p = Pattern.compile("([a-z]+(?:(?:st)|(?:xt)))",
            Pattern.DOTALL);
Matcher m = p.matcher("test text");
while (m.find()) {
   for (int i = 0; i < m.groupCount(); i++) {
      System.out.print(m.group(i + 1) + " ");
   }
   System.out.println();
}
```

В результате список состоит только из фрагментов, полностью соответствующих регулярному выражению:

```
test
text
```

Обратите внимание на регулярное выражение в предыдущем примере:

```
"([a-z]+((st)|(xt)))"
```

Здесь мы использовали метасимвол `|`, который позволяет сделать выбор между альтернативными значениями. Выражение `n|m` соответствует одному из символов: `n` или `m`:

`красн((ая)|(ое))` — красная или красное, но не красный.

К найденному фрагменту в круглых скобках внутри шаблона можно обратиться с помощью механизма обратных ссылок. Для этого порядковый номер круглых скобок в шаблоне указывается после слэша — например, `\1`. Нумерация скобок внутри шаблона начинается с `1`. Для примера получим текст между одинаковыми парными тегами:

```
Pattern p = Pattern.compile("<([a-zA-Z]+)>(.*?)</\\1>",
            Pattern.DOTALL);
Matcher m = p.matcher("<b>Text1</b>Text2<I>Text3</I>");
while (m.find()) {
   for (int i = 0; i < m.groupCount(); i++) {
      System.out.print(m.group(i + 1) + " ");
   }
   System.out.println();
}
```

Результат:

```
b Text1
I Text3
```

Фрагментам внутри круглых скобок можно дать имена. Для этого после открывающей круглой скобки следует указать комбинацию символов `?<name>`. В качестве примера разберем на составные части адрес электронной почты:

```
Pattern p = Pattern.compile(
   "(?<name>[a-z0-9_.-]+)@(?<host>(?:[a-z0-9-]+\\.)+[a-z]{2,6})");
Matcher m = p.matcher("user@mail.ru");
while (m.find()) {
   System.out.println(m.group("name"));
   System.out.println(m.group("host"));
}
```

Результат:

```
user
mail.ru
```

Обратите внимание на то, что мы указали название группы в методе `group()`. Хотя могли бы, как обычно, указать индекс группы.

Чтобы внутри шаблона обратиться к именованным фрагментам, используется следующий синтаксис: `\k<name>`. Для примера получим текст между одинаковыми парными тегами:

```
Pattern p = Pattern.compile(
   "<(?<tag>[a-zA-Z]+)>(?<text>.*?)</(\\k<tag>)>",
   Pattern.DOTALL);
Matcher m = p.matcher("<b>Text1</b>Text2<I>Text3</I>");
while (m.find()) {
   System.out.println(m.group("text"));
}
```

Результат:

```
Text1
Text3
```

Кроме того, внутри круглых скобок могут быть расположены следующие конструкции:

- `(?=...)` — **положительный просмотр вперед. Выведем все слова, после которых расположена запятая:**

```
Pattern p = Pattern.compile("\\w+(?=[,])",
            Pattern.DOTALL | Pattern.CASE_INSENSITIVE);
Matcher m = p.matcher("text1, text2, text3 text4");
while (m.find()) {
   System.out.print(m.group() + " ");
} // text1 text2
```

- `(?!...)` — **отрицательный просмотр вперед. Выведем все слова, после которых нет запятой:**

```
Pattern p = Pattern.compile("[a-z]+[0-9](?![,])",
            Pattern.DOTALL | Pattern.CASE_INSENSITIVE);
Matcher m = p.matcher("text1, text2, text3 text4");
while (m.find()) {
   System.out.print(m.group() + " ");
} // text3 text4
```

- `(?<=...)` — **положительный просмотр назад. Выведем все слова, перед которыми расположена запятая с пробелом:**

```
Pattern p = Pattern.compile("(?<=[,][ ])[a-z]+[0-9]",
            Pattern.DOTALL | Pattern.CASE_INSENSITIVE);
Matcher m = p.matcher("text1, text2, text3 text4");
while (m.find()) {
   System.out.print(m.group() + " ");
} // text2 text3
```

- `(?<!...)` — **отрицательный просмотр назад. Выведем все слова, перед которыми расположен пробел, но перед пробелом нет запятой:**

```
Pattern p = Pattern.compile("(?<![,]) ([a-z]+[0-9])",
            Pattern.DOTALL | Pattern.CASE_INSENSITIVE);
Matcher m = p.matcher("text1, text2, text3 text4");
```

```
while (m.find()) {
   System.out.print(m.group(1) + " ");
} // text4
```

Рассмотрим небольшой пример. Допустим, необходимо получить все слова, расположенные после тире, причем перед тире и после слов должны следовать пробельные символы:

```
Pattern p = Pattern.compile("\\s\\-([a-z0-9]+)\\s",
            Pattern.DOTALL | Pattern.CASE_INSENSITIVE);
Matcher m = p.matcher("-word1 -word2 -word3 -word4 -word5");
while (m.find()) {
   System.out.print(m.group(1) + " ");
} // word2 word4
```

Как видно из примера, мы получили только два слова вместо пяти. Первое и последнее слово не попали в результат, т. к. расположены в начале и в конце строки. Чтобы эти слова попали в результат, необходимо добавить альтернативные выборы: `(^|\\s)` — для начала строки и `(\\s|$)` — для конца строки. Чтобы найденные выражения внутри круглых скобок не попали в результат, следует добавить символы `?:` после открывающей скобки:

```
Pattern p = Pattern.compile("(?:^|\\s)\\-([a-z0-9]+)(?:\\s|$)",
            Pattern.DOTALL | Pattern.CASE_INSENSITIVE);
Matcher m = p.matcher("-word1 -word2 -word3 -word4 -word5");
while (m.find()) {
   System.out.print(m.group(1) + " ");
} // word1 word3 word5
```

Первое и последнее слово успешно попали в результат. Почему же слова `word2` и `word4` не попали в список совпадений? Ведь и перед тире есть пробел, и после слова есть пробел. Чтобы понять причину, рассмотрим поиск по шагам. Первое слово успешно попадает в результат, т. к. перед тире расположено начало строки, и после слова есть пробел. После поиска указатель перемещается, и строка для дальнейшего поиска примет следующий вид:

```
"-word1 <Указатель>-word2 -word3 -word4 -word5"
```

Обратите внимание на то, что перед фрагментом `-word2` больше нет пробела, и тире не расположено в начале строки. Поэтому следующим совпадением станет слово `word3`, и указатель снова будет перемещен:

```
"-word1 -word2 -word3 <Указатель>-word4 -word5"
```

Опять перед фрагментом `-word4` нет пробела, и тире не расположено в начале строки. Поэтому следующим совпадением станет слово `word5`, и поиск будет завершен. Таким образом, слова `word2` и `word4` не попадают в результат, поскольку пробел до фрагмента уже был использован в предыдущем поиске. Чтобы этого избежать, следует воспользоваться положительным просмотром вперед `(?=...)`:

```
Pattern p = Pattern.compile("(?:^|\\s)\\-([a-z0-9]+)(?=\\s|$)",
            Pattern.DOTALL | Pattern.CASE_INSENSITIVE);
```

```
Matcher m = p.matcher("-word1 -word2 -word3 -word4 -word5");
while (m.find()) {
   System.out.print(m.group(1) + " ");
} // word1 word2 word3 word4 word5
```

В этом примере мы заменили фрагмент `(?:\\s|$)` на `(?=\\s|$)`. Поэтому все слова успешно попали в список совпадений.

7.4. Поиск всех совпадений с шаблоном

Как вы уже знаете из *разд. 7.1*, выполнить проверку полного соответствия строки шаблону регулярного выражения позволяют метод `matches()` из класса `Matcher`, статический метод `matches()` из класса `Pattern`, а также метод `matches()` из класса `String`. Методы возвращают значение `true`, если строка полностью соответствует шаблону, и значение `false` — в противном случае. Пример проверки:

```
// Без указания модификаторов
if ("\n".matches("^.$"))
     System.out.println("Да");
else System.out.println("Нет");           // Нет
// С указанием модификатора DOTALL
if ("\n".matches("(?s)^.$"))
     System.out.println("Да");            // Да
else System.out.println("Нет");
```

Символы привязки к началу и концу строки в этом случае не играют никакой роли, поскольку проверка в любом случае производится на полное соответствие шаблону, но так нагляднее.

Для проверки соответствия начала строки шаблону можно воспользоваться методом `lookingAt()` из класса `Matcher`. Метод возвращает значение `true`, если начало строки соответствует шаблону, и значение `false` — в противном случае. Формат метода:

```
public boolean lookingAt()
```

Пример:

```
Pattern p = Pattern.compile("[0-9]+");
Matcher m = p.matcher("str123");
System.out.println(m.lookingAt()); // false
m = p.matcher("123str");
System.out.println(m.lookingAt()); // true
```

Если нужно выполнить сравнение шаблона с любой частью строки, то следует воспользоваться методом `find()` из класса `Matcher`. Метод возвращает значение `true`, если в строке есть фрагмент, соответствующий шаблону, и значение `false` — в противном случае. Форматы метода:

```
public boolean find()
public boolean find(int start)
```

В этом случае привязки к началу и концу строки имеют значение:

```
Pattern p = Pattern.compile("[0-9]+");
Matcher m = p.matcher("str123");
if (m.find()) System.out.println("Да");  // Да
else          System.out.println("Нет");
p = Pattern.compile("[0-9]+$");          // Привязка к концу строки
m = p.matcher("str123");
if (m.find()) System.out.println("Да");  // Да
else          System.out.println("Нет");
p = Pattern.compile("^[0-9]+");          // Привязка к началу строки
m = p.matcher("str123");
if (m.find()) System.out.println("Да");
else          System.out.println("Нет"); // Нет
```

Если параметр `start` не указан, то поиск производится с начала строки, в противном случае параметр задает начальный индекс позиции в строке:

```
Pattern p = Pattern.compile("[0-9]+");
Matcher m = p.matcher("str123str");
if (m.find(0)) System.out.println("Да");  // Да
else          System.out.println("Нет");
if (m.find(6)) System.out.println("Да");
else          System.out.println("Нет");  // Нет
```

При каждом вызове метода `find()` указатель текущей позиции поиска перемещается ближе к концу строки, и как только совпадений больше не окажется, метод вернет значение `false`. Благодаря этому мы можем указать метод `find()` в качестве условия в цикле `while` и найти все совпадения с шаблоном. Например, посчитаем количество совпадений:

```
int count = 0;
Pattern p = Pattern.compile("[0-9]+");
Matcher m = p.matcher("10 20 30");
while (m.find()) {
   count++;
}
System.out.print(count); // 3
```

Чаще всего нам нужно получить не количество совпадений, а текст фрагмента, совпадающий с шаблоном. Чтобы это сделать, нужно воспользоваться следующими методами из класса `Matcher`:

- `group()` — возвращает фрагмент, полностью совпадающий с шаблоном, или только фрагмент внутри круглых скобок. Форматы метода:

  ```
  public String group()
  public String group(int group)
  public String group(String name)
  ```

  Если параметр не указан или указано значение `0`, то метод возвращает фрагмент, полностью совпадающий с шаблоном. Выведем все найденные числа:

```
Pattern p = Pattern.compile("[0-9]+");
Matcher m = p.matcher("10 20 30");
while (m.find()) {
   System.out.print(m.group() + " ");
} // 10 20 30
```

Если указано число больше `0` или строка, то возвращается фрагмент, заключенный внутри шаблона в круглые скобки, с указанным индексом или названием. Давайте получим полное совпадение с шаблоном, название тега и текст между парными тегами:

```
Pattern p = Pattern.compile(
   "<(?<tag>[a-zA-Z]+)>(?<text>.*?)</(\\k<tag>)>",
   Pattern.DOTALL);
Matcher m = p.matcher("<b>Text1</b>Text2<i>Text3</i>");
while (m.find()) {
   System.out.println(m.group());
   System.out.println(m.group(1) + " -> " + m.group(2));
   System.out.println(
      m.group("tag") + " >> " + m.group("text"));
}
```

Результат:

```
<b>Text1</b>
b -> Text1
b >> Text1
<i>Text3</i>
i -> Text3
i >> Text3
```

Обратите внимание на то, что если фрагменту не нашлось соответствия, то вместо пустой строки метод вернет значение `null`. Поэтому не забудьте проверить возвращаемое значение перед использованием, иначе во время выполнения произойдет ошибка:

```
Pattern p = Pattern.compile("([a-z]+((st)|(xt)))",
            Pattern.DOTALL);
Matcher m = p.matcher("test text");
while (m.find()) {
   for (int i = 0; i < m.groupCount(); i++) {
       System.out.print(m.group(i + 1) + " ");
   }
   System.out.println();
}
```

Результат:

```
test st st null
text xt null xt
```

- `groupCount()` — возвращает количество групп в шаблоне. Формат метода:

```
public int groupCount()
```

Пример использования метода `groupCount()` приведен в описании метода `group()`.

- `start()` — возвращает начальный индекс найденного фрагмента. Форматы метода:

```
public int start()
public int start(int group)
public int start(String name)
```

- `end()` — возвращает конечный индекс найденного фрагмента. Форматы метода:

```
public int end()
public int end(int group)
public int end(String name)
```

Если в методах `start()` и `end()` параметр не указан или равен `0`, то методы возвращают индексы для фрагмента полностью соответствующему шаблону, в противном случае — индексы фрагмента внутри круглых скобок. Выведем индексы разными способами:

```
Pattern p = Pattern.compile(
   "<(?<tag>[a-zA-Z]+)>(?<text>.*?)</(\\k<tag>)>",
   Pattern.DOTALL);
Matcher m = p.matcher("<b>Text1</b>Text2<i>Text3</i>");
while (m.find()) {
   System.out.print(m.group() + " ");
   System.out.println(m.start() + "..." + m.end());

   System.out.print(m.group(1) + " ");
   System.out.print(m.start(1) + "..." + m.end(1));
   System.out.print(" -> " + m.group(2) + " ");
   System.out.println(m.start(2) + "..." + m.end(2));

   System.out.print(m.group("tag") + " ");
   System.out.print(
            m.start("tag") + "..." + m.end("tag"));
   System.out.print(" >> " + m.group("text") + " ");
   System.out.println(
            m.start("text") + "..." + m.end("text"));
}
```

Результат:

```
<b>Text1</b> 0...12
b 1...2 -> Text1 3...8
b 1...2 >> Text1 3...8
```

```
<i>Text3</i> 17...29
i 18...19 -> Text3 20...25
i 18...19 >> Text3 20...25
```

Обратите внимание на то, что конечный индекс на единицу больше индекса последнего символа фрагмента. Благодаря этому можно передать индексы в различные строковые методы — например, в метод `substring()`. Выведем фрагменты, используя индексы, полученные в предыдущем примере:

```
String s = "<b>Text1</b>Text2<i>Text3</i>";
System.out.println(s.substring(17, 29)); // <i>Text3</i>
System.out.println(s.substring(18, 19)); // i
System.out.println(s.substring(20, 25)); // Text3
```

Все эти методы будут работать только при условии, что предыдущий поиск с помощью метода `find()` закончился удачно (метод `find()` вернул значение `true`). В противном случае любая попытка обращения к методам будет приводить к ошибке во время выполнения программы.

При каждом вызове метода `find()` указатель текущей позиции поиска перемещается ближе к концу строки, и как только совпадений больше не окажется, метод вернет значение `false`. Если нужно снова начать поиск с начала строки или использовать другую строку, то следует вызвать метод `reset()`. Форматы метода:

```
public Matcher reset()
public Matcher reset(CharSequence input)
```

Пример:

```
Pattern p = Pattern.compile("[0-9]+");
Matcher m = p.matcher("10 20 30");
m.find();
System.out.println(m.group()); // 10
m.reset();
m.find();
System.out.println(m.group()); // 10
m.reset("40 50 60");
m.find();
System.out.println(m.group()); // 40
```

Начиная с Java 9, в классе `Matcher` доступен метод `results()`, который позволяет получить поток данных с результатами поиска. Формат метода:

```
public Stream<MatchResult> results()
```

Пример:

```
Pattern p = Pattern.compile("[0-9]+");
Matcher m = p.matcher("10 20 30");
m.results().forEachOrdered(mr -> System.out.println(mr.group()));
```

Результат:

```
10
20
30
```

7.5. Замена в строке

Для замены в строке с помощью регулярных выражений предназначены следующие методы из класса `Matcher`:

- `replaceFirst()` — ищет первое совпадение с шаблоном и заменяет его указанной строкой. Форматы метода:

```
public String replaceFirst(String replacement)
public String replaceFirst(
                Function<MatchResult, String> replacer)
```

Пример:

```
Pattern p = Pattern.compile("[0-9]+");
Matcher m = p.matcher("10 20 30");
System.out.print(m.replaceFirst("+")); // + 20 30
```

Второй формат доступен, начиная с Java 9. Изменим регистр букв первого совпадения с шаблоном:

```
Pattern p = Pattern.compile("[a-z]+");
Matcher m = p.matcher("aa bb cc");
System.out.print(m.replaceFirst( mr -> {
   return mr.group().toUpperCase();
})); // AA bb cc
```

Вместо метода `replaceFirst()` из класса `Matcher` можно воспользоваться методом `replaceFirst()` из класса `String`. Формат метода:

```
public String replaceFirst(String regex, String replacement)
```

В первом параметре указывается шаблон регулярного выражения в виде строки, а во втором параметре — строка для замены:

```
System.out.print(
   "10 20 30".replaceFirst("[0-9]+", "+")); // + 20 30
```

- `replaceAll()` — ищет все совпадения с шаблоном и заменяет их указанным значением. Если совпадения не найдены, возвращается исходная строка. Форматы метода:

```
public String replaceAll(String replacement)
public String replaceAll(
                Function<MatchResult, String> replacer)
```

Пример:

```
Pattern p = Pattern.compile("[0-9]+");
Matcher m = p.matcher("10 20 30");
System.out.print(m.replaceAll("+")); // + + +
```

Второй формат доступен, начиная с Java 9:

```
Pattern p = Pattern.compile("[a-z]+");
Matcher m = p.matcher("aa bb cc");
```

```
System.out.print(m.replaceAll( mr -> {
   return mr.group().toUpperCase();
})); // AA BB CC
```

Вместо метода `replaceAll()` из класса `Matcher` можно воспользоваться методом `replaceAll()` из класса `String`. Формат метода:

```
public String replaceAll(String regex, String replacement)
```

В первом параметре указывается шаблон регулярного выражения в виде строки, а во втором параметре — строка для замены:

```
System.out.print(
   "10 20 30".replaceAll("[0-9]+", "+")); // + + +
```

Внутри нового фрагмента (в параметре `replacement`) можно использовать обратные ссылки `$номер` и `${name}`, соответствующие группам внутри шаблона. В качестве примера поменяем два тега местами:

```
Pattern p = Pattern.compile(
       "<(?<tag1>[a-z]+)><(?<tag2>[a-z]+)>");
Matcher m = p.matcher("<br><hr>");
System.out.print(m.replaceAll("<$2><$1>")); // <hr><br>
System.out.print(
       m.replaceAll("<${tag2}><${tag1}>")); // <hr><br>
```

Так как символ `$` является специальным, его необходимо экранировать с помощью слэша:

```
Pattern p = Pattern.compile("[0-9]+");
Matcher m = p.matcher("10 20 30");
System.out.print(m.replaceAll("\\$")); // $ $ $
```

7.6. Разбиение строки по шаблону: метод *split()*

Метод `split()` из класса `Pattern` разбивает строку по шаблону и возвращает массив подстрок. Форматы метода:

```
public String[] split(CharSequence input)
public String[] split(CharSequence input, int limit)
```

Пример:

```
Pattern p = Pattern.compile("[\\s,.]+");
String[] arr = p.split("word1, word2\nword3\r\nword4.word5");
System.out.println(Arrays.toString(arr));
// [word1, word2, word3, word4, word5]
```

Если во втором параметре задано число, то в массиве будет содержаться указанное количество подстрок. Если подстрок больше указанного количества, то последний элемент будет содержать остаток строки:

```
Pattern p = Pattern.compile("[\\s,.]+");
String[] arr = p.split("word1.word2.word3.word4.word5", 3);
System.out.println(Arrays.toString(arr));
// [word1, word2, word3.word4.word5]
```

Если разделитель не найден в строке, то список будет содержать только один элемент с исходной строкой:

```
Pattern p = Pattern.compile("[0-9]+");
String[] arr = p.split("word.word.word");
System.out.println(Arrays.toString(arr));
// [word.word.word]
```

Вместо метода `split()` из класса `Pattern` можно воспользоваться методом `split()` из класса `String`. Форматы метода:

```
public String[] split(String regex)
public String[] split(String regex, int limit)
```

Пример:

```
String[] arr = "word1.word2.word3.word4.word5".split("[\\s,.]+");
System.out.println(Arrays.toString(arr));
// [word1, word2, word3, word4, word5]
arr = "word1.word2.word3.word4.word5".split("[\\s,.]+", 3);
System.out.println(Arrays.toString(arr));
// [word1, word2, word3.word4.word5]
```

Метод `splitAsStream()` из класса `Pattern` разбивает строку по шаблону и возвращает поток с данными. Формат метода:

```
import java.util.stream.Stream;
public Stream<String> splitAsStream(CharSequence input)
```

Пример:

```
Pattern p = Pattern.compile("\\s+");
Stream<String> stream = p.splitAsStream("1 2 3");
stream.forEachOrdered( s -> System.out.println( s ));
/*
1
2
3
 */
```

ГЛАВА 8

Работа с датой и временем (классический способ)

Начиная с версии 8, язык Java поддерживает два способа работы с датой и временем. В этой главе мы рассмотрим классический способ, которым пользуется большинство программистов, а в следующей главе изучим способ, который был добавлен в Java SE 8.

8.1. Класс *Date*: получение количества миллисекунд, прошедших с 1 января 1970 года

Класс `Date` позволяет получить целое число типа `long`, представляющее количество миллисекунд, прошедших с 1 января 1970 года. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.util.Date;
```

Класс `Date` получает начальное значение времени от операционной системы с помощью метода `currentTimeMillis()` из класса `System`. Поэтому, если вам просто нужно получить метку времени, достаточно использовать этот метод. Формат метода:

```
public static long currentTimeMillis()
```

Пример:

```
System.out.println(System.currentTimeMillis()); // 1520639744938
```

8.1.1. Создание экземпляра класса *Date*

Создать экземпляр класса `Date` позволяют следующие конструкторы:

```
Date()
Date(long date)
```

Первый конструктор создает экземпляр класса `Date` со значением текущих даты и времени:

```
Date d = new Date();
System.out.println(d.toString()); // Sat Mar 10 02:56:51 MSK 2018
System.out.println(d.getTime());  // 1520639811554
```

Второй конструктор получает в качестве параметра количество миллисекунд:

```
Date d = new Date(1520639811554L);
System.out.println(d.toString()); // Sat Mar 10 02:56:51 MSK 2018
System.out.println(d.getTime());  // 1520639811554
```

Большинство методов класса `Date` помечены как устаревшие и не рекомендуются к использованию, поэтому мы даже не будем их рассматривать. Работа с классом `Date` сводится к трем действиям: вводу количества миллисекунд, сравнению и выводу количества миллисекунд или строки специального формата. Для этого предназначены следующие методы:

- `setTime()` — устанавливает новое значение количества миллисекунд. Формат метода:

  ```
  public void setTime(long time)
  ```

  Пример:

  ```
  Date d = new Date();
  d.setTime(1520639811554L);
  System.out.println(d.toString()); // Sat Mar 10 02:56:51 MSK 2018
  System.out.println(d.getTime());  // 1520639811554
  ```

- `getTime()` — возвращает количество миллисекунд. Формат метода:

  ```
  public long getTime()
  ```

- `toString()` — возвращает строковое представление даты и времени на основе количества миллисекунд. Формат метода:

  ```
  public String toString()
  ```

- `equals()` — сравнивает две даты. Если даты равны, то метод возвращает значение `true`, в противном случае — значение `false`. Формат метода:

  ```
  public boolean equals(Object obj)
  ```

  Пример:

  ```
  Date d = new Date(1520639811554L);
  if (d.equals(new Date(1520639811554L)))
     System.out.println("Равны");            // Равны
  else System.out.println("Нет");
  if (d.equals(new Date(1520639811553L)))
     System.out.println("Равны");
  else System.out.println("Нет");             // Нет
  ```

- `compareTo()` — сравнивает две даты. Возвращает значение `0` — если даты равны, положительное значение — если текущий объект больше `anotherDate`, отрицательное значение — если текущий объект меньше `anotherDate`. Формат метода:

  ```
  public int compareTo(Date anotherDate)
  ```

Пример:

```
Date d = new Date(1520639811554L);
System.out.println(
   d.compareTo(new Date(1520639811554L))); // 0
System.out.println(
   d.compareTo(new Date(1520639811553L))); // 1
System.out.println(
   d.compareTo(new Date(1520639811555L))); // -1
```

- `before()` — возвращает значение `true`, если текущий объект меньше `anotherDate`, и значение `false` — в противном случае. Формат метода:

```
public boolean before(Date anotherDate)
```

Пример:

```
Date d = new Date(1520639811554L);
System.out.println(
   d.before(new Date(1520639811554L))); // false
System.out.println(
   d.before(new Date(1520639811553L))); // false
System.out.println(
   d.before(new Date(1520639811555L))); // true
```

- `after()` — возвращает значение `true`, если текущий объект больше `anotherDate`, и значение `false` — в противном случае. Формат метода:

```
public boolean after(Date anotherDate)
```

Пример:

```
Date d = new Date(1520639811554L);
System.out.println(
   d.after(new Date(1520639811554L))); // false
System.out.println(
   d.after(new Date(1520639811553L))); // true
System.out.println(
   d.after(new Date(1520639811555L))); // false
```

8.1.2. Форматирование даты и времени

Строка, возвращаемая методом `toString()` из класса `Date`, совсем не то, что хотелось бы получить. Количество миллисекунд также ни о чем не говорит пользователю. Чтобы преобразовать экземпляр класса `Date` в более приемлемый вид, следует воспользоваться статическим методом `format()` из класса `String` или методом `printf()` из класса `PrintStream`. Первый метод возвращает строку, а второй — сразу выводит строку в окно консоли. Оба метода зависят от настройки локали. Если локаль не указана явным образом, то используются настройки по умолчанию. Форматы методов:

```
public static String format(String format, Object... args)
public static String format(Locale l, String format, Object... args)
```

```
public PrintStream printf(String format, Object... args)
public PrintStream printf(Locale l, String format, Object... args)
```

Мы уже рассматривали эти методы при изучении форматирования строк в *разд. 6.13*, но оставили без внимания способ форматирования даты и времени. Сейчас мы этот способ и рассмотрим.

В параметре `format` задается строка специального формата, внутри которой могут быть указаны спецификаторы, имеющие следующий формат для даты и времени:

```
%[<Индекс>][<Флаги>][<Ширина>]t<Тип преобразования даты>
```

По умолчанию первому спецификатору соответствует первое значение, второму спецификатору — второе значение и т. д. Параметр `<Индекс>` позволяет изменить этот порядок и указать индекс значения. Обратите внимание на то, что индексы нумеруются с `1`, а не с нуля. После индекса указывается символ `$`. Так как в качестве значения указывается объект класса `Date` и только один раз, то при работе с датой и временем индекс указывается почти всегда. Выведем текущую дату в привычном формате:

```
Date d = new Date();
System.out.printf("%1$td.%1$tm.%1$tY", d); // 10.03.2018
```

Вместо указания индекса и знака `$` можно использовать символ `<`, который означает тот же индекс, что и предыдущий. Выведем текущее время в привычном формате:

```
Date d = new Date();
System.out.printf("%tH:%<tM:%<tS", d);    // 20:19:57
```

Если задан параметр `<Ширина>`, то можно дополнительно в параметре `<Флаги>` указать символ `-`, означающий выравнивание по левому краю области:

```
Date d = new Date();
System.out.printf("'%1$-6tH'", d); // '20    '
System.out.printf("'%1$6tH'", d);  // '    20'
```

В строке формата можно комбинировать обычные спецификаторы и спецификаторы для даты и времени. Выведем текущие дату и время с предваряющим текстом:

```
// import java.util.Locale;
Date d = new Date();
System.out.printf(new Locale("ru", "RU"),
   "%s %2$td.%2$tm.%2$tY %2$tH:%2$tM:%2$tS", "Сегодня:", d);
// Сегодня: 10.03.2018 20:19:57
```

В параметре `<Тип преобразования даты>` могут быть указаны следующие значения:

- `c` — представление даты и времени в текущей локали:

```
Locale locale_en_US = new Locale("en", "US");
Locale locale_ru_RU = new Locale("ru", "RU");
Date d = new Date(1520702397754L);
System.out.println(
   String.format("%tc", d)); // сб мар. 10 20:19:57 MSK 2018
```

```
System.out.printf("%tc", d); // сб мар. 10 20:19:57 MSK 2018
System.out.println();
System.out.println(
   String.format(locale_ru_RU, "%tc", d));
// сб мар. 10 20:19:57 MSK 2018
System.out.printf(locale_ru_RU, "%tc", d);
// сб мар. 10 20:19:57 MSK 2018
System.out.println();
System.out.println(
   String.format(locale_en_US, "%tc", d));
// Sat Mar 10 20:19:57 MSK 2018
System.out.printf(locale_en_US, "%tc", d);
// Sat Mar 10 20:19:57 MSK 2018
```

- `Y` — год из четырех цифр (например, 2018);
- `C` — первые две цифры года (например, 20);
- `y` — последние две цифры года (от 00 до 99);
- `m` — номер месяца с предваряющим нулем (от 01 до 12);
- `b` или `h` — аббревиатура месяца в зависимости от настроек локали:

```
System.out.printf(locale_ru_RU, "%tb\n", d); // мар.
System.out.printf(locale_en_US, "%tb\n", d); // Mar
```

- `B` — название месяца в зависимости от настроек локали:

```
System.out.printf(locale_ru_RU, "%tB\n", d); // марта
System.out.printf(locale_en_US, "%tB\n", d); // March
```

- `d` — номер дня в месяце с предваряющим нулем (от 01 до 31);
- `e` — номер дня в месяце без предваряющего нуля (от 1 до 31);
- `F` — дата в формате год-месяц-день (2018-04-24);
- `D` — дата в формате месяц/день/год (04/24/18);
- `j` — день с начала года (от 001 до 366);
- `a` — аббревиатура дня недели в зависимости от настроек локали:

```
System.out.printf(locale_ru_RU, "%ta\n", d); // сб
System.out.printf(locale_en_US, "%ta\n", d); // Sat
```

- `A` — название дня недели в зависимости от настроек локали:

```
System.out.printf(locale_ru_RU, "%tA\n", d); // суббота
System.out.printf(locale_en_US, "%tA\n", d); // Saturday
```

- `H` — часы в 24-часовом формате с предваряющим нулем (от 00 до 23);
- `k` — часы в 24-часовом формате без предваряющего нуля (от 0 до 23);
- `I` — часы в 12-часовом формате с предваряющим нулем (от 01 до 12);
- `l` — часы в 12-часовом формате без предваряющего нуля (от 1 до 12);

- `M` — минуты (от 00 до 59);
- `S` — секунды (от 00 до 60);
- `p` — `AM` или `PM` в верхнем или нижнем регистре (в зависимости от регистра буквы `T`):

```
System.out.printf(locale_ru_RU, "%tp\n", d); // пп
System.out.printf(locale_en_US, "%tp\n", d); // pm
System.out.printf(locale_ru_RU, "%Tp\n", d); // ПП
System.out.printf(locale_en_US, "%Tp\n", d); // PM
```

- `T` — время в 24-часовом режиме в формате вида 20:19:57;
- `r` — время в 12-часовом режиме в формате вида 08:19:57 PM;
- `R` — время в 24-часовом режиме без секунд в формате вида 20:19;
- `L` — количество миллисекунд из трех цифр (например, 788);
- `Q` — количество миллисекунд, прошедших с 1 января 1970 года (например, 1461518397788);
- `s` — количество секунд, прошедших с 1 января 1970 года (например, 1461518397);
- `N` — количество наносекунд (например, 788000000);
- `Z` — название часового пояса (например, MSK);
- `z` — смещение часового пояса (например, +0300).

Выведем текущие дату и время таким образом, чтобы день недели и месяц были написаны по-русски (листинг 8.1).

Листинг 8.1. Вывод текущих даты и времени

```
import java.util.Date;
import java.util.Locale;

public class MyClass {
   public static void main(String[] args) {
      Locale locale_ru_RU = new Locale("ru", "RU");
      System.out.printf(locale_ru_RU,
         "Сегодня:\n"
       + "%1$td %1$tB %1$tY %1$tH:%1$tM:%1$tS\n"
       + "%1$td.%1$tm.%1$tY", new Date());
   }
}
```

Выведет:

```
Сегодня:
23 марта 2018 20:19:57
23.03.2018
```

8.2. Класс *Calendar*: получение доступа к отдельным компонентам даты и времени

Класс `Calendar` позволяет получить доступ к отдельным компонентам даты и времени. Прежде чем использовать класс `Calendar`, необходимо его импортировать с помощью инструкции:

```
import java.util.Calendar;
```

8.2.1. Создание объекта: метод *getInstance()*

Для создания объекта предназначен статический метод `getInstance()`. Форматы метода:

```
public static Calendar getInstance()
public static Calendar getInstance(Locale aLocale)
public static Calendar getInstance(TimeZone zone)
public static Calendar getInstance(TimeZone zone, Locale aLocale)
```

Первый формат создает объект с настройками локали и часовым поясом по умолчанию. Остальные форматы позволяют указать локаль и (или) часовой пояс явным образом. Посмотреть все настройки объекта можно с помощью метода `toString()`:

```
Calendar c = Calendar.getInstance();
System.out.println(c.toString());
```

8.2.2. Получение компонентов даты и времени

Получить отдельные компоненты даты и времени позволяет метод `get()`. Формат метода:

```
public int get(int field)
```

В параметре `field` используются следующие статические константы из класса `Calendar`: `ERA` (0), `YEAR` (1), `MONTH` (2), `WEEK_OF_YEAR` (3), `WEEK_OF_MONTH` (4), `DATE` (5), `DAY_OF_MONTH` (5), `DAY_OF_YEAR` (6), `DAY_OF_WEEK` (7) и `DAY_OF_WEEK_IN_MONTH` (8), `AM_PM` (9), `HOUR` (10), `HOUR_OF_DAY` (11), `MINUTE` (12), `SECOND` (13), `MILLISECOND` (14), `ZONE_OFFSET` (15) и `DST_OFFSET` (16) — эра, год, месяц, неделя в году, неделя в месяце, день в месяце, день в году, день недели, час, минута, секунда, миллисекунда и др. Все эти константы можно также указывать в методе `set()`, который позволяет установить соответствующие компоненты даты и времени. Описание констант приведено в листинге 8.2 (см. далее).

- При указании константы `MONTH` возвращаются значения, соответствующие следующим константам из класса `Calendar`: `JANUARY` (0), `FEBRUARY` (1), `MARCH` (2), `APRIL` (3), `MAY` (4), `JUNE` (5), `JULY` (6), `AUGUST` (7), `SEPTEMBER` (8), `OCTOBER` (9), `NOVEMBER` (10), `DECEMBER` (11) и `UNDECIMBER` (12) — они содержат целочисленные значения (тип `int`), соответствующие одноименным месяцам. Обратите внимание: нумерация

месяцев начинается с 0: январю соответствует индекс 0, а декабрю — 11 (последнее значение не применяется в григорианском календаре):

```
Calendar c = Calendar.getInstance();
c.setTimeInMillis(1520702397754L);
System.out.printf("%tc", c);
// сб мар. 10 20:19:57 MSK 2018
System.out.println(c.get(Calendar.MONTH)); // 2
if (c.get(Calendar.MONTH) == Calendar.MARCH) {
   System.out.println("Март");
} // Март
```

- При указании константы DAY_OF_WEEK возвращаются значения, соответствующие следующим константам из класса Calendar: MONDAY (2), TUESDAY (3), WEDNESDAY (4), THURSDAY (5), FRIDAY (6), SATURDAY (7) и SUNDAY (1) — они содержат целочисленные значения (тип int), соответствующие одноименным дням недели. Обратите внимание: нумерация начинается с 1, кроме того, первым днем недели считается воскресенье, а не понедельник:

```
Calendar c = Calendar.getInstance();
c.setTimeInMillis(1520702397754L);
System.out.printf("%tc", c);
// сб мар. 10 20:19:57 MSK 2018
System.out.println(c.get(Calendar.DAY_OF_WEEK)); // 7
if (c.get(Calendar.DAY_OF_WEEK) == Calendar.SATURDAY) {
   System.out.println("Суббота");
} // Суббота
```

- При указании константы AM_PM возвращаются значения, соответствующие следующим константам из класса Calendar: AM (0) и PM (1) — первая и вторая половина дня в 12-часовом режиме:

```
Calendar c = Calendar.getInstance();
c.setTimeInMillis(1520702397754L);
System.out.printf("%tc", c);
// сб мар. 10 20:19:57 MSK 2018
System.out.println(c.get(Calendar.AM_PM)); // 1
if (c.get(Calendar.AM_PM) == Calendar.PM) {
   System.out.println("PM");
} // PM
```

В качестве примера выведем значения всех компонентов даты и времени (листинг 8.2).

Листинг 8.2. Вывод компонентов даты и времени

```
import java.util.Calendar;
import java.util.Locale;

public class MyClass {
   public static void main(String[] args) {
```

```
        Locale.setDefault(new Locale("ru", "RU"));
        Calendar c = Calendar.getInstance();
        c.setTimeInMillis(1520702397754L);
        System.out.printf("%tc\n", c);
        // сб мар. 10 20:19:57 MSK 2018
        System.out.println(c.get(Calendar.ERA));            // 1
        // Год из четырех цифр
        System.out.println(c.get(Calendar.YEAR));           // 2018
        // Номер месяца от 0 до 11
        System.out.println(c.get(Calendar.MONTH));          // 2
        System.out.println(Calendar.MARCH);                 // 2
        // Неделя в году
        System.out.println(c.get(Calendar.WEEK_OF_YEAR));   // 10
        // Неделя в месяце
        System.out.println(c.get(Calendar.WEEK_OF_MONTH)); // 2
        // Номер дня в месяце
        System.out.println(c.get(Calendar.DATE));           // 10
        System.out.println(c.get(Calendar.DAY_OF_MONTH));   // 10
        // День с начала года
        System.out.println(c.get(Calendar.DAY_OF_YEAR));    // 69
        // День недели от 1 (воскресенье) до 7 (суббота)
        System.out.println(c.get(Calendar.DAY_OF_WEEK));    // 7
        System.out.println(Calendar.SATURDAY);              // 7
        System.out.println(c.get(
                        Calendar.DAY_OF_WEEK_IN_MONTH)); // 2
        // Calendar.AM или Calendar.PM
        System.out.println(c.get(Calendar.AM_PM));          // 1
        System.out.println(Calendar.PM);                    // 1
        // Часы в 12-часовом формате
        System.out.println(c.get(Calendar.HOUR));           // 8
        // Часы в 24-часовом формате
        System.out.println(c.get(Calendar.HOUR_OF_DAY));    // 20
        // Минуты
        System.out.println(c.get(Calendar.MINUTE));         // 19
        // Секунды
        System.out.println(c.get(Calendar.SECOND));         // 57
        // Миллисекунды
        System.out.println(c.get(Calendar.MILLISECOND));    // 754
        // Смещение часового пояса (+3 Москва)
        System.out.println(c.get(Calendar.ZONE_OFFSET));    // 10800000
        System.out.println(3 * 60 * 60 * 1000);             // 10800000
        System.out.printf("%tz\n", c);                      // +0300
        System.out.println(c.get(Calendar.DST_OFFSET));     // 0
    }
}
```

Для получения количества миллисекунд, прошедших с 1 января 1970 года, можно воспользоваться методом `getTimeInMillis()`. Формат метода:

```
public long getTimeInMillis()
```

Пример:

```
Calendar c = Calendar.getInstance();
System.out.println(c.getTimeInMillis()); // 1520702397754
```

Чтобы получить экземпляр класса `Date`, можно после создания объекта класса `Calendar` вызвать метод `getTime()`:

```
// import java.util.Date;
Calendar c = Calendar.getInstance();
Date d = c.getTime();
```

Получив экземпляр класса `Date`, мы можем выполнить форматирование с помощью метода `format()` из класса `String` или метода `printf()` из класса `PrintStream`. Хотя на самом деле даже необязательно вызывать метод `getTime()` — достаточно просто указать объект класса `Calendar` в этих методах вместо объекта класса `Date`. Выведем текущие дату и время, используя только класс `Calendar` (листинг 8.3).

Листинг 8.3. Вывод текущих даты и времени

```
import java.util.Calendar;
import java.util.Locale;

public class MyClass {
   public static void main(String[] args) {
      Locale.setDefault(new Locale("ru", "RU"));
      Calendar c = Calendar.getInstance();
      System.out.printf(
         "Сегодня:\n"
       + "%1$td %1$tB %1$tY %1$tH:%1$tM:%1$tS\n"
       + "%1$td.%1$tm.%1$tY", c);
   }
}
```

Примерный результат:

```
Сегодня:
23 марта 2018 22:51:51
23.03.2018
```

С помощью метода `getDisplayName()` из класса `Calendar` можно получить символьное название для указанного компонента. Название выводится в соответствии с локалью. Если название не может быть выведено, метод вернет значение `null`. Формат метода:

```
public String getDisplayName(int field, int style, Locale locale)
```

В первом параметре указываются точно такие же константы, что и в методе `get()`, во втором параметре задается стиль с помощью констант `SHORT` (или `SHORT_FORMAT`), `SHORT_STANDALONE`, `LONG` (или `LONG_FORMAT`) и `LONG_STANDALONE`, а в третьем параметре — локаль. Выведем название месяцев в разных стилях для русской локали (листинг 8.4).

Листинг 8.4. Вывод названий месяца

```
import java.util.Calendar;
import java.util.Locale;

public class MyClass {
   public static void main(String[] args) {
      Locale ru_RU = new Locale("ru", "RU");
      Calendar c = Calendar.getInstance();
      c.setTimeInMillis(1520702397754L);
      System.out.println(
         c.getDisplayName(Calendar.MONTH,
                          Calendar.SHORT, ru_RU)); // мар.
      System.out.println(
         c.getDisplayName(Calendar.MONTH,
               Calendar.SHORT_STANDALONE, ru_RU)); // март
      System.out.println(
         c.getDisplayName(Calendar.MONTH,
                           Calendar.LONG, ru_RU)); // марта
      System.out.println(
         c.getDisplayName(Calendar.MONTH,
                Calendar.LONG_STANDALONE, ru_RU)); // март
   }
}
```

8.2.3. Установка компонентов даты и времени

При создании объекта он инициализируется текущим системным временем. Чтобы изменить это значение, используются следующие методы класса `Calendar`:

- `setTime()` — задает время на основе объекта класса `Date`. Формат метода:

  ```
  public final void setTime(Date date)
  ```

  Пример:

  ```
  // import java.util.Date;
  Calendar c = Calendar.getInstance();
  c.setTime(new Date(1520702397754L));
  System.out.printf("%tc", c);
  // сб мар. 10 20:19:57 MSK 2018
  ```

- `setTimeInMillis()` — задает время на основе количества миллисекунд. Формат метода:

```
public void setTimeInMillis(long millis)
```

Пример:

```
Calendar c = Calendar.getInstance();
c.setTimeInMillis(1520702397754L);
System.out.printf("%tc", c);
// сб мар. 10 20:19:57 MSK 2018
```

- `set()` — задает значение для всех или только для отдельных компонентов даты и времени. Форматы метода:

```
public final void set(int year, int month, int date)
public final void set(int year, int month, int date,
                      int hour, int minute)
public final void set(int year, int month, int date,
                      int hour, int minute, int second)
public void set(int field, int value)
```

Первые три формата позволяют изменить год, месяц (от `0` для января до `11` для декабря), день, часы, минуты и секунды:

```
Calendar c = Calendar.getInstance();
c.set(2018, Calendar.APRIL, 24);
System.out.printf("%tc\n", c);
// вт апр. 24 23:04:13 MSK 2018
c.set(2018, Calendar.APRIL, 24, 11, 5);
System.out.printf("%tc\n", c);
// вт апр. 24 11:05:13 MSK 2018
c.set(2018, Calendar.APRIL, 24, 11, 5, 30);
System.out.printf("%tc", c);
// вт апр. 24 11:05:30 MSK 2018
```

Четвертый формат позволяет изменить любую составляющую даты и времени. В первом параметре указываются такие же константы, что и у метода `get()`, а во втором параметре — новое значение. Пример изменения только года:

```
Calendar c = Calendar.getInstance();
c.set(Calendar.YEAR, 2020);
System.out.printf("%tc\n", c);
// вт мар. 10 23:05:54 MSK 2020
```

- `clear()` — сбрасывает значения всех компонентов даты и времени или только значение указанного компонента. Форматы метода:

```
public final void clear()
public final void clear(int field)
```

Пример:

```
Calendar c = Calendar.getInstance();
c.clear(Calendar.YEAR);
```

```
System.out.printf("%tc\n", c);
// вт мар. 10 23:07:36 MSK 1970
c.clear();
System.out.printf("%tc\n", c);
// чт янв. 01 00:00:00 MSK 1970
```

8.2.4. Сравнение объектов

Сравнить два объекта класса `Calendar` позволяют следующие методы:

- `equals()` — сравнивает две даты. Если даты равны, то метод возвращает значение `true`, в противном случае — значение `false`. Формат метода:

  ```
  public boolean equals(Object obj)
  ```

  Пример:

  ```
  Calendar c1 = Calendar.getInstance();
  Calendar c2 = Calendar.getInstance();
  c1.setTimeInMillis(1520702397754L);
  c2.setTimeInMillis(1520702397754L);
  if (c1.equals(c2))
     System.out.println("Равны");            // Равны
  else System.out.println("Нет");
  c2.setTimeInMillis(1520702397753L);
  if (c1.equals(c2))
     System.out.println("Равны");
  else System.out.println("Нет");             // Нет
  ```

- `compareTo()` — сравнивает две даты. Возвращает значение `0` — если даты равны, положительное значение — если текущий объект больше `anotherCalendar`, отрицательное значение — если текущий объект меньше `anotherCalendar`. Формат метода:

  ```
  public int compareTo(Calendar anotherCalendar)
  ```

  Пример:

  ```
  Calendar c1 = Calendar.getInstance();
  Calendar c2 = Calendar.getInstance();
  c1.setTimeInMillis(1520702397754L);
  c2.setTimeInMillis(1520702397754L);
  System.out.println(c1.compareTo(c2)); // 0
  c2.setTimeInMillis(1520702397753L);
  System.out.println(c1.compareTo(c2)); // 1
  c2.setTimeInMillis(1520702397755L);
  System.out.println(c1.compareTo(c2)); // -1
  ```

- `before()` — возвращает значение `true`, если текущий объект меньше `anotherDate`, и значение `false` — в противном случае. Формат метода:

  ```
  public boolean before(Object anotherDate)
  ```

Пример:

```
Calendar c1 = Calendar.getInstance();
Calendar c2 = Calendar.getInstance();
c1.setTimeInMillis(1520702397754L);
c2.setTimeInMillis(1520702397754L);
System.out.println(c1.before(c2));        // false
c2.setTimeInMillis(1520702397753L);
System.out.println(c1.before(c2));        // false
c2.setTimeInMillis(1520702397755L);
System.out.println(c1.before(c2));        // true
```

- `after()` — возвращает значение `true`, если текущий объект больше `anotherDate`, и значение `false` — в противном случае. Формат метода:

```
public boolean after(Object anotherDate)
```

Пример:

```
Calendar c1 = Calendar.getInstance();
Calendar c2 = Calendar.getInstance();
c1.setTimeInMillis(1520702397754L);
c2.setTimeInMillis(1520702397754L);
System.out.println(c1.after(c2));         // false
c2.setTimeInMillis(1520702397753L);
System.out.println(c1.after(c2));         // true
c2.setTimeInMillis(1520702397755L);
System.out.println(c1.after(c2));         // false
```

8.3. Класс *GregorianCalendar*: реализация григорианского и юлианского календарей

Класс `GregorianCalendar` реализует григорианский и юлианский календари. С помощью этого класса можно получить доступ к отдельным компонентам даты и времени, а также выполнить различные манипуляции с датой и временем. Прежде чем использовать этот класс, необходимо его импортировать с помощью инструкции:

```
import java.util.GregorianCalendar;
```

8.3.1. Создание экземпляра класса *GregorianCalendar*

Для создания экземпляра класса `GregorianCalendar` предназначены следующие конструкторы:

```
GregorianCalendar()
GregorianCalendar(int year, int month, int day)
GregorianCalendar(int year, int month, int day,
                  int hour, int minute)
```

```
GregorianCalendar(int year, int month, int day,
                  int hour, int minute, int second)
GregorianCalendar(Locale aLocale)
GregorianCalendar(TimeZone zone)
GregorianCalendar(TimeZone zone, Locale aLocale)
```

Первый конструктор без параметров позволяет создать объект с текущим временем в соответствии с локалью и часовым поясом по умолчанию:

```
GregorianCalendar c = new GregorianCalendar();
System.out.printf("%tc\n", c);
// сб мар. 10 23:14:47 MSK 2018
```

Второй, третий и четвертый конструкторы позволяют задать дату и время. Обратите внимание: нумерация месяцев начинается с `0`, и, чтобы не ошибиться, лучше использовать константы месяцев из класса `Calendar`:

```
GregorianCalendar c =
   new GregorianCalendar(2018, Calendar.APRIL, 24);
System.out.printf("%tc\n", c);
// вт апр. 24 00:00:00 MSK 2018
c = new GregorianCalendar(2018, Calendar.APRIL, 24, 20, 50);
System.out.printf("%tc\n", c);
// вт апр. 24 20:50:00 MSK 2018
c = new GregorianCalendar(2018, Calendar.APRIL, 24, 20, 50, 11);
System.out.printf("%tc\n", c);
// вт апр. 24 20:50:11 MSK 2018
```

Пятый, шестой и седьмой конструкторы позволяют указать локаль и (или) часовой пояс. Если локаль не указана явным образом, то используются настройки локали по умолчанию. Если часовой пояс не указан, то настройки берутся из локали:

```
// import java.util.SimpleTimeZone;
GregorianCalendar c =
    new GregorianCalendar(new Locale("ru", "RU"));
System.out.printf("%tc\n", c);
// сб мар. 10 23:18:48 MSK 2018
c = new GregorianCalendar(
    new SimpleTimeZone(3 * 60 * 60 * 1000, "Europe/Moscow"));
System.out.printf("%tc\n", c);
// сб мар. 10 23:18:48 MSK 2018
```

8.3.2. Установка и получение компонентов даты и времени

Класс `GregorianCalendar` наследует класс `Calendar`, а следовательно, и все его методы. Для установки компонентов даты и времени используются те же самые методы `setTime()`, `setTimeInMillis()` и `set()`, а для очистки — метод `clear()` (подробное описание этих методов приведено в *разд. 8.2.3*). Пример установки значений компонентов даты и времени содержится в листинге 8.5.

Листинг 8.5. Установка значений компонентов даты и времени

```
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.Locale;

public class MyClass {
   public static void main(String[] args) {
      Locale.setDefault(new Locale("ru", "RU"));
      GregorianCalendar c = new GregorianCalendar();
      c.setTime(new Date(1520702397754L));
      System.out.printf("%tc\n", c);
      // сб мар. 10 20:19:57 MSK 2018
      c.setTimeInMillis(1520705497754L);
      System.out.printf("%tc\n", c);
      // сб мар. 10 21:11:37 MSK 2018
      c.clear();
      System.out.printf("%tc\n", c);
      // чт янв. 01 00:00:00 MSK 1970
      c.set(2019, Calendar.APRIL, 24);
      System.out.printf("%tc\n", c);
      // ср апр. 24 00:00:00 MSK 2019
      c.set(2018, Calendar.APRIL, 24, 11, 5);
      System.out.printf("%tc\n", c);
      // вт апр. 24 11:05:00 MSK 2018
      c.set(2018, Calendar.APRIL, 24, 11, 5, 30);
      System.out.printf("%tc\n", c);
      // вт апр. 24 11:05:30 MSK 2018
      c.set(Calendar.YEAR, 2020);
      System.out.printf("%tc\n", c);
      // пт апр. 24 11:05:30 MSK 2020
   }
}
```

Для получения значений компонентов даты и времени используются те же самые методы: `getTime()`, `getTimeInMillis()`, `get()` и `getDisplayName()` (подробное описание этих методов приведено в *разд. 8.2.2*). Пример получения значений компонентов даты и времени содержится в листинге 8.6.

Листинг 8.6. Получение значений компонентов даты и времени

```
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.Locale;
```

```
public class MyClass {
   public static void main(String[] args) {
      Locale ru_RU = new Locale("ru", "RU");
      Locale.setDefault(ru_RU);
      GregorianCalendar c = new GregorianCalendar();
      Date d = c.getTime();
      System.out.printf("%tc\n", d);
      // сб мар. 10 23:24:57 MSK 2018
      long t = c.getTimeInMillis();
      System.out.println(t);
      // 1520713497698
      c.setTimeInMillis(1520713497698L);
      System.out.printf(
         "%1$td.%1$tm.%1$tY %1$tH:%1$tM:%1$tS\n", c);
      // 10.03.2018 23:24:57
      System.out.println(c.get(Calendar.YEAR));          // 2018
      System.out.println(c.get(Calendar.MONTH));         // 2
      System.out.println(c.get(Calendar.DAY_OF_MONTH)); // 10
      System.out.println(c.get(Calendar.HOUR_OF_DAY));   // 23
      System.out.println(c.get(Calendar.MINUTE));        // 24
      System.out.println(c.get(Calendar.SECOND));        // 57
      System.out.println(
         c.getDisplayName(Calendar.MONTH,
             Calendar.LONG_STANDALONE, ru_RU));          // март
   }
}
```

Для установки и получения компонентов даты и времени можно также воспользоваться следующими методами:

- `getTimeZone()` — получает объект с настройками часового пояса. Формат метода:

  ```
  public TimeZone getTimeZone()
  ```

- `setTimeZone()` — позволяет задать настройки часового пояса. Формат метода:

  ```
  public void setTimeZone(TimeZone zone)
  ```

- `getWeeksInWeekYear()` — возвращает количество недель в году. Формат метода:

  ```
  public int getWeeksInWeekYear()
  ```

  Пример:

  ```
  GregorianCalendar c = new GregorianCalendar(2018, 3, 24);
  System.out.println(c.getWeeksInWeekYear()); // 52
  ```

- `isLeapYear()` — возвращает значение `true`, если год високосный, и `false` — в противном случае. Формат метода:

  ```
  public boolean isLeapYear(int year)
  ```

Пример:

```
GregorianCalendar c = new GregorianCalendar();
System.out.println(c.isLeapYear(2016)); // true
System.out.println(c.isLeapYear(2018)); // false
```

- `getFirstDayOfWeek()` — возвращает первый день недели. Для русской локали метод возвращает значение 2, что соответствует константе `Calendar.MONDAY` (понедельник). Формат метода:

```
public int getFirstDayOfWeek()
```

Пример:

```
GregorianCalendar c = new GregorianCalendar();
System.out.println(c.getFirstDayOfWeek()); // 2
System.out.println(Calendar.MONDAY);       // 2
```

8.3.3. Изменение компонентов даты и времени

Если мы используем метод `set()`, то изменяется значение только указанного компонента. Если необходимо прибавить или вычесть какой-либо период времени, то следует воспользоваться методом `add()`. Формат метода:

```
public void add(int field, int amount)
```

В первом параметре указываются те же самые константы, что и в методах `set()` и `get()` (см. *разд. 8.2.2*). Во втором параметре положительное значение задает прибавление периода, а отрицательное — вычитание периода. Добавим и вычтем один день:

```
GregorianCalendar c = new GregorianCalendar(2018, 3, 30);
// Прибавляем один день
c.add(Calendar.DAY_OF_MONTH, 1);
System.out.printf("%tc\n", c);
// вт мая 01 00:00:00 MSK 2018
// Вычитаем один день
c.add(Calendar.DAY_OF_MONTH, -1);
System.out.printf("%tc\n", c);
// пн апр. 30 00:00:00 MSK 2018
```

Как видно из примера, добавление одного дня к дате 30.04.2018 изменило дату абсолютно правильно — на 01.05.2018, а не просто прибавило день к текущему значению компонента.

Вместо метода `add()` можно воспользоваться методом `roll()`, который также добавляет или вычитает период, но при этом не изменяет значение старших компонентов. Формат метода:

```
public void roll(int field, int amount)
```

Давайте рассмотрим различие между методами `add()` и `roll()` на примере добавления 11 месяцев к дате 31.03.2018:

```
GregorianCalendar c = new GregorianCalendar(2018, 2, 31);
// Прибавляем 11 месяцев
c.add(Calendar.MONTH, 11);
System.out.printf("%tc\n", c);
// чт февр. 28 00:00:00 MSK 2019
c = new GregorianCalendar(2018, 2, 31);
// Прибавляем 11 месяцев
c.roll(Calendar.MONTH, 11);
System.out.printf("%tc\n", c);
// ср февр. 28 00:00:00 MSK 2018
```

При добавлении 11 месяцев к марту мы получили февраль. В феврале нет дня 31, поэтому значение было изменено на максимальное значение для дня этого месяца в году. Теперь посмотрите на год. Метод `add()` прибавил значение, и получилось 2019. Значение года в случае с методом `roll()` осталось прежним, т. к. этот метод не изменяет значение старших компонентов, а год старше месяца. Вот в этом и заключается различие между методами `add()` и `roll()`.

8.3.4. Сравнение объектов

Для сравнения двух объектов класса `GregorianCalendar` можно использовать те же самые методы: `equals()`, `compareTo()`, `before()` и `after()`, что и в классе `Calendar` (подробное описание этих методов приведено в *разд. 8.2.4*):

```
GregorianCalendar c1 = new GregorianCalendar();
GregorianCalendar c2 = new GregorianCalendar();
c1.setTimeInMillis(1520702397754L);
c2.setTimeInMillis(1520702397754L);
if (c1.equals(c2))
   System.out.println("Равны");            // Равны
System.out.println(c1.compareTo(c2));      // 0
c2.setTimeInMillis(1520702397753L);
System.out.println(c1.compareTo(c2));      // 1
c2.setTimeInMillis(1520702397755L);
System.out.println(c1.compareTo(c2));      // -1
c2.setTimeInMillis(1520702397755L);
System.out.println(c1.before(c2));         // true
System.out.println(c1.after(c2));          // false
```

8.4. Класс *SimpleDateFormat*: форматирование даты и времени

Для форматирования даты и времени, помимо методов `format()` из класса `String` и `printf()` из класса `PrintStream`, можно использовать класс `SimpleDateFormat`. Сначала необходимо импортировать этот класс с помощью инструкции:

```
import java.text.SimpleDateFormat;
```

Для создания объекта предназначены следующие конструкторы:

```
SimpleDateFormat()
SimpleDateFormat(String pattern)
SimpleDateFormat(String pattern, Locale locale)
SimpleDateFormat(String pattern, DateFormatSymbols formatSymbols)
```

Первый конструктор создает объект с настройками шаблона и локали по умолчанию. Строка шаблона будет иметь формат `"dd.MM.yy H:mm"`:

```
SimpleDateFormat df = new SimpleDateFormat();
System.out.println(df.format(new Date(1520702397754L)));
// 10.03.18, 20:19
```

Второй конструктор позволяет указать строку шаблона. Локаль используется по умолчанию:

```
SimpleDateFormat df = new SimpleDateFormat("dd.MM.yyyy");
System.out.println(df.format(new Date(1520702397754L)));
// 10.03.2018
```

Третий конструктор позволяет указать строку шаблона и локаль:

```
SimpleDateFormat df = new SimpleDateFormat("dd MMMM yyyy",
                                  new Locale("ru", "RU"));
System.out.println(df.format(new Date(1520702397754L)));
// 10 марта 2018
df = new SimpleDateFormat("dd MMMM yyyy",
                          new Locale("en", "US"));
System.out.println(df.format(new Date(1520702397754L)));
// 10 March 2018
```

Четвертый конструктор позволяет указать шаблон и экземпляр класса `DateFormatSymbols`. Для примера выведем названия месяцев на русском языке с большой буквы:

```
// import java.text.DateFormatSymbols;
DateFormatSymbols dFormat = new DateFormatSymbols();
dFormat.setMonths(new String[] {"Января", "Февраля",
   "Марта", "Апреля", "Мая", "Июня", "Июля", "Августа",
   "Сентября", "Октября", "Ноября", "Декабря", ""});
SimpleDateFormat df = new SimpleDateFormat("dd MMMM yyyy", dFormat);
System.out.println(df.format(new Date(1520702397754L)));
// 10 Марта 2018
```

Как видно из примеров, для создания форматированной строки на основе шаблона и даты можно использовать метод `format()`. Формат метода:

```
public final String format(Date date)
```

В строке шаблона могут быть указаны следующие значения:

- `G` — эра (например, н. э.);
- `y` или `yyyy` — год из четырех цифр (например, 2018);

- `yy` — последние две цифры года (от 00 до 99);
- `M` — номер месяца без предваряющего нуля (от 1 до 12);
- `MM` — номер месяца с предваряющим нулем (от 01 до 12);
- `MMM` — аббревиатура месяца в зависимости от настроек локали (например, янв.);
- `MMMM` — название месяца в зависимости от настроек локали (например, январь);
- `dd` — номер дня в месяце с предваряющим нулем (от 01 до 31);
- `d` — номер дня в месяце без предваряющего нуля (от 1 до 31);
- `w` — неделя в году без предваряющего нуля;
- `ww` — неделя в году с предваряющим нулем;
- `W` — неделя в месяце без предваряющего нуля;
- `WW` — неделя в месяце с предваряющим нулем;
- `D` — день с начала года (от 001 до 366);
- `F` — день недели в месяце без предваряющего нуля;
- `FF` — день недели в месяце с предваряющим нулем;
- `E` — аббревиатура дня недели в зависимости от настроек локали (например, пт);
- `EEEE` — название дня недели в зависимости от настроек локали (например, пятница);
- `H` — часы в 24-часовом формате без предваряющего нуля (от 0 до 23);
- `HH` — часы в 24-часовом формате с предваряющим нулем (от 00 до 23);
- `h` — часы в 12-часовом формате без предваряющего нуля (от 1 до 12);
- `hh` — часы в 12-часовом формате с предваряющим нулем (от 01 до 12);
- `k` — часы в 24-часовом формате без предваряющего нуля (от 1 до 24);
- `K` — часы в 12-часовом формате без предваряющего нуля (от 0 до 11);
- `m` — минуты без предваряющего нуля (от 0 до 59);
- `mm` — минуты с предваряющим нулем (от 00 до 59);
- `s` — секунды без предваряющего нуля (от 0 до 59);
- `ss` — секунды с предваряющим нулем (от 00 до 59);
- `a` — AM или PM в верхнем регистре (например, AM);
- `S` — миллисекунды (например, 571);
- `z` — название часового пояса (например, MSK);
- `Z` — смещение часового пояса (например, +0300).

Если в строке шаблона мы хотим указать какой-либо текст, то его необходимо заключить в апострофы:

```
"'year:' yyyy"
```

Результат:

```
year: 2018
```

Чтобы указать символ апострофа, его нужно удвоить:

```
"'year:' '' yyyy ''"
```

Результат:

```
year: ' 2018 '
```

Выведем текущие дату и время таким образом, чтобы день недели и месяц были написаны по-русски (листинг 8.7). Кроме того, вместо класса `Date` используем класс `GregorianCalendar`.

Листинг 8.7. Класс `SimpleDateFormat`: вывод текущих даты и времени

```
import java.text.SimpleDateFormat;
import java.util.GregorianCalendar;
import java.util.Locale;

public class MyClass {
   public static void main(String[] args) {
     GregorianCalendar c = new GregorianCalendar();
     SimpleDateFormat df = new SimpleDateFormat(
                 "'Сегодня:' EEEE dd MMMM yyyy HH:mm",
                 new Locale("ru", "RU"));
     System.out.println(df.format(c.getTime()));
   }
}
```

Примерный результат:

```
Сегодня: пятница 23 марта 2018 11:05
```

8.5. Класс *DateFormatSymbols*: получение (задание) названий компонентов даты на языке, соответствующем указанной локали

Класс `DateFormatSymbols` позволяет получить (или задать) названия компонентов даты на языке, соответствующем указанной локали. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.text.DateFormatSymbols;
```

Создать экземпляр класса `DateFormatSymbols` позволяют следующие конструкторы:

```
DateFormatSymbols()
DateFormatSymbols(Locale locale)
```

Первый конструктор использует настройки локали по умолчанию, а второй — позволяет указать локаль явным образом:

```
DateFormatSymbols dFormat1 = new DateFormatSymbols();
DateFormatSymbols dFormat2 = new DateFormatSymbols(
                             new Locale("en", "US"));
```

Для создания объекта можно также воспользоваться методом `getInstance()`. Форматы метода:

```
public static final DateFormatSymbols getInstance()
public static final DateFormatSymbols getInstance(Locale locale)
```

Первый формат использует настройки локали по умолчанию, а второй — позволяет указать локаль явным образом:

```
DateFormatSymbols dFormat1 = DateFormatSymbols.getInstance();
DateFormatSymbols dFormat2 = DateFormatSymbols.getInstance(
                             new Locale("en", "US"));
```

Получить различные значения позволяют следующие методы:

- `getShortWeekdays()` — возвращает массив с сокращенными названиями дней недели. Формат метода:

  ```
  public String[] getShortWeekdays()
  ```

  Пример:

  ```
  DateFormatSymbols dFormat1 = DateFormatSymbols.getInstance();
  DateFormatSymbols dFormat2 = DateFormatSymbols.getInstance(
                               new Locale("en", "US"));
  System.out.println(Arrays.toString(dFormat1.getShortWeekdays()));
  System.out.println(Arrays.toString(dFormat2.getShortWeekdays()));
  ```

  Результат:

  ```
  [, вс, пн, вт, ср, чт, пт, сб]
  [, Sun, Mon, Tue, Wed, Thu, Fri, Sat]
  ```

- `getWeekdays()` — возвращает массив с полными названиями дней недели. Формат метода:

  ```
  public String[] getWeekdays()
  ```

  Результат:

  ```
  [, воскресенье, понедельник, вторник, среда, четверг, пятница, суббота]
  [, Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday]
  ```

- `getShortMonths()` — возвращает массив с сокращенными названиями месяцев. Формат метода:

  ```
  public String[] getShortMonths()
  ```

  Результат:

  ```
  [янв., февр., мар., апр., мая, июн., июл., авг., сент., окт.,
   нояб., дек., ]
  [Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec, ]
  ```

- `getMonths()` — возвращает массив с полными названиями месяцев. Формат метода:

```
public String[] getMonths()
```

Результат:

```
[января, февраля, марта, апреля, мая, июня, июля, августа,
 сентября, октября, ноября, декабря, ]
[January, February, March, April, May, June, July, August,
 September, October, November, December, ]
```

- `getEras()` — возвращает массив с названиями эры. Формат метода:

```
public String[] getEras()
```

Результат:

```
[до н.э., н.э.]
[BC, AD]
```

- `getAmPmStrings()` — возвращает массив с сокращениями для обозначения времени до и после полудня. Формат метода:

```
public String[] getAmPmStrings()
```

Результат:

```
[ДП, ПП]
[AM, PM]
```

Задать новые значения позволяют следующие методы:

- `setShortWeekdays()` — задает новый массив с сокращенными названиями дней недели. Формат метода:

```
public void setShortWeekdays(String[] newShortWeekdays)
```

- `setWeekdays()` — задает новый массив с полными названиями дней недели. Формат метода:

```
public void setWeekdays(String[] newWeekdays)
```

- `setShortMonths()` — задает новый массив с сокращенными названиями месяцев. Формат метода:

```
public void setShortMonths(String[] newShortMonths)
```

- `setMonths()` — задает новый массив с полными названиями месяцев. Формат метода:

```
public void setMonths(String[] newMonths)
```

- `setEras()` — задает новый массив с названиями эры. Формат метода:

```
public void setEras(String[] newEras)
```

- `setAmPmStrings()` — задает новый массив с сокращениями для обозначения до и после полудня. Формат метода:

```
public void setAmPmStrings(String[] newAmpms)
```

Объект с измененными названиями мы можем передать конструктору класса `SimpleDateFormat`. Для примера выведем названия месяцев на русском языке с большой буквы, а название дня недели — на английском языке:

```
DateFormatSymbols dFormat = new DateFormatSymbols(
                            new Locale("en", "US"));
dFormat.setMonths(new String[] {"Января", "Февраля",
   "Марта", "Апреля", "Мая", "Июня", "Июля", "Августа",
   "Сентября", "Октября", "Ноября", "Декабря", ""});
SimpleDateFormat df = new SimpleDateFormat("EEEE dd MMMM yyyy",
                                           dFormat);
System.out.println(df.format(new Date(1520702397754L)));
// Saturday 10 Марта 2018
```

8.6. «Засыпание» программы и измерение времени ее выполнения

Метод `sleep()` из класса `Thread` прерывает выполнение потока на указанное количество миллисекунд. По истечении срока программа продолжит работу. Форматы метода:

```
public static void sleep(long millis)
                  throws InterruptedException
public static void sleep(long millis, int nanos)
                   throws InterruptedException
```

В качестве примера создадим имитацию выполнения какого-либо процесса, а также измерим время выполнения программы с помощью методов `currentTimeMillis()` и `nanoTime()` из класса `System` (листинг 8.8).

Листинг 8.8. «Засыпание» программы и измерение времени ее выполнения

```
public class MyClass {
   public static void main(String[] args) {
      long t1 = System.currentTimeMillis();
      long t2 = System.nanoTime();
      System.out.println("0%");
      for (int i = 5; i < 101; i += 5) {
         try {
            Thread.sleep(1000); // Имитация процесса
         }
         catch (InterruptedException e) {
            e.printStackTrace();
         }
         System.out.println(i + "%");
      }
```

```
        System.out.println(System.currentTimeMillis() - t1);
        System.out.println(System.nanoTime() - t2);
    }
}
```

В результате, с периодичностью в 1000 миллисекунд программа выводит строки с индикацией хода выполнения, и в самом конце мы получим два числа:

```
20014
20013709432
```

Первое число отображает разницу между двумя запросами системного времени, которые мы делали с помощью метода `currentTimeMillis()`. Метод возвращает количество миллисекунд, прошедших с 1 января 1970 года. Если мы вычтем первую метку из второй, то получим время выполнения программы в миллисекундах. Формат метода:

```
public static long currentTimeMillis()
```

Второе число является разницей между значениями, возвращаемыми методом `nanoTime()`. В этом случае возвращается значение в наносекундах, и оно не связано с системным временем. Формат метода:

```
public static long nanoTime()
```

С помощью методов `currentTimeMillis()` и `nanoTime()` мы можем измерять время выполнения фрагментов кода и на основании результата производить различные оптимизации. Например, написали мы один алгоритм, измерили, затем переписали его иначе, опять измерили. Сравнили два результата и оставили в программе самый эффективный алгоритм. С помощью этих методов можно также искать фрагменты кода, которые выполняются слишком медленно, и пытаться выполнить оптимизацию этих фрагментов, изменяя что-либо в программе.

ГЛАВА 9

Работа с датой и временем (в версиях Java SE 8 и выше)

Вы наверняка заметили несоответствие названий классов их предназначению в классическом способе работы с датой и временем. Класс `Date` содержит время, а не дату. Класс `Calendar` не является календарем. Форматирование даты и времени находится в пакете `java.text` и т. д. Чтобы исправить все проблемы классического способа, в Java SE 8 был добавлен новый способ для работы с датой и временем. В результате разработчики создали огромное количество классов, которые были размещены по пакетам `java.time`, `java.time.format`, `java.time.zone` и др.

Так как классов много, и они часто используются сообща, чтобы не плодить множество строк с импортом, можно импортировать из пакета сразу все классы. Для этого следует указать после названия пакета точку и символ `*`. Например, импортировать все названия классов из пакета `java.time` можно с помощью такой инструкции:

```
import java.time.*;
```

9.1. Класс *LocalDate*: дата

Класс `LocalDate` позволяет производить операции над датами. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.time.LocalDate;
```

С помощью констант `MAX` и `MIN` можно получить максимальное и минимальное значение:

```
System.out.println(LocalDate.MAX);
// +999999999-12-31
System.out.println(LocalDate.MIN);
// -999999999-01-01
```

9.1.1. Создание экземпляра класса *LocalDate*

Создать экземпляр класса `LocalDate` позволяют следующие статические методы:

- `now()` — создает объект с текущей системной датой. Формат метода:

```
public static LocalDate now()
```

Пример:

```
LocalDate d = LocalDate.now();
System.out.println(d);          // 2018-03-23
```

- `of()` — **позволяет указать в параметрах конкретную дату из года, месяца и дня. Форматы метода:**

```
public static LocalDate of(int year, int month, int day)
public static LocalDate of(int year, Month month, int day)
```

В первом формате в параметре `month` **указывается индекс месяца от** `1` **(январь) до** `12` **(декабрь):**

```
LocalDate d = LocalDate.of(2018, 4, 27);
System.out.println(d);          // 2018-04-27
```

Во втором формате месяц указывается в виде константы из перечисления `Month` **(**`JANUARY`, `FEBRUARY`, `MARCH`, `APRIL`, `MAY`, `JUNE`, `JULY`, `AUGUST`, `SEPTEMBER`, `OCTOBER`, `NOVEMBER` **или** `DECEMBER`**). Прежде чем использовать перечисление** `Month`**, необходимо выполнить его импорт с помощью инструкции:**

```
import java.time.Month;
```

Пример:

```
LocalDate d = LocalDate.of(2018, Month.APRIL, 27);
System.out.println(d);          // 2018-04-27
```

- `ofEpochDay()` — **создает объект на основе количества дней, прошедших с 1 января 1970 года. Формат метода:**

```
public static LocalDate ofEpochDay(long epochDay)
```

Пример:

```
LocalDate d = LocalDate.ofEpochDay(
              (new Date()).getTime() / (24*60*60*1000));
System.out.println(d);          // 2018-03-10
d = LocalDate.ofEpochDay(0);
System.out.println(d);          // 1970-01-01
```

- `ofYearDay()` — **создает объект на основе года и номера дня в году. Формат метода:**

```
public static LocalDate ofYearDay(int year, int dayOfYear)
```

Пример:

```
LocalDate d = LocalDate.ofYearDay(2018, 118);
System.out.println(d);          // 2018-04-28
```

- `ofInstant()` — **создает объект на основе экземпляра класса** `Instant` **(см. *разд. 9.4*) и информации о часовом поясе (см. *разд. 9.8*). Метод доступен, начиная с Java 9. Формат метода:**

```
import java.time.Instant;
import java.time.ZoneId;
public static LocalDate ofInstant(Instant instant, ZoneId zone)
```

Пример:

```
LocalDate d = LocalDate.ofInstant(Instant.now(),
                                  ZoneId.of("Europe/Moscow"));
System.out.println(d);          // 2018-03-10
```

- `parse()` — создает объект на основе строки. Форматы метода:

```
public static LocalDate parse(CharSequence text)
public static LocalDate parse(CharSequence text,
                              DateTimeFormatter formatter)
```

Первый вариант метода разбирает строку в соответствии с форматом:

```
<Год из 4-х цифр>-<месяц из 2-х цифр>-<день из 2-х цифр>
```

Пример:

```
LocalDate d = LocalDate.parse("2018-04-27");
System.out.println(d);          // 2018-04-27
```

Второй вариант метода позволяет дополнительно указать объект класса `DateTimeFormatter` с пользовательскими настройками формата. Прежде чем использовать класс `DateTimeFormatter`, необходимо выполнить его импорт с помощью инструкции:

```
import java.time.format.DateTimeFormatter;
```

Пример:

```
LocalDate d = LocalDate.parse("2018-04-27",
              DateTimeFormatter.ISO_LOCAL_DATE);
System.out.println(d);          // 2018-04-27
```

9.1.2. Установка и получение компонентов даты

Для установки и получения компонентов даты предназначены следующие методы:

- `withYear()` — устанавливает новое значение года. Формат метода:

```
public LocalDate withYear(int year)
```

- `withMonth()` — устанавливает новое значение месяца. Формат метода:

```
public LocalDate withMonth(int month)
```

- `withDayOfMonth()` — устанавливает новое значение дня. Формат метода:

```
public LocalDate withDayOfMonth(int dayOfMonth)
```

Пример:

```
LocalDate d = LocalDate.of(2018, 4, 27);
d = d.withYear(2019);
d = d.withMonth(7);
d = d.withDayOfMonth(28);
System.out.println(d);          // 2019-07-28
```

- `withDayOfYear()` — **устанавливает новые значения месяца и дня на основании номера дня в году. Формат метода:**

```
public LocalDate withDayOfYear(int dayOfYear)
```

Пример:

```
LocalDate d = LocalDate.of(2018, 4, 27);
d = d.withDayOfYear(119);
System.out.println(d);          // 2018-04-29
```

- `getYear()` — **возвращает год. Формат метода:**

```
public int getYear()
```

- `getMonthValue()` **и** `getMonth()` — **возвращают месяц. Форматы методов:**

```
public int getMonthValue()
public Month getMonth()
```

- `getDayOfMonth()` — **возвращает день в месяце. Формат метода:**

```
public int getDayOfMonth()
```

Пример:

```
LocalDate d = LocalDate.of(2018, 4, 27);
System.out.println(d.getYear());          // 2018
System.out.println(d.getMonthValue());    // 4
System.out.println(d.getDayOfMonth());    // 27
```

- `getDayOfYear()` — **возвращает номер дня в году. Формат метода:**

```
public int getDayOfYear()
```

Пример:

```
LocalDate d = LocalDate.of(2018, 4, 27);
System.out.println(d.getDayOfYear());    // 117
```

- `getDayOfWeek()` — **возвращает день недели в виде значения из перечисления** `DayOfWeek` **(содержит константы** `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` **и** `SUNDAY`**). Прежде чем использовать перечисление** `DayOfWeek`**, необходимо выполнить его импорт с помощью инструкции:**

```
import java.time.DayOfWeek;
```

Формат метода:

```
public DayOfWeek getDayOfWeek()
```

Пример:

```
LocalDate d = LocalDate.of(2018, 4, 27);
System.out.println(d.getDayOfWeek());    // FRIDAY
```

- `lengthOfMonth()` — **возвращает количество дней в месяце. Формат метода:**

```
public int lengthOfMonth()
```

Пример:

```
LocalDate d = LocalDate.of(2018, 4, 27);
System.out.println(d.lengthOfMonth());   // 30
```

- `lengthOfYear()` — возвращает количество дней в году. Формат метода:

```
public int lengthOfYear()
```

Пример:

```
LocalDate d = LocalDate.of(2018, 4, 27);
System.out.println(d.lengthOfYear());   // 365
```

- `isLeapYear()` — возвращает значение `true`, если год високосный, и `false` — в противном случае. Формат метода:

```
public boolean isLeapYear()
```

- `toEpochDay()` — возвращает количество дней, прошедших с 1 января 1970 года. Формат метода:

```
public long toEpochDay()
```

Пример:

```
LocalDate d = LocalDate.of(2018, 4, 27);
System.out.println(d.toEpochDay());   // 17648
```

9.1.3. Прибавление и вычитание значений

Для прибавления и вычитания значений предназначены следующие методы:

- `plusYears()` — прибавляет к дате указанное количество лет. Формат метода:

```
public LocalDate plusYears(long years)
```

- `plusMonths()` — прибавляет к дате указанное количество месяцев. Формат метода:

```
public LocalDate plusMonths(long months)
```

- `plusWeeks()` — прибавляет к дате указанное количество недель. Формат метода:

```
public LocalDate plusWeeks(long weeks)
```

- `plusDays()` — прибавляет к дате указанное количество дней. Формат метода:

```
public LocalDate plusDays(long days)
```

Пример:

```
LocalDate d = LocalDate.of(2018, 4, 27);
System.out.println(d.plusDays(4));   // 2018-05-01
System.out.println(d.plusWeeks(4));  // 2018-05-25
System.out.println(d.plusMonths(1)); // 2018-05-27
System.out.println(d.plusYears(4));  // 2022-04-27
```

- `minusYears()` — вычитает из даты указанное количество лет. Формат метода:

```
public LocalDate minusYears(long years)
```

- `minusMonths()` — вычитает из даты указанное количество месяцев. Формат метода:

```
public LocalDate minusMonths(long months)
```

- `minusWeeks()` — вычитает из даты указанное количество недель. Формат метода:

```
public LocalDate minusWeeks(long weeks)
```

- `minusDays()` — вычитает из даты указанное количество дней. Формат метода:

```
public LocalDate minusDays(long days)
```

Пример:

```
LocalDate d = LocalDate.of(2018, 4, 30);
System.out.println(d.minusDays(4));   // 2018-04-26
System.out.println(d.minusWeeks(4));  // 2018-04-02
System.out.println(d.minusMonths(2)); // 2018-02-28
System.out.println(d.minusYears(4));  // 2014-04-30
```

Обратите внимание: ни один метод не изменяет текущий объект. Все методы возвращают новый объект класса `LocalDate`.

9.1.4. Преобразование объекта класса *LocalDate* в объект класса *LocalDateTime*

С помощью следующих методов можно преобразовать объект класса `LocalDate` в объект класса `LocalDateTime`:

- `atStartOfDay()` — добавляет нулевые значения для времени и возвращает объект класса `LocalDateTime`. Формат метода:

```
public LocalDateTime atStartOfDay()
```

Пример:

```
LocalDate d = LocalDate.of(2018, 4, 30);
System.out.println(d.atStartOfDay());   // 2018-04-30T00:00
```

- `atTime()` — добавляет указанные значения в часах, минутах, секундах, наносекундах или в виде объекта класса `LocalTime` и возвращает объект класса `LocalDateTime`. Форматы метода:

```
public LocalDateTime atTime(int hour, int minute)
public LocalDateTime atTime(int hour, int minute, int second)
public LocalDateTime atTime(int hour, int minute, int second,
                            int nanoOfSecond)
public LocalDateTime atTime(LocalTime time)
```

Пример:

```
LocalDate d = LocalDate.of(2018, 4, 30);
System.out.println(d.atTime(10, 2));
// 2018-04-30T10:02
```

```
System.out.println(d.atTime(10, 2, 15));
// 2018-04-30T10:02:15
System.out.println(d.atTime(10, 2, 20, 25));
// 2018-04-30T10:02:20.000000025
System.out.println(d.atTime(LocalTime.of(20, 17)));
// 2018-04-30T20:17
```

9.1.5. Сравнение объектов

Сравнить два объекта класса `LocalDate` позволяют следующие методы:

- `equals()` — сравнивает две даты. Если даты равны, то метод возвращает значение `true`, в противном случае — значение `false`. Формат метода:

  ```
  public boolean equals(Object obj)
  ```

  Пример:

  ```
  LocalDate d1 = LocalDate.of(2018, 4, 30);
  LocalDate d2 = LocalDate.of(2018, 4, 30);
  LocalDate d3 = LocalDate.of(2018, 4, 29);
  System.out.println(d1.equals(d2)); // true
  System.out.println(d1.equals(d3)); // false
  ```

- `isEqual()` — сравнивает две даты. Если даты равны, то метод возвращает значение `true`, в противном случае — значение `false`. Формат метода:

  ```
  public boolean isEqual(ChronoLocalDate other)
  ```

- `compareTo()` — сравнивает две даты. Возвращает значение `0` — если даты равны, положительное значение — если текущий объект больше `other`, отрицательное значение — если текущий объект меньше `other`. Формат метода:

  ```
  public int compareTo(ChronoLocalDate other)
  ```

  Пример:

  ```
  LocalDate d1 = LocalDate.of(2018, 4, 30);
  LocalDate d2 = LocalDate.of(2018, 4, 30);
  LocalDate d3 = LocalDate.of(2018, 4, 29);
  System.out.println(d1.compareTo(d2)); // 0
  System.out.println(d1.compareTo(d3)); // 1
  d3 = LocalDate.of(2018, 5, 1);
  System.out.println(d1.compareTo(d3)); // -1
  ```

- `isBefore()` — возвращает значение `true`, если текущий объект меньше `other`, и значение `false` — в противном случае. Формат метода:

  ```
  public boolean isBefore(ChronoLocalDate other)
  ```

  Пример:

  ```
  LocalDate d1 = LocalDate.of(2018, 4, 30);
  LocalDate d2 = LocalDate.of(2018, 4, 30);
  ```

```
LocalDate d3 = LocalDate.of(2018, 4, 29);
System.out.println(d1.isBefore(d2)); // false
System.out.println(d1.isBefore(d3)); // false
d3 = LocalDate.of(2018, 5, 1);
System.out.println(d1.isBefore(d3)); // true
```

- `isAfter()` — возвращает значение `true`, если текущий объект больше `other`, и значение `false` — в противном случае. Формат метода:

```
public boolean isAfter(ChronoLocalDate other)
```

Пример:

```
LocalDate d1 = LocalDate.of(2018, 4, 30);
LocalDate d2 = LocalDate.of(2018, 4, 30);
LocalDate d3 = LocalDate.of(2018, 4, 29);
System.out.println(d1.isAfter(d2)); // false
System.out.println(d1.isAfter(d3)); // true
d3 = LocalDate.of(2018, 5, 1);
System.out.println(d1.isAfter(d3)); // false
```

9.1.6. Преобразование даты в строку

Преобразовать объект класса `LocalDate` в строку позволяют следующие методы:

- `toString()` — преобразует объект в строку, имеющую формат:

```
<Год из 4-х цифр>-<месяц из 2-х цифр>-<день из 2-х цифр>
```

Формат метода:

```
public String toString()
```

- `format()` — преобразует объект в строку, соответствующую формату, заданному с помощью класса `DateTimeFormatter`. Формат метода:

```
public String format(DateTimeFormatter formatter)
```

Пример:

```
LocalDate d = LocalDate.of(2018, 4, 30);
System.out.println(d.toString()); // 2018-04-30
System.out.println(
   d.format(DateTimeFormatter.ofPattern("dd.MM.uuuu")));
   // 30.04.2018
```

9.1.7. Создание календаря на месяц и год

К этому моменту мы имеем все необходимые данные для создания календаря. С помощью класса `LocalDate` мы можем получить и количество дней в месяце, и день недели. Давайте потренируемся в программировании и выведем в окне консоли календарь на один месяц, а также календарь на весь год. Все эти действия мы реализуем в виде методов `printCalendarOnMonth()` (календарь на месяц) и

`printCalendarOnYear()` (календарь на год). Прежде чем рассматривать реализацию этих методов, создадим несколько вспомогательных методов:

- `getNameMonths()` — внутри этого метода вернем массив строк с названиями месяцев на русском языке (листинг 9.1);
- `getNameWeeks()` — этот метод будет возвращать массив строк с сокращенными названиями дней недели на русском языке (листинг 9.2);
- `onCenter()` — внутри этого метода сделаем так, чтобы названия месяцев выводились по центру блока (листинг 9.3). Причем, мы сделаем два одноименных метода, но с разным количеством параметров, чтобы можно было изменять символ-заполнитель (по умолчанию таким символом будет пробел).

Листинг 9.1. Метод `getNameMonths()`

```
public static String[] getNameMonths() {
   return new String[] {"Январь", "Февраль", "Март",
           "Апрель", "Май", "Июнь", "Июль", "Август",
           "Сентябрь", "Октябрь", "Ноябрь", "Декабрь"};
}
```

Листинг 9.2. Метод `getNameWeeks()`

```
public static String[] getNameWeeks() {
   return new String[] {"Пн", "Вт", "Ср", "Чт",
                        "Пт", "Сб", "Вс"};
}
```

Реализация методов `getNameMonths()` и `getNameWeeks()` очень простая. Во-первых, оба метода являются статическими (об этом говорит ключевое слово `static`). Благодаря этому, мы можем использовать методы без создания экземпляра класса. Во-вторых, оба метода возвращают массив строк (об этом говорит тип `String[]` после ключевого слова `static`). Методы не принимают никаких параметров, поэтому после названия методов указываются пустые круглые скобки. При объявлении методов был указан тип возвращаемого значения, поэтому внутри метода мы обязаны вернуть массив строк с помощью оператора `return`. Чтобы не плодить лишних переменных, мы создаем анонимный массив, заполняем значениями и сразу возвращаем его из метода. Разместите эти методы внутри блока класса сразу за блоком метода `main()`. Обратите внимание: не внутри блока метода `main()`, а сразу после закрывающей фигурной скобки, но обязательно внутри блока класса.

Давайте попробуем вызвать эти методы из блока метода `main()`. Для вызова метода необходимо указать его имя и после имени внутри круглых скобок передать какие-либо значения. Наши методы не принимают никаких параметров, поэтому указываем пустые круглые скобки. Так как методы возвращают массив строк, необходимо создать переменную с тем же самым типом:

```
String[] arr = getNameMonths();
System.out.println(arr[0]); // Январь
```

Здесь мы объявили переменную `arr`, имеющую тип `String[]`, и присвоили ей значение, возвращаемое методом `getNameMonths()`. Так как мы имеем дело с массивом, то для доступа к отдельному его элементу необходимо указать индекс этого элемента внутри квадратных скобок. В приведенном примере мы вывели значение первого элемента массива. Еще раз напомню, что нумерация элементов массива начинается с `0`.

Итак, когда выполнение программы доходит до места вызова метода, то происходит передача управления инструкциям внутри метода. Как только внутри метода встречается оператор `return`, управление передается обратно в точку вызова метода, и результат выполнения метода становится доступен для дальнейшей работы. Все удобство работы с методами заключается в возможности многократного использования кода, расположенного внутри метода. Метод имеет имя, по которому к нему можно обратиться, и может быть расположен в любом месте блока класса или вообще в другом классе.

Листинг 9.3. Методы `onCenter()`

```
public static String onCenter(String s, int length, char ch) {
   if (s == null) return "";
   int sLength = s.length();
   if (length <= 0 || sLength == 0) return "";
   if (sLength == length) return s;
   if (sLength > length) return s.substring(0, length);
   int start = (length - sLength) / 2;
   int end = length - start - sLength;
   char[] arrStart = new char[start];
   char[] arrEnd = new char[end];
   Arrays.fill(arrStart, ch);
   Arrays.fill(arrEnd, ch);
   return String.valueOf(arrStart) + s +
          String.valueOf(arrEnd);
}

public static String onCenter(String s, int length) {
   return onCenter(s, length, ' ');
}
```

Статические методы `onCenter()` мы будем использовать для выравнивания текста по центру внутри строки с заданной длиной. Чтобы можно было задействовать разные символы-заполнители, мы создали два метода с одинаковыми именами, но разным количеством параметров. Такая реализация называется *перегрузкой методов*. Посмотрим на форматы метода:

```
public static String onCenter(String s, int length, char ch)
public static String onCenter(String s, int length)
```

Первый параметр задает текст, который нужно вывести по центру строки, длиной, указанной во втором параметре. Отличие между методами состоит в третьем параметре, с помощью которого указывается символ-разделитель. Если этот параметр не указан, то вызывается метод, у которого два параметра. Внутри этого метода мы просто вызываем одноименный метод с тремя параметрами, дополнительно указывая в третьем параметре символ пробела, и результат возвращаем с помощью оператора `return`. Давайте попробуем произвести выравнивание:

```
System.out.println("'" + onCenter("text", 20) + "'");
// '        text        '
```

Символы апострофов мы добавили только для того, чтобы были видны границы области, внутри которой производится выравнивание текста. Как видите, спереди и сзади текста здесь добавлены пробелы таким образом, чтобы текст был выведен примерно по центру. Если мы укажем символ-разделитель явным образом, то будет вызван метод с тремя параметрами. При этом метод с двумя параметрами вызван не будет. Давайте укажем символ `*`:

```
System.out.println("'" + onCenter("text", 20, '*') + "'");
// '********text********'
```

Обратите внимание на то, что третий параметр имеет тип `char`, следовательно, мы должны указать символ внутри апострофов. Не путайте символ внутри апострофов и символ внутри кавычек! В первом случае мы создадим символ, а во втором случае — строку из одного символа. Это разные типы данных.

Внутри метода `onCenter()` с тремя параметрами сначала выполняются проверки:

- если вместо строки переменная `s` содержит значение `null`, то возвращаем пустую строку с помощью оператора `return`. Как только внутри метода встретилась первая инструкция `return`, управление сразу передается в точку вызова метода, и остальные инструкции внутри метода не выполняются. Любая объектная переменная может иметь специальное значение `null`, означающее, что переменная не ссылается ни на какой объект. Это значение строковая переменная может получить, если она не была инициализирована явным образом;
- если длина строки меньше или равна нулю или длина текста равна нулю, то возвращаем пустую строку;
- если длина текста равна длине строки, то просто возвращаем текст. Добавлять какие-либо символы просто некуда;
- если длина текста больше длины строки, то с помощью метода `substring()` обрезаем ее до длины строки и возвращаем фрагмент.

Нужны ли эти проверки? Первые два случая обычно возникают при ошибках в остальной части программы. После обработки этих ошибок внутри метода проблем не возникнет, однако не мешало бы указать на эти ошибки явным образом, а не исправлять их. Следуя обычной практике, для первых двух случаев надо было бы сгенерировать исключение и прервать выполнение программы. Как говорится, явное лучше неявного. С другой стороны, если ошибку совершил не программист, а пользователь, введя неверное значение, то лучше либо обработать его ошибки,

либо вывести описание ошибки на человеческом языке. Ведь когда мы сгенерируем исключение, виртуальная машина выведет стандартное описание, которое пользователю ни о чем не скажет. Тогда он просто отругает вас за непрофессионализм разработчика, как ему будет казаться, и, вполне возможно, больше не станет пользоваться вашей программой.

Проверка в двух последних случаях избавляет нас от лишних действий. Ведь если длина текста равна длине строки, то и делать-то больше нечего. Цель достигнута. Просто возвращаем текст целиком и выходим из метода. В последнем случае, когда длина текста больше длины строки, мы тоже ничего добавить не можем. Однако тут есть различные пути развития ситуации. Во-первых, можно поступить, как и было сделано, — просто обрезать текст до длины строки. Чаще всего ширина области задана строго, и текст большей длины все равно не поместится в эту область. Во-вторых, можно просто вернуть весь текст. Тут уже логика зависит от разработчика. Например, метод `printf()` возвращает весь текст, если он не помещается в области:

```
System.out.println("'" + onCenter("text", 2) + "'");
// 'te'
System.out.printf("'%2s'", "text");
// 'text'
```

Следующие две инструкции:

```
int start = (length - sLength) / 2;
int end = length - start - sLength;
```

вычисляют, какое количество символов-заполнителей нужно добавить в начало (переменная `start`) и конец строки (переменная `end`), чтобы текст располагался примерно посередине строки. После этого можно создать два символьных массива, указав их размер с помощью переменных `start` и `end`. Далее с помощью метода `fill()` из класса `Arrays` мы заполняем эти массивы символом-заполнителем, указанным в третьем параметре при вызове метода. Обратите внимание: прежде чем использовать класс `Arrays`, необходимо добавить в начало программы следующую инструкцию:

```
import java.util.Arrays;
```

Затем мы преобразуем символьные массивы в строку (с помощью метода `valueOf()` из класса `String`), путем конкатенации строк формируем итоговую строку и возвращаем ее с помощью оператора `return`.

Теперь рассмотрим метод `printCalendarOnMonth()`, позволяющий вывести календарь на указанный месяц в указанном году (листинг 9.4).

Листинг 9.4. Метод `printCalendarOnMonth()`

```
public static void printCalendarOnMonth(
       int year, int month, LocalDate currDay) {
   if (currDay == null || month < 1 || month > 12)
      return;
```

```
    LocalDate d = LocalDate.of(year, month, 1);
    // Выводим название месяца и год
    String[] nameMonths = getNameMonths();
    System.out.println(onCenter(
       nameMonths[d.getMonthValue() - 1], 28));
    // Выводим названия дней недели
    String[] nameWeek = getNameWeeks();
    for (int i = 0; i < nameWeek.length; i++) {
       System.out.print(onCenter(nameWeek[i], 4));
    }
    // Формируем отступ для первой строки
    int indent = 0;
    DayOfWeek firstDayOfWeek = DayOfWeek.MONDAY;
    LocalDate d2 = d.withDayOfMonth(1);
    DayOfWeek currDayOfWeek = d2.getDayOfWeek();
    while (firstDayOfWeek != currDayOfWeek) {
       indent++;
       d2 = d2.minusDays(1);
       currDayOfWeek = d2.getDayOfWeek();
    }
    if (indent != 0) System.out.println();
    for (int i = 0; i < indent; i++) {
       System.out.print("    ");
    }
    // Выводим числа месяца
    for (int i = 1, j = d.lengthOfMonth() + 1; i < j; i++) {
       // Если текущий день недели равен первому дню,
       // то вставляем символ перевода строки
       if (d.withDayOfMonth(i).getDayOfWeek()
            == firstDayOfWeek) System.out.println();
       // Выводим число
       System.out.printf("%3d", i);
       // Если текущий день, то помечаем его символом *
       if (d.withDayOfMonth(i).equals(currDay))
            System.out.print("*");
       else System.out.print(" ");
    }
    System.out.println();
}
```

Давайте взглянем на формат метода:

```
public static void printCalendarOnMonth(
                  int year, int month, LocalDate currDay)
```

Как и все другие методы, он является статическим. Ключевое слово `void` означает, что метод не возвращает никакого значения. В качестве параметров метод принимает год, месяц (число от `1` до `12`) и объект класса `LocalDate`, содержащий инфор-

мацию о дате. Эту информацию мы будем использовать, чтобы пометить этот день в календаре с помощью символа `*`. Почему не просто число, а именно объект? Дело в том, что метод может принимать произвольные значения года и месяца. Если мы укажем просто число, то оно будет отмечено во всех месяцах. Как вы увидите далее, мы воспользуемся этим же методом для формирования календаря на год. Объект класса `LocalDate` содержит полную информацию о дате, включая и день, и месяц, и год. Поэтому будет отмечен именно день, который совпадает со всеми тремя значениями. Это может быть текущий день или просто день рождения вашего друга.

Далее идет опять проверка входящих значений. Если объектная переменная не содержит значения, или значение месяца меньше `1`, или значение месяца больше `12`, то мы возвращаем значение с помощью оператора `return`. Постойте, скажете вы. При чем здесь оператор `return`, мы ведь объявили, что метод ничего не возвращает? Ну, так мы же ничего и не возвращаем. Инструкция ведь не содержит значения после ключевого слова `return`. В этом случае оператор `return` используется для выхода из метода и передачи управления инструкции, следующей за вызовом метода `printCalendarOnMonth()`.

Следующая инструкция создает объект класса `LocalDate` на основе года, месяца и первого числа. Обратите внимание: прежде чем использовать класс `LocalDate`, необходимо в начале программы указать, например, такую инструкцию:

```
import java.time.*;
```

Символ `*` означает, что мы импортируем все идентификаторы из пакета, в том числе и класс `LocalDate`. При желании можно импортировать только класс `LocalDate` с помощью инструкции:

```
import java.time.LocalDate;
```

но тогда придется импортировать еще и перечисление `DayOfWeek` с помощью инструкции:

```
import java.time.DayOfWeek;
```

Импорт всех идентификаторов удобен, т. к. при этом не придется импортировать каждый класс по отдельности, но, с другой стороны, это приводит к засорению пространства имен не используемыми идентификаторами, что может привести к конфликту имен.

С помощью следующей инструкции:

```
String[] nameMonths = getNameMonths();
```

мы получаем массив с названиями месяцев из метода `getNameMonths()`, который мы уже рассматривали ранее.

Далее мы определяем название месяца по индексу, используя метод `getMonthValue()` для получения номера месяца. Метод возвращает число от `1` до `12`, а массив нумеруется с `0` до `11`, поэтому из значения вычитается единица. Затем необходимо выполнить выравнивание названия месяца по центру области шириной 28 символов

(7 дней недели по 4 символа на дату). Для этого нам и пригодится метод `onCenter()`, который мы уже рассмотрели ранее.

Название месяца мы вывели, теперь нужно вывести сокращенные названия дней недели. Для этого получаем массив с названиями при помощи метода `getNameWeeks()`, который мы также рассмотрели ранее. Далее внутри цикла `for` перебираем все элементы массива и выводим названия с помощью метода `print()`. Здесь нам опять пригодился метод `onCenter()`. Обратили внимание, сколько раз мы уже использовали метод `onCenter()`? Думаете всего два раза? Нет. Мы обращаемся к этому методу только внутри цикла аж семь раз. Если вызов метода заменить телом метода, а цикл развернуть по отдельному шагу, то наша программа превратится в ужасного монстра. Однако с помощью методов и циклов мы можем сокращать лишний код. Как только вы видите, что один и тот же код используется несколько раз, сразу стоит задуматься о вынесении этого кода в отдельный метод. Как говорится: разделяй и властвуй.

Название дней недели вывели, теперь можно начать выводить дни. Тут сразу же возникнет проблема. Далеко не во всех месяцах первый день месяца совпадает с первым днем недели, в нашем случае — с понедельником. Нам ведь нужно вывести числа так, чтобы они совпадали с соответствующими названиями дней недели. Для этого сначала необходимо рассчитать отступ, который следует добавить перед выводом первого числа.

Чтобы рассчитать этот отступ, сначала объявляем целочисленную переменную `indent` и присваиваем ей начальный отступ, равный нулю. Далее объявляем переменную `firstDayOfWeek`, которая будет содержать значение из перечисления `DayOfWeek`, соответствующее первому дню недели. В нашем случае это будет значение `MONDAY`, соответствующее понедельнику. Следующая инструкция создает новый объект класса `LocalDate` с установленным днем на начало месяца. Затем объявляем переменную `currDayOfWeek` и сохраняем в ней название дня недели, соответствующее первому числу месяца. Мы не знаем заранее, где нужно остановиться, поэтому вместо цикла `for` будем использовать цикл `while`. На каждой итерации цикла мы станем прибавлять единицу к значению переменной `indent` и сдвигать дату на один день назад с помощью метода `minusDays()`. Цикл будет продолжаться, пока название первого дня недели не совпадет с названием дня недели в переменной `currDayOfWeek`. Таким образом, если первое число совпадет с понедельником, то цикл не станет выполняться совсем, и переменная `indent` будет содержать начальное значение. В противном случае переменная будет содержать значение отступа. С помощью цикла `for` выводим отступ.

Следующий цикл `for` выводит числа от `1` до значения, возвращаемого методом `lengthOfMonth() + 1`. Чтобы этот метод не вызывался на каждой итерации цикла, мы присваиваем значение переменной `j` в первом блоке цикла `for` и сравнение производим со значением этой переменной. Если день недели для текущего дня совпадает с первым днем недели, то выводим символ перевода строки. Далее с помощью метода `printf()` выводим текущее значение дня в поле шириной `3` символа. Если дата совпадает с датой, указанной в третьем параметре, то выводим символ `*`, в противном случае — один пробел. Обратите внимание, что сравнение производится с помощью метода `equals()`, а не оператора `==`. Дело в том, что оператор `==`

стал бы сравнивать лишь ссылки на объекты, а не значения внутри объектов. И в заключение вставляем символ перевода строки.

Давайте вызовем метод `printCalendarOnMonth()` и посмотрим на результат:

```
printCalendarOnMonth(2018, 3, LocalDate.now());
```

Результат:

```
            Март
 Пн  Вт  Ср  Чт  Пт  Сб  Вс
                 1   2   3   4
  5   6   7   8   9  10  11*
 12  13  14  15  16  17  18
 19  20  21  22  23  24  25
 26  27  28  29  30  31
```

На этом мы не остановимся и выведем календарь на весь год (листинг 9.5).

Листинг 9.5. Метод `printCalendarOnYear()`

```
public static void printCalendarOnYear(
                    int year, LocalDate currDay) {
   System.out.println(
      onCenter("Календарь на " + year + " год", 28));
   for (int i = 1; i <= 12; i++) {
      System.out.println();
      printCalendarOnMonth(year, i, currDay);
   }
}
```

Метод `printCalendarOnYear()` до безобразия прост. В качестве параметров он получает год и объект с датой, которая будет помечена символом `*`. Далее внутри метода выводим название календаря, выравнивая его по центру с помощью метода `onCenter()` (опять он пригодился!). Внутри цикла `for` вызываем метод `printCalendarOnMonth()` и передаем ему год, индекс месяца и объект с датой. Никакой проверки корректности ввода не требуется, т. к. она выполняется внутри метода `printCalendarOnMonth()`. Давайте вызовем метод и посмотрим на календарь текущего года, в котором текущий день помечен символом `*`:

```
printCalendarOnYear(LocalDate.now().getYear(), LocalDate.now());
```

Весь календарь в книге мы приводить не станем, а покажем только его шапку и календарь на первый месяц:

```
   Календарь на 2018 год

            Январь
 Пн  Вт  Ср  Чт  Пт  Сб  Вс
  1   2   3   4   5   6   7
  8   9  10  11  12  13  14
 15  16  17  18  19  20  21
 22  23  24  25  26  27  28
 29  30  31
```

Чтобы сложилось целостное представление о программе, давайте выведем ее схему (листинг 9.6). Просто замените комментарии кодом соответствующего листинга.

Листинг 9.6. Вывод календаря на год

```
import java.time.*;
import java.util.Arrays;

public class MyClass {

   public static void main(String[] args) {
      printCalendarOnYear(LocalDate.now().getYear(),
                          LocalDate.now());
   }

   // Вывод календаря на весь год (листинг 9.5)

   // Вывод календаря на один месяц (листинг 9.4)

   // Выравнивание значения по центру (листинг 9.3)

   // Названия месяцев (листинг 9.1)

   // Сокращенные названия дней недели (листинг 9.2)
}
```

Напомню, что класс `LocalDate` доступен только начиная с Java 8, поэтому этот код не будет работать в предыдущих версиях языка. Попробуйте потренироваться и переделайте программу, используя класс `GregorianCalendar` вместо класса `LocalDate`.

9.2. Класс *LocalTime*: время

Класс `LocalTime` позволяет производить операции над временем. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.time.LocalTime;
```

С помощью констант `MAX` и `MIN` можно получить максимальное и минимальное значение:

```
System.out.println(LocalTime.MAX);
// 23:59:59.999999999
System.out.println(LocalTime.MIN);
// 00:00
```

9.2.1. Создание экземпляра класса *LocalTime*

Создать экземпляр класса `LocalTime` позволяют следующие статические методы:

- `now()` — создает объект с текущим локальным временем. Формат метода:

```
public static LocalTime now()
```

Пример:

```
LocalTime t = LocalTime.now();
System.out.println(t.toString()); // 12:04:36.253
```

- `of()` — создает объект на основе часов, минут, секунд и наносекунд. Форматы метода:

```
public static LocalTime of(int hour, int minute)
public static LocalTime of(int hour, int minute, int second)
public static LocalTime of(int hour, int minute, int second,
                           int nanoOfSecond)
```

Пример:

```
LocalTime t = LocalTime.of(12, 4);
System.out.println(t.toString()); // 12:04
t = LocalTime.of(12, 4, 36);
System.out.println(t.toString()); // 12:04:36
t = LocalTime.of(12, 4, 36, 253000000);
System.out.println(t.toString()); // 12:04:36.253
```

- `ofSecondOfDay()` — создает объект на основе количества секунд. Формат метода:

```
public static LocalTime ofSecondOfDay(long secondOfDay)
```

Пример:

```
LocalTime t = LocalTime.ofSecondOfDay(5 * 60 * 60);
System.out.println(t.toString()); // 05:00
```

- `ofNanoOfDay()` — создает объект на основе количества наносекунд. Формат метода:

```
public static LocalTime ofNanoOfDay(long nanoOfDay)
```

При указании значения обратите внимание на букву `L`. Если эту букву не указать, то по умолчанию будет использован тип `int`. Однако результат выражения может выходить за диапазон для типа `int`, в результате наступит переполнение, и значение станет отрицательным:

```
LocalTime t = LocalTime.ofNanoOfDay(
                    5 * 60 * 60 * 1_000_000_000L);
System.out.println(t.toString()); // 05:00
System.out.println(5 * 60 * 60 * 1_000_000_000);
// -207937536
System.out.println(5 * 60 * 60 * 1_000_000_000L);
// 18000000000000
```

- `ofInstant()` — создает объект на основе экземпляра класса `Instant` (см. *разд. 9.4*) и информации о часовом поясе (см. *разд. 9.8*). Метод доступен, начиная с Java 9. Формат метода:

```
import java.time.Instant;
import java.time.ZoneId;
public static LocalTime ofInstant(Instant instant,
                                  ZoneId zone)
```

Пример:

```
LocalTime t = LocalTime.ofInstant(Instant.now(),
                        ZoneId.of("Europe/Moscow"));
System.out.println(t.toString()); // 12:04:36.253
```

- `parse()` — создает объект из строки специального формата. Форматы метода:

```
import java.time.format.DateTimeFormatter;
public static LocalTime parse(CharSequence text)
public static LocalTime parse(CharSequence text,
                              DateTimeFormatter formatter)
```

Первый вариант метода использует формат времени по умолчанию:

```
LocalTime t = LocalTime.parse("22:12");
System.out.println(t.toString()); // 22:12
t = LocalTime.parse("22:12:03");
System.out.println(t.toString()); // 22:12:03
t = LocalTime.parse("22:12:03.789");
System.out.println(t.toString()); // 22:12:03.789
```

Второй вариант метода позволяет указать произвольный формат, заданный с помощью объекта класса `DateTimeFormatter`:

```
LocalTime t = LocalTime.parse("22.12",
              DateTimeFormatter.ofPattern("HH.mm"));
System.out.println(t.toString()); // 22:12
```

9.2.2. Установка и получение компонентов времени

Для установки и получения компонентов времени предназначены следующие методы:

- `withHour()` — устанавливает новое значение часа. Формат метода:

```
public LocalTime withHour(int hour)
```

- `withMinute()` — устанавливает новое значение минут. Формат метода:

```
public LocalTime withMinute(int minute)
```

- `withSecond()` — устанавливает новое значение секунд. Формат метода:

```
public LocalTime withSecond(int second)
```

- `withNano()` — устанавливает новое значение наносекунд. Формат метода:

```
public LocalTime withNano(int nanoOfSecond)
```

Пример:

```
LocalTime t = LocalTime.of(22, 12, 36);
t = t.withHour(11);
t = t.withMinute(28);
t = t.withSecond(5);
t = t.withNano(788000000);
System.out.println(t.toString()); // 11:28:05.788
```

- `getHour()` — возвращает часы. Формат метода:

```
public int getHour()
```

- `getMinute()` — возвращает минуты. Формат метода:

```
public int getMinute()
```

- `getSecond()` — возвращает секунды. Формат метода:

```
public int getSecond()
```

- `getNano()` — возвращает наносекунды. Формат метода:

```
public int getNano()
```

Пример:

```
LocalTime t = LocalTime.of(22, 12, 36, 788000000);
System.out.println(t.getHour());   // 22
System.out.println(t.getMinute()); // 12
System.out.println(t.getSecond()); // 36
System.out.println(t.getNano());   // 788000000
```

- `toSecondOfDay()` — возвращает количество секунд с начала дня. Формат метода:

```
public int toSecondOfDay()
```

Пример:

```
LocalTime t = LocalTime.of(22, 12, 36, 788000000);
System.out.println(t.toSecondOfDay());   // 79956
```

- `toNanoOfDay()` — возвращает количество наносекунд с начала дня. Формат метода:

```
public long toNanoOfDay()
```

Пример:

```
LocalTime t = LocalTime.of(22, 12, 36, 788000000);
System.out.println(t.toNanoOfDay()); // 79956788000000
```

9.2.3. Прибавление и вычитание значений

Для прибавления и вычитания значений предназначены следующие методы:

- `plusHours()` — прибавляет ко времени указанное количество часов. Формат метода:

```
public LocalTime plusHours(long hours)
```

- `plusMinutes()` — прибавляет ко времени указанное количество минут. Формат метода:

```
public LocalTime plusMinutes(long minutes)
```

- `plusSeconds()` — прибавляет ко времени указанное количество секунд. Формат метода:

```
public LocalTime plusSeconds(long seconds)
```

- `plusNanos()` — прибавляет ко времени указанное количество наносекунд. Формат метода:

```
public LocalTime plusNanos(long nanos)
```

Пример:

```
LocalTime t = LocalTime.of(22, 12, 36, 788000000);
System.out.println(t.plusHours(3));    // 01:12:36.788
System.out.println(t.plusMinutes(50)); // 23:02:36.788
System.out.println(t.plusSeconds(30)); // 22:13:06.788
System.out.println(
            t.plusNanos(500000000L)); // 22:12:37.288
```

- `minusHours()` — вычитает из времени указанное количество часов. Формат метода:

```
public LocalTime minusHours(long hours)
```

- `minusMinutes()` — вычитает из времени указанное количество минут. Формат метода:

```
public LocalTime minusMinutes(long minutes)
```

- `minusSeconds()` — вычитает из времени указанное количество секунд. Формат метода:

```
public LocalTime minusSeconds(long seconds)
```

- `minusNanos()` — вычитает из времени указанное количество наносекунд. Формат метода:

```
public LocalTime minusNanos(long nanos)
```

Пример:

```
LocalTime t = LocalTime.of(22, 12, 36, 788000000);
System.out.println(t.minusHours(26));   // 20:12:36.788
System.out.println(t.minusMinutes(50)); // 21:22:36.788
System.out.println(t.minusSeconds(30)); // 22:12:06.788
System.out.println(
            t.minusNanos(854000000L)); // 22:12:35.934
```

Обратите внимание: ни один метод не изменяет текущий объект. Все методы возвращают новый объект класса `LocalTime`.

9.2.4. Преобразование объекта класса *LocalTime* в объект класса *LocalDateTime*

Преобразовать объект класса `LocalTime` в объект класса `LocalDateTime` позволяет метод `atDate()`. Формат метода:

```
public LocalDateTime atDate(LocalDate date)
```

Пример:

```
LocalTime t = LocalTime.of(22, 12, 36);
LocalDate d = LocalDate.of(2018, 4, 30);
System.out.println(t.atDate(d)); // 2018-04-30T22:12:36
```

9.2.5. Сравнение объектов

Сравнить два объекта класса `LocalTime` позволяют следующие методы:

- `equals()` — сравнивает два объекта. Если объекты равны, то метод возвращает значение `true`, в противном случае — значение `false`. Формат метода:

  ```
  public boolean equals(Object obj)
  ```

  Пример:

  ```
  LocalTime t1 = LocalTime.of(22, 12, 36);
  LocalTime t2 = LocalTime.of(22, 12, 36);
  LocalTime t3 = LocalTime.of(22, 12, 10);
  System.out.println(t1.equals(t2)); // true
  System.out.println(t1.equals(t3)); // false
  ```

- `compareTo()` — сравнивает два объекта. Возвращает значение `0` — если объекты равны, положительное значение — если текущий объект больше `other`, отрицательное значение — если текущий объект меньше `other`. Формат метода:

  ```
  public int compareTo(LocalTime other)
  ```

  Пример:

  ```
  LocalTime t1 = LocalTime.of(22, 12, 36);
  LocalTime t2 = LocalTime.of(22, 12, 36);
  LocalTime t3 = LocalTime.of(22, 12, 10);
  System.out.println(t1.compareTo(t2)); // 0
  System.out.println(t1.compareTo(t3)); // 1
  t3 = LocalTime.of(22, 12, 55);
  System.out.println(t1.compareTo(t3)); // -1
  ```

- `isBefore()` — возвращает значение `true`, если текущий объект меньше `other`, и значение `false` — в противном случае. Формат метода:

  ```
  public boolean isBefore(LocalTime other)
  ```

  Пример:

  ```
  LocalTime t1 = LocalTime.of(22, 12, 36);
  LocalTime t2 = LocalTime.of(22, 12, 36);
  LocalTime t3 = LocalTime.of(22, 12, 10);
  ```

```
System.out.println(t1.isBefore(t2)); // false
System.out.println(t1.isBefore(t3)); // false
t3 = LocalTime.of(22, 12, 55);
System.out.println(t1.isBefore(t3)); // true
```

- `isAfter()` — возвращает значение `true`, если текущий объект больше `other`, и значение `false` — в противном случае. Формат метода:

```
public boolean isAfter(LocalTime other)
```

Пример:

```
LocalTime t1 = LocalTime.of(22, 12, 36);
LocalTime t2 = LocalTime.of(22, 12, 36);
LocalTime t3 = LocalTime.of(22, 12, 10);
System.out.println(t1.isAfter(t2)); // false
System.out.println(t1.isAfter(t3)); // true
t3 = LocalTime.of(22, 12, 55);
System.out.println(t1.isAfter(t3)); // false
```

9.2.6. Преобразование времени в строку

Преобразовать объект класса `LocalTime` в строку позволяют следующие методы:

- `toString()` — преобразует объект в строку в соответствии с шаблоном по умолчанию. Формат метода:

```
public String toString()
```

- `format()` — преобразует объект в строку, соответствующую формату, заданному с помощью класса `DateTimeFormatter`. Формат метода:

```
import java.time.format.DateTimeFormatter;
public String format(DateTimeFormatter formatter)
```

Пример:

```
LocalTime t = LocalTime.of(22, 12, 36, 788000000);
System.out.println(
   t.format(DateTimeFormatter.ofPattern("HH:mm")));
                                  // 22:12
System.out.println(t.toString()); // 22:12:36.788
t = LocalTime.of(22, 12, 36);
System.out.println(t.toString()); // 22:12:36
t = LocalTime.of(22, 12);
System.out.println(t.toString()); // 22:12
```

9.3. Класс *LocalDateTime*: дата и время

Класс `LocalDateTime` позволяет производить операции над датой и временем. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.time.LocalDateTime;
```

С помощью констант MAX и MIN можно получить максимальное и минимальное значение:

```
System.out.println(LocalDateTime.MAX);
// +999999999-12-31T23:59:59.999999999
System.out.println(LocalDateTime.MIN);
// -999999999-01-01T00:00
```

9.3.1. Создание экземпляра класса *LocalDateTime*

Создать экземпляр класса LocalDateTime позволяют следующие статические методы:

- now() — создает объект с текущей локальной датой и временем. Формат метода:

  ```
  public static LocalDateTime now()
  ```

  Пример:

  ```
  LocalDateTime dt = LocalDateTime.now();
  System.out.println(dt); // 2018-03-24T23:28:11.449687200
  ```

- of() — создает объект на основе различных компонентов даты и времени. Форматы метода:

  ```
  public static LocalDateTime of(int year, int month, int day,
                                 int hour, int minute)
  public static LocalDateTime of(int year, int month, int day,
                                 int hour, int minute, int second)
  public static LocalDateTime of(int year, int month, int day,
                                 int hour, int minute, int second,
                                 int nanoOfSecond)
  public static LocalDateTime of(int year, Month month, int day,
                                 int hour, int minute)
  public static LocalDateTime of(int year, Month month, int day,
                                 int hour, int minute, int second)
  public static LocalDateTime of(int year, Month month, int day,
                                 int hour, int minute, int second,
                                 int nanoOfSecond)
  public static LocalDateTime of(LocalDate date, LocalTime time)
  ```

  Пример:

  ```
  LocalDateTime dt = LocalDateTime.of(2018, 4, 28, 20, 5);
  System.out.println(dt); // 2018-04-28T20:05
  dt = LocalDateTime.of(2018, 4, 28, 20, 5, 6);
  System.out.println(dt); // 2018-04-28T20:05:06
  dt = LocalDateTime.of(2018, 4, 28, 20, 5, 6, 788000000);
  System.out.println(dt); // 2018-04-28T20:05:06.788
  dt = LocalDateTime.of(2018, Month.APRIL, 28, 20, 5);
  System.out.println(dt); // 2018-04-28T20:05
  dt = LocalDateTime.of(2018, Month.APRIL, 28, 20, 5, 6);
  System.out.println(dt); // 2018-04-28T20:05:06
  ```

```
dt = LocalDateTime.of(
          2018, Month.APRIL, 28, 20, 5, 6, 788000000);
System.out.println(dt); // 2018-04-28T20:05:06.788
LocalDate d = LocalDate.of(2018, 4, 28);
LocalTime t = LocalTime.of(20, 5, 6, 788000000);
dt = LocalDateTime.of(d, t);
System.out.println(dt); // 2018-04-28T20:05:06.788
```

- `ofEpochSecond()` — создает объект на основе количества секунд, прошедших с 1 января 1970 года, наносекунд и объекта класса `ZoneOffset`, задающего часовой пояс. Формат метода:

```
import java.time.ZoneOffset;
public static LocalDateTime ofEpochSecond(long epochSecond,
                        int nanoOfSecond, ZoneOffset offset)
```

Пример:

```
LocalDateTime dt = LocalDateTime.ofEpochSecond(
    System.currentTimeMillis()/1000, 0, ZoneOffset.of("+3"));
System.out.println(dt); // 2018-03-14T21:42:54
```

В этом примере для получения количества миллисекунд мы воспользовались методом `currentTimeMillis()` из класса `System`. Так как метод возвращает миллисекунды, мы делим значение на тысячу, чтобы получить секунды;

- `ofInstant()` — создает объект на основе экземпляра класса `Instant` и часового пояса. Формат метода:

```
public static LocalDateTime ofInstant(
                            Instant instant, ZoneId zone)
```

Пример:

```
LocalDateTime dt = LocalDateTime.ofInstant(
      Instant.now(), ZoneId.of("Europe/Moscow"));
System.out.println(dt); // 2018-03-14T21:55:28.809
```

- `parse()` — создает объект из строки специального формата. Форматы метода:

```
public static LocalDateTime parse(CharSequence text)
public static LocalDateTime parse(CharSequence text,
                                  DateTimeFormatter formatter)
```

Первый вариант метода использует формат даты и времени по умолчанию:

```
LocalDateTime dt = LocalDateTime.parse("2018-04-28T21:55:28");
System.out.println(dt); // 2018-04-28T21:55:28
```

Второй вариант метода позволяет указать произвольный формат, заданный с помощью объекта класса `DateTimeFormatter`:

```
LocalDateTime dt = LocalDateTime.parse("28.04.2018 21:55",
          DateTimeFormatter.ofPattern("dd.MM.uuuu HH:mm"));
System.out.println(dt); // 2018-04-28T21:55
```

Примечание

Для создания объекта класса `LocalDateTime` можно также воспользоваться методами из *разд. 9.1.4* и *9.2.4*.

9.3.2. Установка и получение компонентов даты и времени

Для установки и получения компонентов даты и времени предназначены следующие методы:

- `withYear()` — **устанавливает новое значение года. Формат метода:**

```
public LocalDateTime withYear(int year)
```

- `withMonth()` — **устанавливает новое значение месяца. Формат метода:**

```
public LocalDateTime withMonth(int month)
```

- `withDayOfMonth()` — **устанавливает новое значение дня. Формат метода:**

```
public LocalDateTime withDayOfMonth(int day)
```

Пример:

```
LocalDateTime dt =
   LocalDateTime.of(2018, 4, 27, 22, 12, 36, 788000000);
dt = dt.withYear(2020);
dt = dt.withMonth(7);
dt = dt.withDayOfMonth(28);
System.out.println(dt);   // 2020-07-28T22:12:36.788
```

- `withDayOfYear()` — **устанавливает новое значение месяца и дня на основании номера дня в году. Формат метода:**

```
public LocalDateTime withDayOfYear(int dayOfYear)
```

Пример:

```
LocalDateTime dt =
   LocalDateTime.of(2018, 4, 27, 22, 12, 36, 788000000);
dt = dt.withDayOfYear(119);
System.out.println(dt);   // 2018-04-29T22:12:36.788
```

- `withHour()` — **устанавливает новое значение часа. Формат метода:**

```
public LocalDateTime withHour(int hour)
```

- `withMinute()` — **устанавливает новое значение минут. Формат метода:**

```
public LocalDateTime withMinute(int minute)
```

- `withSecond()` — **устанавливает новое значение секунд. Формат метода:**

```
public LocalDateTime withSecond(int second)
```

- `withNano()` — **устанавливает новое значение наносекунд. Формат метода:**

```
public LocalDateTime withNano(int nanoOfSecond)
```

Пример:

```
LocalDateTime dt =
   LocalDateTime.of(2018, 4, 27, 22, 12, 36, 788000000);
dt = dt.withHour(11);
dt = dt.withMinute(28);
dt = dt.withSecond(5);
dt = dt.withNano(555000000);
System.out.println(dt);      // 2018-04-27T11:28:05.555
```

- `getYear()` — **возвращает год. Формат метода:**

```
public int getYear()
```

- `getMonthValue()` **и** `getMonth()` — **возвращают месяц. Форматы методов:**

```
public int getMonthValue()
public Month getMonth()
```

- `getDayOfMonth()` — **возвращает день в месяце. Формат метода:**

```
public int getDayOfMonth()
```

Пример:

```
LocalDateTime dt =
   LocalDateTime.of(2018, 4, 27, 22, 12, 36, 788000000);
System.out.println(dt.getYear());         // 2018
System.out.println(dt.getMonthValue());   // 4
System.out.println(dt.getMonth());        // APRIL
System.out.println(dt.getDayOfMonth());   // 27
```

- `getDayOfYear()` — **возвращает номер дня в году. Формат метода:**

```
public int getDayOfYear()
```

Пример:

```
LocalDateTime dt =
   LocalDateTime.of(2018, 4, 27, 22, 12, 36, 788000000);
System.out.println(dt.getDayOfYear());   // 117
```

- `getDayOfWeek()` — **возвращает день недели в виде значения из перечисления** `DayOfWeek` **(содержит константы** `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` **и** `SUNDAY`**). Прежде чем использовать перечисление** `DayOfWeek`**, необходимо выполнить его импорт с помощью инструкции:**

```
import java.time.DayOfWeek;
```

Формат метода:

```
public DayOfWeek getDayOfWeek()
```

Пример:

```
LocalDateTime dt =
   LocalDateTime.of(2018, 4, 27, 22, 12, 36, 788000000);
System.out.println(dt.getDayOfWeek());   // FRIDAY
```

- `getHour()` — **возвращает часы. Формат метода:**

  ```
  public int getHour()
  ```

- `getMinute()` — **возвращает минуты. Формат метода:**

  ```
  public int getMinute()
  ```

- `getSecond()` — **возвращает секунды. Формат метода:**

  ```
  public int getSecond()
  ```

- `getNano()` — **возвращает наносекунды. Формат метода:**

  ```
  public int getNano()
  ```

  Пример:

  ```
  LocalDateTime dt =
     LocalDateTime.of(2018, 4, 27, 22, 12, 36, 788000000);
  System.out.println(dt.getHour());   // 22
  System.out.println(dt.getMinute()); // 12
  System.out.println(dt.getSecond()); // 36
  System.out.println(dt.getNano());   // 788000000
  ```

Класс `LocalDateTime` **не содержит некоторых методов, существующих в классах** `LocalDate` **и** `LocalTime`**, — например, невозможно проверить, является ли год високосным. Однако можно выполнить преобразование объекта класса** `LocalDateTime` **в объекты классов** `LocalDate` **и** `LocalTime` **с помощью следующих методов:**

- `toLocalDate()` — **возвращает объект класса** `LocalDate`**. Формат метода:**

  ```
  public LocalDate toLocalDate()
  ```

  Пример:

  ```
  LocalDateTime dt =
     LocalDateTime.of(2018, 4, 27, 22, 12, 36, 788000000);
  LocalDate d = dt.toLocalDate();
  // Является ли год високосным?
  System.out.println(d.isLeapYear());    // false
  // Сколько дней в месяце?
  System.out.println(d.lengthOfMonth()); // 30
  // Сколько дней в году?
  System.out.println(d.lengthOfYear());  // 365
  ```

- `toLocalTime()` — **возвращает объект класса** `LocalTime`**. Формат метода:**

  ```
  public LocalTime toLocalTime()
  ```

  Пример:

  ```
  LocalDateTime dt =
     LocalDateTime.of(2018, 4, 27, 22, 12, 36, 788000000);
  LocalTime t = dt.toLocalTime();
  // Сколько секунд прошло с начала дня?
  System.out.println(t.toSecondOfDay());    // 79956
  ```

9.3.3. Прибавление и вычитание значений

Для прибавления и вычитания значений предназначены следующие методы:

- `plusYears()` — прибавляет указанное количество лет. Формат метода:

```
public LocalDateTime plusYears(long years)
```

- `plusMonths()` — прибавляет указанное количество месяцев. Формат метода:

```
public LocalDateTime plusMonths(long months)
```

- `plusWeeks()` — прибавляет указанное количество недель. Формат метода:

```
public LocalDateTime plusWeeks(long weeks)
```

- `plusDays()` — прибавляет указанное количество дней. Формат метода:

```
public LocalDateTime plusDays(long days)
```

Пример:

```
LocalDateTime dt =
   LocalDateTime.of(2018, 4, 27, 22, 12, 36, 788000000);
System.out.println(dt.plusYears(2));  // 2020-04-27T22:12:36.788
System.out.println(dt.plusMonths(1)); // 2018-05-27T22:12:36.788
System.out.println(dt.plusWeeks(4));  // 2018-05-25T22:12:36.788
System.out.println(dt.plusDays(4));   // 2018-05-01T22:12:36.788
```

- `plusHours()` — прибавляет указанное количество часов. Формат метода:

```
public LocalDateTime plusHours(long hours)
```

- `plusMinutes()` — прибавляет указанное количество минут. Формат метода:

```
public LocalDateTime plusMinutes(long minutes)
```

- `plusSeconds()` — прибавляет указанное количество секунд. Формат метода:

```
public LocalDateTime plusSeconds(long seconds)
```

- `plusNanos()` — прибавляет указанное количество наносекунд. Формат метода:

```
public LocalDateTime plusNanos(long nanos)
```

Пример:

```
LocalDateTime dt =
   LocalDateTime.of(2018, 4, 27, 22, 12, 36, 788000000);
System.out.println(
          dt.plusHours(3));    // 2018-04-28T01:12:36.788
System.out.println(
          dt.plusMinutes(50)); // 2018-04-27T23:02:36.788
System.out.println(
          dt.plusSeconds(30)); // 2018-04-27T22:13:06.788
System.out.println(
     dt.plusNanos(500000000L)); // 2018-04-27T22:12:37.288
```

- `minusYears()` — вычитает указанное количество лет. Формат метода:

```
public LocalDateTime minusYears(long years)
```

- `minusMonths()` — вычитает указанное количество месяцев. Формат метода:

```
public LocalDateTime minusMonths(long months)
```

- `minusWeeks()` — вычитает указанное количество недель. Формат метода:

```
public LocalDateTime minusWeeks(long weeks)
```

- `minusDays()` — вычитает указанное количество дней. Формат метода:

```
public LocalDateTime minusDays(long days)
```

Пример:

```
LocalDateTime dt =
   LocalDateTime.of(2018, 4, 30, 22, 12, 36, 788000000);
System.out.println(dt.minusYears(4));  // 2014-04-30T22:12:36.788
System.out.println(dt.minusMonths(2)); // 2018-02-28T22:12:36.788
System.out.println(dt.minusWeeks(4));  // 2018-04-02T22:12:36.788
System.out.println(dt.minusDays(4));   // 2018-04-26T22:12:36.788
```

- `minusHours()` — вычитает указанное количество часов. Формат метода:

```
public LocalDateTime minusHours(long hours)
```

- `minusMinutes()` — вычитает указанное количество минут. Формат метода:

```
public LocalDateTime minusMinutes(long minutes)
```

- `minusSeconds()` — вычитает указанное количество секунд. Формат метода:

```
public LocalDateTime minusSeconds(long seconds)
```

- `minusNanos()` — вычитает указанное количество наносекунд. Формат метода:

```
public LocalDateTime minusNanos(long nanos)
```

Пример:

```
LocalDateTime dt =
   LocalDateTime.of(2018, 4, 30, 22, 12, 36, 788000000);
System.out.println(
      dt.minusHours(26));          // 2018-04-29T20:12:36.788
System.out.println(
      dt.minusMinutes(50));        // 2018-04-30T21:22:36.788
System.out.println(
      dt.minusSeconds(30));        // 2018-04-30T22:12:06.788
System.out.println(
      dt.minusNanos(854000000L)); // 2018-04-30T22:12:35.934
```

9.3.4. Сравнение объектов

Сравнить два объекта класса `LocalDateTime` позволяют следующие методы:

- `equals()` — сравнивает два объекта. Если объекты равны, то метод возвращает значение `true`, в противном случае — значение `false`. Формат метода:

```
public boolean equals(Object obj)
```

Пример:

```
LocalDateTime dt1 = LocalDateTime.of(2018, 4, 30, 22, 12);
LocalDateTime dt2 = LocalDateTime.of(2018, 4, 30, 22, 12);
LocalDateTime dt3 = LocalDateTime.of(2018, 4, 29, 22, 12);
System.out.println(dt1.equals(dt2)); // true
System.out.println(dt1.equals(dt3)); // false
```

- `isEqual()` — сравнивает два объекта. Если объекты равны, то метод возвращает значение `true`, в противном случае — значение `false`. Формат метода:

```
public boolean isEqual(ChronoLocalDateTime<?> other)
```

- `compareTo()` — сравнивает два объекта. Возвращает значение `0` — если объекты равны, положительное значение — если текущий объект больше `other`, отрицательное значение — если текущий объект меньше `other`. Формат метода:

```
public int compareTo(ChronoLocalDateTime<?> other)
```

Пример:

```
LocalDateTime dt1 = LocalDateTime.of(2018, 4, 30, 22, 12);
LocalDateTime dt2 = LocalDateTime.of(2018, 4, 30, 22, 12);
LocalDateTime dt3 = LocalDateTime.of(2018, 4, 29, 22, 12);
System.out.println(dt1.compareTo(dt2)); // 0
System.out.println(dt1.compareTo(dt3)); // 1
dt3 = LocalDateTime.of(2018, 5, 1, 2, 1);
System.out.println(dt1.compareTo(dt3)); // -1
```

- `isBefore()` — возвращает значение `true`, если текущий объект меньше `other`, и значение `false` — в противном случае. Формат метода:

```
public boolean isBefore(ChronoLocalDateTime<?> other)
```

Пример:

```
LocalDateTime dt1 = LocalDateTime.of(2018, 4, 30, 22, 12);
LocalDateTime dt2 = LocalDateTime.of(2018, 4, 30, 22, 12);
LocalDateTime dt3 = LocalDateTime.of(2018, 4, 29, 22, 12);
System.out.println(dt1.isBefore(dt2)); // false
System.out.println(dt1.isBefore(dt3)); // false
dt3 = LocalDateTime.of(2018, 5, 1, 2, 1);
System.out.println(dt1.isBefore(dt3)); // true
```

- `isAfter()` — возвращает значение `true`, если текущий объект больше `other`, и значение `false` — в противном случае. Формат метода:

```
public boolean isAfter(ChronoLocalDateTime<?> other)
```

Пример:

```
LocalDateTime dt1 = LocalDateTime.of(2018, 4, 30, 22, 12);
LocalDateTime dt2 = LocalDateTime.of(2018, 4, 30, 22, 12);
LocalDateTime dt3 = LocalDateTime.of(2018, 4, 29, 22, 12);
```

```
System.out.println(dt1.isAfter(dt2)); // false
System.out.println(dt1.isAfter(dt3)); // true
dt3 = LocalDateTime.of(2018, 5, 1, 2, 1);
System.out.println(dt1.isAfter(dt3)); // false
```

9.3.5. Преобразование даты и времени в строку

Преобразовать объект класса `LocalDateTime` в строку позволяют следующие методы:

- `toString()` — преобразует объект в строку в соответствии с шаблоном по умолчанию. Формат метода:

  ```
  public String toString()
  ```

- `format()` — преобразует объект в строку, соответствующую формату, заданному с помощью класса `DateTimeFormatter`. Формат метода:

  ```
  public String format(DateTimeFormatter formatter)
  ```

  Пример:

  ```
  LocalDateTime dt =
     LocalDateTime.of(2018, 4, 30, 22, 12, 36, 788000000);
  System.out.println(dt.format(
     DateTimeFormatter.ofPattern("dd.MM.uuuu HH:mm:ss")));
  // 30.04.2018 22:12:36
  System.out.println(dt.toString());
  // 2018-04-30T22:12:36.788
  ```

9.4. Класс *Instant*: представление времени в наносекундах, прошедших с 1 января 1970 года

Класс `Instant` представляет время в наносекундах, прошедших с 1 января 1970 года. Так как количество наносекунд может превышать диапазон значений для типа `long`, значение хранится в двух переменных: первая переменная содержит количество секунд, прошедших с 1 января 1970 года (тип `long`), а вторая — остаток наносекунд (тип `int`). Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.time.Instant;
```

9.4.1. Создание экземпляра класса *Instant*

Создать экземпляр класса `Instant` позволяют следующие статические методы:

- `now()` — создает объект на основе текущего системного времени. Обратите внимание: отображение времени будет соответствовать часовому поясу UTC, а не часовому поясу из локали. Формат метода:

  ```
  public static Instant now()
  ```

Пример:

```
Instant ins = Instant.now();
System.out.println(ins);  // 2018-03-13T22:08:03.129249600Z
LocalDateTime dt = LocalDateTime.ofInstant(
   Instant.now(), ZoneId.of("Europe/Moscow"));
System.out.println(dt);   // 2018-03-14T01:08:03.176125900
```

- `ofEpochMilli()` — создает объект на основе количества миллисекунд, прошедших с 1 января 1970 года. Формат метода:

```
public static Instant ofEpochMilli(long epochMilli)
```

Пример:

```
Instant ins = Instant.ofEpochMilli(
                      System.currentTimeMillis());
System.out.println(ins);  // 2018-03-13T22:10:35.811Z
ins = Instant.ofEpochMilli((new Date()).getTime());
System.out.println(ins);  // 2018-03-13T22:10:35.858Z
```

- `ofEpochSecond()` — создает объект на основе количества секунд, прошедших с 1 января 1970 года, и наносекунд. Форматы метода:

```
public static Instant ofEpochSecond(long epochSecond)
public static Instant ofEpochSecond(long epochSecond,
                                    long nanoAdjustment)
```

Пример:

```
Instant ins = Instant.ofEpochSecond(
              System.currentTimeMillis() / 1000);
System.out.println(ins);  // 2018-03-13T22:14:47Z
ins = Instant.ofEpochSecond(
              ((new Date()).getTime()) / 1000);
System.out.println(ins);  // 2018-03-13T22:14:47Z
ins = Instant.ofEpochSecond(1520702397L, 380000000L);
System.out.println(ins);  // 2018-03-10T17:19:57.380Z
```

- `parse()` — создает объект на основе строки специального формата. Формат метода:

```
public static Instant parse(CharSequence text)
```

Пример:

```
Instant ins = Instant.parse("2018-04-28T22:50:18.380Z");
System.out.println(ins);  // 2018-04-28T22:50:18.380Z
```

Создать объект класса `Instant` можно также на основе объекта класса `Date`. Для этого предназначен метод `toInstant()`. Формат метода:

```
public Instant toInstant()
```

Пример:

```
Instant ins = (new Date()).toInstant();
System.out.println(ins);
// 2018-03-13T22:18:31.315Z
System.out.println(Date.from(ins));
// Wed Mar 14 01:18:31 MSK 2018
```

Как видно из примера, преобразовать объект класса `Instant` в объект класса `Date` позволяет статический метод `from()`. Формат метода:

```
public static Date from(Instant instant)
```

Для использования класса `Date` необходимо выполнить его импорт с помощью инструкции:

```
import java.util.Date;
```

9.4.2. Получение компонентов времени

Для получения компонентов времени предназначены следующие методы:

- `getEpochSecond()` — возвращает значение первой составляющей (количество секунд, прошедших с 1 января 1970 года). Формат метода:

  ```
  public long getEpochSecond()
  ```

- `getNano()` — возвращает значение второй составляющей (остаток наносекунд). Формат метода:

  ```
  public int getNano()
  ```

- `toEpochMilli()` — возвращает количество миллисекунд, прошедших с 1 января 1970 года. Формат метода:

  ```
  public long toEpochMilli()
  ```

  Пример:

  ```
  Instant ins = Instant.now();
  System.out.println(ins); // 2018-03-13T22:23:37.693619Z
  System.out.println(ins.getEpochSecond()); // 1520979817
  System.out.println(ins.getNano());        // 693619000
  System.out.println(ins.toEpochMilli());   // 1520979817693
  ```

9.4.3. Прибавление и вычитание значений

Для прибавления и вычитания значений предназначены следующие методы:

- `plusSeconds()` — прибавляет указанное количество секунд. Формат метода:

  ```
  public Instant plusSeconds(long secondsToAdd)
  ```

- `plusNanos()` — прибавляет указанное количество наносекунд. Формат метода:

  ```
  public Instant plusNanos(long nanosToAdd)
  ```

- `plusMillis()` — прибавляет указанное количество миллисекунд. Формат метода:

```
public Instant plusMillis(long millisToAdd)
```

Пример:

```
Instant ins = Instant.ofEpochSecond(1520702397L, 460000000L);
ins = ins.plusSeconds(366L);
System.out.println(ins.getEpochSecond()); // 1520702763
ins = ins.plusNanos(40000000L);
System.out.println(ins.getNano());        // 500000000
ins = ins.plusMillis(333000L);
System.out.println(ins.getEpochSecond()); // 1520703096
```

- `minusSeconds()` — вычитает указанное количество секунд. Формат метода:

```
public Instant minusSeconds(long secondsToSubtract)
```

- `minusNanos()` — вычитает указанное количество наносекунд. Формат метода:

```
public Instant minusNanos(long nanosToSubtract)
```

- `minusMillis()` — вычитает указанное количество миллисекунд. Формат метода:

```
public Instant minusMillis(long millisToSubtract)
```

Пример:

```
Instant ins = Instant.ofEpochSecond(1520702397L, 460000000L);
ins = ins.minusSeconds(85634L);
System.out.println(ins.getEpochSecond()); // 1520616763
ins = ins.minusNanos(460000000L);
System.out.println(ins.getNano());        // 0
ins = ins.minusMillis(1800000001L);
System.out.println(ins.getEpochSecond()); // 1518816762
System.out.println(ins.getNano());        // 999000000
```

9.4.4. Сравнение объектов

Сравнить два объекта класса `Instant` позволяют следующие методы:

- `equals()` — сравнивает два объекта. Если объекты равны, то метод возвращает значение `true`, в противном случае — значение `false`. Формат метода:

```
public boolean equals(Object other)
```

Пример:

```
Instant ins1 = Instant.ofEpochSecond(634L, 0);
Instant ins2 = Instant.ofEpochSecond(634L, 0);
Instant ins3 = Instant.ofEpochSecond(633L, 0);
System.out.println(ins1.equals(ins2)); // true
System.out.println(ins1.equals(ins3)); // false
```

- `compareTo()` — сравнивает два объекта. Возвращает значение `0` — если объекты равны, положительное значение — если текущий объект больше `other`, отрицательное значение — если текущий объект меньше `other`. Формат метода:

```
public int compareTo(Instant other)
```

Пример:

```
Instant ins1 = Instant.ofEpochSecond(634L, 0);
Instant ins2 = Instant.ofEpochSecond(634L, 0);
Instant ins3 = Instant.ofEpochSecond(633L, 0);
System.out.println(ins1.compareTo(ins2)); // 0
System.out.println(ins1.compareTo(ins3)); // 1
ins3 = Instant.ofEpochSecond(635L, 0);
System.out.println(ins1.compareTo(ins3)); // -1
```

- `isBefore()` — возвращает значение `true`, если текущий объект меньше `other`, и значение `false` — в противном случае. Формат метода:

```
public boolean isBefore(Instant other)
```

Пример:

```
Instant ins1 = Instant.ofEpochSecond(634L, 0);
Instant ins2 = Instant.ofEpochSecond(634L, 0);
Instant ins3 = Instant.ofEpochSecond(633L, 0);
System.out.println(ins1.isBefore(ins2)); // false
System.out.println(ins1.isBefore(ins3)); // false
ins3 = Instant.ofEpochSecond(635L, 0);
System.out.println(ins1.isBefore(ins3)); // true
```

- `isAfter()` — возвращает значение `true`, если текущий объект больше `other`, и значение `false` — в противном случае. Формат метода:

```
public boolean isAfter(Instant other)
```

Пример:

```
Instant ins1 = Instant.ofEpochSecond(634L, 0);
Instant ins2 = Instant.ofEpochSecond(634L, 0);
Instant ins3 = Instant.ofEpochSecond(633L, 0);
System.out.println(ins1.isAfter(ins2)); // false
System.out.println(ins1.isAfter(ins3)); // true
ins3 = Instant.ofEpochSecond(635L, 0);
System.out.println(ins1.isAfter(ins3)); // false
```

9.4.5. Преобразование объекта класса *Instant* в объект класса *LocalDateTime*

Для преобразования объекта класса `Instant` в объект класса `LocalDateTime` предназначены следующие методы из класса `LocalDateTime`:

- `ofEpochSecond()` — создает объект на основе количества секунд, прошедших с 1 января 1970 года, наносекунд и объекта класса `ZoneOffset`, задающего часовой пояс. Формат метода:

```
public static LocalDateTime ofEpochSecond(long epochSecond,
                          int nanoOfSecond, ZoneOffset offset)
```

Пример:

```
Instant ins = Instant.now();
System.out.println(ins); // 2018-03-13T22:41:14.263906300Z
LocalDateTime dt = LocalDateTime.ofEpochSecond(
   ins.getEpochSecond(), ins.getNano(), ZoneOffset.of("+3"));
System.out.println(dt); // 2018-03-14T01:41:14.263906300
```

- `ofInstant()` — создает объект на основе экземпляра класса `Instant` и часового пояса. Формат метода:

```
public static LocalDateTime ofInstant(Instant instant,
                                       ZoneId zone)
```

Пример:

```
LocalDateTime dt = LocalDateTime.ofInstant(
   Instant.now(), ZoneId.of("Europe/Moscow"));
System.out.println(dt); // 2018-03-14T01:42:35.193949100
```

9.5. Класс *DateTimeFormatter*: форматирование даты и времени

Класс `DateTimeFormatter` применяется для форматированного вывода даты и времени, а также для указания формата при преобразовании строки в объекты классов `LocalDate`, `LocalTime`, `LocalDateTime` и `Instant`. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.time.format.DateTimeFormatter;
```

9.5.1. Создание экземпляра класса *DateTimeFormatter*

Создать экземпляр класса `DateTimeFormatter` позволяют следующие статические методы:

- `ofLocalizedDate()` — создает объект с форматом, соответствующим представлению даты в текущей локали. Формат метода:

```
public static DateTimeFormatter ofLocalizedDate(
                                   FormatStyle dateStyle)
```

В качестве значения параметра указывается одна из констант (`FULL`, `LONG`, `MEDIUM` или `SHORT`) перечисления `FormatStyle`. Прежде чем использовать это перечисление, необходимо выполнить его импорт с помощью инструкции:

```
import java.time.format.FormatStyle;
```

Выведем дату в разных вариантах:

```
LocalDateTime dt =
   LocalDateTime.of(2018, 4, 30, 2, 5, 9, 788000000);
DateTimeFormatter dtf =
      DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL);
System.out.println(dt.format(dtf));
// понедельник, 30 апреля 2018 г.
dtf = DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG);
System.out.println(dt.format(dtf)); // 30 апреля 2018 г.
dtf = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
System.out.println(dt.format(dtf)); // 30.04.18
dtf = DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);
System.out.println(dt.format(dtf)); // 30 апр. 2018 г.
```

- `ofLocalizedTime()` — создает объект с форматом, соответствующим представлению времени в текущей локали. Формат метода:

```
public static DateTimeFormatter ofLocalizedTime(
                                    FormatStyle timeStyle)
```

В качестве значения параметра указывается одна из констант (`MEDIUM` или `SHORT`) перечисления `FormatStyle`:

```
LocalDateTime dt =
   LocalDateTime.of(2018, 4, 30, 2, 5, 9, 788000000);
DateTimeFormatter dtf =
      DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);
System.out.println(dt.format(dtf)); // 2:05
dtf = DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM);
System.out.println(dt.format(dtf)); // 2:05:09
```

- `ofLocalizedDateTime()` — создает объект с форматом, соответствующим представлению даты и времени в текущей локали. Форматы метода:

```
public static DateTimeFormatter ofLocalizedDateTime(
              FormatStyle dateTimeStyle)
public static DateTimeFormatter ofLocalizedDateTime(
              FormatStyle dateStyle, FormatStyle timeStyle)
```

В первом формате указывается стиль для даты и времени, а во втором формате первое значение задает стиль даты, а второе — стиль времени:

```
LocalDateTime dt =
   LocalDateTime.of(2018, 4, 30, 2, 5, 9, 788000000);
DateTimeFormatter dtf =
      DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);
System.out.println(dt.format(dtf)); // 30.04.18, 2:05
dtf = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);
System.out.println(dt.format(dtf)); // 30 апр. 2018 г., 2:05:09
dtf = DateTimeFormatter.ofLocalizedDateTime(
         FormatStyle.FULL, FormatStyle.MEDIUM);
```

```
System.out.println(dt.format(dtf));
// понедельник, 30 апреля 2018 г., 2:05:09
```

- `ofPattern()` — создает объект с форматом, соответствующим строке шаблона и текущей или указанной локали. Форматы метода:

```
public static DateTimeFormatter ofPattern(String pattern)
public static DateTimeFormatter ofPattern(String pattern,
                                          Locale locale)
```

Пример:

```
LocalDateTime dt =
   LocalDateTime.of(2018, 4, 30, 2, 5, 9, 788000000);
DateTimeFormatter dtf =
      DateTimeFormatter.ofPattern("dd MMMM uuuu");
System.out.println(dt.format(dtf)); // 30 апреля 2018
dtf = DateTimeFormatter.ofPattern("dd MMMM uuuu",
                                  new Locale("en", "US"));
System.out.println(dt.format(dtf)); // 30 April 2018
```

9.5.2. Специальные символы в строке шаблона

В строке шаблона в методе `ofPattern()`, а также в методах `parse()` и `format()` классов `LocalDate`, `LocalTime`, `LocalDateTime` и `Instant`, могут быть указаны следующие специальные символы (примеры указаны для русской локали):

- `G` — эра (например, н. э.). Буква может повторяться до пяти раз, и в английской локали можно получить значения: AD, Anno Domini и A;
- `y` или `yyyy` — год в эре из четырех цифр (например, 2018);
- `yy` — последние две цифры года в эре (от 00 до 99);
- `u` или `uuuu` — год из четырех цифр (например, 2018);
- `uu` — последние две цифры года (от 00 до 99);
- `M` или `L` — номер месяца без предваряющего нуля (от 1 до 12);
- `MM` или `LL` — номер месяца с предваряющим нулем (от 01 до 12);
- `MMM` — аббревиатура месяца в зависимости от настроек локали (например, апр);
- `MMMM` — название месяца в зависимости от настроек локали (например, апреля);
- `LLL` — аббревиатура месяца в зависимости от настроек локали (например, апр.);
- `LLLL` — название месяца в зависимости от настроек локали (например, апрель);
- `dd` — номер дня в месяце с предваряющим нулем (от 01 до 31);
- `d` — номер дня в месяце без предваряющего нуля (от 1 до 31);
- `w` — неделя в году без предваряющего нуля;
- `ww` — неделя в году с предваряющим нулем;
- `W` — неделя в месяце без предваряющего нуля;

- `D` — день с начала года без предваряющего нуля (от 1 до 366);
- `DDD` — день с начала года с предваряющим нулем (от 001 до 366);
- `F` — день недели в месяце без предваряющего нуля (началом недели считается день недели, соответствующий первому дню месяца, — например, если первый день месяца пятница, то пятница будет первым днем недели, а четверг — седьмым);
- `E` — аббревиатура дня недели в зависимости от настроек локали (например, чт);
- `EEEE` — название дня недели в зависимости от настроек локали (например, четверг);
- `H` — часы в 24-часовом формате без предваряющего нуля (от 0 до 23);
- `HH` — часы в 24-часовом формате с предваряющим нулем (от 00 до 23);
- `h` — часы в 12-часовом формате без предваряющего нуля (от 1 до 12);
- `hh` — часы в 12-часовом формате с предваряющим нулем (от 01 до 12);
- `k` — часы в 24-часовом формате без предваряющего нуля (от 1 до 24);
- `kk` — часы в 24-часовом формате с предваряющим нулем (от 01 до 24);
- `K` — часы в 12-часовом формате без предваряющего нуля (от 0 до 11);
- `KK` — часы в 12-часовом формате с предваряющим нулем (от 00 до 11);
- `m` — минуты без предваряющего нуля (от 0 до 59);
- `mm` — минуты с предваряющим нулем (от 00 до 59);
- `s` — секунды без предваряющего нуля (от 0 до 59);
- `ss` — секунды с предваряющим нулем (от 00 до 59);
- `a` — AM или PM в верхнем регистре (например, AM);
- `S` — миллисекунды. Буква может повторяться до 9 раз. От количества букв зависит количество цифр — например, `SSS` соответствует трем цифрам: 571;
- `n` — наносекунды (например, 788000000);
- `Z` — смещение часового пояса (например, +0300);
- `ZZZZ` — смещение часового пояса (например, GMT+03:00);
- `ZZZZZ` — смещение часового пояса (например, +03:00);
- `O` — смещение часового пояса (например, GMT+3);
- `X` — смещение часового пояса (например, +03);
- `XX` — смещение часового пояса (например, +0300);
- `XXX` — смещение часового пояса (например, +03:00);
- `z` — название часового пояса (например, MSK);
- `zzzz` — название часового пояса (например, Moscow Standard Time);
- `VV` — название часового пояса (например, Europe/Moscow).

При использовании букв, отвечающих за часовые пояса, следует учитывать, что классы `LocalDate`, `LocalTime`, `LocalDateTime` и `Instant` ничего не знают о часовых поясах, поэтому попытка вывести название или смещение часового пояса приведет к исключению и выполнение программы будет остановлено. Для работы с часовыми поясами необходимо использовать классы `ZonedDateTime` (название и смещение часового пояса) и `OffsetDateTime` (смещение часового пояса).

Для преобразования объекта класса `LocalDateTime` в объекты классов `ZonedDateTime` и `OffsetDateTime` предназначены следующие методы из класса `LocalDateTime`:

- `atZone()` — возвращает объект класса `ZonedDateTime`. Формат метода:

```
public ZonedDateTime atZone(ZoneId zone)
```

В параметре `zone` указывается экземпляр класса `ZoneId`. Создать объект этого класса можно следующим образом:

```
ZoneId z = ZoneId.of("Europe/Moscow");
```

Получить массив с названиями всех поддерживаемых часовых зон позволяет статический метод `getAvailableZoneIds()` из класса `ZoneId`. Чтобы указать зону по умолчанию, достаточно вызвать статический метод `systemDefault()`:

```
LocalDateTime dt = LocalDateTime.of(2018, 4, 23, 12, 51, 19);
ZonedDateTime zdt = dt.atZone(ZoneId.systemDefault());
DateTimeFormatter dtf =
      DateTimeFormatter.ofPattern("Z ZZZZ ZZZZZ z");
System.out.println(zdt.format(dtf));
// +0300 GMT+03:00 +03:00 MSK
System.out.println(zdt.toString());
// 2018-04-23T12:51:19+03:00[Europe/Moscow]
```

Не забудьте импортировать задействованные классы с помощью инструкций:

```
import java.time.ZoneId;
import java.time.ZonedDateTime;
```

- `atOffset()` — возвращает экземпляр класса `OffsetDateTime`. Формат метода:

```
public OffsetDateTime atOffset(ZoneOffset offset)
```

В параметре `offset` указывается экземпляр класса `ZoneOffset`. Создать объект этого класса можно с помощью статических методов `ofHours()` и `of()` из класса `ZoneOffset`:

```
ZoneOffset z1 = ZoneOffset.ofHours(3);
ZoneOffset z2 = ZoneOffset.of("+3");
System.out.println(z1.toString()); // +03:00
System.out.println(z2.toString()); // +03:00
```

Пример использования метода `atOffset()`:

```
LocalDateTime dt = LocalDateTime.of(2018, 4, 23, 12, 51, 19);
OffsetDateTime odt = dt.atOffset(ZoneOffset.ofHours(3));
DateTimeFormatter dtf =
      DateTimeFormatter.ofPattern("Z ZZZZ ZZZZZ");
```

```
System.out.println(odt.format(dtf));
// +0300 GMT+03:00 +03:00
System.out.println(odt.toString());
// 2018-04-23T12:51:19+03:00
```

Не забудьте импортировать задействованные классы с помощью инструкций:

```
import java.time.ZoneOffset;
import java.time.OffsetDateTime;
```

Если в строке шаблона мы хотим указать какой-либо текст, то его необходимо заключить в апострофы:

```
"'year:' uuuu"
```

Результат:

```
year: 2018
```

Чтобы указать символ апострофа, его нужно удвоить:

```
"'year:' '' uuuu ''"
```

Результат:

```
year: ' 2018 '
```

Выведем текущие дату и время таким образом, чтобы день недели и месяц были написаны сначала на русском, а затем на английском языке (листинг 9.7).

Листинг 9.7. Класс `DateTimeFormatter`

```
import java.time.*;
import java.time.format.DateTimeFormatter;
import java.util.Locale;

public class MyClass {
   public static void main(String[] args) {
      LocalDateTime dt = LocalDateTime.now();
      DateTimeFormatter dtf =
         DateTimeFormatter.ofPattern(
            "EEEE dd MMMM uuuu HH:mm", new Locale("ru", "RU"));
      System.out.println("Сегодня:");
      System.out.println(dt.format(dtf));
      dtf = dtf.withLocale(new Locale("en", "US"));
      System.out.println(dt.format(dtf));
   }
}
```

Примерный результат:

```
Сегодня:
пятница 23 марта 2018 20:29
Friday 23 March 2018 20:29
```

В этом примере мы воспользовались методом `withLocale()`, чтобы изменить локаль. Как и во всех остальных классах, возвращается новый объект, а не изменяется старый. Формат метода:

```
public DateTimeFormatter withLocale(Locale locale)
```

Иногда может также пригодиться метод `withZone()`, который устанавливает часовой пояс и возвращает новый объект. Формат метода:

```
public DateTimeFormatter withZone(ZoneId zone)
```

Например, мы имеем строку с датой и временем, но в ней отсутствуют данные о часовом поясе. Если мы эту строку передадим методу `parse()` из класса `ZonedDateTime`, то получим ошибку. Чтобы избежать ошибки, достаточно с помощью метода `withZone()` указать часовой пояс:

```
DateTimeFormatter dtf =
      DateTimeFormatter.ofPattern("uuuu-MM-dd HH:mm:ss");
dtf = dtf.withZone(ZoneId.systemDefault());
ZonedDateTime zdt = ZonedDateTime.parse("2018-04-23 12:51:19", dtf);
System.out.println(zdt.toString());
// 2018-04-23T12:51:19+03:00[Europe/Moscow]
```

9.6. Класс *Period*: разница между двумя датами

Если нам необходимо определить разницу между двумя датами, то следует воспользоваться классом `Period`. Предварительно необходимо импортировать этот класс с помощью инструкции:

```
import java.time.Period;
```

> ***Примечание***
>
> Существует также класс `Duration`, позволяющий работать с диапазоном времени. За подробной информацией по этому классу обращайтесь к документации.

Создать объект класса `Period` позволяют следующие статические методы:

- `between()` — создает объект, содержащий разницу между двумя датами, представленными классом `LocalDate`. Формат метода:

  ```
  public static Period between(LocalDate start, LocalDate end)
  ```

  Пример:

  ```
  LocalDate d1 = LocalDate.of(2018, 4, 12);
  LocalDate d2 = LocalDate.of(2018, 6, 23);
  Period p = Period.between(d1, d2);
  System.out.println(p); // P2M11D
  ```

- `of()` — создает объект на основе количества лет, месяцев и дней. Формат метода:

  ```
  public static Period of(int years, int months, int days)
  ```

Пример:

```
Period p = Period.of(10, 5, 4);
System.out.println(p); // P10Y5M4D
```

- `ofYears()`, `ofMonths()` **и** `ofDays()` **— создают объект на основе только года, месяца или дня. Форматы методов:**

```
public static Period ofYears(int years)
public static Period ofMonths(int months)
public static Period ofDays(int days)
```

Пример:

```
System.out.println(Period.ofYears(10)); // P10Y
System.out.println(Period.ofMonths(5)); // P5M
System.out.println(Period.ofDays(4));   // P4D
```

- `ofWeeks()` **— создает объект на основе количества недель. В результате мы получим нулевые значения года и месяца, а число дней будет равно количеству дней в указанном количестве недель. Формат метода:**

```
public static Period ofWeeks(int weeks)
```

Пример:

```
System.out.println(Period.ofWeeks(10)); // P70D
```

- `parse()` **— создает объект на основании строки специального формата. Формат метода:**

```
public static Period parse(CharSequence text)
```

Пример:

```
System.out.println(Period.parse("P10Y2M11D")); // P10Y2M11D
```

Изменить значение отдельного компонента позволяют методы `withYears()`, `withMonths()` **и** `withDays()`. **Все методы возвращают новый объект. Форматы методов:**

```
public Period withYears(int years)
public Period withMonths(int months)
public Period withDays(int days)
```

Пример:

```
Period p = Period.parse("P10Y2M11D");
System.out.println(p.withYears(0));  // P2M11D
System.out.println(p.withMonths(0)); // P10Y11D
System.out.println(p.withDays(0));   // P10Y2M
```

Для добавления значения предназначены методы `plusYears()`, `plusMonths()` **и** `plusDays()`, **а для вычитания — методы** `minusYears()`, `minusMonths()` **и** `minusDays()`. **Форматы методов:**

```
public Period plusYears(long years)
public Period plusMonths(long months)
public Period plusDays(long days)
```

```
public Period minusYears(long years)
public Period minusMonths(long months)
public Period minusDays(long days)
```

Пример:

```
Period p = Period.parse("P10Y2M11D");
System.out.println(p.plusYears(6));    // P16Y2M11D
System.out.println(p.plusMonths(20));  // P10Y22M11D
System.out.println(p.plusDays(50));    // P10Y2M61D
System.out.println(p.minusYears(6));   // P4Y2M11D
System.out.println(p.minusMonths(20)); // P10Y-18M11D
System.out.println(p.minusDays(50));   // P10Y2M-39D
```

С помощью метода `multipliedBy()` можно умножить значение каждого компонента на указанное значение. Метод возвращает новый объект. Формат метода:

```
public Period multipliedBy(int scalar)
```

Пример:

```
Period p = Period.parse("P10Y2M11D");
System.out.println(p.multipliedBy(5));   // P50Y10M55D
```

Метод `negated()` позволяет изменить знак каждого компонента на противоположный. Формат метода:

```
public Period negated()
```

Пример:

```
Period p = Period.parse("P10Y2M-11D");
System.out.println(p.negated());    // P-10Y-2M11D
```

Получить значения отдельных компонентов позволяют методы `getYears()`, `getMonths()` и `getDays()`. Форматы методов:

```
public int getYears()
public int getMonths()
public int getDays()
```

Пример:

```
Period p = Period.parse("P10Y2M11D");
System.out.println(p.getYears());   // 10
System.out.println(p.getMonths());  // 2
System.out.println(p.getDays());    // 11
```

Метод `toTotalMonths()` позволяет получить общее количество месяцев с учетом значения компонентов, представляющих год и месяц. Формат метода:

```
public long toTotalMonths()
```

Пример:

```
Period p = Period.parse("P10Y2M11D");
System.out.println(p.toTotalMonths());   // 122
System.out.println(10 * 12 + 2);         // 122
```

Для сравнения и проверки значений предназначены следующие методы:

- `equals()` — сравнивает два объекта. Если объекты равны, то метод возвращает значение `true`, в противном случае — значение `false`. Формат метода:

  ```
  public boolean equals(Object obj)
  ```

  Пример:

  ```
  Period p = Period.parse("P1Y2M1D");
  System.out.println(p.equals(Period.parse("P1Y2M1D"))); // true
  System.out.println(p.equals(Period.parse("P1Y2M2D"))); // false
  ```

- `isNegative()` — возвращает значение `true`, если хотя бы один компонент содержит отрицательное значение, и `false` — в противном случае. Формат метода:

  ```
  public boolean isNegative()
  ```

  Пример:

  ```
  Period p = Period.parse("P1Y2M1D");
  System.out.println(p.isNegative());      // false
  p = Period.parse("P1Y-2M1D");
  System.out.println(p.isNegative());      // true
  ```

- `isZero()` — возвращает значение `true`, если все компоненты содержат значение `0`, и `false` — в противном случае. Формат метода:

  ```
  public boolean isZero()
  ```

  Пример:

  ```
  Period p = Period.parse("P1Y2M1D");
  System.out.println(p.isZero());      // false
  p = Period.parse("P0Y0M0D");
  System.out.println(p.isZero());      // true
  ```

После изменения значения компонентов можно выполнить нормализацию значений компонентов, представляющих год и месяц. Значение компонента, представляющего день, останется прежним, т. к. в разных месяцах количество дней различается. Для этого предназначен метод `normalized()`. Формат метода:

```
public Period normalized()
```

Пример:

```
Period p = Period.parse("P1Y42M51D");
System.out.println(p.normalized());  // P4Y6M51D
p = Period.parse("P-1Y42M51D");
System.out.println(p.normalized());  // P2Y6M51D
```

Итак, мы создали объект периода. Что мы с ним можем сделать, кроме получения значений отдельных компонентов? Мы можем прибавить или вычесть период из даты и получить дату, которая настанет, например, через 50 дней. Для этого классы `LocalDate`, `LocalDateTime`, `ZonedDateTime` и `OffsetDateTime` содержат методы `plus()` и

minus(), позволяющие указать в качестве значения объект класса Period. Прибавим и вычтем 50 дней из даты:

```
LocalDate d = LocalDate.of(2018, 4, 12);
Period p = Period.parse("P50D");
System.out.println(d.plus(p));      // 2018-06-01
System.out.println(d.minus(p));     // 2018-02-21
```

9.7. Получение количества дней между двумя датами

Теперь рассмотрим возможность получения количества дней между двумя датами. Это нужно, например, для вычисления количества дней, оставшихся до дня рождения. С помощью метода between() из класса Period мы можем получить объект, который будет содержать не только количество дней, но и количество месяцев. А вот сколько дней в месяце, объект класса Period не знает. Чтобы получить именно количество дней, необходимо воспользоваться методом until(). Этот метод содержит практически все классы, которые мы рассмотрели в этой главе. Формат метода:

```
public long until(Temporal endExclusive, TemporalUnit unit)
```

В первом параметре указывается экземпляр классов LocalDate, LocalTime, LocalDateTime и др. Во втором параметре можно указать одну из констант перечисления ChronoUnit. Для классов, работающих со временем, допустимы константы NANOS, MICROS, MILLIS, SECONDS, MINUTES, HOURS и HALF_DAYS, а для классов, хранящих дату, — DAYS, WEEKS, MONTHS, YEARS, DECADES, CENTURIES, MILLENNIA и ERAS. Для классов, хранящих одновременно и дату, и время, можно использовать все эти константы. Прежде чем работать с перечислением ChronoUnit, необходимо выполнить его импорт с помощью инструкции:

```
import java.time.temporal.ChronoUnit;
```

Для получения количества дней между датами воспользуемся константой DAYS:

```
LocalDate d1 = LocalDate.of(2018, 4, 12);
LocalDate d2 = LocalDate.of(2018, 6, 1);
System.out.println(d1.until(d2, ChronoUnit.DAYS));   // 50
```

Для вычисления разницы в днях можно также использовать такой вариант:

```
LocalDate d1 = LocalDate.of(2018, 4, 12);
LocalDate d2 = LocalDate.of(2018, 6, 1);
System.out.println(ChronoUnit.DAYS.between(d1, d2)); // 50
```

При использовании классов, хранящих одновременно и дату, и время, следует учитывать, что в этом случае мы сможем получить только количество целых дней, т. к. в расчет идет и время. Если время хотя бы на секунду отличается, то результат будет другим:

```
LocalDateTime dt1 = LocalDateTime.of(2018, 4, 12, 0, 0, 1);
LocalDateTime dt2 = LocalDateTime.of(2018, 6, 1, 0, 0, 0);
System.out.println(dt1.until(dt2, ChronoUnit.DAYS)); // 49
```

9.8. Получение времени в разных часовых поясах

Как уже отмечалось ранее (см. *разд. 9.5.2*), классы `LocalDate`, `LocalTime`, `LocalDateTime` и `Instant` ничего не знают о часовых поясах — они позволяют работать только с определенными компонентами даты и времени. Для работы с часовыми поясами необходимо использовать классы `ZonedDateTime` (название и смещение часового пояса) и `OffsetDateTime` (смещение часового пояса).

Для преобразования объекта класса `LocalDateTime` в объекты классов `ZonedDateTime` и `OffsetDateTime` предназначены следующие методы из класса `LocalDateTime`:

- `atZone()` — возвращает объект класса `ZonedDateTime`. Формат метода:

  ```
  public ZonedDateTime atZone(ZoneId zone)
  ```

  В параметре `zone` указывается экземпляр класса `ZoneId`. Создать объект этого класса можно следующим образом:

  ```
  ZoneId z = ZoneId.of("Europe/Moscow");
  ```

  Получить массив с названиями всех поддерживаемых часовых зон позволяет статический метод `getAvailableZoneIds()` из класса `ZoneId`. Чтобы указать зону по умолчанию, достаточно вызвать статический метод `systemDefault()`. Пример:

  ```
  LocalDateTime dt = LocalDateTime.now();
  ZonedDateTime zdt = dt.atZone(ZoneId.systemDefault());
  System.out.println(zdt);
  // 2018-03-14T03:56:24.158531200+03:00[Europe/Moscow]
  ```

  Не забудьте импортировать задействованные классы с помощью инструкций:

  ```
  import java.time.ZoneId;
  import java.time.ZonedDateTime;
  ```

- `atOffset()` — возвращает экземпляр класса `OffsetDateTime`. Формат метода:

  ```
  public OffsetDateTime atOffset(ZoneOffset offset)
  ```

  В параметре `offset` указывается экземпляр класса `ZoneOffset`. Создать объект этого класса можно с помощью статических методов `ofHours()` и `of()` из класса `ZoneOffset`:

  ```
  ZoneOffset z1 = ZoneOffset.ofHours(3);
  ZoneOffset z2 = ZoneOffset.of("+3");
  System.out.println(z1.toString()); // +03:00
  System.out.println(z2.toString()); // +03:00
  ```

  Пример использования метода `atOffset()`:

  ```
  LocalDateTime dt = LocalDateTime.now();
  OffsetDateTime odt = dt.atOffset(ZoneOffset.ofHours(3));
  System.out.println(odt);
  // 2018-03-14T03:58:58.430253100+03:00
  ```

Не забудьте импортировать задействованные классы с помощью инструкций:

```
import java.time.ZoneOffset;
import java.time.OffsetDateTime;
```

Классы `ZonedDateTime` и `OffsetDateTime` имеют практически такие же методы, что и класс `LocalDateTime`, за исключением того, что они дополнительно содержат информацию о часовом поясе. Поэтому мы не станем здесь повторяться, а рассмотрим только дополнительные методы в классе `ZonedDateTime`. Описание остальных методов вы найдете в *разд. 9.3*. Информацию по классу `OffsetDateTime` можно найти в документации.

Создать объект класса `ZonedDateTime` позволяют следующие статические методы:

- `now()` — создает объект на основе текущего системного времени. Форматы метода:

```
public static ZonedDateTime now()
public static ZonedDateTime now(ZoneId zone)
```

Первый формат использует часовой пояс из системных настроек:

```
ZonedDateTime zdt = ZonedDateTime.now();
System.out.println(zdt);
// 2018-03-14T04:01:10.627549300+03:00[Europe/Moscow]
```

Второй формат позволяет сразу указать часовой пояс. При этом текущее системное время будет преобразовано во время в указанном часовом поясе. Получим текущую дату и время в Париже:

```
ZonedDateTime zdt = ZonedDateTime.now(ZoneId.of("Europe/Paris"));
System.out.println(zdt);
// 2018-03-14T02:01:58.383009600+01:00[Europe/Paris]
DateTimeFormatter dtf = DateTimeFormatter.ofPattern(
                                    "dd.MM.uuuu HH:mm");
System.out.println(zdt.format(dtf));
// 14.03.2018 02:01
```

- `of()` — создает объект на основе отдельных компонентов даты и времени или объектов, а также часового пояса. Форматы метода:

```
public static ZonedDateTime of(int year, int month, int day,
                               int hour, int minute, int second,
                               int nanoOfSecond, ZoneId zone)
public static ZonedDateTime of(LocalDate date, LocalTime time,
                               ZoneId zone)
public static ZonedDateTime of(LocalDateTime dateTime,
                               ZoneId zone)
```

Пример указания отдельных компонентов даты, времени и часового пояса:

```
ZonedDateTime zdt = ZonedDateTime.of(2018, 5, 1,
                    21, 34, 0, 0, ZoneId.of("Europe/Paris"));
System.out.println(zdt);
// 2018-05-01T21:34+02:00[Europe/Paris]
```

Второй формат создает объект на основе объектов классов `LocalDate` и `LocalTime`, а также часового пояса:

```
LocalDate d = LocalDate.of(2018, 5, 1);
LocalTime t = LocalTime.of(21, 34, 0, 0);
ZonedDateTime zdt = ZonedDateTime.of(d, t,
                    ZoneId.of("Europe/Paris"));
System.out.println(zdt);
// 2018-05-01T21:34+02:00[Europe/Paris]
```

Третий формат создает объект на основе объекта класса `LocalDateTime` и часового пояса:

```
LocalDateTime dt = LocalDateTime.of(2018, 5, 1, 21, 34);
ZonedDateTime zdt = ZonedDateTime.of(dt,
                    ZoneId.of("Europe/Paris"));
System.out.println(zdt);
// 2018-05-01T21:34+02:00[Europe/Paris]
```

- `ofInstant()` — создает объект на основе объекта класса `Instant` и часового пояса. Формат метода:

```
public static ZonedDateTime ofInstant(Instant instant,
                                      ZoneId zone)
```

Пример:

```
Instant ins = Instant.now();
ZonedDateTime zdt = ZonedDateTime.ofInstant(ins,
                    ZoneId.of("Europe/Paris"));
System.out.println(zdt);
// 2018-03-14T02:06:19.963426200+01:00[Europe/Paris]
```

- `parse()` — создает объект из строки специального формата. Форматы метода:

```
public static ZonedDateTime parse(CharSequence text)
public static ZonedDateTime parse(CharSequence text,
                                  DateTimeFormatter formatter)
```

Первый вариант метода использует формат по умолчанию:

```
ZonedDateTime zdt = ZonedDateTime.parse(
   "2018-05-01T21:34:19+02:00[Europe/Paris]");
System.out.println(zdt.toString());
// 2018-05-01T21:34:19+02:00[Europe/Paris]
```

Второй вариант метода позволяет указать формат с помощью объекта класса `DateTimeFormatter`:

```
DateTimeFormatter dtf =
   DateTimeFormatter.ofPattern("uuuu-MM-dd HH:mm:ss VV");
ZonedDateTime zdt = ZonedDateTime.parse(
   "2018-05-01 21:34:19 Europe/Paris", dtf);
System.out.println(zdt.toString());
// 2018-05-01T21:34:19+02:00[Europe/Paris]
```

Если часовой пояс отсутствует в строке, то его можно указать с помощью метода `withZone()` из класса `DateTimeFormatter`:

```
DateTimeFormatter dtf =
   DateTimeFormatter.ofPattern("uuuu-MM-dd HH:mm:ss");
dtf = dtf.withZone(ZoneId.of("Europe/Paris"));
ZonedDateTime zdt = ZonedDateTime.parse(
                    "2018-05-01 21:34:19", dtf);
System.out.println(zdt.toString());
// 2018-05-01T21:34:19+02:00[Europe/Paris]
```

Получить название часового пояса и смещение позволяют следующие методы:

- `getZone()` — возвращает объект класса `ZoneId`. Формат метода:

```
public ZoneId getZone()
```

- `getOffset()` — возвращает объект класса `ZoneOffset`. Формат метода:

```
public ZoneOffset getOffset()
```

Пример:

```
ZonedDateTime zdt = ZonedDateTime.of(2018, 5, 1,
                       21, 34, 0, 0, ZoneId.of("Europe/Paris"));
ZoneId zid = zdt.getZone();
System.out.println(zid.toString()); // Europe/Paris
ZoneOffset zo = zdt.getOffset();
System.out.println(zo.toString());  // +02:00
```

Преобразовать дату и время из одного часового пояса в другой позволяет метод `withZoneSameInstant()`. Формат метода:

```
public ZonedDateTime withZoneSameInstant(ZoneId zone)
```

В качестве примера выведем текущую дату и время в Москве, Париже, Лондоне, Брюсселе, Якутске и Хельсинки (листинг 9.8).

Листинг 9.8. Получение времени в разных часовых поясах

```
import java.time.*;
import java.time.format.DateTimeFormatter;

public class MyClass {
   public static void main(String[] args) {
      DateTimeFormatter dtf =
         DateTimeFormatter.ofPattern("dd.MM.uuuu HH:mm");
      String[] zones = {"Europe/Moscow", "Europe/Paris",
                        "Europe/London", "Europe/Brussels",
                        "Asia/Yakutsk", "Europe/Helsinki"};
      ZonedDateTime zdt = ZonedDateTime.now();
      for (int i = 0; i < zones.length; i++) {
         zdt = zdt.withZoneSameInstant(ZoneId.of(zones[i]));
```

```
            System.out.println(zones[i] + ": " + zdt.format(dtf));
        }
    }
}
```

Примерный результат:

```
Europe/Moscow: 14.03.2018 04:19
Europe/Paris: 14.03.2018 02:19
Europe/London: 14.03.2018 01:19
Europe/Brussels: 14.03.2018 02:19
Asia/Yakutsk: 14.03.2018 10:19
Europe/Helsinki: 14.03.2018 03:19
```

ГЛАВА 10

Пользовательские методы

Метод — это именованный фрагмент кода внутри класса, который можно вызвать из другого метода того же класса или из метода другого класса, при условии, что метод является статическим и общедоступным. В предыдущих главах мы уже не раз использовали методы — например, с помощью статического метода `currentTimeMillis()` из класса `System` получали количество миллисекунд, прошедших с 1 января 1970 года. В этой главе мы рассмотрим создание пользовательских методов, которые позволят уменьшить избыточность программного кода и повысить его структурированность.

Если вы знакомы с другими языками программирования, то вам известны понятия «функция» или «процедура». По существу, статический метод примерно то же самое, но он всегда привязан к какому-либо классу и не может быть определен вне класса. Процедурного стиля программирования в языке Java нет, поэтому любая функция или процедура всегда являются методом класса. Методы в языке Java могут быть статическими и обычными. *Статический* метод, по существу, — это функция, объявленная в пространстве имен класса. Статический общедоступный метод можно вызвать из любого места программы без необходимости создания экземпляра класса. *Обычный* метод привязан к классу и может быть вызван только через экземпляр класса. В этой главе мы рассмотрим статические методы, а в следующей главе приступим к изучению объектно-ориентированного программирования и обычным методам.

10.1. Создание статического метода

Статический общедоступный метод описывается по следующей схеме:

```
public static <Тип результата> <Название метода>(
                    [[final] <Тип> <Название параметра1>
                    [, ..., [final] <Тип> <Название параметраN>]]) {
   [<Тело метода>]
   [return[ <Возвращаемое значение>];]
}
```

Ключевое слово `public` означает, что метод является общедоступным, и к нему можно обратиться как внутри класса, так и из другого класса. Существуют и другие модификаторы доступа, которые мы рассмотрим в следующей главе. Пока же упростим подходы и сделаем все методы общедоступными.

Ключевое слово `static` означает, что метод является статическим и может быть вызван без необходимости создания экземпляра класса. Если это ключевое слово не указать, то метод будет обычным.

Параметр `<Тип результата>` задает тип значения, которое возвращает метод с помощью оператора `return`. Если метод не возвращает никакого значения, то вместо типа указывается ключевое слово `void`.

Название метода должно быть допустимым уникальным идентификатором, к которому предъявляются такие же требования, что и к названиям переменных. Название метода не может начинаться с цифры и зависит от регистра символов. В языке Java принято указывать название метода со строчной (маленькой) буквы (например, `main`). Если название состоит из нескольких слов, то первое слово задается со строчной буквы, а последующие слова — с прописной (заглавной) буквы слитно с предыдущими (например, `nanoTime` или `currentTimeMillis`).

После названия метода, внутри круглых скобок, указываются типы и названия параметров через запятую. Если метод не принимает параметров, то указываются только круглые скобки. Название параметра является локальной переменной. Эта переменная создается при вызове метода, а после выхода из метода удаляется. Таким образом, локальная переменная видна только внутри метода. Если название локальной переменной совпадает с названием глобальной переменной, то все операции будут производиться с локальной переменной, а значение глобальной не изменится. Чтобы в этом случае обратиться к глобальной переменной, необходимо перед названием переменной указать название класса, после которого идет символ «точка»:

```
public static int sum(int x, int y) {
   int z = x + y;       // Обращение к локальной переменной x
   z = MyClass.x + z;   // Обращение к глобальной переменной x
   return z;
}
```

Если значение параметра не изменяется внутри метода, то его можно объявить константным с помощью ключевого слова `final`:

```
public static void print(final int x) {
   System.out.println(x);
}
```

После описания параметров, внутри фигурных скобок, размещаются инструкции, которые будут выполняться при каждом вызове метода. Фигурные скобки указываются в любом случае, даже если тело метода состоит только из одной инструкции или вообще не содержит инструкций. Точка с запятой после закрывающей фигурной скобки не ставится. Существует несколько стилей размещения фигурных скобок:

```
// Стиль 1
<Название метода> (<Параметры>) {
   // Инструкции
}
// Стиль 2
<Название метода> (<Параметры>)
{
   // Инструкции
}
```

Какой стиль использовать, зависит от личного предпочтения программиста или от правил оформления кода, принятых в той или иной фирме. Главное, чтобы стиль оформления внутри одной программы был одинаковым. Многие программисты считают `Стиль 2` наиболее приемлемым, т. к. открывающая и закрывающая скобки расположены друг под другом. Однако в этой книге мы будем использовать `Стиль 1`, поскольку этот стиль мне более по душе. Ведь если размещать инструкции с равным отступом, то блок кода выделяется визуально, и следить за положением фигурных скобок просто излишне. Тем более, что редактор кода позволяет подсветить парные скобки.

Вернуть значение из метода позволяет оператор `return`. Если перед названием метода указано ключевое слово `void`, то оператора `return` может не быть. Однако если необходимо досрочно прервать выполнение метода, то оператор `return` указывается без возвращаемого значения. После исполнения этого оператора выполнение инструкций внутри метода останавливается, и управление передается обратно в точку вызова метода. Это означает, что инструкции после оператора `return` никогда не будут выполнены:

```
public static void test() {
   System.out.println("Текст до оператора return");
   if (10 > 0) return;
   System.out.println("Эта инструкция никогда не будет выполнена!");
}
```

При использовании оператора `return` не должно быть неоднозначных ситуаций. Например, в этом случае возвращаемое значение зависит от условия:

```
public static int test(int x, int y) {
   if (x > 0) {
      return x + y;
   }
}
```

Чтобы избежать подобной ситуации, следует в конце тела метода вставить оператор `return` со значением по умолчанию:

```
public static int test(int x, int y) {
   if (x > 0) {
      return x + y;
   }
   return x;
}
```

10.2. Вызов статического метода

При вызове статического метода внутри класса указывается название метода, после которого, внутри круглых скобок, передаются значения. Если метод не принимает параметров, то указываются только круглые скобки. Если метод возвращает значение, то его можно присвоить переменной или просто проигнорировать. Необходимо заметить, что количество и тип параметров в определении метода должны совпадать с количеством и типом параметров при вызове, иначе будет выведено сообщение об ошибке. Пример вызова трех методов:

```
printOK();              // Вызываем метод без параметров
print("Сообщение");     // Метод выведет текст Сообщение
n = sum(3, 5);          // Переменной n будет присвоено значение 8
```

Переданные значения присваиваются переменным, расположенным в той же позиции в определении метода. Так, при использовании метода `sum()` переменной `x` будет присвоено значение `3`, а переменной `y` — значение `5`. Результат выполнения метода присваивается переменной `n`.

В качестве примера создадим три разных статических метода и вызовем их (листинг 10.1).

Листинг 10.1. Создание статических методов и их вызов

```
public class MyClass {
   // Определения методов
   public static void main(String[] args) {
      int n;
      // Вызов методов
      n = sum(3, 5);
      print("Сообщение");                    // Сообщение
      print("n = " + n);                    // n = 8
      printOK();                            // OK
   }

   public static int sum(int x, int y) {    // Два параметра
      return x + y;
   }

   public static void print(String s) {     // Один параметр
      System.out.println(s);
   }

   public static void printOK() {           // Без параметров
      System.out.println("OK");
   }
}
```

Если определения статических методов расположены в одном классе, а вызов производится в методе другого класса, то при вызове перед названием метода указывается название класса и оператор «точка». Кроме того, если класс находится в другом файле, необходимо предварительно импортировать этот класс с помощью оператора `import`. В качестве примера создадим в том же файле класс `Class1` с методом `print()` и вызовем его из метода `main()` класса `MyClass` (листинг 10.2). Кроме того, импортируем класс `LocalDate` и вызовем метод `now()` из этого класса.

Листинг 10.2. Вызов статических методов из другого класса

```
import java.time.LocalDate;                    // Импорт класса

public class MyClass {
   public static void main(String[] args) {
      // Вызов методов из другого класса
      Class1.print("Сообщение");
      // Вызов метода из другого пакета
      LocalDate d = LocalDate.now();
      Class1.print(d.toString());
   }
}

class Class1 {
   public static void print(String s) {
      System.out.println(s);
   }
}
```

10.3. Перегрузка методов

В предыдущем разделе мы создали метод `sum()`, предназначенный для суммирования двух целых чисел. В один прекрасный момент возникнет ситуация, когда потребуется произвести суммирование вещественных чисел. Что в этом случае делать? Создавать метод с другим названием? В языке Java существует выход из этой ситуации — *перегрузка метода*. Перегрузка метода — это возможность использования одного названия для нескольких методов, различающихся типом параметров или их количеством. Изменения только типа возвращаемого значения недостаточно для перегрузки метода. В качестве примера перегрузим метод `sum()` таким образом, чтобы его название можно было использовать для суммирования как целых чисел, так и вещественных (листинг 10.3).

Листинг 10.3. Перегрузка методов

```
public class MyClass {
   public static void main(String[] args) {
      // Суммирование целых чисел
      System.out.println(sum(10, 20));      // 30
```

```
        // Суммирование вещественных чисел
        System.out.println(sum(10.5, 20.7)); // 31.2
        // Какой метод будет вызван?
        System.out.println(sum(10L, 20L));
    }

    public static int sum(int x, int y) {
        return x + y;
    }

    public static double sum(double x, double y) {
        return x + y;
    }
}
```

Если при вызове метода указаны значения, имеющие другой тип данных, — например, методу `sum()` были переданы числа типа `long`, — то будет выбран перегруженный метод с типом, в который можно автоматически преобразовать значения без потерь. Правда, при условии, что тип возвращаемого значения также может быть преобразован без потерь. В нашем примере мы просто выводим результат, и тип возвращаемого значения не важен, поэтому будет выбран метод с типом параметров `double`. Иными словами, тип `long` преобразуется в тип `double`, и результат будет иметь тип `double`. Если преобразование невозможно, то компилятор выведет сообщение об ошибке.

Перегрузка методов также используется для реализации параметров со значениями по умолчанию, т. к. в языке Java отсутствует возможность присвоить параметру какое-либо начальное значение. В этом случае создаются два метода с разным количеством параметров, после чего в одном из методов вызывается другой метод, и ему передается значение по умолчанию (листинг 10.4).

Листинг 10.4. Параметры со значениями по умолчанию

```
public class MyClass {
    public static void main(String[] args) {
        print("Произвольное сообщение");  // Произвольное сообщение
        print();                          // Сообщение по умолчанию
    }

    public static void print(String s) {
        System.out.println(s);
    }

    public static void print() {
        print("Сообщение по умолчанию"); // Вызываем первый метод
    }
}
```

10.4. Способы передачи параметров в метод

Как вы уже знаете, после названия метода, внутри круглых скобок, указываются типы и названия параметров через запятую. Если метод не принимает параметров, то указываются только круглые скобки. Название параметра является локальной переменной. Эта переменная создается при вызове метода, а после выхода из метода удаляется. Таким образом, локальная переменная видна только внутри метода, и ее значение между вызовами не сохраняется. Если название локальной переменной совпадает с названием глобальной переменной, то все операции будут производиться с локальной переменной, а значение глобальной не изменится. Чтобы в этом случае обратиться к глобальной переменной, необходимо перед ее названием указать название класса и оператор «точка»:

```
MyClass.x
```

При вызове метода указывается название метода, после которого, внутри круглых скобок, передаются значения. Если метод не принимает параметров, то указываются только круглые скобки. Количество и тип параметров в определении функции должны совпадать с количеством и типом параметров при вызове. Если тип не совпадает, то производится попытка автоматически преобразовать типы без потерь данных. Если это невозможно, то будет выведено сообщение об ошибке. Переданные значения присваиваются переменным, расположенным в той же позиции в определении метода. Так, при вызове метода `sum(10, 20)` (формат `int sum(int x, int y)`) переменной `x` будет присвоено значение `10`, а переменной `y` — значение `20`.

В метод передается копия значения переменной. То есть изменение значения внутри функции не затронет значения исходной переменной. Пример передачи параметра по значению приведен в листинге 10.5.

Листинг 10.5. Передача параметра по значению

```
public class MyClass {
   public static void main(String[] args) {
      int x = 10;
      func(x);
      System.out.println(x); // 10, а не 30
   }

   public static void func(int x) {
      x = x + 20; // Значение нигде не сохраняется!
   }
}
```

Передача элементарных типов и неизменяемых объектов (например, строк) никогда не изменяет значения исходных переменных. Однако если передается изменяемый объект, то копируется лишь ссылка, а не объект. Опять-таки копируется значение, но значением таких переменных является ссылка. Если мы внутри метода присвоим

переменной другое значение, то оно не затронет значения исходной переменной, но попытка обратиться через методы к объекту изменит свойства объекта. Так как копируется ссылка, то изменение отразится и на исходной переменной (листинг 10.6).

Листинг 10.6. Передача ссылки на изменяемый объект

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class MyClass {
   public static void main(String[] args) {
      GregorianCalendar d = new GregorianCalendar();
      System.out.printf("%tc\n", d);
      // пт мар. 23 23:39:46 MSK 2018
      func1(d);
      System.out.printf("%tc\n", d); // Не изменилось
      // пт мар. 23 23:39:46 MSK 2018
      func2(d);
      System.out.printf("%tc\n", d); // Изменилось!!!
      // вт апр. 24 23:39:46 MSK 2018
   }

   public static void func1(GregorianCalendar d) {
      d = new GregorianCalendar(2018, Calendar.APRIL, 24);
   }

   public static void func2(GregorianCalendar d) {
      d.set(2018, Calendar.APRIL, 24);
   }
}
```

Итак, внутри метода `func1()` мы пытаемся присвоить другое значение. Так как переменные при вызове и в параметре являются разными (хоть они и одноименные), то попытка присвоить значение внутри метода не изменит значения переменной при вызове. Обе переменные содержат ссылку на один объект, но это разные переменные. Присваивание значения внутри метода лишь изменит количество ссылок на объект, а не сам объект.

Внутри метода `func2()` мы обращаемся к методу объекта и изменяем значение компонентов даты с помощью метода `set()`. Здесь мы уже имеем дело не со ссылкой, а с самим объектом. В результате, т. к. исходная переменная содержит ссылку на тот же самый объект, значение изменилось и в исходной переменной.

Если нужно избежать изменения объекта внутри метода, то следует создать копию объекта и передать ее в метод. Создать копию объекта позволяет метод `clone()`. Формат метода:

```
public Object clone()
```

Так как метод возвращает экземпляр класса `Object`, необходимо выполнить приведение типов к исходному типу объекта (листинг 10.7).

Листинг 10.7. Передача копии объекта

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class MyClass {
   public static void main(String[] args) {
      GregorianCalendar d = new GregorianCalendar();
      System.out.printf("%tc\n", d);
      // пт мар. 23 23:39:46 MSK 2018
      func((GregorianCalendar)d.clone()); // Передаем копию
      System.out.printf("%tc\n", d);      // Не изменилось
      // пт мар. 23 23:39:46 MSK 2018
   }

   public static void func(GregorianCalendar d) {
      d.set(2018, Calendar.APRIL, 24);
   }
}
```

В большинстве случаев передача ссылки на объект это более благо, нежели зло. Ведь объекты могут быть очень большими, и создание копии может повлечь за собой напрасную трату ресурсов. Создавайте копию объекта, только если это действительно важно.

10.5. Передача и возвращение массивов

При указании массива в качестве значения параметра копируется ссылка на массив. Сам массив не передается. Все точно так же, как и с изменяемыми объектами. Если мы изменим массив внутри метода, то изменения будут доступны и через исходную переменную (листинг 10.8).

Листинг 10.8. Передача массива в метод

```
import java.util.Arrays;

public class MyClass {
   public static void main(String[] args) {
      int[] arr = {1, 2, 3, 4, 5};
      System.out.println(Arrays.toString(arr));
      // [1, 2, 3, 4, 5]
      func(arr);
      System.out.println(Arrays.toString(arr)); // Изменился!!!
```

```
        // [9, 2, 3, 4, 5]
    }

    public static void func(int[] arr) {
        arr[0] = 9;
    }
}
```

Для передачи массива в метод и возвращения массива можно использовать анонимные массивы. В этом случае мы не создаем какую-либо промежуточную переменную, а сразу передаем ссылку на анонимный массив. В качестве примера передадим анонимный массив в метод и внутри этого метода преобразуем элементы в строку через указанный разделитель (листинг 10.9).

Листинг 10.9. Передача анонимного массива в метод

```
public class MyClass {
    public static void main(String[] args) {
        String result = "";
        // Передаем анонимный массив
        result = join(new int[] {1, 2, 3, 4, 5}, "; ");
        System.out.println(result);
    }

    public static String join(int[] a, String s) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < a.length; i++) {
            sb.append(String.valueOf(a[i]));
            if (i != a.length - 1) sb.append(s);
        }
        return sb.toString();
    }
}
```

Возвращение массива из метода выполняется точно так же, как и возвращение значения другого типа. Мы не можем вернуть сам массив — всегда возвращается только ссылка на массив, например, созданный внутри метода. Передадим массив в метод, внутри которого умножим каждый элемент массива на число, указанное во втором параметре, и вернем новый массив (листинг 10.10).

Листинг 10.10. Возвращение массива из метода

```
import java.util.Arrays;

public class MyClass {
    public static void main(String[] args) {
        int[] arr1 = {1, 2, 3, 4, 5};
```

```
        int[] arr2;
        arr2 = changeArr(arr1, 2);
        System.out.println(Arrays.toString(arr1));
        // [1, 2, 3, 4, 5]
        System.out.println(Arrays.toString(arr2));
        // [2, 4, 6, 8, 10]
    }

    public static int[] changeArr(int[] a, int x) {
        int[] arr = new int[a.length];
        for (int i = 0; i < a.length; i++) {
            arr[i] = a[i] * x;
        }
        return arr; // Возвращаем новый массив
    }
}
```

При возврате массива иногда может случиться ситуация, что и возвращать-то нечего. Например, перебрали все элементы исходного массива и не нашли элементов, соответствующих условию. В этом случае пригодится массив нулевого размера, который можно вернуть вместо значения `null`. Пример создания массива нулевого размера:

```
int[] arr = new int[0];
System.out.println(Arrays.toString(arr)); // []
System.out.println(arr.length);           // 0
```

10.6. Передача произвольного количества значений

В метод можно передать произвольное количество значений. Для этого используется следующий синтаксис описания параметра:

```
<Тип>... <Название переменной>
```

Если метод принимает какое-либо количество обязательных параметров, то объявление параметра с переменным количеством значений должно быть последним. Например, мы много раз в предыдущих главах пользовались методом `printf()`. Формат метода:

```
public PrintStream printf(String format, Object... args)
```

Параметр `format` является обязательным параметром, поэтому он расположен перед параметром `args`, которой может принимать произвольное количество значений:

```
System.out.printf("%d %d", 10, 20);           // 10 20
System.out.printf("%d %d %.2f", 10, 20, 3.5); // 10 20 3,50
```

Если ожидается произвольное количество значений, то их может не быть и вовсе:

```
System.out.printf("10 20");                   // 10 20
```

Получить количество переданных значений позволяет свойство `length`. В качестве примера выполним суммирование произвольного количества целых чисел (листинг 10.11).

Листинг 10.11. Суммирование произвольного количества целых чисел

```
public class MyClass {
   public static void main(String[] args) {
      System.out.println(sum(1, 2, 3));       // 6
      System.out.println(sum(1, 2, 3, 4, 5)); // 15
      int[] arr = {1, 2, 3, 4, 5};
      System.out.println(sum(arr));           // 15
   }

   public static int sum(int... a) {
      int result = 0;
      System.out.println("Значений: " + a.length);
      for (int b: a) {
         result += b;
      }
      return result;
   }
}
```

Обратите внимание: вместо указания значений через запятую мы можем передать массив со значениями:

```
int[] arr = {1, 2, 3, 4, 5};
System.out.println(sum(arr));           // 15
```

Если в качестве типа указать класс `Object`, то в метод можно будет передать произвольное количество значений любого типа (листинг 10.12).

Листинг 10.12. Передача произвольного количества значений любого типа

```
import java.util.Date;

public class MyClass {
   public static void main(String[] args) {
      func(1, 2.5, "строка");
      func(1, 2.5, 300L, "строка", new Date());
   }

   public static void func(Object... a) {
      for (Object b: a) {
         System.out.println(b);
      }
   }
}
```

10.7. Рекурсия

Рекурсия — это возможность метода вызывать самого себя. При каждом вызове метода создается новый набор локальных переменных. Рекурсию удобно использовать для перебора объекта, имеющего заранее неизвестную структуру, или выполнения неопределенного количества операций. Типичным применением рекурсии является вычисление факториала числа (листинг 10.13).

Листинг 10.13. Вычисление факториала

```
public class MyClass {
   public static void main(String[] args) {
      for (long i = 3; i < 11; i++) {
         System.out.println(i + "! = " + factorial(i));
      }
   }

   public static long factorial(long n) {
      if (n <= 1) return 1;
      else return n * factorial(n - 1);
   }
}
```

Результат выполнения:

```
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

ГЛАВА 11

Объектно-ориентированное программирование

Объектно-ориентированное программирование (*ООП*) — это способ организации программы, позволяющий использовать один и тот же код многократно. Как вы уже знаете, язык Java является полностью объектно-ориентированным языком. Весь его код всегда размещен внутри блока класса. До этой главы мы рассматривали класс как пространство имен для статических методов. В отличие от использования статических методов, ООП позволяет не только разделить программу на фрагменты, но и описать предметы реального мира в виде объектов, а также организовать связи между этими объектами.

11.1. Основные понятия

Основным «кирпичиком» ООП является *класс*. Класс включает набор переменных (называемых *полями* или *свойствами* класса) и методов для управления этими переменными. В совокупности поля и методы называются *членами класса*. После создания класса его название становится новым типом данных. Тем самым пользовательские классы расширяют возможности языка Java.

На основе класса можно создать множество объектов. При этом для каждого объекта создается свой набор локальных переменных. Например, при изучении строк мы рассматривали класс `String`. При объявлении переменной название класса указывалось вместо типа данных:

```
String s1 = "Строка1";
String s2 = "Строка2";
```

В этом примере определены два объекта: `s1` и `s2`, которые являются экземплярами класса `String`. Каждый объект хранит информацию о своей строке, указанной при инициализации. Изменение одного объекта не затрагивает значение другого объекта.

Чтобы иметь возможность манипулировать строкой, класс `String` предоставляет множество методов. Например, с помощью метода `length()` мы получали количест-

во символов в строке. При вызове метода его название указывается после названия объекта через точку:

```
System.out.println(s1.length());
```

Помимо методов, класс `String` перегружает операторы: + и +=. Например, для конкатенации используется оператор +:

```
s1 = s1 + s2;
```

Следует заметить, что язык Java не поддерживает перегрузку операторов для пользовательских классов. По существу, класс `String` является единственным классом, для которого перегрузка выполнена.

Проводя аналогию с реальным миром, можно сказать, что телевизор является экземпляром класса (объектом), а проект, по которому создавался телевизор, представляет собой класс. По этому проекту можно создать множество телевизоров (множество экземпляров класса). Кнопки и разъемы на корпусе телевизора, да и сам корпус, являются отдельными объектами соответствующего класса. Сам процесс нажатия кнопки, приводящий к включению или выключению телевизора, переключению канала и т. д., выполняется с помощью методов. Для сохранения текущего состояния кнопки, а также размеров кнопки, ее цвета и др. предназначены поля. Одна и та же кнопка может выполнять разные действия. Например, одиночное нажатие кнопки выполняет одно действие, а удержание кнопки (перегрузка метода) в течение пяти секунд приводит к выполнению совершенно другого действия.

Все содержимое телевизора находится внутри корпуса и скрыто от глаз. Чтобы пользоваться телевизором, абсолютно не нужно знать, как он устроен внутри, достаточно иметь кнопки (интерфейс доступа) и руководство пользователя (документация к классу). Точно так же разработчик класса может предоставить интерфейс доступа, а остальную часть кода защитить от изменения (сокрытие данных в ООП называется *инкапсуляцией*). В дальнейшем разработчик класса имеет возможность изменить внутреннюю реализацию класса, при этом не изменяя интерфейс доступа.

В один прекрасный момент разработчик телевизора решает выпустить новую модель, просто добавив внутрь корпуса DVD-плеер. При этом большинство блоков внутри телевизора останутся от предыдущей модели, а основные изменения коснутся корпуса. Таким образом, телевизор с DVD-плеером *наследует* конструкцию предыдущей модели, добавляет новую функциональность (DVD-плеер) и переопределяет некоторые методы.

В результате наследования в производном классе может измениться поведение какого-либо метода. Например, автобус и гусеничный трактор являются наследниками класса «транспортное средство», в котором определен метод, описывающий способ движения. Совершенно очевидно, что движение автобуса отличается от движения гусеничного трактора. Поэтому класс «автобус» может наследовать метод от класса «транспортное средство», не переопределяя его, а класс «гусеничный трактор» должен этот метод переопределить. Во всех упомянутых здесь классах теперь имеется доступ к одноименному методу, но реализации этого метода в них

различаются. Тем не менее, этот метод выполняет одно и то же действие — движение транспортного средства. В ООП такое явление называется *полиморфизмом*.

Итак, мы познакомились с тремя основными понятиями ООП:

- *инкапсуляция* — сокрытие данных внутри класса;
- *наследование* — возможность создания производных классов на основе базового класса. При этом производный класс автоматически получает возможности базового класса и может добавить новую функциональность или переопределить некоторые его методы;
- *полиморфизм* — смысл действия, которое выполняет одноименный метод, зависит от объекта, над которым это действие выполняется.

Что же должно быть представлено в виде классов, а что в виде методов или полей? Если слово является именем существительным (автомобиль, телевизор, кнопка и т. д.), то оно может быть описано как класс. Метод описывает действие, применяемое к объекту, — например: начать движение, нажать кнопку, изменить цвет и т. д. Поле предназначено для сохранения текущего состояния объекта и его характеристик — например, размера кнопки и ее цвета, признака, нажата она или нет.

11.2. Создание класса и экземпляра класса

Класс описывается с помощью ключевого слова `class` по следующей схеме (для упрощения некоторые элементы временно опущены):

```
[<Модификаторы>] class <Название класса> {
   <Определения членов класса>
}
```

После ключевого слова `class` задается название класса, которое становится новым типом данных. Название класса должно быть допустимым идентификатором, к которому предъявляются такие же требования, что и к названиям переменных. Название класса не может начинаться с цифры и зависит от регистра символов. В языке Java принято указывать название класса с большой буквы (например, `Date`). Если название состоит из нескольких слов, то первое слово пишется с большой буквы, а последующие слова — также с большой буквы и слитно с предыдущими (например, `GregorianCalendar` или `SimpleDateFormat`). Обычно класс размещают в отдельном файле, который имеет такое же название, что и класс, вплоть до регистра символов. Расширение файла, как и обычно, — java. После компиляции будет создан одноименный файл с расширением class, содержащий байт-код класса.

Внутри фигурных скобок располагаются определения членов класса: полей и методов. Порядок их расположения может быть произвольным, но обычно придерживаются одного из стилей:

- вначале идут объявления полей класса, а затем определения методов;
- вначале идут определения методов, а затем объявления полей класса.

Внутри каждой группы также следует придерживаться следующего порядка:

- общедоступные члены;
- члены, область которых ограничена пакетом;
- приватные члены.

Перед названием класса могут быть указаны различные модификаторы. Пока мы ограничимся лишь модификатором `public`, который означает, что класс является общедоступным, и к нему можно обратиться из другого класса. Если модификатор `public` не указан, то класс будет доступен только внутри пакета.

Давайте создадим в редакторе Eclipse новый класс с названием `Point`. Для этого в меню **File** выбираем пункт **New | Class**. В открывшемся окне в поле **Name** вводим `Point`, пакет пока оставляем по умолчанию и нажимаем кнопку **Finish**. Редактор создаст файл Point.java в папке C:\JavaSE\projects\Test\src и откроет его для редактирования в отдельной вкладке. Причем внутри файла уже будет вставлен следующий код:

```
public class Point {

}
```

Определение класса только описывает новый тип данных, а не объявляет переменную. Чтобы объявить переменную, перед ее именем указывается название класса. Давайте внутри метода `main()` класса `MyClass` объявим переменную `p`, имеющую тип `Point`:

```
Point p;
```

Но это только полдела. После объявления переменной необходимо выполнить инициализацию (создать экземпляр класса `Point` и сохранить ссылку на него в переменной `p`). Для этого после оператора присваивания указывается ключевое слово `new`, после которого идут название класса и пустые круглые скобки:

```
Point p;
p = new Point();
```

Как и обычно, объявление переменной и ее инициализацию можно разместить на одной строке:

```
Point p = new Point();
```

Начиная с Java 10, перед переменной вместо названия класса можно указать слово `var`:

```
var p = new Point();
```

Если переменная не инициализирована, то она не ссылается ни на какой объект, и любое обращение к такой переменной приведет к ошибке.

Если мы хотим объявить переменную, но пока не готовы создать экземпляр класса, то переменной можно временно присвоить значение `null`, которое означает, что переменная ни на какой объект не ссылается:

```
Point p = null;
```

Когда мы готовы создать экземпляр класса, то просто присваиваем ссылку на объект переменной:

```
p = new Point();
```

Как только объект нам больше не нужен, присваиваем переменной либо ссылку на другой объект, либо значение `null`:

```
p = null;
```

Все объекты в языке Java создаются в динамической памяти. Виртуальная машина выделяет память при создании объекта и автоматически освобождает память, когда на объект не будет ни одной ссылки в программе. В других языках, например в C++, работа с динамической памятью лежит на плечах программиста, который частенько забывает освободить память, что приводит к «утечке» памяти. В языке Java программист в управлении динамической памятью практически не участвует (ну разве что может создать объект и обнулить количество ссылок на объект), а значит, и совершить ошибку не сможет, и «утечки» памяти не произойдет.

11.3. Объявление полей внутри класса

Объявление поля внутри класса производится по следующей схеме:

```
[<Модификатор>] <Тип> <Название поля>;
```

В параметре `<Модификатор>` может быть указан один из модификаторов доступа:

- `public` — открытое. Поле доступно для внешнего использования;
- `private` — закрытое. Поле доступно только внутри класса, в котором объявлено. Чаще всего поля класса объявляются как закрытые;
- `protected` — защищенное. Поле недоступно для внешнего использования, но доступно для класса, в котором объявлено, и для потомков этого класса, а также внутри классов пакета.

Если модификатор доступа не указан, то используется *модификатор по умолчанию*. Иерархия модификаторов по уровню доступа (от закрытого до полностью общедоступного) выглядит следующим образом:

```
private
по умолчанию
protected
public
```

Модификаторы доступа предназначены для контроля значения полей класса, которые используются только для внутренней реализации класса. Например, если в поле предполагается хранение определенных значений, то перед присвоением значения мы можем проверить соответствие значения некоторому условию внутри общедоступного метода. Если же любой пользователь будет иметь возможность ввести что угодно, минуя нашу проверку, то ни о каком контроле не может быть и речи. Такая концепция сокрытия данных называется *инкапсуляцией*.

Присвоить или получить значение поля можно через экземпляр класса с помощью точечной нотации:

```
<Переменная>.<Название поля> = <Значение>;
<Значение> = <Переменная>.<Название поля>;
```

Добавим внутри блока класса `Point` объявление двух общедоступных целочисленных полей: `x` и `y`:

```
public class Point {
   public int x;
   public int y;
}
```

Теперь внутри метода `main()` класса `MyClass` присвоим полям значения, а затем получим их и выведем в окно консоли:

```
public class MyClass {
   public static void main(String[] args) {
      Point p = new Point();
      p.x = 10;
      p.y = 20;
      System.out.println(p.x + " " + p.y); // 10 20
   }
}
```

Мы можем создать множество экземпляров класса `Point`, и каждый экземпляр будет иметь свои собственные значения полей `x` и `y`, не зависящие от других экземпляров:

```
public class MyClass {
   public static void main(String[] args) {
      Point p = new Point();
      Point p2 = new Point();
      p.x = 10;
      p.y = 20;
      p2.x = 30;
      p2.y = 40;
      System.out.println(p.x + " " + p.y);   // 10 20
      System.out.println(p2.x + " " + p2.y); // 30 40
   }
}
```

Внутри одного класса допускается использование объектов другого класса. В качестве примера используем класс `Point` для описания координат прямоугольника внутри класса `Rectangle` (прямоугольник). Вначале создадим класс `Rectangle`:

```
public class Rectangle {
   public Point topLeft = new Point();
   public Point bottomRight = new Point();
}
```

Теперь создадим экземпляр класса и обратимся к полям:

```
public class MyClass {
   public static void main(String[] args) {
      Rectangle rect = new Rectangle();
      rect.topLeft.x = 0;
      rect.topLeft.y = 0;
      rect.bottomRight.x = 100;
      rect.bottomRight.y = 100;
      System.out.println(rect.topLeft.x + " " +
            rect.topLeft.y + " " +
            rect.bottomRight.x + " " +
            rect.bottomRight.y); // 0 0 100 100
   }
}
```

Как видно из примера, для обращения к полям класса `Point` необходимо вначале получить доступ к полям класса `Rectangle`, а затем уже с помощью точечной нотации можно изменить и значение поля класса `Point`:

```
rect.topLeft.x = 0;
```

11.4. Определение методов внутри класса

Определение обычных методов внутри класса осуществляется по той же схеме, что и при определении статических методов, которые мы рассмотрели в предыдущей главе. Но есть очень существенная разница в области применения. Во-первых, ключевое слово `static` для обычных методов не указывается. Во-вторых, статические методы имеют доступ только к статическим членам класса, тогда как обычные методы имеют доступ и к статическим членам класса, и к обычным.

Перед возвращаемым типом обычно указывается один из модификаторов доступа: `public`, `private` или `protected`. Смысл их точно такой же, что и у модификаторов полей.

В предыдущем разделе мы сделали поля класса `Point` общедоступными и нарушили один из принципов ООП. Давайте это исправим и объявим поля с помощью ключевого слова `private`. Чтобы иметь возможность контролировать изменение значений полей, добавим несколько общедоступных методов (листинг 11.1).

Листинг 11.1. Содержимое файла Point.java

```
public class Point {
   private int x;
   private int y;

   public void setX(int x) {
      this.x = x;
   }
```

```
    public void setY(int y) {
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}
```

Теперь обратиться к полям напрямую через экземпляр класса уже не получится:

```
Point p = new Point();
p.x = 10;              // Ошибка! Поле объявлено как private
```

Чтобы присвоить значение полю `x`, необходимо использовать метод `setX()`. Чтобы обратиться к методу, после названия экземпляра класса ставится точка, а затем указывается название метода и внутри следующих за ним круглых скобок передаются какие-либо значения (если метод не принимает параметры, указываются только круглые скобки):

```
Point p = new Point();
p.setX(10);            // OK
```

Внутри этого метода при необходимости в любой момент времени мы можем добавить код, проверяющий корректность нового значения. Таким образом соблюдается принцип сокрытия данных, называемый *инкапсуляцией*.

Получить значение поля `x` позволяет метод `getX()`:

```
System.out.println(p.getX());
```

> ***Пояснение***
>
> В языке Java принято использовать приставку `set` для имен методов, изменяющих значение поля, и приставку `get` — для имен методов, возвращающих значение поля.

Теперь давайте посмотрим на доступ к полям внутри методов. Когда мы получаем значение поля, то можем к нему обратиться просто по названию. Методы, изменяющие значения полей, принимают одноименные локальные переменные. И эти локальные переменные перекрывают имена одноименных полей. То есть, если мы просто укажем имя поля, то обратимся на самом деле к локальной переменной. Чтобы получить доступ к полю, используется ключевое слово `this`, которое ссылается на экземпляр класса:

```
public void setX(int x) {
    this.x = x;
}
```

Ключевое слово `this` мы могли бы использовать и в методах `getX()` и `getY()`, но там нет конфликта имен. Однако многие программисты всегда явно указывают ключевое слово `this`, чтобы отличать поля класса от локальных переменных.

11.5. Конструкторы класса

Чтобы при создании экземпляра класса присвоить начальные значения полям, необходимо создать метод, имеющий такое же имя, что и название класса. Такой метод называется *конструктором*. Конструктор всегда автоматически вызывается при создании объекта с помощью оператора `new`.

Конструктор может иметь перегруженные версии, отличающиеся типом параметров или их количеством. Тип возвращаемого значения не указывается.

Если внутри класса нет конструктора, то автоматически создается *конструктор по умолчанию*, который не имеет параметров. В предыдущих примерах мы как раз пользовались конструктором по умолчанию:

```
Point p = new Point(); // Вызывается конструктор по умолчанию
System.out.println(p.getX() + " " + p.getY()); // 0 0
```

Если внутри класса объявлен пользовательский конструктор, то конструктор по умолчанию не создается. Это означает, что если вы создали конструктор с одним параметром, то при создании объекта обязательно нужно будет указывать параметр. Чтобы иметь возможность создания объекта без указания параметров, следует дополнительно создать конструктор без параметров. Создадим в классе `Point` два конструктора: один без параметров, а второй с двумя параметрами (листинг 11.2).

Листинг 11.2. Дополненное конструкторами содержимое файла Point.java

```
public class Point {
   private int x;
   private int y;
   // Конструктор без параметров
   public Point() {
      this.x = 0;
      this.y = 0;
   }
   // Конструктор с двумя параметрами
   public Point(int x, int y) {
      setX(x);
      setY(y);
   }

   public void setX(int x) {
      this.x = x;
   }
```

```
    public void setY(int y) {
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}
```

Теперь мы можем создать объект двумя способами:

```
Point p1 = new Point();
System.out.println(p1.getX() + " " + p1.getY()); // 0 0
Point p2 = new Point(20, 30);
System.out.println(p2.getX() + " " + p2.getY()); // 20 30
```

В первом способе вызывается конструктор без параметров, а во втором — конструктор с двумя параметрами.

> ***Примечание***
>
> Если вы знакомы с другими языками программирования, то вам наверняка знакомо такое понятие, как *деструктор класса*. В других языках — это метод, вызываемый перед уничтожением объекта. Внутри этого метода обычно производится освобождение динамической памяти и ресурсов. В языке Java управление динамической памятью осуществляет виртуальная машина, а не программист, поэтому в нем нет такого понятия, как деструктор. Существует метод `finalize()`, но полагаться на него нельзя, т. к. он вообще может быть не вызван или будет вызван, когда виртуальной машине потребуется освободить ресурсы, а не когда это нужно программисту. Кроме того, в Java 9 метод объявлен устаревшим.

11.6. Явная инициализация полей класса

При объявлении полей можно сразу присвоить им начальные значения. Пример явной инициализации:

```
public class Rectangle {
    public Point topLeft = new Point();
    public Point bottomRight = new Point();
}
```

Если в этом случае мы не произведем инициализацию явным образом или внутри конструктора, то эти поля примут значение `null`. Любое обращение к классу `Point` в этом случае станет приводить к ошибке, т. к. объектов в нем существовать не будет.

11.7. Инициализационные блоки

Помимо явной инициализации и использования конструкторов существует третий способ инициализации полей класса. При этом способе инициализация полей производится внутри анонимного блока (фрагмента кода внутри фигурных скобок). Такой блок называется *инициализационным блоком*. Давайте переделаем реализацию класса `Rectangle` и используем инициализационный блок:

```
public class Rectangle {
   public Point topLeft;
   public Point bottomRight;
   // Инициализационный блок
   {
      topLeft = new Point();
      bottomRight = new Point(2, 3);
   }
}
```

Обычно инициализационный блок размещают в самом конце блока класса, т. к. объявления полей должны быть расположены раньше блока.

Итак, мы имеем дело с тремя способами инициализации. А в какой последовательности они выполняются? Давайте рассмотрим это на примере (листинг 11.3).

Листинг 11.3. Порядок инициализации

```
public class MyClass {
   public static void main(String[] args) {
      Class1 c = new Class1();
      System.out.println(c.s);
   }
}

class Class1 {
   public String s = "Явная инициализация";

   public Class1() {
      System.out.println(s);
      s = "Конструктор";
   }
   // Инициализационный блок
   {
      System.out.println(s);
      s = "Блок";
   }
}
```

Результат будет выглядеть следующим образом:

```
Явная инициализация
Блок
Конструктор
```

Обратите внимание на реализацию класса `Class1` — мы поместили ее в одном файле с классом `MyClass`. В этом случае мы не можем использовать модификатор `public`, и класс `Class1` будет доступен только внутри пакета. Обычно мы создаем класс, чтобы им все пользовались, поэтому класс размещаем в одноименном файле и делаем его общедоступным. Однако в учебных целях можно воспользоваться и таким способом. Вряд ли кто-то захочет воспользоваться классом `Class1`.

Инициализационный блок может быть *статическим*. В этом случае перед блоком указывается ключевое слово `static`. Внутри статического блока мы не имеем доступа к обычным полям класса, а только к статическим переменным и константам класса. Инструкции внутри статического блока выполняются только один раз, тогда как инструкции внутри обычного блока выполняются при каждом создании экземпляра класса. Еще раз обращаю ваше внимание на то, что *внутри статического инициализационного блока мы имеем доступ только к статическим членам класса*. Пример использования статического блока приведен в листинге 11.4.

Листинг 11.4. Статический инициализационный блок

```
public class MyClass {
   public static void main(String[] args) {
      System.out.println(Class1.x);  // 10
      System.out.println(Class1.PI); // 3.14
   }
}

class Class1 {
   public static int x;
   public static final double PI;
   // Статический инициализационный блок
   static {
      x = 10;
      PI = 3.14;
   }
}
```

11.8. Вызов одного конструктора из другого

Конструктор класса может иметь множество перегруженных версий с разными параметрами. Например, как вы уже знаете, перегрузка часто используется для реализации параметров со значением по умолчанию. В этом случае нам необходимо вызвать другой конструктор и передать ему значение по умолчанию. Для вызова

одного конструктора из другого внутри одного класса используется следующий синтаксис:

```
this([<Значения>]);
```

Если значения не передаются, то указываются пустые круглые скобки. Если значения указаны, то автоматически будет выбран конструктор, соответствующий количеству параметров и их типу (листинг 11.5).

Листинг 11.5. Вызов одного конструктора из другого

```
public class MyClass {
   public static void main(String[] args) {
      Class1 c = new Class1();
      System.out.println(c.getX()); // 25
      c = new Class1(33);
      System.out.println(c.getX()); // 33
   }
}

class Class1 {
   private int x;

   public Class1() {
      this(25);     // Вызываем конструктор Class1(int x)
   }
   public Class1(int x) {
      this.x = x;
   }
   public int getX() {
      return x;
   }
}
```

11.9. Создание констант класса

Создать константу класса можно несколькими способами:

- объявив *статическую константу класса*. Статическая константа класса существует в единственном экземпляре. Присвоить начальное значение можно при объявлении константы или внутри статического инициализационного блока. Пример объявления и инициализации статической константы класса:

  ```
  public static final double PI = 3.14;
  ```

  Обращение к статической константе класса вне класса может быть выполнено без создания экземпляра. При этом указываются название класса, оператор «точка» и название константы:

  ```
  System.out.println(Class1.PI);
  ```

Можно также обратиться через экземпляр класса, но лучше избегать такого способа:

```
Class1 c = new Class1(10);
System.out.println(c.PI);   // Лучше использовать Class1.PI
```

Внутри класса можно указать просто имя константы, `<Название класса>.<Константа>` или `this.<Константа>`. Последний вариант лучше не использовать;

- объявив *обычную константу класса*. Каждый экземпляр класса может иметь свое значение константы. Присвоить начальное значение можно при объявлении константы, внутри обычного инициализационного блока или внутри конструктора класса. Пример объявления и инициализации обычной константы класса:

```
public final int MY_CONST = 3;
```

Обращение к обычной константе класса вне класса может быть выполнено только после создания экземпляра класса. При этом указывается объект, оператор «точка» и название константы:

```
Class1 c = new Class1(10);
System.out.println(c.MY_CONST);
```

Внутри класса можно указать просто имя константы или `this.<Константа>`.

Как видно из примеров, объявление констант выполняется с помощью ключевого слова `final`. Присвоить значение константе можно только один раз. Любая попытка повторного присваивания ей значения приведет к ошибке. Различные варианты создания констант и способы доступа к ним вне класса показаны в листинге 11.6.

Листинг 11.6. Создание констант внутри класса

```
public class MyClass {
   public static void main(String[] args) {
      // Обратиться можно без создания экземпляра
      System.out.println(Class1.PI);  // 3.14
      Class1 c = new Class1(10);
      // Только через экземпляр класса
      System.out.println(c.MY_CONST); // 10
      c = new Class1(20);
      System.out.println(c.MY_CONST); // 20
   }
}

class Class1 {
   // Обычная константа класса
   public final int MY_CONST;
   // Статическая константа класса
   public static final double PI = 3.14;
```

```
    public Class1(int x) {
        MY_CONST = x;
    }
}
```

Как вы уже знаете, переменная содержит лишь ссылку на объект, а не сам объект. Это обстоятельство важно учитывать при создании констант, ведь константа не должна изменять своего значения. Если мы присваиваем константе начальное значение, имеющее элементарный тип или ссылку на неизменяемый объект (например, присваиваем строку), то все нормально, ведь повторно присвоить какое-либо значение константе нельзя. Однако ситуация будет другой, если мы присвоим константе ссылку на изменяемый объект. Да, впоследствии мы не сможем изменить ссылку, но мы ведь можем обратиться к методу объекта и изменить тем самым сам объект! Помните об этом при создании констант и не допускайте ошибок!

11.10. Статические члены класса

Внутри класса можно создать переменную, константу или метод, которые будут доступны без создания экземпляра класса. Для этого перед объявлением переменной, константы или метода следует указать ключевое слово `static`. Статические члены класса существуют в единственном экземпляре, независимо от количества созданных объектов. Обратите внимание на то, что внутри статического метода невозможно получить доступ к обычным полям, — только к статическим членам класса.

Присвоить начальное значение можно при объявлении переменной (или константы) или внутри статического инициализационного блока. Статические переменные класса инициализируются автоматически, если им не были присвоены начальные значения. Целочисленным переменным присваивается значение `0`, вещественным — `0.0`, логическим — `false`, объектным — `null`. Пример объявления и инициализации:

```
// Статическая переменная
public static int x = 10;
// Статическая константа класса
public static final double PI = 3.14;
```

При обращении к статической переменной или статической константе класса вне класса указывается название класса, оператор «точка» и название переменной или константы:

```
System.out.println(Class1.x);
System.out.println(Class1.PI);
```

Можно также обратиться через экземпляр класса, но лучше избегать такого способа:

```
Class1 c = new Class1();
System.out.println(c.x);   // Лучше использовать Class1.x
System.out.println(c.PI);  // Лучше использовать Class1.PI
```

Внутри класса можно указать просто имя переменной или константы, <Название класса>.<Имя> или this.<Имя>. Последний вариант лучше не использовать.

При вызове статического метода вне класса указывается название класса, оператор «точка» и название метода, после которого внутри круглых скобок передаются значения. Если метод не принимает параметров, то указываются только круглые скобки. Если метод возвращает значение, то его можно присвоить переменной или просто проигнорировать:

```
System.out.println(Class1.sum(1, 2));
```

Можно также обратиться через экземпляр класса, но лучше избегать такого способа:

```
Class1 c = new Class1();
System.out.println(c.sum(1, 2)); // Лучше использовать Class1.sum(1, 2)
```

Внутри класса можно указать просто имя метода, <Название класса>.<Метод> или this.<Метод>. Последний вариант лучше не использовать.

Различные варианты создания статических членов класса и способы доступа к ним вне класса показаны в листинге 11.7.

Листинг 11.7. Статические члены класса

```
public class MyClass {
   public static void main(String[] args) {
      System.out.println(Class1.PI);          // 3.14
      System.out.println(Class1.x);           // 10
      Class1.x += 10;
      System.out.println(Class1.x);           // 20
      System.out.println(Class1.sum(1, 2));   // 3
   }
}

class Class1 {
   // Статическая переменная
   public static int x;
   // Статическая константа класса
   public static final double PI = 3.14;
   // Статический метод
   public static int sum(int i, int j) {
      return i + j;
   }
   // Статический инициализационный блок
   static {
      x = 10;
   }
}
```

11.11. Методы-фабрики

Как вы уже знаете, имя конструктора класса всегда совпадает с именем класса, вплоть до регистра символов. Если нам необходимо дать осмысленное название или вернуть значение другого класса, то можно создать статический метод, который будет использоваться для создания объектов. Название такого метода может быть произвольным и тип возвращаемого значения тоже. Такие методы принято называть *методами-фабриками* (листинг 11.8).

Листинг 11.8. Методы-фабрики

```
public class MyClass {
   public static void main(String[] args) {
      // Так нельзя, у конструктора модификатор private
      // Class1 c1 = new Class1(10);
      Class1 c2 = Class1.getInstance(10);
      Class1 c3 = Class1.getInstance(45);
      System.out.println(c2.getX());  // 10
      System.out.println(c3.getX());  // 45
   }
}

class Class1 {
   private int x;

   private Class1(int x) {
      this.x = x;
   }
   // Статический метод-фабрика
   public static Class1 getInstance(int x) {
      return new Class1(x);
   }
   public int getX() { return this.x; }
}
```

11.12. Наследование

Наследование — это возможность создания *производных классов* на основе *базового класса*. При этом производный класс автоматически получает возможности базового класса и может добавить новую функциональность или переопределить некоторые методы. Базовый класс называют также *суперклассом* или *родительским классом*, а производный класс — *подклассом* или *дочерним классом*. Пример создания иерархии классов приведен в листинге 11.9.

Листинг 11.9. Наследование

```
public class MyClass {
   public static void main(String[] args) {
      C obj = new C();
      obj.func1();        // A.func1()
      obj.func2();        // B.func2()
      obj.func3();        // C.func3()
   }
}

class A {             // Базовый класс
   public void func1() { System.out.println("A.func1()"); }
}
class B extends A {  // Класс B наследует класс A
   public void func2() { System.out.println("B.func2()"); }
}
class C extends B {  // Класс C наследует классы A и B
   public void func3() { System.out.println("C.func3()"); }
}
```

В этом примере вначале класс `B` наследует все члены класса `A`. Затем объявляется класс `C`, который наследует все члены и класса `B`, и класса `A`. Каждый класс добавляет один новый метод. Таким образом, экземпляр класса `C` имеет доступ ко всем методам классов `A` и `B`. Класс `A` называется *базовым классом* или *суперклассом*, а класс `B` — *производным классом* или *подклассом*. В то же время класс `B` является базовым для класса `C`. При наследовании используется следующий формат определения класса:

```
[<Модификатор>] class <Производный класс> extends <Базовый класс> {
   <Описание членов класса>
}
```

Следует учитывать, что наследуются только общедоступные (объявленные с помощью ключевого слова `public`) и защищенные члены (объявленные посредством ключевого слова `protected`). Закрытые члены класса (объявленные с помощью ключевого слова `private`) не наследуются, и доступа к ним внутри производного класса нет.

Для передачи значений конструкторам базовых классов используется следующий синтаксис:

```
super([<Значения>]);
```

Продемонстрируем передачу значений конструкторам базовых классов (листинг 11.10).

Листинг 11.10. Передача значений конструкторам базовых классов

```
public class MyClass {
   public static void main(String[] args) {
      C obj = new C(10, 20, 30);
      System.out.println("A.x = " + obj.x +
         "; B.y = " + obj.x + "; C.z = " + obj.z);
   }
}

class A {
   public int x;
   public A(int x) {                    // Конструктор
      this.x = x;
      System.out.println("A.A()");
   }
}
class B extends A {
   public int y;
   public B(int x, int y) {             // Конструктор
      super(x);                         // Вызов A.A(x)
      this.y = y;
      System.out.println("B.B()");
   }
}
class C extends B {
   public int z;
   public C(int x, int y, int z) {      // Конструктор
      super(x, y);                      // Вызов B.B(x, y)
      this.z = z;
      System.out.println("C.C()");
   }
}
```

Результат выполнения программы:

```
A.A()
B.B()
C.C()
A.x = 10; B.y = 10; C.z = 30
```

Примечание

Язык Java не поддерживает множественное наследование.

11.13. Переопределение методов базового класса

В результате наследования в производном классе может измениться поведение какого-либо метода. В *разд. 11.1* мы уже рассматривали такой случай, однако здесь невредно обратиться к нему и еще раз... Там мы для примера допустили, что автобус и гусеничный трактор являются наследниками класса «транспортное средство», в котором определен метод, описывающий способ движения. Совершенно очевидно, что движение автобуса отличается от движения гусеничного трактора. Поэтому класс «автобус» может наследовать метод от класса «транспортное средство», не переопределяя его, а класс «гусеничный трактор» должен этот метод переопределить. Во всех упомянутых здесь классах теперь имеется доступ к одноименному методу, но реализации этого метода в них различаются. Тем не менее, рассматриваемый метод выполняет одно и то же действие — движение транспортного средства. В ООП такое явление называется *полиморфизмом* (смысл действия, которое выполняет одноименный метод, зависит от объекта, над которым это действие выполняется).

Чтобы переопределить метод базового класса, достаточно в производном классе создать одноименный метод с той же самой сигнатурой. В этом случае будет вызван метод только из производного класса. Чтобы вызвать метод из базового класса, внутри производного класса используется следующий синтаксис:

```
super.<Метод>([<Значения>])
```

Рассмотрим переопределение метода на примере (листинг 11.11).

Листинг 11.11. Переопределение метода базового класса

```
public class MyClass {
   public static void main(String[] args) {
      B obj = new B();
      obj.func();
   }
}

class A {
   public void func() {
      System.out.println("A.func()");
   }
}
class B extends A {
   public void func() {
      System.out.println("B.func()");
      super.func();                    // Вызываем метод базового класса
   }
}
```

В процессе переопределения метода мы можем что-нибудь напутать и создать одноименный метод, но с другой сигнатурой. В результате мы получим перегрузку метода, а не переопределение. Компилятор ведь не знает, что мы хотим сделать, а перегрузка — это допустимая операция, и никаких ошибок здесь нет. Чтобы сообщить компилятору о переопределении, перед описанием метода следует указать аннотацию `@Override`. Обычно ее размещают на отдельной строке перед описанием метода. Увидев аннотацию `@Override`, компилятор проверит наличие метода с указанной сигнатурой в базовых классах. Если такого метода нет, то выведет сообщение об ошибке. В результате мы застрахуем себя от ошибочных действий. Пример указания аннотации:

```
class B extends A {
   @Override
   public void func() {
      System.out.println("B.func()");
      super.func();                    // Вызываем метод базового класса
   }
}
```

11.14. Финальные классы и методы

В ряде случаев может возникнуть необходимость предотвратить наследование всего класса (примером финального класса является класс `String`) или запретить переопределение отдельных методов в производном классе. Для этого используется ключевое слово `final`. Чтобы запретить наследование всего класса, ключевое слово `final` указывается перед ключевым словом `class`:

```
final class A {
   public void func() {
      System.out.println("A.func()");
   }
}
```

При запрете переопределения метода ключевое слово `final` указывается перед типом возвращаемого значения:

```
class A {
   public final void func() {
      System.out.println("A.func()");
   }
}
```

11.15. Абстрактные классы и методы

Абстрактные методы содержат только объявление метода без реализации. Создать экземпляр класса, в котором объявлен абстрактный метод, нельзя. Предполагается, что производный класс должен переопределить метод и реализовать его

функциональность. В языке Java абстрактные методы объявляются с помощью ключевого слова `abstract` по следующей схеме:

```
[<Модификатор>] abstract <Тип> <Название>([<Параметры>]);
```

Обратите внимание: метод не имеет блока. Сразу после списка параметров за закрывающей круглой скобкой ставится точка с запятой. Если внутри класса существует хотя бы один абстрактный метод, то весь класс следует объявить абстрактным, добавив перед ключевым словом `class` ключевое слово `abstract`. Следует учитывать, что абстрактный класс может, помимо абстрактных, содержать и обычные методы. Кроме того, класс может быть объявлен абстрактным, даже если он не содержит абстрактных методов. Пример объявления и замещения абстрактного метода приведен в листинге 11.12.

Листинг 11.12. Абстрактные классы и методы

```
public class MyClass {
   public static void main(String[] args) {
      B obj = new B();
      obj.func();         // B.func()
      obj.print();        // A.print()
   }
}

abstract class A {
   public abstract void func();  // Абстрактный метод
   public void print() {         // Обычный метод
      System.out.println("A.print()");
   }
}
class B extends A {
   @Override
   public void func() {          // Замещаем метод
      System.out.println("B.func()");
   }
}
```

11.16. Вложенные классы

До сих пор мы создавали классы или в отдельных файлах, или в одном файле. Тем не менее, классы могут быть объявлены внутри другого класса или даже внутри какого-либо метода. Такие классы называются *вложенными*.

11.16.1. Обычные вложенные классы

Если определение класса расположено внутри другого класса, то такой класс можно скрыть от других классов. Так как класс вложен внутрь блока внешнего класса,

то он имеет доступ ко всем членам внешнего класса, включая закрытые члены. Для доступа к членам внешнего класса можно указать просто имя члена или воспользоваться следующим синтаксисом:

```
<Имя внешнего класса>.this.<Член класса>
```

Для создания экземпляра вложенного класса внутри внешнего класса используется один из следующих способов:

```
<Переменная> = new <Имя внутреннего класса>();
<Переменная> = this.new <Имя внутреннего класса>();
<Переменная> = new <Имя внешнего класса>.<Имя внутреннего класса>();
```

Пример использования вложенного класса приведен в листинге 11.13.

Листинг 11.13. Вложенные классы

```
public class MyClass {
   public static void main(String[] args) {
      A obj = new A(10);
      obj.func();
   }
}

class A {
   private B b;
   private int x;

   public A(int x) {
      this.b = this.new B(); // или this.b = new B();
                             // или this.b = new A.B();
      this.x = x;
   }

   public void func() {
      System.out.println("A.func()");
      b.func();
   }

   private class B {                    // Вложенный класс
      public void func() {
         System.out.println("A.B.func()");
         System.out.println(A.this.x); // Доступ к A.x
         System.out.println(x);        // Можно и так
      }
   }
}
```

Если вложенный класс объявлен общедоступным (public), защищенным (protected) или модификатор не указан, то мы можем создать экземпляр вложенного класса через экземпляр внешнего класса (листинг 11.14).

Листинг 11.14. Общедоступный вложенный класс

```
public class MyClass {
   public static void main(String[] args) {
      A obj = new A();
      A.B obj2 = obj.new B();          // Создание экземпляра
      obj2.func();
   }
}

class A {
   private int x = 10;

   public class B {                    // Вложенный класс
      public void func() {
         System.out.println("A.B.func()");
         System.out.println(A.this.x);
      }
   }
}
```

11.16.2. Статические вложенные классы

Вложенный класс можно объявить статическим, указав перед ключевым словом `class` ключевое слово `static`. В этом случае вложенный класс имеет доступ только к статическим членам внешнего класса, в том числе и к закрытым. Если статический класс объявлен закрытым, то создать экземпляр этого класса можно только в методах внешнего класса. Если статический класс объявлен общедоступным, то можно создать экземпляр этого класса без создания экземпляра внешнего класса. Для этого используется следующий синтаксис:

```
<Имя внешнего класса>.<Имя внутреннего класса> <Переменная> =
            new <Имя внешнего класса>.<Имя внутреннего класса>();
```

Пример использования общедоступного статического вложенного класса приведен в листинге 11.15.

Листинг 11.15. Статические вложенные классы

```
public class MyClass {
   public static void main(String[] args) {
      A.B obj = new A.B();
      obj.func();
      A obj2 = new A();
      obj2.func();
   }
}
```

```
class A {
   private static int x = 10;
   private B obj = new B();
   public void func() {
      System.out.println("A.func()");
      obj.func();
   }

   public static class B {     // Статический вложенный класс
      public void func() {
         System.out.println("A.B.func()");
         System.out.println(x);
      }
   }
}
```

11.16.3. Локальные вложенные классы

Класс может быть вложен в блок метода какого-либо класса. В этом случае нельзя использовать модификаторы доступа, т. к. вложенный класс всегда будет виден только внутри блока метода. Такие классы называются *локальными вложенными классами*. Внутри локального вложенного класса есть доступ ко всем членам внешнего класса и к локальным константам внутри метода (к обычным локальным переменным доступа нет, исключением являются переменные, которым значение присваивается один раз при инициализации и больше внутри метода не изменяется). Локальные вложенные классы не могут содержать статических членов. Исключением являются статические константы. Создать экземпляр локального вложенного класса можно только после блока, описывающего этот класс.

Пример использования локального вложенного класса приведен в листинге 11.16.

Листинг 11.16. Локальные вложенные классы

```
public class MyClass {
   public static void main(String[] args) {
      A obj = new A();
      obj.func(10);
   }
}

class A {
   private int z = 30;

   public void func(final int x) {
      int y = 20; // Значение присвоено только один раз!

      class B {                    // Локальный вложенный класс
         private int k = 40;
```

```
        public void func() {
            System.out.println("B.func()");
            System.out.println(x); // Доступ к локальной константе
            System.out.println(y);
            System.out.println(z); // Доступ к полям класса A
            System.out.println(A.this.z);
            System.out.println(k); // Доступ к полям класса B
            System.out.println(this.k);
        }
    }

    System.out.println("A.func()");
    B obj = new B();
    obj.func();
  }
}
```

11.16.4. Анонимные вложенные классы

Анонимные вложенные классы также создаются внутри какого-либо метода, но они не имеют названия и должны наследовать какой-либо класс (или реализовывать интерфейс), т. к. не могут иметь конструктора (имя конструктора, как вам уже известно, должно совпадать с именем класса, а этого имени у анонимных классов нет). Но мы можем передать параметры конструктору базового класса, указав их внутри круглых скобок. Схема создания анонимного вложенного класса выглядит следующим образом:

```
<Имя базового класса> <Переменная> =
  new <Имя базового класса>(
        [<Параметры для конструктора базового класса>]) {
  // Определение анонимного класса
};
```

Внутри анонимного вложенного класса есть доступ ко всем общедоступным или защищенным членам базового класса (закрытые члены не наследуются) и к локальным константам внутри метода (к обычным локальным переменным доступа нет, исключением являются переменные, которым значение присваивается один раз при инициализации и больше внутри метода не изменяется). Анонимные вложенные классы не могут содержать статических членов. Исключением являются статические константы.

Пример использования анонимного вложенного класса приведен в листинге 11.17.

Листинг 11.17. Анонимные вложенные классы

```
public class MyClass {
  public static void main(String[] args) {
    final int x = 10;
```

```
      A obj = new A() {              // Анонимный вложенный класс
         private int y = 20;

         @Override
         public void func() {
            System.out.println("?.func()");
            System.out.println(x); // Доступ к локальной константе
            System.out.println(y); // Доступ к полю этого класса
            System.out.println(this.y);
            System.out.println(z); // Доступ к полю класса A
            System.out.println(super.z);
         }
      };
      obj.func();
   }
}

class A {
   public int z = 30;

   public void func() {
      System.out.println("A.func()");
   }
}
```

11.17. Приведение типов

Обратите внимание на код листинга 11.17. Мы присваиваем переменной, имеющей тип базового класса, ссылку на экземпляр производного класса. Переменные в языке Java являются полиморфными и при объявлении типа одного из базовых классов могут хранить ссылки на объекты производных классов. При вызове метода происходит динамическое связывание и вызывается метод производного класса, при условии, что этот метод существует в базовом классе и переопределен в производном классе. Таким образом, в коде из листинга 11.17 будет выполнен метод `func()` из анонимного вложенного класса, а не из класса `A`, — здесь работает принцип полиморфизма: смысл действия, которое выполняет одноименный метод, зависит от объекта, над которым это действие выполняется.

Давайте рассмотрим полиморфизм на примере иерархии классов (листинг 11.18).

Листинг 11.18. Полиморфизм

```
public class MyClass {
   public static void main(String[] args) {
      A objA = new A();
      B objB = new B();
      C objC = new C();
```

```
        objA.func();        // A.func()
        objB.func();        // B.func()
        objC.func();        // C.func()
        objC.func2();       // C.func2()
        A obj1 = new A();
        A obj2 = new B();
        A obj3 = new C();
        obj1.func();        // A.func()
        obj2.func();        // B.func()
        obj3.func();        // C.func()
        // obj3.func2();    // Ошибка
    }
}

class A {
    public void func() {
        System.out.println("A.func()");
    }
}
class B extends A {
    @Override
    public void func() {
        System.out.println("B.func()");
    }
}
class C extends B {
    @Override
    public void func() {
        System.out.println("C.func()");
    }
    public void func2() {
        System.out.println("C.func2()");
    }
}
```

Итак, в первом случае мы создаем объекты классов как обычно, указывая в качестве типа название соответствующего класса. Во втором случае мы присваиваем ссылку переменным, имеющим тип базового класса. При этом используется динамическое связывание, и будет вызван метод производного класса, хотя мы и объявили тип базового класса.

Обратите внимание на вызов метода `func2()` из класса `C`. Здесь мы получим ошибку, поскольку такого метода нет в базовом классе. Чтобы вызвать этот метод, необходимо выполнить приведение типов к типу `C`. Приведение выполняется точно так же, как и приведение элементарных типов, но дополнительно название переменной указывается внутри круглых скобок:

```
((C)obj3).func2();    // C.func2()
```

Вначале мы выполнили приведение типа переменной `obj3` к типу `C`, а затем с помощью точечной нотации обратились к методу `func2()` из класса `C`. Теперь все работает.

Если при операции приведения указать класс, не являющийся наследником, то возникнет ошибка. Чтобы избежать этой ошибки, необходимо перед приведением типов выполнить проверку с помощью оператора `instanceof`. Синтаксис оператора:

```
<Переменная> instanceof <Класс>
```

Выражение вернет значение `true`, если можно выполнить приведение типа к указанному классу, и значение `false` — в противном случае. Пример проверки:

```
if (obj3 instanceof C) {
   ((C)obj3).func2();   // C.func2()
}
```

11.18. Класс *Object*

Каждый класс в языке Java явно или неявно наследует класс `Object`. Например, такой код:

```
class A {
}
```

полностью эквивалентен такому:

```
class A extends Object {
}
```

Следовательно, все пользовательские классы наследуют методы класса `Object`. Рассмотрим наиболее значимые методы класса `Object`:

- `toString()` — возвращает строковое представление объекта. Этот метод вызывается при попытке указания объекта, где ожидается тип `String`, — например, при использовании метода `println()` объекта `System.out` (`System.out.println(obj)`) или при операции конкатенации (`"" + obj`). По умолчанию возвращается название класса и адрес объекта в памяти компьютера. Не очень уж и полезная информация... Но в пользовательских классах мы можем переопределить этот метод и внутри него возвращать нужную нам строку:

```
public class Point {
   private int x;
   private int y;
   public Point() {
      this(0, 0);
   }
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
   @Override
```

```
    public String toString() {
        return "[" + this.x + ", " + this.y + "]";
    }
}
```

Теперь создадим объект и вызовем метод `toString()` разными способами:

```
Point p = new Point(20, 51);
System.out.println(p);              // [20, 51]
System.out.println(p.toString()); // [20, 51]
String s = "" + p;
System.out.println(s);              // [20, 51]
```

- `equals()` — сравнивает два объекта и возвращает `true`, если объекты равны, и `false` — в противном случае. Если мы создадим два объекта класса `Point` с одинаковыми координатами точек и сравним эти объекты с помощью метода `equals()`, то получим значение `false`. Координаты одинаковые, но объекты разные. Чтобы сравнивались координаты, а не ссылки, необходимо переопределить метод `equals()` в классе `Point`:

```
public class Point {
    private int x;
    private int y;
    public Point() {
        this(0, 0);
    }
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    @Override
    public boolean equals(Object obj) {
        // Если параметр равен null
        if (obj == null) return false;
        // Если тот же объект
        if (this == obj) return true;
        // Если классы не совпадают
        if (this.getClass() != obj.getClass())
            return false;
        // Здесь мы знаем, что obj объект класса Point
        // Выполняем приведение к типу Point
        Point p = (Point)obj;
        return this.x == p.x && this.y == p.y;
    }
}
```

Теперь создадим объекты и вызовем метод `equals()` разными способами:

```
Point p1 = new Point(20, 51);
Point p2 = new Point(20, 51);
Point p3 = new Point(1, 51);
```

```
System.out.println(p1.equals(p2)); // true
System.out.println(p1.equals(p3)); // false
p1 = p3;
System.out.println(p1.equals(p3)); // true
```

- `hashCode()` — возвращает хеш-код объекта. *Хеш-код* — это целое число, по умолчанию генерируемое на основе адреса объекта. Если мы сейчас сравним хеш-коды объектов `p1` и `p2` из предыдущего примера, то они будут разными, хотя на самом деле раз мы реализовали метод `equals()`, и он возвращает равенство объектов, то и хеш-коды у них должны совпадать. Если вы переопределили метод `equals()`, то должны переопределить и метод `hashCode()`, и наоборот. Давайте переопределим метод `hashCode()` и вернем свой хеш-код. Для генерации хеш-кода воспользуемся методом `hash()` из класса `Objects`:

```
import java.util.Objects;

public class Point {
   private int x;
   private int y;
   public Point() {
      this(0, 0);
   }
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
   @Override
   public int hashCode() {
      return Objects.hash(this.x, this.y);
   }
}
```

Теперь создадим объекты и вызовем метод `hashCode()` разными способами:

```
Point p1 = new Point(20, 51);
Point p2 = new Point(20, 51);
Point p3 = new Point(51, 20);
System.out.println(p1.hashCode()); // 1632
System.out.println(p2.hashCode()); // 1632
System.out.println(p3.hashCode()); // 2562
```

Честно говоря, при использовании метода `hash()` можно подобрать координаты, которые будут разными, однако хеш-код у них будет одинаковым, но шансы такого совпадения в реальном проекте не высоки. При желании вы можете реализовать свой алгоритм формирования хеш-кода, тем более что класс `Objects` доступен только начиная с Java 7. Вот пример одинакового хеш-кода при разных координатах:

```
Point p1 = new Point(20, 981);
Point p2 = new Point(51, 20);
```

```
System.out.println(p1.hashCode()); // 2562
System.out.println(p2.hashCode()); // 2562
```

Для автоматической генерации методов `equals()` и `hashCode()` можно воспользоваться возможностями программы Eclipse. Для этого в меню **Source** выбираем пункт **Generate hashCode() and equals()**;

- `getClass()` — возвращает экземпляр класса `Class<T>`, который содержит множество методов, позволяющих исследовать пользовательский класс на основе объекта. Например, с помощью методов `getName()` и `getCanonicalName()` можно получить название класса в виде строки:

```
System.out.println(p1.getClass().getCanonicalName()); // Point
System.out.println(p1.getClass().getName());          // Point
```

Метод `getSuperclass()` позволяет получить объект класса `Class<T>` для базового класса:

```
System.out.println(p1.getClass().getSuperclass().getName());
// java.lang.Object
```

Кроме того, мы уже использовали метод `getClass()` при переопределении метода `equals()`:

```
// Если классы не совпадают
if (this.getClass() != obj.getClass())
   return false;
```

Названия классов содержат специальное свойство `class`, которое ссылается на экземпляр класса `Class<T>`. Например, предыдущую инструкцию можно записать так:

```
if (obj.getClass() != Point.class)
   return false;
```

11.19. Массивы объектов

Объявление массива объектов ничем не отличается от объявления массива значений, имеющих элементарный тип. Массив объектов можно создать так:

```
<Название класса>[] <Название переменной> =
                  new <Название класса>[<Количество объектов>];
```

Например, у нас есть класс `A` с двумя общедоступными полями, имеющими разный тип данных:

```
class A {
   public int x;
   public String s = "";
}
```

Объявить массив объектов класса `A` из трех элементов можно так:

```
A[] arr = new A[2];
```

Все элементы такого массива получат значение по умолчанию. Для объектов значением по умолчанию будет `null`. Любая попытка обратиться к полям класса `A` через значение `null` приведет к ошибке. Поэтому после объявления массива объектов необходимо создать объекты и присвоить ссылки на них элементам массива:

```
arr[0] = new A();
arr[1] = new A();
```

Можно присвоить ссылки сразу при объявлении:

```
A[] arr = { new A(), new A() };
```

Наш класс не содержит конструктора, поэтому все поля получат значения по умолчанию. Так как строки относятся к объектам, поле `s` по умолчанию получило бы значение `null`, но раз мы выполнили явную инициализацию пустой строкой, то значение изменено не будет и, следовательно, ошибки обращения к строковым методам через значение `null` не произойдет. Чтобы присвоить значения полям, необходимо вначале указать индекс элемента внутри квадратных скобок, а затем с помощью точечной нотации обратиться к полям объекта:

```
arr[0].x = 10;
arr[0].s = "строка1";
arr[1].x = 20;
arr[1].s = "строка2";
```

Получить значения элементов массива можно с помощью оператора `for`:

```
for (A obj: arr) {
   System.out.println(obj.x);
   System.out.println(obj.s);
}
```

Как вы уже знаете, объектная переменная может хранить ссылки на объекты и производных классов. Например, наш класс `A` является наследником класса `Object`, поэтому мы можем объявить тип массива `Object` и добавлять в этот массив вообще любые объекты, поскольку все классы являются наследниками класса `Object`. Однако при обращении к членам объектов необходимо выполнить приведение типов:

```
Object[] arr = { new A(), new A() };
((A)arr[0]).x = 10;
((A)arr[0]).s = "строка1";
((A)arr[1]).x = 20;
((A)arr[1]).s = "строка2";
for (Object obj: arr) {
   System.out.println(((A)obj).x);
   System.out.println(((A)obj).s);
}
```

В этом примере мы знаем, что в массиве только объекты класса `A`, но если мы добавляем любые объекты, то перед каждой операцией приведения необходимо

выполнять проверку с помощью оператора `instanceof`. Ведь если приведение не может быть выполнено, то возникнет ошибка.

11.20. Передача объектов в метод и возврат объектов

Если метод принимает объект, то класс объекта указывается в качестве типа данных в объявлении параметра. Следует помнить, что при вызове метода и передаче объекта параметр получит копию значения, а значением является ссылка на объект. Если объект является изменяемым, то его свойства можно будет изменить внутри метода через эту копию ссылки.

Если метод объявлен внутри одноименного класса, что и класс объекта, то внутри метода можно будет обратиться ко всем членам класса, включая закрытые. Если метод объявлен в другом классе, то к закрытым методам доступа не будет (листинг 11.19).

Листинг 11.19. Передача объектов в метод

```
public class MyClass {
   public static void main(String[] args) {
      A obj = new A();
      A.func(obj);
      MyClass.func(obj);
      System.out.println(obj.s);       // Новая строка
   }

   public static void func(A obj) {    // Вне класса A
      //System.out.println(obj.x);     // Ошибка. x - private
      System.out.println(obj.getX()); // OK
      System.out.println(obj.s);
      obj.s = "Новая строка";          // Можно изменить объект
   }
}

class A {
   private int x = 10;
   public String s = "Строка";

   public static void func(A obj) { // Внутри класса A
      System.out.println(obj.x);    // OK, хотя x - private
      System.out.println(obj.s);
      obj.x = 50;                   // Можно изменить объект
   }
```

```
    public int getX() {
        return this.x;
    }
}
```

Если нужно передать в метод объекты произвольного класса, то указывается тип `Object`. Внутри такого метода необходимо проверить возможность приведения к какому-либо типу и выполнить приведение до каких-либо действий с объектом. Изменим метод `func()` из класса `A` (см. листинг 11.19) и укажем в качестве типа класс `Object`:

```
public static void func(Object obj) { // Внутри класса A
    if (obj.getClass() == A.class) {
        A objA = (A)obj;
        System.out.println(objA.x);
        System.out.println(objA.s);
        objA.x = 50;
    }
}
```

Для передачи иерархии классов и использования полиморфизма указывается тип базового класса. В этом случае будет вызван одноименный метод из производного класса. Чтобы вызвать какой-либо специфический метод производного класса, необходимо сначала проверить возможность приведения с помощью оператора `instanceof` и затем выполнить приведение.

Внутри метода мы можем создать объект и вернуть ссылку на него с помощью оператора `return`. Например, создадим метод, внутри которого вернем объект класса `Date`:

```
public static Date newDate() {
    Date d = new Date();
    return d;
}
```

Вызовем этот метод и присвоим результат переменной:

```
Date d1 = newDate();
System.out.println(d1); // Sat Mar 24 03:07:36 MSK 2018
```

Теперь попробуем просто изменить объект и вернуть ссылку:

```
public static Date newDate(Date d, long t) {
    d.setTime(t);
    return d;
}
```

В этом случае мы получим проблему:

```
Date d1 = newDate();
System.out.println(d1); // Sat Mar 24 03:08:26 MSK 2018
Date d2 = newDate(d1, 124587L);
```

```
System.out.println(d1); // Thu Jan 01 03:02:04 MSK 1970
System.out.println(d2); // Thu Jan 01 03:02:04 MSK 1970
```

Как можно видеть, изменились оба значения. Так как мы в качестве значения параметра получаем ссылку на изменяемый объект, то все изменения свойств объекта внутри метода затронут и исходный объект. Чтобы избежать этой проблемы, необходимо создать копию объекта, изменить в ней какие-либо свойства объекта и вернуть эту копию. Для создания копии объекта можно воспользоваться методом `clone()` из класса `Object`. Формат метода в классе `Date`:

```
public Object clone()
```

Метод возвращает экземпляр класса `Object`, поэтому необходимо выполнить приведение типов. Давайте переделаем наш метод `newDate()`:

```
public static Date newDate(Date d, long t) {
   Date d2 = (Date) d.clone();
   d2.setTime(t);
   return d2;
}
```

Теперь все будет нормально:

```
Date d1 = newDate();
System.out.println(d1); // Sat Mar 24 03:08:26 MSK 2018
Date d2 = newDate(d1, 124587L);
System.out.println(d1); // Sat Mar 24 03:08:26 MSK 2018
System.out.println(d2); // Thu Jan 01 03:02:04 MSK 1970
```

Это очень частая ошибка, причем ее трудно обнаружить сразу. Например, в закрытом поле мы храним изменяемый объект. В методе `getName()` мы возвращаем ссылку на этот объект. И хотя поле объявлено как `private`, этот объект можно будет изменить извне. Всегда возвращайте копию изменяемого объекта, а не копию ссылки на него, иначе принцип инкапсуляции будет нарушен:

```
class A {
   private Date d;
   public A(Date dt) {
      if (dt == null) this.d = new Date();
      else {
         // Так нельзя!!!
         // this.d = dt;
         // Только так
         this.d = (Date) dt.clone(); // this.d = new Date(dt.getTime());
      }
   }
   public Date getD() {
      // Так нельзя!!!
      // return this.d;
      // Только так
      return (Date) this.d.clone(); // return new Date(this.d.getTime());
   }
}
```

Создание копии объекта — это довольно сложная тема. Ведь изменяемый объект может хранить в свою очередь ссылки на изменяемые объекты. Метод `clone()` из класса `Object` создает лишь поверхностную копию и ничего не знает о вложенных изменяемых объектах — он просто копирует все ссылки. Способы правильного создания копии объекта пользовательского класса мы рассмотрим в следующей главе.

11.21. Классы-«обертки» над элементарными типами

Если при объявлении переменной указан тип `Object`, то переменная может хранить данные любого типа, включая данные элементарных типов:

```
Object n = 4;
System.out.println(n); // 4
```

Почему, ведь элементарные типы не являются объектами какого-либо класса? Давайте вызовем метод `getClass()` и посмотрим на результат:

```
Object n = 4;
System.out.println(n.getClass()); // class java.lang.Integer
```

Здесь мы получили класс `Integer`, а не тип `int`. Этот класс является классом-«оберткой» для элементарного типа `int`. Когда мы присваиваем значение объектной переменной, значение автоматически преобразовывается (как бы «оборачивается») в объект класса `Integer`. Давайте рассмотрим все классы-«обертки» и соответствующие им элементарные типы, а так же способы создания объектов и получения значений элементарных типов:

- `Boolean` — тип `boolean`:

```
Boolean a = Boolean.valueOf(true);
boolean b = a.booleanValue();
```

- `Character` — тип `char`:

```
Character c = Character.valueOf('w');
char ch = c.charValue();
```

- `Byte` — тип `byte`:

```
Byte x = Byte.valueOf((byte) 4);
byte y = x.byteValue();
```

- `Short` — тип `short`:

```
Short x = Short.valueOf((short) 4);
short y = x.shortValue();
```

- `Integer` — тип `int`:

```
Integer x = Integer.valueOf(4);
int y = x.intValue();
```

- Long — тип long:

```
Long x = Long.valueOf(254L);
long y = x.longValue();
```

- Float — тип float:

```
Float x = Float.valueOf(2.4f);
float y = x.floatValue();
```

- Double — тип double:

```
Double x = Double.valueOf(2.45);
double y = x.doubleValue();
```

Можно просто присвоить объектной переменной значение элементарного типа и компилятор автоматически «упакует» его в экземпляр соответствующего класса:

```
Integer x = 10;
int y = x;
int z = (int) x;
System.out.println(y); // 10
System.out.println(z); // 10
```

Такой процесс называют *автоупаковкой*. Как видно из примера, присвоение объекта целочисленной переменной автоматически преобразует объект в число. Хотя мы можем выполнить эту операцию и явным образом:

```
Integer x = Integer.valueOf(10);
int y = x.intValue();
System.out.println(y); // 10
```

> ***Примечание***
>
> Начиная с Java 9, конструкторы классов-«оберток» объявлены устаревшими. Вместо них следует использовать статический метод `valueOf()`.

ГЛАВА 12

Интерфейсы

Интерфейс похож на абстрактный класс и также содержит только сигнатуру методов без реализации. Однако интерфейс не является классом, и невозможно создать экземпляр интерфейса. Интерфейс — лишь требование к реализации класса. Класс, реализующий интерфейс, гарантирует, что внутри него определены методы, которые описаны внутри блока интерфейса. Если класс не переопределяет хотя бы один метод, то он становится абстрактным классом, и создать экземпляр этого класса будет нельзя.

Зачем нужны интерфейсы? Во-первых, для создания обработчиков различных событий. В языке Java до 8-й версии не было понятия функции обратного вызова. И невозможно было передать в метод ссылку на другой метод. Вместо этого в метод передается ссылка на объект, реализующий какой-либо интерфейс. Внутри блока интерфейса указывается сигнатура метода, который будет вызван при наступлении события. Нам достаточно в своем классе реализовать интерфейс и переопределить метод, указанный в интерфейсе. Далее создаем экземпляр класса и передаем его в метод регистрации обработчика события. При наступлении события — например, пользователь нажал кнопку, — этот метод будет вызван.

Во-вторых, интерфейсы полезны при командной работе над проектом. Например, команда программистов выполняет три задачи. Первый программист реализует ввод данных, второй — их обработку, а третий — вывод обработанных данных. Если эти программисты не договорятся заранее, то они не смогут состыковать свои классы. Договариваясь, второй программист предлагает первому: «Реализуй интерфейс с методом `read()`», а третьему: «Реализуй интерфейс с методом `write()`», названия интерфейсов такие-то, сигнатура методов такая-то. Все! Теперь каждый программист занимается своей работой. Первый программист создает множество классов (например, получение данных из файла, получение данных из Интернета и т. д.) и в каждом классе реализует интерфейс получения данных (метод `read()`). Третий программист создает множество классов (например, вывод на консоль, вывод в файл, передача по сети Интернет, печать на принтере и т. д.) для вывода результата и в каждом классе реализует интерфейс вывода (метод `write()`). Второй программист может подключить эти классы к своим классам, указав не названия классов, а названия интерфейсов. Поэтому не важно, как называются классы, —

главное, чтобы они соответствовали требованиям интерфейса. Давайте рассмотрим это на примере.

12.1. Создание интерфейса

Для создания файла с интерфейсом в программе Eclipse из меню **File** выбираем пункт **New | Interface**. В открывшемся окне (рис. 12.1) в поле **Name** вводим название `IRead` и нажимаем кнопку **Finish**. В результате будет создан файл **IRead.java**, содержащий следующий код:

```
public interface IRead {

}
```

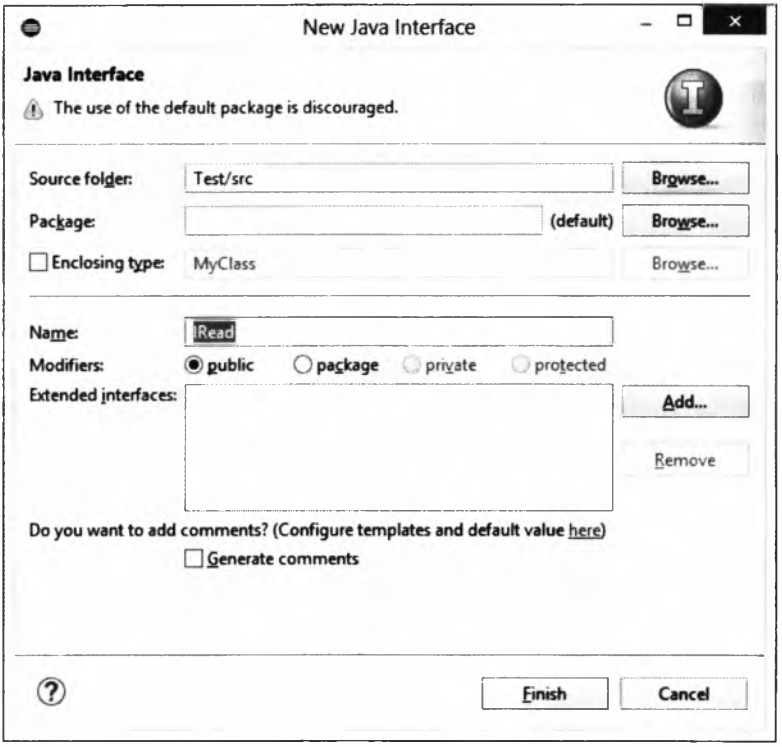

Рис. 12.1. Создание интерфейса

Почти, как создание класса. Отдельный файл с названием, совпадающим с названием интерфейса. Ключевое слово `public`, означающее, что интерфейс является общедоступным, и название интерфейса в том же месте, что и в классах. Только вместо ключевого слова `class` используется ключевое слово `interface`. Но интерфейс не является классом, он всего лишь содержит требования, предъявляемые к классу, его реализующему. Требования выражаются в сигнатурах методов, которые должен определить класс. В нашем случае давайте запишем требование реализовать метод `read()`:

```
public interface IRead {
   String read();
}
```

Как видно из примера, мы указали только тип возвращаемого значения и название метода. После пустых круглых скобок, означающих, что метод не принимает никаких параметров, ставится точка с запятой. Обратите внимание: метод не содержит блока с реализацией. Класс должен переопределить этот метод и выполнить его реализацию. Какой будет эта реализация, каждый класс решает сам, — главное, чтобы метод вернул строку с данными.

Теперь создадим еще один файл с интерфейсом `IWrite` и методом `write()`:

```
public interface IWrite {
   void write(String s);
}
```

Как видно из сигнатуры, метод `write()` принимает строку и ничего не возвращает. Итак, у нас есть требования к классам ввода и вывода. Теперь давайте создадим классы, реализующие эти интерфейсы.

Чтобы указать, что класс реализует какой-либо интерфейс, после названия класса (и выражения наследования, если оно есть) указываются ключевое слово `implements` и название интерфейса:

```
class A implements IRead {
}
```

Этим мы говорим, что класс `A` реализует интерфейс `IRead`. Если класс наследует какой-либо другой класс, то ключевое слово `implements` указывается после названия базового класса:

```
class B extends A implements IRead {

}
```

Здесь класс `B` наследует класс `A` и реализует интерфейс `IRead`.

Чтобы в редакторе Eclipse автоматически создать метод, требующий реализации в интерфейсе, наводим указатель мыши на название класса (оно у нас сейчас подчеркивается красной волнистой линией) и во всплывающем окне выбираем пункт **Add unimplemented methods** — заготовка метода будет вставлена внутрь блока класса. Давайте из метода просто вернем какую-либо строку:

```
class A implements IRead {
   @Override
   public String read() {
      return "строка 1";
   }
}
```

Аннотация `@Override` говорит компилятору, что мы переопределяем метод. В этом случае компилятор проверит сигнатуру метода и выведет сообщение об ошибке,

если мы перегружаем метод, а не переопределяем его. В принципе, эту аннотацию можно и не указывать, но тогда компилятор ни о чем нас не предупредит.

Перед типом возвращаемого значения указывается ключевое слово `public`. Все методы, описанные в интерфейсе, в большинстве случаев являются общедоступными. Внутри интерфейса мы можем опустить ключевое слово `public`, а вот при переопределении метода обязаны указать его явным образом.

Давайте создадим еще один класс, в котором вернем другую строку, а также определим какой-либо метод и закрытое поле:

```
class B implements IRead {
   private String s = "строка 2";
   @Override
   public String read() {
      return this.s;
   }
   public void print() {
      System.out.println(this.s);
   }
}
```

Реализация классов для ввода данных у нас есть. Мы упростили классы, но в реальной жизни класс `A` мог бы получать данные из файла, а класс `B` — из Интернета. Причем эти классы могли бы иметь множество других методов и полей, как и обычные классы.

Теперь создадим класс `C`, реализующий интерфейс `IWrite`:

```
class C implements IWrite {
   @Override
   public void write(String s) {
      System.out.println(s);
   }
}
```

В этом случае мы также поступаем упрощенно — просто выводим строку в окно консоли. Нам сейчас главное понять суть интерфейсов, а не разбираться с реализацией ввода и вывода.

Далее создадим класс `D`, внутри которого будет производиться обработка данных (например, просто изменим регистр символов в строке), полученных из разных источников с помощью интерфейса `IRead`, и вывод данных согласно интерфейсу `IWrite`:

```
class D {
   private IRead ir;
   private IWrite iw;
   private String str = "";

   public D(IRead r, IWrite w) {
      this.ir = r;
```

```
      this.iw = w;
      this.str = ir.read();              // Получаем данные
   }
   public void change() {
      this.str = this.str.toUpperCase(); // Обрабатываем данные
   }
   public void print() {
      this.iw.write(this.str);           // Выводим данные
   }
}
```

Обратите внимание на типы данных полей. Там нет классов A, B и C. Вместо этих классов указываются названия интерфейсов. Нам не важно, как называются классы ввода и вывода, нам важно, чтобы эти классы реализовывали интерфейсы ввода или вывода. Теперь мы готовы запустить процесс ввода, обработки и вывода:

```
public class MyClass {
   public static void main(String[] args) {
      IRead r1 = new A();
      B r2     = new B();
      IWrite w = new C();
      D obj1   = new D(r1, w);
      obj1.change();
      obj1.print();           // СТРОКА 1
      D obj2   = new D(r2, w);
      obj2.change();
      obj2.print();           // СТРОКА 2
   }
}
```

При объявлении переменной мы можем, как обычно, указать в качестве типа название класса:

```
B r2     = new B();
```

или название интерфейса:

```
IRead r1 = new A();
IWrite w = new C();
```

Разница лишь в наборе методов, доступных через переменную. Например, через переменную r2 доступен как метод из интерфейса, так и остальные методы, и мы можем вызвать метод print() из класса B:

```
r2.print(); // строка 2
```

Если типом переменной будет название интерфейса, то окажутся доступны только методы, объявленные внутри блока интерфейса (а также методы, наследованные от класса Object). Как вы уже знаете, переменные в языке Java являются полиморфными, т. е. переменная с типом базового класса может содержать ссылки на объекты производных классов. С интерфейсами все точно так же. Переменная с типом

интерфейса может содержать ссылки на классы, внутри которых этот интерфейс реализован. Поэтому мы можем значение переменной `r2` спокойно передать конструктору класса `D`. Внутри этого класса поля объявлены с типом интерфейсов, поэтому через них будут доступны только методы, объявленные в интерфейсе. А нам ведь больше ничего и не надо — нужно получить данные и вывести обработанные данные. Никакой другой информации о реализации классов ввода и вывода нам знать не требуется. И без этой информации мы прекрасно справились с задачей.

Чтобы сложилась целостная картина, приведем полное содержание файла MyClass.java (листинг 12.1). Интерфейсы у нас расположены в отдельных файлах в одном каталоге с файлом MyClass.java.

Листинг 12.1. Интерфейсы: содержимое файла MyClass.java

```
public class MyClass {
   public static void main(String[] args) {
      IRead r1 = new A();
      B r2     = new B();
      IWrite w = new C();
      D obj1   = new D(r1, w);
      obj1.change();
      obj1.print();           // СТРОКА 1
      D obj2   = new D(r2, w);
      obj2.change();
      obj2.print();           // СТРОКА 2
   }
}

class A implements IRead {
   @Override
   public String read() {
      return "строка 1";
   }
}

class B implements IRead {
   private String s = "строка 2";
   @Override
   public String read() {
      return this.s;
   }
   public void print() {
      System.out.println(this.s);
   }
}

class C implements IWrite {
   @Override
```

```
   public void write(String s) {
      System.out.println(s);
   }
}

class D {
   private IRead ir;
   private IWrite iw;
   private String str = "";

   public D(IRead r, IWrite w) {
      this.ir = r;
      this.iw = w;
      this.str = ir.read();              // Получаем данные
   }
   public void change() {
      this.str = this.str.toUpperCase(); // Обрабатываем данные
   }
   public void print() {
      this.iw.write(this.str);           // Выводим данные
   }
}
```

12.2. Реализация нескольких интерфейсов

Один класс может реализовывать сразу несколько интерфейсов. В этом случае названия интерфейсов указываются после ключевого слова `implements` через запятую. Создадим класс `E`, который реализует и интерфейс ввода, и интерфейс вывода:

```
class E implements IRead, IWrite {
   @Override
   public String read() {
      return "строка 3";
   }
   @Override
   public void write(String s) {
      System.out.println(s);
   }
}
```

Теперь используем этот класс в нашем процессе ввода, обработки и вывода:

```
public class MyClass {
   public static void main(String[] args) {
      E rw = new E();
      D obj1   = new D(rw, rw);
      obj1.change();
```

```
        obj1.print();           // СТРОКА 3
    }
}
```

В этом случае мы просто передали один объект класса E и для операции ввода, и для операции вывода.

12.3. Расширение интерфейсов

Интерфейсы могут расширяться за счет наследования базовых интерфейсов. Причем интерфейсы поддерживают множественное наследование, в отличие от классов. Чтобы наследовать интерфейс, необходимо указать ключевое слово `extends` и название базового интерфейса. Если используется множественное наследование, то базовые интерфейсы перечисляются через запятую. Создадим интерфейс `IReadWrite`, который наследует интерфейсы `IRead` и `IWrite`:

```
interface IReadWrite extends IRead, IWrite {
}
```

Теперь создадим класс G, который реализует этот интерфейс:

```
class G implements IReadWrite {
    @Override
    public String read() {
        return "строка 4";
    }
    @Override
    public void write(String s) {
        System.out.println(s);
    }
}
```

Так как новый интерфейс содержит методы и для ввода, и для вывода, мы можем его использовать в нашем процессе ввода, обработки и вывода как обычно:

```
public class MyClass {
    public static void main(String[] args) {
        G rw = new G();
        D obj1   = new D(rw, rw);
        obj1.change();
        obj1.print();           // СТРОКА 4
    }
}
```

12.4. Создание статических констант внутри интерфейса

Помимо сигнатур методов, в блоке интерфейса можно создавать общедоступные статические константы. Так как создать допускается только статические константы, ключевые слова `static` и `final` обычно не указываются:

```
interface IConst {
   int MY_CONST1 = 10;
   int MY_CONST2 = 20;
   int MY_CONST3 = 30;
   void print();
}
```

В этом примере мы создали три целочисленные статические константы и объявление метода. Класс, реализующий этот интерфейс, должен переопределить только метод `print()`. Константы просто наследуются классом и доступны через название класса или через название интерфейса:

```
class H implements IConst {
   @Override
   public void print() {
      System.out.println(H.MY_CONST1); // 10
      System.out.println(H.MY_CONST2); // 20
      System.out.println(H.MY_CONST3); // 30
   }
}
```

Создадим экземпляр этого класса и выведем значения констант:

```
public class MyClass {
   public static void main(String[] args) {
      H obj = new H();
      obj.print();
      System.out.println(H.MY_CONST1);      // 10
      System.out.println(IConst.MY_CONST1); // 10
   }
}
```

12.5. Создание статических методов внутри интерфейса

Внутри блока интерфейса можно определить *статические методы* с реализацией. Такие методы доступны внутри блока интерфейса, а также через название интерфейса. Переопределить такие методы внутри класса нельзя, и через экземпляр класса или название класса они недоступны. Пример интерфейса с двумя статическими методами:

```
interface IStatic {
   static void test() {
      print();
   }
   static void print() {
      System.out.println("Привет");
   }
}
```

Теперь создадим класс, реализующий этот интерфейс:

```
class K implements IStatic {
   public K() {
      IStatic.test();
   }
}
```

Вызовем конструктор класса, создавая объект, а также получим доступ к статическим методам вне класса K:

```
public class MyClass {
   public static void main(String[] args) {
      K k = new K();     // Привет
      IStatic.print();  // Привет
   }
}
```

Начиная с Java 9, статические методы могут быть объявлены закрытыми с помощью модификатора доступа `private`. При этом закрытый метод будет доступен только внутри методов интерфейса. Пример интерфейса с двумя статическими методами, один из которых объявлен закрытым:

```
public interface IStaticPrivate {
   static void test() {
      print();
   }
   private static void print() {
      System.out.println("Привет");
   }
}
```

Получить доступ к закрытому методу `print()` можно только из метода `test()`:

```
public class MyClass {
   public static void main(String[] args) {
      IStaticPrivate.test();     // Привет
      // IStaticPrivate.print(); // Ошибка
   }
}
```

12.6. Методы по умолчанию и закрытые методы

Начиная с Java 8, внутри блока интерфейса можно определить *методы по умолчанию* с реализацией. Причиной такого нововведения стала необходимость внесения изменений в старые интерфейсы. Если просто добавить объявление какого-либо нового метода, то все классы, реализующие эти интерфейсы, автоматически станут абстрактными, и потребуется их модификация. Введение метода по умолчанию позволяет добавить новый метод и сразу реализовать блок с кодом. Старые классы

будут работать и так, а новые могут дополнительно переопределить метод по умолчанию.

Метод по умолчанию объявляется с помощью ключевого слова `default`:

```
interface IDefault {
   default void print() {
      System.out.println("Привет");
   }
}
```

Теперь создадим два класса. Класс `L` просто реализует интерфейс, а класс `M` — переопределяет метод по умолчанию:

```
class L implements IDefault {
}
class M implements IDefault {
   @Override
   public void print() {
      System.out.println("Прощай");
   }
}
```

Создадим экземпляры этих классов и вызовем метод `print()`:

```
public class MyClass {
   public static void main(String[] args) {
      L obj = new L();
      obj.print();          // Привет
      M obj2 = new M();
      obj2.print();         // Прощай
   }
}
```

Язык Java допускает указание нескольких интерфейсов через запятую при реализации. Если внутри двух интерфейсов окажутся два одноименных метода по умолчанию с одной и той же сигнатурой, то компилятор не сможет выбрать один метод и выведет сообщение об ошибке. В таком случае класс, реализующий эти интерфейсы, должен переопределить этот метод в обязательном порядке. Давайте создадим еще один интерфейс:

```
interface IDefault2 {
   default void print() {
      System.out.println("Привет, Вася");
   }
}
```

Теперь создадим класс и внутри переопределенного метода обратимся к одноименным методам по умолчанию в интерфейсах:

```
class N implements IDefault, IDefault2 {
   @Override
```

```
    public void print() {
        IDefault.super.print();
        IDefault2.super.print();
    }
}
```

Как видно из примера, вначале указывается название интерфейса, затем ключевое слово super, а далее название метода. Создадим экземпляр класса и выведем результат:

```
public class MyClass {
    public static void main(String[] args) {
        N obj = new N();
        obj.print();          // Привет
                              // Привет, Вася
    }
}
```

Начиная с Java 9, внутри блока интерфейса можно объявить закрытый обычный метод с реализацией. При этом метод будет доступен только внутри методов по умолчанию, а также внутри других закрытых методов:

```
public interface IPrivate {
    default void test() {
        print();
    }
    private void print() {
        System.out.println("Привет");
    }
}
```

Создадим класс, реализующий этот интерфейс:

```
class P implements IPrivate {
}
```

Создадим экземпляр класса и выведем результат:

```
public class MyClass {
    public static void main(String[] args) {
        P obj = new P();
        obj.test();          // Привет
    }
}
```

12.7. Интерфейсы и обратный вызов

Интерфейсы используются также для создания обработчиков различных событий. Как уже отмечалось ранее, в языке Java до 8-й версии отсутствовало понятие функции обратного вызова, и невозможно было передать в метод ссылку на другой метод. Вместо этого в метод передается ссылка на объект, реализующий какой-

либо интерфейс, а внутри блока интерфейса указывается сигнатура метода, который будет вызван при наступлении события. Нам достаточно в своем классе реализовать интерфейс и переопределить метод, указанный в интерфейсе. Далее создаем экземпляр класса и передаем его в метод регистрации обработчика события. При наступлении события — например, пользователь нажал кнопку, — этот метод будет вызван (листинг 12.2).

Листинг 12.2. Интерфейсы и обратный вызов

```
public class MyClass {
   public static void main(String[] args) {
      MyButton button = new MyButton();
      IClick ic = new A();
      button.reg(ic);           // Регистрация обработчика
      for (int i = 0; i < 5; i++) {
         button.click();        // Генерируем нажатие
         try {
            Thread.sleep(1000); // Имитация ожидания
         }
         catch (InterruptedException e) {
            e.printStackTrace();
            System.exit(1);
         }
      }
   }
}

interface IClick {
   void onClick();
}

class A implements IClick {
   @Override
   public void onClick() {
      System.out.println("Нажата кнопка");
   }
}
class MyButton {
   private IClick ic = null;
   public void reg(IClick ic) { // Регистрация обработчика
      this.ic = ic;
   }
   public void click() {        // Нажатие кнопки
      if (this.ic != null)
         this.ic.onClick();
   }
}
```

В этом примере мы создали интерфейс `IClick` и говорим, что классы, реализующие интерфейс, должны содержать метод `onClick()`, который будет вызван при нажатии кнопки пользователем. Мы пока не рассматривали создание оконных приложений, поэтому упростим и инсценируем эту ситуацию. Создаем класс `A`, реализующий интерфейс, и переопределяем метод `onClick()`. Внутри этого метода просто выводим сообщение в окно консоли. Внутри класса `MyButton` (описывающего кнопку) есть поле, в котором хранится ссылка на обработчик события нажатия кнопки. Тип этого поля соответствует названию интерфейса. С помощью метода `reg()` производится регистрация обработчика, а метод `click()` — имитирует нажатие кнопки. Внутри метода `main()` создаем экземпляр кнопки и экземпляр обработчика. Далее регистрируем обработчик и внутри цикла имитируем нажатие кнопки. Каждый раз в окно консоли будет выводиться сообщение о факте нажатия кнопки.

Назначить обработчик события можно также с помощью анонимных вложенных классов. В этом случае класс `A` нам не нужен — достаточно создать анонимный вложенный класс и внутри него переопределить метод `onClick()`. В качестве базового класса следует указывать название интерфейса. Давайте изменим содержимое класса `MyClass` (листинг 12.3).

Листинг 12.3. Использование анонимных вложенных классов в качестве обработчика

```
public class MyClass {
   public static void main(String[] args) {
      MyButton button = new MyButton();
      button.reg(new IClick() {  // Анонимный вложенный класс
         @Override
         public void onClick() {
            System.out.println("Нажата кнопка");
         }
      });
      for (int i = 0; i < 5; i++) {
         button.click();        // Генерируем нажатие
         try {
            Thread.sleep(1000); // Имитация ожидания
         }
         catch (InterruptedException e) {
            e.printStackTrace();
            System.exit(1);
         }
      }
   }
}
```

12.8. Функциональные интерфейсы и лямбда-выражения

Вы заметили, как упростился код при использовании анонимных вложенных классов? Начиная с Java 8, вместо обычных классов, реализующих интерфейс, и анонимных вложенных классов можно использовать *лямбда-выражения*, которые позволяют еще больше уменьшить код. Например, вот эту инструкцию из предыдущего листинга:

```
button.reg(new IClick() {  // Анонимный вложенный класс
   @Override
   public void onClick() {
      System.out.println("Нажата кнопка");
   }
});
```

можно записать так:

```
button.reg(
   () -> System.out.println("Нажата кнопка")
);
```

Код стал очень простым. Хотя мы и не реализовывали интерфейс и не переопределяли метод, лямбда-выражение сделало всю работу за нас. Во-первых, оно не имеет названия. Компилятор сам подставит название метода `onClick()` из интерфейса после компиляции. Круглые скобки означают передачу параметра в метод. Метод `onClick()` ничего не принимает, поэтому указываются только круглые скобки. После символов -> указывается тело метода. В нашем случае оно состоит из одной инструкции, поэтому точка с запятой в конце не ставится и фигурные скобки не указываются. Это выражение, а не законченная инструкция.

Следует заметить, что лямбда-выражение можно использовать только совместно с *функциональными интерфейсами*. Что же это такое? Функциональный интерфейс — это интерфейс, описывающий только один метод, который необходимо переопределить. Внутри такого интерфейса могут быть методы по умолчанию и закрытые методы с реализацией, статические методы и статические константы. Чтобы явно определить функциональный интерфейс, перед объявлением следует указать аннотацию `@FunctionalInterface`. Она не является обязательной, но позволит избежать различных ошибок — например, добавления объявлений дополнительных абстрактных методов:

```
@FunctionalInterface
interface IClick {
   void onClick();
}
```

Теперь наш интерфейс `IClick` на все сто процентов является функциональным интерфейсом, и мы можем смело использовать лямбда-выражения везде, где ожидается этот интерфейс.

В Java 8 существует целый пакет `java.util.function`, содержащий множество готовых функциональных интерфейсов. Импортировать все интерфейсы из него позволяет следующая инструкция:

```
import java.util.function.*;
```

Рассмотрим основные функциональные интерфейсы из этого пакета:

- `Function<T, R>` — реализует следующий вызов:

```
R apply(T t)
```

Большие буквы `T` и `R` — это произвольные типы данных. Вместо этих букв необходимо подставить названия классов. Если необходимо использовать элементарные типы, то следует указывать названия классов-«оберток» над элементарными типами. Давайте передадим в метод лямбда-выражение, соответствующее этому интерфейсу:

```
import java.util.function.*;

public class MyClass {
   public static void main(String[] args) {
      MyClass.test( n -> n * 2, 10 );
   }
   public static void test(
         Function<Integer, Integer> f, int x) {
      System.out.println(f.apply(x));
   }
}
```

Мы описали интерфейс как:

```
Function<Integer, Integer>
```

Следовательно, сигнатура метода `apply()` будет такой:

```
Integer apply(Integer t)
```

Этой сигнатуре соответствует лямбда-выражение:

```
n -> n * 2
```

Метод принимает параметр `n` (название локальной переменной, доступной внутри лямбда-выражения) и возвращает `n * 2`. Выражение перед символами `->` является описанием параметров метода, а после — телом метода. Чтобы блок с параметрами выглядел более наглядно, можно указать круглые скобки:

```
(n) -> n * 2
```

Тело метода состоит из одного выражения, поэтому фигурные скобки не указываются. Кроме того, в этом случае нет оператора `return`. Возвращается результат выполнения выражения `n * 2`. Мы можем явно указать фигурные скобки, но тогда необходимо явно указать и оператор `return`, а также точку с запятой в конце инструкции:

```
(n) -> { return n * 2; }
```

При желании мы можем указать несколько инструкций, разделяя их точками с запятой:

```
(n) -> {
   n = n * 2;
   return n;
}
```

- `BiFunction<T, U, R>` — реализует следующий вызов:

```
R apply(T t, U u)
```

Просуммируем два числа и вернем результат:

```
import java.util.function.*;

public class MyClass {
   public static void main(String[] args) {
      MyClass.test( (a, b) -> a + b, 10, 20 );  // 30
   }
   public static void test(
      BiFunction<Integer, Integer, Integer> f, int x, int y) {
      System.out.println(f.apply(x, y));
   }
}
```

В этом случае метод принимает два параметра, поэтому круглые скобки нужно указать обязательно, как и в случае отсутствия параметров:

```
(a, b) -> a + b
```

- `Predicate<T>` — реализует следующий вызов:

```
boolean test(T t)
```

Напишем проверку неравенства целого числа нулю:

```
import java.util.function.*;

public class MyClass {
   public static void main(String[] args) {
      MyClass.test( n -> n != 0, 10 );  // true
      MyClass.test( n -> n != 0, 0 );   // false
   }
   public static void test(
         Predicate<Integer> f, int x) {
      System.out.println(f.test(x));
   }
}
```

- `BiPredicate<T, U>` — реализует следующий вызов:

```
boolean test(T t, U u)
```

Проверим числа на равенство:

```
import java.util.function.*;

public class MyClass {
   public static void main(String[] args) {
      MyClass.test( (a, b) -> a == b, 10, 10 );  // true
      MyClass.test( (a, b) -> a == b, 10, 20 );  // false
   }
   public static void test(
         BiPredicate<Integer, Integer> f, int x, int y) {
      System.out.println(f.test(x, y));
   }
}
```

- `Consumer<T>` — **реализует следующий вызов:**

```
void accept(T t)
```

Выведем значение любого объекта в окно консоли:

```
import java.util.function.*;

public class MyClass {
   public static void main(String[] args) {
      MyClass.test(
         (obj) -> System.out.println(obj), 10); // 10
      MyClass.test(
         (obj) -> System.out.println(obj), 50); // 50
   }
   public static void test(
          Consumer<Object> f, Object obj) {
      f.accept(obj);
   }
}
```

- `BiConsumer<T, U>` — **реализует следующий вызов:**

```
void accept(T t, U u)
```

Просуммируем два числа и выведем результат в окно консоли:

```
import java.util.function.*;

public class MyClass {
   public static void main(String[] args) {
      MyClass.test(
         (a, b) -> System.out.println(a + b), 10, 15); // 25
   }
   public static void test(
         BiConsumer<Integer, Integer> f, int x, int y) {
      f.accept(x, y);
   }
}
```

- `Supplier<T>` — реализует следующий вызов:

```
T get()
```

Вернем случайное целое число от 0 до 100 включительно:

```
import java.util.Random;
import java.util.function.*;

public class MyClass {
   public static void main(String[] args) {
      MyClass.test(
         () -> {
            Random r = new Random();
            return r.nextInt(101);
         });
   }
   public static void test(
         Supplier<Integer> f) {
      System.out.println(f.get());
   }
}
```

> ***Примечание***
>
> **Функциональные интерфейсы из пакета** `java.util.function` **содержат также методы по умолчанию. Описание этих методов можно найти в документации.**

Давайте еще раз выведем уже рассмотренные функциональные интерфейсы и соответствующие им сигнатуры методов, а также остальные интерфейсы из пакета `java.util.function` (листинг 12.4).

Листинг 12.4. Функциональные интерфейсы

```
Function<T, R>                R apply(T t)
   UnaryOperator<T>           T apply(T t)
      IntUnaryOperator        int applyAsInt(int x)
      LongUnaryOperator       long applyAsLong(long x)
      DoubleUnaryOperator     double applyAsDouble(double x)
   IntFunction<R>             R apply(int x)
   LongFunction<R>            R apply(long x)
   DoubleFunction<R>          R apply(double x)
   ToIntFunction<T>           int applyAsInt(T x)
      LongToIntFunction       int applyAsInt(long x)
      DoubleToIntFunction     int applyAsInt(double x)
   ToLongFunction<T>          long applyAsLong(T x)
      IntToLongFunction       long applyAsLong(int x)
      DoubleToLongFunction    long applyAsLong(double x)
   ToDoubleFunction<T>        double applyAsDouble(T x)
      IntToDoubleFunction     double applyAsDouble(int x)
      LongToDoubleFunction    double applyAsDouble(long x)
```

```
BiFunction<T, U, R>              R apply(T t, U u)
   ToIntBiFunction<T,U>          int applyAsInt(T t, U u)
   ToLongBiFunction<T,U>         long applyAsLong(T t, U u)
   ToDoubleBiFunction<T,U>       double applyAsDouble(T t, U u)
   BinaryOperator<T>             T apply(T t, T u)
      IntBinaryOperator          int applyAsInt(int x, int y)
      LongBinaryOperator         long applyAsLong(long x, long y)
      DoubleBinaryOperator       double applyAsDouble(double x, double y)

Predicate<T>                     boolean test(T t)
   IntPredicate                  boolean test(int x)
   LongPredicate                 boolean test(long x)
   DoublePredicate               boolean test(double x)

BiPredicate<T, U>                boolean test(T t, U u)

Consumer<T>                      void accept(T t)
   IntConsumer                   void accept(int x)
   LongConsumer                  void accept(long x)
   DoubleConsumer                void accept(double x)

BiConsumer<T, U>                 void accept(T t, U u)
   ObjIntConsumer<T>             void accept(T t, int y)
   ObjLongConsumer<T>            void accept(T t, long y)
   ObjDoubleConsumer<T>          void accept(T t, double y)

Supplier<T>                      T get()
   BooleanSupplier               boolean getAsBoolean()
   IntSupplier                   int getAsInt()
   LongSupplier                  long getAsLong()
   DoubleSupplier                double getAsDouble()
```

12.9. Область видимости лямбда-выражений

Внутри лямбда-выражения мы имеем доступ:

- к константам, объявленным внутри метода, содержащего лямбда-выражение. При условии, что константа объявлена до лямбда-выражения. Внутри лямбда-выражения изменить значение константы нельзя;
- к локальным переменным, объявленным внутри метода, содержащего лямбда-выражение, которые не меняют своего значения после инициализации. При условии, что локальная переменная объявлена до лямбда-выражения. Внутри лямбда-выражения изменить значение такой переменной нельзя;
- к статическим методам;
- к статическим переменным класса, а также к константам класса. Внутри лямбда-выражения можно изменить значение статической переменной класса;

- к полям и методам экземпляра, если лямбда-выражение создается внутри обычного метода. Внутри лямбда-выражения можно изменить значение поля.

Обратите внимание: значение поля (и значение статической переменной класса) будет соответствовать значению на момент вызова, а не на момент создания лямбда-выражения. Если вы хотите сохранить значение поля на момент создания, то следует присвоить значение поля локальной переменной.

Рассмотрим область видимости лямбда-выражений на примере (листинг 12.5).

Листинг 12.5. Область видимости лямбда-выражений

```
import java.util.function.*;

public class MyClass {
   public static void main(String[] args) {
      A obj = new A();
      obj.print();
   }
}

class A {
   public static int sx = 40;
   public static final int MY_CONST = 50;
   private int cy = 125;

   public void print() {
      int x = 10;
      final int y = 20;
      // Сохранение текущего значения
      int cy_tmp = this.cy;
      int sx_tmp = A.sx;
      // x = 1; // Так нельзя!
      // Создание лямбда-выражения
      Function<Integer, Integer> f = n -> {
         // cy_tmp  - значение this.cy на момент создания лямбды
         // this.cy - значение на момент вызова лямбды
         System.out.println(cy_tmp);  // 125
         System.out.println(this.cy); // 10
         System.out.println(sx_tmp);  // 40
         System.out.println(A.sx);    // 50
         // n = n + z;  // Нельзя! z не определена
         // x = 1;      // Нельзя! x - константа
         A.sx += 10;    // Изменение статической переменной
         this.cy += 5;  // Изменение поля
         A.test();      // Доступ к статическому методу
         this.echo();   // Доступ к обычному методу
         return n + x + y + A.sx + MY_CONST + this.cy;
      };
```

```
        int z = 2;          // К этой переменной нет доступа
        this.cy = 10;       // Изменение значения после создания лямбды
        A.sx = 50;
        System.out.println(f.apply(10)); // Вызов лямбда-выражения
    }
    public void echo() {
        System.out.println("echo()");
    }
    public static void test() {
        System.out.println("test()");
    }
}
```

12.10. Ссылки на методы

Как вы уже знаете, в метод регистрации обработчика события мы можем передать объект класса, реализующего интерфейс обработчика, анонимный вложенный класс, а также лямбда-выражение. Начиная с Java 8, доступен еще один способ, который в других языках называется *функцией обратного вызова*. Смысл этого способа сводится к передаче в метод регистрации обработчика события ссылки на какой-либо метод. Благодаря функциональным интерфейсам, появилась возможность не реализовывать интерфейс внутри класса, а просто указывать ссылку на метод, сигнатура которого соответствует сигнатуре метода внутри функционального интерфейса. Причем название метода внутри класса может быть произвольным — важна только сигнатура. Вызов метода при наступлении события производится через название метода, объявленного внутри функционального интерфейса. Давайте переделаем код из листинга 12.2 и передадим ссылку на метод (листинг 12.6).

Листинг 12.6. Ссылки на методы и обратный вызов

```
public class MyClass {
    public static void main(String[] args) {
        MyButton button = new MyButton();
        A a = new A();
        button.reg(a::printOnClick); // Передаем ссылку на метод
        for (int i = 0; i < 5; i++) {
            button.click();          // Генерируем нажатие
            try {
                Thread.sleep(1000);  // Имитация ожидания
            }
            catch (InterruptedException e) {
                e.printStackTrace();
                System.exit(1);
            }
        }
    }
}
```

```
interface IClick {
   void onClick();
}

class A {
   private int x = 10;
   public void printOnClick() {
      System.out.println("Нажата кнопка. x = " + this.x);
      this.x++;
   }
}
class MyButton {
   private IClick ic = null;
   public void reg(IClick ic) { // Регистрация обработчика
      this.ic = ic;
   }
   public void click() {        // Нажатие кнопки
      if (this.ic != null)
         this.ic.onClick();
   }
}
```

Итак, мы имеем функциональный интерфейс `IClick`, который содержит описание сигнатуры метода `onClick()`. Метод ничего не принимает и ничего не возвращает. Внутри класса `A` определен метод `printOnClick()` с сигнатурой, в точности соответствующей сигнатуре метода внутри интерфейса. В исходном примере мы реализовывали интерфейс `IClick` и переопределяли метод `onClick()`. В этом примере мы ничего не реализовываем, и даже метод называется абсолютно по-другому. Теперь взглянем на код регистрации обработчика:

```
MyButton button = new MyButton();
A a = new A();
button.reg(a::printOnClick); // Передаем ссылку на метод
```

Здесь мы вначале создаем объект кнопки и объект класса `A`. Далее передаем в метод `reg()` ссылку на метод `printOnClick()`. Обычно обращение к методу класса производится с помощью точечной нотации, и после названия метода указываются круглые скобки с параметрами внутри или без. При передаче ссылки на метод, перед названием метода вместо точки указываются два двоеточия, а после названия — никаких круглых скобок. Вызов метода обработчика производится следующим образом:

```
this.ic.onClick();
```

Как видно из примера, при вызове указывается название метода из функционального интерфейса, а не название метода, ссылку на который мы передали при регистрации обработчика. Таким образом, мы можем сохранить ссылку в переменной или

передать ее в какой-либо метод, а потом вызвать этот метод через название метода внутри функционального интерфейса:

```
A a = new A();
IClick ic = a::printOnClick;      // Сохраняем ссылку на метод
ic.onClick();                     // Вызываем этот метод
```

В предыдущем разделе мы рассмотрели встроенные функциональные интерфейсы. Давайте потренируемся и сохраним ссылки на различные методы. Начнем с интерфейса `Consumer<T>`:

```
Consumer<T>                    void accept(T t)
```

Такой сигнатуре, например, соответствует метод `println()`. Сохраним ссылку на этот метод, а затем вызовем и передадим какую-либо строку:

```
Consumer<Object> f = System.out::println;    // Ссылка на метод
f.accept("Строка 1");                        // Строка 1
f = (obj) -> System.out.println(obj);        // Лямбда-выражение
f.accept("Строка 2");                        // Строка 2
```

Теперь рассмотрим интерфейс `DoubleUnaryOperator`:

```
DoubleUnaryOperator      double applyAsDouble(double x)
```

Такой сигнатуре, например, соответствует статический метод `sqrt()` из класса `Math`:

```
DoubleUnaryOperator f = Math::sqrt;
double y =  f.applyAsDouble(100.0);
System.out.println(y);                  // 10.0
f = (x) -> Math.sqrt(x);
y = f.applyAsDouble(100.0);
System.out.println(y);                  // 10.0
```

Рассмотрим и интерфейс `BiFunction<T, U, R>`:

```
BiFunction<T, U, R>            R apply(T t, U u)
```

Такой сигнатуре, например, соответствует статический метод `pow()` из класса `Math`:

```
BiFunction<Double, Double, Double> f = Math::pow;
double y =  f.apply(3.0, 3.0);
System.out.println(y);                  // 27.0
f = (a, b) -> Math.pow(a, b);
y = f.apply(3.0, 3.0);
System.out.println(y);                  // 27.0
```

Можно также указать ссылку на конструктор класса. В этом случае вместо названия метода указывается ключевое слово `new`:

```
Supplier<A> f = A::new;
```

Давайте рассмотрим сохранение ссылки на конструктор класса и использование интерфейса `Supplier<T>` на примере (листинг 12.7).

Листинг 12.7. Ссылка на конструктор класса

```
import java.util.function.*;

public class MyClass {
   public static void main(String[] args) {
      Supplier<A> f = A::new;
      A obj = f.get();
      System.out.println(obj.getX());        // 10
      f = () -> new A();
      obj = f.get();
      System.out.println(obj.getX());        // 10
   }
}

class A {
   private int x;
   public A() {
      this.x = 10;
   }
   public int getX() {
      return this.x;
   }
}
```

Надеюсь, что принцип вам понятен. Берем подходящий функциональный интерфейс и указываем его в качестве типа данных. Далее сохраняем в переменной ссылку на общедоступный обычный метод, статический метод, конструктор или на лямбда-выражение. Когда будет необходимость, вызываем метод через название, указанное внутри блока интерфейса. Интерфейсы, объявленные в пакете `java.util.function`, подходят в большинстве случаев, так что не забывайте поглядывать на листинг 12.4. При необходимости вы можете создать свой собственный функциональный интерфейс.

12.11. Интерфейс *Comparable*

Если сравнивать объекты с помощью оператора ==, то будут сравниваться ссылки на объекты, а не свойства объектов. Для сравнения именно свойств объектов на равенство можно переопределить метод `equals()`, как мы уже делали в предыдущей главе. Если же нужно знать, не только равны ли объекты, но и больше или меньше они друг друга, например, для сортировки, то следует реализовать интерфейс `Comparable<T>`. Буква `T` означает произвольный тип. Обычно вместо этой буквы указывается название класса. Внутри интерфейса объявлен метод `compareTo()` со следующей сигнатурой:

```
int compareTo(T other);
```

Метод должен возвращать значение `0` — если объекты равны, положительное число — если текущий объект больше `other`, отрицательное число — если текущий объект меньше `other`. Причем должно соблюдаться следующее равенство:

```
знак(x.compareTo(y)) == -знак(y.compareTo(x))
```

Иными словами, если первое выражение возвратит положительное значение, то второе должно вернуть отрицательное (равенство самих значений не важно, учитывается только знак). Если первое выражение вернуло значение `0`, то и второе должно вернуть значение `0`. Если внутри одного выражения генерируется исключение, то и внутри второго также должно генерироваться исключение (это касается наследования и сравнения объектов базового и производного классов).

Создадим класс, реализующий интерфейс `Comparable<T>` (листинг 12.8).

Листинг 12.8. Интерфейс `Comparable<T>`

```
public class MyClass {
   public static void main(String[] args) {
      A obj1 = new A(10);
      A obj2 = new A(10);
      A obj3 = new A(3);
      A obj4 = new A(15);
      System.out.println(obj1.compareTo(obj2)); // 0
      System.out.println(obj1.compareTo(obj3)); // 1
      System.out.println(obj1.compareTo(obj4)); // -1
      System.out.println(obj4.compareTo(obj1)); // 1
   }
}

class A implements Comparable<A> {
   private int x;
   public A(int x) {
      this.x = x;
   }
   @Override
   public int compareTo(A other) {
      if (this.x > other.x) return 1;
      if (this.x < other.x) return -1;
      return 0;
   }
}
```

Метод не обязан возвращать только числа `1` или `–1`. Допустимо любое другое значение, но важен лишь знак числа. Например, метод `compareTo()` можно реализовать следующим образом:

```
public int compareTo(A other) {
   return this.x - other.x;
}
```

В этом случае важно следить, чтобы не было переполнения значения для типа данных, — ведь если оно случится, то знак может измениться на противоположный, и результат сравнения будет неправильным.

12.12. Интерфейс *Cloneable*

Интерфейс `Cloneable` следует реализовать, чтобы можно было правильно создавать копию объекта. Как уже говорилось ранее, метод `clone()` из класса `Object` создает «поверхностную» копию объекта, т. к. ничего не знает о свойствах объекта. Если объект содержит изменяемые объекты в полях, то будут скопированы лишь ссылки на эти объекты. В результате мы получим две ссылки на один объект, а не два объекта. Любое изменение объекта через одну ссылку приведет к изменению значения и в другой ссылке. Кроме того, метод `clone()` объявлен в классе `Object` как защищенный. Чтобы он стал общедоступным, необходимо реализовать интерфейс `Cloneable`.

Интерфейс `Cloneable`, в отличие от других интерфейсов, не содержит объявление метода. Он является лишь гарантией, что программист знает, что делает, когда объявляет метод общедоступным. Метод в любом случае в классе существует, т. к. он наследуется из класса `Object`. Вопрос лишь, защищенный он или общедоступный. Сигнатура метода:

```
Object clone() throws CloneNotSupportedException;
```

Метод `clone()` из класса `Object` может генерировать исключение `CloneNotSupportedException`. Внутри переопределенного метода можно обработать это исключение или просто добавить выражение `throws CloneNotSupportedException` в заголовок метода. Создадим класс, реализующий интерфейс `Cloneable`, и обработаем исключение внутри метода (листинг 12.9).

Листинг 12.9. Интерфейс `Cloneable`

```
import java.util.Date;

public class MyClass {
   public static void main(String[] args) {
      A obj1 = new A(10);
      A obj2 = obj1.clone();
      System.out.println(obj1.getD());
      System.out.println(obj2.getD());
      obj2.newDate(1245L);
      System.out.println(obj1.getD());
      System.out.println(obj2.getD());
   }
}

class A implements Cloneable {
   private int x;
   private Date d;   // Изменяемый объект!
```

```
    public A(int x) {
        this.x = x;
        this.d = new Date();
    }
    public int getX() {
        return this.x;
    }
    public Date getD() {
        // Изменяемые объекты возвращаем только так!
        return (Date) this.d.clone();
    }
    public void newDate(long time) {
        this.d.setTime(time);
    }
    @Override
    public A clone() {
        A obj = null;
        try {
            obj = (A) super.clone(); // Вызов метода из класса Object
        }
        catch (CloneNotSupportedException e) {
            e.printStackTrace();
            System.exit(1);
        }
        obj.d = (Date) this.d.clone();
        return obj;
    }
}
```

В этом примере класс `A` содержит изменяемое значение в поле `d`. Если создать «поверхностную» копию объекта, то с помощью метода `newDate()` можно будет изменить свойства объекта даты. Попробуйте закомментировать инструкцию:

```
obj.d = (Date) this.d.clone();
```

Результат выполнения программы будет примерно таким:

```
Sun Mar 25 03:12:09 MSK 2018
Sun Mar 25 03:12:09 MSK 2018
Thu Jan 01 03:00:01 MSK 1970
Thu Jan 01 03:00:01 MSK 1970
```

Последние две строки говорят, что значение изменилось в двух объектах. Если закомментированную инструкцию сделать опять активной, то результат станет иным:

```
Sun Mar 25 03:12:40 MSK 2018
Sun Mar 25 03:12:40 MSK 2018
Sun Mar 25 03:12:40 MSK 2018
Thu Jan 01 03:00:01 MSK 1970
```

Теперь все нормально, и мы получили именно копию объекта, включая копии изменяемых объектов внутри класса.

Обратите внимание на сигнатуру метода `clone()` при переопределении. Во-первых, метод `clone()` объявлен общедоступным, а не защищенным. Во-вторых, в качестве типа возвращаемого значения мы указали класс `A` вместо класса `Object`. Возвращение объекта производного класса вместо объекта базового класса является допустимой операцией при переопределении методов.

ГЛАВА 13

Обобщенные типы

При изучении функциональных интерфейсов в предыдущей главе мы столкнулись с какими-то заглавными (большими) буквами внутри угловых скобок вместо указания типа данных. Например:

```
Function<T, R>
```

Внутри интерфейса объявление метода выглядит следующим образом:

```
R apply(T t);
```

Такие заглавные буквы (`T` и `R`) символизируют произвольные типы данных и называются *обобщенными типами данных*. Вместо этих букв надо подставить названия классов. Если нужно использовать элементарные типы, то следует указывать названия классов-«оберток» над элементарными типами. Например, необходимо реализовать такое лямбда-выражение:

```
Function<T, R> f = (T n) -> n * 2;
```

Пока еще это всего лишь шаблон нашего выражения. Чтобы этот код стал рабочим, необходимо вместо обобщенного типа `T` подставить тип параметра, а вместо обобщенного типа `R` — тип результата выполнения. Так как буквы разные, мы можем использовать как разные типы, так и одинаковые. Например, внутри лямбда-выражения умножим целое число на `2` и вернем результат, имеющий также целочисленный тип:

```
Function<Integer, Integer> f = (Integer n) -> n * 2;
int x = f.apply(20);
System.out.println(x); // 40
```

Здесь вместо букв `T` и `R` мы указали один и тот же тип `Integer`, который является классом-«оберткой» над элементарным типом `int`. При работе с обобщенными типами мы не можем использовать элементарные типы. Например, следующая инструкция не будет скомпилирована:

```
Function<int, int> f = (int n) -> n * 2; // Ошибка!
```

Всегда следует использовать классы-«обертки» над элементарными типами. При указании значений не нужно явно создавать экземпляр класса `Integer`. Мы можем

просто указать значение, и компилятор автоматически «упакует» его в экземпляр соответствующего класса:

```
Integer y = 10;
int z = y;
System.out.println(z); // 10
```

Такой процесс называют *автоупаковкой*. Как видно из примера, присвоение объекта целочисленной переменной автоматически преобразует объект в число. Хотя мы можем выполнить эту операцию и явным образом:

```
Integer y = Integer.valueOf(10);
int z = y.intValue();
System.out.println(z); // 10
```

Преимущество обобщенных типов (в народе называемых *дженериками* — запомните это слово, оно часто используется для обозначения обобщенных типов на различных форумах в Интернете) заключается в возможности создания множества различных версий. Например, оператор * мы можем использовать не только с переменными, имеющими тип `int`. Давайте укажем тип `double`:

```
Function<Double, Double> f = (n) -> n * 2;
double x = f.apply(20.5);
System.out.println(x); // 41.0
```

На основании одного шаблона мы создали два различных варианта. Существуют и ограничения использования обобщенных типов. Не все классы можно указать при объявлении, т. к. внутри лямбда-выражения могут быть различные выражения, ограничивающие использование произвольных типов. В нашем примере таким ограничителем является оператор *. Компилятор не позволит нам указать типы, которые не поддерживает этот оператор. Ведь компилятор знает, какие типы мы указали при объявлении, и проверит возможность их использования.

Напомню, что функциональные интерфейсы и лямбда-выражения, которые мы использовали в этих примерах, поддерживаются языком Java, только начиная с восьмой версии, однако обобщенные типы («дженерики») введены в состав языка еще в пятой версии. Давайте рассмотрим обобщенные типы подробно.

13.1. Зачем нужны обобщенные типы?

Предположим, что необходимо внутри класса хранить объекты любого класса. Мы знаем, что переменные с типом базового класса могут хранить объекты любого производного класса. Для любого класса базовым будет класс `Object`. Отлично, создаем класс `Box` с полем, имеющим тип `Object`, и методы доступа к нему:

```
class Box {
   private Object obj;

   public Box(Object obj) {
      this.setObj(obj);
   }
```

```
   public Object getObj() {
      return obj;
   }

   public void setObj(Object obj) {
      this.obj = obj;
   }
}
```

Проверим возможность сохранения любого объекта:

```
Box box1 = new Box("Строка");
Box box2 = new Box(10);
```

Действительно работает. Но сохранить объект — это всего лишь полдела. После того, как мы достали объект, необходимо выполнить приведение типов:

```
String s = (String) box1.getObj();
int x = (int) box2.getObj();
System.out.println(s); // Строка
System.out.println(x); // 10
```

Пока мы знаем, что хранит класс `Box`, все будет под нашим контролем. Однако если мы не знаем этого, то придется проверить тип (с помощью оператора `instanceof` или метода `getClass()`) перед приведением типов:

```
Box box = new Box("Строка");
box.setObj(10);              // Мы можем менять объект!
String s = "";
int x = 0;
if (box.getObj() instanceof String) {
   s = (String) box.getObj();
   System.out.println(s);
}
else if (box.getObj().getClass() == Integer.class) {
   x = (int) box.getObj();
   System.out.println(x);
}
```

Проблема в том, что классов может быть неограниченное количество, и все их с помощью этого способа не проверить, нельзя также ограничить добавление произвольных типов, и компилятор не предупредит. Обобщенные типы позволяют решить эти проблемы. Давайте переделаем наш класс и сделаем его обобщенным:

```
class Box <T> {
   private T obj;

   public Box(T obj) {
      this.setObj(obj);
   }
```

```
    public T getObj() {
        return obj;
    }

    public void setObj(T obj) {
        this.obj = obj;
    }
}
```

Здесь мы просто заменили класс `Object` обобщенным типом `T` и добавили объявление обобщенного типа в заголовок класса. Теперь создадим экземпляр этого класса, заменив обобщенный тип типом `String`:

```
Box<String> box = new Box<String>("Строка");
box.setObj("Строка 2");          // OK
// box.setObj(10);               // Ошибка
String s = box.getObj();
System.out.println(s);           // Строка 2
```

Реальный тип указывается внутри угловых скобок после названия класса. Обратите внимание: никаких операций приведения в этом коде нет, т. к. мы указали, объекты какого класса должны храниться в классе `Box<T>`. Положить на хранение объект другого класса уже нельзя, и ошибка будет выявлена на этапе компиляции, а не на этапе выполнения, поэтому никаких проверок типов выполнять не нужно, — все приведения типов компилятор добавит самостоятельно.

Что делать, если мы хотим хранить числа? Все просто. Нужно создать объект класса `Box<T>`, указав тип `Integer`:

```
Box<Integer> box = new Box<Integer>(20);
// box.setObj("Строка 2");       // Ошибка
box.setObj(10);                  // OK
int x = box.getObj();
System.out.println(x);           // 10
```

Таким образом мы можем хранить данные любого типа, но одновременно только одного. При этом компилятор контролирует типы и не позволит положить на хранение объект другого типа. Лучше уж получить ошибку на этапе компиляции, чем в процессе выполнения программы.

13.2. Обобщенные классы

Определение обобщенного класса выглядит следующим образом:

```
[Модификаторы] class Название <Обобщенные типы через запятую> {
    // Блок класса
}
```

После названия класса указываются угловые скобки, внутри которых мы можем объявить обобщенный тип, — т. е. те самые заглавные буквы, что мы использовали

в примерах ранее. Давайте создадим обобщенный класс с одним обобщенным типом T, полем и конструктором класса:

```
class ClassA <T> {
   public T obj;

   public ClassA(T obj) {
      this.obj = obj;
   }
}
```

Буква T — это обобщенный тип, который мы можем использовать внутри класса вместо реального типа данных. Посмотрите на объявление поля obj — вместо реального типа данных указан обобщенный тип T. Точно так же обобщенный тип указан и в параметре конструктора класса. Когда мы будем создавать экземпляр этого класса, то укажем реальный тип данных:

```
ClassA<Integer> c = new ClassA<Integer>(10);
System.out.println(c.obj); // 10
```

При создании экземпляра обобщенного класса реальный тип указывается после названия класса внутри угловых скобок. В нашем примере вместо обобщенного типа T будет подставлен тип Integer.

Угловые скобки не обязательно должны указываться слитно с названием класса — можно и так:

```
ClassA <Integer> c = new ClassA <Integer> (10);
```

Вместо обобщенного типа T при создании экземпляра класса можно указать любой реальный объектный тип. Например, тип Double:

```
ClassA<Double> c = new ClassA<Double>(10.5);
System.out.println(c.obj); // 10.5
```

Или тип Date:

```
ClassA<Date> c = new ClassA<Date>(new Date());
System.out.println(c.obj.getTime()); // 1521938128101
```

При объявлении класса можно указать сразу несколько обобщенных типов через запятую:

```
class ClassB <T1, T2> {
   public T1 obj1;
   public T2 obj2;
   public T2 obj3;
   int x;

   public ClassB(T1 obj1, T2 obj2, T2 obj3, int x) {
      this.obj1 = obj1;
      this.obj2 = obj2;
```

```
        this.obj3 = obj3;
        this.x = x;
    }
}
```

В этом примере объявлены два обобщенных типа: `T1` и `T2`. Обратите внимание на то, что один обобщенный тип используется при объявлении двух полей. Мы можем объявить сколько угодно полей с обобщенным типом. Кроме того, внутри обобщенного класса можно использовать и реальные типы данных, и обычные методы. В приведенном примере мы объявили целочисленное поле `x`. Создадим экземпляр такого класса:

```
ClassB<Integer, Double> c =
      new ClassB<Integer, Double>(10, 1.5, 45.9, 8);
System.out.println(c.obj1); // 10
System.out.println(c.obj2); // 1.5
System.out.println(c.obj3); // 45.9
System.out.println(c.x);    // 8
```

Если мы при объявлении переменной ошибемся и укажем внутри угловых скобок разные типы, то компилятор выведет сообщение об ошибке:

```
ClassB<Integer, Double> c =
      new ClassB<Integer, Integer>(10, 1.5, 45.9, 8);
```

В этом примере после ключевого слова `new` внутри угловых скобок вместо типа `Double` указан тип `Integer`, что является ошибкой. На самом деле, начиная с Java 7, разрешается после ключевого слова `new` внутри угловых скобок вообще не задавать типы, ведь они и так указаны при объявлении переменной. Достаточно просто поставить пустые угловые скобки:

```
ClassB<Integer, Double> c = new ClassB<>(10, 1.5, 45.9, 8);
```

Такие пустые угловые скобки принято называть *ромбовидным оператором*.

Начиная с Java 10, вместо типа данных мы можем указать слово `var`:

```
var c = new ClassB<Integer, Double>(10, 1.5, 45.9, 8);
```

Используя ромбовидный оператор, эту инструкцию можно сделать еще короче:

```
var c = new ClassB<>(10, 1.5, 45.9, 8);
```

13.3. Ограничение обобщенного типа

При объявлении обобщенного типа можно сразу ограничить его названием базового класса или интерфейса. Для этого после обобщенного типа указывается ключевое слово `extends`, а после него — название одного базового класса и, через символ `&`, названия одного или нескольких интерфейсов. Обратите внимание: базовый класс всегда указывается перед интерфейсами. Например, ограничим обобщенный тип классами, производными от класса `Number`:

```
class ClassA <T extends Number> {
   public T obj;

   public ClassA(T obj) {
      this.obj = obj;
   }
}
```

Класс `Number` является базовым для классов `Integer` и `Double`, поэтому мы можем их использовать вместо обобщенного типа:

```
ClassA<Integer> c1 = new ClassA<Integer>(10);
System.out.println(c1.obj); // 10
ClassA<Double> c2 = new ClassA<Double>(10.5);
System.out.println(c2.obj); // 10.5
```

А вот класс `Date` не является наследником `Number` и мы не можем его использовать:

```
ClassA<Date> c = new ClassA<Date>(new Date()); // Ошибка!
```

Мы можем ограничить обобщенный тип несколькими интерфейсами. В этом случае они записываются через символ `&` — обычно используется запятая, но она уже занята для записи обобщенных типов, поэтому разработчики для указания нескольких интерфейсов и ввели новый символ (листинг 13.1).

Листинг 13.1. Ограничение обобщенного типа двумя интерфейсами

```
public class MyClass {
   public static void main(String[] args) {
      ClassA<A> c = new ClassA<A>(new A());
      c.test();
   }
}

interface ITest1 {
   void test1();
}
interface ITest2 {
   void test2();
}

class A implements ITest1, ITest2 {
   @Override
   public void test1() { System.out.println("test1()"); }
   @Override
   public void test2() { System.out.println("test2()"); }
}

class ClassA <T extends ITest1 & ITest2> {
   public T obj;
```

```
   public ClassA(T obj) {
      this.obj = obj;
   }
   public void test() {
      this.obj.test1();
      this.obj.test2();
   }
}
```

Всякий раз, когда вы что-то пытаетесь сделать с переменной, имеющей обобщенный тип (без указанного ограничения) внутри класса, и у вас не получается это сделать, представьте, что переменная является экземпляром класса `Object`. То, что вы пытаетесь сделать, можно сделать с экземпляром класса `Object`? Через обобщенный тип мы имеем доступ только к методам класса `Object`. Когда мы накладываем ограничение классом, то становятся доступными методы этого класса, и обобщенный тип будет иметь тип ограничивающего класса. Если наложено ограничение интерфейсом, то обобщенный тип будет иметь тип интерфейса, и мы сможем получить доступ к методам, объявленным в интерфейсе.

13.4. Обобщенные методы

Если метод является статическим, то обобщенный тип, объявленный в заголовке обобщенного класса, внутри статического метода использовать нельзя. Такое же ограничение имеют статические переменные класса. Объявление обобщенного типа для статического метода выполняется по следующей схеме:

```
[Модификатор] static <Обобщенный_тип>
              Тип_результата Название_метода(Параметры) {
}
```

Внутри угловых скобок мы можем указать обобщенный тип, а также ограничение типа. Вызов обобщенного метода осуществляется по следующей схеме:

```
Класс.<Тип>Название_метода(Значения)
```

Название типа вообще можно опустить:

```
Класс.Название_метода(Значения)
```

Пример использования обобщенных статических методов показан в листинге 13.2.

Листинг 13.2. Статические обобщенные методы

```
public class MyClass {
   public static void main(String[] args) {
      MyClass.<Integer>print(10);
      MyClass.<String>print("Строка");
      MyClass.print(10);
      MyClass.print("Строка");
      A obj = new A();
```

```
      MyClass.<A>echo(obj);
      MyClass.echo(obj);
   }
   public static <T> void print(T obj) {
      System.out.println(obj.toString());
   }
   // Ограничение интерфейсом ITest
   public static <T extends ITest> void echo(T obj) {
      obj.print(); // Вызов метода из интерфейса
   }
}
interface ITest {
   void print();
}
class A implements ITest {
   private int x = 20;
   @Override
   public void print() {
      System.out.println("x = " + this.x);
   }
}
```

Внутри обычных методов, а также в качестве типа параметров и типа возвращаемого значения, мы можем использовать обобщенный тип, объявленный в заголовке обобщенного класса. Кроме того, мы можем дополнительно указать обобщенный тип, как и в случае со статическим обобщенным методом. Причем, обобщенные обычные методы могут быть как внутри обобщенного класса, так и внутри обычного класса (листинг 13.3).

Листинг 13.3. Обобщенные методы

```
public class MyClass {
   public static void main(String[] args) {
      A obj1 = new A();
      obj1.<Integer>print(10);
      obj1.print(10);

      B<Integer> obj2 = new B<Integer>(50);
      obj2.<String>print(10, "строка");
      obj2.print(10, "строка");
   }
}
class A {
   // Обобщенный метод внутри обычного класса
   public <T> void print(T obj) {
      System.out.println(obj.toString());
   }
}
```

```
class B <T> {
   private T obj;
   // Обобщенный тип в параметре конструктора
   public B(T obj) {
      this.obj = obj;
   }
   // Дополнительный обобщенный тип
   public <U> void print(T obj1, U obj2) {
      System.out.println(obj1.toString());
      System.out.println(obj2.toString());
      System.out.println(this.obj.toString());
   }
}
```

13.5. Маски типов

При передаче объектов обобщенных классов в качестве значения параметра метода, а также при объявлении переменной, можно использовать следующие конструкции:

- `<?>` — **любой тип;**
- `<? extends` Класс *или* `T>` — **класс** Класс **и все производные классы;**
- `<? super` Класс *или* `T>` — **класс** Класс **и все базовые классы.**

Пример использования масок типов приведен в листинге 13.4.

Листинг 13.4. Маски типов

```
public class MyClass {
   public static void main(String[] args) {
      Box<Integer> obj1 = new Box<Integer>(10);
      Box<Double> obj2 = new Box<Double>(5.6);
      MyClass.print1(obj1);            // 10
      // Можно только Box<Integer>
      // MyClass.print1(obj2);
      MyClass.print2(obj1);            // 10
      MyClass.print2(obj2);            // 5.6
      MyClass.print3(obj1);            // 10.0
      MyClass.print3(obj2);            // 5.6
      MyClass.print4(obj1);            // 10.0
      MyClass.print4(obj2);            // 5.6
      MyClass.print5(obj1);            // 10
      // Можно только Box<Integer> и Box<Number>
      // MyClass.print5(obj2);
      Box<Number> obj3 = new Box<Number>(5.6);
      MyClass.print5(obj3);            // 5.6
   }
```

```
    public static void print1(Box<Integer> obj) {
        System.out.println(obj.getObj().intValue());
    }
    // Любые типы
    public static void print2(Box<?> obj) {
        System.out.println(obj.getObj().toString());
    }
    // Класс Number и производные классы
    public static void print3(Box<? extends Number> obj) {
        System.out.println(obj.getObj().doubleValue());
        // Тип не знаем
        // obj.setObj(obj.getObj()); // Ошибка
    }
    // Класс Number и производные классы
    public static <T extends Number> void print4(Box<T> obj) {
        System.out.println(obj.getObj().doubleValue());
        // Дополнительно знаем тип T
        obj.setObj(obj.getObj());      // OK
    }
    // Класс Integer и базовые классы (Number и Object)
    public static void print5(Box<? super Integer> obj) {
        System.out.println(obj.getObj().toString());
    }
}

class Box <T> {
    private T obj;

    public Box(T obj) {
        this.setObj(obj);
    }

    public T getObj() {
        return obj;
    }

    public void setObj(T obj) {
        this.obj = obj;
    }
}
```

Обратите внимание на методы `print3()` и `print4()`. Первый метод использует маски типов, а второй — ограничение обобщенного типа. Разница между этими способами передачи значений в том, что в первом методе мы ничего не знаем об исходном типе, поэтому не можем изменить объект, а во втором методе — знаем тип `T` и спокойно можем изменить объект. Чтобы узнать тип при использовании масок, необ-

ходимо создать вспомогательный обобщенный метод и вызвать его. Давайте заменим метод `print3()` следующим кодом:

```
public static void print3(Box<? extends Number> obj) {
   System.out.println(obj.getObj().doubleValue());
   // Тип не знаем
   // obj.setObj(obj.getObj()); // Ошибка
   MyClass.test(obj);
}
public static <T> void test(Box<T> obj) {
   obj.setObj(obj.getObj());      // OK
}
```

Внутри метода `test()` мы уже знаем тип и можем изменить объект.

13.6. Наследование обобщенных классов

Обобщенные классы могут участвовать в иерархии наследования. Предположим, у нас есть обобщенный класс `A`:

```
class A <T> {
   private T obj;

   public A(T obj) {
      this.obj = obj;
   }
   public T getObj() {
      return this.obj;
   }
   public void test() {
      System.out.println("A obj = " + this.obj);
   }
}
```

Чтобы наследовать этот класс, необходимо передать информацию о типе в производный класс в строке заголовка, а затем внутри конструктора передать информацию в базовый класс. Если внутри производного класса используется свой обобщенный тип, то в строке заголовка необходимо указать оба обобщенных типа через запятую. Создадим класс `B`, наследующий класс `A`:

```
class B <T, U> extends A <T> { // Передаем информацию о типе T
   private U n;

   public B(T obj, U n) {
      super(obj);              // Передаем в базовый класс
      this.n = n;
   }
   public U getN() {
      return this.n;
   }
```

```
    @Override
    public void test() {
        super.test();
        System.out.println("B n = " + this.n);
    }
}
```

При создании экземпляра класса необходимо внутри угловых скобок указать реальные типы вместо обобщенных:

```
B<Integer, Double> obj = new B<Integer, Double>(10, 5.6);
obj.test();
int x = obj.getObj();
double y = obj.getN();
System.out.println("x = " + x + " y = " + y);
```

Обычный класс может наследовать обобщенный класс. Например, создадим класс C, который наследует обобщенный класс A:

```
class C <T> extends A <T> { // Передаем информацию о типе T
    private int x;

    public C(T obj, int x) {
        super(obj);
        this.x = x;
    }
    public int getX() {
        return this.x;
    }
    @Override
    public void test() {
        super.test();
        System.out.println("C x = " + this.x);
    }
}
```

Создадим экземпляр этого класса:

```
C<Integer> obj = new C<Integer>(10, 5);
obj.test();
int x = obj.getObj();
int y = obj.getX();
System.out.println("x = " + x + " y = " + y);
```

При наследовании вместо обобщенного типа можно сразу указать реальный тип данных. Давайте создадим класс D, наследующий классы C и A:

```
class D extends C <Integer> {
    private int y;

    public D(Integer obj, int x, int y) {
        super(obj, x);
```

```
        this.y = y;
    }
    public int getY() {
        return this.y;
    }
    @Override
    public void test() {
        super.test();
        System.out.println("D y = " + this.y);
    }
}
```

В этом случае класс становится обычным, и экземпляр создается, как при использовании обычных классов:

```
D obj = new D(10, 5, 3);
obj.test();
int x = obj.getObj();
int y = obj.getX();
int z = obj.getY();
System.out.println("x = " + x + " y = " + y + " z = " + z);
```

Итак, у нас есть иерархия классов. Класс `D` наследует классы `C` и `A`. Как вы уже знаете, переменная с типом базового класса может хранить объекты производных классов. При использовании обобщенных классов все точно так же, только дополнительно учитывается обобщенный тип. Этот тип должен быть одинаковым:

```
A<Integer> obj = new A<Integer>(10);
obj.test();
obj = new C<Integer>(10, 5);
obj.test();
obj = new D(10, 5, 3);
obj.test();
// obj = new C<Double>(10.5, 5);          // Ошибка
A<Double> obj2 = new C<Double>(10.5, 5); // OK
obj2.test();
```

Необходимо понимать, что при полиморфизме не учитывается наследственность обобщенного типа. Например, класс `Number` является базовым для классов `Integer`, `Long` и `Double`. Следовательно, мы можем написать так:

```
Number x = Integer.valueOf(10);
Number y = Long.valueOf(10L);
Number z = Double.valueOf(10.2);
```

При использовании этих типов вместо обобщенного типа мы так написать не можем:

```
A<Number> x = new A<Integer>(10);          // Ошибка
A<Number> y = new A<Long>(10L);            // Ошибка
A<Number> z = new A<Double>(10.2);         // Ошибка
```

A<Number>, A<Integer>, A<Long> и A<Double> — абсолютно разные типы, между ними нет отношений. Поэтому только так:

```
A<Integer> x = new A<Integer>(10);          // OK
A<Long>    y = new A<Long>(10L);            // OK
A<Double>  z = new A<Double>(10.2);         // OK
```

Можно также объявить переменную с использованием маски типа и указать класс Number в качестве базового типа после ключевого слова extends:

```
A<? extends Number> x = new A<Integer>(10);     // OK
A<? extends Number> y = new A<Long>(10L);       // OK
A<? extends Number> z = new A<Double>(10.2);    // OK
A<? extends Number> a = new C<Integer>(10, 5); // OK
A<? extends Number> b = new D(10, 5, 3);        // OK
```

Тип Number может быть базовым для обобщенного типа, поэтому класс A<Number> может хранить объекты классов Integer, Long и Double:

```
A<Number>  a = new A<Number>(10);           // OK
A<Number>  b = new A<Number>(10L);          // OK
A<Number>  c = new A<Number>(10.2);         // OK
```

13.7. Обобщенные интерфейсы

Обобщенный интерфейс объявляется так же, как и обычный интерфейс, только после названия внутри угловых скобок указываются обобщенные типы через запятую. Этими обобщенными типами обозначаются типы параметров и типы возвращаемого значения. Пример интерфейса с одним обобщенным типом:

```
interface ITest1 <T> {
   void test(T t);
}
```

Пример обобщенного класса, реализующего этот интерфейс:

```
class A <T> implements ITest1 <T> {
   private T obj;

   public A(T obj) {
      this.obj = obj;
   }
   public T getObj() {
      return this.obj;
   }
   @Override
   public void test(T t) {
      System.out.println("A obj = " + this.obj);
      System.out.println("t = " + t);
   }
}
```

При объявлении переменной можно указать как тип класса, так и тип интерфейса:

```
A<Integer> obj1 = new A<Integer>(10);
obj1.test(15);
ITest1<Integer> obj2 = new A<Integer>(33);
obj2.test(81);
```

Если интерфейс содержит несколько обобщенных типов, то они приводятся внутри угловых скобок через запятую, при этом в объявлении класса, реализующем интерфейс, эти обобщенные типы также должны быть. Пример интерфейса с двумя обобщенными типами:

```
interface ITest2 <T, U> {
   void test1(T t);
   void test2(U u);
}
```

Пример класса, реализующего этот интерфейс:

```
class B <T, U> implements ITest2 <T, U> {
   private T obj;

   public B(T obj) {
      this.obj = obj;
   }
   public T getObj() {
      return this.obj;
   }
   @Override
   public void test1(T t) {
      System.out.println("t = " + t);
   }
   @Override
   public void test2(U u) {
      System.out.println("u = " + u);
   }
}
```

Создание экземпляра через название класса и интерфейса:

```
B<Integer, Double> obj1 = new B<Integer, Double>(10);
obj1.test1(15);
obj1.test2(20.5);
ITest2<Integer, Double> obj2 = new B<Integer, Double>(10);
obj2.test1(88);
obj2.test2(85.5);
```

Предыдущие классы были все обобщенными. Однако и простой класс может реализовать обобщенный интерфейс. Для этого при объявлении класса вместо обоб-

щенного типа указывается реальный тип. Создадим обычный класс C, реализующий интерфейс ITest1<T>:

```
class C implements ITest1 <Integer> {
   private int obj;

   public C(int obj) {
      this.obj = obj;
   }
   public int getObj() {
      return this.obj;
   }
   @Override
   public void test(Integer t) {
      System.out.println("C obj = " + this.obj);
      System.out.println("t = " + t);
   }
}
```

Создание экземпляра через название класса и интерфейса:

```
C obj1 = new C(10);
obj1.test(15);
ITest1<Integer> obj2 = new C(20);
obj2.test(88);
```

Обратите внимание: после названия интерфейса внутри угловых скобок необходимо указать реальный тип, совпадающий с типом, указанным при объявлении класса C.

13.8. Ограничения на использование обобщенных типов

При использовании обобщенных классов следует учитывать, что виртуальная машина ничего не знает об обобщенных классах, — она работает с обычными классами. На этапе компиляции производится преобразование обобщенного класса в обычный «сырой» класс, в котором все обобщенные типы заменяются либо классом Object, либо классом или интерфейсом, указанным в качестве ограничителя сразу после ключевого слова extends. При этом компилятор в нужных местах вставляет приведение типа к реальному типу, который мы указали при создании экземпляра класса. Например, возьмем класс A из предыдущего раздела, создадим два разных экземпляра и выведем название их класса:

```
A<Integer> obj1 = new A<Integer>(10);
A<Double> obj2 = new A<Double>(33.5);
System.out.println(obj1.getClass()); // class A
System.out.println(obj2.getClass()); // class A
```

Классы A<Integer> и A<Double> компилятор превращает в «сырой» класс A. Бессмысленно сравнивать эти классы с помощью оператора instanceof — он всегда вернет истину, т. к. это один и тот же «сырой» класс A. Компилятор даже не позволит их сравнивать. Только, если вместо реального типа подставить маску <?>.

Напомню вам резюме из *разд. 13.3*. Всякий раз, когда вы что-то пытаетесь сделать с переменной, имеющей обобщенный тип (без указанного ограничения) внутри класса, и у вас не получается это сделать, представьте, что переменная является экземпляром класса Object. То, что вы пытаетесь сделать, можно сделать с экземпляром класса Object? Когда мы накладываем ограничение классом, то становятся доступными методы этого класса, и обобщенный тип будет иметь тип ограничивающего класса. Если накладывается ограничение интерфейсом, то обобщенный тип будет иметь тип интерфейса, и мы сможем получить доступ к методам, объявленным в интерфейсе. Чем больше ограничений, тем больше мы можем сделать, но тем меньше смысла в использовании обобщенных классов.

Так как внутри «сырого» класса поля имеют объектный тип, мы не можем использовать элементарные типы. Вместо элементарных типов следует указывать названия классов-«оберток» над элементарными типами.

Внутри обобщенного класса нельзя создать экземпляр обобщенного типа и использовать выражение T.class, т. к. тип неизвестен:

```
T objT = new T(); // Ошибка!
```

Нельзя создать массив объектов обобщенного класса:

```
A<Integer>[] arr = new A<Integer>[5]; // Ошибка
```

Вместо создания массива мы можем воспользоваться обобщенным классом ArrayList, который подробно рассмотрим в следующей главе:

```
A<Integer> obj1 = new A<Integer>(10);
A<Integer> obj2 = new A<Integer>(33);
ArrayList< A<Integer> > arr = new ArrayList< A<Integer> >();
arr.add(obj1);
arr.add(obj2);
System.out.println(arr.get(0).getObj()); // 10
System.out.println(arr.get(1).getObj()); // 33
```

Не забудьте импортировать класс ArrayList с помощью инструкции:

```
import java.util.ArrayList;
```

Нельзя внутри обобщенного класса объявить статический член, использующий обобщенный тип, объявленный на уровне класса. При использовании статических методов можно использовать обобщенный тип, указанный после ключевого слова static:

```
public static <T> void print(T obj) {
   System.out.println(obj.toString());
}
```

Преобразование обобщенного класса в «сырой» класс может преподнести множество сюрпризов. Особенно на этапе наследования и переопределения методов базового обобщенного класса. Со всеми сюрпризами и дополнительными ограничениями вас познакомит компилятор при попытке создания реального обобщенного класса. Мы же на этом заканчиваем знакомство с обобщенными классами, но не прощаемся с ними. В двух следующих главах будет рассмотрена целая коллекция готовых обобщенных классов. Если вы не станете создавать свои обобщенные классы, то уж пользоваться этими готовыми классами будете с вероятностью 100%, — настолько они удобны в использовании и полезны в большинстве случаев.

ГЛАВА 14

Коллекции. Списки и очереди

Библиотека языка Java содержит множество обобщенных классов, реализующих различные структуры данных: списки, очереди, множества и словари. Все они объединены в единый каркас *коллекций*, внутри которого существует разделение на *интерфейсы* и *реализации*. В интерфейсах описан базовый набор методов, который должен реализовать какой-либо класс. Если класс создает список, то он должен реализовать интерфейс `List<E>`, если одностороннюю очередь — то интерфейс `Queue<E>` и т. д. Благодаря этому мы можем указать интерфейс в качестве типа данных и хранить в переменной ссылку на объект класса, реализующего этот интерфейс. Причем классы могут по-разному реализовывать управление структурой. Например, класс `ArrayList<E>` хранит список в виде последовательности (как в массиве), а класс `LinkedList<E>` — в виде отдельных элементов, содержащих ссылки на предыдущий и последующий элементы. В каждом из этих вариантов есть как достоинства, так и недостатки. Например, мы решили, что нам подходит класс `ArrayList<E>`, и создаем экземпляр класса:

```
List<Integer> arr = new ArrayList<Integer>();
```

Через некоторое время пришло осознание, что класс `LinkedList<E>` нам подходит лучше. Благодаря тому, что мы указали в качестве типа переменной название интерфейса, достаточно будет изменить только одну строку кода:

```
List<Integer> arr = new LinkedList<Integer>();
```

Весь остальной код будет работать без дополнительных изменений, но с другой реализацией.

14.1. Интерфейс *Collection<E>*

Интерфейс `Collection<E>` является базовым для большинства коллекций (исключением являются словари). Прежде чем использовать этот интерфейс, необходимо выполнить его импорт с помощью инструкции:

```
import java.util.Collection;
```

Иерархия наследования:

```
Iterable<T> - Collection<E>
```

Интерфейс `Collection<E>` используется в следующих иерархиях:

```
Список:     Iterable<T> - Collection<E> - List<E>
Очередь:    Iterable<T> - Collection<E> - Queue<E> - Deque<E>
Множество:  Iterable<T> - Collection<E> - Set<E> -
                          SortedSet<E> - NavigableSet<E>
```

Полный список методов, объявленных в этом и других интерфейсах, можно найти в документации. Если здесь просто привести форматы этих методов, то они ни о чем вам не скажут. В дальнейшем перед форматом метода мы будем указывать, к какому интерфейсу принадлежит метод, чтобы вы могли наглядно видеть, в каких интерфейсах объявлен метод. Ведь один метод может быть объявлен сразу в нескольких интерфейсах.

14.2. Интерфейсы *Iterable<T>* и *Iterator<T>*

Интерфейсы `Iterable<T>` и `Iterator<T>` используются для перебора элементов коллекции, а также для перебора любых объектов с помощью цикла `for each`. Интерфейс `Iterable<T>` содержит объявление одного метода:

```
Iterator<T> iterator()
```

Начиная с Java 8, интерфейс содержит также метод по умолчанию `forEach()`, который позволяет перебирать элементы коллекции вместо цикла `for each`. Формат метода:

```
default void forEach(Consumer<? super T> action)
```

В качестве значения параметра можно указать ссылку на метод или лямбда-выражение. Они принимают один параметр (текущий элемент коллекции) и ничего не возвращают:

```
// import java.util.Collections;
// import java.util.ArrayList;
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
arr.forEach( elem -> System.out.println(elem) );
```

Интерфейс `Iterator<T>` содержит объявление следующих методов:

- `hasNext()` — метод возвращает значение `true`, если можно получить следующий элемент коллекции, и `false` — в противном случае. Формат метода:

  ```
  public boolean hasNext()
  ```

- `next()` — перемещает указатель на одну позицию и возвращает элемент слева от указателя. Если был достигнут конец коллекции, метод генерирует исключение `NoSuchElementException`, поэтому, прежде чем вызвать этот метод, следует предварительно выполнить проверку с помощью метода `hasNext()`. Формат метода:

  ```
  public T next()
  ```

- `remove()` — метод по умолчанию. Если метод реализован, то он удаляет элемент, расположенный слева от указателя. Формат метода:

```
default void remove()
```

Позиция указателя всегда расположена между элементами. Сразу после создания итератора указатель находится перед первым элементом, поэтому попытка вызвать метод `remove()` приведет к ошибке. Вначале необходимо вызвать метод `next()` и переместить указатель, а лишь затем вызвать метод `remove()`:

```
// import java.util.*;
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
System.out.println(arr.toString()); // [1, 2, 3, 4]
Iterator<Integer> it = arr.iterator();
// it.remove(); // Ошибка
it.next();
it.remove();
System.out.println(arr.toString()); // [2, 3, 4]
```

Если вызвать метод `remove()` повторно, то опять возникнет ошибка. Перед повторным вызовом метода `remove()` снова следует вызвать метод `next()` и переместить указатель:

```
// import java.util.*;
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
System.out.println(arr.toString()); // [1, 2, 3, 4]
Iterator<Integer> it = arr.iterator();
it.next();
it.remove();
// it.remove(); // Ошибка
it.next();
it.remove();
System.out.println(arr.toString()); // [3, 4]
```

Последовательно вызывая метод `next()`, можно обойти все элементы коллекции. Например, используем цикл `while` и выведем все элементы в окно консоли:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
Iterator<Integer> it = arr.iterator();
while (it.hasNext()) {
   System.out.println(it.next());
}
```

При использовании цикла `for each` можно просто указать объект коллекции внутри цикла:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
```

```
for (Integer item: arr) {
   System.out.println(item);
}
```

Если внутри программы используются несколько итераторов, то важно понимать, что после модификации коллекции одним итератором второй итератор станет недействительным и будет генерировать исключение `ConcurrentModificationException`:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
Iterator<Integer> it = arr.iterator();
Iterator<Integer> it2 = arr.iterator();
it2.next();    // OK
it2.remove(); // OK
it.next();     // Ошибка! ConcurrentModificationException
```

В начале раздела мы упомянули, что с помощью цикла `for each` можно перебирать любые объекты, реализующие интерфейсы `Iterable<T>` и `Iterator<T>`. Давайте создадим класс, реализующий эти интерфейсы. Класс будет хранить строку, которую можно будет перебирать посимвольно (листинг 14.1).

Листинг 14.1. Интерфейсы `Iterable<T>` и `Iterator<T>`

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class MyClass {
   public static void main(String[] args) {
      PrintChar obj = new PrintChar("Строка");
      for (Character ch: obj) {
         System.out.println(ch);
      }
      obj.forEach( ch -> System.out.println(ch) );
      while (obj.hasNext()) {
         System.out.println(obj.next());
      }
   }
}

class PrintChar implements Iterable<Character>,
                           Iterator<Character> {
   private String str;
   private int index = 0;

   public PrintChar(String str) {
      this.str = str;
   }
```

```
    @Override
    public Iterator<Character> iterator() {
        return this;
    }
    @Override
    public boolean hasNext() {
        if (this.index < this.str.length()) return true;
        this.index = 0;
        return false;
    }
    @Override
    public Character next() {
        if (this.index >= this.str.length()) {
            this.index = 0;
            throw new NoSuchElementException();
        }
        this.index++;
        return this.str.charAt(this.index - 1);
    }
    @Override
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

14.3. Интерфейсы *Comparable<T>* и *Comparator<T>*

Чтобы можно было сравнивать элементы коллекции и сортировать их, необходимо чтобы объекты поддерживали интерфейс `Comparable<T>`. Интерфейс содержит один метод:

```
int compareTo(T other);
```

Метод должен возвращать значение `0` — если объекты равны, положительное число — если текущий объект больше `other`, отрицательное число — если текущий объект меньше `other`. Реализацию интерфейса `Comparable<T>` мы уже рассматривали с вами при изучении интерфейсов (см. *разд. 12.11*).

Класс может реализовать только одну версию интерфейса `Comparable<T>`. Однако часто приходится сортировать элементы коллекции в обратном порядке, или по каким-либо другим полям, или объекты вообще не поддерживают интерфейс `Comparable<T>`. В этом случае можно реализовать интерфейс `Comparator<T>` отдельно от основного класса и передать ссылку на объект в какой-либо метод. Интерфейс `Comparator<T>` содержит объявление одного метода:

```
int compare(T a, T b)
```

Метод должен возвращать значение `0` — если объекты равны, положительное число — если `a` больше `b`, отрицательное число — если `a` меньше `b`. Если класс реализует интерфейс `Comparable<T>`, то достаточно вернуть результат выполнения выражения `a.compareTo(b)`.

На самом деле интерфейс `Comparator<T>` содержит еще объявление метода `equals()`, но этот метод по умолчанию наследуется всеми классами из класса `Object`. Лучше, однако, если ваш класс будет содержать реализацию методов `equals()` и `hashCode()`. Объявление метода:

```
boolean equals(Object obj)
```

Реализовать функциональный интерфейс `Comparator<T>` можно в отдельном классе, внутри анонимного вложенного класса или с помощью лямбда-выражения (способ доступен, только начиная с Java 8). Давайте рассмотрим все эти способы на примере сортировки пользовательского объекта внутри списка по двум полям (листинг 14.2). Причем сделаем сортировку и по возрастанию, и по убыванию, а также создадим реализацию методов `compareTo()`, `equals()`, `hashCode()` и `toString()`.

Листинг 14.2. Интерфейсы `Comparable<T>` и `Comparator<T>`

```
import java.util.ArrayList;
import java.util.Comparator;

public class MyClass {
   public static void main(String[] args) {
      ArrayList<A> arr = new ArrayList<A>();
      arr.add(new A(1, 3));
      arr.add(new A(3, 1));
      arr.add(new A(2, 4));
      System.out.println(arr.toString());
      // [{1;3}, {3;1}, {2;4}]
      // Сортировка по x
      arr.sort(null);
      System.out.println(arr.toString());
      // [{1;3}, {2;4}, {3;1}]
      // Сортировка по y
      arr.sort(new Comparator<A>() {
         @Override
         public int compare(A a, A b) {
            if (a.getY() > b.getY()) return 1;
            if (a.getY() < b.getY()) return -1;
            return 0;
         }
      });
      System.out.println(arr.toString());
      // [{3;1}, {1;3}, {2;4}]
      // Сортировка по x по убыванию
      arr.sort(new ReverseX());
```

```
        System.out.println(arr.toString());
        // [{3;1}, {2;4}, {1;3}]
        // Сортировка по y по убыванию
        arr.sort( (a, b) -> {
              if (a.getY() > b.getY()) return -1;
              if (a.getY() < b.getY()) return 1;
              return 0;
           }
        );
        System.out.println(arr.toString());
        // [{2;4}, {1;3}, {3;1}]
    }
}

class A implements Comparable<A> {
    private int x;
    private int y;
    public A(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
    @Override
    public int compareTo(A other) {
        if (this.x > other.x) return 1;
        if (this.x < other.x) return -1;
        return 0;
    }
    @Override
    public String toString() {
        return "{" + this.x + ";" + this.y + "}";
    }
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + x;
        result = prime * result + y;
        return result;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
```

```
        if (obj == null) return false;
        if (this.getClass() != obj.getClass())
           return false;
        A other = (A) obj;
        if (this.x != other.x) return false;
        if (this.y != other.y) return false;
        return true;
    }
}

class ReverseX implements Comparator<A> {
    @Override
    public int compare(A a, A b) {
        return b.compareTo(a);
    }
}
```

Примечание

В Java 8 внутри интерфейса `Comparator<T>` были добавлены различные статические методы и методы по умолчанию. Полный список этих методов вы найдете в документации.

14.4. Интерфейс *List<E>*

Интерфейс `List<E>` описывает список. Иерархия наследования:

```
Iterable<T> - Collection<E> - List<E>
```

Прежде чем использовать этот интерфейс, необходимо выполнить его импорт с помощью инструкции:

```
import java.util.List;
```

Интерфейс `List<E>` реализуют следующие классы:

- `ArrayList<E>` — динамический список. Плюсы: быстрый доступ по индексу и быстрый перебор элементов коллекции. Минусы: медленная вставка, добавление и удаление элементов (требуется время на перемещение всех или некоторых элементов);
- `LinkedList<E>` — связанный список, элементы которого хранят ссылки на предыдущий и последующий элементы. Благодаря этому, вставка и удаление элементов происходит быстро, достаточно изменить несколько ссылок, а не перемещать элементы, как это делает класс `ArrayList<E>`. Но доступ к элементам по индексу осуществляется медленно. С помощью этого класса можно также реализовать очередь;
- `Vector<E>` — синхронизированный динамический список. Аналогичен по работе классу `ArrayList<E>`, но, в отличие от класса `ArrayList<E>`, класс `Vector<E>` явля-

ется синхронизированным и может быть использован для доступа из разных потоков. Из-за потерь на синхронизацию класс `Vector<E>` работает медленнее, чем класс `ArrayList<E>`.

14.5. Класс *ArrayList<E>*: динамический список

Класс `ArrayList<E>` реализует динамический упорядоченный список произвольных элементов (включая `null` и экземпляры обобщенных классов), как в обычном массиве. Добавить элементы можно в любое место списка, при этом размер списка будет автоматически увеличиваться по мере необходимости (в отличие от обычных массивов). Получить доступ к элементу списка можно по индексу или с помощью итератора. При удалении элемента производится сдвиг всех последующих элементов на одну позицию влево. При этом емкость списка не изменяется. Прежде чем использовать класс `ArrayList<E>`, необходимо импортировать его с помощью инструкции:

```
import java.util.ArrayList;
```

Класс `ArrayList<E>` реализует следующие интерфейсы:

```
Iterable<E>, Collection<E>, List<E>, RandomAccess, Serializable, Cloneable
```

14.5.1. Создание объекта

Создать список позволяют следующие конструкторы класса `ArrayList<E>`:

```
ArrayList()
ArrayList(int initialCapacity)
ArrayList(Collection<? extends E> c)
```

Первый конструктор создает пустой список с емкостью в десять элементов:

```
ArrayList<Integer> arr1 = new ArrayList<Integer>();
System.out.println(arr1.size());     // 0
arr1.add(10);
arr1.add(33);
System.out.println(arr1.get(0));     // 10
System.out.println(arr1.get(1));     // 33
System.out.println(arr1.toString()); // [10, 33]
```

Второй конструктор позволяет указать начальную емкость списка. Не путайте понятия «емкость» и «размер» списка. *Емкость* — это зарезервированное количество элементов, а *размер* — это количество реальных элементов списка. Емкость следует указывать, если планируется вставка большого количества элементов. Если зарезервированной емкости не хватит, то при достижении максимального количества элементов будет создан новый массив с емкостью:

```
Новая_емкость = (Старая_емкость * 3) / 2 + 1
```

При этом существующие элементы списка будут скопированы в новый массив. Чтобы не происходило лишнего пересоздания массива при каждой нехватке емкости, можно использовать второй конструктор и сразу указать необходимую начальную емкость:

```
ArrayList<Integer> arr1 = new ArrayList<Integer>(100);
System.out.println(arr1.size());      // 0
for (int i = 1; i < 101; i++) {
   arr1.add(i);
}
System.out.println(arr1.get(0));      // 1
System.out.println(arr1.get(99));     // 100
System.out.println(arr1.size());      // 100
```

Если бы мы сразу не указали начальную емкость, то выполнялись бы следующие промежуточные этапы:

```
Новая_емкость = 10
Новая_емкость = (10 * 3) / 2 + 1
Новая_емкость = (16 * 3) / 2 + 1
Новая_емкость = (25 * 3) / 2 + 1
Новая_емкость = (38 * 3) / 2 + 1
Новая_емкость = (58 * 3) / 2 + 1
Новая_емкость = (88 * 3) / 2 + 1
```

Целых шесть лишних этапов копирования элементов! А что, если бы мы хотели создать список из миллиона элементов? Этих этапов было бы непозволительно много. Поэтому, если вы изначально знаете максимальное количество элементов, следует под них зарезервировать место, указав начальную емкость в конструкторе класса или после создания с помощью метода `ensureCapacity()`. Формат метода:

```
public void ensureCapacity(int minCapacity)
```

Если указанное значение емкости меньше количества элементов списка, то список изменен не будет. Если больше, то емкость будет увеличена.

При удалении элементов емкость не изменяется, что может привести к напрасной трате памяти. Чтобы уменьшить емкость, следует вызвать метод `trimToSize()`. Формат метода:

```
public void trimToSize()
```

Третий конструктор позволяет создать список на основе другой коллекции. Создадим новый список на основе списка из предыдущего примера:

```
ArrayList<Integer> arr2 = new ArrayList<Integer>(arr1);
System.out.println(arr2.get(0));      // 1
System.out.println(arr2.get(99));     // 100
System.out.println(arr2.size());      // 100
```

Первоначально список всегда пустой. Если необходимо заполнить список одинаковыми значениями, то можно воспользоваться статическим методом `nCopies()` из

класса `Collections` для генерации значений, а затем передать результат конструктору класса. Формат метода:

```
import java.util.Collections;
public static <T> List<T> nCopies(int n, T o)
```

Пример заполнения списка одинаковыми значениями:

```
ArrayList<Integer> arr =
   new ArrayList<Integer>(Collections.nCopies(10, 0));
System.out.println(arr.toString());
// [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

В Java 9 в интерфейс `List<E>` был добавлен статический метод `of()`, который создает неизменяемый список. Форматы метода:

```
// Интерфейс List<E>
public static <E> List<E> of()
public static <E> List<E> of(E e1[, ..., E e10])
public static <E> List<E> of(E... elements)
```

Первый формат создает пустой неизменяемый список:

```
List<Integer> arr = List.<Integer>of();
System.out.println(arr.size()); // 0
// Ошибка! Нельзя добавить элемент, т. к. список неизменяемый
// arr.add(20);
```

Второй формат позволяет указать от одного до десяти элементов через запятую:

```
List<Integer> arr1 = List.of(1);
System.out.println(arr1); // [1]
List<Integer> arr2 = List.of(1, 2, 3, 4, 5);
System.out.println(arr2); // [1, 2, 3, 4, 5]
// Ошибка! Нельзя добавить элемент, т. к. список неизменяемый
// arr2.add(20);
Integer[] arrInt = {1, 2, 3, 4, 5};
List<Integer[]> arr3 = List.<Integer[]>of(arrInt);
System.out.println(arr3.size()); // 1
```

Третий формат позволяет указать произвольное количество элементов через запятую или в виде массива:

```
List<Integer> arr1 = List.of(1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6);
System.out.println(arr1); // [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6]
Integer[] arrInt = {1, 2, 3, 4, 5};
List<Integer> arr2 = List.of(arrInt);
System.out.println(arr2); // [1, 2, 3, 4, 5]
// Ошибка! Нельзя добавить элемент, т. к. список неизменяемый
// arr2.add(20);
```

В Java 10 в интерфейс `List<E>` был добавлен статический метод `copyOf()`, который создает неизменяемый список на основе другой коллекции. Формат метода:

```
// Интерфейс List<E>
public static <E> List<E> copyOf(Collection<? extends E> coll)
```

Пример:

```
var arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5);
List<Integer> arr2 = List.copyOf(arr);
System.out.println(arr2.toString()); // [1, 2, 3, 4, 5]
// Ошибка! Нельзя добавить элемент, т. к. список неизменяемый
// arr2.add(20);
```

Создать пустой неизменяемый список позволяет статический метод `emptyList()` из класса `Collections`. Такой список удобно возвращать из метода при отсутствии значений. Формат метода:

```
import java.util.Collections;
public static final <T> List<T> emptyList()
```

Пример:

```
List<Integer> arr = Collections.<Integer>emptyList();
System.out.println(arr.size()); // 0
// Ошибка! Нельзя добавить элемент, т. к. список неизменяемый
// arr.add(20);
```

14.5.2. Вставка элементов

Вставить элементы позволяют следующие методы:

- `add()` — добавляет один элемент. Форматы метода:

```
// Интерфейсы Collection<E> и List<E>
public boolean add(E e)
// Интерфейс List<E>
public void add(int index, E e)
```

Первый формат метода позволяет добавить элемент в конец списка. Метод возвращает значение `true`, если элемент был добавлен, и `false` — в противном случае:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
System.out.println(arr.add(10));        // true
arr.add(20);
System.out.println(arr.toString());     // [10, 20]
```

Второй формат позволяет добавить элемент в позицию, заданную индексом. Добавим элементы в начало и конец списка:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
arr.add(10);
arr.add(0, 20);           // В начало списка
arr.add(arr.size(), 33);  // В конец списка
System.out.println(arr.toString());     // [20, 10, 33]
```

- `addAll()` — добавляет несколько элементов из другой коллекции. Форматы метода:

```
// Интерфейсы Collection<E> и List<E>
public boolean addAll(Collection<? extends E> c)
// Интерфейс List<E>
public boolean addAll(int index, Collection<? extends E> c)
```

Первый формат добавляет элементы коллекции в конец списка. Метод возвращает значение `true`, если элементы были добавлены, и `false` — в противном случае:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
arr.add(10);
arr.add(20);
System.out.println(arr.toString());     // [10, 20]
ArrayList<Integer> arr2 = new ArrayList<Integer>();
arr2.add(30);
arr2.addAll(arr);
System.out.println(arr2.toString());    // [30, 10, 20]
```

Второй формат позволяет указать позицию вставки. При этом существующие элементы списка после указанного индекса сместятся к концу списка. Метод возвращает значение `true`, если элементы были добавлены, и `false` — в противном случае. Добавим элементы коллекции в начало списка:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
arr.add(10);
arr.add(20);
System.out.println(arr.toString());     // [10, 20]
ArrayList<Integer> arr2 = new ArrayList<Integer>();
arr2.add(30);
arr2.addAll(0, arr); // В начало списка
System.out.println(arr2.toString());    // [10, 20, 30]
```

Для добавления элементов можно также воспользоваться статическим методом `addAll()` из класса `Collections`. Прежде чем использовать этот класс, необходимо его импортировать с помощью инструкции:

```
import java.util.Collections;
```

Формат метода:

```
public static <T> boolean addAll(Collection<? super T> c, T... elements)
```

В первом параметре указывается ссылка на коллекцию, а во втором параметре можно указать значения через запятую или обычный массив. Новые элементы будут добавлены в конец списка. Метод возвращает значение `true`, если элементы были добавлены, и `false` — в противном случае:

```
Integer[] arrInt = {1, 2, 3};
ArrayList<Integer> arr = new ArrayList<Integer>();
```

```
Collections.addAll(arr, arrInt);     // Добавляем элементы из массива
Collections.addAll(arr, 4, 5, 6);    // Добавляем элементы
System.out.println(arr.toString()); // [1, 2, 3, 4, 5, 6]
```

С помощью статического метода `copy()` из класса `Collections` можно скопировать элементы из одного списка в начало другого. При этом исходный список должен содержать такое же количество элементов или большее, в противном случае будет ошибка. Формат метода:

```
import java.util.Collections;
public static <T> void copy(List<? super T> dest,
                            List<? extends T> src)
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5);
ArrayList<Integer> arr2 = new ArrayList<Integer>();
Collections.addAll(arr2, 6, 7, 8);
Collections.copy(arr, arr2);
System.out.println(arr.toString()); // [6, 7, 8, 4, 5]
```

14.5.3. Определение количества элементов

Для определения количества элементов в списке предназначен метод `size()`. Формат метода:

```
// Интерфейсы Collection<E> и List<E>
public int size()
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3);    // Добавляем элементы
System.out.println(arr.size());      // 3
System.out.println(arr.toString()); // [1, 2, 3]
```

С помощью метода `isEmpty()` можно проверить, содержит список элементы или нет. Метод возвращает значение `true`, если список пустой, и `false` — в противном случае. Формат метода:

```
// Интерфейсы Collection<E> и List<E>
public boolean isEmpty()
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
System.out.println(arr.isEmpty()); // true
Collections.addAll(arr, 1, 2, 3);
System.out.println(arr.isEmpty()); // false
```

14.5.4. Удаление элементов

Для удаления элементов предназначены следующие методы:

- `clear()` — удаляет все элементы из списка. Формат метода:

```
// Интерфейсы Collection<E> и List<E>
public void clear()
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3);
System.out.println(arr.toString()); // [1, 2, 3]
arr.clear();
System.out.println(arr.toString()); // []
```

Этот метод можно также использовать совместно с методом `subList()` для очистки диапазона списка от начального индекса до конечного (не включая элемент с этим индексом):

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
arr.subList(0, 2).clear();
System.out.println(arr.toString()); // [3, 4, 5, 6]
```

- `remove()` — удаляет один элемент из списка. Форматы метода:

```
// Интерфейс List<E>
public E remove(int index)
// Интерфейсы Collection<E> и List<E>
public boolean remove(Object o)
```

Первый формат удаляет элемент, расположенный по индексу, и возвращает его. Остальные элементы с указанного индекса сдвигаются к началу списка. Если индекс не существует, то генерируется исключение:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3);
System.out.println(arr.toString()); // [1, 2, 3]
System.out.println(arr.remove(1));  // 2
System.out.println(arr.toString()); // [1, 3]
```

Второй формат находит в списке первый элемент, соответствующий указанному объекту, и удаляет его. Остальные элементы сдвигаются к началу списка. Метод возвращает значение `true`, если элемент был удален, и `false` — в противном случае:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 2);
System.out.println(arr.toString()); // [1, 2, 3, 2]
System.out.println(arr.remove(Integer.valueOf(2)));  // true
System.out.println(arr.remove(Integer.valueOf(4)));  // false
System.out.println(arr.toString()); // [1, 3, 2]
```

- `removeAll()` — удаляет из списка все элементы, соответствующие элементам указанной коллекции. Метод возвращает значение `true`, если элементы были удалены, и `false` — в противном случае. Формат метода:

```
// Интерфейсы Collection<E> и List<E>
public boolean removeAll(Collection<?> c)
```

Пример:

```
ArrayList<Integer> arrDel = new ArrayList<Integer>();
Collections.addAll(arrDel, 1, 2);
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 2);
System.out.println(arr.toString()); // [1, 2, 3, 2]
System.out.println(arr.removeAll(arrDel));  // true
System.out.println(arr.removeAll(arrDel));  // false
System.out.println(arr.toString()); // [3]
```

- `removeIf()` — позволяет указать ссылку на метод, внутри которого нужно выполнить сравнение и вернуть логическое значение. Из списка будут удалены все элементы, для которых метод вернул значение `true`. Вместо указания ссылки на метод удобно использовать лямбда-выражение. Метод возвращает значение `true`, если элементы были удалены, и `false` — в противном случае. Метод доступен, начиная с Java 8. Формат метода:

```
// Интерфейсы Collection<E> и List<E>
default boolean removeIf(Predicate<? super E> filter)
```

Удалим все элементы, значение которых меньше `4`:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 2, 4, 5, 6);
System.out.println(arr.toString()); // [1, 2, 3, 2, 4, 5, 6]
System.out.println(
   arr.removeIf(elem -> elem < 4)); // true
System.out.println(arr.toString()); // [4, 5, 6]
```

- `retainAll()` — удаляет из списка все элементы, которые не содержатся в указанной коллекции. Метод возвращает значение `true`, если элементы были удалены, и `false` — в противном случае. Формат метода:

```
// Интерфейсы Collection<E> и List<E>
public boolean retainAll(Collection<?> c)
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6, 6);
ArrayList<Integer> arr2 = new ArrayList<Integer>();
Collections.addAll(arr2, 4, 5, 6);
System.out.println(arr.retainAll(arr2)); // true
System.out.println(arr.toString());      // [4, 5, 6, 6]
System.out.println(arr.retainAll(arr2)); // false
```

14.5.5. Доступ к элементам

Для доступа к элементам предназначены следующие методы:

- `get()` — возвращает элемент, расположенный по указанному индексу. Если индекс не существует, то генерируется исключение. Формат метода:

```
// Интерфейс List<E>
public E get(int index)
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3);
System.out.println(arr.get(0)); // 1
System.out.println(arr.get(1)); // 2
System.out.println(arr.get(2)); // 3
```

- `set()` — заменяет элемент, расположенный по указанному индексу, другим элементом. Возвращает старый элемент. Если индекс не существует, то генерируется исключение. Формат метода:

```
// Интерфейс List<E>
public E set(int index, E element)
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3);
System.out.println(arr.toString()); // [1, 2, 3]
System.out.println(arr.set(1, 8));  // 2
System.out.println(arr.toString()); // [1, 8, 3]
```

- `subList()` — возвращает срез списка от индекса `fromIndex` до `toIndex` (не включая элемент с этим индексом). Формат метода:

```
// Интерфейс List<E>
public List<E> subList(int fromIndex, int toIndex)
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
List<Integer> arr2 = arr.subList(0, 2);
System.out.println(arr2.toString());   // [1, 2]
arr2 = arr.subList(2, arr.size());
System.out.println(arr2.toString());   // [3, 4, 5, 6]
```

Обратите внимание на то, что все действия с этим срезом повлияют на исходный список:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
List<Integer> arr2 = arr.subList(0, 2);
System.out.println(arr2.toString());   // [1, 2]
```

```
arr2.set(0, 88);
System.out.println(arr2.toString());   // [88, 2]
System.out.println(arr.toString());    // [88, 2, 3, 4]
```

Благодаря этому можно, например, очистить диапазон списка с помощью метода `clear()`:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
arr.subList(0, 2).clear();
System.out.println(arr.toString()); // [3, 4, 5, 6]
```

Для заполнения всех элементов одним и тем же значением можно воспользоваться статическим методом `fill()` из класса `Collections`. Формат метода:

```
import java.util.Collections;
public static <T> void fill(List<? super T> list, T obj)
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 0, 0, 0, 0, 0);
System.out.println(arr.toString());   // [0, 0, 0, 0, 0]
Collections.fill(arr, 1);
System.out.println(arr.toString());   // [1, 1, 1, 1, 1]
```

14.5.6. Поиск и замена элементов в списке

Для поиска и замены элементов в списке предназначены следующие методы:

- `indexOf()` — находит первый элемент, соответствующий указанному объекту, и возвращает его индекс или значение –1, означающее, что элемент не найден. Формат метода:

  ```
  // Интерфейс List<E>
  public int indexOf(Object o)
  ```

  Пример:

  ```
  ArrayList<Integer> arr = new ArrayList<Integer>();
  Collections.addAll(arr, 1, 2, 3, 2);
  System.out.println(arr.indexOf(2)); // 1
  System.out.println(arr.indexOf(4)); // -1
  ```

- `lastIndexOf()` — находит последний элемент, соответствующий указанному объекту, и возвращает его индекс или значение –1, означающее, что элемент не найден. Формат метода:

  ```
  // Интерфейс List<E>
  public int lastIndexOf(Object o)
  ```

  Пример:

  ```
  ArrayList<Integer> arr = new ArrayList<Integer>();
  Collections.addAll(arr, 1, 2, 3, 2);
  ```

```
System.out.println(arr.lastIndexOf(2)); // 3
System.out.println(arr.lastIndexOf(4)); // -1
```

- `contains()` — возвращает значение `true`, если элемент существует в списке, и `false` — в противном случае. Формат метода:

```
// Интерфейсы Collection<E> и List<E>
public boolean contains(Object o)
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 2);
System.out.println(arr.contains(2)); // true
System.out.println(arr.contains(4)); // false
```

- `containsAll()` — возвращает значение `true`, если все элементы из указанной коллекции существуют в списке, и `false` — в противном случае. Формат метода:

```
// Интерфейсы Collection<E> и List<E>
public boolean containsAll(Collection<?> c)
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
ArrayList<Integer> arr2 = new ArrayList<Integer>();
Collections.addAll(arr2, 4, 3, 2);
System.out.println(arr.containsAll(arr2)); // true
arr2.add(8);
System.out.println(arr.containsAll(arr2)); // false
```

- `replaceAll()` — в качестве параметра принимает ссылку на метод, внутри которого нужно вернуть новый элемент или старый. Метод доступен, начиная с Java 8. Формат метода:

```
// Интерфейс List<E>
default void replaceAll(UnaryOperator<E> operator)
```

Умножим каждый элемент списка на 2, используя лямбда-выражение:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
arr.replaceAll( elem -> elem * 2 );
System.out.println(arr.toString()); // [2, 4, 6, 8, 10, 12]
```

Для замены всех вхождений значения `oldVal` новым значением `newVal` можно воспользоваться статическим методом `replaceAll()` из класса `Collections`. Метод возвращает значение `true`, если были замены в списке, и `false` — в противном случае. Формат метода:

```
import java.util.Collections;
public static <T> boolean replaceAll(List<T> list, T oldVal, T newVal)
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 2, 5, 2);
System.out.println(Collections.replaceAll(arr, 2, 88)); // true
System.out.println(arr.toString()); // [1, 88, 3, 88, 5, 88]
System.out.println(Collections.replaceAll(arr, 7, 88)); // false
```

Выполнить бинарный поиск элемента внутри отсортированного списка позволяет статический метод `binarySearch()` из класса `Collections`. Метод возвращает индекс найденного элемента. Если элемент не найден, то возвращает отрицательное число, указывающее позицию вставки (определяется по формуле: `-(index) - 1`) для сохранения порядка сортировки. Если элементов несколько, то возвращается индекс любого из элементов, а не самого первого. Если список не отсортирован, то результат не определен. Форматы метода:

```
import java.util.Collections;
public static <T> int binarySearch(
       List<? extends Comparable<? super T>> list, T key)
public static <T> int binarySearch(List<? extends T> list,
       T key, Comparator<? super T> c)
```

Пример использования первого формата:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 3, 2, 3, 2, 8);
arr.sort(null);
System.out.println(arr.toString()); // [1, 2, 2, 3, 3, 8]
int index = Collections.binarySearch(arr, 3);
System.out.println(index); // 4
index = Collections.binarySearch(arr, 5);
System.out.println(index); // -6
arr.add(-(index) - 1, 5);
System.out.println(arr.toString()); // [1, 2, 2, 3, 3, 5, 8]
```

Второй формат позволяет дополнительно указать способ сравнения. Этим же способом должен быть отсортирован и список. Пример поиска строки без учета регистра символов:

```
ArrayList<String> arr = new ArrayList<String>();
Collections.addAll(arr, "еда", "единица1", "Единый", "Единица2");
arr.sort(String.CASE_INSENSITIVE_ORDER);
System.out.println(arr.toString());
// [еда, единица1, Единица2, Единый]
int index = Collections.binarySearch(arr, "Единица1",
            String.CASE_INSENSITIVE_ORDER);
System.out.println(index); // 1
```

Статический метод `frequency()` из класса `Collections` позволяет посчитать количество элементов с указанным значением. Формат метода:

```
import java.util.Collections;
public static int frequency(Collection<?> c, Object o)
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 3, 2, 3, 2, 2);
System.out.println(Collections.frequency(arr, 2)); // 3
System.out.println(Collections.frequency(arr, 1)); // 1
System.out.println(Collections.frequency(arr, 8)); // 0
```

Для получения позиций вхождения одного списка в другой предназначены статические методы `indexOfSubList()` и `lastIndexOfSubList()` из класса `Collections`. Форматы методов:

```
import java.util.Collections;
public static int indexOfSubList(List<?> source, List<?> target)
public static int lastIndexOfSubList(List<?> source, List<?> target)
```

Метод `indexOfSubList()` возвращает индекс первого вхождения одного списка в другой, а метод `lastIndexOfSubList()` — индекс последнего вхождения. Если вхождение не найдено, то возвращается значение –1:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6, 3, 4, 5);
ArrayList<Integer> arr2 = new ArrayList<Integer>();
Collections.addAll(arr2, 3, 4);
System.out.println(
   Collections.indexOfSubList(arr, arr2));     // 2
System.out.println(
   Collections.lastIndexOfSubList(arr, arr2)); // 6
arr2.add(8);
System.out.println(
   Collections.indexOfSubList(arr, arr2));     // -1
System.out.println(
   Collections.lastIndexOfSubList(arr, arr2)); // -1
```

14.5.7. Поиск минимального и максимального значений в списке

Для поиска минимального и максимального значений в списке предназначены следующие статические методы из класса `Collections`:

- `min()` — выполняет поиск минимального значения в списке и возвращает элемент. Форматы метода:

```
import java.util.Collections;
public static <T extends Object & Comparable<? super T>> T
       min(Collection<? extends T> coll)
import java.util.Comparator;
public static <T> T min(Collection<? extends T> coll,
                        Comparator<? super T> comp)
```

Первый формат выполняет поиск, используя для сравнения метод `compareTo()` из интерфейса `Comparable<T>`:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
Integer min = Collections.min(arr);
System.out.println(min); // 1
```

Второй формат позволяет во втором параметре указать экземпляр класса, реализующего интерфейс `Comparator<T>`:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
Integer min = Collections.min(arr, new Comparator<Integer>() {
   @Override
   public int compare(Integer x, Integer y) {
      return x.compareTo(y);
   }
});
System.out.println(min); // 1
```

- `max()` — **выполняет поиск максимального значения в списке и возвращает элемент. Форматы метода:**

```
import java.util.Collections;
public static <T extends Object & Comparable<? super T>> T
       max(Collection<? extends T> coll)
import java.util.Comparator;
public static <T> T max(Collection<? extends T> coll,
                        Comparator<? super T> comp)
```

Первый формат выполняет поиск, используя для сравнения метод `compareTo()` из интерфейса `Comparable<T>`:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
Integer max = Collections.max(arr);
System.out.println(max); // 6
```

Второй формат позволяет во втором параметре указать экземпляр класса, реализующего интерфейс `Comparator<T>`:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
Integer max = Collections.max(arr, new Comparator<Integer>() {
   @Override
   public int compare(Integer x, Integer y) {
      return x.compareTo(y);
   }
});
System.out.println(max); // 6
```

14.5.8. Преобразование массива в список и списка в массив

Для преобразования обычного массива в список можно воспользоваться статическим методом `addAll()` из класса `Collections`. Прежде чем использовать этот класс, необходимо его импортировать с помощью инструкции:

```
import java.util.Collections;
```

Формат метода:

```
public static <T> boolean addAll(Collection<? super T> c, T... elements)
```

В первом параметре указывается ссылка на коллекцию, а во втором параметре можно указать значения через запятую или обычный массив. Новые элементы будут добавлены в конец списка. Метод возвращает значение `true`, если элементы были добавлены, и `false` — в противном случае:

```
Integer[] arrInt = {1, 2, 3};
ArrayList<Integer> arr = new ArrayList<Integer>();
arr.add(55);
Collections.addAll(arr, arrInt);     // Добавляем элементы из массива
System.out.println(arr.toString()); // [55, 1, 2, 3]
```

Статический метод `asList()` из класса `Arrays` позволяет «обернуть» список вокруг обычного массива. При этом мы можем выполнять большинство действий, предназначенных для списка, и все изменения затронут исходный массив. Следует учитывать, что мы можем получить элемент и изменить его, можем перебирать все элементы, но нельзя добавить новый элемент или удалить существующий. Размер списка будет фиксированным. Формат метода:

```
import java.util.Arrays;
public static <T> List<T> asList(T... a)
```

Пример:

```
Integer[] arrInt = {1, 2, 3, 4};
List<Integer> arr = Arrays.asList(arrInt);
System.out.println(arr.toString()); // [1, 2, 3, 4]
arr.set(0, 88);
System.out.println(arr.toString()); // [88, 2, 3, 4]
// Исходный массив изменился
System.out.println(Arrays.toString(arrInt)); // [88, 2, 3, 4]
```

Вместо массива можно указать несколько значений через запятую:

```
List<Integer> arr = Arrays.asList(1, 2, 3, 4);
System.out.println(arr.toString()); // [1, 2, 3, 4]
```

Чтобы получить копии элементов массива, нужно передать «обернутый» массив конструктору класса `ArrayList<E>`:

```
Integer[] arrInt = {1, 2, 3, 4};
ArrayList<Integer> arr =
   new ArrayList<Integer>(Arrays.asList(arrInt));
```

```
System.out.println(arr.toString());          // [1, 2, 3, 4]
arr.set(0, 88);
System.out.println(arr.toString());          // [88, 2, 3, 4]
// Исходный массив не изменился
System.out.println(Arrays.toString(arrInt)); // [1, 2, 3, 4]
```

Преобразовать список в массив позволяет метод `toArray()`. Форматы метода:

```
// Интерфейсы Collection<E> и List<E>
public Object[] toArray()
public <T> T[] toArray(T[] a)
```

Первый формат возвращает массив объектов класса `Object`:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3);
System.out.println(arr.toString());          // [1, 2, 3]
Object[] arrObj = arr.toArray();
System.out.println(Arrays.toString(arrObj)); // [1, 2, 3]
Integer x = (Integer) arrObj[0];
System.out.println(x);                       // 1
```

Второй формат позволяет в качестве параметра указать ссылку на массив, который будет заполнен элементами из списка. Метод возвращает ссылку на заполненный массив:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3);
System.out.println(arr.toString());          // [1, 2, 3]
Integer[] arrInt = arr.toArray(new Integer[arr.size()]);
System.out.println(Arrays.toString(arrInt)); // [1, 2, 3]
```

В этом примере мы создаем новый массив, указывая количество элементов списка, и передаем его методу `toArray()`. Метод заполнит массив значениями из списка и вернет ссылку на него.

Если длина указанного массива будет меньше размера списка, то метод создаст новый массив того же типа, заполнит его элементами из списка и вернет ссылку на него. Массив, указанный в качестве параметра, изменен в этом случае не будет:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3);
System.out.println(arr.toString());          // [1, 2, 3]
Integer[] arrInt = arr.toArray(new Integer[0]);
System.out.println(Arrays.toString(arrInt)); // [1, 2, 3]
```

Если длина указанного массива больше размера списка, то элементы будут скопированы в этот массив, начиная с нулевого индекса. После копирования всех элементов будет добавлен еще один элемент со значением `null`. В этом случае метод вернет ссылку на первоначальный массив:

```
Integer[] arrInt = {8, 8, 8, 8, 8, 8, 8};
ArrayList<Integer> arr = new ArrayList<Integer>();
```

```
Collections.addAll(arr, 1, 2, 3);
System.out.println(arr.toString());          // [1, 2, 3]
arrInt = arr.toArray(arrInt);
System.out.println(Arrays.toString(arrInt));
// [1, 2, 3, null, 8, 8, 8]
```

14.5.9. Перемешивание и переворачивание списка

Для перемешивания и переворачивания списка предназначены следующие статические методы из класса `Collections`:

- `shuffle()` — перемешивает элементы списка случайным образом. Форматы метода:

```
import java.util.Collections;
public static void shuffle(List<?> list)
import java.util.Random;
public static void shuffle(List<?> list, Random rnd)
```

Первый формат использует генератор случайных чисел по умолчанию:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
Collections.shuffle(arr);
System.out.println(arr.toString()); // [4, 5, 3, 2, 6, 1]
```

Второй формат позволяет указать ссылку на генератор случайных чисел:

```
Random rnd = new Random();
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
Collections.shuffle(arr, rnd);
System.out.println(arr.toString()); // [5, 1, 3, 6, 4, 2]
```

- `reverse()` — меняет порядок следования элементов на противоположный (как бы переворачивает список). Формат метода:

```
import java.util.Collections;
public static void reverse(List<?> list)
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
Collections.reverse(arr);
System.out.println(arr.toString()); // [6, 5, 4, 3, 2, 1]
```

- `rotate()` — сдвигает все элементы на указанное расстояние. Элементы из конца списка будут перемещены в начало. Метод как бы вращает элементы внутри списка, при этом не меняя их последовательность. Формат метода:

```
import java.util.Collections;
public static void rotate(List<?> list, int distance)
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5);
Collections.rotate(arr, 2);
System.out.println(arr.toString()); // [4, 5, 1, 2, 3]
Collections.rotate(arr, 3);
System.out.println(arr.toString()); // [1, 2, 3, 4, 5]
```

- `swap()` — меняет два элемента местами. Формат метода:

```
import java.util.Collections;
public static void swap(List<?> list, int i, int j)
```

Поменяем местами первый и второй элементы:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
Collections.swap(arr, 0, 1);
System.out.println(arr.toString());   // [2, 1, 3, 4]
```

14.5.10. Сортировка элементов списка

Для сортировки элементов списка предназначен метод `sort()` (начиная с Java 8). Формат метода:

```
// Интерфейс List<E>
default void sort(Comparator<? super E> c)
```

Если в качестве параметра указать значение `null`, то будет произведена сортировка по возрастанию с использованием метода `compareTo()` из интерфейса `Comparable<T>`:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 4, 5, 3, 2, 6, 1);
arr.sort(null);
System.out.println(arr.toString()); // [1, 2, 3, 4, 5, 6]
```

Если нужно выполнить сортировку объектов, не поддерживающих интерфейс `Comparable<T>`, или отсортировать список в обратном порядке, то можно реализовать интерфейс `Comparator<T>`:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 4, 5, 3, 2, 6, 1);
arr.sort(new Comparator<Integer>() {
   @Override
   public int compare(Integer x, Integer y) {
      return y.compareTo(x);
   }
});
System.out.println(arr.toString()); // [6, 5, 4, 3, 2, 1]
```

Для сортировки в обратном порядке можно также воспользоваться статическим методом `reverseOrder()` из класса `Collections`. Форматы метода:

```
import java.util.Collections;
public static <T> Comparator<T> reverseOrder()
public static <T> Comparator<T> reverseOrder(Comparator<T> cmp)
```

Пример сортировки списка в обратном порядке:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 4, 5, 3, 2, 6, 1);
arr.sort(Collections.reverseOrder());
System.out.println(arr.toString()); // [6, 5, 4, 3, 2, 1]
arr.sort(Collections.reverseOrder((x, y) -> y.compareTo(x)));
System.out.println(arr.toString()); // [1, 2, 3, 4, 5, 6]
```

Начиная с Java 8, внутри интерфейса `Comparator<T>` **были добавлены статические методы** `naturalOrder()` **и** `reverseOrder()`**, которые можно использовать для сортировки списка по возрастанию и убыванию соответственно. Форматы методов:**

```
import java.util.Comparator;
static <T extends Comparable<? super T>> Comparator<T> naturalOrder()
static <T extends Comparable<? super T>> Comparator<T> reverseOrder()
```

Пример сортировки по возрастанию и убыванию:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 4, 5, 3, 2, 6, 1);
arr.sort(Comparator.naturalOrder());
System.out.println(arr.toString()); // [1, 2, 3, 4, 5, 6]
arr.sort(Comparator.reverseOrder());
System.out.println(arr.toString()); // [6, 5, 4, 3, 2, 1]
```

Если список содержит строки, и необходимо отсортировать элементы без учета регистра символов, то можно воспользоваться статической константой `CASE_INSENSITIVE_ORDER` **из класса** `String`**:**

```
public static final Comparator<String> CASE_INSENSITIVE_ORDER
```

Пример сортировки списка строк без учета регистра символов:

```
ArrayList<String> arr = new ArrayList<String>();
Collections.addAll(arr, "единица1", "Единый", "Единица2");
arr.sort(null);
System.out.println(arr.toString());
// [Единица2, Единый, единица1]
arr.sort(String.CASE_INSENSITIVE_ORDER);
System.out.println(arr.toString());
// [единица1, Единица2, Единый]
```

Для сортировки элементов списка можно также воспользоваться статическим методом `sort()` **из класса** `Collections`**. Форматы метода:**

```
import java.util.Collections;
public static <T extends Comparable<? super T>> void sort(List<T> list)
import java.util.Comparator;
public static <T> void sort(List<T> list,
                            Comparator<? super T> c)
```

Пример сортировки списка в прямом и обратном порядке разными способами:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 4, 5, 3, 2, 6, 1);
Collections.sort(arr);
System.out.println(arr.toString()); // [1, 2, 3, 4, 5, 6]
Collections.sort(arr, Collections.reverseOrder());
System.out.println(arr.toString()); // [6, 5, 4, 3, 2, 1]
Collections.sort(arr, (x, y) -> {
   return x.compareTo(y);
});
System.out.println(arr.toString()); // [1, 2, 3, 4, 5, 6]
Collections.sort(arr, new Comparator<Integer>() {
   @Override
   public int compare(Integer x, Integer y) {
      return y.compareTo(x);
   }
});
System.out.println(arr.toString()); // [6, 5, 4, 3, 2, 1]
```

14.5.11. Перебор элементов списка

Перебрать все элементы списка можно с помощью цикла `for each`, итератора и метода `forEach()` (начиная с Java 8). Пример использования цикла `for each`:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
for (Integer item: arr) {
   System.out.println(item);
}
```

Чтобы получить итератор, необходимо вызвать метод `iterator()`. Формат метода:

```
// Интерфейсы Iterable<T>, Collection<E> и List<E>
public Iterator<E> iterator()
```

Пример перебора элементов списка с помощью итератора и цикла `while`:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
Iterator<Integer> it = arr.iterator();
while (it.hasNext()) {
   System.out.println(it.next());
}
```

Начиная с Java 8, для перебора элементов используется также метод `forEach()`. Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
arr.forEach( elem -> System.out.println(elem) );
```

Если нам необходимо изменять что-то в списке, то можно воспользоваться перебором списка с помощью цикла `for`. Умножим все элементы списка на 2:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
int x = 0;
for (int i = 0, j = arr.size(); i < j; i++) {
   x = arr.get(i);
   x *= 2;
   arr.set(i, x);
}
System.out.println(arr.toString()); // [2, 4, 6, 8]
```

Начиная с Java 8, доступен также метод `replaceAll()`, принимающий в качестве параметра ссылку на метод, внутри которого нужно вернуть новый элемент или старый. Формат метода:

```
// Интерфейс List<E>
public void replaceAll(UnaryOperator<E> operator)
```

Умножим каждый элемент списка на 2, используя лямбда-выражение:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
arr.replaceAll( elem -> elem * 2 );
System.out.println(arr.toString()); // [2, 4, 6, 8]
```

14.5.12. Интерфейс *ListIterator<E>*

Для перебора элементов массива можно также воспользоваться методами из интерфейса `ListIterator<E>`. Прежде чем использовать этот интерфейс, необходимо выполнить его импорт с помощью инструкции:

```
import java.util.ListIterator;
```

Получить этот итератор из списка позволяет метод `listIterator()`. Форматы метода:

```
// Интерфейс List<E>
public ListIterator<E> listIterator()
public ListIterator<E> listIterator(int index)
```

Интерфейс `ListIterator<E>` содержит объявление следующих методов:

- `hasNext()` — метод возвращает значение `true`, если можно получить следующий элемент коллекции, и `false` — в противном случае. Формат метода:

  ```
  public boolean hasNext()
  ```

- `next()` — перемещает указатель на одну позицию и возвращает элемент слева от указателя. Если был достигнут конец коллекции, метод генерирует исключение `NoSuchElementException`, поэтому, прежде чем вызвать этот метод, следует предварительно выполнить проверку с помощью метода `hasNext()`. Формат метода:

  ```
  public E next()
  ```

Переберем все элементы списка с начала и до конца:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
ListIterator<Integer> it = arr.listIterator();
while (it.hasNext()) {
   System.out.println(it.next());
}
```

- `hasPrevious()` — метод возвращает значение `true`, если можно получить предыдущий элемент коллекции, и `false` — в противном случае. Формат метода:

```
public boolean hasPrevious()
```

- `previous()` — перемещает указатель на одну позицию назад и возвращает элемент справа от указателя. Формат метода:

```
public E previous()
```

Переберем все элементы списка с конца до начала:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
ListIterator<Integer> it = arr.listIterator(arr.size());
while (it.hasPrevious()) {
   System.out.println(it.previous());
}
```

- `remove()` — удаляет элемент слева от указателя (если вызван метод `next()`) или справа (если вызван метод `previous()`). Формат метода:

```
public void remove()
```

Перед удалением элемента или перед повторной операцией удаления необходимо вызвать метод `next()` или `previous()`:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
ListIterator<Integer> it = arr.listIterator();
// it.remove(); // Ошибка
it.next();
it.remove();
// it.remove(); // Ошибка
it.next();
it.remove();
System.out.println(arr.toString()); // [3, 4]
```

- `nextIndex()` — возвращает индекс следующего элемента. Если позиция указателя за последним элементом, то метод вернет количество элементов списка. Формат метода:

```
public int nextIndex()
```

- `previousIndex()` — возвращает индекс предыдущего элемента. Если позиция указателя перед первым элементом, то метод вернет значение `-1`. Формат метода:

```
public int previousIndex()
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
ListIterator<Integer> it = arr.listIterator();
System.out.println(it.previousIndex()); // -1
it.next();
System.out.println(it.nextIndex());     // 1
System.out.println(it.previousIndex()); // 0
it = arr.listIterator(arr.size());
System.out.println(it.nextIndex());     // 4
it.previous();
System.out.println(it.nextIndex());     // 3
System.out.println(it.previousIndex()); // 2
```

- `add()` — добавляет элемент слева от указателя. Формат метода:

```
public void add(E e)
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3);
ListIterator<Integer> it = arr.listIterator();
it.add(0); // В начало списка
System.out.println(arr.toString()); // [0, 1, 2, 3]
it = arr.listIterator(arr.size());
it.add(4); // В конец списка
System.out.println(arr.toString()); // [0, 1, 2, 3, 4]
```

- `set()` — заменяет элемент, возвращенный последним вызовом метода `next()` или `previous()`. Если перед этим был вызван метод `add()` или метод `remove()`, то следует предварительно вызвать метод `next()` или `previous()`, иначе будет ошибка. Формат метода:

```
public void set(E e)
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3);
ListIterator<Integer> it = arr.listIterator();
it.next();
it.set(0); // Изменяем первый элемент
System.out.println(arr.toString()); // [0, 2, 3]
it = arr.listIterator(arr.size());
it.previous();
it.set(88); // Изменяем последний элемент
System.out.println(arr.toString()); // [0, 2, 88]
```

14.6. Интерфейсы *Queue<E>* и *Deque<E>*

Интерфейс `Queue<E>` описывает одностороннюю очередь, а интерфейс `Deque<E>` — двухстороннюю. Иерархия наследования:

```
Iterable<T> - Collection<E> - Queue<E> - Deque<E>
```

Прежде чем использовать эти интерфейсы, необходимо выполнить их импорт с помощью инструкций:

```
import java.util.Queue;
import java.util.Deque;
```

Интерфейсы `Queue<E>` и `Deque<E>` реализуют следующие классы:

- `ArrayDeque<E>` — двухсторонняя очередь. Добавить элементы можно только в начало или в конец очереди. Получить элементы также можно только из начала или из конца очереди, но не из середины;
- `LinkedList<E>` — связанный список с возможностями двухсторонней очереди;
- `PriorityQueue<E>` — реализует очередь с приоритетами.

Пример реализации очереди «первым пришел, первым ушел» (обычная очередь, например, за билетами в кинотеатр, — чем раньше встали в очередь, тем быстрее купите билет):

```
Queue<Integer> arr = new ArrayDeque<Integer>();
arr.offer(1);           // Добавляем элементы в очередь
arr.offer(2);
arr.offer(3);
arr.offer(4);
while (!arr.isEmpty()) { // Обрабатываем элементы
   System.out.print(arr.poll() + " ");
} // 1 2 3 4
```

Или с помощью методов `add()` и `remove()`, которые генерируют исключение в случае неудачи:

```
Queue<Integer> arr = new ArrayDeque<Integer>();
arr.add(1);             // Добавляем элементы в очередь
arr.add(2);
arr.add(3);
arr.add(4);
while (!arr.isEmpty()) { // Обрабатываем элементы
   System.out.print(arr.remove() + " ");
} // 1 2 3 4
```

Пример реализации стека (очереди, в которой пришедший последним уходит первым):

```
Deque<Integer> arr = new ArrayDeque<Integer>();
arr.push(1);            // Добавляем элементы в стек
arr.push(2);
```

```
arr.push(3);
arr.push(4);
while (!arr.isEmpty()) { // Обрабатываем элементы
   System.out.print(arr.pop() + " ");
} // 4 3 2 1
```

14.7. Класс *ArrayDeque<E>*: двухсторонняя очередь

Класс `ArrayDeque<E>` реализует двухстороннюю очередь. Добавить элементы можно только в начало или в конец очереди. Получить элементы также можно только из начала или из конца очереди, но не из середины. Этот класс не позволяет хранить элементы со значением `null`, т. к. существуют методы, возвращающие это значение для индикации ошибки. Прежде чем использовать класс `ArrayDeque<E>`, необходимо импортировать его с помощью инструкции:

```
import java.util.ArrayDeque;
```

Класс `ArrayDeque<E>` реализует следующие интерфейсы:

```
Iterable<E>, Collection<E>, Deque<E>, Queue<E>, Serializable, Cloneable
```

14.7.1. Создание объекта

Создать очередь позволяют следующие конструкторы класса `ArrayDeque<E>`:

```
ArrayDeque()
ArrayDeque(int numElements)
ArrayDeque(Collection<? extends E> c)
```

Первый конструктор создает пустую очередь с емкостью в 16 элементов, второй — позволяет указать емкость, а третий — заполняет очередь из элементов указанной коллекции:

```
ArrayDeque<Integer> arr =
   new ArrayDeque<Integer>(Collections.nCopies(10, 0));
System.out.println(arr.toString());
// [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

14.7.2. Вставка элементов

Вставить элементы позволяют следующие методы:

- `push()` — добавляет элемент в начало очереди. Формат метода:

  ```
  // Интерфейс Deque<E>
  public void push(E e)
  ```

  Пример:

  ```
  ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
  arr.add(20);
  ```

```
arr.push(10);
System.out.println(arr.toString());     // [10, 20]
```

- `addFirst()` — **добавляет элемент в начало очереди. Формат метода:**

```
// Интерфейс Deque<E>
public void addFirst(E e)
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
arr.add(20);
arr.addFirst(10);
System.out.println(arr.toString());     // [10, 20]
```

- `offerFirst()` — **добавляет элемент в начало очереди. Метод возвращает значение** `true`**, если элемент был добавлен. Формат метода:**

```
// Интерфейс Deque<E>
public boolean offerFirst(E e)
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
arr.add(20);
System.out.println(arr.offerFirst(10)); // true
System.out.println(arr.toString());     // [10, 20]
```

- `add()` — **добавляет элемент в конец очереди. Метод возвращает значение** `true`**, если элемент был добавлен. В противном случае генерируется исключение. Формат метода:**

```
// Интерфейсы Collection<E>, Queue<E> и Deque<E>
public boolean add(E e)
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
System.out.println(arr.add(10));        // true
arr.add(20);
System.out.println(arr.toString());     // [10, 20]
```

- `addLast()` — **добавляет элемент в конец очереди. Формат метода:**

```
// Интерфейс Deque<E>
public void addLast(E e)
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
arr.add(20);
arr.addLast(10);
System.out.println(arr.toString());     // [20, 10]
```

- `offer()` — **добавляет элемент в конец очереди. Метод возвращает значение** `true`**, если элемент был добавлен. Формат метода:**

```
// Интерфейсы Queue<E> и Deque<E>
public boolean offer(E e)
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
arr.add(20);
System.out.println(arr.offer(10));      // true
System.out.println(arr.toString());     // [20, 10]
```

- `offerLast()` — добавляет элемент в конец очереди. Метод возвращает значение `true`, если элемент был добавлен. Формат метода:

```
// Интерфейс Deque<E>
public boolean offerLast(E e)
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
arr.add(20);
System.out.println(arr.offerLast(10)); // true
System.out.println(arr.toString());    // [20, 10]
```

- `addAll()` — добавляет несколько элементов из другой коллекции. Метод возвращает значение `true`, если элементы были добавлены. Формат метода:

```
// Интерфейсы Collection<E>, Queue<E> и Deque<E>
public boolean addAll(Collection<? extends E> c)
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
ArrayList<Integer> arr2 = new ArrayList<Integer>();
Collections.addAll(arr2, 1, 2, 3);
arr.addAll(arr2);
System.out.println(arr.toString()); // [1, 2, 3]
```

Для добавления элементов можно также воспользоваться статическим методом `addAll()` из класса `Collections`:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3);
System.out.println(arr.toString()); // [1, 2, 3]
```

14.7.3. Определение количества элементов

Для определения количества элементов предназначен метод `size()`. Формат метода:

```
// Интерфейсы Collection<E>, Queue<E> и Deque<E>
public int size()
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3);
```

```
System.out.println(arr.size());     // 3
System.out.println(arr.toString()); // [1, 2, 3]
```

С помощью метода `isEmpty()` можно проверить, содержит очередь элементы или нет. Метод возвращает значение `true`, если очередь пуста, и `false` — в противном случае. Формат метода:

```
// Интерфейсы Collection<E>, Queue<E> и Deque<E>
public boolean isEmpty()
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
System.out.println(arr.isEmpty()); // true
Collections.addAll(arr, 1, 2, 3);
System.out.println(arr.isEmpty()); // false
```

14.7.4. Удаление элементов

Для удаления элементов предназначены следующие методы:

- `clear()` — удаляет все элементы из очереди. Формат метода:

  ```
  // Интерфейсы Collection<E>, Queue<E> и Deque<E>
  public void clear()
  ```

  Пример:

  ```
  ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
  Collections.addAll(arr, 1, 2, 3);
  System.out.println(arr.toString()); // [1, 2, 3]
  arr.clear();
  System.out.println(arr.toString()); // []
  ```

- `removeAll()` — удаляет из очереди все элементы, соответствующие элементам указанной коллекции. Метод возвращает значение `true`, если элементы были удалены, и `false` — в противном случае. Формат метода:

  ```
  // Интерфейсы Collection<E>, Queue<E> и Deque<E>
  public boolean removeAll(Collection<?> c)
  ```

  Пример:

  ```
  ArrayList<Integer> arrDel = new ArrayList<Integer>();
  Collections.addAll(arrDel, 1, 2);
  ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
  Collections.addAll(arr, 1, 2, 3, 2);
  System.out.println(arr.toString()); // [1, 2, 3, 2]
  System.out.println(arr.removeAll(arrDel));  // true
  System.out.println(arr.removeAll(arrDel));  // false
  System.out.println(arr.toString()); // [3]
  ```

- `removeIf()` — позволяет указать ссылку на метод, внутри которого нужно выполнить сравнение и вернуть логическое значение. Из очереди будут удалены

все элементы, для которых метод вернул значение `true`. Вместо указания ссылки на метод удобно использовать лямбда-выражение. Метод возвращает значение `true`, если элементы были удалены, и `false` — в противном случае. Метод доступен, начиная с Java 8. Формат метода:

```
// Интерфейсы Collection<E>, Queue<E> и Deque<E>
default boolean removeIf(Predicate<? super E> filter)
```

Удалим все элементы, значения которых меньше 4:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3, 2, 4, 5, 6);
System.out.println(arr.toString()); // [1, 2, 3, 2, 4, 5, 6]
System.out.println(
   arr.removeIf(elem -> elem < 4)); // true
System.out.println(arr.toString()); // [4, 5, 6]
```

- `retainAll()` — удаляет из очереди все элементы, которые не содержатся в указанной коллекции. Метод возвращает значение `true`, если элементы были удалены, и `false` — в противном случае. Формат метода:

```
// Интерфейсы Collection<E>, Queue<E> и Deque<E>
public boolean retainAll(Collection<?> c)
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6, 6);
ArrayList<Integer> arr2 = new ArrayList<Integer>();
Collections.addAll(arr2, 4, 5, 6);
System.out.println(arr.retainAll(arr2)); // true
System.out.println(arr.toString());      // [4, 5, 6, 6]
System.out.println(arr.retainAll(arr2)); // false
```

- `remove()` — удаляет один элемент из очереди. Форматы метода:

```
// Интерфейсы Collection<E>, Queue<E> и Deque<E>
public boolean remove(Object o)
// Интерфейсы Queue<E> и Deque<E>
public E remove()
```

Первый формат находит в очереди первый элемент, соответствующий указанному объекту, и удаляет его. Метод возвращает значение `true`, если элемент был удален, и `false` — в противном случае:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3, 2, 5, 2, 6);
System.out.println(arr.remove(2));  // true
System.out.println(arr.toString()); // [1, 3, 2, 5, 2, 6]
```

Второй формат удаляет первый элемент из очереди и возвращает его. Если очередь пуста, то генерируется исключение `NoSuchElementException`:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2);
System.out.println(arr.remove());      // 1
System.out.println(arr.toString());    // [2]
```

- `removeFirstOccurrence()` — находит в очереди первый элемент, соответствующий указанному объекту, и удаляет его. Метод возвращает значение `true`, если элемент был удален, и `false` — в противном случае. Формат метода:

```
// Интерфейс Deque<E>
public boolean removeFirstOccurrence(Object o)
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3, 2, 5, 2, 6);
System.out.println(
     arr.removeFirstOccurrence(2)); // true
System.out.println(arr.toString()); // [1, 3, 2, 5, 2, 6]
```

- `removeLastOccurrence()` — находит в очереди последний элемент, соответствующий указанному объекту, и удаляет его. Метод возвращает значение `true`, если элемент был удален, и `false` — в противном случае. Формат метода:

```
// Интерфейс Deque<E>
public boolean removeLastOccurrence(Object o)
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3, 2, 5, 2, 6);
System.out.println(
      arr.removeLastOccurrence(2)); // true
System.out.println(arr.toString()); // [1, 2, 3, 2, 5, 6]
```

14.7.5. Получение элементов из очереди

Получить элементы из очереди позволяют следующие методы:

- `getFirst()` — возвращает первый элемент из очереди, не удаляя его. Если очередь пуста, то генерируется исключение `NoSuchElementException`. Формат метода:

```
// Интерфейс Deque<E>
public E getFirst()
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2);
System.out.println(arr.getFirst());  // 1
System.out.println(arr.toString());  // [1, 2]
```

- `element()` — возвращает первый элемент из очереди, не удаляя его. Если очередь пуста, то генерируется исключение `NoSuchElementException`. Формат метода:

```
// Интерфейсы Queue<E> и Deque<E>
public E element()
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2);
System.out.println(arr.element());   // 1
System.out.println(arr.toString());  // [1, 2]
```

- `peekFirst()` — возвращает первый элемент из очереди, не удаляя его. Если очередь пуста, то метод возвращает значение `null`. Формат метода:

```
// Интерфейс Deque<E>
public E peekFirst()
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
System.out.println(arr.peekFirst()); // null
Collections.addAll(arr, 1, 2);
System.out.println(arr.peekFirst()); // 1
System.out.println(arr.toString());  // [1, 2]
```

- `peek()` — возвращает первый элемент из очереди, не удаляя его. Если очередь пуста, то метод возвращает значение `null`. Формат метода:

```
// Интерфейсы Queue<E> и Deque<E>
public E peek()
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
System.out.println(arr.peek());      // null
Collections.addAll(arr, 1, 2);
System.out.println(arr.peek());      // 1
System.out.println(arr.toString());  // [1, 2]
```

- `remove()` — удаляет первый элемент из очереди и возвращает его. Если очередь пуста, то генерируется исключение `NoSuchElementException`. Формат метода:

```
// Интерфейсы Queue<E> и Deque<E>
public E remove()
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2);
System.out.println(arr.remove());      // 1
System.out.println(arr.toString());    // [2]
```

- `pop()` — **удаляет первый элемент из очереди и возвращает его. Если очередь пуста, то генерируется исключение** `NoSuchElementException`. **Формат метода:**

```
// Интерфейс Deque<E>
public E pop()
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2);
System.out.println(arr.pop());          // 1
System.out.println(arr.toString());     // [2]
```

- `removeFirst()` — **удаляет первый элемент из очереди и возвращает его. Если очередь пуста, то генерируется исключение** `NoSuchElementException`. **Формат метода:**

```
// Интерфейс Deque<E>
public E removeFirst()
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2);
System.out.println(arr.removeFirst()); // 1
System.out.println(arr.toString());    // [2]
```

- `poll()` — **удаляет первый элемент из очереди и возвращает его. Если очередь пуста, то метод возвращает значение** `null`. **Формат метода:**

```
// Интерфейсы Queue<E> и Deque<E>
public E poll()
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
System.out.println(arr.poll());         // null
Collections.addAll(arr, 1, 2);
System.out.println(arr.poll());         // 1
System.out.println(arr.toString());     // [2]
```

- `pollFirst()` — **удаляет первый элемент из очереди и возвращает его. Если очередь пуста, то метод возвращает значение** `null`. **Формат метода:**

```
// Интерфейс Deque<E>
public E pollFirst()
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
System.out.println(arr.pollFirst());    // null
Collections.addAll(arr, 1, 2);
System.out.println(arr.pollFirst());    // 1
System.out.println(arr.toString());     // [2]
```

- `getLast()` — возвращает последний элемент из очереди, не удаляя его. Если очередь пуста, то генерируется исключение `NoSuchElementException`. Формат метода:

```
// Интерфейс Deque<E>
public E getLast()
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2);
System.out.println(arr.getLast());     // 2
System.out.println(arr.toString());    // [1, 2]
```

- `peekLast()` — возвращает последний элемент из очереди, не удаляя его. Если очередь пуста, то метод возвращает значение `null`. Формат метода:

```
// Интерфейс Deque<E>
public E peekLast()
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
System.out.println(arr.peekLast());    // null
Collections.addAll(arr, 1, 2);
System.out.println(arr.peekLast());    // 2
System.out.println(arr.toString());    // [1, 2]
```

- `removeLast()` — удаляет последний элемент и возвращает его. Если очередь пуста, то генерируется исключение `NoSuchElementException`. Формат метода:

```
// Интерфейс Deque<E>
public E removeLast()
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2);
System.out.println(arr.removeLast());  // 2
System.out.println(arr.toString());    // [1]
```

- `pollLast()` — удаляет последний элемент и возвращает его. Если очередь пуста, то метод возвращает значение `null`. Формат метода:

```
// Интерфейс Deque<E>
public E pollLast()
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
System.out.println(arr.pollLast());    // null
Collections.addAll(arr, 1, 2);
System.out.println(arr.pollLast());    // 2
System.out.println(arr.toString());    // [1]
```

14.7.6. Проверка существования элементов в очереди

Проверить наличие элементов в очереди позволяют следующие методы:

- `contains()` — возвращает значение `true`, если элемент существует в очереди, и `false` — в противном случае. Формат метода:

```
// Интерфейсы Collection<E>, Queue<E> и Deque<E>
public boolean contains(Object o)
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3, 2);
System.out.println(arr.contains(2)); // true
System.out.println(arr.contains(4)); // false
```

- `containsAll()` — возвращает значение `true`, если все элементы из указанной коллекции существуют в очереди, и `false` — в противном случае. Формат метода:

```
// Интерфейсы Collection<E>, Queue<E> и Deque<E>
public boolean containsAll(Collection<?> c)
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
ArrayDeque<Integer> arr2 = new ArrayDeque<Integer>();
Collections.addAll(arr2, 4, 3, 2);
System.out.println(arr.containsAll(arr2)); // true
arr2.add(8);
System.out.println(arr.containsAll(arr2)); // false
```

Статический метод `frequency()` из класса `Collections` позволяет посчитать количество элементов с указанным значением. Формат метода:

```
import java.util.Collections;
public static int frequency(Collection<?> c, Object o)
```

Пример:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 3, 2, 3, 2, 2);
System.out.println(Collections.frequency(arr, 2)); // 3
System.out.println(Collections.frequency(arr, 1)); // 1
System.out.println(Collections.frequency(arr, 8)); // 0
```

14.7.7. Поиск минимального и максимального значений в очереди

Для поиска минимального и максимального значений в очереди предназначены следующие статические методы из класса `Collections`:

- `min()` — выполняет поиск минимального значения и возвращает элемент. Форматы метода:

```
import java.util.Collections;
public static <T extends Object & Comparable<? super T>> T
       min(Collection<? extends T> coll)
import java.util.Comparator;
public static <T> T min(Collection<? extends T> coll,
                        Comparator<? super T> comp)
```

Первый формат выполняет поиск, используя для сравнения метод `compareTo()` из интерфейса `Comparable<T>`:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
Integer min = Collections.min(arr);
System.out.println(min); // 1
```

Второй формат позволяет во втором параметре указать экземпляр класса, реализующего интерфейс `Comparator<T>`:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
Integer min = Collections.min(arr, new Comparator<Integer>() {
   @Override
   public int compare(Integer x, Integer y) {
      return x.compareTo(y);
   }
});
System.out.println(min); // 1
```

- `max()` — выполняет поиск максимального значения и возвращает элемент. Форматы метода:

```
import java.util.Collections;
public static <T extends Object & Comparable<? super T>> T
       max(Collection<? extends T> coll)
import java.util.Comparator;
public static <T> T max(Collection<? extends T> coll,
                        Comparator<? super T> comp)
```

Первый формат выполняет поиск, используя для сравнения метод `compareTo()` из интерфейса `Comparable<T>`:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
Integer max = Collections.max(arr);
System.out.println(max); // 6
```

Второй формат позволяет во втором параметре указать экземпляр класса, реализующего интерфейс `Comparator<T>`:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5, 6);
Integer max = Collections.max(arr, new Comparator<Integer>() {
   @Override
```

```
    public int compare(Integer x, Integer y) {
       return x.compareTo(y);
    }
});
System.out.println(max); // 6
```

14.7.8. Преобразование массива в очередь и очереди в массив

Для преобразования обычного массива в очередь можно воспользоваться статическим методом `addAll()` из класса `Collections`. Формат метода:

```
import java.util.Collections;
public static <T> boolean addAll(Collection<? super T> c, T... elements)
```

В первом параметре указывается ссылка на коллекцию, а во втором параметре можно указать значения через запятую или обычный массив. Новые элементы будут добавлены в конец очереди. Метод возвращает значение `true`, если элементы были добавлены, и `false` — в противном случае:

```
Integer[] arrInt = {1, 2, 3};
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
arr.add(55);
Collections.addAll(arr, arrInt);     // Добавляем элементы из массива
System.out.println(arr.toString()); // [55, 1, 2, 3]
```

Статический метод `asList()` из класса `Arrays` позволяет «обернуть» список вокруг обычного массива. Такой «обернутый» массив мы можем передать конструктору класса `ArrayDeque<E>`:

```
Integer[] arrInt = {1, 2, 3};
ArrayDeque<Integer> arr =
   new ArrayDeque<Integer>(Arrays.asList(arrInt));
System.out.println(arr.toString());          // [1, 2, 3]
```

Преобразовать очередь в массив позволяет метод `toArray()`. Форматы метода:

```
// Интерфейсы Collection<E>, Queue<E> и Deque<E>
public Object[] toArray()
public <T> T[] toArray(T[] a)
```

Первый формат возвращает массив объектов класса `Object`:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3);
System.out.println(arr.toString());          // [1, 2, 3]
Object[] arrObj = arr.toArray();
System.out.println(Arrays.toString(arrObj)); // [1, 2, 3]
Integer x = (Integer) arrObj[0];
System.out.println(x);                       // 1
```

Второй формат позволяет в качестве параметра указать ссылку на массив, который будет заполнен элементами из очереди. Метод возвращает ссылку на заполненный массив:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3);
System.out.println(arr.toString());          // [1, 2, 3]
Integer[] arrInt = arr.toArray(new Integer[arr.size()]);
System.out.println(Arrays.toString(arrInt)); // [1, 2, 3]
```

В этом примере мы создаем новый массив, указывая количество элементов очереди, и передаем его методу `toArray()`. Метод заполнит массив значениями из очереди и вернет ссылку на него.

Если длина указанного массива будет меньше размера очереди, то метод создаст новый массив того же типа, заполнит его элементами из очереди и вернет ссылку на него. Массив, указанный в качестве параметра, изменен в этом случае не будет:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3);
System.out.println(arr.toString());          // [1, 2, 3]
Integer[] arrInt = arr.toArray(new Integer[0]);
System.out.println(Arrays.toString(arrInt)); // [1, 2, 3]
```

Если длина указанного массива больше размера очереди, то элементы будут скопированы в этот массив, начиная с нулевого индекса. После копирования всех элементов будет добавлен еще один элемент со значением `null`. В этом случае метод вернет ссылку на первоначальный массив:

```
Integer[] arrInt = {8, 8, 8, 8, 8, 8, 8};
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3);
System.out.println(arr.toString());          // [1, 2, 3]
arrInt = arr.toArray(arrInt);
System.out.println(Arrays.toString(arrInt));
// [1, 2, 3, null, 8, 8, 8]
```

14.7.9. Перебор элементов очереди

Перебрать все элементы очереди можно с помощью цикла `for each`, итератора и метода `forEach()` (начиная с Java 8). Пример использования цикла `for each`:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
for (Integer item: arr) {
   System.out.println(item);
}
```

Чтобы получить итератор, необходимо вызвать метод `iterator()`. Формат метода:

```
// Интерфейсы Iterable<T>, Collection<E>, Queue<E> и Deque<E>
public Iterator<E> iterator()
```

Пример перебора элементов очереди с помощью итератора и цикла `while`:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
Iterator<Integer> it = arr.iterator();
while (it.hasNext()) {
   System.out.println(it.next());
}
```

Метод `descendingIterator()` позволяет получить обратный итератор. Формат метода:

```
// Интерфейс Deque<E>
public Iterator<E> descendingIterator()
```

Пример перебора очереди в обратном порядке:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
Iterator<Integer> it = arr.descendingIterator();
while (it.hasNext()) {
   System.out.print(it.next() + " ");
} // 4 3 2 1
```

Начиная с Java 8, для перебора элементов используется также метод `forEach()`:

```
ArrayDeque<Integer> arr = new ArrayDeque<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
arr.forEach( elem -> System.out.println(elem) );
```

Если нужно не просто перебрать и посмотреть элементы, но и удалить их из очереди, то можно использовать цикл `while` в сочетании с методом `isEmpty()` и каким-либо методом, удаляющим и возвращающим элемент (в зависимости от типа очереди):

```
Deque<Integer> arr = new ArrayDeque<Integer>();
arr.push(1);            // Добавляем элементы в стек
arr.push(2);
arr.push(3);
arr.push(4);
while (!arr.isEmpty()) { // Обрабатываем элементы
   System.out.print(arr.pop() + " ");
} // 4 3 2 1
```

14.8. Класс *PriorityQueue<E>*: очередь с приоритетами

Класс `PriorityQueue<E>` реализует очередь с приоритетами. Из очереди первым возвращается элемент с наибольшим приоритетом. Значения `null` добавлять нельзя. Прежде чем использовать класс `PriorityQueue<E>`, необходимо импортировать его с помощью инструкции:

```
import java.util.PriorityQueue;
```

Класс `PriorityQueue<E>` реализует следующие интерфейсы:

```
Iterable<E>, Collection<E>, Queue<E>, Serializable
```

Создать очередь с приоритетами позволяют следующие конструкторы класса `PriorityQueue<E>`:

```
PriorityQueue()
PriorityQueue(int initialCapacity)
PriorityQueue(Collection<? extends E> c)
PriorityQueue(PriorityQueue<? extends E> c)
PriorityQueue(SortedSet<? extends E> c)
PriorityQueue(Comparator<? super E> comparator)
PriorityQueue(int initialCapacity, Comparator<? super E> comparator)
```

Первый конструктор позволяет создать очередь емкостью 11 элементов и определением приоритета по умолчанию (с помощью метода `compareTo()` из интерфейса `Comparable<T>`):

```
PriorityQueue<Integer> arr = new PriorityQueue<Integer>();
arr.add(1);           // Добавляем элементы в очередь
arr.add(2);
arr.add(3);
arr.add(4);
System.out.println(arr.size()); // 4
while (!arr.isEmpty()) { // Обрабатываем элементы
   System.out.print(arr.remove() + " ");
} // 1 2 3 4
```

Второй конструктор позволяет задать начальную емкость (определение приоритета по умолчанию):

```
PriorityQueue<Integer> arr = new PriorityQueue<Integer>(15);
arr.offer(1);           // Добавляем элементы в очередь
arr.offer(2);
arr.offer(1);
arr.offer(4);
while (!arr.isEmpty()) { // Обрабатываем элементы
   System.out.print(arr.poll() + " ");
} // 1 1 2 4
```

Третий, четвертый и пятый конструкторы создают очередь на основе другой коллекции:

```
ArrayList<Integer> arr2 = new ArrayList<Integer>();
Collections.addAll(arr2, 5, 1, 2, 3);
PriorityQueue<Integer> arr = new PriorityQueue<Integer>(arr2);
while (!arr.isEmpty()) {
   System.out.print(arr.poll() + " ");
} // 1 2 3 5
```

Шестой конструктор позволяет указать способ определения приоритета. Изменим приоритет на противоположный:

```
PriorityQueue<Integer> arr =
   new PriorityQueue<Integer>(new Comparator<Integer>() {
      @Override
      public int compare(Integer a, Integer b) {
         return b.compareTo(a);
      }
   });
Collections.addAll(arr, 5, 1, 2, 3);
while (!arr.isEmpty()) {
   System.out.print(arr.poll() + " ");
} // 5 3 2 1
```

Седьмой конструктор позволяет указать начальную емкость очереди и способ определения приоритета:

```
PriorityQueue<Integer> arr =
   new PriorityQueue<Integer>(20, Collections.reverseOrder());
Collections.addAll(arr, 5, 1, 2, 3);
while (!arr.isEmpty()) {
   System.out.print(arr.poll() + " ");
} // 5 3 2 1
```

Для добавления элементов используются методы `add()` и `offer()` из интерфейса `Queue<E>`, а для удаления и возвращения элемента — методы `remove()` и `poll()`. Метод `peek()` возвращает элемент с наибольшим приоритетом без удаления, а метод `size()` служит для получения количества элементов в очереди. Можно воспользоваться и другими методами из интерфейсов `Queue<E>` и `Collection<E>`. Все эти методы мы уже рассматривали ранее.

Перебрать все элементы очереди можно с помощью цикла `for each`, итератора и метода `forEach()` (начиная с Java 8). Следует учитывать, что элементы хранятся без сортировки по приоритету, поэтому последовательность перебора может быть произвольной. Пример использования цикла `for each`:

```
PriorityQueue<Integer> arr = new PriorityQueue<Integer>();
Collections.addAll(arr, 1, 2, 3, 2, 4);
for (Integer item: arr) {
   System.out.println(item);
}
```

Чтобы получить итератор, необходимо вызвать метод `iterator()`. Пример перебора элементов очереди с помощью итератора и цикла `while`:

```
PriorityQueue<Integer> arr = new PriorityQueue<Integer>();
Collections.addAll(arr, 1, 2, 3, 2, 4);
Iterator<Integer> it = arr.iterator();
while (it.hasNext()) {
   System.out.println(it.next());
}
```

Начиная с Java 8, для перебора элементов используется также метод `forEach()`. Пример:

```
PriorityQueue<Integer> arr = new PriorityQueue<Integer>();
Collections.addAll(arr, 1, 2, 3, 2, 4);
arr.forEach( elem -> System.out.println(elem) );
```

Если необходимо получить список из значений очереди в отсортированном состоянии в соответствии с порядком определения приоритета, то методу `sort()` можно передать результат выполнения метода `comparator()`. Формат метода:

```
public Comparator<? super E> comparator()
```

Если используется определение приоритета по умолчанию, то метод вернет значение `null`, в противном случае — способ сравнения:

```
PriorityQueue<Integer> arr =
   new PriorityQueue<Integer>(Collections.reverseOrder());
Collections.addAll(arr, 1, 2, 3, 2, 4, 1);
System.out.println(arr.toString()); // [4, 3, 2, 1, 2, 1]
ArrayList<Integer> arr2 = new ArrayList<Integer>(arr);
arr2.sort(arr.comparator());
System.out.println(arr2.toString()); // [4, 3, 2, 2, 1, 1]
```

14.9. Класс *LinkedList<E>*: связанный список и очередь

Класс `LinkedList<E>` реализует связанный список, элементы которого хранят ссылки на предыдущий и последующий элементы. Благодаря этому вставка и удаление элементов происходит быстро — достаточно изменить несколько ссылок, а не перемещать элементы, как это делает динамический список. Но доступ к элементам по индексам осуществляется медленно, т. к. приходится каждый раз искать элемент с начала или конца списка, ведь связанный список не содержит индексов. С помощью класса `LinkedList<E>` можно также реализовать очередь. Этот класс позволяет хранить элементы со значением `null`. Прежде чем использовать класс `LinkedList<E>`, необходимо импортировать его с помощью инструкции:

```
import java.util.LinkedList;
```

Класс `LinkedList<E>` реализует следующие интерфейсы:

```
Iterable<E>, Collection<E>, List<E>, Queue<E>, Deque<E>, Serializable, Cloneable
```

Создать связанный список позволяют следующие конструкторы класса `LinkedList<E>`:

```
LinkedList()
LinkedList(Collection<? extends E> c)
```

Первый конструктор создает пустой связанный список:

```
LinkedList<Integer> arr = new LinkedList<Integer>();
System.out.println(arr.size());     // 0
Collections.addAll(arr, 1, 2, 3);
System.out.println(arr.toString()); // [1, 2, 3]
```

Второй конструктор позволяет создать связанный список на основе другой коллекции. Заполним список десятью нулями:

```
LinkedList<Integer> arr =
   new LinkedList<Integer>(Collections.nCopies(10, 0));
System.out.println(arr.size());     // 10
System.out.println(arr.toString());
// [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Обратите внимание: в связанном списке нет такого понятия, как емкость. Поэтому в классе `LinkedList<E>` нет методов `ensureCapacity()` и `trimToSize()`, как в классе `ArrayList<E>`. Связанный список не является массивом, и элементы могут быть расположены как угодно. Каждый элемент содержит ссылку на предыдущий и последующий элементы. При добавлении элементов не производится никакое копирование, просто изменяются значения нескольких ссылок. При удалении так же просто изменяются несколько ссылок. Поэтому добавление и удаление элементов выполняется быстро. А вот доступ по индексу выполняется медленно. Каждый раз, когда вы указываете индекс, производится поиск элемента с начала или с конца списка, а не используется прямой доступ, как в классе `ArrayList<E>`.

Класс `LinkedList<E>` реализует сразу три интерфейса: `List<E>`, `Queue<E>` и `Deque<E>`, поэтому содержит все методы из этих интерфейсов, а также из интерфейса `Collection<E>`. При изучении классов `ArrayList<E>` (интерфейс `List<E>`) и `ArrayDeque<E>` (интерфейсы `Queue<E>` и `Deque<E>`) мы уже рассмотрели эти методы, поэтому не будем повторяться — просто откройте предыдущие разделы и в примерах подставьте класс `LinkedList<E>` вместо классов `ArrayList<E>` и `ArrayDeque<E>`.

При использовании метода `remove()` следует учитывать, что простое указание числа в качестве параметра означает индекс удаляемого элемента. Чтобы удалить именно число, необходимо указать объект класса `Integer`:

```
LinkedList<Integer> arr = new LinkedList<Integer>();
Collections.addAll(arr, 1, 5, 3, 2);
// Удаляем элемент с индексом 2
System.out.println(arr.remove(2));                  // 3
System.out.println(arr.toString());                 // [1, 5, 2]
// Удаляем элемент со значением 2
System.out.println(arr.remove(Integer.valueOf(2))); // true
System.out.println(arr.toString());                 // [1, 5]
```

Пример использования класса `LinkedList<E>` в качестве списка приведен в листинге 14.3.

Листинг 14.3. Использование класса LinkedList<E> в качестве списка

```
import java.util.*;

public class MyClass {
   public static void main(String[] args) {
      List<Integer> arr = new LinkedList<Integer>();
      arr.add(20);                          // В конец списка
      arr.add(0, 10);                       // В начало списка
      arr.add(arr.size(), 30);              // В конец списка
      System.out.println(arr.toString()); // [10, 20, 30]
      // Доступ по индексу
      System.out.println(arr.get(1));      // 20
      arr.set(1, 88);
      System.out.println(arr.toString()); // [10, 88, 30]
      // Сортировка
      arr.sort(null);
      System.out.println(arr.toString()); // [10, 30, 88]
      arr.sort(Collections.reverseOrder());
      System.out.println(arr.toString()); // [88, 30, 10]
      // Перебор элементов
      for (Integer item: arr) {
         System.out.print(item + " ");
      } // 88 30 10
      System.out.println();
      arr.forEach( elem -> System.out.print(elem + " ") );
      // 88 30 10
      System.out.println();
      Iterator<Integer> it = arr.iterator();
      while (it.hasNext()) {
         System.out.print(it.next() + " ");
      } // 88 30 10
      System.out.println();
      ListIterator<Integer> it2 = arr.listIterator();
      while (it2.hasNext()) {
         System.out.print(it2.next() + " ");
      } // 88 30 10
      System.out.println();
      // В обратном порядке
      ListIterator<Integer> it3 = arr.listIterator(arr.size());
      while (it3.hasPrevious()) {
         System.out.print(it3.previous() + " ");
      } // 10 30 88
      System.out.println();
      // Изменение всех элементов
      int x = 0;
```

```
      for (int i = 0, j = arr.size(); i < j; i++) {
         x = arr.get(i);
         x *= 2;
         arr.set(i, x);
      }
      System.out.println(arr.toString()); // [176, 60, 20]
   }
}
```

Примеры реализации очереди «первым пришел, первым ушел» (обычная очередь, например, за билетами в кинотеатр, — чем раньше встали в очередь, тем быстрее купите билет) и стека (очереди, в которой пришедший последним уходит первым) приведены в листинге 14.4.

Листинг 14.4. Использование класса `LinkedList<E>` в качестве очереди и стека

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Deque;

public class MyClass {
   public static void main(String[] args) {
      // Очередь (первым пришел, первым ушел)
      Queue<Integer> arr = new LinkedList<Integer>();
      arr.offer(1);           // Добавляем элементы в очередь
      arr.offer(2);
      arr.offer(3);
      arr.offer(4);
      while (!arr.isEmpty()) { // Обрабатываем элементы
         System.out.print(arr.poll() + " ");
      } // 1 2 3 4
      System.out.println();
      Queue<Integer> arr2 = new LinkedList<Integer>();
      arr2.add(1);             // Добавляем элементы в очередь
      arr2.add(2);
      arr2.add(3);
      arr2.add(4);
      while (!arr2.isEmpty()) { // Обрабатываем элементы
         System.out.print(arr2.remove() + " ");
      } // 1 2 3 4
      System.out.println();
      // Стек (последний уходит первым)
      Deque<Integer> arr3 = new LinkedList<Integer>();
      arr3.push(1);            // Добавляем элементы в стек
      arr3.push(2);
      arr3.push(3);
      arr3.push(4);
```

```
        while (!arr3.isEmpty()) { // Обрабатываем элементы
            System.out.print(arr3.pop() + " ");
        } // 4 3 2 1
    }
}
```

14.10. Класс *Vector<E>*: синхронизированный динамический список

Класс `Vector<E>` реализует динамический упорядоченный список произвольных элементов, как в обычном массиве. Добавить элементы можно в любое место списка, при этом размер списка будет автоматически увеличиваться по мере необходимости (в отличие от обычных массивов). Получить доступ к элементу списка можно по индексу или с помощью итератора. При удалении элемента производится сдвиг всех последующих элементов на одну позицию влево. При этом емкость списка не изменяется. В отличие от класса `ArrayList<E>`, класс `Vector<E>` является синхронизированным и может быть использован для доступа из разных потоков. Из-за потерь на синхронизацию класс `Vector<E>` работает медленнее, чем класс `ArrayList<E>`. Прежде чем использовать класс `Vector<E>`, необходимо импортировать его с помощью инструкции:

```
import java.util.Vector;
```

Класс `Vector<E>` реализует следующие интерфейсы:

```
Iterable<E>, Collection<E>, List<E>, RandomAccess, Serializable, Cloneable
```

14.10.1. Создание объекта

Создать список позволяют следующие конструкторы класса `Vector<E>`:

```
Vector()
Vector(int initialCapacity)
Vector(int initialCapacity, int capacityIncrement)
Vector(Collection<? extends E> c)
```

Первый конструктор создает пустой список с емкостью в десять элементов:

```
Vector<Integer> arr = new Vector<Integer>();
System.out.println(arr.size());     // 0
arr.add(10);
arr.add(33);
System.out.println(arr.get(0));     // 10
System.out.println(arr.get(1));     // 33
System.out.println(arr.toString()); // [10, 33]
```

Второй конструктор позволяет указать начальную емкость списка:

```
Vector<Integer> arr = new Vector<Integer>(100);
System.out.println(arr.size());     // 0
```

```
for (int i = 1; i < 101; i++) {
   arr.add(i);
}
System.out.println(arr.get(0));      // 1
System.out.println(arr.get(99));     // 100
System.out.println(arr.size());      // 100
```

Третий конструктор позволяет указать начальную емкость списка и величину приращения, а четвертый — создает список на основе другой коллекции. Заполним список десятью нулями:

```
Vector<Integer> arr = new Vector<Integer>(Collections.nCopies(10, 0));
System.out.println(arr.toString());
// [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

14.10.2. Методы класса *Vector<E>*

Класс `Vector<E>` реализует интерфейс `List<E>`, поэтому содержит все методы из этого интерфейса, а также из интерфейса `Collection<E>`. При изучении класса `ArrayList<E>` мы уже рассмотрели эти методы, поэтому не будем повторяться — просто откройте предыдущие разделы и в примерах подставьте класс `Vector<E>` вместо класса `ArrayList<E>`. Например, если в листинге 14.3 инструкцию:

```
List<Integer> arr = new LinkedList<Integer>();
```

заменить инструкцией:

```
List<Integer> arr = new Vector<Integer>();
```

то все будет работать точно так же. Если не хотите зависеть от реализаций, объявляйте переменную с типом интерфейса. При необходимости выбрать другую реализацию, достаточно будет изменить одну строку в программе.

Класс `Vector<E>` был реализован задолго до создания каркаса коллекций, и его доработали, чтобы вписаться в этот каркас, поэтому, в отличие от других коллекций, он является синхронизированным по умолчанию и содержит некоторые дополнительные методы:

- `setSize()` — задает размер списка. Формат метода:

  ```
  public void setSize(int newSize)
  ```

  Если текущий размер списка меньше указанного значения, то новые элементы получат значение `null`, а если больше — лишние элементы будут удалены:

  ```
  Vector<Integer> arr = new Vector<Integer>();
  System.out.println(arr.size()); // 0
  arr.setSize(5);
  System.out.println(arr.size()); // 5
  System.out.println(arr.toString());
  // [null, null, null, null, null]
  arr.setSize(2);
  ```

```
System.out.println(arr.size()); // 2
System.out.println(arr.toString());
// [null, null]
```

- `ensureCapacity()` — задает рекомендуемую емкость списка. Если указанное значение емкости меньше количества элементов списка, то список изменен не будет. Если больше, то емкость будет увеличена. Формат метода:

```
public void ensureCapacity(int minCapacity)
```

- `capacity()` — возвращает текущую емкость списка. Формат метода:

```
public int capacity()
```

Пример:

```
Vector<Integer> arr = new Vector<Integer>();
System.out.println(arr.capacity()); // 10
arr.ensureCapacity(20);
System.out.println(arr.capacity()); // 20
```

- `trimToSize()` — уменьшает емкость списка до размера списка. Формат метода:

```
public void trimToSize()
```

Пример:

```
Vector<Integer> arr = new Vector<Integer>();
arr.setSize(11);
arr.ensureCapacity(40);
System.out.println(arr.capacity()); // 40
arr.trimToSize();
System.out.println(arr.capacity()); // 11
```

- `addElement()` — добавляет один элемент в конец списка. Формат метода:

```
public void addElement(E obj)
```

Пример:

```
Vector<Integer> arr = new Vector<Integer>();
arr.addElement(10);
arr.addElement(20);
System.out.println(arr.toString()); // [10, 20]
```

- `insertElementAt()` — вставляет элемент в указанную позицию. Формат метода:

```
public void insertElementAt(E obj, int index)
```

Пример:

```
Vector<Integer> arr = new Vector<Integer>();
arr.addElement(10);
arr.addElement(20);
arr.insertElementAt(5, 0);            // В начало
arr.insertElementAt(25, arr.size()); // В конец
System.out.println(arr.toString());  // [5, 10, 20, 25]
```

- `firstElement()` — возвращает первый элемент списка. Формат метода:

```
public E firstElement()
```

- `lastElement()` — возвращает последний элемент списка. Формат метода:

```
public E lastElement()
```

- `elementAt()` — возвращает элемент с указанным индексом. Формат метода:

```
public E elementAt(int index)
```

Пример:

```
Vector<Integer> arr = new Vector<Integer>();
arr.addElement(10);
arr.addElement(20);
arr.addElement(30);
System.out.println(arr.firstElement()); // 10
System.out.println(arr.lastElement());  // 30
System.out.println(arr.elementAt(1));   // 20
```

- `setElementAt()` — изменяет значение элемента с указанным индексом. Формат метода:

```
public void setElementAt(E obj, int index)
```

Пример:

```
Vector<Integer> arr = new Vector<Integer>();
arr.addElement(10);
arr.addElement(20);
arr.setElementAt(30, 0);
System.out.println(arr.toString()); // [30, 20]
```

- `removeElementAt()` — удаляет элемент с указанным индексом. Формат метода:

```
public void removeElementAt(int index)
```

Пример:

```
Vector<Integer> arr = new Vector<Integer>();
arr.addElement(10);
arr.addElement(20);
arr.removeElementAt(0);
System.out.println(arr.toString()); // [20]
```

- `removeElement()` — удаляет первый элемент с указанным значением. Формат метода:

```
public boolean removeElement(Object obj)
```

Пример:

```
Vector<Integer> arr = new Vector<Integer>();
arr.addElement(10);
arr.addElement(20);
```

```
arr.addElement(10);
arr.removeElement(10);
System.out.println(arr.toString()); // [20, 10]
```

- `removeAllElements()` — удаляет все элементы из списка. Формат метода:

```
public void removeAllElements()
```

Пример:

```
Vector<Integer> arr = new Vector<Integer>();
arr.addElement(10);
arr.removeAllElements();
System.out.println(arr.toString()); // []
```

- `indexOf()` — находит первый элемент, соответствующий указанному объекту, и возвращает его индекс или значение –1, означающее, что элемент не найден. Во втором параметре указывается начальный индекс. Формат метода:

```
public int indexOf(Object o, int index)
```

Пример:

```
Vector<Integer> arr = new Vector<Integer>();
arr.addElement(10);
arr.addElement(20);
arr.addElement(10);
System.out.println(arr.indexOf(10, 0)); // 0
System.out.println(arr.indexOf(10, 1)); // 2
```

- `lastIndexOf()` — находит последний элемент, соответствующий указанному объекту, и возвращает его индекс или значение –1, означающее, что элемент не найден. Во втором параметре указывается начальный индекс. Формат метода:

```
public int lastIndexOf(Object o, int index)
```

Пример:

```
Vector<Integer> arr = new Vector<Integer>();
arr.addElement(10);
arr.addElement(20);
arr.addElement(10);
System.out.println(
      arr.lastIndexOf(10, arr.size() - 1)); // 2
System.out.println(arr.lastIndexOf(10, 1)); // 0
```

14.10.3. Интерфейс *Enumeration<E>*

Интерфейс `Enumeration<E>` содержит объявления методов, позволяющих осуществить перебор элементов. Прежде чем использовать интерфейс `Enumeration<E>`, необходимо импортировать его с помощью инструкции:

```
import java.util.Enumeration;
```

Интерфейс `Enumeration<E>` содержит объявление следующих методов:

- `hasMoreElements()` — метод возвращает значение `true`, если можно получить следующий элемент, и `false` — в противном случае. Формат метода:

  ```
  public boolean hasMoreElements()
  ```

- `nextElement()` — перемещает указатель на одну позицию и возвращает элемент слева от указателя. Если был достигнут конец коллекции, метод генерирует исключение `NoSuchElementException`, поэтому, прежде чем вызвать этот метод, следует предварительно выполнить проверку с помощью метода `hasMoreElements()`. Формат метода:

  ```
  public E nextElement()
  ```

Получить итератор позволяет метод `elements()` из класса `Vector<E>`. Формат метода:

```
public Enumeration<E> elements()
```

Пример перебора элементов списка:

```
Vector<Integer> arr = new Vector<Integer>();
arr.addElement(10);
arr.addElement(20);
Enumeration<Integer> it = arr.elements();
while (it.hasMoreElements()) {
   System.out.print(it.nextElement() + " ");
} // 10 20
```

Интерфейс `Enumeration<E>` применялся до появления каркаса коллекций, однако до сих пор встречается в ранее созданном коде. Если вы работаете с интерфейсом `List<E>`, то лучше использовать итераторы из интерфейсов `Iterator<E>` и `ListIterator<E>`:

```
Vector<Integer> arr = new Vector<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
Iterator<Integer> it = arr.iterator();
while (it.hasNext()) {
   System.out.print(it.next() + " ");
} // 1 2 3 4
System.out.println();
ListIterator<Integer> it2 = arr.listIterator();
while (it2.hasNext()) {
   System.out.print(it2.next() + " ");
} // 1 2 3 4
System.out.println();
it2 = arr.listIterator(arr.size());
while (it2.hasPrevious()) {
   System.out.print(it2.previous() + " ");
} // 4 3 2 1
```

Если ожидается объект, реализующий интерфейс `Enumeration<E>`, но коллекция не реализует этот интерфейс, то можно воспользоваться статическим методом `enumeration()` из класса `Collections`. Формат метода:

```
import java.util.Collections;
public static <T> Enumeration<T> enumeration(Collection<T> c)
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
Enumeration<Integer> it = Collections.enumeration(arr);
while (it.hasMoreElements()) {
   System.out.print(it.nextElement() + " ");
} // 1 2 3 4
```

С помощью статического метода `list()` из класса `Collections` можно преобразовать объект, реализующий интерфейс `Enumeration<E>`, в объект класса `ArrayList<E>`. Формат метода:

```
import java.util.Collections;
public static <E> ArrayList<E> list(Enumeration<E> e)
```

Пример:

```
Vector<Integer> arr = new Vector<Integer>();
Collections.addAll(arr, 1, 2, 3, 4);
ArrayList<Integer> arr2 = Collections.list(arr.elements());
System.out.println(arr2.toString());  // [1, 2, 3, 4]
```

14.11. Класс *Stack<E>*: стек

Класс `Stack<E>` наследует класс `Vector<E>` и добавляет функционал стека (очереди, в которой пришедший последним уходит первым). Прежде чем использовать класс `Stack<E>`, необходимо импортировать его с помощью инструкции:

```
import java.util.Stack;
```

Класс `Stack<E>` реализует следующие интерфейсы:

```
Iterable<E>, Collection<E>, List<E>, RandomAccess, Serializable, Cloneable
```

Создать экземпляр класса `Stack<E>` без элементов позволяет следующий конструктор:

```
Stack()
```

Класс `Stack<E>` наследует все методы из класса `Vector<E>` и добавляет несколько методов:

- `push()` — добавляет элемент в конец очереди. Формат метода:

  ```
  public E push(E item)
  ```

Пример:

```
Stack<Integer> arr = new Stack<Integer>();
arr.push(1);
arr.push(2);
arr.push(3);
System.out.println(arr.toString());  // [1, 2, 3]
```

- `peek()` — **возвращает последний элемент из очереди, не удаляя его. Если очередь пуста, то генерируется исключение** `EmptyStackException`. **Формат метода:**

```
public E peek()
```

Пример:

```
Stack<Integer> arr = new Stack<Integer>();
arr.push(1);
arr.push(2);
arr.push(3);
System.out.println(arr.peek());      // 3
System.out.println(arr.toString());  // [1, 2, 3]
```

- `pop()` — **удаляет последний элемент из очереди и возвращает его. Если очередь пуста, то генерируется исключение** `EmptyStackException`. **Формат метода:**

```
public E pop()
```

Пример:

```
Stack<Integer> arr = new Stack<Integer>();
arr.push(1);
arr.push(2);
arr.push(3);
System.out.println(arr.pop());       // 3
System.out.println(arr.toString());  // [1, 2]
```

- `empty()` — **возвращает значение** `true`, **если очередь пуста, и** `false` — **в противном случае. Формат метода:**

```
public boolean empty()
```

- `search()` — **возвращает номер позиции элемента с конца очереди, соответствующего указанному объекту. Обратите внимание: не индекс элемента, а позицию с конца очереди. Формат метода:**

```
public int search(Object o)
```

Пример:

```
Stack<Integer> arr = new Stack<Integer>();
arr.push(1);
arr.push(2);
arr.push(1);
arr.push(3);
System.out.println(arr.search(1)); // 2
```

```
arr.push(1);
System.out.println(arr.search(1)); // 1
```

Добавим элементы в стек, а затем получим их в цикле:

```
Stack<Integer> arr = new Stack<Integer>();
arr.push(1);            // Добавляем элементы в стек
arr.push(2);
arr.push(3);
arr.push(4);
while (!arr.empty()) { // Обрабатываем элементы
   System.out.print(arr.pop() + " ");
} // 4 3 2 1
```

14.12. Класс *BitSet*: набор битов

Класс `BitSet` не входит в каркас коллекций, но можно сказать, что он описывает список логических значений (набора битов — например, флагов). Прежде чем использовать класс `BitSet`, необходимо импортировать его с помощью инструкции:

```
import java.util.BitSet;
```

Создать объект позволяют следующие конструкторы класса `BitSet`:

```
BitSet()
BitSet(int nbits)
```

Первый конструктор создает объект с настройками по умолчанию, а второй — позволяет указать начальное количество битов (биты в наборе нумеруются с `0`, как и элементы массива):

```
BitSet bset = new BitSet();
bset.set(0);
bset.set(1);
bset.set(9);
System.out.println(bset.length());   // 10
System.out.println(bset.toString()); // {0, 1, 9}
```

Создать объект на основе массива позволяет также статический метод `valueOf()` из класса `BitSet`. Форматы метода:

```
public static BitSet valueOf(byte[] bytes)
public static BitSet valueOf(long[] longs)
```

Пример:

```
BitSet bset = BitSet.valueOf(new byte[] {0b1011001});
System.out.println(bset.toString()); // {0, 3, 4, 6}
```

Класс `BitSet` содержит следующие методы:

- `length()` — возвращает «логическую длину» набора (индекс последнего установленного флага + 1). Формат метода:

  ```
  public int length()
  ```

- `size()` — возвращает количество битов, зарезервированных под набор. Формат метода:

```
public int size()
```

- `isEmpty()` — возвращает значение `true`, если ни один флаг в наборе не установлен, и `false` — в противном случае. Формат метода:

```
public boolean isEmpty()
```

- `cardinality()` — возвращает количество установленных флагов. Формат метода:

```
public int cardinality()
```

Пример:

```
BitSet bset = new BitSet();
bset.set(0, 5);
System.out.println(bset.cardinality()); // 5
```

- `set()` — устанавливает или сбрасывает флаги. Форматы метода:

```
public void set(int bitIndex)
public void set(int bitIndex, boolean value)
public void set(int fromIndex, int toIndex)
public void set(int fromIndex, int toIndex, boolean value)
```

Первый формат устанавливает флаг с указанным индексом. Второй формат позволяет установить (`value = true`) флаг или сбросить (`value = false`) его. Третий формат устанавливает диапазон флагов от индекса `fromIndex` до `toIndex` (не включая битов с этим индексом). Четвертый формат позволяет установить (`value = true`) флаги или сбросить (`value = false`) их для диапазона:

```
BitSet bset = new BitSet();
bset.set(0);
bset.set(1, true);
bset.set(3, 5);
bset.set(7, 9, true);
bset.set(7, false);
System.out.println(bset.length());   // 9
System.out.println(bset.size());     // 64
System.out.println(bset.toString()); // {0, 1, 3, 4, 8}
```

- `clear()` — сбрасывает флаги. Форматы метода:

```
public void clear()
public void clear(int bitIndex)
public void clear(int fromIndex, int toIndex)
```

Первый формат сбрасывает все флаги, второй — флаг с указанным индексом, а третий — диапазон от индекса `fromIndex` до `toIndex` (не включая битов с этим индексом):

```
BitSet bset = new BitSet();
bset.set(0, 9);
bset.clear(0);
```

```
bset.clear(3, 8);
System.out.println(bset.toString()); // {1, 2, 8}
```

- `flip()` — инвертирует значение флага. Форматы метода:

```
public void flip(int bitIndex)
public void flip(int fromIndex, int toIndex)
```

Первый формат инвертирует значение флага с указанным индексом, а второй — диапазон флагов от индекса `fromIndex` до `toIndex` (не включая битов с этим индексом):

```
BitSet bset = new BitSet();
bset.set(0, 9);
bset.flip(0);
bset.flip(3, 8);
System.out.println(bset.toString()); // {1, 2, 8}
```

- `get()` — позволяет получить значения флагов. Форматы метода:

```
public boolean get(int bitIndex)
public BitSet get(int fromIndex, int toIndex)
```

Первый формат возвращает значение флага с указанным индексом, а второй — возвращает диапазон значений от индекса `fromIndex` до `toIndex` (не включая битов с этим индексом):

```
BitSet bset = new BitSet();
bset.set(2, 5);
System.out.println(bset.get(0));    // false
System.out.println(bset.get(2));    // true
System.out.println(bset.get(0, 4)); // {2, 3}
```

- `and()` — двоичное И. Формат метода:

```
public void and(BitSet set)
```

Пример:

```
BitSet bset  = BitSet.valueOf(new byte[] {0b1100100});
BitSet bset2 = BitSet.valueOf(new byte[] {0b1001011});
bset.and(bset2);
System.out.printf("%8s\n",
   Integer.toBinaryString(
               bset.toByteArray()[0])); //  1000000
```

- `or()` — двоичное ИЛИ. Формат метода:

```
public void or(BitSet set)
```

Пример:

```
BitSet bset  = BitSet.valueOf(new byte[] {0b1100100});
BitSet bset2 = BitSet.valueOf(new byte[] {0b1001011});
bset.or(bset2);
System.out.printf("%8s\n",
   Integer.toBinaryString(
               bset.toByteArray()[0])); //  1101111
```

- `xor()` — **двоичное исключающее ИЛИ. Формат метода:**

```
public void xor(BitSet set)
```

Пример:

```
BitSet bset  = BitSet.valueOf(new byte[] {0b1100100});
BitSet bset2 = BitSet.valueOf(new byte[] {0b0111010});
bset.xor(bset2);
System.out.printf("%8s\n",
   Integer.toBinaryString(
               bset.toByteArray()[0])); //  1011110
```

- `andNot()` — **сбрасывает все флаги, установленные в указанном наборе. Формат метода:**

```
public void andNot(BitSet set)
```

Пример:

```
BitSet bset  = BitSet.valueOf(new byte[] {0b1111111});
BitSet bset2 = BitSet.valueOf(new byte[] {0b0111010});
bset.andNot(bset2);
System.out.printf("%8s\n",
   Integer.toBinaryString(
               bset.toByteArray()[0])); //  1000101
```

- `intersects()` — **возвращает значение** `true`**, если в указанном наборе есть хотя бы один установленный флаг на той же позиции, что и в текущем наборе, и** `false` **— в противном случае. Формат метода:**

```
public boolean intersects(BitSet set)
```

Пример:

```
BitSet bset  = BitSet.valueOf(new byte[] {0b1000101});
BitSet bset2 = BitSet.valueOf(new byte[] {0b0111010});
System.out.println(bset.intersects(bset2)); // false
BitSet bset3 = BitSet.valueOf(new byte[] {0b0111011});
System.out.println(bset.intersects(bset3)); // true
```

- `toByteArray()` **и** `toLongArray()` — **возвращают массивы со значениями из набора. Форматы методов:**

```
public byte[] toByteArray()
public long[] toLongArray()
```

Пример:

```
BitSet bset = BitSet.valueOf(new byte[] {0b1000101});
System.out.println(Arrays.toString(bset.toByteArray()));
// [69]
System.out.println(Arrays.toString(bset.toLongArray()));
// [69]
System.out.printf("%8s\n", Integer.toBinaryString(69));
//  1000101
```

ГЛАВА 15

Коллекции. Множества и словари

В этой главе мы продолжим изучение каркаса коллекций и рассмотрим две структуры: множества и словари. *Множество* — это набор уникальных элементов, с которым можно сравнивать другие элементы, чтобы определить, принадлежат ли они этому множеству. *Словари* (или *ассоциативные массивы*) — это наборы элементов, доступ к которым осуществляется не по индексу, а по уникальному ключу. Обратите внимание: словари не реализуют интерфейсы `Iterable<T>` и `Collection<E>`, они являются самостоятельными структурами.

15.1. Интерфейс *Set<E>*

Интерфейс `Set<E>` описывает набор (множество), состоящий из уникальных элементов. Для быстрого поиска элементов используется *хеш-таблица*. Значения этой таблицы состоят из целых чисел (тип `int`), которые можно получить путем вызова метода `hashCode()` из класса `Object`. В качестве исходных данных для формирования хеш-кода используется адрес объекта в памяти компьютера, поэтому объекты одного класса с одинаковыми значениями полей получат разные хеш-коды. Если вы хотите этого избежать, то внутри вашего класса следует переопределить методы `equals()` и `hashCode()`. Как это сделать, мы уже рассматривали ранее: и при изучении объектно-ориентированного программирования (см. *разд. 11.18*), и при изучении коллекций в предыдущей главе (см. листинг 14.2).

> ***ОБРАТИТЕ ВНИМАНИЕ!***
>
> **При использовании изменяемых объектов нельзя изменять их свойства (от которых зависит хеш-код) после добавления во множество. Ведь после изменения хеш-кода объект будет расположен внутри хеш-таблицы не на своем месте и может быть потерян.**

Прежде чем использовать интерфейс `Set<E>`, необходимо выполнить его импорт с помощью инструкции:

```
import java.util.Set;
```

Иерархия наследования:

```
Iterable<T> - Collection<E> - Set<E>
```

Интерфейс `Set<E>` реализуют следующие классы:

- `HashSet<E>` — множество, в котором уникальные элементы расположены в произвольном порядке;
- `LinkedHashSet<E>` — множество, в котором запоминается порядок вставки элементов;
- `TreeSet<E>` — набор уникальных элементов, хранимых в отсортированном порядке в соответствии с алгоритмом «красно-черное дерево». Хеш-таблица не используется.

Рассмотрим основные операции с множествами:

- объединение множеств (метод `addAll()`):

```
Set<Integer> set1 = new HashSet<Integer>();
Collections.addAll(set1, 1, 2, 3);
Set<Integer> set2 = new HashSet<Integer>();
Collections.addAll(set2, 3, 4, 5);
Set<Integer> set3 = new HashSet<Integer>(set1);
set3.addAll(set2);
System.out.println(set3.toString()); // [1, 2, 3, 4, 5]
```

- разница множеств (метод `removeAll()`):

```
Set<Integer> set1 = new HashSet<Integer>();
Collections.addAll(set1, 1, 2, 3);
Set<Integer> set2 = new HashSet<Integer>();
Collections.addAll(set2, 1, 2, 4);
Set<Integer> set3 = new HashSet<Integer>(set1);
set3.removeAll(set2);
System.out.println(set3.toString()); // [3]
```

- пересечение множеств (метод `retainAll()`):

```
Set<Integer> set1 = new HashSet<Integer>();
Collections.addAll(set1, 1, 2, 3);
Set<Integer> set2 = new HashSet<Integer>();
Collections.addAll(set2, 1, 2, 4);
Set<Integer> set3 = new HashSet<Integer>(set1);
set3.retainAll(set2);
System.out.println(set3.toString()); // [1, 2]
```

15.2. Класс *HashSet<E>*: множество, в котором уникальные элементы расположены в произвольном порядке

Класс `HashSet<E>` реализует множество, в котором уникальные элементы расположены в произвольном порядке. Прежде чем использовать класс `HashSet<E>`, необходимо импортировать его с помощью инструкции:

```
import java.util.HashSet;
```

Класс `HashSet<E>` реализует следующие интерфейсы:

```
Iterable<E>, Collection<E>, Set<E>, Serializable, Cloneable
```

15.2.1. Создание объекта

Создать множество позволяют следующие конструкторы класса `HashSet<E>`:

```
HashSet()
HashSet(int initialCapacity)
HashSet(int initialCapacity, float loadFactor)
HashSet(Collection<? extends E> c)
```

Первый конструктор создает пустое множество с емкостью в 16 элементов и коэффициентом заполнения `0.75`:

```
Set<Integer> set = new HashSet<Integer>();
System.out.println(set.size());     // 0
set.add(10);
set.add(33);
System.out.println(set.toString()); // [33, 10]
```

Второй конструктор позволяет указать начальную емкость множества, а третий — дополнительно — коэффициент заполнения (вещественное число от `0.0` до `1.0`), при достижении которого произойдет расширение множества:

```
Set<Integer> set = new HashSet<Integer>(16, 0.75f);
Collections.addAll(set, 1, 2, 2, 2, 2, 3);
System.out.println(set.toString()); // [1, 2, 3]
```

Четвертый конструктор создает множество на основе другой коллекции. В итоге мы получим только уникальные элементы коллекции:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 2, 2, 2, 3);
Set<Integer> set = new HashSet<Integer>(arr);
System.out.println(set.toString()); // [1, 2, 3]
```

В Java 9 в интерфейс `Set<E>` был добавлен статический метод `of()`, который создает неизменяемое множество. Форматы метода:

```
// Интерфейс Set<E>
public static <E> Set<E> of()
public static <E> Set<E> of(E el[, ..., E e10])
public static <E> Set<E> of(E... elements)
```

Первый формат создает пустое неизменяемое множество:

```
Set<Integer> set = Set.<Integer>of();
System.out.println(set.size()); // 0
// Ошибка! Нельзя добавить элемент, т. к. множество неизменяемое
// set.add(20);
```

Второй формат позволяет указать от одного до десяти уникальных элементов через запятую:

```
Set<Integer> set1 = Set.of(1);
System.out.println(set1); // [1]
Set<Integer> set2 = Set.of(1, 2, 3, 4, 5);
System.out.println(set2); // [1, 3, 2, 5, 4]
// Ошибка! Нельзя добавить элемент, т. к. множество неизменяемое
// set2.add(20);
Integer[] arrInt = {1, 2, 3, 4, 5};
Set<Integer[]> set3 = Set.<Integer[]>of(arrInt);
System.out.println(set3.size()); // 1
```

Третий формат позволяет указать произвольное количество уникальных элементов через запятую или в виде массива:

```
Set<Integer> set1 = Set.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11);
System.out.println(set1); // [5, 4, 7, 6, 1, 3, 2, 9, 8, 11, 10]
Integer[] arrInt = {1, 2, 3, 4, 5};
Set<Integer> set2 = Set.of(arrInt);
System.out.println(set2); // [5, 4, 1, 3, 2]
// Ошибка! Нельзя добавить элемент, т. к. множество неизменяемое
// set2.add(20);
```

Обратите внимание: элементы, передаваемые методу `of()`, должны быть уникальными. В противном случае генерируется исключение `IllegalArgumentException`:

```
Set<Integer> set = Set.of(1, 2, 3, 3);
// Исключение IllegalArgumentException
// duplicate element: 3
```

В Java 10 в интерфейс `Set<E>` был добавлен статический метод `copyOf()`, который создает неизменяемое множество на основе другой коллекции. Формат метода:

```
// Интерфейс Set<E>
public static <E> Set<E> copyOf(Collection<? extends E> coll)
```

Пример:

```
var arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5);
Set<Integer> set = Set.copyOf(arr);
System.out.println(set.toString()); // [4, 5, 1, 2, 3]
// Ошибка! Нельзя добавить элемент, т. к. множество неизменяемое
// set.add(20);
```

Создать пустое неизменяемое множество позволяет статический метод `emptySet()` из класса `Collections`. Такое множество удобно возвращать из метода при отсутствии значений. Формат метода:

```
import java.util.Collections;
public static final <T> Set<T> emptySet()
```

Пример:

```
Set<Integer> set = Collections.<Integer>emptySet();
System.out.println(set.size()); // 0
// Ошибка! Нельзя добавить элемент, т. к. множество неизменяемое
// set.add(20);
```

15.2.2. Вставка элементов

Вставить элементы позволяют следующие методы:

- `add()` — добавляет один элемент. Метод возвращает значение `true`, если элемент был добавлен, и `false` — в противном случае. Формат метода:

```
// Интерфейсы Collection<E> и Set<E>
public boolean add(E e)
```

Пример:

```
HashSet<Integer> set = new HashSet<Integer>();
System.out.println(set.add(1));     // true
System.out.println(set.add(2));     // true
System.out.println(set.add(1));     // false
System.out.println(set.add(1));     // false
System.out.println(set.add(3));     // true
System.out.println(set.toString()); // [1, 2, 3]
```

- `addAll()` — добавляет несколько элементов из другой коллекции. Метод возвращает значение `true`, если элементы были добавлены, и `false` — в противном случае. Формат метода:

```
// Интерфейсы Collection<E> и Set<E>
public boolean addAll(Collection<? extends E> c)
```

Пример:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 2, 2, 2, 3);
HashSet<Integer> set = new HashSet<Integer>();
System.out.println(set.addAll(arr)); // true
System.out.println(set.toString());  // [1, 2, 3]
```

Для добавления элементов можно также воспользоваться статическим методом `addAll()` из класса `Collections`. Формат метода:

```
import java.util.Collections;
public static <T> boolean addAll(Collection<? super T> c, T... elements)
```

В первом параметре указывается ссылка на коллекцию, а во втором параметре можно указать значения через запятую или обычный массив. Метод возвращает значение `true`, если элементы были добавлены, и `false` — в противном случае.

Пример:

```
HashSet<Integer> set = new HashSet<Integer>();
System.out.println(
   Collections.addAll(set, 1, 2, 2, 2, 2, 3)); // true
System.out.println(set.toString());            // [1, 2, 3]
```

15.2.3. Определение количества элементов

Для определения количества элементов предназначен метод `size()`. Формат метода:

```
// Интерфейсы Collection<E> и Set<E>
public int size()
```

Пример:

```
HashSet<Integer> set = new HashSet<Integer>();
Collections.addAll(set, 1, 2, 2, 2, 2, 3);
System.out.println(set.size());     // 3
System.out.println(set.toString()); // [1, 2, 3]
```

С помощью метода `isEmpty()` можно проверить, содержит множество элементы или нет. Метод возвращает значение `true`, если множество пустое, и `false` — в противном случае. Формат метода:

```
// Интерфейсы Collection<E> и Set<E>
public boolean isEmpty()
```

Пример:

```
HashSet<Integer> set = new HashSet<Integer>();
System.out.println(set.isEmpty()); // true
Collections.addAll(set, 1, 2, 2, 2, 2, 3);
System.out.println(set.isEmpty()); // false
```

15.2.4. Удаление элементов

Для удаления элементов предназначены следующие методы:

- `clear()` — удаляет все элементы из множества. Формат метода:

  ```
  // Интерфейсы Collection<E> и Set<E>
  public void clear()
  ```

  Пример:

  ```
  HashSet<Integer> set = new HashSet<Integer>();
  Collections.addAll(set, 1, 2, 2, 2, 2, 3);
  System.out.println(set.toString()); // [1, 2, 3]
  set.clear();
  System.out.println(set.toString()); // []
  ```

- `remove()` — удаляет элемент из множества. Метод возвращает значение `true`, если элемент был удален, и `false` — в противном случае. Формат метода:

```
// Интерфейсы Collection<E> и Set<E>
public boolean remove(Object o)
```

Пример:

```
HashSet<Integer> set = new HashSet<Integer>();
Collections.addAll(set, 1, 2, 2, 2, 2, 3);
System.out.println(set.toString()); // [1, 2, 3]
System.out.println(set.remove(1));  // true
System.out.println(set.remove(8));  // false
System.out.println(set.toString()); // [2, 3]
```

- `removeAll()` — удаляет из множества все элементы, соответствующие элементам указанной коллекции. Метод возвращает значение `true`, если элементы были удалены, и `false` — в противном случае. Формат метода:

```
// Интерфейсы Collection<E> и Set<E>
public boolean removeAll(Collection<?> c)
```

Пример:

```
ArrayList<Integer> arrDel = new ArrayList<Integer>();
Collections.addAll(arrDel, 1, 2);
HashSet<Integer> set = new HashSet<Integer>();
Collections.addAll(set, 1, 2, 3);
System.out.println(set.removeAll(arrDel));  // true
System.out.println(set.removeAll(arrDel));  // false
System.out.println(set.toString());         // [3]
```

- `removeIf()` — позволяет указать ссылку на метод, внутри которого нужно выполнить сравнение и вернуть логическое значение. Из множества будут удалены все элементы, для которых метод вернул значение `true`. Вместо указания ссылки на метод удобно использовать лямбда-выражение. Метод возвращает значение `true`, если элементы были удалены, и `false` — в противном случае. Метод доступен, начиная с Java 8. Формат метода:

```
// Интерфейсы Collection<E> и Set<E>
default boolean removeIf(Predicate<? super E> filter)
```

Удалим все элементы, значения которых меньше `4`:

```
HashSet<Integer> set = new HashSet<Integer>();
Collections.addAll(set, 1, 2, 3, 2, 4, 5, 6);
System.out.println(set.toString()); // [1, 2, 3, 4, 5, 6]
System.out.println(
   set.removeIf(elem -> elem < 4)); // true
System.out.println(set.toString()); // [4, 5, 6]
```

- `retainAll()` — удаляет из множества все элементы, которые не содержатся в указанной коллекции. Метод возвращает значение `true`, если элементы были удалены, и `false` — в противном случае. Формат метода:

```
// Интерфейсы Collection<E> и Set<E>
public boolean retainAll(Collection<?> c)
```

Пример:

```
HashSet<Integer> set = new HashSet<Integer>();
Collections.addAll(set, 1, 2, 3, 4, 5, 6);
ArrayList<Integer> arr2 = new ArrayList<Integer>();
Collections.addAll(arr2, 4, 5, 6);
System.out.println(set.retainAll(arr2)); // true
System.out.println(set.toString());      // [4, 5, 6]
System.out.println(set.retainAll(arr2)); // false
```

15.2.5. Проверка существования элементов

Проверить существование элементов во множестве позволяют следующие методы:

- `contains()` — возвращает значение `true`, если элемент существует во множестве, и `false` — в противном случае. Формат метода:

```
// Интерфейсы Collection<E> и Set<E>
public boolean contains(Object o)
```

Пример:

```
HashSet<Integer> set = new HashSet<Integer>();
Collections.addAll(set, 1, 2, 3);
System.out.println(set.contains(2)); // true
System.out.println(set.contains(8)); // false
```

- `containsAll()` — возвращает значение `true`, если все элементы из указанной коллекции существуют во множестве, и `false` — в противном случае. Формат метода:

```
// Интерфейсы Collection<E> и Set<E>
public boolean containsAll(Collection<?> c)
```

Пример:

```
HashSet<Integer> set = new HashSet<Integer>();
Collections.addAll(set, 1, 2, 3, 4, 5, 6);
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 4, 3, 2);
System.out.println(set.containsAll(arr)); // true
arr.add(8);
System.out.println(set.containsAll(arr)); // false
```

15.2.6. Преобразование массива во множество и множества в массив

Для преобразования обычного массива во множество можно воспользоваться статическим методом `addAll()` из класса `Collections`. Формат метода:

```
import java.util.Collections;
public static <T> boolean addAll(Collection<? super T> c, T... elements)
```

В первом параметре указывается ссылка на коллекцию, а во втором параметре можно указать значения через запятую или обычный массив. Новые уникальные элементы будут добавлены во множество. Метод возвращает значение `true`, если элементы были добавлены, и `false` — в противном случае:

```
Integer[] arrInt = {1, 2, 3, 8, 9};
HashSet<Integer> set = new HashSet<Integer>();
Collections.addAll(set, 1, 2, 3, 4);
Collections.addAll(set, arrInt);    // Добавляем элементы из массива
System.out.println(set.toString()); // [1, 2, 3, 4, 8, 9]
```

Преобразовать множество в массив позволяет метод `toArray()`. Форматы метода:

```
// Интерфейсы Collection<E> и Set<E>
public Object[] toArray()
public <T> T[] toArray(T[] a)
```

Первый формат возвращает массив объектов класса `Object`:

```
HashSet<Integer> set = new HashSet<Integer>();
Collections.addAll(set, 1, 2, 3);
System.out.println(set.toString());          // [1, 2, 3]
Object[] arrObj = set.toArray();
System.out.println(Arrays.toString(arrObj)); // [1, 2, 3]
Integer x = (Integer) arrObj[0];
System.out.println(x);                       // 1
```

Второй формат позволяет в качестве параметра указать ссылку на массив, который будет заполнен элементами из множества. Если передать нулевой массив, то внутри метода будет создан новый массив указанного типа. Метод возвращает ссылку на заполненный массив:

```
HashSet<Integer> set = new HashSet<Integer>();
Collections.addAll(set, 1, 2, 3);
System.out.println(set.toString());          // [1, 2, 3]
Integer[] arrInt = set.toArray(new Integer[0]);
System.out.println(Arrays.toString(arrInt)); // [1, 2, 3]
```

Если длина указанного массива больше размера множества, то элементы будут скопированы в этот массив, начиная с нулевого индекса. После копирования всех элементов будет добавлен еще один элемент со значением `null`. В этом случае метод вернет ссылку на первоначальный массив:

```
HashSet<Integer> set = new HashSet<Integer>();
Collections.addAll(set, 1, 2, 3);
System.out.println(set.toString());          // [1, 2, 3]
Integer[] arrInt = {8, 8, 8, 8, 8, 8, 8};
arrInt = set.toArray(arrInt);
System.out.println(Arrays.toString(arrInt));
// [1, 2, 3, null, 8, 8, 8]
```

15.2.7. Перебор элементов множества

Перебрать все элементы множества можно с помощью цикла `for each`, итератора и метода `forEach()` (начиная с Java 8). Обратите внимание на то, что элементы будут получены в произвольном порядке. Пример использования цикла `for each`:

```
HashSet<Integer> set = new HashSet<Integer>();
Collections.addAll(set, 1, 2, 3);
for (Integer item: set) {
   System.out.println(item);
}
```

Чтобы получить итератор, необходимо вызвать метод `iterator()`. Формат метода:

```
// Интерфейсы Iterable<T>, Collection<E> и Set<E>
public Iterator<E> iterator()
```

Пример перебора элементов множества с помощью итератора и цикла `while`:

```
HashSet<Integer> set = new HashSet<Integer>();
Collections.addAll(set, 1, 2, 3);
Iterator<Integer> it = set.iterator();
while (it.hasNext()) {
   System.out.println(it.next());
}
```

Начиная с Java 8, для перебора элементов используется также метод `forEach()`:

```
HashSet<Integer> set = new HashSet<Integer>();
Collections.addAll(set, 1, 2, 3);
set.forEach( elem -> System.out.println(elem) );
```

15.3. Класс *LinkedHashSet<E>*: множество, в котором запоминается порядок вставки элементов

Класс `LinkedHashSet<E>` реализует множество, в котором запоминается порядок вставки элементов. Добавить во множество можно только уникальный элемент. Прежде чем использовать класс `LinkedHashSet<E>`, необходимо импортировать его с помощью инструкции:

```
import java.util.LinkedHashSet;
```

Класс `LinkedHashSet<E>` реализует следующие интерфейсы:

```
Iterable<E>, Collection<E>, Set<E>, Serializable, Cloneable
```

Создать объект позволяют следующие конструкторы класса `LinkedHashSet<E>`:

```
LinkedHashSet()
LinkedHashSet(int initialCapacity)
LinkedHashSet(int initialCapacity, float loadFactor)
LinkedHashSet(Collection<? extends E> c)
```

Первый конструктор создает пустое множество с емкостью в 16 элементов и коэффициентом заполнения 0.75:

```
Set<Integer> set = new LinkedHashSet<Integer>();
System.out.println(set.size());     // 0
set.add(10);
set.add(33);
System.out.println(set.toString()); // [10, 33]
```

Второй конструктор позволяет указать начальную емкость множества, а третий — дополнительно — коэффициент заполнения (вещественное число от 0.0 до 1.0), при достижении которого произойдет расширение множества:

```
Set<Integer> set = new LinkedHashSet<Integer>(16, 0.75f);
Collections.addAll(set, 3, 1, 2, 2, 2, 2, 3);
System.out.println(set.toString()); // [3, 1, 2]
```

Четвертый конструктор создает множество на основе другой коллекции. В итоге мы получим только уникальные элементы коллекции:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 2, 2, 2, 3);
Set<Integer> set = new LinkedHashSet<Integer>(arr);
System.out.println(set.toString()); // [1, 2, 3]
```

Класс LinkedHashSet<E> наследует класс HashSet<E> и не добавляет никаких новых методов — различие только во внутренней реализации хранения элементов: класс HashSet<E> не гарантирует никакого порядка получения элементов, а класс LinkedHashSet<E> — запоминает порядок вставки элементов. Пример перебора элементов в том же порядке, что был и при добавлении:

```
LinkedHashSet<Integer> set = new LinkedHashSet<Integer>();
Collections.addAll(set, 1, 2, 3);
set.add(8);
set.add(0);
set.add(4);
for (Integer item: set) {
   System.out.print(item + " ");
} // 1 2 3 8 0 4
```

15.4. Интерфейсы *SortedSet<E>* и *NavigableSet<E>*

Интерфейс SortedSet<E> описывает набор уникальных элементов, хранимых в отсортированном порядке. Чтобы можно было выполнять сортировку, необходимо реализовать интерфейс Comparable<T> или передать конструктору объект, реализующий интерфейс Comparator<T>. Метод hashCode() в этом случае не используется, и никакая хеш-таблица не создается, однако метод equals() следует переопределить, а значит, и метод hashCode() также должен быть переопределен в классе. Ин-

терфейс NavigableSet<E> описывает методы, предназначенные для получения элементов и навигации по набору в обратном порядке.

Прежде чем использовать эти интерфейсы, необходимо выполнить их импорт с помощью инструкций:

```
import java.util.SortedSet;
import java.util.NavigableSet;
```

Иерархия наследования:

```
Iterable<T> - Collection<E> - Set<E> - SortedSet<E> - NavigableSet<E>
```

Интерфейсы SortedSet<E> и NavigableSet<E> реализует класс TreeSet<E>.

15.5. Класс *TreeSet<E>*: набор уникальных элементов, хранимых в отсортированном порядке

Класс TreeSet<E> описывает набор уникальных элементов, хранимых в отсортированном порядке в соответствии с алгоритмом «красно-черное дерево». Прежде чем использовать класс TreeSet<E>, необходимо импортировать его с помощью инструкции:

```
import java.util.TreeSet;
```

Класс TreeSet<E> реализует следующие интерфейсы:

```
Iterable<E>, Collection<E>, Set<E>, SortedSet<E>, NavigableSet<E>,
Serializable, Cloneable
```

15.5.1. Создание объекта

Создать объект позволяют следующие конструкторы класса TreeSet<E>:

```
TreeSet()
TreeSet(Comparator<? super E> comparator)
TreeSet(Collection<? extends E> c)
TreeSet(SortedSet<E> s)
```

Первый конструктор создает пустое множество со способом сравнения по умолчанию (интерфейс Comparable<T>):

```
TreeSet<Integer> set = new TreeSet<Integer>();
Collections.addAll(set, 4, 4, 8, 3, 1, 3, 0);
System.out.println(set.toString()); // [0, 1, 3, 4, 8]
```

Второй конструктор позволяет указать способ сравнения (интерфейс Comparator<T>):

```
TreeSet<Integer> set =
   new TreeSet<Integer>(Collections.reverseOrder());
Collections.addAll(set, 4, 4, 8, 3, 1, 3, 0);
System.out.println(set.toString()); // [8, 4, 3, 1, 0]
```

Третий конструктор создает объект на основе другой коллекции, используя способ сравнения по умолчанию:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 4, 4, 8, 3, 1, 3, 0);
TreeSet<Integer> set = new TreeSet<Integer>(arr);
System.out.println(set.toString()); // [0, 1, 3, 4, 8]
```

Четвертый конструктор создает объект на основе другого объекта, реализующего интерфейс `SortedSet<E>`. Способ сравнения берется из этого объекта. Пример сортировки в обратном порядке:

```
TreeSet<Integer> set2 =
   new TreeSet<Integer>(Collections.reverseOrder());
Collections.addAll(set2, 4, 4, 8, 3, 1, 3, 0);
TreeSet<Integer> set = new TreeSet<Integer>(set2);
System.out.println(set.toString()); // [8, 4, 3, 1, 0]
```

Класс `TreeSet<E>` реализует интерфейс `Set<E>`, поэтому он содержит все методы из этого интерфейса. Мы уже рассматривали эти методы применительно к классу `HashSet<E>`, поэтому не будем повторяться, — просто вернитесь к началу главы и замените во всех примерах класс `HashSet<E>` классом `TreeSet<E>`. Все примеры будут работать.

15.5.2. Методы из интерфейса *SortedSet<E>*

Класс `TreeSet<E>` реализует следующие методы из интерфейса `SortedSet<E>`:

- `comparator()` — возвращает объект сравнения или значение `null`, если используется сравнение по умолчанию. Формат метода:

  ```
  // Интерфейсы SortedSet<E> и NavigableSet<E>
  public Comparator<? super E> comparator()
  ```

- `first()` — возвращает первый элемент. Если набор пустой, генерируется исключение. Формат метода:

  ```
  // Интерфейсы SortedSet<E> и NavigableSet<E>
  public E first()
  ```

- `last()` — возвращает последний элемент. Если набор пустой, генерируется исключение. Формат метода:

  ```
  // Интерфейсы SortedSet<E> и NavigableSet<E>
  public E last()
  ```

  Пример:

  ```
  TreeSet<Integer> set = new TreeSet<Integer>();
  Collections.addAll(set, 4, 4, 8, 3, 1, 3);
  System.out.println(set.toString()); // [1, 3, 4, 8]
  System.out.println(set.first());    // 1
  System.out.println(set.last());     // 8
  ```

- `headSet()` — **возвращает срез от указанного значения (не включая элемент с этим значением) до начала набора. Обратите внимание на то, что все действия с этим набором повлияют на исходный набор. Формат метода:**

```
// Интерфейсы SortedSet<E> и NavigableSet<E>
public SortedSet<E> headSet(E toElement)
```

Пример:

```
TreeSet<Integer> set =
   new TreeSet<Integer>(Collections.reverseOrder());
Collections.addAll(set, 4, 4, 8, 3, 1, 3, 5);
System.out.println(set.toString());    // [8, 5, 4, 3, 1]
SortedSet<Integer> sSet = set.headSet(4);
System.out.println(sSet.toString());   // [8, 5]
// Изменения влияют на исходный набор
sSet.clear();
System.out.println(sSet.toString());   // []
System.out.println(set.toString());    // [4, 3, 1]
```

- `tailSet()` — **возвращает срез от указанного значения (включая элемент с этим значением) до конца набора. Обратите внимание на то, что все действия с этим набором повлияют на исходный набор. Формат метода:**

```
// Интерфейсы SortedSet<E> и NavigableSet<E>
public SortedSet<E> tailSet(E fromElement)
```

Пример:

```
TreeSet<Integer> set =
   new TreeSet<Integer>(Collections.reverseOrder());
Collections.addAll(set, 4, 4, 8, 3, 1, 3, 5);
System.out.println(set.toString());    // [8, 5, 4, 3, 1]
SortedSet<Integer> sSet = set.tailSet(4);
System.out.println(sSet.toString());   // [4, 3, 1]
// Изменения влияют на исходный набор
sSet.clear();
System.out.println(sSet.toString());   // []
System.out.println(set.toString());    // [8, 5]
```

- `subSet()` — **возвращает срез от значения** `fromElement` **(включая элемент с этим значением) до значения** `toElement` **(не включая элемент с этим значением). Обратите внимание на то, что все действия с этим набором повлияют на исходный набор. Формат метода:**

```
// Интерфейсы SortedSet<E> и NavigableSet<E>
public SortedSet<E> subSet(E fromElement, E toElement)
```

Пример:

```
TreeSet<Integer> set =
   new TreeSet<Integer>(Collections.reverseOrder());
Collections.addAll(set, 4, 4, 8, 3, 1, 3, 5);
```

```
System.out.println(set.toString());    // [8, 5, 4, 3, 1]
SortedSet<Integer> sSet = set.subSet(8, 3);
System.out.println(sSet.toString());   // [8, 5, 4]
// Изменения влияют на исходный набор
sSet.clear();
System.out.println(sSet.toString());   // []
System.out.println(set.toString());    // [3, 1]
```

Помимо рассмотренных методов через интерфейс `SortedSet<E>` доступны все методы из интерфейсов `Set<E>`, `Collection<E>` и `Iterable<E>`.

15.5.3. Методы из интерфейса *NavigableSet<E>*

Класс `TreeSet<E>` реализует следующие методы из интерфейса `NavigableSet<E>`:

- `pollFirst()` — удаляет первый элемент и возвращает его. Если набор пуст, то метод возвращает значение `null`. Формат метода:

```
// Интерфейс NavigableSet<E>
public E pollFirst()
```

- `pollLast()` — удаляет последний элемент и возвращает его. Если набор пуст, то метод возвращает значение `null`. Формат метода:

```
// Интерфейс NavigableSet<E>
public E pollLast()
```

Пример:

```
TreeSet<Integer> set = new TreeSet<Integer>();
Collections.addAll(set, 4, 4, 8, 3, 1, 3);
System.out.println(set.toString());   // [1, 3, 4, 8]
System.out.println(set.pollFirst());  // 1
System.out.println(set.toString());   // [3, 4, 8]
System.out.println(set.pollLast());   // 8
System.out.println(set.toString());   // [3, 4]
```

- `higher()` — возвращает элемент с бóльшим значением, чем указано (если сортировка в обратном порядке, то, наоборот, с меньшим значением). Если такого элемента нет, то метод возвращает значение `null`. Формат метода:

```
// Интерфейс NavigableSet<E>
public E higher(E e)
```

Пример:

```
TreeSet<Integer> set = new TreeSet<Integer>();
Collections.addAll(set, 4, 4, 8, 3, 1, 3);
System.out.println(set.toString());   // [1, 3, 4, 8]
System.out.println(set.higher(1));    // 3
System.out.println(set.higher(9));    // null
```

- `ceiling()` — возвращает элемент с бóльшим или равным значением, чем указано (если сортировка в обратном порядке, то, наоборот, с меньшим или равным зна-

чением). Если такого элемента нет, то метод возвращает значение `null`. Формат метода:

```
// Интерфейс NavigableSet<E>
public E ceiling(E e)
```

Пример:

```
TreeSet<Integer> set = new TreeSet<Integer>();
Collections.addAll(set, 4, 4, 8, 3, 1, 3);
System.out.println(set.toString());   // [1, 3, 4, 8]
System.out.println(set.ceiling(1));    // 1
System.out.println(set.ceiling(2));    // 3
System.out.println(set.ceiling(9));    // null
```

- `lower()` — возвращает элемент с меньшим значением, чем указано (если сортировка в обратном порядке, то, наоборот, с бóльшим значением). Если такого элемента нет, то метод возвращает значение `null`. Формат метода:

```
// Интерфейс NavigableSet<E>
public E lower(E e)
```

Пример:

```
TreeSet<Integer> set = new TreeSet<Integer>();
Collections.addAll(set, 4, 4, 8, 3, 1, 3);
System.out.println(set.toString());   // [1, 3, 4, 8]
System.out.println(set.lower(1));     // null
System.out.println(set.lower(2));     // 1
System.out.println(set.lower(9));     // 8
```

- `floor()` — возвращает элемент с меньшим или равным значением, чем указано (если сортировка в обратном порядке, то, наоборот, с бóльшим или равным значением). Если такого элемента нет, то метод возвращает значение `null`. Формат метода:

```
// Интерфейс NavigableSet<E>
public E floor(E e)
```

Пример:

```
TreeSet<Integer> set = new TreeSet<Integer>();
Collections.addAll(set, 4, 4, 8, 3, 1, 3);
System.out.println(set.toString());   // [1, 3, 4, 8]
System.out.println(set.floor(1));     // 1
System.out.println(set.floor(2));     // 1
System.out.println(set.floor(0));     // null
```

- `headSet()` — возвращает срез от указанного значения, включая элемент с этим значением (`inclusive = true`) или не включая его (`inclusive = false`), до начала набора. Обратите внимание на то, что все действия с этим набором повлияют на исходный набор. Форматы метода:

```
// Интерфейсы SortedSet<E> и NavigableSet<E>
public SortedSet<E> headSet(E toElement)
// Интерфейс NavigableSet<E>
public NavigableSet<E> headSet(E toElement,
                               boolean inclusive)
```

В первом формате `inclusive = false`:

```
TreeSet<Integer> set = new TreeSet<Integer>();
Collections.addAll(set, 4, 4, 8, 3, 1, 3, 5);
System.out.println(set.toString());    // [1, 3, 4, 5, 8]
NavigableSet<Integer> sSet = set.headSet(4, false);
System.out.println(sSet.toString());   // [1, 3]
// Изменения влияют на исходный набор
sSet.clear();
System.out.println(sSet.toString());   // []
System.out.println(set.toString());    // [4, 5, 8]
```

- `tailSet()` — **возвращает срез от указанного значения, включая элемент с этим значением** (`inclusive = true`) **или не включая его** (`inclusive = false`), **до конца набора. Обратите внимание на то, что все действия с этим набором повлияют на исходный набор. Форматы метода:**

```
// Интерфейсы SortedSet<E> и NavigableSet<E>
public SortedSet<E> tailSet(E fromElement)
// Интерфейс NavigableSet<E>
public NavigableSet<E> tailSet(E fromElement, boolean inclusive)
```

В первом формате `inclusive = true`:

```
TreeSet<Integer> set = new TreeSet<Integer>();
Collections.addAll(set, 4, 4, 8, 3, 1, 3, 5);
System.out.println(set.toString());    // [1, 3, 4, 5, 8]
NavigableSet<Integer> sSet = set.tailSet(4, true);
System.out.println(sSet.toString());   // [4, 5, 8]
// Изменения влияют на исходный набор
sSet.clear();
System.out.println(sSet.toString());   // []
System.out.println(set.toString());    // [1, 3]
```

- `subSet()` — **возвращает срез от значения** `fromElement`, **включая элемент с этим значением** (`fromInclusive = true`) **или не включая его** (`fromInclusive = false`), **до значения** `toElement`, **включая элемент с этим значением** (`toInclusive = true`) **или не включая его** (`toInclusive = false`). **Обратите внимание на то, что все действия с этим набором повлияют на исходный набор. Форматы метода:**

```
// Интерфейсы SortedSet<E> и NavigableSet<E>
public SortedSet<E> subSet(E fromElement, E toElement)
// Интерфейс NavigableSet<E>
public NavigableSet<E> subSet(
                       E fromElement, boolean fromInclusive,
                       E toElement,   boolean toInclusive)
```

В первом формате `fromInclusive = true` и `toInclusive = false`:

```
TreeSet<Integer> set = new TreeSet<Integer>();
Collections.addAll(set, 4, 4, 8, 3, 1, 3, 5);
System.out.println(set.toString());    // [1, 3, 4, 5, 8]
NavigableSet<Integer> sSet = set.subSet(3, true, 8, false);
System.out.println(sSet.toString());   // [3, 4, 5]
// Изменения влияют на исходный набор
sSet.clear();
System.out.println(sSet.toString());   // []
System.out.println(set.toString());    // [1, 8]
```

- `descendingSet()` — возвращает набор, отсортированный в обратном порядке. Формат метода:

```
// Интерфейс NavigableSet<E>
public NavigableSet<E> descendingSet()
```

Пример:

```
TreeSet<Integer> set = new TreeSet<Integer>();
Collections.addAll(set, 4, 4, 8, 3, 1, 3, 5);
System.out.println(set.toString());    // [1, 3, 4, 5, 8]
NavigableSet<Integer> sSet = set.descendingSet();
System.out.println(sSet.toString());   // [8, 5, 4, 3, 1]
```

- `descendingIterator()` — возвращает итератор, с помощью которого можно обойти набор в обратном порядке. Формат метода:

```
// Интерфейс NavigableSet<E>
public Iterator<E> descendingIterator()
```

Пример:

```
TreeSet<Integer> set = new TreeSet<Integer>();
Collections.addAll(set, 1, 2, 3);
System.out.println(set.toString()); // [1, 2, 3]
Iterator<Integer> it = set.descendingIterator();
while(it.hasNext()) {
   System.out.print(it.next() + " ");
} // 3 2 1
```

Помимо рассмотренных методов через интерфейс `NavigableSet<E>` доступны все методы из интерфейсов `SortedSet<E>`, `Set<E>`, `Collection<E>` и `Iterable<E>`.

15.6. Интерфейс *Map<K, V>*

Интерфейс `Map<K, V>` описывает структуру данных, называемую *словарем* или *ассоциативным массивом*. Доступ к элементам такой структуры осуществляется не по индексу, а по уникальному ключу. Чтобы получить доступ к элементу, необходимо указать ключ, который использовался при сохранении элемента. Для быстро-

го поиска ключа служит хеш-таблица, поэтому не забудьте внутри вашего класса переопределить методы `equals()` и `hashCode()`, если хотите использовать объект в качестве ключа. Объекты, хранимые в качестве значения, не используются в хеш-таблице, но могут при поиске сравниваться с другим объектом.

Прежде чем использовать этот интерфейс, необходимо выполнить его импорт с помощью инструкции:

```
import java.util.Map;
```

> ***Обратите внимание!***
> Интерфейс `Map<K, V>` *не* наследует интерфейсы `Iterable<T>` и `Collection<E>`.

Интерфейс `Map<K, V>` реализуют следующие классы:

- `HashMap<K, V>` — словарь, ключи которого расположены в произвольном порядке;
- `LinkedHashMap<K, V>` — словарь, в котором запоминается порядок вставки элементов или порядок доступа к элементам;
- `TreeMap<K, V>` — словарь, в котором ключи хранятся в отсортированном порядке в соответствии с алгоритмом «красно-черное дерево». Хеш-таблица не используется;
- `Hashtable<K, V>` — словарь. В отличие от класса `HashMap<K, V>` класс `Hashtable<K, V>` является синхронизированным и может быть использован для доступа из разных потоков;
- `Properties` — словарь, состоящий из конфигурационных данных. Ключи и значения такого словаря являются строками. Класс позволяет загрузить данные из файла и сохранить их в файл. Класс `Properties` является синхронизированным и может быть использован для доступа из разных потоков.

Пара ключ/значение описывается с помощью интерфейса `Map.Entry<K, V>`. Создать объект позволяет статический метод `entry()` из интерфейса `Map<K, V>`. Формат метода:

```
// Интерфейс Map<K, V>
public static <K, V> Map.Entry<K, V> entry(K k, V v)
```

Интерфейс `Map.Entry<K, V>` содержит следующие основные методы:

- `getKey()` — возвращает ключ. Формат метода:

  ```
  public K getKey()
  ```

- `getValue()` — возвращает значение. Формат метода:

  ```
  public V getValue()
  ```

  Пример:

  ```
  Map.Entry<String, Integer> obj = Map.entry("a", 10);
  System.out.println(obj.getKey());     // a
  System.out.println(obj.getValue());   // 10
  System.out.println(obj.toString());   // a=10
  ```

15.7. Класс *HashMap<K, V>*: словарь, доступ к элементам которого осуществляется по уникальному ключу

Класс `HashMap<K, V>` описывает словарь, доступ к элементам которого осуществляется не по индексу, а по уникальному ключу. Чтобы получить доступ к элементу, необходимо указать ключ, который использовался при сохранении элемента. Для быстрого поиска ключа служит хеш-таблица. Прежде чем использовать класс `HashMap<K, V>`, необходимо импортировать его с помощью инструкции:

```
import java.util.HashMap;
```

Класс `HashMap<K, V>` реализует следующие интерфейсы:

```
Map<K, V>, Serializable, Cloneable
```

15.7.1. Создание объекта

Создать словарь позволяют следующие конструкторы класса `HashMap<K, V>`:

```
HashMap()
HashMap(int initialCapacity)
HashMap(int initialCapacity, float loadFactor)
HashMap(Map<? extends K, ? extends V> m)
```

Первый конструктор создает пустой словарь с емкостью в 16 элементов и коэффициентом заполнения `0.75`:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
System.out.println(map.toString()); // {a=10, b=20}
```

Второй конструктор позволяет указать начальную емкость словаря, а третий — дополнительно — коэффициент заполнения (вещественное число от `0.0` до `1.0`), при достижении которого произойдет расширение хеш-таблицы словаря:

```
HashMap<String, Integer> map = new HashMap<String, Integer>(16, 0.75f);
map.put("a", 10);
map.put("b", 20);
System.out.println(map.get("a")); // 10
System.out.println(map.get("b")); // 20
```

Четвертый конструктор создает словарь на основе другого объекта, реализующего интерфейс `Map<K, V>`:

```
HashMap<String, Integer> map2 = new HashMap<String, Integer>();
map2.put("c", 30);
HashMap<String, Integer> map = new HashMap<String, Integer>(map2);
map.put("a", 10);
map.put("b", 20);
System.out.println(map.toString()); // {a=10, b=20, c=30}
```

В Java 9 в интерфейс `Map<K, V>` были добавлены статические методы `of()` и `ofEntries()`, которые создают неизменяемый словарь.

Форматы метода `of()`:

```
// Интерфейс Map<K, V>
public static <K, V> Map<K, V> of()
public static <K, V> Map<K, V> of(K k1, V v1[, ..., K k10, V v10])
```

Первый формат создает пустой неизменяемый словарь:

```
Map<String, Integer> map = Map.<String, Integer>of();
System.out.println(map.size()); // 0
// Ошибка! Нельзя добавить элемент, т. к. словарь неизменяемый
// map.put("a", 10);
```

Второй формат позволяет указать от одной до десяти пар ключ/значение через запятую:

```
Map<String, Integer> map1 = Map.of("a", 10);
System.out.println(map1); // {a=10}
Map<String, Integer> map2 = Map.of("a", 10, "b", 20);
System.out.println(map2); // {a=10, b=20}
// Ошибка! Нельзя добавить элемент, т. к. словарь неизменяемый
// map2.put("c", 30);
```

Формат метода `ofEntries()`:

```
// Интерфейс Map<K, V>
public static <K, V> Map<K, V> ofEntries(
       Map.Entry<? extends K, ? extends V>... entries)
```

Пример:

```
Map<String, Integer> map = Map.ofEntries(Map.entry("a", 10),
                                         Map.entry("b", 20));
System.out.println(map); // {b=20, a=10}
// Ошибка! Нельзя добавить элемент, т. к. словарь неизменяемый
// map.put("c", 30);
```

В Java 10 в интерфейс `Map<K, V>` был добавлен статический метод `copyOf()`, который создает неизменяемый словарь на основе другого словаря. Формат метода:

```
// Интерфейс Map<K, V>
public static <K, V> Map<K, V> copyOf(Map<? extends K, ? extends V> map)
```

Пример:

```
var map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
System.out.println(map);  // {a=10, b=20}
Map<String, Integer> map2 = Map.copyOf(map);
System.out.println(map2); // {a=10, b=20}
```

```
// Ошибка! Нельзя добавить элемент, т. к. словарь неизменяемый
// map2.put("c", 30);
```

Создать пустой неизменяемый словарь позволяет статический метод `emptyMap()` из класса `Collections`. Такой словарь удобно возвращать из метода при отсутствии значений. Формат метода:

```
import java.util.Collections;
public static final <K, V> Map<K, V> emptyMap()
```

Пример:

```
Map<String, Integer> map = Collections.<String, Integer>emptyMap();
System.out.println(map.size()); // 0
// Ошибка! Нельзя добавить элемент, т. к. словарь неизменяемый
// map.put("c", 30);
```

15.7.2. Вставка элементов

Вставить элементы позволяют следующие методы:

- `put()` — добавляет один элемент или изменяет значение существующего элемента. Метод возвращает значение `null`, если ключа не было в словаре, и старое значение, если ключ существует в словаре (при этом старое значение заменяется новым). Формат метода:

  ```
  // Интерфейс Map<K, V>
  public V put(K key, V value)
  ```

  Пример:

  ```
  HashMap<String, Integer> map = new HashMap<String, Integer>();
  System.out.println(map.put("a", 10)); // null
  System.out.println(map.put("b", 20)); // null
  System.out.println(map.put("b", 30)); // 20
  System.out.println(map.toString());   // {a=10, b=30}
  ```

- `putIfAbsent()` — добавляет один элемент, не изменяя значение существующего элемента. Метод возвращает значение `null`, если ключа не было в словаре, и старое значение, если ключ существует в словаре (при этом старое значение не заменяется новым). Метод доступен, начиная с Java 8. Формат метода:

  ```
  // Интерфейс Map<K, V>
  public V putIfAbsent(K key, V value)
  ```

  Пример:

  ```
  HashMap<String, Integer> map = new HashMap<String, Integer>();
  System.out.println(map.putIfAbsent("a", 10)); // null
  System.out.println(map.putIfAbsent("b", 20)); // null
  System.out.println(map.putIfAbsent("b", 30)); // 20
  System.out.println(map.toString());           // {a=10, b=20}
  ```

- `putAll()` — добавляет несколько элементов из другого объекта, реализующего интерфейс `Map<K, V>`. Если элементы с такими же ключами существуют в словаре, то их значения будут заменены новыми. Формат метода:

```
// Интерфейс Map<K, V>
public void putAll(Map<? extends K, ? extends V> m)
```

Пример:

```
HashMap<String, Integer> map2 = new HashMap<String, Integer>();
map2.put("b", 88);
map2.put("c", 30);
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
System.out.println(map.toString());   // {a=10, b=20}
map.putAll(map2);
System.out.println(map.toString());   // {a=10, b=88, c=30}
```

15.7.3. Определение количества элементов

Для определения количества элементов предназначен метод `size()`. Формат метода:

```
// Интерфейс Map<K, V>
public int size()
```

Пример:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
System.out.println(map.size()); // 2
```

С помощью метода `isEmpty()` можно проверить, содержит словарь элементы или нет. Метод возвращает значение `true`, если словарь пустой, и `false` — в противном случае. Формат метода:

```
// Интерфейс Map<K, V>
public boolean isEmpty()
```

Пример:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
System.out.println(map.isEmpty()); // true
map.put("a", 10);
System.out.println(map.isEmpty()); // false
```

15.7.4. Удаление элементов

Для удаления элементов предназначены следующие методы:

- `clear()` — удаляет все элементы из словаря. Формат метода:

```
// Интерфейс Map<K, V>
public void clear()
```

Пример:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.clear();
System.out.println(map.toString()); // {}
```

- `remove()` — удаляет элемент из словаря. Форматы метода:

```
// Интерфейс Map<K, V>
public V remove(Object key)
public boolean remove(Object key, Object value)
```

Первый формат удаляет элемент с указанным ключом из словаря и возвращает значение удаляемого элемента, если ключ существует в словаре, и `null` — если ключа нет в словаре:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
System.out.println(map.remove("b")); // 20
System.out.println(map.remove("b")); // null
System.out.println(map.toString());  // {a=10}
```

Второй формат удаляет элемент с указанным ключом и значением из словаря и возвращает значение `true`, если элемент удален, и `false` — в противном случае. Если ключ существует, но значение не совпадает с указанным, то элемент удален не будет. Формат доступен, начиная с Java 8:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
System.out.println(map.remove("b", 55)); // false
System.out.println(map.remove("b", 20)); // true
System.out.println(map.toString());      // {a=10}
```

15.7.5. Доступ к элементам

Для доступа к элементам предназначены следующие методы:

- `get()` — возвращает значение элемента с указанным ключом. Если ключ не существует, то метод возвращает значение `null`. Формат метода:

```
// Интерфейс Map<K, V>
public V get(Object key)
```

Пример:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
System.out.println(map.get("b")); // 20
System.out.println(map.get("c")); // null
```

- `getOrDefault()` — возвращает значение элемента с указанным ключом. Если ключ не существует, то метод возвращает значение, указанное во втором параметре. Метод доступен, начиная с Java 8. Формат метода:

```
// Интерфейс Map<K, V>
default V getOrDefault(Object key, V defaultValue)
```

Пример:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
System.out.println(map.getOrDefault("b", 0)); // 20
System.out.println(map.getOrDefault("c", 0)); // 0
```

- `keySet()` — возвращает множество с ключами. Формат метода:

```
// Интерфейс Map<K, V>
public Set<K> keySet()
```

Пример:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
Set<String> set = map.keySet();
System.out.println(set.toString()); // [a, b]
```

- `values()` — возвращает коллекцию со значениями. Формат метода:

```
// Интерфейс Map<K, V>
public Collection<V> values()
```

Пример:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
Collection<Integer> arr = map.values();
System.out.println(arr.toString()); // [10, 20]
```

- `entrySet()` — возвращает множество, состоящее из объектов, реализующих интерфейс `Map.Entry<K, V>`. Эти объекты содержат метод `getKey()` для получения ключа и метод `getValue()` для получения значения. Формат метода:

```
// Интерфейс Map<K, V>
public Set< Map.Entry<K, V> > entrySet()
```

Пример:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
Set< Map.Entry<String, Integer> > set = map.entrySet();
System.out.println(set.toString()); // [a=10, b=20]
```

```
for (Map.Entry<String, Integer> item: set) {
   System.out.println(item.getKey() + ": " + item.getValue());
}
// a: 10
// b: 20
```

15.7.6. Изменение значений элементов

Изменить значение элемента по ключу позволяют следующие методы:

- `put()` — добавляет один элемент или изменяет значение существующего элемента. Метод возвращает значение `null`, если ключа не было в словаре, и старое значение, если ключ существует в словаре (при этом старое значение заменяется новым). Формат метода:

```
// Интерфейс Map<K, V>
public V put(K key, V value)
```

Пример:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
System.out.println(map.put("a", 10)); // null
System.out.println(map.put("b", 20)); // null
System.out.println(map.put("b", 30)); // 20
System.out.println(map.toString());   // {a=10, b=30}
```

- `replace()` — изменяет значение элемента по ключу. Метод доступен, начиная с Java 8. Форматы метода:

```
// Интерфейс Map<K, V>
default V replace(K key, V value)
default boolean replace(K key, V oldValue, V newValue)
```

Первый формат изменяет значение элемента с указанным ключом и возвращает старое значение или значение `null`, если ключ отсутствует в словаре (новый элемент при этом не добавляется):

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
System.out.println(map.replace("b", 30)); // 20
System.out.println(map.replace("c", 50)); // null
System.out.println(map.toString());       // {a=10, b=30}
```

Второй формат изменяет значение, только если текущее значение совпадает со значением параметра `oldValue`. Метод возвращает значение `true`, если значение было изменено, и `false` — в противном случае:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
```

```
System.out.println(map.replace("b", 20, 30)); // true
System.out.println(map.replace("b", 50, 10)); // false
System.out.println(map.toString());           // {a=10, b=30}
```

- `replaceAll()` — применяет `function` ко всем элементам словаря. `function` принимает два параметра (ключ и значение) и возвращает новое значение. Метод `replaceAll()` доступен, начиная с Java 8. Формат метода:

```
// Интерфейс Map<K, V>
default void replaceAll(
   BiFunction<? super K, ? super V, ? extends V> function)
```

Внутри лямбда-выражения умножим каждое значение на 2:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
map.replaceAll( (k, v) -> map.get(k) * 2 );
System.out.println(map.toString()); // {a=20, b=40}
```

15.7.7. Проверка существования элементов

Проверить наличие элементов в словаре позволяют следующие методы:

- `containsKey()` — возвращает значение `true`, если ключ существует в словаре, и `false` — в противном случае. Формат метода:

```
// Интерфейс Map<K, V>
public boolean containsKey(Object key)
```

Пример:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
System.out.println(map.containsKey("b")); // true
System.out.println(map.containsKey("c")); // false
```

- `containsValue()` — возвращает значение `true`, если указанное значение существует в словаре, и `false` — в противном случае. Формат метода:

```
// Интерфейс Map<K, V>
public boolean containsValue(Object value)
```

Пример:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
System.out.println(map.containsValue(10)); // true
System.out.println(map.containsValue(50)); // false
```

15.7.8. Перебор элементов словаря

Перебрать все элементы словаря можно с помощью цикла `for each` и метода `forEach()` (начиная с Java 8). Обратите внимание на то, что элементы будут получены в произвольном порядке. Пример использования цикла `for each`:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
for (String item: map.keySet()) {
   System.out.println(item + ": " + map.get(item));
}
for (Map.Entry<String, Integer> item: map.entrySet()) {
   System.out.println(item.getKey() + ": " + item.getValue());
}
```

Словари не поддерживают итерации, поэтому в первом случае мы получаем множество с ключами и перебираем ключи из него. На каждой итерации через переменную `item` доступен ключ, с помощью которого мы можем получить связанное с ним значение, передав ключ методу `get()`. Во втором случае с помощью метода `entrySet()` мы получаем множество, состоящее из объектов, реализующих интерфейс `Map.Entry<K, V>`. Эти объекты содержат метод `getKey()` для получения ключа и метод `getValue()` для получения значения.

С помощью цикла `for each` мы можем не только перебирать элементы, но и изменять значения. Например, умножим все значения на 2:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
for (String item: map.keySet()) {
   map.put(item, map.get(item) * 2);
}
System.out.println(map.toString()); // {a=20, b=40}
```

Начиная с Java 8, для перебора элементов используется также метод `forEach()`. Формат метода:

```
// Интерфейс Map<K, V>
default void forEach(BiConsumer<? super K, ? super V> action)
```

Пример перебора элементов и изменения значений с использованием лямбда-выражений:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 10);
map.put("b", 20);
map.forEach( (k, v) -> System.out.println(k + ": " + v) );
map.forEach( (k, v) -> map.put(k, map.get(k) * 2) );
System.out.println(map.toString()); // {a=20, b=40}
```

15.8. Класс *LinkedHashMap<K, V>*: словарь, в котором запоминается порядок вставки элементов или порядок доступа к ним

Класс `LinkedHashMap<K, V>` реализует словарь, в котором запоминается порядок вставки элементов или порядок доступа к ним. Прежде чем использовать класс `LinkedHashMap<K, V>`, необходимо импортировать его с помощью инструкции:

```
import java.util.LinkedHashMap;
```

Класс `LinkedHashMap<K, V>` реализует следующие интерфейсы:

```
Map<K, V>, Serializable, Cloneable
```

Создать объект позволяют следующие конструкторы класса `LinkedHashMap<K, V>`:

```
LinkedHashMap()
LinkedHashMap(int initialCapacity)
LinkedHashMap(int initialCapacity, float loadFactor)
LinkedHashMap(int initialCapacity, float loadFactor,
              boolean accessOrder)
LinkedHashMap(Map<? extends K, ? extends V> m)
```

Первый конструктор создает пустой словарь с емкостью в 16 элементов и коэффициентом заполнения `0.75`:

```
LinkedHashMap<String, Integer> map =
                      new LinkedHashMap<String, Integer>();
map.put("b", 20);
map.put("a", 10);
System.out.println(map.toString()); // {b=20, a=10}
```

Второй конструктор позволяет указать начальную емкость словаря, а третий — дополнительно — коэффициент заполнения (вещественное число от `0.0` до `1.0`), при достижении которого произойдет расширение хеш-таблицы словаря:

```
LinkedHashMap<String, Integer> map =
              new LinkedHashMap<String, Integer>(16, 0.75f);
map.put("a", 10);
map.put("b", 20);
System.out.println(map.toString()); // {a=10, b=20}
```

Четвертый конструктор, помимо емкости словаря и коэффициента заполнения, позволяет указать порядок доступа. Если параметр `accessOrder` имеет значение `false` (значение по умолчанию в других конструкторах), то запоминается порядок вставки элементов:

```
LinkedHashMap<String, Integer> map =
    new LinkedHashMap<String, Integer>(16, 0.75f, false);
map.put("b", 20);
map.put("c", 30);
map.put("a", 10);
map.put("b", 25);
```

```
System.out.println(map.get("c"));   // 30
System.out.println(map.toString()); // {b=25, c=30, a=10}
```

Если параметр `accessOrder` имеет значение `true`, то запоминается порядок доступа к элементам, например, с помощью методов `put()` и `get()`:

```
LinkedHashMap<String, Integer> map =
    new LinkedHashMap<String, Integer>(16, 0.75f, true);
map.put("b", 20);
map.put("c", 30);
map.put("a", 10);
map.put("b", 25);
System.out.println(map.get("c"));   // 30
System.out.println(map.toString()); // {a=10, b=25, c=30}
```

Пятый конструктор создает словарь на основе другого объекта, реализующего интерфейс `Map<K, V>`:

```
HashMap<String, Integer> map2 = new HashMap<String, Integer>();
map2.put("c", 30);
LinkedHashMap<String, Integer> map =
                    new LinkedHashMap<String, Integer>(map2);
map.put("a", 10);
map.put("b", 20);
System.out.println(map.toString()); // {c=30, a=10, b=20}
```

Класс `LinkedHashMap<K, V>` наследует класс `HashMap<K, V>` и не добавляет никаких новых общедоступных методов — различие только во внутренней реализации хранения элементов: класс `HashMap<K, V>` не гарантирует никакого порядка получения элементов, а класс `LinkedHashMap<K, V>` — запоминает порядок вставки элементов или порядок доступа к ним. Пример перебора элементов в том же порядке, что был и при добавлении:

```
LinkedHashMap<String, Integer> map =
                    new LinkedHashMap<String, Integer>();
map.put("b", 20);
map.put("a", 10);
map.put("c", 30);
System.out.println(map.toString()); // {b=20, a=10, c=30}
for (String item: map.keySet()) {
   System.out.print(item + ": " + map.get(item) + " ");
} // b: 20 a: 10 c: 30
```

15.9. Интерфейсы *SortedMap<K, V>* и *NavigableMap<K, V>*

Интерфейс `SortedMap<K, V>` описывает словарь, в котором ключи хранятся в отсортированном порядке. Чтобы можно было выполнять сортировку, необходимо реализовать интерфейс `Comparable<T>` или передать конструктору объект, реализую-

щий интерфейс `Comparator<T>`. Метод `hashCode()` в этом случае не используется, и никакая хеш-таблица не создается. Интерфейс `NavigableMap<K, V>` описывает методы, предназначенные для получения элементов и навигации по ключам в обратном порядке.

Прежде чем использовать эти интерфейсы, необходимо выполнить их импорт с помощью инструкций:

```
import java.util.SortedMap;
import java.util.NavigableMap;
```

Иерархия наследования:

```
Map<K, V> - SortedMap<K, V> - NavigableMap<K, V>
```

Интерфейсы `SortedMap<K, V>` и `NavigableMap<K, V>` реализует класс `TreeMap<K, V>`.

15.10. Класс *TreeMap<K, V>*: словарь, в котором ключи хранятся в отсортированном порядке

Класс `TreeMap<K, V>` описывает словарь, в котором ключи хранятся в отсортированном порядке в соответствии с алгоритмом «красно-черное дерево». Прежде чем использовать класс `TreeMap<K, V>`, необходимо импортировать его с помощью инструкции:

```
import java.util.TreeMap;
```

Класс `TreeMap<K, V>` реализует следующие интерфейсы:

```
Map<K, V>, SortedMap<K, V>, NavigableMap<K, V>, Serializable, Cloneable
```

15.10.1. Создание объекта

Создать объект позволяют следующие конструкторы класса `TreeMap<K, V>`:

```
TreeMap()
TreeMap(Comparator<? super K> comparator)
TreeMap(Map<? extends K, ? extends V> m)
TreeMap(SortedMap<K, ? extends V> m)
```

Первый конструктор создает словарь со способом сравнения ключей по умолчанию (интерфейс `Comparable<T>`):

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 20);
map.put("c", 30);
map.put("a", 10);
System.out.println(map.toString()); // {a=10, b=20, c=30}
```

Второй конструктор позволяет указать способ сравнения (интерфейс `Comparator<T>`):

```
TreeMap<String, Integer> map =
    new TreeMap<String, Integer>(Collections.reverseOrder());
```

```
map.put("b", 20);
map.put("c", 30);
map.put("a", 10);
System.out.println(map.toString()); // {c=30, b=20, a=10}
```

Третий конструктор создает объект на основе другого словаря, используя способ сравнения по умолчанию:

```
HashMap<String, Integer> map2 = new HashMap<String, Integer>();
map2.put("c", 30);
TreeMap<String, Integer> map = new TreeMap<String, Integer>(map2);
map.put("b", 20);
map.put("a", 10);
System.out.println(map.toString()); // {a=10, b=20, c=30}
```

Четвертый конструктор создает объект на основе другого словаря, реализующего интерфейс `SortedMap<K, V>`. Способ сравнения берется из этого словаря. Пример сортировки в обратном порядке:

```
TreeMap<String, Integer> map2 =
   new TreeMap<String, Integer>(Collections.reverseOrder());
map2.put("b", 20);
map2.put("c", 30);
TreeMap<String, Integer> map = new TreeMap<String, Integer>(map2);
map.put("a", 10);
System.out.println(map.toString()); // {c=30, b=20, a=10}
```

Класс `TreeMap<K, V>` реализует интерфейс `Map<K, V>`, поэтому он содержит все методы из этого интерфейса. Мы уже рассматривали эти методы применительно к классу `HashMap<K, V>`, поэтому не будем повторяться, — просто вернитесь к описанию этого класса и замените во всех примерах класс `HashMap<K, V>` классом `TreeMap<K, V>`.

15.10.2. Методы из интерфейса *SortedMap<K, V>*

Класс `TreeMap<K, V>` реализует следующие методы из интерфейса `SortedMap<K, V>`:

- `comparator()` — возвращает объект сравнения или значение `null`, если используется сравнение по умолчанию. Формат метода:

```
// Интерфейсы SortedMap<K, V> и NavigableMap<K, V>
public Comparator<? super K> comparator()
```

- `firstKey()` — возвращает первый ключ. Если словарь пустой, генерируется исключение. Формат метода:

```
// Интерфейсы SortedMap<K, V> и NavigableMap<K, V>
public K firstKey()
```

- `lastKey()` — возвращает последний ключ. Если словарь пустой, генерируется исключение. Формат метода:

```
// Интерфейсы SortedMap<K, V> и NavigableMap<K, V>
public K lastKey()
```

Пример:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 20);
map.put("c", 30);
map.put("a", 10);
System.out.println(map.firstKey()); // a
System.out.println(map.lastKey());  // c
```

- `headMap()` — возвращает срез от указанного ключа (не включая элемент с этим ключом) до начала набора. Обратите внимание на то, что все действия с этим набором повлияют на исходный словарь. Формат метода:

```
// Интерфейсы SortedMap<K, V> и NavigableMap<K, V>
public SortedMap<K, V> headMap(K toKey)
```

Пример:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 2);
map.put("c", 3);
map.put("a", 1);
map.put("d", 4);
System.out.println(map.toString());   // {a=1, b=2, c=3, d=4}
SortedMap<String, Integer> smap = map.headMap("b");
System.out.println(smap); // {a=1}
// Изменения влияют на исходный словарь
smap.clear();
System.out.println(map.toString());   // {b=2, c=3, d=4}
```

- `tailMap()` — возвращает срез от указанного ключа (включая элемент с этим ключом) до конца набора. Обратите внимание на то, что все действия с этим набором повлияют на исходный словарь. Формат метода:

```
// Интерфейсы SortedMap<K, V> и NavigableMap<K, V>
public SortedMap<K, V> tailMap(K fromKey)
```

Пример:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 2);
map.put("c", 3);
map.put("a", 1);
map.put("d", 4);
System.out.println(map.toString());   // {a=1, b=2, c=3, d=4}
SortedMap<String, Integer> smap = map.tailMap("b");
System.out.println(smap); // {b=2, c=3, d=4}
// Изменения влияют на исходный словарь
smap.clear();
System.out.println(map.toString());   // {a=1}
```

- `subMap()` — возвращает срез от ключа `fromKey` (включая элемент с этим ключом) до значения `toKey` (не включая элемент с этим ключом). Обратите внимание на то, что все действия с этим набором повлияют на исходный словарь. Формат метода:

```
// Интерфейсы SortedMap<K, V> и NavigableMap<K, V>
public SortedMap<K, V> subMap(K fromKey, K toKey)
```

Пример:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 2);
map.put("c", 3);
map.put("a", 1);
map.put("d", 4);
System.out.println(map.toString());   // {a=1, b=2, c=3, d=4}
SortedMap<String, Integer> smap = map.subMap("b", "d");
System.out.println(smap); // {b=2, c=3}
// Изменения влияют на исходный словарь
smap.clear();
System.out.println(map.toString());   // {a=1, d=4}
```

Помимо рассмотренных методов через интерфейс `SortedMap<K, V>` доступны все методы из интерфейса `Map<K, V>`.

15.10.3. Методы из интерфейса *NavigableMap<K, V>*

Класс `TreeMap<K, V>` реализует следующие методы из интерфейса `NavigableMap<K, V>`:

- `firstEntry()` — возвращает первый элемент. Если словарь пустой, возвращается значение `null`. Формат метода:

```
// Интерфейс NavigableMap<K, V>
public Map.Entry<K, V> firstEntry()
```

- `lastEntry()` — возвращает последний элемент. Если словарь пустой, возвращается значение `null`. Формат метода:

```
// Интерфейс NavigableMap<K, V>
public Map.Entry<K, V> lastEntry()
```

Пример:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 20);
map.put("c", 30);
map.put("a", 10);
System.out.println(map.firstEntry()); // a=10
System.out.println(map.lastEntry());  // c=30
```

- `pollFirstEntry()` — удаляет первый элемент и возвращает его. Если словарь пустой, возвращается значение `null`. Формат метода:

```
// Интерфейс NavigableMap<K, V>
public Map.Entry<K, V> pollFirstEntry()
```

- `pollLastEntry()` — **удаляет последний элемент и возвращает его. Если словарь пустой, возвращается значение** `null`. **Формат метода:**

```
// Интерфейс NavigableMap<K, V>
public Map.Entry<K, V> pollLastEntry()
```

Пример:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 20);
map.put("c", 30);
map.put("a", 10);
System.out.println(map.pollFirstEntry()); // a=10
System.out.println(map.pollLastEntry());  // c=30
System.out.println(map.toString());       // {b=20}
```

- `higherKey()` — **возвращает ключ с бóльшим значением, чем указано (если сортировка в обратном порядке, то, наоборот, с меньшим значением). Если такого ключа нет, то метод возвращает значение** `null`. **Формат метода:**

```
// Интерфейс NavigableMap<K, V>
public K higherKey(K key)
```

Пример:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 20);
map.put("c", 30);
map.put("a", 10);
System.out.println(map.toString());     // {a=10, b=20, c=30}
System.out.println(map.higherKey("b")); // c
System.out.println(map.higherKey("c")); // null
```

- `higherEntry()` — **возвращает элемент с бóльшим значением ключа, чем указано (если сортировка в обратном порядке, то, наоборот, с меньшим значением). Если такого элемента нет, то метод возвращает значение** `null`. **Формат метода:**

```
// Интерфейс NavigableMap<K, V>
public Map.Entry<K, V> higherEntry(K key)
```

Пример:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 20);
map.put("c", 30);
map.put("a", 10);
System.out.println(map.toString());     // {a=10, b=20, c=30}
System.out.println(map.higherEntry("b")); // c=30
System.out.println(map.higherEntry("c")); // null
```

- `ceilingKey()` — возвращает ключ с бóльшим или равным значением, чем указано (если сортировка в обратном порядке, то, наоборот, с меньшим или равным значением). Если такого ключа нет, то метод возвращает значение `null`. Формат метода:

```
// Интерфейс NavigableMap<K, V>
public K ceilingKey(K key)
```

Пример:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 2);
map.put("d", 4);
map.put("a", 1);
System.out.println(map.toString());       // {a=1, b=2, d=4}
System.out.println(map.ceilingKey("b")); // b
System.out.println(map.ceilingKey("c")); // d
System.out.println(map.ceilingKey("f")); // null
```

- `ceilingEntry()` — возвращает элемент с бóльшим или равным значением ключа, чем указано (если сортировка в обратном порядке, то, наоборот, с меньшим или равным значением). Если такого элемента нет, то метод возвращает значение `null`. Формат метода:

```
// Интерфейс NavigableMap<K, V>
public Map.Entry<K, V> ceilingEntry(K key)
```

Пример:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 2);
map.put("d", 4);
map.put("a", 1);
System.out.println(map.toString());       // {a=1, b=2, d=4}
System.out.println(map.ceilingEntry("b")); // b=2
System.out.println(map.ceilingEntry("c")); // d=4
System.out.println(map.ceilingEntry("f")); // null
```

- `lowerKey()` — возвращает ключ с меньшим значением, чем указано (если сортировка в обратном порядке, то, наоборот, с бóльшим значением). Если такого ключа нет, то метод возвращает значение `null`. Формат метода:

```
// Интерфейс NavigableMap<K, V>
public K lowerKey(K key)
```

Пример:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 2);
map.put("d", 4);
map.put("a", 1);
System.out.println(map.toString());     // {a=1, b=2, d=4}
```

```
System.out.println(map.lowerKey("d")); // b
System.out.println(map.lowerKey("a")); // null
```

- `lowerEntry()` — возвращает элемент с меньшим значением ключа, чем указано (если сортировка в обратном порядке, то, наоборот, с бóльшим значением). Если такого элемента нет, то метод возвращает значение `null`. Формат метода:

```
// Интерфейс NavigableMap<K, V>
public Map.Entry<K, V> lowerEntry(K key)
```

Пример:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 2);
map.put("d", 4);
map.put("a", 1);
System.out.println(map.toString());       // {a=1, b=2, d=4}
System.out.println(map.lowerEntry("d")); // b=2
System.out.println(map.lowerEntry("a")); // null
```

- `floorKey()` — возвращает ключ с меньшим или равным значением, чем указано (если сортировка в обратном порядке, то, наоборот, с бóльшим или равным значением). Если такого ключа нет, то метод возвращает значение `null`. Формат метода:

```
// Интерфейс NavigableMap<K, V>
public K floorKey(K key)
```

Пример:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 2);
map.put("d", 4);
map.put("a", 1);
System.out.println(map.toString());     // {a=1, b=2, d=4}
System.out.println(map.floorKey("d")); // d
System.out.println(map.floorKey("c")); // b
System.out.println(map.floorKey("3")); // null
```

- `floorEntry()` — возвращает элемент с меньшим или равным значением ключа, чем указано (если сортировка в обратном порядке, то, наоборот, с бóльшим или равным значением). Если такого элемента нет, то метод возвращает значение `null`. Формат метода:

```
// Интерфейс NavigableMap<K, V>
public Map.Entry<K, V> floorEntry(K key)
```

Пример:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 2);
map.put("d", 4);
map.put("a", 1);
```

```
System.out.println(map.toString());       // {a=1, b=2, d=4}
System.out.println(map.floorEntry("d")); // d=4
System.out.println(map.floorEntry("c")); // b=2
System.out.println(map.floorEntry("3")); // null
```

- `headMap()` — **возвращает срез от указанного ключа, включая элемент с этим ключом (`inclusive = true`) или не включая его (`inclusive = false`), до начала набора. Обратите внимание на то, что все действия с этим набором повлияют на исходный словарь. Форматы метода:**

```
// Интерфейсы SortedMap<K, V> и NavigableMap<K, V>
public SortedMap<K, V> headMap(K toKey)
// Интерфейс NavigableMap<K, V>
public NavigableMap<K, V> headMap(K toKey,
                                  boolean inclusive)
```

В первом формате `inclusive = false`:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 2);
map.put("c", 3);
map.put("a", 1);
map.put("d", 4);
System.out.println(map.toString());   // {a=1, b=2, c=3, d=4}
NavigableMap<String, Integer> smap = map.headMap("b", false);
System.out.println(smap); // {a=1}
// Изменения влияют на исходный словарь
smap.clear();
System.out.println(map.toString());   // {b=2, c=3, d=4}
```

- `tailMap()` — **возвращает срез от указанного ключа, включая элемент с этим ключом (`inclusive = true`) или не включая его (`inclusive = false`), до конца набора. Обратите внимание на то, что все действия с этим набором повлияют на исходный словарь. Форматы метода:**

```
// Интерфейсы SortedMap<K, V> и NavigableMap<K, V>
public SortedMap<K, V> tailMap(K fromKey)
// Интерфейс NavigableMap<K, V>
public NavigableMap<K, V> tailMap(K fromKey,
                                  boolean inclusive)
```

В первом формате `inclusive = true`:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 2);
map.put("c", 3);
map.put("a", 1);
map.put("d", 4);
System.out.println(map.toString());   // {a=1, b=2, c=3, d=4}
NavigableMap<String, Integer> smap = map.tailMap("b", true);
System.out.println(smap); // {b=2, c=3, d=4}
```

```
// Изменения влияют на исходный словарь
smap.clear();
System.out.println(map.toString());   // {a=1}
```

- `subMap()` — **возвращает срез от ключа** `fromKey`, **включая элемент с этим ключом** (`fromInclusive = true`) **или не включая его** (`fromInclusive = false`), **до ключа** `toKey`, **включая элемент с этим ключом** (`toInclusive = true`) **или не включая его** (`toInclusive = false`). **Обратите внимание на то, что все действия с этим набором повлияют на исходный словарь. Форматы метода:**

```
// Интерфейсы SortedMap<K, V> и NavigableMap<K, V>
public SortedMap<K, V> subMap(K fromKey, K toKey)
// Интерфейс NavigableMap<K, V>
public NavigableMap<K, V> subMap(
                  K fromKey, boolean fromInclusive,
                  K toKey, boolean toInclusive)
```

В первом формате `fromInclusive = true` **и** `toInclusive = false`:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 2);
map.put("c", 3);
map.put("a", 1);
map.put("d", 4);
System.out.println(map.toString());   // {a=1, b=2, c=3, d=4}
NavigableMap<String, Integer> smap =
                           map.subMap("b", true, "d", false);
System.out.println(smap); // {b=2, c=3}
// Изменения влияют на исходный словарь
smap.clear();
System.out.println(map.toString());   // {a=1, d=4}
```

- `navigableKeySet()` — **возвращает набор ключей, отсортированный в прямом порядке. Формат метода:**

```
// Интерфейс NavigableMap<K, V>
public NavigableSet<K> navigableKeySet()
```

Пример:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 2);
map.put("c", 3);
map.put("a", 1);
System.out.println(map.toString());   // {a=1, b=2, c=3}
NavigableSet<String> keys = map.navigableKeySet();
System.out.println(keys.toString());  // [a, b, c]
```

- `descendingKeySet()` — **возвращает набор ключей, отсортированный в обратном порядке. Формат метода:**

```
// Интерфейс NavigableMap<K, V>
public NavigableSet<K> descendingKeySet()
```

Пример:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 2);
map.put("c", 3);
map.put("a", 1);
System.out.println(map.toString());   // {a=1, b=2, c=3}
NavigableSet<String> keys = map.descendingKeySet();
System.out.println(keys.toString());  // [c, b, a]
```

- `descendingMap()` — возвращает словарь с ключами, отсортированными в обратном порядке. Формат метода:

```
// Интерфейс NavigableMap<K, V>
public NavigableMap<K, V> descendingMap()
```

Пример:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
map.put("b", 2);
map.put("c", 3);
map.put("a", 1);
System.out.println(map.toString());   // {a=1, b=2, c=3}
NavigableMap<String, Integer> map2 = map.descendingMap();
System.out.println(map2.toString());  // {c=3, b=2, a=1}
```

Помимо рассмотренных методов через интерфейс `NavigableMap<K, V>` доступны все методы из интерфейсов `Map<K, V>` и `SortedMap<K, V>`.

15.11. Класс *Hashtable<K, V>*: словарь

Класс `Hashtable<K, V>` реализует словарь. В отличие от класса `HashMap<K, V>`, класс `Hashtable<K, V>` является синхронизированным и может быть использован для доступа из разных потоков. Из-за потерь на синхронизацию класс `Hashtable<K, V>` работает медленнее, чем класс `HashMap<K, V>`. Прежде чем использовать класс `Hashtable<K, V>`, необходимо импортировать его с помощью инструкции:

```
import java.util.Hashtable;
```

Класс `Hashtable<K, V>` реализует следующие интерфейсы:

```
Map<K, V>, Serializable, Cloneable
```

Создать словарь позволяют следующие конструкторы класса `Hashtable<K, V>`:

```
Hashtable()
Hashtable(int initialCapacity)
Hashtable(int initialCapacity, float loadFactor)
Hashtable(Map<? extends K, ? extends V> m)
```

Первый конструктор создает пустой словарь с емкостью в 11 элементов и коэффициентом заполнения `0.75`:

```
Hashtable<String, Integer> map = new Hashtable<String, Integer>();
map.put("a", 10);
map.put("b", 20);
System.out.println(map.toString()); // {b=20, a=10}
```

Второй конструктор позволяет указать начальную емкость словаря, а третий — дополнительно — коэффициент заполнения (вещественное число от `0.0` до `1.0`), при достижении которого произойдет расширение хеш-таблицы словаря:

```
Hashtable<String, Integer> map =
                  new Hashtable<String, Integer>(11, 0.75f);
map.put("a", 10);
map.put("b", 20);
System.out.println(map.get("a")); // 10
System.out.println(map.get("b")); // 20
```

Четвертый конструктор создает словарь на основе другого объекта, реализующего интерфейс `Map<K, V>`:

```
HashMap<String, Integer> map2 = new HashMap<String, Integer>();
map2.put("c", 30);
Hashtable<String, Integer> map = new Hashtable<String, Integer>(map2);
map.put("a", 10);
map.put("b", 20);
System.out.println(map.toString()); // {b=20, a=10, c=30}
```

Класс `Hashtable<K, V>` реализует интерфейс `Map<K, V>`, поэтому содержит все методы из этого интерфейса. При изучении класса `HashMap<K, V>` мы уже рассмотрели эти методы, поэтому не будем повторяться, — просто откройте предыдущие разделы и в примерах подставьте класс `Hashtable<K, V>` вместо класса `HashMap<K, V>`.

Класс `Hashtable<K, V>` был реализован задолго до создания каркаса коллекций, и его доработали, чтобы вписаться в этот каркас, поэтому, в отличие от других коллекций, он является синхронизированным по умолчанию и содержит некоторые дополнительные методы:

- `keys()` — возвращает объект, реализующий интерфейс `Enumeration<K>` (см. *разд. 14.10.3*), с ключами из словаря. Формат метода:

  ```
  public Enumeration<K> keys()
  ```

  Пример:

  ```
  Hashtable<String, Integer> map =
                    new Hashtable<String, Integer>();
  map.put("a", 10);
  map.put("b", 20);
  Enumeration<String> it = map.keys();
  while(it.hasMoreElements()) {
     System.out.print(it.nextElement() + " ");
  } // b a
  ```

- `elements()` — возвращает объект, реализующий интерфейс `Enumeration<V>` (см. *разд. 14.10.3*), со значениями из словаря. Формат метода:

```
public Enumeration<V> elements()
```

Пример:

```
Hashtable<String, Integer> map =
                  new Hashtable<String, Integer>();
map.put("a", 10);
map.put("b", 20);
Enumeration<Integer> it = map.elements();
while (it.hasMoreElements()) {
   System.out.print(it.nextElement() + " ");
} // 20 10
```

15.12. Класс *Properties*: словарь, состоящий из конфигурационных данных

Класс `Properties` реализует словарь, состоящий из конфигурационных данных. Ключи и значения такого словаря являются строками. Класс позволяет загрузить данные из конфигурационного файла и сохранить их в файл. Прежде чем использовать класс `Properties`, необходимо импортировать его с помощью инструкции:

```
import java.util.Properties;
```

Класс `Properties` реализует следующие интерфейсы:

```
Map<K, V>, Serializable, Cloneable
```

Создать объект позволяют следующие конструкторы класса `Properties`:

```
Properties()
Properties(Properties defaults)
Properties(int initialCapacity)
```

Первый формат создает пустой объект, а второй — позволяет указать объект со значениями по умолчанию:

```
Properties map = new Properties();
map.setProperty("login", "mylogin");
map.setProperty("password", "85498742");
System.out.println(map.toString());
// {login=mylogin, password=85498742}
Properties map2 = new Properties(map);
System.out.println(map2.getProperty("login"));    // mylogin
System.out.println(map2.getProperty("password")); // 85498742
```

Третий конструктор доступен, начиная с Java 10. Он позволяет указать начальную емкость словаря:

```
Properties map = new Properties(50);
```

Класс `Properties` реализует интерфейс `Map<K, V>`, поэтому содержит все методы из этого интерфейса. Все такие методы мы уже рассматривали ранее. Кроме того, класс `Properties` наследует класс `Hashtable<K, V>` и содержит методы из этого класса, но методы `keys()` и `elements()` всегда возвращают объект `Enumeration<Object>`:

```
Properties map = new Properties();
map.setProperty("a", "10");
map.setProperty("b", "20");
Enumeration<Object> it = map.keys();
String s = "";
while (it.hasMoreElements()) {
   s = (String) it.nextElement();
   System.out.print(s + " ");
} // b a
it = map.elements();
s = "";
while (it.hasMoreElements()) {
   s = (String) it.nextElement();
   System.out.print(s + " ");
} // 20 10
```

Класс `Properties` содержит следующие дополнительные методы:

- `setProperty()` — создает новый элемент или изменяет значение существующего. Если ключа нет в словаре, то добавляет элемент и возвращает значение `null`. Если ключ существует, то перезаписывает значение и возвращает старое значение. Формат метода:

  ```
  public Object setProperty(String key, String value)
  ```

  Пример:

  ```
  Properties map = new Properties();
  map.setProperty("login", "mylogin");
  map.setProperty("password", "85498742");
  System.out.println(map.toString());
  // {login=mylogin, password=85498742}
  ```

- `getProperty()` — возвращает значение по указанному ключу. Форматы метода:

  ```
  public String getProperty(String key)
  public String getProperty(String key, String defaultValue)
  ```

  Если ключ существует, то возвращается значение, а если не существует, то значение `null`, или значение параметра `defaultValue`, или значение по умолчанию из объекта, указанного при создании словаря:

  ```
  Properties map = new Properties();
  map.setProperty("login", "mylogin");
  map.setProperty("password", "85498742");
  System.out.println(map.getProperty("login"));      // mylogin
  System.out.println(map.getProperty("password"));   // 85498742
  ```

```
System.out.println(map.getProperty("id"));         // null
System.out.println(map.getProperty("id", "123")); // 123
```

- `stringPropertyNames()` — возвращает набор с ключами. Формат метода:

```
public Set<String> stringPropertyNames()
```

Пример:

```
Properties map = new Properties();
map.setProperty("login", "mylogin");
map.setProperty("password", "85498742");
System.out.println(map.stringPropertyNames());
// [login, password]
for (String key: map.stringPropertyNames()) {
   System.out.print(key + ": " + map.getProperty(key) + " ");
} // login: mylogin password: 85498742
```

- `store()` — сохраняет словарь в текстовый файл специального формата. Форматы метода:

```
public void store(OutputStream out, String comments)
                                  throws IOException
public void store(Writer writer, String comments)
                                  throws IOException
```

Пример:

```
// import java.io.*;
FileOutputStream fout = null;
try {
   fout = new FileOutputStream("C:\\book\\config.ini");
   Properties map = new Properties();
   map.setProperty("login", "mylogin");
   map.setProperty("password", "85498742");
   map.store(fout, "Comment");
}
catch (IOException e) {
   e.printStackTrace();
}
```

Содержимое файла:

```
#Comment
#Fri Mar 30 11:25:03 MSK 2018
login=mylogin
password=85498742
```

Обратите внимание: файл не может содержать русских букв, т. к. используется кодировка ISO 8859-1. Все русские буквы будут закодированы последовательностями вида `\u041A`. Если нужно использовать другую кодировку, то следует воспользоваться вторым форматом. Сохраним данные в кодировке UTF-8:

```
// import java.io.*;
FileOutputStream fout = null;
OutputStreamWriter writer = null;
try {
   fout = new FileOutputStream("C:\\book\\config2.ini");
   writer = new OutputStreamWriter(fout, "UTF-8");
   Properties map = new Properties();
   map.setProperty("login", "мой логин");
   map.setProperty("password", "мой пароль");
   map.store(writer, "");
}
catch (IOException e) {
   e.printStackTrace();
}
```

- `load()` — **читает данные из файла. Форматы метода:**

```
public void load(InputStream inStream)
                  throws IOException
public void load(Reader reader)
                  throws IOException
```

Пример:

```
// import java.io.*;
FileInputStream fin = null;
try {
   fin = new FileInputStream("C:\\book\\config.ini");
   Properties map = new Properties();
   map.load(fin);
   System.out.println(map.toString());
   // {login=mylogin, password=85498742}
}
catch (IOException e) {
   e.printStackTrace();
}
```

Данные должны быть в кодировке ISO 8859-1. Если используется другая кодировка, то можно воспользоваться вторым форматом. Прочитаем данные в кодировке UTF-8:

```
// import java.io.*;
FileInputStream fin = null;
InputStreamReader reader = null;
try {
   fin = new FileInputStream("C:\\book\\config2.ini");
   reader = new InputStreamReader(fin, "UTF-8");
   Properties map = new Properties();
   map.load(reader);
   System.out.println(map.toString());
```

```
    // {login=мой логин, password=мой пароль}
}
catch (IOException e) {
    e.printStackTrace();
}
```

- `storeToXML()` — **сохраняет словарь в файл в формате XML. Форматы метода:**

```
public void storeToXML(OutputStream os, String comment)
                       throws IOException
public void storeToXML(OutputStream os, String comment,
                       String encoding)
                       throws IOException
public void storeToXML(OutputStream os, String comment,
                       Charset charset)
                       throws IOException
```

Пример:

```
// import java.io.*;
FileOutputStream fout = null;
try {
    fout = new FileOutputStream("C:\\book\\config.xml");
    Properties map = new Properties();
    map.setProperty("login", "мой логин");
    map.setProperty("password", "мой пароль");
    map.storeToXML(fout, "Комментарий", "UTF-8");
}
catch (IOException e) {
    e.printStackTrace();
}
```

Содержимое файла:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM
 "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>Комментарий</comment>
<entry key="login">мой логин</entry>
<entry key="password">мой пароль</entry>
</properties>
```

- `loadFromXML()` — **читает данные из XML-файла. Формат метода:**

```
public void loadFromXML(InputStream in)
       throws IOException, InvalidPropertiesFormatException
```

Пример:

```
// import java.io.*;
FileInputStream fin = null;
```

```
try {
   fin = new FileInputStream("C:\\book\\config.xml");
   Properties map = new Properties();
   map.loadFromXML(fin);
   System.out.println(map.toString());
   // {login=мой логин, password=мой пароль}
}
catch (IOException e) {
   e.printStackTrace();
}
```

ГЛАВА 16

Пакеты, JAR-архивы и модули

Чтобы пользовательский класс можно было использовать в любой программе, необходимо, чтобы этот класс был общедоступным и размещался в отдельном файле с названием, совпадающим с именем класса вплоть до регистра символов. До этой главы мы располагали файлы с классами в той же папке, что и класс с методом `main()`. Теперь представьте, что будет, если мы создадим тысячи классов в одной папке. Во-первых, найти файл с классом будет сложно, даже если список файлов отсортировать по имени. Во-вторых, что делать, если мы хотим использовать классы другого разработчика, у которого могут быть одноименные классы? Ведь разместить одноименные файлы в одной папке нельзя. Чтобы решить эти проблемы и не допустить конфликта имен, в языке Java принято размещать классы внутри *пакетов*. Каждый пакет должен быть расположен внутри отдельной папки. Путь к папке с пакетом должен содержать доменное имя разработчика в сети Интернет в обратном порядке. Например, если домен называется **example.com**, то иерархия папок должна быть следующей:

```
com\
    example\
        классы пакета
```

Так как по определению доменное имя является уникальным, то между классами нескольких разработчиков не возникнет конфликт, — ведь они будут расположены в разных папках.

16.1. Инструкция *import*

Чтобы можно было использовать класс из пакета внутри программы, необходимо импортировать его с помощью инструкции `import`. Мы уже не раз использовали эту инструкцию для подключения классов из библиотеки языка Java. Например, подключали класс `ArrayList<E>`:

```
import java.util.ArrayList;
```

После ключевого слова `import` перед именем класса указывается относительный путь к пакету. Названия папок в пути разделяются с помощью символа «точка».

Например, если бы класс `ArrayList<E>` был расположен в папке com\example, то команда импорта выглядела бы так:

```
import com.example.ArrayList;
```

После импорта класса мы можем обращаться к нему, не указывая путь к пакету:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
```

Если класс расположен в том же пакете, то его не нужно импортировать — он будет найден и без инструкции `import`. Кроме того, не нужно импортировать классы из пакета `java.lang`. Эти классы импортируются автоматически, и мы можем использовать классы `System`, `Math` и др. без инструкции импорта:

```
System.out.println(Math.PI);
```

Инструкция `import` позволяет также импортировать сразу все классы из пакета. В этом случае вместо названия класса указывается символ `*`:

```
import java.util.*;
```

После этой инструкции мы можем использовать все классы из каркаса коллекций, указывая только их имена, а также другие классы, расположенные в этом пакете, — например, класс `Date`:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Date d = new Date();
```

Импорт сразу всех классов из пакета очень удобен, но может произойти конфликт имен. Например, класс `Date` расположен в нескольких пакетах (`java.util` и `java.sql`), и попытка выполнить импорт следующим образом приведет к конфликту имен:

```
import java.util.*;
import java.sql.*;
```

Класс из какого пакета будет использоваться? Редактор Eclipse подчеркнет имя класса. При наведении указателя мыши на имя класса, во всплывающем окне он предложит выбрать один из вариантов: **Explicitly import 'java.sql.Date'** и **Explicitly import 'java.util.Date'**. Например, если мы выберем второй вариант, то редактор автоматически добавит инструкцию импорта:

```
import java.util.*;
import java.util.Date;
import java.sql.*;
```

Теперь конфликта не возникнет, и имя класса `Date` будет связано с пакетом `java.util`. Однако, что делать, если нам нужно использовать оба класса? В этом случае внутри программы перед каждым названием класса `Date` из пакета `java.sql` необходимо добавить путь к пакету:

```
java.sql.Date d = new java.sql.Date(1464124548754L);
```

А классом `Date` из пакета `java.util` можно пользоваться как обычно:

```
Date d2 = new Date();
```

Указывая путь к пакету перед именем класса, можно вообще не импортировать класс, но, согласитесь, что подобный код очень многословен, и набирать его утомительно:

```
java.sql.Date d = new java.sql.Date(1464124548754L);
java.util.Date d2 = new java.util.Date();
```

Редактор Eclipse позволяет преобразовать импорт всех классов из пакета в импорт только используемых в программе классов. Для этого в меню **Source** выбираем пункт **Organize Imports** или нажимаем комбинацию клавиш <Shift>+<Ctrl>+<O>. В результате, например, строка:

```
import java.util.*;
```

будет преобразована в импорт отдельных классов:

```
import java.util.ArrayList;
import java.util.Date;
```

16.2. Импорт статических членов класса

Помимо импорта классов, инструкция `import` позволяет также выполнить импорт статических членов класса. Для этого после ключевого слова `import` вставляется ключевое слово `static`, а затем задается путь к пакету, название класса и после точки указывается символ `*`. Например, выполним импорт всех констант и методов из класса `Math`:

```
import static java.lang.Math.*;
```

Теперь название класса `Math` можно не указывать, хотя можно и указать при необходимости:

```
System.out.println(PI);
System.out.println(Math.PI);
System.out.println(sqrt(25.0));
```

Вместо символа `*` можно указать название статического члена класса. В этом случае будет импортирован только этот член класса. Импортируем только метод `sqrt()` и константу `PI`:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.PI;
```

16.3. Инструкция *package*

Импортировать классы из библиотеки языка Java мы научились, теперь рассмотрим процесс создания собственных пакетов. Для начала создадим пакет в программе Eclipse. Для этого в меню **File** выбираем пункт **New | Package**. В открывшемся окне (рис. 16.1) в поле **Name** вводим имя домена, написанное в обратном порядке (например, `com.example`), а в поле **Source folder** — путь к папке src (например,

Test/src). Нажимаем кнопку **Finish**. В результате в папке Test\src будет создана следующая иерархия папок:

```
com\
    example\
```

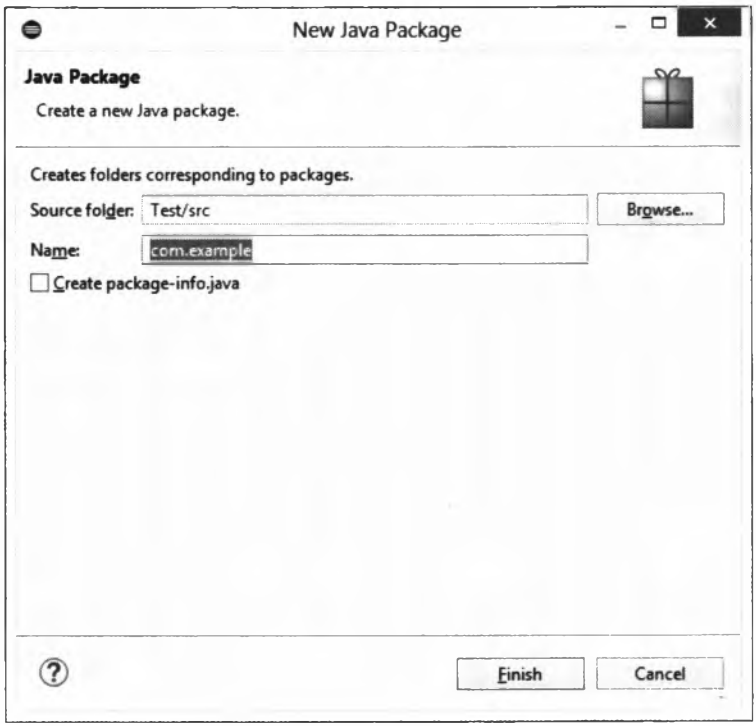

Рис. 16.1. Создание пакета

Теперь добавим в пакет два класса: Class1 и Class2. Для этого в окне **Package Explorer** находим название пакета **com.example**, щелкаем на этом названии правой кнопкой мыши и из контекстного меню выбираем пункт **New | Class**. В открывшемся окне (рис. 16.2) поле **Package** будет автоматически заполнено названием пакета. В поле **Name** вводим название класса (например, Class1), устанавливаем переключатель **public** и нажимаем кнопку **Finish**. В результате в папке Test\src\com\example будет создан файл Class1.java со следующим содержимым:

```
package com.example;

public class Class1 {

}
```

Инструкция package говорит, что класс принадлежит пакету com.example. Эта инструкция должна идти в файле первой, раньше могут быть только комментарии. Если инструкция не указана, то класс будет принадлежать пакету по умолчанию.

Рис. 16.2. Создание класса внутри пакета

Теперь точно так же создаем файл с классом `Class2`, но переключатель устанавливаем в положение **package**, а не **public**. Содержимое файла Class2.java будет выглядеть следующим образом:

```
package com.example;

class Class2 {

}
```

Объявление пакета точно такое же, а вот перед ключевым словом `class` нет модификатора `public`. Давайте попробуем импортировать эти классы в файле с классом `MyClass`:

```
import com.example.Class1; // OK
import com.example.Class2; // Ошибка!
```

При объявлении класса `Class1` был указан модификатор `public`, поэтому этот класс будет успешно импортирован, а вот импорт класса `Class2` закончится ошибкой, т. к. область видимости этого класса ограничена пакетом.

16.4. Пути поиска классов

Редактор Eclipse производит все действия по компиляции файлов пакета незаметно для нас. Давайте попробуем скомпилировать пакеты из командной строки и научимся все это делать самостоятельно. Вначале создадим следующую иерархию из каталогов и исходных файлов:

```
C:\book\
    MyClass.java
    com\
        example\
            Class1.java
            Class2.java
```

Путь до файла MyClass.java должен получиться таким: C:\book\MyClass.java, а до файла Class1.java — C:\book\com\example\Class1.java. Содержимое файла MyClass.java приведено в листинге 16.1, файла Class1.java — в листинге 16.2, а файла Class2.java — в листинге 16.3. Все файлы должны быть в кодировке UTF-8.

Листинг 16.1. Содержимое файла MyClass.java

```
import com.example.Class1;
import java.util.Date;

public class MyClass {
   public static void main(String[] args) {
      Class1 obj = new Class1();
      obj.test();
      Date d = new Date();
      System.out.println(d);
   }
}
```

Листинг 16.2. Содержимое файла Class1.java

```
package com.example;

public class Class1 {
   public static void main(String[] args) {
      Class1 obj = new Class1();
      obj.test();
   }
   public void test() {
      System.out.println("Это Class1");
   }
}
```

Листинг 16.3. Содержимое файла Class2.java

```
package com.example;

public class Class2 {
   public static void main(String[] args) {
      Class2 obj = new Class2();
      obj.test();
   }
   public void test() {
      System.out.println("Это Class2");
   }
}
```

Сначала попробуем скомпилировать файл MyClass.java. Открываем командную строку и переходим в папку C:\book:

```
C:\Users\Unicross>cd C:\book
```

В командной строке отобразится следующее приглашение:

```
C:\book>
```

Если у вас стоит другой путь, то последующие команды работать не будут. Компилируем файл MyClass.java:

```
C:\book>javac -encoding utf-8 MyClass.java
```

Эта команда нам уже знакома. С помощью флага `-encoding` мы указываем кодировку файла, после чего вписываем название компилируемого файла. В результате компиляции рядом с файлом MyClass.java будет создан файл MyClass.class. Если перейти в пакет и посмотреть содержимое папки C:\book\com\example, то можно заметить, что скомпилировался и файл с классом `Class1`. Компилятор самостоятельно нашел этот файл по инструкции `import` внутри файла MyClass.java. Класс `Class2` мы не импортировали, поэтому он так и остался не скомпилированным.

Чтобы скомпилировать все классы внутри пакета `com.example`, следует выполнить следующую команду:

```
C:\book>javac -encoding utf-8 com/example/*.java
```

В этом примере вместо указания конкретного класса вставлен символ `*`, означающий любое название файла с расширением java. Вместо символа `*` можно указать название конкретного класса. Кроме того, перед названием класса или шаблоном следует указать относительный или абсолютный путь к файлам пакета. В нашем случае был указан относительный путь. Теперь все исходные файлы с классами внутри пакета были скомпилированы. Попробуем запустить программу. Запуск файла MyClass.class производится как обычно:

```
C:\book>java MyClass
Это Class1
Fri Mar 30 11:18:50 MSK 2018
```

Все классы внутри пакета `com.example` содержат метод `main()`, поэтому мы их также можем запустить по отдельности. Вообще, любой класс может содержать метод `main()` хотя бы для самотестирования. Чтобы запустить файл из пакета, необходимо перед названием класса указать название пакета. Обратите внимание: не путь к файлу с классом, а название пакета:

```
C:\book>java com.example.Class1
Это Class1

C:\book>java com.example.Class2
Это Class2
```

Все будет хорошо работать, пока текущая папка будет совпадать с папкой поиска классов. Давайте изменим текущую папку и перейдем в папку C:\book\com\:

```
C:\book>cd com

C:\book\com>
```

Начнем с компиляции файлов. Вначале попробуем скомпилировать файл MyClass.java, указав путь к файлу относительно корня диска:

```
C:\book\com>javac -encoding utf-8 /book/MyClass.java
\book\MyClass.java:1: error: package com.example does not exist
import com.example.Class1;
                  ^
\book\MyClass.java:6: error: cannot find symbol
      Class1 obj = new Class1();
      ^
  symbol:   class Class1
  location: class MyClass
\book\MyClass.java:6: error: cannot find symbol
      Class1 obj = new Class1();
                       ^
  symbol:   class Class1
  location: class MyClass
3 errors
```

В результате мы получили ошибку. Путь к исходному файлу мы указали правильно, а вот пути к пакету компилятор не знает и об этом нам и сообщает: `package com.example does not exist` (пакет `com.example` не существует). Если бы мы не импортировали пакет, то программа бы успешно скомпилировалась. Чтобы скомпилировать программу, необходимо дополнительно с помощью флага `-classpath` (или `-cp`) указать путь к папке с классами. Давайте исправим команду:

```
C:\book\com>javac -encoding utf-8 -classpath /book /book/MyClass.java
```

В этой команде мы указали путь к папке C:\book относительно корня диска — теперь все успешно скомпилировалось. Сейчас скомпилируем все файлы пакета `com.example`:

```
javac -encoding utf-8 -classpath /book /book/com/example/*.java
```

Чтобы запустить программу, необходимо опять указать путь к папке с классами явным образом:

```
C:\book\com>java -cp /book MyClass
Это Class1
Fri Mar 30 11:23:07 MSK 2018

C:\book\com>java -cp /book com.example.Class1
Это Class1

C:\book\com>java -classpath /book com.example.Class2
Это Class2
```

Вместо флагов `-classpath` и `-cp` можно предварительно настроить системную переменную `CLASSPATH`, но не на постоянной основе (иначе получите множество проблем в дальнейшем), а только для текущего сеанса. Сделать это можно следующим образом:

```
C:\book\com>set CLASSPATH=/book

C:\book\com>java MyClass
Это Class1
Fri Mar 30 11:24:44 MSK 2018

C:\book\com>java com.example.Class1
Это Class1

C:\book\com>java com.example.Class2
Это Class2
```

Как видно из примера, после настройки системной переменной `CLASSPATH` флаги `-classpath` и `-cp` можно не указывать.

Все бы хорошо, но скомпилированные файлы классов помещаются в одну папку с исходными файлами. Если вы посмотрите, какую структуру каталогов в проекте использует редактор Eclipse, то заметите, что исходные файлы классов расположены в папке src, а скомпилированные файлы классов — в папке bin. Давайте попробуем организовать такое же разделение, но без помощи редактора Eclipse. В папке C:\book создаем две папки: src и bin. Перемещаем файл MyClass.java в папку src. Закрываем командную строку (иначе папка com будет заблокирована) и в эту же папку перемещаем структуру папок пакета. Все скомпилированные файлы классов удаляем, чтобы они нам не мешали. В результате структура каталогов и файлов будет выглядеть так:

```
C:\book\
    bin\
    src\
        MyClass.java
        com\
```

```
example\
    Class1.java
    Class2.java
```

Запускаем командную строку и переходим в папку C:\book:

```
C:\Users\Unicross>cd C:\book

C:\book>
```

Теперь наша задача — скомпилировать файлы таким образом, чтобы исходники остались в папке src, а файлы классов попали в папку bin. Чтобы это сделать, необходимо с помощью флага `-sourcepath` указать корневую папку для исходных файлов, а с помощью флага `-d` — корневую папку с классами:

```
C:\book>javac -encoding utf-8 -sourcepath /book/src -d /book/bin
                                          /book/src/MyClass.java
```

В результате выполнения этой команды внутри папки bin появится скомпилированный файл класса `MyClass`, а также будет создана структура папок пакета, и в папке C:\book\bin\com\example появится скомпилированный файл класса `Class1`. Выполним компиляцию всех классов пакета:

```
C:\book>javac -encoding utf-8 -sourcepath /book/src -d /book/bin
                                    /book/src/com/example/*.java
```

Теперь все классы скомпилированы и находятся в папке bin. Чтобы запустить эти файлы, необходимо с помощью флагов `-classpath` или `-cp` или системной переменной `CLASSPATH` указать путь к корневой папке с классами:

```
C:\book>java -cp /book/bin MyClass
Это Class1
Fri Mar 30 11:37:14 MSK 2018

C:\book>java -classpath /book/bin com.example.Class1
Это Class1

C:\book>set CLASSPATH=/book/bin

C:\book>java com.example.Class2
Это Class2
```

> ***Примечание***
>
> Если файлы классов расположены в разных папках, то пути к классам в Windows указываются через точку с запятой (в UNIX — через двоеточие). Текущая папка обозначается символом «точка».

Если виртуальная машина не найдет импортируемый пакет, то будет сгенерировано исключение `NoClassDefFoundError`. Например, перенесем файл MyClass.class в папку C:\book\bin\com, в командной строке сделаем эту папку текущей и попытаемся запустить файл, указав текущую папку в качестве корневой папки для классов:

```
C:\book\bin\com>java -classpath . MyClass
Exception in thread "main" java.lang.NoClassDefFoundError: com/example/Class1
        at MyClass.main(MyClass.java:6)
Caused by: java.lang.ClassNotFoundException: com.example.Class1
        at java.net.URLClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        ... 1 more
```

Чтобы избежать этой ошибки, всегда правильно указывайте путь к корневой папке с классами, не забывая при этом добавить в путь через точку с запятой и текущую папку (символ «точка») с исполняемым файлом:

```
C:\book\bin\com>java -classpath .;/book/bin MyClass
Это Class1
Fri Mar 30 11:40:39 MSK 2018
```

Итак, если при запуске программы не указан путь к корневой папке классов с помощью флагов `-classpath` или `-cp` или системной переменной `CLASSPATH`, то классы будут искаться в текущей папке. Если указаны флаги, то они задают положение корневой папки классов, при этом текущая папка по умолчанию не добавляется, и значение переменной `CLASSPATH` игнорируется. Если флаги не указаны, то используется значение переменной `CLASSPATH`.

Внутри класса вначале импортируются классы из системного пакета `java.lang` (путь к системным пакетам добавлять в путь поиска классов не нужно), затем по порядку все классы, указанные в инструкциях `import` (при этом учитывается путь поиска классов). Если класс не найден, или при обнаружении одноименных классов возникает конфликт имен, то генерируется исключение.

16.5. JAR-архивы

Файлы классов и пакеты можно помещать в специальные *JAR-архивы*, содержимое которых сжимается с помощью алгоритма ZIP. Помимо классов, эти архивы могут содержать и файлы других типов — например, изображения. Такие файлы называются *ресурсами*. Преимущество JAR-архива состоит в том, что он представляет собой всего один файл небольшого размера вместо целой иерархии папок и файлов пакета. Кроме того, JAR-архив позволяет запускать один класс из архива двойным щелчком на значке архива — правда, при запуске консольного приложения окно консоли в этом случае не отображается, но оконные приложения работают нормально.

16.5.1. Создание JAR-архива

Для создания JAR-архивов служит утилита jar.exe (расположена в папке jdk_версия\bin). Давайте попробуем упаковать наш пакет `com.example` в JAR-архив. Запускаем командную строку и переходим в папку C:\book\bin:

```
C:\Users\Unicross>cd C:\book\bin

C:\book\bin>
```

Теперь создаем JAR-архив:

```
C:\book\bin>jar cvf myjar.jar com/example/*.class
added manifest
adding: com/example/Class1.class(in = 512) (out= 345)(deflated 32%)
adding: com/example/Class2.class(in = 512) (out= 345)(deflated 32%)
```

Вначале указывается название программы, а затем опции: буква `c` означает, что нужно создать новый файл, буква `v` — включает отображение подробного отчета о проделанной работе (три нижние строки и есть отчет), а буква `f` — задает название архива, которое приводится после опций, в нашем случае архив называется `myjar.jar`. После названия архива указываются файлы классов через пробел. В нашем случае мы добавляем в архив все файлы классов из пакета `com.example`.

В результате выполнения программы в папке C:\book\bin будет создан JAR-архив myjar.jar. Открыть и посмотреть содержимое архива позволяет любой ZIP-архиватор — например, 7-Zip. JAR-архив содержит все папки и файлы пакета `com.example`, а также папку META-INF, в которой расположен файл манифеста MANIFEST.MF со следующим содержимым:

```
Manifest-Version: 1.0
Created-By: 10 (Oracle Corporation)
```

Первая строка задает версию манифеста, а вторая строка — версию Java, в которой создан JAR-архив. Помимо этих директив файл манифеста может содержать и другие директивы, а также целые разделы. Обратите внимание: после директивы `Created-By` файл содержит две пустые строки. Файл манифеста обязательно должен в конце содержать пустую строку, иначе будет ошибка.

Добавить в JAR-архив все файлы из каталога C:\book\bin можно так:

```
C:\book\bin>jar cvf myjar2.jar -C /book/bin/ .
```

Теперь попробуем запустить классы из пакета `com.example` внутри JAR-архива:

```
C:\book\bin>java -classpath /book/bin/myjar.jar com.example.Class1
Это Class1

C:\book\bin>java -classpath /book/bin/myjar.jar com.example.Class2
Это Class2
```

Как видно из примера, JAR-архив указывается в качестве корневой папки поиска классов с помощью флага `-classpath`. Далее указывается название класса, перед которым задается название пакета.

16.5.2. Исполняемые JAR-архивы

Как уже говорилось, JAR-архив позволяет запускать один класс из архива путем двойного щелчка на значке архива. Давайте создадим такой исполняемый JAR-архив. Так как для файлов с расширением jar создается ассоциация с программой javaw.exe, окно консоли отображаться не будет, поэтому вместо консольного приложения создадим простейшее оконное приложение. Для этого в папке C:\book создаем файл MyDialog.java с содержимым из листинга 16.4.

Листинг 16.4. Содержимое файла MyDialog.java

```
import javax.swing.JOptionPane;

public class MyDialog {
   public static void main(String[] args) {
   JOptionPane.showMessageDialog(
      null,
      "Все работает!!!", "Заголовок окна",
      JOptionPane.INFORMATION_MESSAGE);
   }
}
```

В командной строке переходим в папку C:\book и компилируем программу:

```
C:\book\bin>cd C:\book

C:\book>javac -encoding utf-8 MyDialog.java
```

Далее создаем исполняемый JAR-архив:

```
C:\book>jar cvfe dialog.jar MyDialog MyDialog.class
added manifest
adding: MyDialog.class(in = 461) (out= 344)(deflated 25%)
```

В этом примере в перечень опций добавлена буква `e`, которая указывает, что JAR-архив является исполняемым. Название класса, метод `main()` которого будет запускаться при двойном щелчке на ярлыке JAR-архива, задается после названия архива. Обратите внимание: название класса указывается без расширения. В самом конце команды задается название класса, добавляемого в архив. В этом случае расширение нужно указать. Файл манифеста исполняемого JAR-архива выглядит следующим образом:

```
Manifest-Version: 1.0
Created-By: 10 (Oracle Corporation)
Main-Class: MyDialog
```

В отличие от манифеста из предыдущего примера, этот манифест содержит дополнительную директиву `Main-Class`, которая задает название класса с методом `main()`. Обратите внимание: название класса и здесь указано без расширения.

После выполнения команды в папке C:\book был создан JAR-архив dialog.jar. Если сделать двойной щелчок на значке этого архива, то отобразится диалоговое окно с заголовком, значком, текстом и кнопкой (рис. 16.3). При нажатии кнопки окно закрывается.

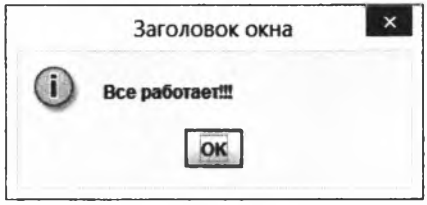

Рис. 16.3. Результат выполнения программы из листинга 16.4

Вместо запуска с помощью двойного щелчка можно запустить такой файл и из командной строки. Это особенно полезно при запуске консольного приложения, не обладающего собственным окном. Для запуска используется следующая команда:

```
C:\book>java -jar /book/dialog.jar
```

В результате отобразится точно такое же диалоговое окно.

16.5.3. Редактирование JAR-архивов

Утилита jar.exe позволяет не только создавать JAR-архивы, но и редактировать их. Давайте отредактируем JAR-архив myjar.jar и сделаем его исполняемым, добавив директиву из собственного файла манифеста. Предварительно в папке C:\book\bin создаем файл манифеста my_manifest.mf со следующим содержимым:

```
Main-Class: com.example.Class1
```

Не забудьте указать пустую строку в конце файла! Теперь добавим эту директиву в файл манифеста JAR-архива myjar.jar:

```
C:\book\bin>jar uvfm myjar.jar my_manifest.mf
updated manifest
```

В этом случае в перечне опций указана буква `u` (вместо буквы `c`), которая означает, что мы хотим модифицировать JAR-архив, а не создать новый. После буквы `f`, задающей название JAR-архива, поставлена буква `m`, которая указывает, что после имени JAR-архива будет задано название файла с манифестом. Директивы из этого файла будут добавлены в существующий внутри JAR-архива файл манифеста. Если такие директивы уже существуют в манифесте, то их значения будут обновлены. Теперь мы можем запустить JAR-архив myjar.jar из командной строки без явного указания класса:

```
C:\book\bin>java -jar /book/bin/myjar.jar
Это Class1
```

Хотя можно указать название класса и явным образом:

```
C:\book\bin>java -classpath /book/bin/myjar.jar com.example.Class2
Это Class2
```

16.5.4. Создание JAR-архива в редакторе Eclipse

Для создания JAR-архива в редакторе Eclipse (предварительно необходимо вставить в файлы Class1.java и Class2.java код из листингов 16.2 и 16.3) в окне **Package Explorer** выделяем название пакета **com.example** и щелкаем на его названии правой кнопкой мыши. Из контекстного меню выбираем пункт **Export**. В открывшемся окне (рис. 16.4) выбираем пункт **Java | JAR file** и нажимаем кнопку **Next**. В следующем окне (рис. 16.5) будет выделен только наш пакет: **com.example**, а если нужно добавить еще что-либо, то достаточно установить флажки напротив нужных пакетов или файлов. Мы специально выбирали пакет в окне **Package Explorer**, чтобы флажки были установлены только для пакета **com.example**. Нажимаем кнопку **Browse** и выбираем нужную папку и название файла — например, папку **C:\book** и название **eclipse_test**. Нажимаем кнопку **Next** два раза. На последнем этапе

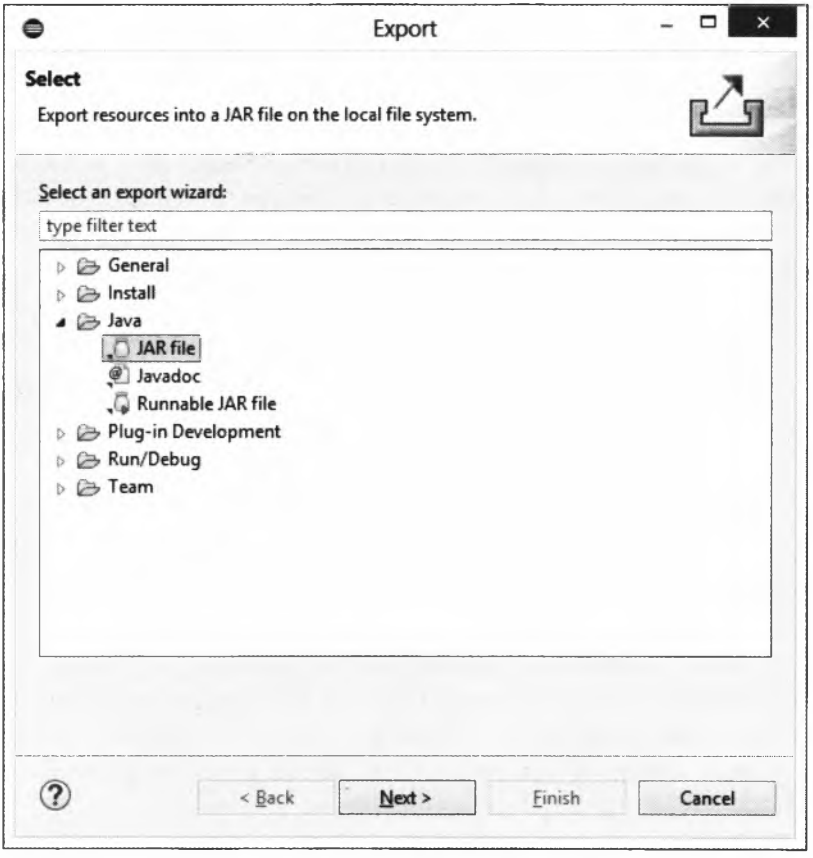

Рис. 16.4. Окно **Export**

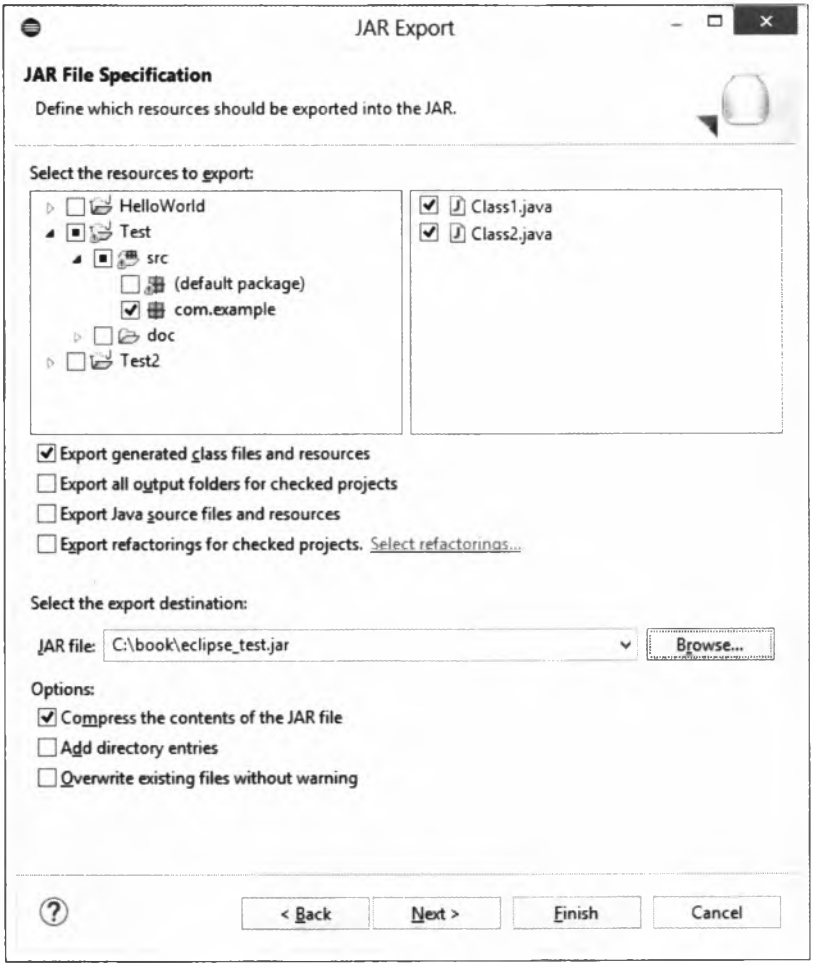

Рис. 16.5. Окно **JAR Export**

(рис. 16.6) в поле **Main class** можно указать название исполняемого класса. Нажимаем кнопку **Browse** и выбираем в открывшемся окне нужный класс — например, **com.example.Class1**. Для создания JAR-архива нажимаем кнопку **Finish**.

В результате этих действий в папке C:\book будет создан файл eclipse_test.jar. Давайте запустим его разными способами:

```
C:\book>java -jar /book/eclipse_test.jar
Это Class1

C:\book>java -classpath /book/eclipse_test.jar com.example.Class1
Это Class1

C:\book>java -classpath /book/eclipse_test.jar com.example.Class2
Это Class2
```

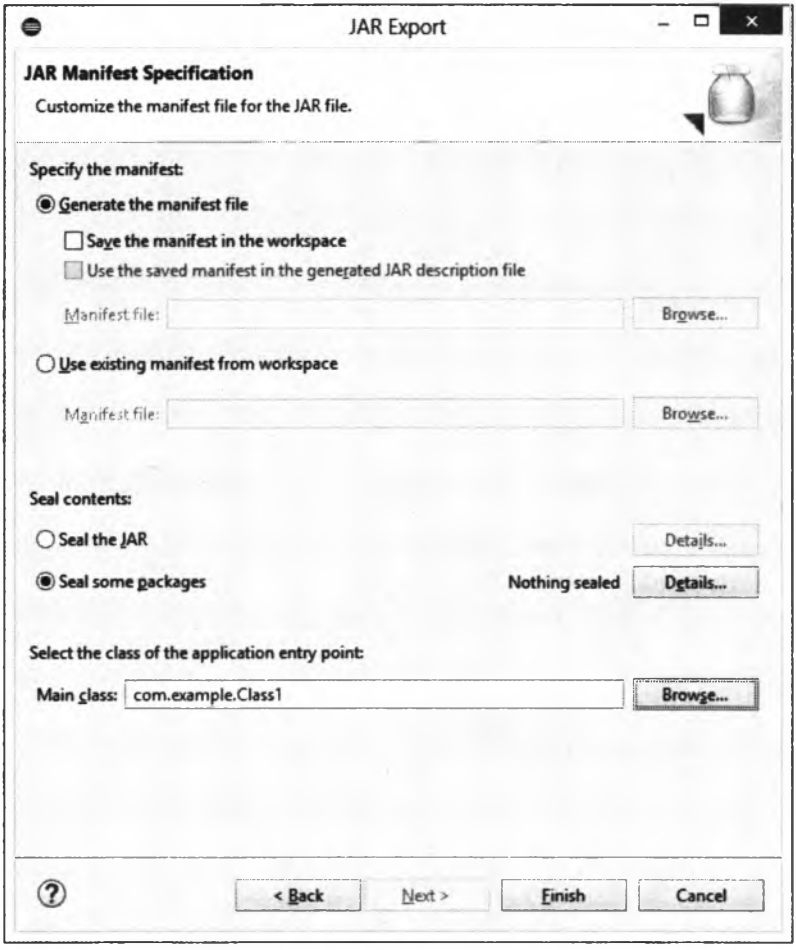

Рис. 16.6. Указание названия исполняемого класса внутри JAR-архива

16.6. Модули

Ключевым нововведением Java 9 является проект Jigsaw, который вводит в язык *модули*. В результате все пакеты стандартной библиотеки языка Java были распределены по модулям с учетом зависимостей. Все классы, которые мы рассматривали в предыдущих главах книги, находятся в модуле с названием `java.base`. Это основной модуль, от которого зависят все остальные модули. Чтобы увидеть полный список модулей стандартной библиотеки языка Java, откройте центральную страницу документации или в командной строке выполните команду `java --list-modules`:

```
C:\book>java --list-modules
java.activation@10
java.base@10
java.compiler@10
... Фрагмент опущен ...
```

```
jdk.zipfs@10
oracle.desktop@10
oracle.net@10
```

16.6.1. Безымянные модули

За время существования языка Java было создано огромное количество пользовательского кода, который ничего не знает о модулях. Такой код, размещенный в путях поиска классов (в CLASSPATH), становится *безымянным модулем*, который имеет доступ ко всем именованным модулям. Поэтому старый код без проблем запускается на новых версиях Java:

```
C:\book\bin>java -classpath /book/bin/myjar.jar com.example.Class1
Это Class1
```

Следует учитывать, что при создании именованных модулей мы не можем добавлять зависимость от безымянного модуля, т. к. у него нет имени.

16.6.2. Автоматические модули

Если JAR-архив со старым кодом разместить в пути поиска модулей, то он становится *автоматическим модулем*. Именем автоматического модуля станет название JAR-архива или фрагмент его названия до первого недопустимого символа. Такой модуль будет экспортировать все свои пакеты и иметь доступ ко всем доступным пакетам других модулей.

Давайте создадим в папке C:\book папку mods и скопируем в нее файл myjar.jar из C:\book\bin. Теперь укажем папку mods в качестве пути поиска модулей и запустим классы из автоматического модуля с названием myjar:

```
C:\book>java --module-path /book/mods --module myjar/com.example.Class1
Это Class1

C:\book>java -p /book/mods -m myjar/com.example.Class2
Это Class2
```

Пути поиска модулей задаются с помощью опции --module-path или -p, а название модуля — с помощью опции --module или -m в формате:

```
--module <Название модуля>[/<Название пакета>.<Название класса>]
```

Если JAR-архив является исполняемым, то можно указать только название модуля:

```
C:\book>java -p /book/mods -m myjar
Это Class1
```

16.6.3. Создание модуля из командной строки

Теперь попробуем создать два именованных модуля. Предварительно в папке C:\book\mods, где мы храним модули, создаем две папки: moduleA и moduleB. В эти папки мы будем складывать скомпилированные файлы.

Далее создаем следующую структуру исходных файлов:

```
C:\book\src\
      moduleA\
            app1\
                  ClassA.java
            app2\
                  ClassB.java
            module-info.java
      moduleB\
            app3\
                  ClassC.java
            module-info.java
```

Модуль `moduleA` содержит два пакета: `app1` и `app2`. В пакете `app1` расположен файл ClassA.java (листинг 16.5), а в пакете `app2` — файл ClassB.java (листинг 16.6). Кроме того, модуль `moduleA` содержит файл module-info.java (листинг 16.7), в котором описываются зависимости от других модулей, экспортируемые пакеты и др.

Листинг 16.5. Содержимое файла C:\book\src\moduleA\app1\ClassA.java

```
package app1;

import app2.ClassB;

public class ClassA {
   public static void main(String[] args) {
      ClassA obj = new ClassA();
      System.out.println(obj.getName());
      ClassB obj2 = new ClassB();
      System.out.println(obj2.getName());
   }
   public String getName() {
      return "Это ClassA";
   }
   public String getMessage() {
      return "Привет из класса ClassA";
   }
}
```

Листинг 16.6. Содержимое файла C:\book\src\moduleA\app2\ClassB.java

```
package app2;

public class ClassB {
   public static void main(String[] args) {
      System.out.println("Это ClassB");
   }
```

```
    public String getName() {
        return "Это ClassB";
    }
}
```

Листинг 16.7. Содержимое файла C:\book\src\moduleA\module-info.java

```
module moduleA {
    requires java.base;
    exports app1;
}
```

Скомпилируем эти файлы в командной строке:

```
C:\book>javac -encoding utf-8 --module-path /book/mods
        --module-version 1.0 -d /book/mods/moduleA
        /book/src/moduleA/module-info.java
        /book/src/moduleA/app1/*.java /book/src/moduleA/app2/*.java

C:\book>
```

В этой команде мы указали следующие опции:

- `-encoding` — задает кодировку исходных файлов;
- `--module-path` — указывает путь поиска модулей;
- `--module-version` — задает версию модуля;
- `-d` — указывает папку, в которую будут записаны скомпилированные файлы.

После опций через пробел указаны файлы, требующие компиляции. В результате компиляции получим следующую структуру файлов и каталогов:

```
C:\book\mods\
      moduleA\
            app1\
                  ClassA.class
            app2\
                  ClassB.class
            module-info.class
```

Попробуем запустить программу из командной строки:

```
C:\book>java --module-path /book/mods --module moduleA/app1.ClassA
Это ClassA
Это ClassB

C:\book>java --module-path /book/mods --module moduleA/app2.ClassB
Это ClassB
```

Модуль `moduleB` содержит пакет `app3`, в котором расположен файл ClassC.java (листинг 16.8). Кроме того, модуль `moduleB` содержит файл module-info.java (листинг 16.9),

в котором описываются зависимости от других модулей, экспортируемые пакеты и др.

Листинг 16.8. Содержимое файла C:\book\src\moduleB\app3\ClassC.java

```
package app3;

import javax.swing.JOptionPane;
import app1.ClassA;

public class ClassC {
   public static void main(String[] args) {
      ClassA obj = new ClassA();
      JOptionPane.showMessageDialog(
         null,
         obj.getMessage(), "Заголовок окна",
         JOptionPane.INFORMATION_MESSAGE);
   }
}
```

Листинг 16.9. Содержимое файла C:\book\src\moduleB\module-info.java

```
module moduleB {
   requires moduleA;
   requires java.desktop;
   exports app3;
}
```

Скомпилируем эти файлы в командной строке:

```
C:\book>javac -encoding utf-8 --module-path /book/mods
        --module-version 1.0 -d /book/mods/moduleB
        /book/src/moduleB/module-info.java
        /book/src/moduleB/app3/*.java

C:\book>
```

В результате компиляции получим следующую структуру файлов и каталогов:

```
C:\book\mods\
      moduleB\
            app3\
                  ClassC.class
            module-info.class
```

Попробуем запустить программу из командной строки:

```
C:\book>java --module-path /book/mods --module moduleB/app3.ClassC
```

В результате отобразится диалоговое окно с приветствием (рис. 16.7).

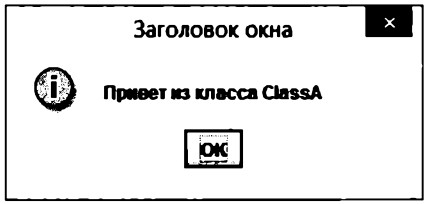

Рис. 16.7. Результат запуска класса `ClassC` **из модуля** `moduleB`

16.6.4. Файл *module-info*

Каждый именованный модуль должен содержать в пакете по умолчанию файл module-info.class, в котором описываются зависимости от других модулей, экспортируемые пакеты и др. До компиляции этот файл имеет расширение java. Структура файла:

```
[open] module <Название модуля> {
   <Инструкции>
}
```

Название модуля должно быть допустимым идентификатором, причем уникальным. Чтобы обеспечить эту уникальность, следует использовать те же правила, что и при именовании пакетов, — например, `com.example.moduleA`. В предыдущем разделе мы сделали имена модулей и пакетов короткими только для удобства и сокращения объема книги.

Внутри фигурных скобок могут быть указаны следующие основные инструкции:

- `requires` — задает название модуля, от которого зависит создаваемый модуль. Формат инструкции:

  ```
  requires [transitive | static] <Название модуля>;
  ```

  Для модуля `moduleA` в файле module-info (см. листинг 16.7) мы указали следующую зависимость:

  ```
  requires java.base;
  ```

  От модуля `java.base` зависят все программы на языке Java, причем сам этот модуль не зависит от других модулей. Так как он является базовым, мы можем его вообще не указывать в файле module-info. Попробуйте удалить эту инструкцию и заново скомпилировать модуль. Никаких проблем не возникнет.

  Для модуля `moduleB` в файле module-info (см. листинг 16.9) мы указали следующие зависимости:

  ```
  requires moduleA;
  requires java.desktop;
  ```

  Модуль `java.desktop` содержит пакеты с классами, позволяющими создавать приложения с графическим интерфейсом. Если мы удалим эту инструкцию, то не сможем использовать эти пакеты и при компиляции получим ошибку:

```
\book\src\moduleB\app3\ClassC.java:3: error: package javax.swing
is not visible
import javax.swing.JOptionPane;
           ^
  (package javax.swing is declared in module java.desktop,
   but module moduleB does not read it)
1 error
```

Внутри класса `ClassC` из модуля `moduleB` мы используем метод `getMessage()` из класса `ClassA`, расположенного в нашем модуле `moduleA`. Поэтому зависимость от модуля `moduleA` также указана с помощью инструкции `requires`;

- `exports` — задает название экспортируемого пакета, расположенного внутри создаваемого модуля. Формат инструкции:

```
exports <Название пакета> [to <Название модуля 1>, ...,
                           <Название модуля N>];
```

Пакеты, указанные в инструкции `exports`, будут доступны для всех других модулей или только для модулей, указанных через запятую после ключевого слова `to`. Если пакет не указан в инструкции `exports`, то он будет недоступен для других модулей.

Для модуля `moduleA` в файле module-info (см. листинг 16.7) мы указали следующую инструкцию экспорта:

```
exports app1;
```

Благодаря этой инструкции, любые модули могут импортировать общедоступные классы из пакета `app1`.

Модуль `moduleA` содержит также пакет `app2`, который не был экспортирован. Классы из этого пакета мы можем использовать внутри модуля `moduleA`, а вот для других модулей пакет будет недоступен. Если в файле с классом `ClassC` добавить следующую инструкцию импорта:

```
import app2.ClassB;
```

то при компиляции получим ошибку:

```
\book\src\moduleB\app3\ClassC.java:5: error: package app2
is not visible
import app2.ClassB;
       ^
  (package app2 is declared in module moduleA, which does
   not export it)
1 error
```

Если после названия пакета задано ключевое слово `to`, то пакет будет доступен только для указанных модулей. Давайте сделаем пакет `app2` доступным только для модулей `moduleB` и `myjar` (автоматический модуль, который мы создали в *разд. 16.6.2*):

```
exports app2 to moduleB, myjar;
```

Помимо инструкций `requires` и `exports`, файл module-info может содержать инструкции `opens` (объявляет пакет открытым), `uses` (описывает зависимость от сервиса) и `provides` (предоставляет сервис):

```
opens <Название пакета> [to <Название модуля 1>, ...,
                         <Название модуля N>];
uses <spiClass>;
provides <spiClass> with <providerClass 1>[, ..., <providerClass N>]
```

16.6.5. Создание модульного JAR-архива

Модульный JAR-архив от обычного отличается наличием файла module-info.class в корне архива. В *разд. 16.6.3* мы создали два модуля, давайте на их основе создадим два модульных JAR-архива из командной строки. Чтобы избежать конфликта имен, сохраним их в папке C:\book\modules. Предварительно эту папку нужно создать.

Вначале создадим файл moduleA.jar:

```
C:\book>jar --create --verbose --file=/book/modules/moduleA.jar
            --module-version=1.0 -C /book/mods/moduleA/ .
added manifest
added module-info: module-info.class
adding: appl/(in = 0) (out= 0)(stored 0%)
adding: appl/ClassA.class(in = 677) (out= 436)(deflated 35%)
adding: app2/(in = 0) (out= 0)(stored 0%)
adding: app2/ClassB.class(in = 498) (out= 326)(deflated 34%)
```

Попробуем его запустить:

```
C:\book>java --module-path /book/modules --module moduleA/appl.ClassA
Это ClassA
Это ClassB

C:\book>java --module-path /book/modules --module moduleA/app2.ClassB
Это ClassB
```

JAR-архив moduleB.jar сделаем исполняемым. Чтобы он мог запускаться с помощью двойного щелчка на значке файла, нужно дополнительно прописать зависимость от moduleA.jar в файле манифеста. Для этого в папке C:\book\src создаем файл MANIFEST.MF со следующим текстом:

```
Manifest-Version: 1.0
Created-By: 10 (Oracle Corporation)
Main-Class: app3.ClassC
Class-Path: moduleA.jar
```

С помощью инструкции `Main-Class` мы указываем исполняемый класс, а с помощью инструкции `Class-Path` — зависимость от moduleA.jar. Не забудьте в конец файла добавить пустую строку, иначе получите ошибку.

Теперь создадим файл moduleB.jar:

```
C:\book>jar --create --verbose --file=/book/modules/moduleB.jar
            --module-version=1.0 --manifest=/book/src/MANIFEST.MF
            -C /book/mods/moduleB/ .
added manifest
added module-info: module-info.class
adding: app3/(in = 0) (out= 0)(stored 0%)
adding: app3/ClassC.class(in = 520) (out= 364)(deflated 30%)
```

Попробуем его запустить:

```
C:\book>java --module-path /book/modules --module moduleB/app3.ClassC
```

В результате отобразится диалоговое окно с приветствием (см. рис. 16.7). Точно такое же окно отобразится при двойном щелчке на значке файла moduleB.jar.

16.6.6. Создание модуля в редакторе Eclipse

Теперь попробуем создать модули в редакторе Eclipse. Вначале создаем два проекта: `moduleA` и `moduleB`. В проект `moduleA` добавляем два пакета: `app1` и `app2`. Далее в пакет `app1` добавляем класс `ClassA` (листинг 16.5), а в пакет `app2` — класс `ClassB` (листинг 16.6). Для создания файла module-info в окне **Package Explorer** щелкаем правой кнопкой мыши на названии проекта **moduleA**. Из контекстного меню выбираем пункт **Configure | Create module-info.java**. В открывшемся окне (рис. 16.8) в поле **Module name** вводим название `moduleA` и нажимаем кнопку **Finish**. Файл module-info.java будет добавлен в пакет по умолчанию, причем все зависимости от других модулей и экспорт всех пакетов проекта прописываются автоматически, и мы получим следующий код:

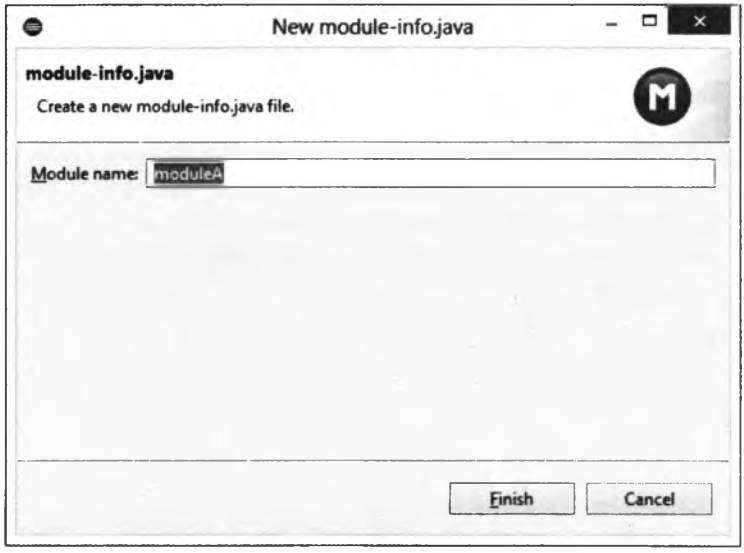

Рис. 16.8. Создание файла module-info.java

```
module moduleA {
   exports app2;
   exports app1;
}
```

Как уже говорилось, зависимость от модуля `java.base` можно не прописывать, поэтому редактор ее и не добавил. Удаляем строку с экспортом пакета `app2` (мы его не хотим экспортировать) и сохраняем файл.

В проекте `moduleB` создаем пакет `app3`. Далее в пакет `app3` добавляем класс `ClassC` (см. листинг 16.8). Для создания файла module-info в окне **Package Explorer** щелкаем правой кнопкой мыши на названии проекта **moduleB**. Из контекстного меню выбираем пункт **Configure | Create module-info.java**. В открывшемся окне в поле **Module name** вводим название `moduleB` и нажимаем кнопку **Finish**. Файл module-info.java будет добавлен в пакет по умолчанию. Содержимое файла, автоматически сгенерированное реактором, выглядит так:

```
module moduleB {
   exports app3;

   requires java.desktop;
}
```

Про системный модуль `java.desktop` редактор знает, а вот про наш модуль `moduleA` — нет. В результате упоминание пакета `app1` при импорте класса `ClassA` внутри файла с классом `ClassC` будет подчеркнуто красной волнистой линией. Чтобы сообщить редактору о существовании этого модуля, в окне **Package Explorer** щелкаем правой кнопкой мыши на названии проекта **moduleB**. Из контекстного меню выбираем пункт **Properties**. В открывшемся окне (рис. 16.9) в списке слева выбираем пункт **Java Build Path** и переходим на вкладку **Projects**. Выделяем пункт **Modulepath** и нажимаем кнопку **Add**. В открывшемся окне (рис. 16.10) устанавливаем флажок **moduleA** и нажимаем кнопку **OK**. Сохраняем изменения.

Теперь переходим в файл с классом `ClassC`, наводим указатель мыши на любой фрагмент, подчеркнутый красной волнистой линией, — например, на название класса `ClassA`, и в открывшемся окне щелкаем на ссылке **Add 'requires moduleA' to module-info.java**. В файл module-info.java будет добавлена зависимость от модуля `moduleA`:

```
module moduleB {
   exports app3;

   requires java.desktop;
   requires moduleA;
}
```

Каждый класс в модулях `moduleA` и `moduleB` содержит метод `main()`, поэтому можно запустить их и посмотреть результат. Все будет работать нормально.

Структура проектов в окне **Package Explorer** показана на рис. 16.11.

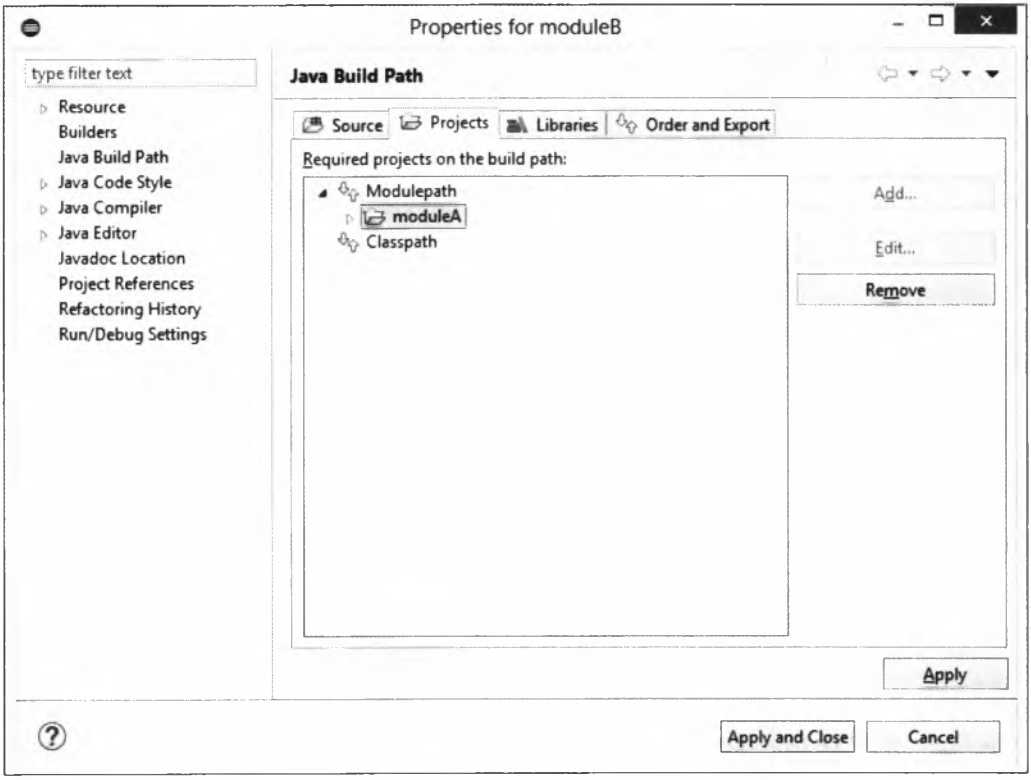

Рис. 16.9. Добавление проекта в путь поиска модулей

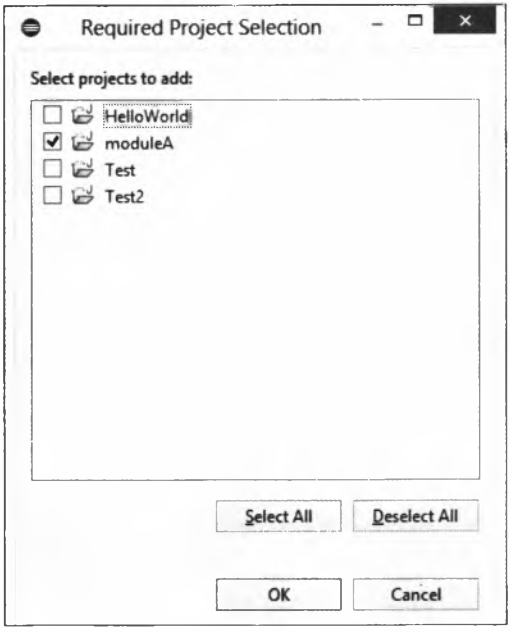

Рис. 16.10. Выбор проекта

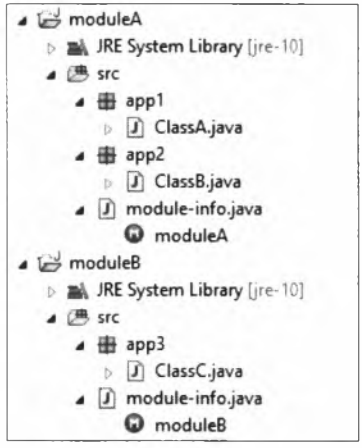

Рис. 16.11. Структура проектов с модулями в окне **Package Explorer**

Теперь создадим модульные JAR-архивы, но вначале удалим все созданные в предыдущем разделе файлы из папки C:\book\modules. Для создания JAR-архива с модулем `moduleA` в окне **Package Explorer** внутри проекта **moduleA** щелкаем правой кнопкой мыши на названии папки **src** и из контекстного меню выбираем пункт **Export**. В открывшемся окне (см. рис. 16.4) выбираем пункт **Java | JAR file** и нажимаем кнопку **Next**. В следующем окне (рис. 16.12) будут выделены все пакеты и файл **module-info.java**. Нажимаем кнопку **Browse** и выбираем нужную папку и название файла: папку **C:\book\modules** и название **moduleA**. Для создания JAR-архива нажимаем кнопку **Finish**. В папке C:\book\modules будет создан файл **moduleA.jar**.

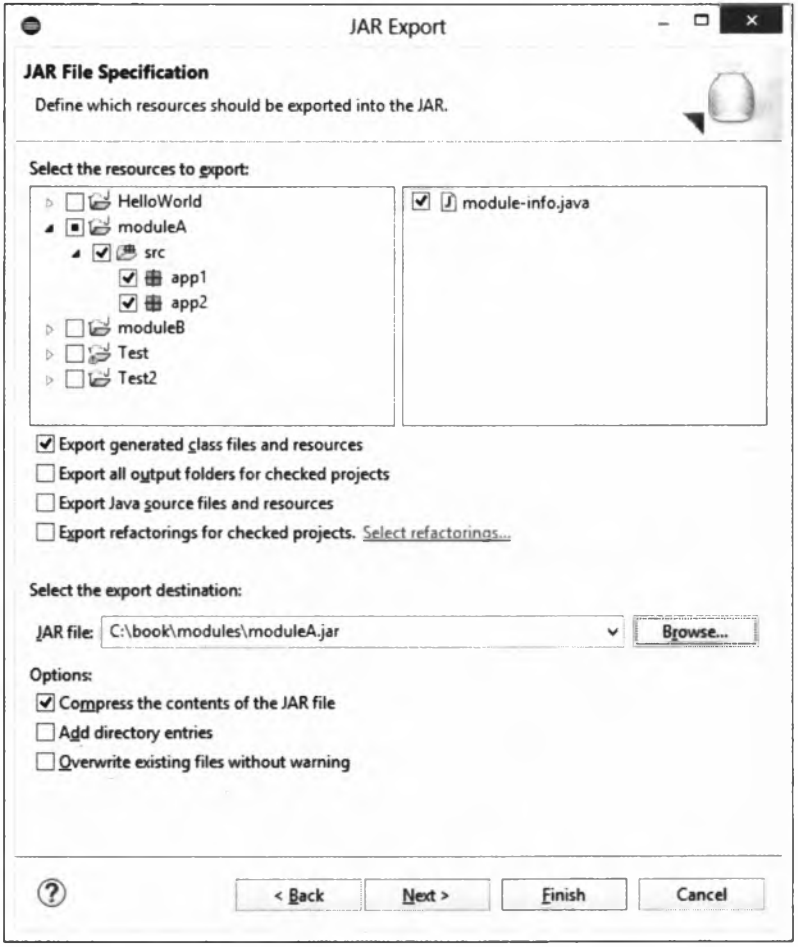

Рис. 16.12. Создание JAR-архива moduleA.jar

Попробуем его запустить из командной строки:

```
C:\book>java --module-path /book/modules --module moduleA/app1.ClassA
Это ClassA
Это ClassB
```

```
C:\book>java --module-path /book/modules --module moduleA/app2.ClassB
Это ClassB
```

Для проекта `moduleB` ранее мы указали зависимость от проекта `moduleA`. Давайте удалим эту зависимость и вместо нее добавим зависимость от созданного файла moduleA.jar. В окне **Package Explorer** щелкаем правой кнопкой мыши на названии проекта **moduleB**. Из контекстного меню выбираем пункт **Properties**. В открывшемся окне (см. рис. 16.9) в списке слева выбираем пункт **Java Build Path** и переходим на вкладку **Projects**. Выделяем пункт **Modulepath | moduleA** и нажимаем кнопку **Remove**. Зависимость от проекта `moduleA` будет удалена. Далее переходим на вкладку **Libraries** и выделяем пункт **Modulepath** (рис. 16.13). Нажимаем кнопку **Add External JARs** и в открывшемся окне находим файл **moduleA.jar**. Сохраняем изменения. Запускаем класс `ClassC` и убеждаемся в работоспособности модуля.

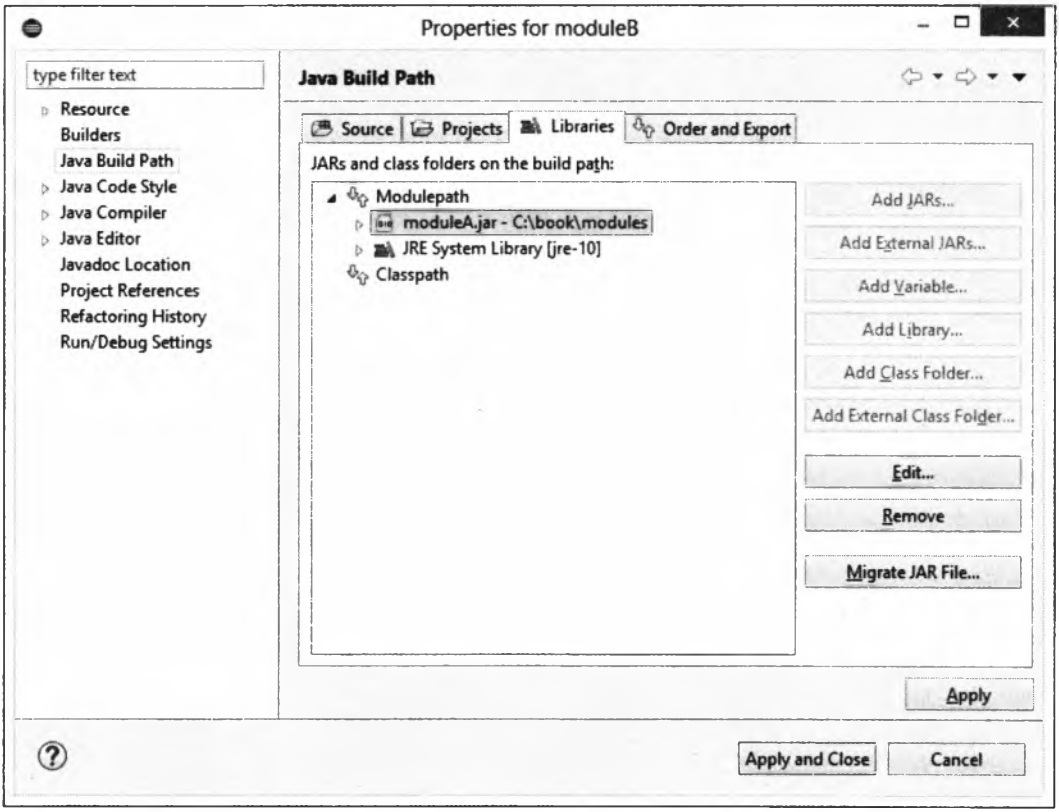

Рис. 16.13. Добавление JAR-архива moduleA.jar в путь поиска модулей

Для создания JAR-архива с модулем `moduleB` в окне **Package Explorer** внутри проекта **moduleB** щелкаем правой кнопкой мыши на названии папки **src** и из контекстного меню выбираем пункт **Export**. В открывшемся окне (см. рис. 16.4) выбираем пункт **Java | JAR file** и нажимаем кнопку **Next**. В следующем окне (рис. 16.14)

будет выделен пакет **app3** и файл **module-info.java**. Нажимаем кнопку **Browse** и выбираем нужную папку и название файла: папку **C:\book\modules** и название **moduleB**. Нажимаем кнопку **Next** два раза. На последнем этапе в поле **Main class** можно указать название исполняемого класса. Нажимаем кнопку **Browse** и в открывшемся окне выбираем класс **app3.ClassC**. Для создания JAR-архива нажимаем кнопку **Finish**. В папке C:\book\modules будет создан файл moduleB.jar.

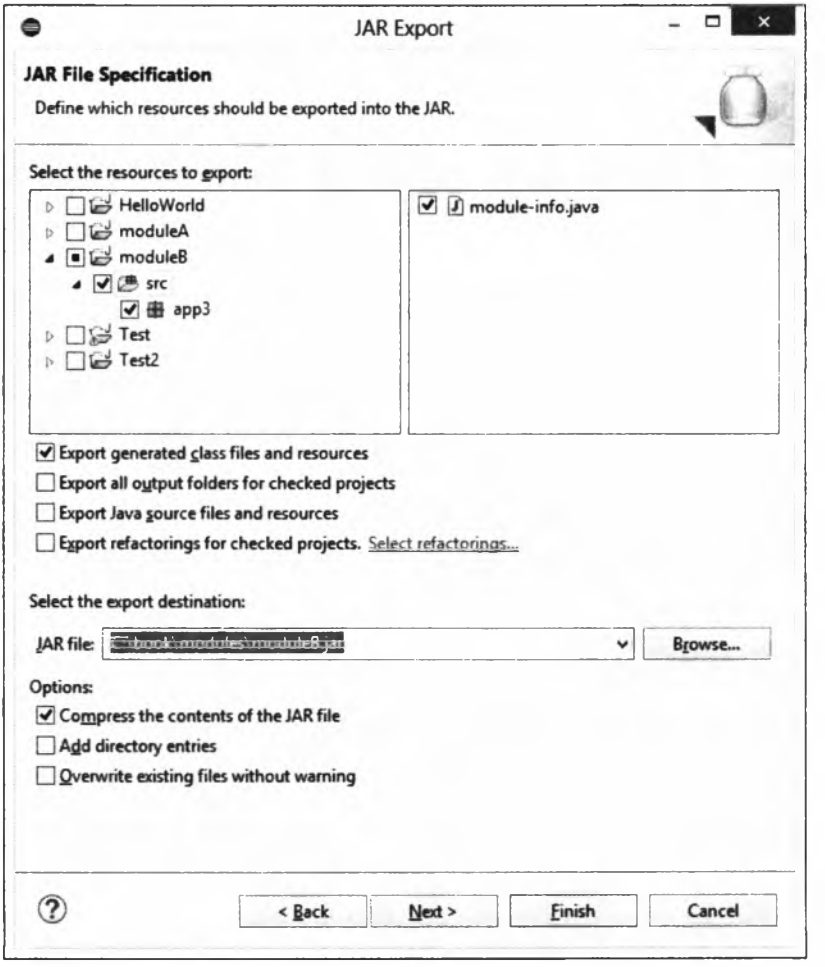

Рис. 16.14. Создание JAR-архива moduleB.jar

Попробуем его запустить из командной строки:

```
C:\book>java --module-path /book/modules --module moduleB/app3.ClassC
```

В результате отобразится диалоговое окно с приветствием (см. рис. 16.7).

Созданный редактором JAR-архив moduleB.jar не содержит указания зависимости от файла moduleA.jar, поэтому с помощью двойного щелчка на значке файла запустить программу не получится. Давайте отредактируем JAR-архив moduleB.jar и добавим

зависимость из командной строки. Предварительно в папке C:\book\src создадим файл MANIFEST.MF со следующим текстом:

```
Class-Path: moduleA.jar
```

Не забудьте в конец файла добавить пустую строку, иначе получите ошибку.

Добавим зависимость из командной строки:

```
C:\book>jar uvfm /book/modules/moduleB.jar /book/src/MANIFEST.MF
updated manifest
updated module-info: module-info.class
```

Теперь программу можно запускать двойным щелчком на значке файла moduleB.jar.

ГЛАВА 17

Обработка ошибок

Если вы когда-нибудь учились водить автомобиль, то наверняка вспомните, что при первой посадке на водительское сиденье все внимание было приковано к трем деталям: рулю, педалям и рычагу переключения передач. Происходящее вне автомобиля уходило на второй план, т. к. вначале нужно было стронуться с места. По мере практики навыки вождения улучшались, и на задний план постепенно уходили уже эти три детали. Как ни странно, но руль и рычаг переключения передач всегда оказывались там, куда вы, не глядя, протягивали руки, а ноги сами находили педали. Теперь все внимание стало занимать происходящее на дороге. Иными словами, вы стали опытным водителем.

В программировании все абсолютно так же. Начинающие программисты больше обращают внимание на первые попавшиеся на глаза операторы, методы и другие элементы языка, а сам алгоритм уходит у них на задний план. Если программа скомпилировалась без ошибок, уже большое счастье, хотя это еще не означает, что программа будет работать правильно. По мере практики мышление программиста меняется, он начинает обращать внимание на мелочи, на форматирование программы, использует более эффективные алгоритмы и в результате всего этого допускает меньше ошибок. Подводя итоги, можно сказать, что начинающий программист просто пишет программу, а опытный программист пытается найти оптимальный алгоритм и предусмотреть поведение программы в различных ситуациях. Однако от ошибок никто не застрахован, поэтому очень важно знать, как быстро найти ошибку.

17.1. Типы ошибок

Существуют три типа ошибок в программе:

- *синтаксические* — это ошибки в имени оператора или метода, отсутствие закрывающей или открывающей кавычек и т. д., т. е. ошибки в синтаксисе языка. Как правило, компилятор предупредит о наличии ошибки, а программа не будет выполняться совсем. Пример синтаксической ошибки (нет точки с запятой после первой инструкции):

```
Date d = new Date()
System.out.println(d);
```

При компиляции будет выведено следующее сообщение об ошибке:

```
Exception in thread "main" java.lang.Error:
      Unresolved compilation problem:
      Syntax error, insert ";"
      to complete LocalVariableDeclarationStatement

      at com.example.mymodule/com.example.app.MyClass.main(
         MyClass.java:8)
```

Цифра внутри круглых скобок (`MyClass.java:8`) обозначает номер строки, в которой обнаружена ошибка. Если после попытки скомпилировать программу в окне **Console** редактора Eclipse (открыть окно можно из меню **Window**, выбрав пункт **Show View | Console**) выполнить щелчок мышью на строке внутри круглых скобок (`MyClass.java:8`), то инструкция с ошибкой станет активной. Кроме того, строка с синтаксической ошибкой подсвечивается редактором красной волнистой линией еще до компиляции программы, а слева от инструкции отображается красный круг с белым крестом внутри. Так что обращайте внимание на различные подчеркивания кода. При синтаксических ошибках подчеркивание будет красного цвета, а при предупреждениях (например, переменная объявлена и инициализирована, но не используется) — желтого. При предупреждениях слева от инструкции дополнительно отображается значок в виде желтой лампочки и треугольника с восклицательным знаком. При наведении указателя мыши на подчеркнутый фрагмент или на значок, расположенный слева, отобразятся текст описания проблемы и возможные способы ее исправления.

Все предупреждения (например, о том, что переменная не используется) можно посмотреть в окне **Problems** (рис. 17.1). Если окно не отображается, то в меню **Window** выберите пункт **Show View | Problems**;

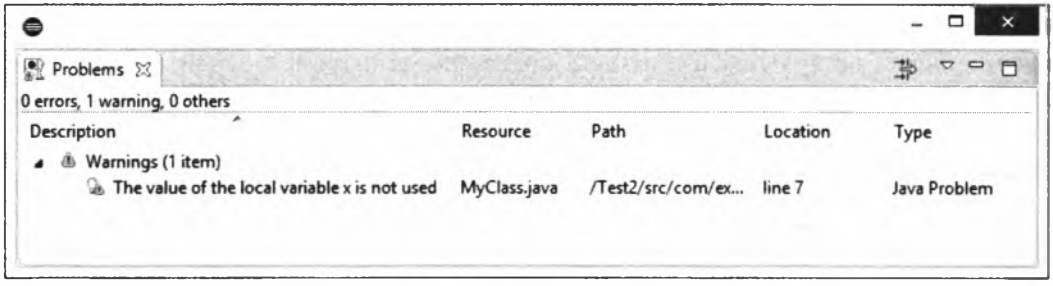

Рис. 17.1. Окно Problems

- *логические* — это ошибки в логике работы программы, которые можно выявить только по результатам ее работы. Как правило, компилятор не предупреждает о наличии такой ошибки, а программа будет выполняться, т. к. не содержит синтаксических ошибок. Логические ошибки весьма трудно выявить, и их поиск часто заканчивается бессонными ночами;

- *ошибки времени выполнения* — это ошибки, которые возникают во время работы программы. В одних случаях ошибки времени выполнения являются следствием логических ошибок, а в других их вызывают внешние события — например, отсутствие прав для записи в файл или то, что файл не найден (пользователь взял и удалил файл, посчитав, что он не нужен), и пр. В этом случае виртуальная машина генерирует исключение. Если в коде не предусмотрена обработка исключения, то программа прерывается и выводится сообщение об ошибке.

17.2. Инструкция *try...catch...finally*

Для обработки исключений предназначена инструкция `try...catch`. Формат инструкции:

```
try {
    <Блок, в котором перехватывается исключение>
}
[catch (<Класс исключения 1> <Переменная>) {
    <Блок, выполняемый при возникновении исключения>
}
[...
catch (<Класс исключения N> <Переменная>) {
    <Блок, выполняемый при возникновении исключения>
}]]
[finally {
    <Блок, выполняемый в любом случае>
}]
```

Инструкции, в которых перехватываются исключения, должны быть расположены внутри блока `try`. Если при выполнении этих инструкций возникнет исключение, то управление будет передано в блок `catch`, который соответствует классу исключения. Классом исключения может выступать встроенный класс или пользовательский класс. Обратите внимание: если исключение не возникло, то инструкции внутри блока `catch` не выполняются.

Начиная с Java 7, вместо нескольких блоков `catch` можно использовать один блок `catch` с несколькими исключениями, записанными через символ `|`. В этом случае управление будет передано в блок `catch` при генерации любого из указанных исключений:

```
try {
   //...
}
catch (NullPointerException | DateTimeException e) {
   System.out.println(
          "NullPointerException или DateTimeException");
}
```

В качестве примера использования инструкции `try...catch` инсценируем деление на `0` и обработаем его:

```
int x = 0, y = 10;
try {
   y = y / x;
   System.out.println("y = " + y); // Выполнена не будет
}
catch (ArithmeticException e) {
   System.out.println("Нельзя делить на 0");
}
```

При возникновении исключения внутри блока `try` управление сразу передается в блок `catch`, соответствующий классу возникшего исключения. В нашем примере классом исключения является встроенный класс `ArithmeticException`. Код, расположенный после инструкции, сгенерировавшей исключение, выполнен не будет, поэтому вывод значения переменной `y` мы не увидим. После выполнения инструкций в блоке `catch` управление передается инструкции, расположенной сразу после инструкции `try...catch`. Иными словами, считается, что исключение обработано, и можно продолжить выполнение программы.

В некоторых случаях необходимо не продолжить выполнение программы, а прервать ее выполнение после перехвата исключения. Например, если продолжать работу не имеет смысла. В этом случае можно внутри блока `catch` произвести завершающие действия (например, проинформировать пользователя о проблеме), а затем прервать работу программы с помощью метода `exit()` из класса `System`:

```
int x = 0, y = 10;
try {
   y = y / x;
}
catch (ArithmeticException e) {
   System.out.println("Нельзя делить на 0");
   System.exit(1); // Прерываем выполнение программы
}
```

Если нет блока `catch`, соответствующего классу исключения, то исключение «всплывает» к обработчику более высокого уровня. Если исключение нигде не обрабатывается в программе, то управление передается обработчику по умолчанию, который останавливает выполнение программы и выводит стандартную информацию об ошибке. Таким образом, в обработчике может быть несколько блоков `catch` с разными классами исключений. Кроме того, один обработчик можно вложить в другой:

```
// import java.time.DateTimeException;
int x = 0, y = 10;
try {
   try {
      y = y / x;
   }
   catch (NullPointerException e) {
      System.out.println("NullPointerException");
   }
```

```
   catch (DateTimeException e) {
      System.out.println("DateTimeException");
   }
   System.out.println("Инструкция после вложенного обработчика");
}
catch (ArithmeticException e) {      // Выполняется этот блок!
   System.out.println("Нельзя делить на 0");
   y = 0;
}
System.out.println("y = " + y); // y = 0
```

В этом примере во вложенном обработчике не указано исключение `ArithmeticException`, поэтому исключение «всплывает» к обработчику более высокого уровня. После обработки исключения управление передается инструкции, расположенной сразу после обработчика. В нашем примере управление будет передано инструкции, выводящей значение переменной `y`. Обратите внимание на то, что инструкция:

```
System.out.println("Инструкция после вложенного обработчика");
```

выполнена не будет.

При указании нескольких блоков `catch` с разными классами исключений следует учитывать иерархию классов исключений. В предыдущем примере все указанные классы являются наследниками класса `RuntimeException`. Если мы укажем этот базовый класс во вложенной инструкции `try...catch`, то сможем перехватить ошибку деления на `0`:

```
int x = 0, y = 10;
try {
   try {
      y = y / x;
   }
   catch (NullPointerException e) {
      System.out.println("NullPointerException");
   }
   catch (RuntimeException e) {            // Выполняется этот блок!
      System.out.println("RuntimeException");
      y = 0;
   }
   System.out.println("Инструкция после вложенного обработчика");
}
catch (ArithmeticException e) {
   System.out.println("Нельзя делить на 0");
   y = 0;
}
System.out.println("y = " + y); // y = 0
```

Результат выполнения будет выглядеть следующим образом:

```
RuntimeException
Инструкция после вложенного обработчика
y = 0
```

Если в одной инструкции `try...catch` находятся блоки `catch` с базовым и производным классами, то вначале должен идти блок с производным классом, а затем блок с базовым, иначе блок с производным классом никогда не будет выполнен. В нашем случае вначале идет блок с классом `NullPointerException`, а лишь затем блок с классом `RuntimeException`.

Если в блоке `catch` указан самый верхний класс в иерархии классов исключений, то он будет перехватывать исключения любых производных классов. Самым верхним классом в иерархии является класс `Throwable`, но его указывать в блоке `catch` не следует, т. к. существуют ошибки, которые пользовательский код обработать никак не может. Обычно указывается его наследник, класс `Exception`:

```
int x = 0, y = 10;
try {
   y = y / x;
}
catch (Exception e) {
   System.out.println("Обработка любых ошибок в коде");
}
```

Получить информацию об обрабатываемом исключении можно через переменную, объявленную в блоке `catch`:

```
int x = 0, y = 10;
try {
   y = y / x;
}
catch (ArithmeticException e) {
   System.out.println(e.toString());
   System.out.println(e.getMessage());
   System.out.println(e.getClass().getName());
   e.printStackTrace();
   y = 0;
}
```

Результат выполнения:

```
java.lang.ArithmeticException: / by zero
/ by zero
java.lang.ArithmeticException
java.lang.ArithmeticException: / by zero
       at com.example.mymodule/com.example.app.MyClass.main(
          MyClass.java:12)
```

Помимо блоков `catch`, инструкция `try...catch` может содержать блок `finally`, инструкции внутри которого выполняются вне зависимости от того, возникло в блоке

try исключение или нет. Обычно внутри этого блока выполняется освобождение каких-либо ресурсов — например, закрытие файлов:

```
int x = 0, y = 10;
try {
   try {
      y = y / x;
   }
   catch (NullPointerException e) {
      System.out.println("NullPointerException");
   }
   finally {
      System.out.println("finally. Блок 1");
   }
   System.out.println("Инструкция после вложенного обработчика");
}
catch (ArithmeticException e) {
   System.out.println("Нельзя делить на 0");
}
finally {
   System.out.println("finally. Блок 2");
}
```

Результат выполнения:

```
finally. Блок 1
Нельзя делить на 0
finally. Блок 2
```

Если существует блок finally, то блоков catch может не быть вовсе:

```
int x = 0, y = 10;
try {
   y = y / x;
}
finally {
   System.out.println("Блок finally");
}
```

Результат выполнения:

```
Блок finally
Exception in thread "main" java.lang.ArithmeticException: / by zero
       at com.example.mymodule/com.example.app.MyClass.main(
          MyClass.java:12)
```

В качестве примера использования инструкции try...catch переделаем нашу программу суммирования произвольного количества целых чисел, введенных пользователем (см. листинг 3.8), таким образом, чтобы при вводе строки вместо числа программа не завершалась с фатальной ошибкой (листинг 17.1). А также предусмотрим возможность выхода из программы при трех неудачных попытках ввода подряд.

Листинг 17.1. Пример использования инструкции `try...catch`

```
package com.example.app;

import java.util.Scanner;

public class MyClass {
   public static void main(String[] args) {
      int x = 0, result = 0, count = 0;
      String s = "";
      Scanner in = new Scanner(System.in);
      System.out.println(
         "Введите 'stop' для получения результата");
      while (true) {
         System.out.print("Введите число: ");
         s = in.nextLine();
         if (s.equals("stop")) break;
         try {
            x = Integer.parseInt(s);
            if (x == 0) break;
            result += x;
            count = 0;
         }
         catch (NumberFormatException e) {
            System.out.println("Необходимо ввести целое число!");
            count++;
            if (count > 3) break; // Три попытки подряд
         }
      }
      System.out.println("Сумма чисел равна: " + result);
   }
}
```

Процесс ввода значений и получения результата выглядит так (значения, введенные пользователем, выделены здесь полужирным шрифтом):

```
Введите 'stop' для получения результата
Введите число: 10
Введите число: str
Необходимо ввести целое число!
Введите число: -5
Введите число:
Необходимо ввести целое число!
Введите число: stop
Сумма чисел равна: 5
```

17.3. Оператор *throw*: генерация исключений

Вместо обработки исключения можно повторно сгенерировать исключение внутри блока `catch`, чтобы передать управление вышестоящему обработчику. Например, можно сгенерировать исключение более высокого уровня. Кроме того, если мы разрабатываем какой-либо метод и внутри него не имеем возможности вернуть корректное значение при некорректном значении параметров, то лучше сгенерировать исключение, чтобы другие пользователи знали о проблеме. Например, метод возвращает целое число. Какое значение мы бы могли вернуть при ошибке? Значение `-1`? Но, это также число. Генерация исключения в этом случае лучшее решение.

Для генерации исключения в программе предназначен оператор `throw`. После оператора указывается объект класса исключения. Это может быть встроенный класс исключений или пользовательский класс, наследующий один из классов встроенных исключений. Давайте создадим два обработчика, один из которых вложен в другой. Внутри блока `try` вложенного обработчика имитируем деление на `0`. Внутри блока `catch` вложенного обработчика перехватим это исключение и сгенерируем его повторно. Повторное исключение перехватим в блоке `catch` внешнего обработчика:

```
int x = 0, y = 10;
try {
   try {
      y = y / x;
   }
   catch (ArithmeticException e) {
      throw new ArithmeticException("Нельзя делить на 0");
   }
   System.out.println("Инструкция после вложенного обработчика");
}
catch (ArithmeticException e) {
   System.out.println(e.getMessage());
   y = 0;
}
System.out.println("y = " + y); // y = 0
```

Обратите внимание на инструкцию:

```
throw new ArithmeticException("Нельзя делить на 0");
```

Здесь мы создаем экземпляр класса `ArithmeticException` и передаем конструктору класса строку с описанием ошибки. Эту строку мы можем получить внутри блока `catch` внешнего обработчика через переменную `e` с помощью метода `getMessage()`.

17.4. Иерархия классов исключений

Все встроенные классы исключений являются наследниками класса `Throwable`. У этого класса есть два непосредственных наследника: `Error` и `Exception`. Класс `Error` описывает ошибки, связанные с нехваткой ресурсов, и внутренние ошибки.

Обработка таких ошибок в программе либо сильно ограничена, либо вообще невозможна. Класс `Exception` описывает ошибки внутри программы. Именно этот класс следует указывать, если вы хотите перехватить все исключения внутри программы. У класса `Exception` есть два основных наследника: `RuntimeException` и `IOException`. Класс `RuntimeException` описывает ошибки внутри программы, а класс `IOException` — ошибки, возникающие при работе с внешними устройствами (например, при работе с файлами). Если мы в блоке `catch` указываем имя базового класса, то блок будет перехватывать исключения всех производных классов.

Класс `Throwable` содержит следующие конструкторы:

```
Throwable()
Throwable(String message)
Throwable(String message, Throwable cause)
Throwable(Throwable cause)
```

Первый конструктор создает объект без дополнительной информации об ошибке:

```
Throwable ex = new Throwable();
System.out.println(ex.getMessage()); // null
```

Второй конструктор позволяет указать описание ошибки:

```
Throwable ex = new Throwable("Описание ошибки");
System.out.println(ex.getMessage()); // Описание ошибки
```

Третий конструктор во втором параметре принимает объект класса `Throwable` с информацией о предыдущей ошибке:

```
int x = 0;
try {
   x = (x + 1) / x;
}
catch (Exception e) {
   Throwable ex = new Throwable("Описание ошибки", e);
   System.out.println(ex.getCause().getMessage()); // / by zero
   System.out.println(ex.getMessage());            // Описание ошибки
}
```

Четвертый конструктор создает объект на основе другого объекта класса `Throwable`:

```
int x = 0;
try {
   x = (x + 1) / x;
}
catch (Exception e) {
   Throwable ex = new Throwable(e);
   System.out.println(ex.getCause().getMessage()); // / by zero
   System.out.println(ex.getMessage());
   // java.lang.ArithmeticException: / by zero
}
```

Обычно при создании пользовательских классов исключений следует наследовать не класс `Throwable`, а один из его наследников: `Exception` или `RuntimeException`.

Если наследовать класс `Exception`, то исключения будут *контролируемыми*, а если `RuntimeException` — то *неконтролируемыми*. Конструкторы этих классов аналогичны конструкторам класса `Throwable`.

Класс `Throwable` содержит следующие основные методы (полный список методов смотрите в документации):

- `toString()` — возвращает класс исключения и, после двоеточия, описание ошибки (если оно имеется). Формат метода:

```
public String toString()
```

Пример:

```
Throwable ex = new Throwable("Описание");
System.out.println(ex.toString());
// java.lang.Throwable: Описание
```

- `getMessage()` и `getLocalizedMessage()` — возвращают описание ошибки или значение `null`, если описания нет. Форматы методов:

```
public String getMessage()
public String getLocalizedMessage()
```

Пример:

```
Throwable ex = new Throwable("Описание");
System.out.println(ex.getMessage());          // Описание
System.out.println(ex.getLocalizedMessage()); // Описание
```

- `getCause()` — возвращает объект с информацией о предыдущей ошибке или значение `null`. Формат метода:

```
public Throwable getCause()
```

Пример:

```
int x = 0;
try {
   x = (x + 1) / x;
}
catch (Exception e) {
   Throwable ex = new Throwable("Описание", e);
   System.out.println(ex.getCause().getMessage()); // / by zero
}
```

- `initCause()` — позволяет указать объект класса `Throwable` с информацией о предыдущей ошибке. Если объект уже был добавлен ранее (например, с помощью конструктора), то генерируется исключение. Метод можно вызвать только один раз, повторный вызов приведет к исключению. Формат метода:

```
public Throwable initCause(Throwable cause)
```

Пример:

```
int x = 0;
try {
   x = (x + 1) / x;
}
catch (Exception e) {
   Throwable ex = new Throwable("Описание").initCause(e);
   System.out.println(ex.getCause().getMessage()); // / by zero
}
```

- `printStackTrace()` — **печатает стек вызовов. Форматы метода:**

```
public void printStackTrace()
public void printStackTrace(PrintStream s)
public void printStackTrace(PrintWriter s)
```

Пример:

```
int x = 0;
try {
   x = (x + 1) / x;
}
catch (Exception e) {
   e.printStackTrace(System.out);
}
```

Результат:

```
java.lang.ArithmeticException: / by zero
    at com.example.mymodule/com.example.app.MyClass.main(
       MyClass.java:10)
```

- `getStackTrace()` — **возвращает массив со стеком вызовов. Формат метода:**

```
public StackTraceElement[] getStackTrace()
```

Пример:

```
int x = 0;
try {
   x = (x + 1) / x;
}
catch (Exception e) {
   StackTraceElement[] st = e.getStackTrace();
   for (StackTraceElement elem: st) {
      System.out.println(elem.getFileName());   // MyClass.java
      System.out.println(elem.getClassName());
      // com.example.app.MyClass
      System.out.println(elem.getModuleName());
      // com.example.mymodule
      System.out.println(elem.getMethodName()); // main
```

```
        System.out.println(elem.getLineNumber()); // 10
        System.out.println(elem.toString());
        // com.example.mymodule/com.example.app.MyClass.main(
        //     MyClass.java:10)
    }
}
```

17.5. Типы исключений

Все классы исключений делятся на два типа: *неконтролируемые* и *контролируемые*. К неконтролируемым исключениям относятся классы, наследующие класс `Error` или `RuntimeException`. Все остальные классы относятся к контролируемым исключениям. При генерации неконтролируемых исключений никаких дополнительных действий не требуется, а вот при генерации контролируемых исключений необходимо в объявлении метода после списка параметров указать классы контролируемых исключений через запятую после ключевого слова `throws`:

```
// import java.io.IOException;
// import java.net.URISyntaxException;
public static void test(int x) throws IOException,
                                      URISyntaxException {
   if (x == 0) {
      throw new IOException();
   }
   else if (x < 0) {
      throw new URISyntaxException("", "");
   }
   // Что-то делаем
}
```

Метод, конечно, искусственный, но главное здесь увидеть, как объявляются несколько классов контролируемых исключений и как генерируются исключения в программе.

Любой код, использующий этот метод, должен сделать выбор между двумя действиями: обработать исключение с помощью инструкции `try...catch` или передать его вверх по иерархии вызовов, указав класс исключения в заголовке метода после ключевого слова `throws`. Подобный выбор нам придется часто делать в следующих главах книги при изучении работы с файлами и потоками.

При наследовании и переопределении метода базового класса, не содержащего объявления контролируемых исключений, нельзя генерировать контролируемые исключения внутри этого переопределенного метода. Вы обязаны обработать все эти исключения, т. к. поведение переопределенного метода изменить нельзя. Однако можно сгенерировать неконтролируемое исключение и добавить в него информацию о предыдущей ошибке.

17.6. Пользовательские классы исключений

Как вы уже знаете, классом исключения может выступать встроенный класс исключения или пользовательский класс, наследующий встроенный класс исключения. Основное преимущество использования классов для обработки исключений заключается в возможности указания базового класса для перехвата всех исключений соответствующих классов-потомков. Например, если пользовательский класс наследует стандартный класс `RuntimeException`, то, указав в блоке `catch` объект класса `RuntimeException`, можно перехватить исключение пользовательского класса. Обратите внимание на то, что блок `catch`, в котором указан объект производного класса, должен быть расположен перед блоком `catch`, в котором указан объект базового класса.

При разработке пользовательских классов исключений обычно наследуется либо класс `Exception` (контролируемые исключения), либо класс `RuntimeException` (неконтролируемые исключения). В качестве примера создадим пользовательский класс исключения, наследующий класс `RuntimeException` (листинг 17.2).

Листинг 17.2. Пользовательские классы исключений

```
package com.example.app;

public class MyClass {
   public static void main(String[] args) {
      try {
         throw new MyException("Описание ошибки");
      }
      catch (MyException e) {
         System.out.println(e.getMessage());
      }
   }
}

@SuppressWarnings("serial")
class MyException extends RuntimeException {
   public MyException() {
      super("Ошибка класса MyException");
   }

   public MyException(String s) {
      super("Ошибка класса MyException\n" + s);
   }

   public MyException(String s, Throwable cause) {
      super("Ошибка класса MyException\n" + s, cause);
   }
}
```

Результат выполнения:

```
Ошибка класса MyException
Описание ошибки
```

17.7. Способы поиска ошибок в программе

В предыдущих разделах мы научились обрабатывать ошибки времени выполнения. Однако наибольшее количество времени программист затрачивает на другой тип ошибок — логические ошибки. В этом случае программа компилируется без ошибок, но результат выполнения программы не соответствует ожидаемому результату. Ситуация еще более осложняется, когда неверный результат проявляется лишь периодически, а не постоянно. Инсценировать такую же ситуацию, чтобы получить этот же неверный результат, бывает крайне сложно и занимает очень много времени. В этом разделе мы рассмотрим лишь «дедовские» (но по-прежнему актуальные) способы поиска ошибок, а также дадим несколько советов по оформлению кода, что будет способствовать быстрому поиску ошибок.

Первое, на что следует обратить внимание, — это форматирование кода. Начинающие программисты обычно не обращают на форматирование никакого внимания, считая этот процесс лишним. А на самом деле зря! Компилятору абсолютно все равно, разместите вы все инструкции на одной строке или выполните форматирование кода. Однако при поиске ошибок форматирование кода позволит найти ошибку гораздо быстрее.

Перед всеми инструкциями внутри блока должно быть расположено одинаковое количество пробелов. Обычно используется три или четыре пробела. От применения символов табуляции лучше отказаться. Если все же их используете, то не следует в одном файле совмещать и пробелы, и табуляцию. Для вложенных блоков количество пробелов умножают на уровень вложенности: если для блока первого уровня вложенности использовалось три пробела, то для блока второго уровня вложенности поставьте шесть пробелов, для третьего уровня — девять пробелов, и т. д. Пример форматирования вложенных блоков приведен в листинге 17.3.

Листинг 17.3. Пример форматирования вложенных блоков

```
int[][] arr =
{
   {1, 2, 3, 4},
   {5, 6, 7, 8}
};
int i = 0, j = 0;
for (i = 0; i < arr.length; i++) {
   for (j = 0; j < arr[i].length; j++) {
      System.out.printf("%3s", arr[i][j]);
   }
   System.out.println();
}
```

Открывающая фигурная скобка может быть расположена как на одной строке с оператором, так и на следующей строке. Какой способ использовать, зависит от предпочтений программиста или от требований по оформлению кода, принятых внутри фирмы. Пример размещения открывающей фигурной скобки на отдельной строке:

```
for (i = 0; i < arr.length; i++)
{
   for (j = 0; j < arr[i].length; j++)
   {
      System.out.printf("%3s", arr[i][j]);
   }
   System.out.println();
}
```

Одна строка кода не должна содержать более 80 символов. Если количество символов больше, то следует выполнить переход на новую строку. При этом продолжение смещается относительно основной инструкции на величину отступа или выравнивается по какому-либо элементу. Иначе строка не помещается на стандартном экране и приходится пользоваться горизонтальной полосой прокрутки, а это очень неудобно при поиске ошибок.

Если программа слишком большая, то следует задуматься о разделении ее на отдельные методы или классы, которые выполняют логически законченные действия. Помните, что отлаживать отдельный метод гораздо легче, чем «спагетти»-код. Причем, прежде чем вставить метод (или класс) в основную программу, его следует протестировать в отдельном проекте, передавая методу различные значения и проверяя результат его выполнения. Не забывайте также, что любой класс может содержать метод `main()`, внутри которого можно выполнять тестирование класса.

Обратите внимание на то, что форматирование кода должно выполняться при написании кода, а не во время поиска ошибок. Этим вы сократите время поиска ошибки и, скорее всего, заметите ошибку еще на этапе написания. Если все же ошибка возникла, то вначале следует инсценировать ситуацию, при которой ошибка проявляется. После этого можно начать поиск ошибки.

Причиной периодических ошибок чаще всего являются внешние данные. Например, если числа получаются от пользователя, а затем производится деление чисел, то вполне возможна ситуация, при которой пользователь введет число `0`. Деление на ноль приведет к ошибке. Следовательно, все данные, которые поступают от пользователей, должны проверяться на соответствие допустимым значениям. Если данные им не соответствуют, то нужно вывести сообщение об ошибке, а затем повторно запросить новое число или прервать выполнение всей программы. Кроме того, нужно обработать возможность того, что пользователь может ввести вовсе не число, а строку. Пример получения числа от пользователя с проверкой корректности данных был показан в листинге 17.1.

Метод `println()` объекта `System.out` удобно использовать для вывода промежуточных значений. В этом случае значения переменных вначале выводятся в самом на-

чале программы и производится проверка соответствия значений. Если значения соответствуют, то инструкция с методом `println()` перемещается на следующую строку программы, опять производится проверка и т. д. Если значения не совпали, то ошибка возникает в инструкции, расположенной перед инструкцией с методом `println()`. Если ошибка в пользовательском методе, то проверку значений производят внутри метода, каждый раз перемещая инструкцию с выводом значений. На одном из этих многочисленных этапов ошибка обычно обнаруживается. В больших программах можно логически догадаться о примерном расположении инструкции с ошибкой и начать поиск ошибки оттуда, а не с самого начала программы.

17.8. Протоколирование

Недостатком использования метода `println()` при поиске ошибок является необходимость убрать все инструкции после нахождения ошибки. При повторном появлении ошибки придется опять расставлять инструкции для вывода промежуточных результатов, а потом снова их удалять. Вместо использования метода `println()` можно воспользоваться средствами протоколирования, предоставляемыми классом `Logger` из пакета `java.util.logging`. Мы можем расставить инструкции вывода промежуточных значений уже на этапе написания кода. В самом начале программы можно включить или отключить протоколирование одной инструкцией. Давайте рассмотрим возможность протоколирования на примере (листинг 17.4).

Листинг 17.4. Протоколирование

```
package com.example.app;

import java.util.logging.FileHandler;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.logging.SimpleFormatter;

public class MyClass {
   public static void main(String[] args) {
      // Отключение протоколирования
      //Logger.getLogger("mylog").setLevel(Level.OFF);
      try {
         Logger mylog = Logger.getLogger("mylog");
         FileHandler fh = new FileHandler("C:\\book\\mylog.txt");
         // Сокращенный формат
         fh.setFormatter(new SimpleFormatter());
         mylog.addHandler(fh);
         mylog.setUseParentHandlers(false);
      }
      catch (Exception e) {
         e.printStackTrace();
      }
```

```
        int[][] arr =
        {
           {1, 2, 3, 4},
           {5, 6, 7, 8}
        };
        int i = 0, j = 0;
        for (i = 0; i < arr.length; i++) {
           for (j = 0; j < arr[i].length; j++) {
              System.out.printf("%3s", arr[i][j]);
              Logger.getLogger("mylog").log(
                 Level.INFO, "i = " + i + "; j = " + j);
           }
           System.out.println();
        }
     }
}
```

Пакет `java.util.logging` входит в состав модуля `java.logging`, поэтому в файле module-info нужно прописать зависимость от этого модуля:

```
module com.example.mymodule {
   requires java.logging;
   exports com.example.app;
}
```

Объект регистратора протокола создается при первом вызове по имени. В нашем случае мы создаем регистратор с именем `mylog`:

```
Logger mylog = Logger.getLogger("mylog");
```

В дальнейшем при указании этого имени метод `getLogger()` будет возвращать тот же самый объект регистратора, поэтому нет смысла сохранять объект в какой-либо переменной и передавать в различные методы. В следующей инструкции создается объект обработчика, и конструктору класса передается путь и имя файла, в который будут записываться отладочные данные:

```
FileHandler fh = new FileHandler("C:\\book\\mylog.txt");
```

По умолчанию запись производится в XML-формате. В результате мы получим множество записей в таком виде:

```
<record>
  <date>2018-04-02T22:57:29.078268200Z</date>
  <millis>1522709849078</millis>
  <nanos>268200</nanos>
  <sequence>0</sequence>
  <logger>mylog</logger>
  <level>INFO</level>
  <class>com.example.app.MyClass</class>
  <method>main</method>
```

```
  <thread>1</thread>
  <message>i = 0; j = 0</message>
</record>
```

Очень подробная информация. Чтобы сократить ее, мы передаем в обработчик объект класса `SimpleFormatter`:

```
fh.setFormatter(new SimpleFormatter());
```

Теперь каждая запись будет выглядеть так:

```
апр. 03, 2018 11:59:30 ДП com.example.app.MyClass main
INFO: i = 0; j = 0
```

В таком формате проще разбираться, да и места он занимает меньше. Запись информации в лог производится с помощью метода `log()`, но предварительно необходимо получить объект регистратора протокола:

```
Logger.getLogger("mylog").log(Level.INFO, "i = " + i + "; j = " + j);
```

Чтобы отключить вывод отладочной информации, достаточно убрать комментарий перед инструкцией:

```
Logger.getLogger("mylog").setLevel(Level.OFF);
```

Пакет `java.util.logging` содержит множество классов, позволяющих выполнить протоколирование в различных форматах. За подробными сведениями обращайтесь к документации.

17.9. Инструкция *assert*

Инструкция `assert` генерирует исключение `AssertionError`, если логическое выражение возвращает значение `false`. Формат инструкции:

```
assert <Логическое выражение>[: <Данные>]
```

Если указан необязательный параметр `<Данные>`, то эти данные будут доступны как описание исключения.

Инструкции `assert` мы можем также расставить на этапе написания кода, без необходимости удаления после отладки. Если программа выполняется как обычно, то эти инструкции игнорируются. Чтобы включить режим отладки, необходимо при запуске программы указать флаг `-enableassertions` (или `-ea`):

```
C:\book>java -enableassertions MyClass
```

После флага можно указать двоеточие и задать название класса или пакета. В этом случае режим отладки включается только для указанного класса или пакета. С помощью флага `-disableassertions` (или `-da`) можно отключить режим отладки для указанного после двоеточия класса или пакета:

```
C:\book>java -ea -da:Class1 MyClass
```

Давайте рассмотрим использование инструкции `assert` на примере (листинг 17.5).

Листинг 17.5. Инструкция assert

```
import java.util.Date;

public class MyClass {
   public static void main(String[] args) {
      Date d = new Date();
      System.out.println(d);
      MyClass.test(-5);
   }
   public static void test(int x) {
      assert x > 0: x + " <= 0";
      System.out.println("test()");
   }
}
```

Сохраняем код в файле MyClass.java в папке C:\book. Запускаем командную строку и выполняем следующие команды:

```
C:\book>javac -encoding utf-8 MyClass.java

C:\book>java MyClass
Tue Apr 03 12:05:37 MSK 2018
test()

C:\book>java -ea MyClass
Tue Apr 03 12:06:20 MSK 2018
Exception in thread "main" java.lang.AssertionError: -5 <= 0
        at MyClass.test(MyClass.java:10)
        at MyClass.main(MyClass.java:7)
```

Как видно из примера, исключение возбуждается только при указании флага `-ea`. При обычном запуске инструкция `assert` игнорируется. Обратите внимание на значение, указанное после названия исключения:

```
java.lang.AssertionError: -5 <= 0
```

Это результат выполнения выражения в параметре `<Данные>` (`x + " <= 0"`).

17.10. Отладка программы в редакторе Eclipse

Сделать поиск ошибок более эффективным позволяет отладчик, встроенный в редактор Eclipse. С его помощью можно выполнять программу по шагам, при этом контролируя значения переменных на каждом шаге. Отладчик позволяет также проверить, соответствует ли порядок выполнения инструкций разработанному ранее алгоритму. В качестве примера мы будем отлаживать программу из листинга 17.6, которая содержит две наиболее часто встречающиеся ошибки: выход за пределы массива и нумерация элементов массива с `1`, а не с `0`.

Листинг 17.6. Отладка программы

```
package com.example.app;                          // 1
public class MyClass {                            // 2
   public static void main(String[] args) {       // 3
      int[][] arr =                               // 4
      {                                           // 5
         {1, 2, 3, 4},                            // 6
         {5, 6, 7, 8}                             // 7
      };                                          // 8
      int i = 0, j = 0;                           // 9
      for (i = 0; i <= arr.length; i++) {         // 10
         for (j = 1; j < arr[i].length; j++) {    // 11
            MyClass.print(arr[i][j]);             // 12
         }                                        // 13
         System.out.println();                    // 14
      }                                           // 15
   }                                              // 16
   public static void print(int x) {              // 17
      String s = String.format("%3s", x);         // 18
      System.out.print(s);                        // 19
   }                                              // 20
}                                                 // 21
```

Прежде чем начать отладку, необходимо пометить строки внутри программы с помощью *точек останова*. Для добавления точки останова делаем строку активной, а затем в меню **Run** выбираем пункт **Toggle Breakpoint** или нажимаем комбинацию клавиш <Shift>+<Ctrl>+<B>, — слева от строки появится кружок, обозначающий точку останова. Повторное действие убирает точку останова. Добавить точку останова можно еще быстрее. Для этого достаточно выполнить двойной щелчок левой кнопкой мыши слева от строки. Повторный двойной щелчок позволяет удалить точку останова. Чтобы изменить свойства точки, временно отключить или удалить ее, следует щелкнуть на ней правой кнопкой мыши и в открывшемся контекстном меню выбрать пункт **Toggle Breakpoint**, **Disable Breakpoint** или **Enable Breakpoint**. Давайте установим в нашей программе две точки останова: напротив строк 9 и 12.

Когда точки останова расставлены, можно начать отладку. Для этого в меню **Run** выбираем пункт **Debug** или нажимаем клавишу <F11>. После запуска отладки интерфейс редактора Eclipse может поменяться кардинальным образом (рис. 17.2). При достижении точки останова выполнение программы прерывается, и отладчик ожидает дальнейших действий программиста. Инструкция, которая будет выполняться на следующем шаге, помечается стрелкой слева от строки. Чтобы прервать отладку, в меню **Run** выбираем пункт **Terminate** или нажимаем комбинацию клавиш <Ctrl>+<F2>.

> ***Обратите внимание!***
>
> Если на компьютере установлена антивирусная программа или включен брандмауэр Windows, то они могут заблокировать запуск отладчика. Необходимо разрешить запуск

отладчика при запросе действия или добавить программу в список исключения брандмауэра, если отладчик не запустился.

После запуска отладчика выполнение программы прервется на строке 9 — в том месте, где установлена первая точка останова. В режиме прерывания можно посмотреть значения различных переменных. Для этого достаточно навести указатель мыши на название переменной. Попробуйте навести указатель на название массива `arr` — в нижней части всплывающего окна отобразятся значения всех элементов массива. Если же навести указатель мыши на переменные `i` и `j`, то никакого значения мы не получим, т. к. поток управления еще не дошел до этих переменных. Их значение мы увидим только после следующего шага.

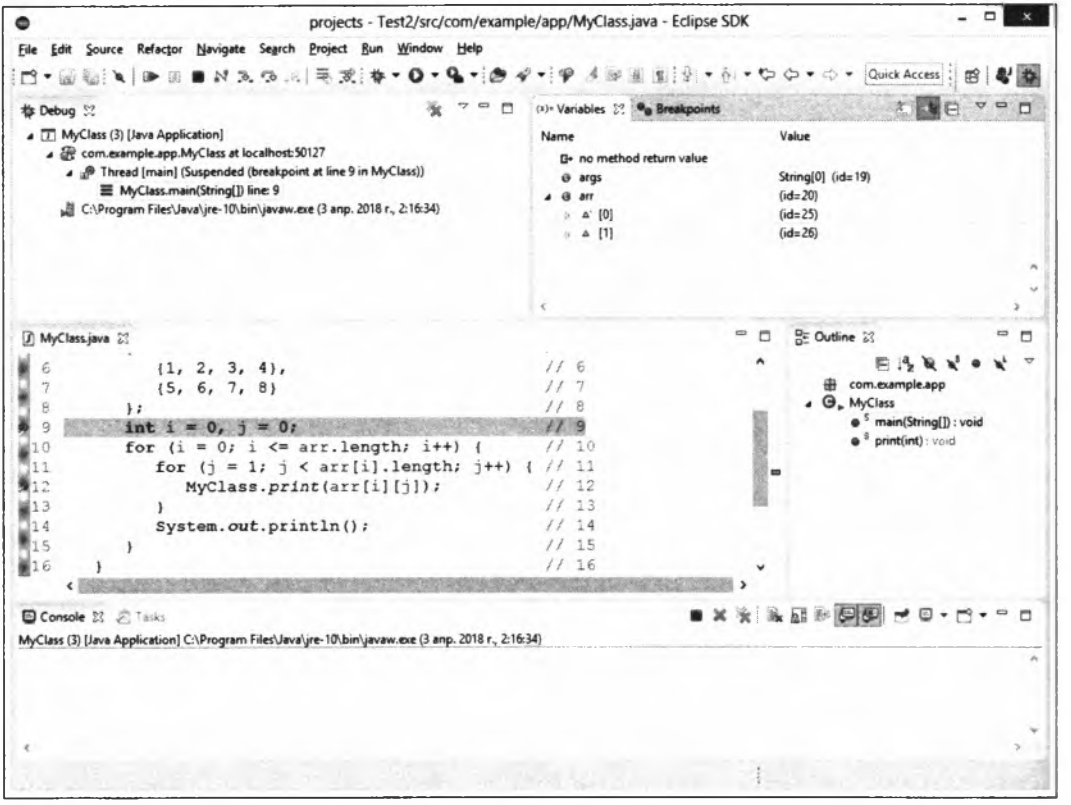

Рис. 17.2. Окно редактора Eclipse после запуска отладки

Посмотреть значения сразу всех переменных можно в окне **Variables** (рис. 17.3). Если окно не отображается, то отобразить его можно выбрав в меню **Window** пункт **Show View | Variables**. При отладке можно контролировать значения отдельных переменных, а не всех сразу. Для этого следует добавить название переменной в окне **Expressions** (рис. 17.4), щелкнув левой кнопкой мыши на надписи **Add new expression**. Чтобы отобразить это окно, в меню **Window** выбираем пункт **Show View | Expressions**. Можно также выделить название переменной и из контекстного меню выбрать пункт **Watch**. Давайте добавим в этом окне переменные `i` и `j`. Сей-

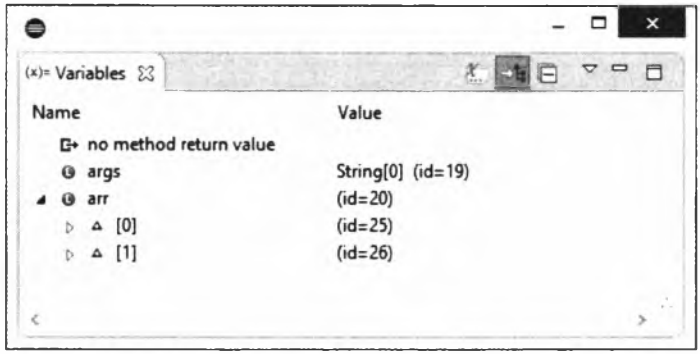

Рис. 17.3. Окно **Variables**

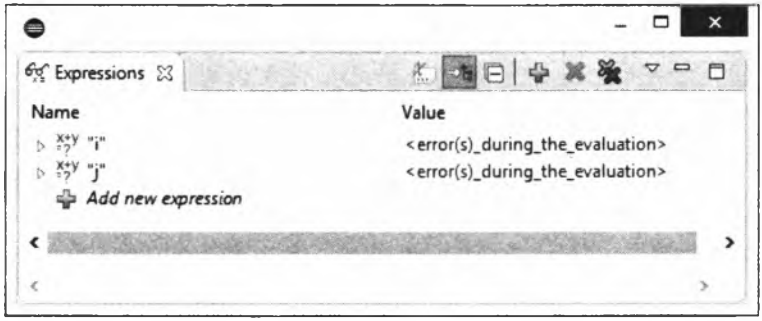

Рис. 17.4. Окно **Expressions**

час значения этих переменных не определены, поэтому в поле **Value** отображаются красные надписи.

Для пошагового выполнения программы предназначены следующие пункты в меню **Run** или соответствующие кнопки на панели инструментов:

- **Step Into** (клавиша <F5>) — выполняет одну инструкцию. Давайте сделаем один шаг и посмотрим на значения переменных `i` и `j` в окне **Expressions** — обе переменные стали видны и получили значение `0`. Делаем еще два шага и останавливаемся на строке 12, которая у нас помечена точкой останова. Если мы сделаем еще один шаг, то попадем внутрь метода `print()`. Обратите внимание на значения переменных `i` и `j` в окне **Expressions** — они опять стали не определены, т. к. мы находимся в области видимости метода `print()`;
- **Step Return** (клавиша <F7>) — позволяет сразу выйти из метода. Если мы не находимся внутри метода, то команда будет недоступна. Давайте нажмем клавишу <F7> и выйдем из метода `print()`. В результате текущей станет строка 11;
- **Step Over** (клавиша <F6>) — выполняет одну инструкцию. Если в этой инструкции производится вызов метода, то метод выполняется, и отладчик переходит в режим ожидания после выхода из метода. Давайте сделаем два шага и опять окажемся на строке 11, не входя в метод `print()`;

- **Resume** (клавиша <F8>) — выполняет инструкции до следующей точки останова или до конца программы при отсутствии точек. Нажимая клавишу <F8>, мы каждый раз будем останавливаться на строке 12 и сможем наблюдать за текущими значениями переменных `i` и `j` в окне **Expressions**, а также за выводом результатов работы программы в окне **Console**. На одном из шагов можно обнаружить, что значение переменной `i` стало равно `2`, и сразу после следующего шага возникает исключение `ArrayIndexOutOfBoundsException`. Если вы не успели этого заметить, то запустите отладку еще раз и наблюдайте за значениями переменных. Так как значение переменной `i` стало равно `2`, а в следующей строке идет обращение к массиву по этому индексу и возникает исключение, следовательно, что-то не так со значением переменной `i`. Смотрим внимательно на предыдущую инструкцию и обнаруживаем неправильное условие окончания цикла в строке 10 (`i <= arr.length`). Обратите внимание: в описании исключения стоит строка с номером 11, а реальная ошибка в строке 10. Исправляем ошибку, и выражение примет вид `i < arr.length`. Опять запускаем отладку, чтобы убедиться в правильности внесенных изменений;
- **Run to Line** (комбинация клавиш <Ctrl>+<R>) — выполняет инструкции до текущей строки при условии, что между инструкциями нет точки останова. Если точка есть, то выполнение будет идти до этой точки. Если мы попробуем перейти сразу к строке 14, то это не получится, т. к. существует точка останова на строке 12. А нам бы хотелось увидеть вывод одной строки массива сразу. Для этого можно временно отключить точку останова, щелкнув на ней правой кнопкой мыши и выбрав в контекстном меню пункт **Disable Breakpoint** (в отключенном состоянии отображается лишь контур кружка без заливки, для повторной активации выбираем пункт **Enable Breakpoint**). Если у нас точек будет много, то каждый раз выбирать пункт из контекстного меню станет утомительно. Можно временно отключить сразу все точки останова, выбрав в меню **Run** пункт **Skip All Breakpoints** (в отключенном состоянии точки отображаются перечеркнутыми, повторное действие сделает все точки останова снова активными). После этого действия можно сразу перейти к строке 14, предварительно сделав ее текущей и нажав комбинацию клавиш <Ctrl>+<R>. После этого действия мы заметим, что вместо четырех значений выводятся только три. Чтобы понять причину, опять выполняем программу по шагам и наблюдаем за значениями переменных в окне **Expressions**. В один прекрасный момент понимаем, что индексы нумеруются с `0`, а не с `1`, и исправляем ошибку в строке 11 (`j = 0` вместо `j = 1`);
- **Terminate** (комбинация клавиш <Ctrl>+<F2>) — останавливает отладку.

Управлять отдельными точками останова или всеми сразу можно в окне **Breakpoints** (рис. 17.5). Чтобы отобразить это окно, следует в меню **Window** выбрать пункт **Show View | Breakpoints**. В окне **Breakpoints** можно изменить свойства точки останова. Например, чтобы указать, что точка должна срабатывать только на *N*-й раз, следует в окне **Breakpoints** выделить строку с точкой, установить флажок **Hit count** и ввести число в текстовое поле. Это очень удобно, если необходимо пропустить часть итераций.

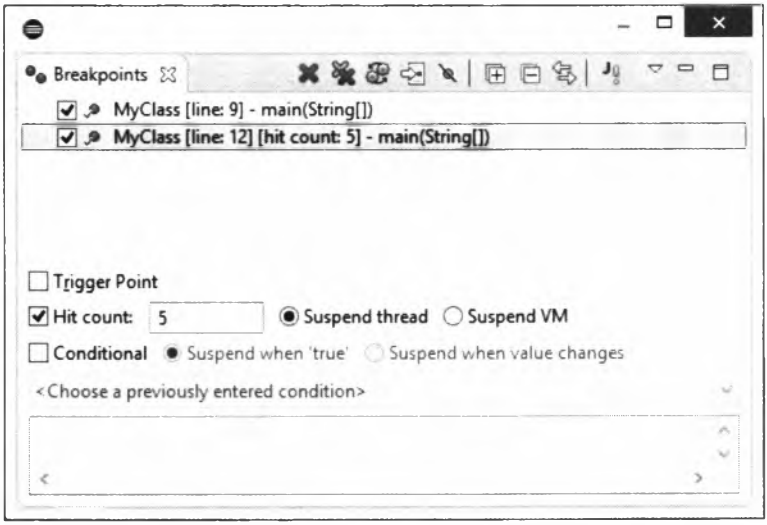

Рис. 17.5. Окно **Breakpoints**

Применение отладки — это самый эффективный способ нахождения ошибок, не требующий вставки никаких инструкций вывода промежуточных значений в текст программы. На каждом шаге мы и так можем наблюдать за значениями всех или только избранных переменных. После отладки не нужно удалять или временно отключать какие-либо инструкции, даже точки останова убирать не нужно. При обычном выполнении точки останова ни на что не влияют. Пользуйтесь отладкой при возникновении любой ошибки, и вы очень быстро ее найдете и исправите.

Примечание

Чтобы из режима **Debug** вернуться в обычный режим, в меню **Window** выбираем пункт **Perspective | Open Perspective | Java** или нажимаем соответствующую кнопку на панели инструментов.

ГЛАВА 18

Работа с файлами и каталогами

Для работы с файловой системой в языке Java предназначены классы `File` (пакет `java.io`) и `Files` (пакет `java.nio.file`). Класс `File` существует с самой первой версии языка. Он содержит базовые методы доступа к файлам и каталогам, но не содержит методов для чтения и записи файлов (для этого в пакете `java.io` применяются потоки, которые мы рассмотрим в следующих двух главах). Класс `Files` доступен, начиная с версии Java 7. Он, как и класс `File`, содержит методы доступа к файлам и каталогам, но дополнительно позволяет писать и читать файлы, работать с символическими ссылками, а также копировать и перемещать файлы. Кроме того, класс `Files` содержит методы для обхода дерева каталогов, позволяя удалять или копировать целую структуру каталогов.

Классы `File` и `Files` используют множество вспомогательных классов и интерфейсов (особенно класс `Files`). Чтобы не было проблем с импортом, добавьте в начало программы следующие инструкции импорта всех классов из пакетов (в зависимости от используемого класса):

- для класса `File`:

```
import java.io.*;
```

- для класса `Files`:

```
import java.nio.file.*;
import java.nio.file.attribute.*;
```

Классы активно используют контролируемые исключения (особенно класс `Files`), поэтому необходимо либо обработать эти исключения с помощью инструкции `try...catch`, либо добавить класс исключения после ключевого слова `throws` в заголовок метода. В целях экономии места в книге мы будем добавлять класс `Exception` после ключевого слова `throws` в заголовок метода `main()`:

```
public static void main(String[] args) throws Exception {
   // Примеры
}
```

18.1. Класс *File*

Класс `File` предназначен для работы с файлами и каталогами. Прежде чем использовать этот класс, необходимо выполнить его импорт с помощью инструкции:

```
import java.io.File;
```

Класс `File` реализует следующие интерфейсы:

```
Serializable, Comparable<File>
```

Класс `File` содержит несколько статических констант:

- `separator` — содержит символ (в виде объекта класса `String`), разделяющий имена каталогов в пути. Символ зависит от операционной системы (в Windows — \, в UNIX — /). Объявление константы:

  ```
  public static final String separator
  ```

  Пример вывода в операционной системе Windows:

  ```
  System.out.println(File.separator);          // \
  ```

- `separatorChar` — содержит символ, разделяющий имена каталогов в пути. Символ зависит от операционной системы. Объявление константы:

  ```
  public static final char separatorChar
  ```

  Пример вывода в операционной системе Windows:

  ```
  System.out.println(File.separatorChar);      // \
  ```

- `pathSeparator` — содержит символ (в виде объекта класса `String`), разделяющий несколько путей. Символ зависит от операционной системы (в Windows — ;, в UNIX — :). Объявление константы:

  ```
  public static final String pathSeparator
  ```

  Пример вывода в операционной системе Windows:

  ```
  System.out.println(File.pathSeparator);      // ;
  ```

- `pathSeparatorChar` — содержит символ, разделяющий несколько путей. Символ зависит от операционной системы. Объявление константы:

  ```
  public static final char pathSeparatorChar
  ```

  Пример вывода в операционной системе Windows:

  ```
  System.out.println(File.pathSeparatorChar); // ;
  ```

18.1.1. Создание объекта

Создать экземпляр класса `File` позволяют следующие конструкторы:

```
File(String pathname)
File(String parent, String child)
File(File parent, String child)
File(URI uri)
```

Первый конструктор принимает полный путь к файлу или каталогу в виде строки:

```
File f = new File("C:\\book\\test.txt");
System.out.println(f.getPath());     // C:\book\test.txt
```

Второй конструктор в первом параметре принимает путь к каталогу, а во втором параметре — название файла или путь, который добавится к значению первого параметра через символ-разделитель:

```
File f = new File("C:\\book" , "test.txt");
System.out.println(f.getPath());     // C:\book\test.txt
```

Третий конструктор в первом параметре принимает путь к каталогу в виде объекта класса `File`, а во втором параметре — название файла или путь, который добавится к значению первого параметра через символ-разделитель:

```
File dir = new File("C:\\book");
File f = new File(dir, "test.txt");
System.out.println(f.getPath());     // C:\book\test.txt
```

Четвертый конструктор создает объект на основе экземпляра класса `URI`:

```
// import java.net.*;
URI uri = null;
try {
   uri = new URI("file:/C:/book/test.txt");
}
catch (URISyntaxException e) {}

File f = new File(uri);
System.out.println(f.getPath());     // C:\book\test.txt
```

18.1.2. Преобразование пути к файлу или каталогу

При указании пути в Windows следует учитывать, что слэш является специальным символом. По этой причине его необходимо удваивать или использовать статическую константу `separator` из класса `File`:

```
File f = new File("C:\\book\\test.txt");
System.out.println(f.getPath());     // C:\book\test.txt
f = new File("C:" + File.separator + "book" +
    File.separator + "test.txt");
System.out.println(f.getPath());     // C:\book\test.txt
```

Если символ слэша не удвоить, то это приведет к ошибкам в дальнейшем:

```
File f = new File("C:\temp\new\file.txt"); // Путь с ошибкой!!!
System.out.println(f.getPath());
```

В этом пути присутствуют сразу три специальных символа: `\t`, `\n` и `\f`. После преобразования специальных символов путь будет выглядеть следующим образом:

```
C:<Табуляция>emp<Перевод строки>ew<Перевод формата>ile.txt
```

Так что не забывайте удваивать слэш в Windows! Вместо обратного слэша можно использовать и прямой слэш:

```
File f = new File("C:/temp/new/file.txt");
System.out.println(f.getPath());  // C:\temp\new\file.txt
```

Путь к файлу или каталогу может быть *абсолютным* или *относительным*. При указании абсолютного пути задается название диска и путь к файлу или каталогу. Вместо абсолютного пути можно указать относительный путь. В этом случае путь определяется с учетом местоположения текущего рабочего каталога. Возможны следующие варианты:

- если файл находится в текущем рабочем каталоге, то можно указать только название файла или добавить перед именем файла точку и слэш:

```
// Файл в текущем рабочем каталоге (C:\book\)
File f = new File("file.txt");
System.out.println(f.getCanonicalPath());  // C:\book\file.txt
f = new File("./file.txt");
System.out.println(f.getCanonicalPath());  // C:\book\file.txt
```

- если файл расположен во вложенной папке, то перед названием файла указываются названия вложенных папок через слэш:

```
// Файл в C:\book\folder1\
File f = new File("folder1/file.txt");
System.out.println(f.getCanonicalPath());
// C:\book\folder1\file.txt
// Файл в C:\book\folder1\folder2\
f = new File("folder1/folder2/file.txt");
System.out.println(f.getCanonicalPath());
// C:\book\folder1\folder2\file.txt
```

- если папка с файлом расположена ниже уровнем, то перед названием файла указываются две точки и слэш ("../"):

```
// Файл в C:\
File f = new File("../file.txt");
System.out.println(f.getCanonicalPath());  // C:\file.txt
```

- если в начале пути расположен слэш, то путь отсчитывается от корня текущего диска:

```
// Файл в C:\book\folder1\
File f = new File("/book/folder1/file.txt");
System.out.println(f.getCanonicalPath());
// C:\book\folder1\file.txt
// Файл в C:\book\folder1\folder2\
f = new File("/book/folder1/folder2/file.txt");
System.out.println(f.getCanonicalPath());
// C:\book\folder1\folder2\file.txt
```

При использовании относительного пути необходимо учитывать местоположение текущего рабочего каталога, т. к. рабочий каталог не всегда совпадает с каталогом, в котором находится файл с программой. Если программа запускается из командной строки, то текущим рабочим каталогом будет каталог, из которого запускается программа. Давайте рассмотрим это на примере (листинг 18.1).

Листинг 18.1. Файл C:\book\MyClass.java. Путь относительно текущего рабочего каталога

```
import java.io.*;

public class MyClass {
   public static void main(String[] args) throws IOException {
      File f = new File("file.txt");
      System.out.println(f.getCanonicalPath());
   }
}
```

Сохраняем файл в кодировке UTF-8. Запускаем командную строку и выполняем следующие команды:

```
C:\Users\Unicross>cd C:\book

C:\book>javac -encoding utf-8 MyClass.java

C:\book>java MyClass
C:\book\file.txt

C:\book>cd ..

C:\>java -cp /book MyClass
C:\file.txt
```

Обратите внимание: в первый раз мы получили путь C:\book\file.txt, а во второй раз — путь C:\file.txt. Если мы указываем относительный путь, то лучше файлы хранить как ресурсы, в этом случае файлы будут искаться относительно путей поиска классов. Если мы хотим сохранить файл в той же папке, в которой находится и класс, то можно предварительно получить путь к классу с помощью метода `getResource()` из класса `Class<T>`. Формат метода:

```
public URL getResource(String name)
```

Получить ссылку на объект класса `Class<T>` можно через название класса или через ключевое слово `this`:

```
System.out.println(MyClass.class.getResource("."));
System.out.println(this.getClass().getResource("."));
```

Давайте переделаем код из листинга 18.1 и определим путь к классу (листинг18.2).

Листинг 18.2. Файл C:\book\MyClass.java. Путь относительно класса

```
import java.io.*;

public class MyClass {
   public static void main(String[] args) throws Exception {
      File dir = new File(MyClass.class.getResource(".").toURI());
      File f = new File(dir, "file.txt");
      System.out.println(f.getCanonicalPath());
   }
}
```

Методы `getCanonicalPath()` и `toURI()` могут генерировать контролируемые исключения разных классов, поэтому в заголовке после ключевого слова `throws` мы указали базовый класс `Exception`. Сохраняем файл, запускаем командную строку и выполняем следующие команды:

```
C:\Users\Unicross>cd C:\book

C:\book>javac -encoding utf-8 MyClass.java

C:\book>java MyClass
C:\book\file.txt

C:\book>cd ..

C:\>java -cp /book MyClass
C:\book\file.txt
```

Теперь наш относительный путь не зависит от текущего рабочего каталога.

Для работы с путями к файлу или каталогу предназначены следующие методы из класса `File`:

- `toString()` — возвращает текстовое представление объекта. Формат метода:

  ```
  public String toString()
  ```

  Пример:

  ```
  File f = new File("C:\\book\\file.txt");
  System.out.println(f.toString());  // C:\book\file.txt
  f = new File("../file.txt");
  System.out.println(f.toString());  // ..\file.txt
  ```

- `getAbsolutePath()` — возвращает абсолютный путь (точки в относительном пути не преобразуются). Формат метода:

  ```
  public String getAbsolutePath()
  ```

  Пример:

  ```
  File f = new File("C:\\book\\file.txt");
  System.out.println(f.getAbsolutePath());  // C:\book\file.txt
  ```

```
f = new File("../file.txt");
System.out.println(f.getAbsolutePath());  // C:\book\..\file.txt
```

- `getAbsoluteFile()` — **метод аналогичен методу** `getAbsolutePath()`, **но возвращает объект класса** `File`. **Формат метода:**

```
public File getAbsoluteFile()
```

- `getCanonicalPath()` — **возвращает абсолютный путь с учетом всех преобразований. Формат метода:**

```
public String getCanonicalPath() throws IOException
```

Метод может генерировать контролируемое исключение, поэтому необходимо либо использовать инструкцию `try...catch`, **либо добавить исключение в заголовок метода после ключевого слова** `throws`:

```
File f = new File("C:\\book\\file.txt");
System.out.println(f.getCanonicalPath());  // C:\book\file.txt
f = new File("../file.txt");
System.out.println(f.getCanonicalPath());  // C:\file.txt
```

- `getCanonicalFile()` — **метод аналогичен методу** `getCanonicalPath()`, **но возвращает объект класса** `File`. **Формат метода:**

```
public File getCanonicalFile() throws IOException
```

- `getPath()` — **возвращает текстовое представление объекта. Формат метода:**

```
public String getPath()
```

Пример:

```
File f = new File("C:\\book\\file.txt");
System.out.println(f.getPath());  // C:\book\file.txt
f = new File("../file.txt");
System.out.println(f.getPath());  // ..\file.txt
```

- `getName()` — **возвращает имя файла или каталога. Формат метода:**

```
public String getName()
```

Пример:

```
File f = new File("C:\\book\\file.txt");
System.out.println(f.getName());  // file.txt
f = new File("C:\\book\\");
System.out.println(f.getName());  // book
```

- `getParent()` — **возвращает путь к родительскому каталогу или значение** `null`. **Формат метода:**

```
public String getParent()
```

Пример:

```
File f = new File("C:\\book\\file.txt");
System.out.println(f.getParent());  // C:\book
f = new File("C:\\book\\");
```

```
System.out.println(f.getParent());  // C:\
f = new File("file.txt");
System.out.println(f.getParent());  // null
```

- `getParentFile()` — метод аналогичен методу `getParent()`, но возвращает объект класса `File` или значение `null`. Формат метода:

```
public File getParentFile()
```

- `toURI()` — возвращает объект класса `URI`. Формат метода:

```
public URI toURI()
```

Пример:

```
File f = new File("C:\\book\\file.txt");
System.out.println(f.toURI());  // file:/C:/book/file.txt
f = new File("../file.txt");
System.out.println(f.toURI());  // file:/C:/book/../file.txt
```

- `toPath()` — возвращает объект, реализующий интерфейс `Path`. Формат метода:

```
public Path toPath()
```

Пример:

```
File f = new File("C:\\book\\file.txt");
System.out.println(f.toPath());  // C:\book\file.txt
f = new File("../file.txt");
System.out.println(f.toPath());  // ..\file.txt
```

- `isAbsolute()` — возвращает значение `true`, если путь абсолютный, и `false` — если относительный. Формат метода:

```
public boolean isAbsolute()
```

Пример:

```
File f = new File("C:\\book\\file.txt");
System.out.println(f.isAbsolute());  // true
f = new File("../file.txt");
System.out.println(f.isAbsolute());  // false
```

18.1.3. Работа с дисками

Для работы с дисками предназначены следующие методы из класса `File`:

- `listRoots()` — возвращает массив с названиями всех дисков. Метод является статическим. Формат метода:

```
public static File[] listRoots()
```

Пример:

```
File[] list = File.listRoots();
for (File f: list) {
   System.out.print(f.toString() + " ");
} // C:\ D:\ E:\
```

- `getTotalSpace()` — возвращает полный размер диска. Формат метода:

```
public long getTotalSpace()
```

- `getFreeSpace()` — возвращает количество свободного места на диске. Формат метода:

```
public long getFreeSpace()
```

- `getUsableSpace()` — возвращает количество свободного места на диске, доступного виртуальной машине. Формат метода:

```
public long getUsableSpace()
```

Пример:

```
File f = new File("D:\\");
System.out.printf("%,d\n", f.getTotalSpace());
// 427 424 739 328
System.out.printf("%,d\n", f.getFreeSpace());
// 182 450 044 928
System.out.printf("%,d\n", f.getUsableSpace());
// 182 450 044 928
```

18.1.4. Работа с каталогами

Для работы с каталогами предназначены следующие методы из класса `File`:

- `isDirectory()` — возвращает значение `true`, если объект содержит путь к каталогу, и `false` — в противном случае. Формат метода:

```
public boolean isDirectory()
```

Пример:

```
File f = new File("C:\\book\\");
System.out.println(f.isDirectory()); // true
f = new File("C:\\book\\file.txt");
System.out.println(f.isDirectory()); // false
```

- `mkdir()` — создает каталог. Возвращает значение `true`, если каталог успешно создан, и `false` — в противном случае. Формат метода:

```
public boolean mkdir()
```

Пример:

```
File f = new File("C:\\book\\folder1\\");
System.out.println(f.mkdir()); // true
```

- `mkdirs()` — создает все каталоги в пути. Возвращает значение `true`, если каталоги успешно созданы, и `false` — в противном случае. Формат метода:

```
public boolean mkdirs()
```

Пример:

```
File f = new File("C:\\book\\folder2\\folder3\\");
System.out.println(f.mkdirs()); // true
```

- `renameTo()` — переименовывает каталог. Возвращает значение `true`, если каталог успешно переименован, и `false` — в противном случае. Формат метода:

```
public boolean renameTo(File dest)
```

Пример:

```
File f = new File("C:\\book\\folder1\\");
System.out.println(
   f.renameTo(new File("C:\\book\\folder0\\"))); // true
```

- `list()` — возвращает массив с названиями файлов и каталогов внутри указанного каталога. Форматы метода:

```
public String[] list()
public String[] list(FilenameFilter filter)
```

Пример:

```
File f = new File("C:\\book\\");
String[] list = f.list();
for (String s: list) {
   System.out.println(s);
   File file = new File(f, s);
   System.out.println("--- Каталог? " + file.isDirectory());
   System.out.println("--- Файл? " + file.isFile());
}
```

Во втором формате в качестве параметра можно указать объект, реализующий интерфейс `FilenameFilter`. Интерфейс содержит объявление одного метода:

```
boolean accept(File dir, String name)
```

Если внутри метода вернуть значение `true`, то элемент попадет в массив с результатами. Интерфейс является функциональным, поэтому в качестве параметра в Java 8 мы можем указать лямбда-выражение. Выведем только файлы, имеющие расширение java:

```
File f = new File("C:\\book\\");
String[] list = f.list(
      (dir, name) -> name.toLowerCase().endsWith(".java"));
for (String s: list) {
   System.out.println(s);
}
```

- `listFiles()` — возвращает массив с файлами и каталогами внутри указанного каталога. Форматы метода:

```
public File[] listFiles()
public File[] listFiles(FilenameFilter filter)
public File[] listFiles(FileFilter filter)
```

Пример:

```
File f = new File("C:\\book\\");
File[] list = f.listFiles();
for (File obj: list) {
   System.out.println(obj.toString());
   System.out.println("--- Каталог? " + obj.isDirectory());
   System.out.println("--- Файл? " + obj.isFile());
}
```

Второй формат аналогичен второму формату метода `list()`. В третьем формате в качестве параметра можно указать объект, реализующий интерфейс `FileFilter`. Интерфейс содержит объявление одного метода:

```
boolean accept(File pathname)
```

Если внутри метода вернуть значение `true`, то элемент попадет в массив с результатами. Интерфейс является функциональным, поэтому в качестве параметра в Java 8 мы можем указать лямбда-выражение. Выведем только файлы, имеющие расширение java:

```
File f = new File("C:\\book\\");
File[] list = f.listFiles(
   (file) -> file.getName().toLowerCase().endsWith(".java"));
for (File obj: list) {
   System.out.println(obj.toString());
}
```

- `delete()` — удаляет каталог. Обратите внимание: каталог должен быть пустой. Возвращает значение `true`, если каталог удален, и `false` — в противном случае. Формат метода:

```
public boolean delete()
```

Пример:

```
File f = new File("C:\\book\\folder0\\");
System.out.println(f.delete());      // true
```

- `exists()` — возвращает значение `true`, если каталог существует, и `false` — в противном случае. Формат метода:

```
public boolean exists()
```

Пример:

```
File f = new File("C:\\book\\folder0\\");
System.out.println(f.exists());      // false
f = new File("C:\\book\\folder2\\");
System.out.println(f.exists());      // true
```

18.1.5. Работа с файлами

Для работы с файлами предназначены следующие методы из класса `File`:

- `isFile()` — возвращает значение `true`, если объект содержит путь к файлу, и `false` — в противном случае. Формат метода:

```
public boolean isFile()
```

Пример:

```
File f = new File("C:\\book\\");
System.out.println(f.isFile()); // false
f = new File("C:\\book\\file.txt");
System.out.println(f.isFile()); // true
```

- `exists()` — возвращает значение `true`, если файл существует, и `false` — в противном случае. Формат метода:

```
public boolean exists()
```

Пример:

```
File f = new File("C:\\book\\file.txt");
System.out.println(f.exists());          // true
f = new File("C:\\book\\file2.txt");
System.out.println(f.exists());          // false
```

- `createNewFile()` — создает новый файл, если файл не существует. Возвращает значение `true`, если файл успешно создан, и `false` — в противном случае. Формат метода:

```
public boolean createNewFile() throws IOException
```

Пример:

```
File f = new File("C:\\book\\file.txt");
System.out.println(f.createNewFile());   // false
f = new File("C:\\book\\file2.txt");
System.out.println(f.createNewFile());   // true
```

- `createTempFile()` — создает временный файл. Метод является статическим. Форматы метода:

```
public static File createTempFile(String prefix,String suffix)
                            throws IOException
public static File createTempFile(String prefix, String suffix,
                    File directory) throws IOException
```

В параметре `prefix` необходимо указать не менее трех символов. К этим символам будет добавлено случайно сгенерированное число. В параметре `suffix` можно задать расширение файла. Если этот параметр имеет значение `null`, то расширением будет tmp. В параметре `directory` можно задать пользовательский каталог. Если этот параметр имеет значение `null`, то файл создается в системной папке для временных файлов:

```
File dir = new File("C:\\book\\");
File f = File.createTempFile("mytmp", null, dir);
System.out.println(f); // C:\book\mytmp17615399026568862378.tmp
File f2 = File.createTempFile("tmp", null, null);
System.out.println(f2);
// C:\Users\Unicross\AppData\Local\Temp\
//                                tmp15356944168258988064.tmp
```

- `renameTo()` — **переименовывает файл. Возвращает значение** `true`, **если файл успешно переименован, и** `false` **— в противном случае. Формат метода:**

```
public boolean renameTo(File dest)
```

Пример:

```
File f = new File("C:\\book\\file2.txt");
System.out.println(
   f.renameTo(new File("C:\\book\\file3.txt"))); // true
```

- `delete()` — **удаляет файл. Возвращает значение** `true`, **если файл удален, и** `false` **— в противном случае. Формат метода:**

```
public boolean delete()
```

Пример:

```
File f = new File("C:\\book\\file3.txt");
System.out.println(f.delete()); // true
```

- `deleteOnExit()` — **удаляет файл или каталог после завершения работы программы. Формат метода:**

```
public void deleteOnExit()
```

Пример (понаблюдайте за содержимым каталога C:\book в течение 5 секунд):

```
File dir = new File("C:\\book\\");
File f = File.createTempFile("del", null, dir);
System.out.println(f); // C:\book\del13157661722678848959.tmp
f.deleteOnExit();
Thread.sleep(5000);    // Засыпаем на 5 секунд
System.out.println("exit");
```

- `length()` — **возвращает размер файла. Если файл не найден, то возвращается значение** `0`. **Формат метода:**

```
public long length()
```

Пример:

```
File f = new File("C:\\book\\MyClass.java");
System.out.println(f.length()); // 295
```

- `isHidden()` — **возвращает значение** `true`, **если файл скрытый, и** `false` **— в противном случае. Формат метода:**

```
public boolean isHidden()
```

- `lastModified()` — возвращает дату (количество миллисекунд с 1 января 1970 года) последнего изменения файла. Формат метода:

```
public long lastModified()
```

- `setLastModified()` — устанавливает дату (количество миллисекунд с 1 января 1970 года) последнего изменения файла. Формат метода:

```
public boolean setLastModified(long time)
```

Пример:

```
File f = new File("C:\\book\\file.txt");
Date d = new Date(f.lastModified());
System.out.println(d);
// Wed Mar 21 12:42:49 MSK 2018
System.out.println(
   f.setLastModified(
      (new Date(d.getTime() - 24*60*60*1000)).getTime()));
System.out.println(new Date(f.lastModified()));
// Tue Mar 20 12:42:49 MSK 2018
```

18.1.6. Права доступа к файлам и каталогам

В операционной системе семейства UNIX каждому объекту (файлу или каталогу) назначаются права доступа для каждой разновидности пользователей: владельца, группы и прочих. Могут быть назначены следующие права доступа:

- чтение;
- запись;
- выполнение.

Права доступа обозначаются буквами:

- `r` — файл можно читать, а содержимое каталога можно просматривать;
- `w` — файл можно модифицировать, удалять и переименовывать, а в каталоге можно создавать или удалять файлы. Каталог можно переименовать или удалить;
- `x` — файл можно выполнять, а в каталоге можно выполнять операции над файлами, в том числе производить поиск файлов в нем.

Права доступа к файлу определяются записью типа:

```
-rw-r--r--
```

Первый символ (`-`) означает, что это файл, и не задает никаких прав доступа. Далее три символа (`rw-`) задают права доступа для владельца (чтение и запись). Символ (`-`) здесь означает, что права доступа на выполнение нет. Следующие три символа (`r--`) задают права доступа для группы (только чтение). Ну, и последние три символа (`r--`) задают права для всех остальных пользователей (только чтение).

Права доступа к каталогу определяются такой строкой:

```
drwxr-xr-x
```

Первый символ (d) означает, что это каталог. Владелец может выполнять в каталоге любые действия (rwx), а группа и все остальные пользователи — только читать и выполнять поиск (r-x). Для того чтобы каталог можно было просматривать, должны быть установлены права на выполнение (x).

Права доступа также могут быть обозначены и числом. Такие числа называются *масками прав доступа*. Число состоит из трех цифр: от 0 до 7. Первая цифра задает права для владельца, вторая — для группы, а третья — для всех остальных пользователей. Например, права доступа -rw-r--r-- соответствуют числу 644.

Для проверки или изменения прав доступа из программы используются следующие методы из класса File:

- canWrite() — возвращает значение true, если в файл или каталог можно записать, и false — в противном случае. Формат метода:

  ```
  public boolean canWrite()
  ```

- canRead() — возвращает значение true, если файл или каталог доступен для чтения, и false — в противном случае. Формат метода:

  ```
  public boolean canRead()
  ```

  Пример:

  ```
  File f = new File("C:\\book\\file.txt");
  System.out.println(f.canWrite()); // true
  System.out.println(f.canRead());  // true
  ```

- canExecute() — возвращает значение true, если файл или каталог выполняемый, и false — в противном случае. Формат метода:

  ```
  public boolean canExecute()
  ```

- setExecutable() — устанавливает права на выполнение. Возвращает значение true, если права изменены, и false — в противном случае. Форматы метода:

  ```
  public boolean setExecutable(boolean executable)
  public boolean setExecutable(boolean executable,
                               boolean ownerOnly)
  ```

- setReadable() — устанавливает права на чтение. Возвращает значение true, если права изменены, и false — в противном случае. Форматы метода:

  ```
  public boolean setReadable(boolean readable)
  public boolean setReadable(boolean readable,
                             boolean ownerOnly)
  ```

- setWritable() — устанавливает права на запись. Возвращает значение true, если права изменены, и false — в противном случае. Форматы метода:

```
public boolean setWritable(boolean writable)
public boolean setWritable(boolean writable,
                           boolean ownerOnly)
```

Сделаем файл доступным только для чтения, а потом отменим эту установку:

```
File f = new File("C:\\book\\file.txt");
System.out.println(f.canWrite());          // true
System.out.println(f.setWritable(false));  // true
System.out.println(f.canWrite());          // false
System.out.println(f.setWritable(true));   // true
System.out.println(f.canWrite());          // true
```

- `setReadOnly()` — делает файл доступным только для чтения. Возвращает значение `true`, если права изменены, и `false` — в противном случае. Формат метода:

```
public boolean setReadOnly()
```

Пример:

```
File f = new File("C:\\book\\file.txt");
System.out.println(f.canWrite());          // true
System.out.println(f.setReadOnly());       // true
System.out.println(f.canWrite());          // false
```

18.2. Класс *Files*

Начиная с версии 7, в языке Java доступен класс `Files`, который содержит множество статических методов, предназначенных для работы с файлами, каталогами и символическими ссылками. Прежде чем использовать этот класс, необходимо выполнить его импорт с помощью инструкции:

```
import java.nio.file.Files;
```

18.2.1. Класс *Paths* и интерфейс *Path*

Путь к файлам и каталогам описывается с помощью интерфейса `Path`. Создать объект, реализующий этот интерфейс, позволяет статический метод `get()` из класса `Paths`. Прежде чем использовать класс `Paths` и интерфейс `Path`, необходимо выполнить их импорт с помощью инструкций:

```
import java.nio.file.Path;
import java.nio.file.Paths;
```

Форматы метода `get()`:

```
public static Path get(String first, String... more)
public static Path get(URI uri)
```

Первый формат позволяет указать отдельные компоненты пути в виде строк. Строки будут объединены через символ, используемый для разделения каталогов в операционной системе:

```
Path p = Paths.get("C:\\book\\file.txt");
System.out.println(p.toString()); // C:\book\file.txt
p = Paths.get("C:", "book", "file.txt");
System.out.println(p.toString()); // C:\book\file.txt
```

Второй формат создает путь на основе объекта класса `URI`:

```
// import java.net.*;
URI uri = null;
try {
   uri = new URI("file:/C:/book/file.txt");
}
catch (URISyntaxException e) {}

Path p = Paths.get(uri);
System.out.println(p.toString()); // C:\book\file.txt
```

Путь можно создать на основе объекта класса `File`, вызвав метод `toPath()`. Формат метода:

```
public Path toPath()
```

Пример:

```
File f = new File("../file.txt");
Path p = f.toPath();
System.out.println(p.toString());  // ..\file.txt
```

Выполнить обратную операцию позволяет метод `toFile()` из интерфейса `Path`. Формат метода:

```
public File toFile()
```

Пример:

```
Path p = Paths.get("C:\\book\\file.txt");
File f = p.toFile();
System.out.println(f.toString());  // C:\book\file.txt
```

Интерфейс `Path` содержит следующие основные методы:

- `toString()` — возвращает текстовое представление объекта. Формат метода:

  ```
  public String toString()
  ```

  Пример:

  ```
  Path p = Paths.get("C:\\book\\file.txt");
  System.out.println(p.toString());  // C:\book\file.txt
  p = Paths.get("../file.txt");
  System.out.println(p.toString());  // ..\file.txt
  ```

- `isAbsolute()` — возвращает значение `true`, если путь абсолютный, и `false` — если относительный. Формат метода:

  ```
  public boolean isAbsolute()
  ```

Пример:

```
Path p = Paths.get("C:\\book\\file.txt");
System.out.println(p.isAbsolute());  // true
p = Paths.get("../file.txt");
System.out.println(p.isAbsolute());  // false
```

- `toAbsolutePath()` — возвращает абсолютный путь (точки в относительном пути не преобразуются). Формат метода:

```
public Path toAbsolutePath()
```

Пример:

```
Path p = Paths.get("C:\\book\\file.txt");
System.out.println(p.toAbsolutePath()); // C:\book\file.txt
p = Paths.get("../file.txt");
System.out.println(p.toAbsolutePath()); // C:\book\..\file.txt
```

- `normalize()` — преобразует все специальные символы в абсолютном пути, производя нормализацию пути. Формат метода:

```
public Path normalize()
```

Пример:

```
Path p = Paths.get("C:\\book\\..\\file.txt");
System.out.println(p.normalize());  // C:\file.txt
p = Paths.get("C:/book\\../file.txt");
System.out.println(p.normalize());  // C:\file.txt
```

- `toRealPath()` — возвращает нормализованный путь, проверяя при этом существование файла. Если файл не существует, то генерируется исключение `java.nio.file.NoSuchFileException`. Формат метода:

```
public Path toRealPath(LinkOption... options)
                       throws IOException
```

В параметре `options` можно указать значение `NOFOLLOW_LINKS` из перечисления `java.nio.file.LinkOption`, которое запрещает преобразование символических ссылок:

```
Path p = Paths.get("C:\\book\\folder1\\..\\file.txt");
System.out.println(p.toRealPath()); // C:\book\file.txt
```

- `getRoot()` — возвращает название диска или значение `null`, если путь не содержит название диска. Формат метода:

```
public Path getRoot()
```

- `getParent()` — возвращает путь к родительскому каталогу или значение `null`. Формат метода:

```
public Path getParent()
```

- `getFileName()` — возвращает имя файла или каталога. Формат метода:

```
public Path getFileName()
```

Пример:

```
Path p = Paths.get("C:\\book\\folder1\\file.txt");
System.out.println(p.getRoot());      // C:\
System.out.println(p.getParent());    // C:\book\folder1
System.out.println(p.getFileName()); // file.txt
```

- `getNameCount()` — возвращает количество элементов в пути (название диска не учитывается). Формат метода:

```
public int getNameCount()
```

- `getName()` — возвращает элемент пути по индексу. Формат метода:

```
public Path getName(int index)
```

Пример перебора всех элементов пути:

```
Path p = Paths.get("C:\\book\\folder1\\file.txt");
System.out.println(p.getNameCount());      // 3
for (int i = 0, j = p.getNameCount(); i < j; i++) {
   System.out.print(p.getName(i) + " ");
} // book folder1 file.txt
```

- `iterator()` — возвращает итератор, с помощью которого можно перебрать элементы в пути (кроме названия диска). Формат метода:

```
import java.util.Iterator;
public Iterator<Path> iterator()
```

Пример:

```
Path p = Paths.get("C:\\book\\folder1\\file.txt");
Iterator<Path> it = p.iterator();
while (it.hasNext()) {
   System.out.print(it.next() + " ");
} // book folder1 file.txt
for (Path elem: p) {
   System.out.print(elem + " ");
} // book folder1 file.txt
```

- `forEach()` — позволяет перебрать элементы в пути (кроме названия диска). Метод доступен, начиная с Java 8. Формат метода:

```
public void forEach(Consumer<? super T> action)
```

Пример:

```
Path p = Paths.get("C:\\book\\folder1\\file.txt");
p.forEach( (elem) -> System.out.print(elem + " ") );
// book folder1 file.txt
```

- `subpath()` — возвращает фрагмент пути от элемента с индексом `beginIndex` до элемента с индексом `endIndex` (не включая элемент с этим индексом). Формат метода:

```
public Path subpath(int beginIndex, int endIndex)
```

Пример:

```
Path p = Paths.get("C:\\book\\folder1\\file.txt");
System.out.println(p.subpath(1, 3)); // folder1\file.txt
```

- `resolve()` — добавляет относительный путь к текущему пути. Если в параметре указан абсолютный путь, то возвращается именно он. Форматы метода:

```
public Path resolve(Path other)
public Path resolve(String other)
```

Пример:

```
Path p = Paths.get("C:\\book\\folder1");
Path p2 = Paths.get("file.txt");
System.out.println(p.resolve(p2));
// C:\book\folder1\file.txt
System.out.println(p.resolve("file2.txt"));
// C:\book\folder1\file2.txt
```

- `resolveSibling()` — заменяет последний элемент значением из параметра. Если в параметре указан абсолютный путь, то возвращается именно он. Форматы метода:

```
public Path resolveSibling(Path other)
public Path resolveSibling(String other)
```

Пример:

```
Path p = Paths.get("C:\\book\\folder1\\file.txt");
Path p2 = Paths.get("file1.txt");
System.out.println(p.resolveSibling(p2));
// C:\book\folder1\file1.txt
System.out.println(p.resolveSibling("file2.txt"));
// C:\book\folder1\file2.txt
```

- `startsWith()` — возвращает значение `true`, если текущий путь начинается с указанного в параметре пути, и `false` — в противном случае. Форматы метода:

```
public boolean startsWith(Path other)
public boolean startsWith(String other)
```

Пример:

```
Path p = Paths.get("C:\\book\\folder1\\file.txt");
Path p2 = Paths.get("C:\\book");
System.out.println(p.startsWith(p2));          // true
System.out.println(p.startsWith("D:\\book"));  // false
```

- `endsWith()` — возвращает значение `true`, если текущий путь заканчивается указанным в параметре путем, и `false` — в противном случае. Форматы метода:

```
public boolean endsWith(Path other)
public boolean endsWith(String other)
```

Пример:

```
Path p = Paths.get("C:\\book\\folder1\\file.txt");
Path p2 = Paths.get("folder1\\file.txt");
System.out.println(p.endsWith(p2));          // true
System.out.println(p.endsWith("file2.txt")); // false
```

- `toUri()` — **возвращает объект класса** `URI` **с текущим путем. Формат метода:**

```
import java.net.URI;
public URI toUri()
```

Пример:

```
Path p = Paths.get("C:\\book\\file.txt");
URI uri = p.toUri();
System.out.println(uri); // file:///C:/book/file.txt
```

- `getFileSystem()` — **возвращает объект класса** `FileSystem`**. Формат метода:**

```
import java.nio.file.FileSystem;
public FileSystem getFileSystem()
```

Пример получения символа-разделителя в пути (метод `getSeparator()`**) и имен всех дисков с помощью итератора:**

```
Path p = Paths.get("C:\\book\\file.txt");
FileSystem fs = p.getFileSystem();
System.out.println(fs.getSeparator()); // \
Iterator<Path> it = fs.getRootDirectories().iterator();
while (it.hasNext()) {
   System.out.print(it.next() + " ");
} // C:\ D:\ E:\
```

- `equals()` — **возвращает значение** `true`**, если два объекта пути являются одинаковыми, и** `false` **— в противном случае. Формат метода:**

```
public boolean equals(Object other)
```

Пример:

```
Path p = Paths.get("C:\\book\\utf16.txt");
Path p2 = Paths.get("C:\\BOOK\\UTF16.txt");
System.out.println(p.equals(p2));            // true
```

- `compareTo()` — **сравнивает два объекта пути. Формат метода:**

```
public int compareTo(Path other)
```

Пример:

```
Path p = Paths.get("C:\\book\\utf16.txt");
Path p2 = Paths.get("C:\\BOOK\\UTF16.txt");
System.out.println(p.compareTo(p2));           // 0
p2 = Paths.get("C:\\book\\utf17.txt");
System.out.println(p.compareTo(p2));           // -1
```

```
p2 = Paths.get("C:\\book\\utf15.txt");
System.out.println(p.compareTo(p2));            // 1
```

18.2.2. Работа с дисками

Для работы с дисками предназначены следующие методы:

- `getRootDirectories()` — возвращает итератор, с помощью которого можно получить названия всех дисков в системе. Формат метода:

```
import java.nio.file.FileSystem;
public Iterable<Path> getRootDirectories()
```

Пример:

```
Path p = Paths.get("C:\\book\\file.txt");
FileSystem fs = p.getFileSystem();
for (Path obj: fs.getRootDirectories()) {
   System.out.print(obj.toString() + " ");
} // C:\ D:\ E:\
```

- `getFileStores()` — возвращает итератор, с помощью которого можно получить информацию о доступных дисках. На каждой итерации доступен объект класса `FileStore`. Формат метода:

```
import java.nio.file.FileSystems;
import java.nio.file.FileStore;
public Iterable<FileStore> getFileStores()
```

Пример:

```
FileSystem fs = FileSystems.getDefault();
for (FileStore obj: fs.getFileStores()) {
   System.out.print(obj.toString() + " ");
} // OS (C:) DATA (D:)
```

- `getFileStore()` — возвращает объект класса `FileStore`, с помощью которого можно получить дополнительную информацию о доступном диске. Если диск не существует или не готов, или объект, на который ссылается путь, не существует на диске, то генерируется исключение. Формат метода:

```
import java.nio.file.Files;
import java.nio.file.FileStore;
public static FileStore getFileStore(Path path)
                                throws IOException
```

Пример:

```
Path p = Paths.get("D:");
FileStore fs = Files.getFileStore(p);
System.out.println(fs.toString()); // DATA (D:)
System.out.printf("%,d\n", fs.getTotalSpace());
// 427 424 739 328
```

Класс `FileStore` содержит следующие методы:

- `toString()` — возвращает текстовое представление о типе и названии диска. Формат метода:

```
public String toString()
```

Пример:

```
FileStore fs = Files.getFileStore(Paths.get("D:"));
System.out.println(fs.toString()); // DATA (D:)
```

- `name()` — возвращает тип диска. Формат метода:

```
public String name()
```

Пример:

```
FileStore fs = Files.getFileStore(Paths.get("C:"));
System.out.println(fs.name()); // OS
fs = Files.getFileStore(Paths.get("D:"));
System.out.println(fs.name()); // DATA
```

- `type()` — возвращает тип файловой системы. Формат метода:

```
public String type()
```

Пример:

```
FileStore fs = Files.getFileStore(Paths.get("D:"));
System.out.println(fs.type()); // NTFS
```

- `getTotalSpace()` — возвращает полный размер диска. Формат метода:

```
public long getTotalSpace() throws IOException
```

- `getUnallocatedSpace()` — возвращает количество свободного места на диске. Формат метода:

```
public long getUnallocatedSpace() throws IOException
```

- `getUsableSpace()` — возвращает количество свободного места на диске, доступного виртуальной машине. Формат метода:

```
public long getUsableSpace() throws IOException
```

Пример:

```
FileStore fs = Files.getFileStore(Paths.get("D:"));
System.out.printf("%,d\n", fs.getTotalSpace());
// 427 424 739 328
System.out.printf("%,d\n", fs.getUnallocatedSpace());
// 182 450 044 928
System.out.printf("%,d\n", fs.getUsableSpace());
// 182 450 044 928
```

- `isReadOnly()` — возвращает значение `true`, если диск доступен только для чтения, и `false` — в противном случае. Формат метода:

```
public boolean isReadOnly()
```

Пример:

```
FileStore fs = Files.getFileStore(Paths.get("D:"));
System.out.println(fs.isReadOnly()); // false
```

18.2.3. Работа с каталогами

Для работы с каталогами предназначены следующие методы из класса `Files`:

- `isDirectory()` — возвращает значение `true`, если объект содержит путь к каталогу, и `false` — в противном случае. Формат метода:

```
public static boolean isDirectory(Path path,
                                  LinkOption... options)
```

Пример:

```
Path p = Paths.get("C:\\book\\");
System.out.println(Files.isDirectory(p)); // true
p = Paths.get("C:\\book\\file.txt");
System.out.println(Files.isDirectory(p)); // false
```

- `createDirectory()` — создает каталог. Возвращает путь, если каталог успешно создан, в противном случае генерируется исключение. Формат метода:

```
public static Path createDirectory(Path dir,
                  FileAttribute<?>... attrs)
                        throws IOException
```

Параметр `attrs` задает значения атрибутов каталога — например, права доступа в UNIX (описание этого параметра приведено в *разд. 18.2.7*):

```
Path p = Paths.get("C:\\book\\folder1\\");
System.out.println(Files.createDirectory(p));
// C:\book\folder1
```

- `createDirectories()` — создает все каталоги в пути. Возвращает путь, если каталоги успешно созданы. Формат метода:

```
public static Path createDirectories(Path dir,
                    FileAttribute<?>... attrs)
                          throws IOException
```

Пример:

```
Path p = Paths.get("C:\\book\\folder3\\folder4\\");
System.out.println(Files.createDirectories(p));
// C:\book\folder3\folder4
```

- `createTempDirectory()` — создает временный каталог. К строке `prefix` будет добавлено случайное число. Метод возвращает путь к созданному каталогу. Форматы метода:

```
public static Path createTempDirectory(Path dir,
        String prefix, FileAttribute<?>... attrs)
                              throws IOException
```

```
public static Path createTempDirectory(String prefix,
                          FileAttribute<?>... attrs)
                                throws IOException
```

Параметр `attrs` задает значения атрибутов каталога — например, права доступа в UNIX (описание этого параметра приведено в *разд. 18.2.7*):

```
Path p = Paths.get("C:\\book\\");
System.out.println(Files.createTempDirectory(p, "tmp"));
// C:\book\tmp5876097479015506533
System.out.println(Files.createTempDirectory("tmp"));
// C:\Users\Unicross\AppData\Local\Temp\tmp4765578470243500002
```

- `newDirectoryStream()` — возвращает поток, с помощью которого можно прочитать содержимое каталога. Форматы метода:

```
import java.nio.file.DirectoryStream;
public static DirectoryStream<Path>
             newDirectoryStream(Path dir) throws IOException
public static DirectoryStream<Path> newDirectoryStream(Path dir,
             DirectoryStream.Filter<? super Path> filter)
                                    throws IOException
public static DirectoryStream<Path> newDirectoryStream(Path dir,
             String glob)           throws IOException
```

Пример чтения всего содержимого каталога:

```
Path p = Paths.get("C:\\book\\");
try (DirectoryStream<Path> stream =
                          Files.newDirectoryStream(p)) {
   for (Path obj: stream) {
      System.out.println(obj.toString());
      System.out.println("--- Каталог? " +
                         Files.isDirectory(obj));
      System.out.println("--- Файл? " +
                         Files.isRegularFile(obj));
   }
}
```

В этом примере мы использовали новую для нас форму инструкции `try`. После ключевого слова `try` внутри круглых скобок создается объект потока. Если внутри фигурных скобок возникнет исключение, то объект потока будет автоматически закрыт. Это достигается за счет реализации интерфейса `AutoCloseable`.

Во втором формате в качестве второго параметра можно указать объект, реализующий интерфейс `DirectoryStream.Filter<? super Path>`. Интерфейс содержит объявление одного метода:

```
boolean accept(T entry)
```

Если внутри метода вернуть значение `true`, то элемент попадет в поток с результатами. Интерфейс является функциональным, поэтому в качестве параметра

в Java 8 мы можем указать лямбда-выражение. Выведем только файлы, имеющие расширение java:

```
Path p = Paths.get("C:\\book\\");
try (DirectoryStream<Path> stream =
     Files.newDirectoryStream(p, (path) ->
        path.toString().toLowerCase().endsWith(".java"))) {
   for (Path obj: stream) {
      System.out.println(obj.toString());
   }
}
```

В третьем формате во втором параметре указывается строка специального формата, внутри которой можно использовать следующие символы:

- `*` — любое количество любых символов;
- `?` — любой одиночный символ;
- `[<Символы>]` — позволяет указать символы, которые должны быть на этом месте в пути. Можно задать символы или указать диапазон через тире;
- `{<Список через запятую>}` — задает список с возможными значениями (после запятой не должно быть пробела).

Пример получения всех файлов с расширениями java и class:

```
Path p = Paths.get("C:\\book\\");
try (DirectoryStream<Path> stream =
     Files.newDirectoryStream(p, "*.{java,class}")) {
   for (Path obj: stream) {
      System.out.println(obj.toString());
   }
}
```

- `list()` — возвращает объект, реализующий интерфейс `Stream<T>`, с помощью которого можно перебрать все элементы каталога. Формат метода:

```
import java.util.stream.Stream;
public static Stream<Path> list(Path dir)
                     throws IOException
```

Пример вывода всех элементов каталога и только файлов с расширением java:

```
Path p = Paths.get("C:\\book\\");
try (Stream<Path> stream = Files.list(p)) {
   stream.forEachOrdered(
      (obj) -> System.out.println(obj.toString())
   );
}
try (Stream<Path> stream = Files.list(p)) {
   stream.filter(
      (path) -> path.toString().endsWith(".java")
```

```
    ).forEachOrdered(
       (obj) -> System.out.println(obj.toString())
    );
}
```

- `delete()` — удаляет каталог. Обратите внимание: каталог должен быть пустой. Если не удалось удалить каталог, то генерируется исключение. Формат метода:

```
public static void delete(Path path) throws IOException
```

Пример:

```
Path p = Paths.get("C:\\book\\folder1\\");
Files.delete(p);
```

- `deleteIfExists()` — удаляет каталог. Обратите внимание: каталог должен быть пустой, а если это не так, то генерируется исключение. Возвращает значение `true`, если каталог удален, и `false` — в противном случае. Формат метода:

```
public static boolean deleteIfExists(Path path)
                            throws IOException
```

Пример:

```
Path p = Paths.get("C:\\book\\folder2\\folder3\\");
System.out.println(Files.deleteIfExists(p)); // true
```

- `exists()` — возвращает значение `true`, если каталог существует, и `false` — в противном случае. Формат метода:

```
public static boolean exists(Path path,
                             LinkOption... options)
```

Пример:

```
Path p = Paths.get("C:\\book\\folder2\\folder3\\");
System.out.println(Files.exists(p)); // false
p = Paths.get("C:\\book\\folder2\\");
System.out.println(Files.exists(p)); // true
```

- `notExists()` — возвращает значение `true`, если каталог не существует, и `false` — в противном случае. Формат метода:

```
public static boolean notExists(Path path,
                                LinkOption... options)
```

Пример:

```
Path p = Paths.get("C:\\book\\");
System.out.println(Files.notExists(p)); // false
p = Paths.get("C:\\book2\\");
System.out.println(Files.notExists(p)); // true
```

18.2.4. Обход дерева каталогов

Для обхода дерева каталогов предназначены методы `walkFileTree()` и `walk()` из класса `Files`. Форматы метода `walkFileTree()`:

```
public static Path walkFileTree(Path start,
              FileVisitor<? super Path> visitor) throws IOException
public static Path walkFileTree(Path start,
              Set<FileVisitOption> options, int maxDepth,
              FileVisitor<? super Path> visitor) throws IOException
```

В первом формате в первом параметре указывается начальный каталог, а во втором параметре — объект, реализующий интерфейс `FileVisitor<T>`. Интерфейс содержит следующие методы:

- `visitFile()` — вызывается для каждого файла в каталоге. Если используется второй формат метода `walkFileTree()` и указано ограничение `maxDepth`, то параметр может быть и каталогом. Формат метода:

```
public FileVisitResult visitFile(T file,
      BasicFileAttributes attrs) throws IOException
```

- `preVisitDirectory()` — вызывается для каждого каталога перед чтением его содержимого. То есть после вызова этого метода будет вызван метод `visitFile()`. Формат метода:

```
public FileVisitResult preVisitDirectory(T dir,
      BasicFileAttributes attrs) throws IOException
```

- `postVisitDirectory()` — метод вызывается для каждого каталога после чтения всех файлов и каталогов внутри него. Внутри этого метода можно удалить каталог, предварительно удалив все файлы внутри метода `visitFile()`. Формат метода:

```
public FileVisitResult postVisitDirectory(T dir,
                  IOException e) throws IOException
```

- `visitFileFailed()` — метод вызывается, когда не получается открыть каталог или прочитать атрибуты файла. Формат метода:

```
public FileVisitResult visitFileFailed(Path file,
               IOException e) throws IOException
```

Все эти методы реализованы в классе `SimpleFileVisitor<T>`, который мы можем наследовать или использовать при создании анонимного вложенного класса. Достаточно переопределить и реализовать нужные нам методы. Прежде чем использовать этот класс, необходимо выполнить его импорт с помощью инструкции:

```
import java.nio.file.SimpleFileVisitor;
```

Внутри методов необходимо вернуть значение из перечисления `FileVisitResult`:

```
import java.nio.file.FileVisitResult;
```

- CONTINUE — продолжить обход дерева;
- TERMINATE — прекратить обход дерева;
- SKIP_SUBTREE — продолжить обход без просмотра содержимого каталога. Значение можно вернуть только внутри метода preVisitDirectory();
- SKIP_SIBLINGS — если значение возвращается из метода preVisitDirectory(), то продолжить обход без просмотра содержимого текущего каталога (метод postVisitDirectory() при этом не вызывается). Если значение возвращается из метода visitFile(), то продолжить обход без просмотра содержимого текущего каталога и всех вложенных каталогов.

Второй формат метода walkFileTree() удобно использовать при создании копии дерева каталогов. В первом параметре указывается начальный каталог, во втором — множество со значениями из перечисления FileVisitOption (содержит значение FOLLOW_LINKS), в третьем — глубина вложенности (нумерация с 0), а в четвертом — объект, реализующий интерфейс FileVisitor<T>.

Рассмотрим последовательность обхода дерева каталогов на примере. Вначале реализуем методы preVisitDirectory() и visitFile() (листинг 18.3).

Листинг 18.3. Обход дерева каталогов в прямом порядке

```
package com.example.app;

import java.nio.file.*;
import java.nio.file.attribute.*;
import java.io.IOException;

public class MyClass {
   public static void main(String[] args) throws Exception {
      Path p = Paths.get("C:\\book\\folder0\\");
      Files.walkFileTree(p, new SimpleFileVisitor<Path>() {
         @Override
         public FileVisitResult preVisitDirectory(Path dir,
            BasicFileAttributes attrs) throws IOException {
            System.out.println("dir: " + dir.toString());
            return FileVisitResult.CONTINUE;
         }
         @Override
         public FileVisitResult visitFile(Path file,
            BasicFileAttributes attrs) throws IOException {
            System.out.println("file: " + file.toString());
            return FileVisitResult.CONTINUE;
         }
      });
   }
}
```

Результат обхода:

```
dir: C:\book\folder0
file: C:\book\folder0\file.txt
dir: C:\book\folder0\folder1
file: C:\book\folder0\folder1\file.txt
dir: C:\book\folder0\folder1\folder2
file: C:\book\folder0\folder1\folder2\file.txt
```

Обход дерева каталогов в этом порядке удобно использовать для поиска элементов и их копирования. Теперь реализуем методы `visitFile()` и `postVisitDirectory()` (листинг 18.4).

Листинг 18.4. Обход дерева каталогов в обратном порядке

```
package com.example.app;

import java.nio.file.*;
import java.nio.file.attribute.*;
import java.io.IOException;

public class MyClass {
   public static void main(String[] args) throws Exception {
      Path p = Paths.get("C:\\book\\folder0\\");
      Files.walkFileTree(p, new SimpleFileVisitor<Path>() {
         @Override
         public FileVisitResult visitFile(Path file,
            BasicFileAttributes attrs) throws IOException {
            System.out.println("file: " + file.toString());
            return FileVisitResult.CONTINUE;
         }
         @Override
         public FileVisitResult postVisitDirectory(Path dir,
                     IOException e) throws IOException {
            System.out.println("dir: " + dir.toString());
            return FileVisitResult.CONTINUE;
         }
      });
   }
}
```

Результат обхода:

```
file: C:\book\folder0\file.txt
file: C:\book\folder0\folder1\file.txt
file: C:\book\folder0\folder1\folder2\file.txt
dir: C:\book\folder0\folder1\folder2
dir: C:\book\folder0\folder1
dir: C:\book\folder0
```

Такой порядок обхода удобно использовать для удаления элементов. Вы ведь помните, что удалить можно только пустой каталог. Вот мы и можем вначале удалить файлы (внутри метода `visitFile()`), а затем останутся пустые подкаталоги, которые можно удалить внутри метода `postVisitDirectory()`.

Для обхода дерева каталогов можно также воспользоваться методом `walk()` из класса `Files`. Метод доступен, начиная с Java 8. Форматы метода:

```
public static Stream<Path> walk(Path start,
             FileVisitOption... options) throws IOException
public static Stream<Path> walk(Path start, int maxDepth,
             FileVisitOption... options) throws IOException
```

В первом параметре указывается начальный каталог. Параметр `maxDepth` задает глубину вложенности (нумерация с `0`), а параметр `options` — значения из перечисления `FileVisitOption` (содержит значение `FOLLOW_LINKS`). Пример использования метода `walk()` показан в листинге 18.5.

Листинг 18.5. Обход дерева каталогов с помощью метода `walk()`

```
package com.example.app;

import java.nio.file.*;
import java.util.Collections;
import java.util.stream.Stream;

public class MyClass {
   public static void main(String[] args) throws Exception {
      Path p = Paths.get("C:\\book\\folder0\\");
      try (Stream<Path> stream = Files.walk(p, 3)) {
         stream.forEachOrdered(
            (path) -> System.out.println(path.toString())
         );
      }
      System.out.println();
      try (Stream<Path> stream2 = Files.walk(p)) {
         stream2.sorted(Collections.reverseOrder()).forEachOrdered(
            (path) -> System.out.println(path.toString())
         );
      }
   }
}
```

Результат обхода:

```
C:\book\folder0
C:\book\folder0\file.txt
C:\book\folder0\folder1
C:\book\folder0\folder1\file.txt
```

```
C:\book\folder0\folder1\folder2
C:\book\folder0\folder1\folder2\file.txt

C:\book\folder0\folder1\folder2\file.txt
C:\book\folder0\folder1\folder2
C:\book\folder0\folder1\file.txt
C:\book\folder0\folder1
C:\book\folder0\file.txt
C:\book\folder0
```

Для поиска элемента внутри дерева каталогов можно воспользоваться методом `find()`. Метод доступен, начиная с Java 8. Формат метода:

```
import java.util.stream.Stream;
public static Stream<Path> find(Path start, int maxDepth,
              BiPredicate<Path, BasicFileAttributes> matcher,
              FileVisitOption... options) throws IOException
```

В первом параметре указывается начальный каталог, а во втором — максимальный уровень вложенности. Третий параметр позволяет указать ссылку на метод или лямбда-выражение, реализующие функциональный интерфейс `BiPredicate<Path, BasicFileAttributes>`. Внутри метода необходимо вернуть значение `true`, если элемент соответствует критериям поиска, и `false` — в противном случае. Параметр `options` задает значения из перечисления `FileVisitOption` (содержит значение `FOLLOW_LINKS`). Метод возвращает объект `Stream<Path>`. Пример поиска всех текстовых файлов:

```
Path p = Paths.get("C:\\book\\folder0\\");
try (Stream<Path> stream = Files.find(p, 4, (f, attrs) -> {
         return f.toString().endsWith(".txt");
      }))
{
   stream.forEachOrdered(
      (path) -> System.out.println(path.toString())
   );
}
```

18.2.5. Работа с файлами

Для работы с файлами предназначены следующие методы из класса `Files`:

- `isRegularFile()` — возвращает значение `true`, если объект содержит путь к файлу, и `false` — в противном случае. Формат метода:

  ```
  public static boolean isRegularFile(Path path,
                                      LinkOption... options)
  ```

  По умолчанию метод обрабатывает символические ссылки. Если в параметре `options` указать значение `NOFOLLOW_LINKS` из перечисления `LinkOption`, то символические ссылки обрабатываться не будут:

```
Path p = Paths.get("C:\\book\\");
System.out.println(Files.isRegularFile(p)); // false
p = Paths.get("C:\\book\\file.txt");
System.out.println(Files.isRegularFile(p)); // true
```

- `exists()` — возвращает значение `true`, если файл существует, и `false` — в противном случае. Формат метода:

```
public static boolean exists(Path path,
                             LinkOption... options)
```

Пример:

```
Path p = Paths.get("C:\\book\\file.txt");
System.out.println(Files.exists(p)); // true
p = Paths.get("C:\\book\\file2.txt");
System.out.println(Files.exists(p)); // false
```

- `notExists()` — возвращает значение `true`, если файл не существует, и `false` — в противном случае. Формат метода:

```
public static boolean notExists(Path path,
                                LinkOption... options)
```

Пример:

```
Path p = Paths.get("C:\\book\\file.txt");
System.out.println(Files.notExists(p)); // false
p = Paths.get("C:\\book\\file2.txt");
System.out.println(Files.notExists(p)); // true
```

- `createFile()` — создает новый файл, если файл не существует. Если файл существует, то генерируется исключение. Возвращает путь к созданному файлу. Формат метода:

```
public static Path createFile(Path path,
             FileAttribute<?>... attrs) throws IOException
```

Параметр `attrs` задает значения атрибутов файла — например, права доступа в UNIX (описание этого параметра приведено в *разд. 18.2.7*):

```
Path p = Paths.get("C:\\book\\file2.txt");
System.out.println(Files.createFile(p)); // C:\book\file2.txt
```

- `createTempFile()` — создает временный файл. Форматы метода:

```
public static Path createTempFile(String prefix,
             String suffix, FileAttribute<?>... attrs)
                                throws IOException
public static Path createTempFile(Path dir, String prefix,
             String suffix, FileAttribute<?>... attrs)
                                throws IOException
```

Параметр `prefix` задает начальные символы названия файла. К этим символам будет добавлено случайно сгенерированное число. В параметре `suffix` можно

задать расширение файла. Если этот параметр имеет значение `null`, то расширением будет tmp. В параметре `dir` можно задать пользовательский каталог. Если этот параметр не указан, то файл создается в системной папке для временных файлов. Параметр `attrs` задает значения атрибутов файла — например, права доступа в UNIX (описание этого параметра приведено в *разд. 18.2.7*):

```
Path dir = Paths.get("C:\\book\\");
Path p = Files.createTempFile(dir, "mytmp", null);
System.out.println(p);
// C:\book\mytmp7779328372162434148.tmp
p = Files.createTempFile("tmp", ".tmp");
System.out.println(p);
// C:\Users\Unicross\AppData\Local\Temp\
//                                  tmp6598314850043915181.tmp
```

- `delete()` — удаляет файл. При ошибке генерируется исключение. Формат метода:

  ```
  public static void delete(Path path) throws IOException
  ```

  Пример:

  ```
  Path p = Paths.get("C:\\book\\file2.txt");
  Files.delete(p);
  ```

- `deleteIfExists()` — удаляет файл. Возвращает значение `true`, если файл удален, и `false` — в противном случае. Формат метода:

  ```
  public static boolean deleteIfExists(Path path)
                              throws IOException
  ```

  Пример:

  ```
  Path p = Paths.get("C:\\book\\file2.txt");
  System.out.println(Files.deleteIfExists(p)); // false
  ```

- `size()` — возвращает размер файла в байтах. Если файл не найден, то генерируется исключение. Формат метода:

  ```
  public static long size(Path path) throws IOException
  ```

  Пример:

  ```
  Path p = Paths.get("C:\\book\\file.txt");
  System.out.println(Files.size(p)); // 18
  ```

- `isHidden()` — возвращает значение `true`, если файл скрытый, и `false` — в противном случае. Если файл не найден, то генерируется исключение. Формат метода:

  ```
  public static boolean isHidden(Path path)
                       throws IOException
  ```

  Пример:

  ```
  Path p = Paths.get("C:\\book\\file.txt");
  System.out.println(Files.isHidden(p)); // false
  ```

- `getLastModifiedTime()` — возвращает дату (объект класса `FileTime`) последнего изменения файла. Формат метода:

```
import java.nio.file.attribute.FileTime;
public static FileTime getLastModifiedTime(Path path,
              LinkOption... options) throws IOException
```

- `setLastModifiedTime()` — устанавливает дату последнего изменения файла. Формат метода:

```
import java.nio.file.attribute.FileTime;
public static Path setLastModifiedTime(Path path,
              FileTime time) throws IOException
```

Пример:

```
Path p = Paths.get("C:\\book\\file.txt");
Date d = new Date(Files.getLastModifiedTime(p).toMillis());
System.out.println(d); // Mon Mar 19 03:42:49 MSK 2018
FileTime time = FileTime.fromMillis(d.getTime() - 24*60*60*1000);
System.out.println(Files.setLastModifiedTime(p, time));
// C:\book\file.txt
```

Для получения количества миллисекунд используется метод `toMillis()` из класса `FileTime`. Преобразовать количество миллисекунд в объект класса `FileTime` позволяет статический метод `fromMillis()`;

- `probeContentType()` — возвращает MIME-тип файла по расширению или значение `null`. Наличие файла по пути не проверяется. Формат метода:

```
public static String probeContentType(Path path)
                              throws IOException
```

Пример:

```
Path p = Paths.get("C:\\book\\test.txt");
System.out.println(Files.probeContentType(p)); // text/plain
p = Paths.get("C:\\book\\test.html");
System.out.println(Files.probeContentType(p)); // text/html
p = Paths.get("C:\\book\\test.gif");
System.out.println(Files.probeContentType(p)); // image/gif
```

- `isSameFile()` — возвращает значение `true`, если два объекта пути ссылаются на один и тот же файл, и `false` — в противном случае. Если файлы не найдены, то генерируется исключение. Если пути эквивалентны или отличие только в названии диска, то проверка наличия файла не производится. Формат метода:

```
public static boolean isSameFile(Path path, Path path2)
                                    throws IOException
```

Пример:

```
Path p = Paths.get("C:\\book\\file.txt");
Path p2 = Paths.get("C:\\BOOK\\FILE.txt");
System.out.println(Files.isSameFile(p, p2)); // true
```

18.2.6. Права доступа к файлам и каталогам

Для проверки или изменения прав доступа из программы используются следующие методы из класса `Files`:

- `isWritable()` — возвращает значение `true`, если в файл или каталог можно записать, и `false` — в противном случае. Формат метода:

```
public static boolean isWritable(Path path)
```

- `isReadable()` — возвращает значение `true`, если файл или каталог доступен для чтения, и `false` — в противном случае. Формат метода:

```
public static boolean isReadable(Path path)
```

Пример:

```
Path p = Paths.get("C:\\book\\file.txt");
System.out.println(Files.isWritable(p)); // true
System.out.println(Files.isReadable(p)); // true
```

- `isExecutable()` — возвращает значение `true`, если файл или каталог выполняемый, и `false` — в противном случае. Формат метода:

```
public static boolean isExecutable(Path path)
```

Установка прав доступа к файлам и каталогам осуществляется с помощью атрибутов. Атрибуты можно указать при создании файла или каталога с помощью параметра `attrs` или после создания с помощью метода `getFileAttributeView()` (см. *разд. 18.2.7*).

18.2.7. Атрибуты файлов и каталогов

Получить или изменить значения различных параметров файлов или каталогов можно с помощью атрибутов. Все классы для работы с атрибутами расположены в пакете `java.nio.file.attribute`. Классов очень много, поэтому проще импортировать сразу все классы с помощью инструкции:

```
import java.nio.file.attribute.*;
```

Для работы с атрибутами предназначены следующие методы из класса `Files`:

- `getAttribute()` — возвращает значение атрибута по его имени. Формат метода:

```
public static Object getAttribute(Path path,
              String attribute, LinkOption... options)
                                  throws IOException
```

Пример получения даты создания, модификации и последнего доступа к файлу, а также размера файла (получить список названий атрибутов позволяет метод `readAttributes()`):

```
Path p = Paths.get("C:\\book\\file.txt");
FileTime ft = (FileTime) Files.getAttribute(p,
                          "lastModifiedTime");
```

```
System.out.println(ft.toString());
// 2018-03-18T00:42:49.666Z
ft = (FileTime) Files.getAttribute(p, "lastAccessTime");
System.out.println(ft.toString());
// 2018-03-21T00:42:49.666889Z
ft = (FileTime) Files.getAttribute(p, "creationTime");
System.out.println(ft.toString());
// 2018-03-21T00:42:49.666889Z
long size = (long) Files.getAttribute(p, "size");
System.out.println(size); // 18
```

- `readAttributes()` — позволяет получить значения различных атрибутов. Форматы метода:

```
public static Map<String, Object>
              readAttributes(Path path, String attributes,
                 LinkOption... options) throws IOException
public static <A extends BasicFileAttributes> A
              readAttributes(Path path, Class<A> type,
              LinkOption... options)  throws IOException
```

В первом формате в параметре `attributes` можно указать значения: "`*`" (все атрибуты), "`size,lastModifiedTime`" (указанные атрибуты), "`posix:*`" (все атрибуты `posix`) и "`posix:permissions`" (указанные атрибуты из `posix`):

```
Path p = Paths.get("C:\\book\\file.txt");
Map<String, Object> attrs = Files.readAttributes(p, "*");
System.out.println(attrs.keySet());
// [lastAccessTime, lastModifiedTime, size, creationTime,
// isSymbolicLink, isRegularFile, fileKey, isOther, isDirectory]
```

Теперь рассмотрим пример использования второго формата. Получим дату создания, модификации и последнего доступа к файлу, а также размер файла (полный список методов интерфейса `BasicFileAttributes` смотрите в документации):

```
Path p = Paths.get("C:\\book\\file.txt");
BasicFileAttributes attr =
    Files.readAttributes(p, BasicFileAttributes.class);
System.out.println(attr.lastModifiedTime());
// 2018-03-18T00:42:49.666Z
System.out.println(attr.lastAccessTime());
// 2018-03-21T00:42:49.666889Z
System.out.println(attr.creationTime());
// 2018-03-21T00:42:49.666889Z
System.out.println(attr.size()); // 18
```

Вместо интерфейса `BasicFileAttributes` можно указать интерфейсы `DosFileAttributes` (в Windows) или `PosixFileAttributes` (в UNIX). Реализация интерфейсов зависит от используемой операционной системы. Если интерфейс

не подходит для текущей операционной системы, то генерируется исключение. Пример использования интерфейса `DosFileAttributes`:

```
Path p = Paths.get("C:\\book\\file.txt");
DosFileAttributes attr =
    Files.readAttributes(p, DosFileAttributes.class);
System.out.println(attr.isHidden());  // false
System.out.println(attr.isArchive()); // true
System.out.println(attr.isSystem());  // false
```

- `getFileAttributeView()` — позволяет изменить значения атрибутов. Формат метода:

```
public static <V extends FileAttributeView> V
       getFileAttributeView(Path path, Class<V> type,
                            LinkOption... options)
```

Пример изменения даты модификации и последнего доступа к файлу (полный список методов интерфейса `BasicFileAttributeView` смотрите в документации):

```
Path p = Paths.get("C:\\book\\file.txt");
BasicFileAttributeView attr =
   Files.getFileAttributeView(p, BasicFileAttributeView.class);
FileTime ft = FileTime.fromMillis((new Date()).getTime());
attr.setTimes(ft, ft, null);
```

Формат метода `setTimes()`:

```
public void setTimes(FileTime lastModifiedTime,
            FileTime lastAccessTime, FileTime createTime)
                                     throws IOException
```

Вместо интерфейса `BasicFileAttributeView` можно указать интерфейсы `DosFileAttributeView` (в Windows) или `PosixFileAttributeView` (в UNIX). Реализация интерфейсов зависит от используемой операционной системы. Пример использования интерфейса `DosFileAttributeView` (делаем файл скрытым):

```
Path p = Paths.get("C:\\book\\file.txt");
DosFileAttributeView attr =
   Files.getFileAttributeView(p, DosFileAttributeView.class);
attr.setHidden(true);
```

С помощью метода `setPermissions()` из интерфейса `PosixFileAttributeView` можно установить права доступа для файлов и каталогов в операционной системе UNIX. Формат метода:

```
public void setPermissions(Set<PosixFilePermission> perms)
                                     throws IOException
```

Создать множество `Set<PosixFilePermission>` из строкового представления прав доступа позволяет статический метод `fromString()` из класса `PosixFilePermissions`. Формат метода:

```
import java.nio.file.attribute.*;
public static Set<PosixFilePermission> fromString(String perms)
```

Пример:

```
Set<PosixFilePermission> s =
                          PosixFilePermissions.fromString("rw-r--r--");
System.out.println(s);
// [OWNER_READ, OWNER_WRITE, GROUP_READ, OTHERS_READ]
```

Выполнить обратное преобразование позволяет статический метод `toString()` из класса `PosixFilePermissions`. Формат метода:

```
import java.nio.file.attribute.*;
public static String toString(Set<PosixFilePermission> perms)
```

Пример:

```
System.out.println(PosixFilePermissions.toString(s));
// rw-r--r--
```

Получить и задать права доступа в операционной системе UNIX позволяют также методы `getPosixFilePermissions()` и `setPosixFilePermissions()` из класса `Files`. Форматы методов:

```
public static Set<PosixFilePermission>
       getPosixFilePermissions(Path path, LinkOption... options)
                                          throws IOException
public static Path setPosixFilePermissions(Path path,
        Set<PosixFilePermission> perms)      throws IOException
```

Задать права доступа можно при создании файла или каталога с помощью параметра `attrs` в методах `createFile()` и `createDirectory()` класса `Files`. Форматы методов:

```
public static Path createFile(Path path,
              FileAttribute<?>... attrs) throws IOException
public static Path createDirectory(Path dir,
              FileAttribute<?>... attrs) throws IOException
```

Создать объект, реализующий интерфейс `FileAttribute<T>`, позволяет статический метод `asFileAttribute()` из класса `PosixFilePermissions`. Формат метода:

```
import java.nio.file.attribute.*;
public static FileAttribute< Set<PosixFilePermission> >
       asFileAttribute(Set<PosixFilePermission> perms)
```

Пример:

```
Set<PosixFilePermission> s =
                   PosixFilePermissions.fromString("rw-r--r--");
FileAttribute< Set<PosixFilePermission> > obj =
                   PosixFilePermissions.asFileAttribute(s);
System.out.println(obj.value());
// [OWNER_WRITE, OTHERS_READ, OWNER_READ, GROUP_READ]
```

18.2.8. Копирование и перемещение файлов и каталогов

Для копирования и перемещения файлов и каталогов предназначены следующие методы из класса `Files`:

- `copy()` — создает копию файла или каталога. Формат метода:

```
public static Path copy(Path source, Path target,
       CopyOption... options) throws IOException
```

Если файл уже существует, и не указана опция `REPLACE_EXISTING`, то генерируется исключение. Если копируется каталог, то будет скопировано только его имя без содержимого. Чтобы скопировать все содержимое каталога, необходимо дополнительно воспользоваться методом `walkFileTree()` (см. *разд. 18.2.4*). В параметре `options` могут быть указаны следующие значения из перечисления `StandardCopyOption`:

- `REPLACE_EXISTING` — если файл существует, то он будет заменен. По умолчанию генерируется исключение;
- `COPY_ATTRIBUTES` — копирует атрибуты файла.

Кроме того, можно указать значение `NOFOLLOW_LINKS` из перечисления `LinkOption`.

Пример копирования файла с заменой существующего файла:

```
Path p = Paths.get("C:\\book\\file.txt");
Path p2 = Paths.get("C:\\book\\file2.txt");
System.out.println(Files.copy(p, p2,
     StandardCopyOption.REPLACE_EXISTING));
// C:\book\file2.txt
```

- `move()` — перемещает файл или каталог. Формат метода:

```
public static Path move(Path source, Path target,
       CopyOption... options) throws IOException
```

Если файл уже существует, и не указана опция `REPLACE_EXISTING`, то генерируется исключение. Если каталог перемещается в пределах одного диска, то будет перемещено все его содержимое. Если перемещение осуществляется на другой диск, и каталог не пустой, то генерируется исключение. В параметре `options` могут быть указаны следующие значения из перечисления `StandardCopyOption`:

- `REPLACE_EXISTING` — если файл существует, то он будет заменен. По умолчанию генерируется исключение;
- `ATOMIC_MOVE` — перемещение выполняется как единая операция. Если это невозможно, то генерируется исключение.

Пример перемещения файла:

```
Path p = Paths.get("C:\\book\\file2.txt");
Path p2 = Paths.get("C:\\book\\folder2\\file2.txt");
```

```
System.out.println(Files.move(p, p2));
// C:\book\folder2\file2.txt
```

18.2.9. Чтение и запись файлов

Для записи файлов предназначены следующие методы из класса `Files`:

- `write()` — записывает данные в файл. Форматы метода:

```
public static Path write(Path path, byte[] bytes,
       OpenOption... options)  throws IOException
import java.nio.charset.Charset;
public static Path write(Path path,
       Iterable<? extends CharSequence> lines,
       Charset cs, OpenOption... options)
                              throws IOException
public static Path write(Path path,
        Iterable<? extends CharSequence> lines,
        OpenOption... options) throws IOException
```

По умолчанию создается новый файл (опция `CREATE`), и поток открывается на запись (опция `WRITE`). Если файл существует, то он будет перезаписан (опция `TRUNCATE_EXISTING`). Пример записи в файл в кодировке по умолчанию:

```
Path p = Paths.get("C:\\book\\data.txt");
String s = "строка1\nстрока2\nстрока3\n";
System.out.println(Files.write(p, s.getBytes()));
// C:\book\data.txt
```

Пример записи в файл в кодировке windows-1251:

```
Path p = Paths.get("C:\\book\\cp1251.txt");
String s = "строка1\nстрока2\nстрока3\n";
byte[] bytes = s.getBytes("cp1251");
System.out.println(Files.write(p, bytes));
// C:\book\cp1251.txt
```

Если необходимо добавить записи в уже существующий файл, то следует явным образом указать опцию `APPEND` (если файл не существует, то генерируется исключение):

```
Path p = Paths.get("C:\\book\\cp1251.txt");
String s = "строка4\nстрока5\n";
byte[] bytes = s.getBytes("cp1251");
System.out.println(Files.write(p, bytes,
                              StandardOpenOption.APPEND));
// C:\book\cp1251.txt
```

Второй и третий форматы позволяют записать в файл данные из объектов, реализующих интерфейс `Iterable<T>`. Параметр `cs` задает кодировку, а если кодировка не задана, то используется кодировка UTF-8 (кодировку, начиная с Java 8,

можно не указывать). Данные добавляются в файл построчно через разделитель, используемый в системе по умолчанию (в Windows — \r\n). Пример записи в файл строк из списка в кодировке windows-866:

```
Path p = Paths.get("C:\\book\\cp866.txt");
ArrayList<String> arr = new ArrayList<String>();
arr.add("строка1");
arr.add("строка2");
arr.add("строка3");
System.out.println(Files.write(p, arr,
                                Charset.forName("cp866")));
// C:\book\cp866.txt
```

- newOutputStream() — открывает файл на запись и возвращает поток. Формат метода:

```
import java.io.OutputStream;
public static OutputStream newOutputStream(Path path,
              OpenOption... options) throws IOException
```

По умолчанию создается новый файл (опция CREATE), и поток открывается на запись (опция WRITE). Если файл существует, то он будет перезаписан (опция TRUNCATE_EXISTING). Пример записи в файл в кодировке UTF-8:

```
Path p = Paths.get("C:\\book\\utf8.txt");
OutputStream out = null;
try {
   out = Files.newOutputStream(p);
   String s = "строка1\nстрока2\nстрока3\n";
   byte[] bytes = s.getBytes("utf-8");
   out.write(bytes);                    // Записываем данные
}
finally {
   if (out != null) out.close();        // Закрываем поток
}
```

- copy() — записывает все данные из потока в файл. Формат метода:

```
import java.io.InputStream;
public static long copy(InputStream in, Path target,
       CopyOption... options) throws IOException
```

Пример:

```
// import java.io.ByteArrayInputStream;
Path p = Paths.get("C:\\book\\utf16.txt");
String s = "строка1\nстрока2\nстрока3\n";
byte[] bytes = s.getBytes("utf-16");
InputStream in = null;
try {
   in = new ByteArrayInputStream(bytes);
```

```
    System.out.println(Files.copy(in, p));
}
finally {
    if (in != null) in.close();       // Закрываем поток
}
```

Для чтения файлов предназначены следующие методы из класса `Files`:

- `readAllBytes()` — читает все байты из файла в массив. Формат метода:

```
public static byte[] readAllBytes(Path path)
                        throws IOException
```

Если файл не существует, то генерируется исключение. После чтения файл автоматически закрывается. Если файл большой, то этот метод лучше не использовать. Пример чтения всего файла:

```
Path p = Paths.get("C:\\book\\utf8.txt");
byte[] bytes = Files.readAllBytes(p);
String s = new String(bytes, "utf-8");
System.out.println(s);
System.out.println(Arrays.toString(bytes));
```

- `readAllLines()` — читает весь файл построчно в список строк. Форматы метода:

```
public static List<String> readAllLines(Path path, Charset cs)
                                          throws IOException
public static List<String> readAllLines(Path path)
                                          throws IOException
```

Если файл не существует, то генерируется исключение. После чтения файл автоматически закрывается. Если файл большой, то этот метод лучше не использовать. В параметре `cs` можно указать кодировку данных, а если кодировка не указана, то используется кодировка UTF-8 (кодировку, начиная с Java 8, можно не указывать). Пример чтения всего файла в кодировке windows-1251:

```
Path p = Paths.get("C:\\book\\cp1251.txt");
List<String> arr = Files.readAllLines(p,
                        Charset.forName("cp1251"));
for (String item: arr) {
    System.out.println(item);
}
```

- `newInputStream()` — открывает файл на чтение и возвращает поток. Формат метода:

```
import java.io.InputStream;
public static InputStream newInputStream(Path path,
             OpenOption... options) throws IOException
```

Пример чтения всего файла в кодировке windows-1251:

```
Path p = Paths.get("C:\\book\\cp1251.txt");
InputStream in = null;
```

```
try {
   in = Files.newInputStream(p);
   byte[] bytes = new byte[in.available()];
   in.read(bytes);                    // Читаем все данные
   String s = new String(bytes, "cp1251");
   System.out.println(s);
}
finally {
   if (in != null) in.close();        // Закрываем поток
}
```

- `lines()` — **читает файл построчно в поток. Метод доступен, начиная с Java 8. Форматы метода:**

```
import java.util.stream.Stream;
public static Stream<String> lines(Path path)
                          throws IOException
public static Stream<String> lines(Path path,
             Charset cs) throws IOException
```

В параметре `cs` **можно указать кодировку данных, а если кодировка не указана, то используется кодировка UTF-8. Пример чтения всего файла в кодировке windows-1251:**

```
Path p = Paths.get("C:\\book\\cp1251.txt");
try (Stream<String> stream =
        Files.lines(p, Charset.forName("cp1251")))
{
   stream.forEachOrdered( (s) -> System.out.println(s) );
}
```

- `copy()` — **копирует все байты из файла в поток. Формат метода:**

```
import java.io.OutputStream;
public static long copy(Path source, OutputStream out)
                                  throws IOException
```

Пример:

```
// import java.io.ByteArrayOutputStream;
Path p = Paths.get("C:\\book\\utf16.txt");
ByteArrayOutputStream out = new ByteArrayOutputStream();
try {
   Files.copy(p, out);
   String s = new String(out.toByteArray(), "utf-16");
   System.out.println(s);
}
finally {
   if (out != null) out.close();      // Закрываем поток
}
```

В параметре `OpenOption... options` могут быть указаны следующие основные значения (полный список смотрите в документации) из перечисления `StandardOpenOption`:

```
import java.nio.file.StandardOpenOption;
```

- `CREATE` — создать новый файл, если он не существует;
- `CREATE_NEW` — создать новый файл. Если файл уже существует, генерируется исключение;
- `READ` — открыть файл для чтения;
- `WRITE` — открыть файл для записи;
- `TRUNCATE_EXISTING` — если используется значение `WRITE`, то данные внутри файла будут удалены;
- `APPEND` — добавить данные. Если используется значение `WRITE`, то данные будут добавлены в конец файла.

18.3. Получение сведений об операционной системе

При работе с файловой системой может потребоваться получить сведения об используемой операционной системе, т. к. некоторые классы зависят от операционной системы. Получить эти данные позволяет статический метод `getProperty()` из класса `System`. Формат метода:

```
public static String getProperty(String key)
```

В параметре можно указать следующие названия свойств:

- `java.version` — версия используемой виртуальной машины Java:

```
System.out.println(System.getProperty("java.version"));
// 10
```

- `java.home` — путь к JRE:

```
System.out.println(System.getProperty("java.home"));
// C:\Program Files\Java\jre-10
```

- `java.class.path` — пути поиска классов:

```
System.out.println(System.getProperty("java.class.path"));
// .;/book;/book/folder1
```

- `java.io.tmpdir` — путь к папке с временными файлами:

```
System.out.println(System.getProperty("java.io.tmpdir"));
// C:\Users\Unicross\AppData\Local\Temp\
```

- `os.name` — название операционной системы:

```
System.out.println(System.getProperty("os.name"));
// Windows 8
```

- `file.separator` — **символ-разделитель каталогов в пути:**

```
System.out.println(System.getProperty("file.separator"));
// \
```

- `path.separator` — **символ-разделитель путей:**

```
System.out.println(System.getProperty("path.separator"));
// ;
```

- `line.separator` — **разделитель строк (в Windows — `\r\n`, в UNIX — `\n`):**

```
byte[] bytes = System.getProperty("line.separator").getBytes();
System.out.println(Arrays.toString(bytes));
// [13, 10]
```

- `user.name` — **имя пользователя:**

```
System.out.println(System.getProperty("user.name"));
// Unicross
```

- `user.home` — **домашний каталог пользователя:**

```
System.out.println(System.getProperty("user.home"));
// C:\Users\Unicross
```

- `user.dir` — **текущий рабочий каталог (каталог, из которого запущена программа, это не обязательно местонахождение запущенного файла):**

```
System.out.println(System.getProperty("user.dir"));
// C:\book
```

Чтобы получить путь к файлу с классом, можно воспользоваться следующим кодом:

```
File dir = new File(MyClass.class.getResource(".").toURI());
System.out.println(dir.getCanonicalPath());
```

Пути после запуска из командной строки (первая строка — результат `user.dir`, вторая — `getResource(".")`):

```
C:\book>javac -encoding utf-8 MyClass.java

C:\book>java -cp /book MyClass
C:\book
C:\book

C:\book>cd ..

C:\>java -cp /book MyClass
C:\
C:\book
```

- `user.country` — **аббревиатура страны пользователя:**

```
System.out.println(System.getProperty("user.country"));
// RU
```

- `user.language` — **аббревиатура языка пользователя:**

```
System.out.println(System.getProperty("user.language"));
// ru
```

Полный список всех свойств позволяет получить метод `getProperties()` **из класса** `System`. **Формат метода:**

```
import java.util.Properties;
public static Properties getProperties()
```

Выведем все свойства:

```
Properties map = System.getProperties();
for (String obj: map.stringPropertyNames()) {
   System.out.println(obj);
}
```

ГЛАВА 19

Байтовые потоки ввода/вывода

В предыдущей главе мы изучили работу с файловой системой, научились создавать и удалять файлы и каталоги, перемещать и копировать файлы, читать содержимое каталога и целого дерева каталогов. Однако почти не рассматривали работу с самими файлами, т. к. класс `File` вообще не содержит методов для чтения и записи файлов, а класс `Files` — во-первых, доступен только с седьмой версии языка, а во-вторых, хотя и содержит методы для чтения и записи файлов, но позволяет читать файлы только целиком. Если файл имеет большой размер, то вполне возможна ситуация, когда оперативной памяти просто не хватит. Гораздо эффективнее не читать весь файл целиком, а обрабатывать его по частям.

Для чтения и записи файлов в языке Java используются *потоки*. Объект, из которого можно считывать данные, называется *потоком ввода*, а объект, в который можно записывать данные, называется *потоком вывода*. В качестве источника данных для объекта может быть не только файл, но и другие источники, — например: массивы, сокеты, окно консоли. Потоки ввода/вывода делятся на два типа: *байтовые* и *символьные*. Байтовые потоки оперируют отдельными байтами и используются для чтения и записи двоичных данных. Символьные потоки оперируют символами в различных кодировках и используются для чтения и записи текста. В этой главе мы рассмотрим использование байтовых потоков применительно к файлам, а символьные потоки будем изучать в следующей главе.

Классов потоков очень много, и чтобы не было проблем с импортом, добавьте в начало программы инструкцию:

```
import java.io.*;
```

19.1. Базовые классы байтовых потоков ввода/вывода

Классы байтовых потоков ввода/вывода организованы в виде иерархии. Во главе этой иерархии находятся абстрактные классы `InputStream` (потоки ввода) и `OutputStream` (потоки вывода).

19.1.1. Класс *OutputStream*

Абстрактный класс `OutputStream` является базовым для байтовых потоков вывода. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.OutputStream;
```

В классе `OutputStream` объявлены следующие методы:

- `write()` — записывает данные в поток. Форматы метода:

```
public abstract void write(int b) throws IOException
public void write(byte[] b, int off, int len)
                                  throws IOException
public void write(byte[] b)       throws IOException
```

  Первый формат записывает в поток один байт, второй — `len` байтов из массива `b`, начиная с индекса `off`, а третий — все байты из массива `b`;

- `flush()` — сбрасывает данные из буфера в файл. Формат метода:

```
// Интерфейс Flushable
public void flush()               throws IOException
```

- `close()` — сбрасывает данные из буфера в файл, закрывает поток и освобождает ресурсы. Дальнейшая работа с потоком становится невозможной. Формат метода:

```
// Интерфейсы Closeable и AutoCloseable
public void close()               throws IOException
```

Как видно из форматов, все методы могут генерировать контролируемое исключение даже при закрытии потока. Поэтому необходимо предусмотреть обработку исключений и обязательно закрыть поток, даже если исключение возникло. Если поток не закрыть, то данные, помещенные в буфер (при его наличии), не будут сброшены в файл. Пример записи в файл с обработкой исключений приведен в листинге 19.1.

Листинг 19.1. Запись данных в файл с обработкой исключений

```
package com.example.app;

import java.io.*;

public class MyClass {
   public static void main(String[] args) {
      OutputStream out = null;
      try {
         try {
            out = new FileOutputStream("C:\\book\\test.txt");
            byte[] bytes = "строка".getBytes("cp1251");
            out.write(bytes[0]);
```

```
            out.write(bytes, 0, 3);
            out.write(bytes);
         }
         finally {
            if (out != null) out.close();
         }
      }
      catch (UnsupportedEncodingException e) {
         System.err.println("Проблемы с кодировкой");
         System.exit(1);
      }
      catch (IOException e) {
         System.err.println("Не удалось записать в файл");
         System.exit(1);
      }
      System.out.println("Данные успешно записаны в файл");
   }
}
```

В дальнейших примерах мы будем сокращать код и помещать базовый класс `Exception` в заголовок метода `main()` после ключевого слова `throws`:

```
public static void main(String[] args) throws Exception {
   // Примеры
}
```

19.1.2. Класс *FileOutputStream*

Класс `FileOutputStream` является реализацией абстрактного класса `OutputStream` и позволяет записывать данные в файл. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.FileOutputStream;
```

Иерархия наследования:

```
Object - OutputStream - FileOutputStream
```

Класс `FileOutputStream` реализует следующие интерфейсы:

```
Closeable, Flushable, AutoCloseable
```

Создать объект позволяют следующие конструкторы класса `FileOutputStream`:

```
FileOutputStream(String name) throws FileNotFoundException
FileOutputStream(String name, boolean append)
                                    throws FileNotFoundException
FileOutputStream(File file)   throws FileNotFoundException
FileOutputStream(File file, boolean append)
                                    throws FileNotFoundException
FileOutputStream(FileDescriptor fdObj)
```

Первый формат позволяет указать абсолютный или относительный путь к файлу в виде строки. Если файл не существует, то он будет создан. Если файл существует, то он будет перезаписан. Если не удалось создать или открыть файл на запись, то генерируется исключение:

```
OutputStream out = null;
try {
   out = new FileOutputStream("C:\\book\\test.txt");
   byte[] bytes = "строка".getBytes("cp1251");
   out.write(bytes);
}
finally {
   if (out != null) out.close();
}
```

Второй формат, помимо пути к файлу, принимает логическое значение. Если в параметре `append` указано значение `true`, то данные будут записаны в конец файла без удаления существующих данных, а если `false` — то файл будет перезаписан. Если файл не существует, то он будет создан. Добавим данные в конец файла:

```
OutputStream out = null;
try {
   out = new FileOutputStream("C:\\book\\test.txt", true);
   byte[] bytes = "\nстрока2".getBytes("cp1251");
   out.write(bytes);
}
finally {
   if (out != null) out.close();
}
```

Третий и четвертый форматы вместо строки принимают объект класса `File`:

```
OutputStream out = null;
File file = new File("C:\\book\\test.txt");
try {
   out = new FileOutputStream(file);
   byte[] bytes = "строка".getBytes("cp1251");
   out.write(bytes);
}
finally {
   if (out != null) out.close();
}
```

Пятый формат создает поток на основе дескриптора существующего потока (объект класса `FileDescriptor`). Выведем данные на консоль, а не в файл:

```
// import java.io.FileDescriptor;
OutputStream out = new FileOutputStream(FileDescriptor.out);
byte[] bytes = "строка".getBytes("utf-8");
out.write(bytes);
out = null;
```

В этом случае мы не можем вызвать метод `close()`, т. к. закроется стандартный поток вывода, и мы больше не сможем выводить данные на консоль.

Класс `FileDescriptor` содержит следующие свойства:

- `out` — содержит дескриптор стандартного потока вывода `System.out` (по умолчанию поток связан с консолью). Объявление свойства:

```
public static final FileDescriptor out
```

- `err` — содержит дескриптор стандартного потока вывода сообщений об ошибках `System.err` (по умолчанию поток связан с консолью). Объявление свойства:

```
public static final FileDescriptor err
```

- `in` — содержит дескриптор стандартного потока ввода `System.in` (по умолчанию поток связан с консолью). Объявление свойства:

```
public static final FileDescriptor in
```

Класс `FileOutputStream` наследует все методы из класса `OutputStream` и реализует их. Кроме того, он содержит два дополнительных метода:

- `getFD()` — возвращает дескриптор потока. Формат метода:

```
public final FileDescriptor getFD() throws IOException
```

- `getChannel()` — возвращает `FileChannel` объект, связанный с потоком. Формат метода:

```
public FileChannel getChannel()
```

19.1.3. Класс *InputStream*

Абстрактный класс `InputStream` является базовым для байтовых потоков ввода. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.InputStream;
```

В классе `InputStream` объявлены следующие методы:

- `read()` — читает данные из потока. Форматы метода:

```
public abstract int read() throws IOException
public int read(byte[] b, int off, int len)
                          throws IOException
public int read(byte[] b)  throws IOException
```

Первый формат читает из потока один байт и возвращает его, если достигнут конец потока, метод вернет значение `-1`. Пример чтения всего файла по одному байту:

```
// import java.nio.file.*;
byte[] bytes = "string".getBytes();
Files.write(Paths.get("C:\\book\\test.txt"), bytes);
```

```
InputStream in = null;
try {
   in = new FileInputStream("C:\\book\\test.txt");
   int c;
   while ((c = in.read()) != -1) {
      System.out.print( (char)c + " " );
   } // s t r i n g
}
finally {
   if (in != null) in.close();
}
```

Второй формат читает из потока `len` байтов и записывает их в массив `b`, начиная с индекса `off`. Метод возвращает количество прочтенных байтов или значение `-1`, если достигнут конец потока. Пример считывания по пять байтов:

```
byte[] buffer = new byte[5];
InputStream in = null;
try {
   in = new FileInputStream("C:\\book\\test.txt");
   int n;
   while ((n = in.read(buffer, 0, 5)) != -1) {
      for (int i = 0; i < n; i++) {
         System.out.print( (char)buffer[i] + " " );
      }
   } // s t r i n g
}
finally {
   if (in != null) in.close();
}
```

Третий формат читает из потока количество байтов, равное размеру массива `b` или меньшее его, и записывает их в массив `b`. Метод возвращает количество прочтенных байтов или значение `-1`, если достигнут конец потока. Прочтем все доступные байты из файла в массив:

```
InputStream in = null;
try {
   in = new FileInputStream("C:\\book\\test.txt");
   byte[] buffer = new byte[in.available()];
   in.read(buffer);
   for (int i = 0; i < buffer.length; i++) {
      System.out.print( (char)buffer[i] + " " );
   } // s t r i n g
}
finally {
   if (in != null) in.close();
}
```

- `available()` — возвращает количество байтов, которое можно прочитать без блокировки текущего процесса (если ресурс, с которым связан поток, не отвечает, то текущий процесс блокируется до момента получения данных). Формат метода:

```
public int available() throws IOException
```

- `readAllBytes()` — читает все данные из потока и возвращает массив. Метод доступен, начиная с Java 9. Формат метода:

```
public byte[] readAllBytes() throws IOException
```

- `readNBytes()` — читает из потока `len` байтов и записывает их в массив `b`, начиная с индекса `off`. Метод возвращает количество прочтенных байтов или значение `0`, если достигнут конец потока. Метод доступен, начиная с Java 9. Формат метода:

```
public int readNBytes(byte[] b, int off, int len)
                              throws IOException
```

- `transferTo()` — читает все данные из потока и записывает их в выходной поток. Метод возвращает количество переданных байтов. Метод доступен, начиная с Java 9. Формат метода:

```
public long transferTo(OutputStream out)
                              throws IOException
```

Пример:

```
InputStream in = null;
OutputStream out = null;
try {
   in = new FileInputStream("C:\\book\\test.txt");
   out = new FileOutputStream("C:\\book\\test2.txt");
   System.out.println(in.transferTo(out));
}
finally {
   if (in != null) in.close();
   if (out != null) out.close();
}
```

- `skip()` — пропускает указанное количество байтов и возвращает фактическое количество пропущенных байтов. Формат метода:

```
public long skip(long n) throws IOException
```

- `markSupported()` — возвращает значение `true`, если класс потока поддерживает возможность установки маркеров, и `false` — в противном случае. Формат метода:

```
public boolean markSupported()
```

- `mark()` — устанавливает маркер на текущей позиции. После прочтения `readlimit` байтов поток может забыть о существовании маркера. Формат метода:

```
public void mark(int readlimit)
```

- `reset()` — возвращается к позиции, помеченной маркером. Если класс не поддерживает возможность установки маркера (метод `markSupported()` возвращает `false`), то класс может генерировать исключение. Формат метода:

```
public void reset() throws IOException
```

- `close()` — закрывает поток и освобождает ресурсы. Формат метода:

```
// Интерфейсы Closeable и AutoCloseable
public void close() throws IOException
```

Как видно из форматов, все методы могут генерировать контролируемое исключение даже при закрытии потока. Поэтому необходимо предусмотреть обработку исключений. Пример чтения из файла в кодировке windows-1251 с обработкой исключений приведен в листинге 19.2.

Листинг 19.2. Чтение из файла с обработкой исключений

```
package com.example.app;

import java.io.*;

public class MyClass {
   public static void main(String[] args) {
      InputStream in = null;
      try {
         try {
            in = new FileInputStream("C:\\book\\cp1251.txt");
            byte[] buffer = new byte[in.available()];
            in.read(buffer);
            String s = new String(buffer, "cp1251");
            System.out.print(s);
         }
         finally {
            if (in != null) in.close();
         }
      }
      catch (UnsupportedEncodingException e) {
         System.err.println("Проблемы с кодировкой");
         System.exit(1);
      }
      catch (IOException e) {
         System.err.println("Не удалось прочитать файл");
         System.exit(1);
      }
   }
}
```

19.1.4. Класс *FileInputStream*

Класс `FileInputStream` является реализацией абстрактного класса `InputStream` и позволяет читать данные из файла. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.FileInputStream;
```

Иерархия наследования:

```
Object - InputStream - FileInputStream
```

Класс `FileInputStream` реализует следующие интерфейсы:

```
Closeable, AutoCloseable
```

Создать объект позволяют следующие конструкторы класса `FileInputStream`:

```
FileInputStream(String name) throws FileNotFoundException
FileInputStream(File file)   throws FileNotFoundException
FileInputStream(FileDescriptor fdObj)
```

Первый формат позволяет указать абсолютный или относительный путь к файлу в виде строки. Если файл не существует, то генерируется исключение:

```
InputStream in = null;
try {
   in = new FileInputStream("C:\\book\\test.txt");
   byte[] buffer = new byte[in.available()];
   in.read(buffer);
   for (int i = 0; i < buffer.length; i++) {
      System.out.print( (char)buffer[i] + " " );
   } // s t r i n g
}
finally {
   if (in != null) in.close();
}
```

Второй формат вместо строки принимает объект класса `File`:

```
InputStream in = null;
File file = new File("C:\\book\\test.txt");
try {
   in = new FileInputStream(file);
   int c;
   while ((c = in.read()) != -1) {
      System.out.print( (char)c + " " );
   } // s t r i n g
}
finally {
   if (in != null) in.close();
}
```

Третий формат создает поток на основе дескриптора существующего потока (объект класса `FileDescriptor`). Прочитаем данные из консоли, а не из файла:

```
InputStream in = new FileInputStream(FileDescriptor.in);
System.out.println("Введите символ и нажмите <Enter>:");
int c = in.read();
System.out.print( c );
in = null;
```

Класс `FileInputStream` наследует все методы из класса `InputStream` и реализует их (кроме методов для работы с маркерами). Кроме того, он содержит два дополнительных метода:

- `getFD()` — возвращает дескриптор потока. Формат метода:

  ```
  public final FileDescriptor getFD() throws IOException
  ```

- `getChannel()` — возвращает `FileChannel` объект, связанный с потоком. Формат метода:

  ```
  public FileChannel getChannel()
  ```

19.2. Интерфейс *AutoCloseable* и инструкция *try-with-resources*

Начиная с Java 7, в синтаксис языка был добавлен интерфейс `AutoCloseable` с единственным методом `close()`. Совместно с этим интерфейсом в язык была добавлена инструкция `try-with-resources`. Формат инструкции:

```
try (Создание потока 1[; ...; Создание потока N]) {
    <Блок, в котором перехватывается исключение>
}
[catch (<Класс исключения 1> <Переменная>) {
    <Блок, выполняемый при возникновении исключения>
}
[...
catch (<Класс исключения N> <Переменная>) {
    <Блок, выполняемый при возникновении исключения>
}]]
[finally {
    <Блок, выполняемый в любом случае>
}]
```

Инструкция `try-with-resources` похожа на инструкцию `try...catch...finally`, но дополнительно содержит выражение внутри круглых скобок после ключевого слова `try`. Если внутри круглых скобок создать экземпляр класса, реализующего интерфейс `AutoCloseable`, то после выхода из блока `try` будет автоматически вызван метод `close()`, — вне зависимости от того, возникло внутри блока `try` исключение или нет. Можно создать сразу несколько экземпляров, разделив инструкции точкой

с запятой. Следует учитывать, что переменные будут видны только внутри блока try, — в других блоках доступа к ним нет, и все ресурсы будут закрыты еще до передачи управления другим блокам.

Переделаем программу из листинга 19.1 и используем инструкцию try-with-resources (листинг 19.3).

Листинг 19.3. Инструкция try-with-resources

```
package com.example.app;

import java.io.*;

public class MyClass {
   public static void main(String[] args) {
      try (OutputStream out =
              new FileOutputStream("C:\\book\\test.txt")) {
         byte[] bytes = "строка".getBytes("cp1251");
         out.write(bytes);
      }
      catch (UnsupportedEncodingException e) {
         System.err.println("Проблемы с кодировкой");
         System.exit(1);
      }
      catch (IOException e) {
         System.err.println("Не удалось записать в файл");
         System.exit(1);
      }
      System.out.println("Данные успешно записаны в файл");
   }
}
```

Начиная с Java 9, переменную можно объявить вне круглых скобок, а внутри круглых скобок указать только название переменной. В этом случае поток также будет закрыт автоматически. Например, эту инструкцию из листинга 19.3:

```
try (OutputStream out =
        new FileOutputStream("C:\\book\\test.txt")) {
```

можно записать так:

```
OutputStream out = new FileOutputStream("C:\\book\\test.txt");
try (out) {
```

В этом случае исключение, которое может быть сгенерировано при создании объекта, нужно обрабатывать дополнительно.

19.3. Буферизованные байтовые потоки

Базовые байтовые потоки ввода и вывода не содержат буфер. Добавить буфер позволяют классы `BufferedInputStream` (для потока ввода) и `BufferedOutputStream` (для потока вывода). Благодаря использованию буфера повышается производительность при вводе и выводе данных. Согласитесь, что обращаться каждый раз к файлу для получения всего одного байта просто расточительно. Гораздо эффективнее прочитать сразу фрагмент файла в буфер (по существу — в массив в памяти), а затем уже считывать из него по одному байту. Может возникнуть вопрос, почему бы сразу не считывать весь файл в массив? Все очень просто. Файл может быть очень большим, и оперативной памяти просто не хватит. Лучше читать файл фрагментами, используя при этом буфер.

19.3.1. Класс *BufferedOutputStream*

Класс `BufferedOutputStream` добавляет буфер для потока вывода. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.BufferedOutputStream;
```

Иерархия наследования:

```
Object - OutputStream - FilterOutputStream - BufferedOutputStream
```

Класс `BufferedOutputStream` реализует следующие интерфейсы:

```
Closeable, Flushable, AutoCloseable
```

Создать объект позволяют следующие конструкторы класса `BufferedOutputStream`:

```
BufferedOutputStream(OutputStream out)
BufferedOutputStream(OutputStream out, int size)
```

Конструкторы в первом параметре принимают объект потока вывода. Второй конструктор позволяет указать размер буфера, при достижении которого данные будут записаны в выходной поток. С помощью метода `flush()` можно сбросить данные в выходной поток, не дожидаясь заполнения буфера. При использовании буфера важно закрывать поток явным образом, иначе данные могут так и остаться в буфере.

Класс `BufferedOutputStream` наследует все методы из класса `OutputStream`, реализует их и не добавляет никаких новых методов:

```
OutputStream out = null;
BufferedOutputStream buf = null;
try {
   out = new FileOutputStream("C:\\book\\test.txt");
   buf = new BufferedOutputStream(out);
   byte[] bytes = "строка".getBytes("cp1251");
   buf.write(bytes);
   buf.flush();        // Сбрасываем содержимое буфера
}
```

```
finally {
   if (buf != null) buf.close();
}
```

19.3.2. Класс *BufferedInputStream*

Класс `BufferedInputStream` добавляет буфер для потока ввода. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.BufferedInputStream;
```

Иерархия наследования:

```
Object - InputStream - FilterInputStream - BufferedInputStream
```

Класс `BufferedInputStream` реализует следующие интерфейсы:

```
Closeable, AutoCloseable
```

Создать объект позволяют следующие конструкторы класса `BufferedInputStream`:

```
BufferedInputStream(InputStream in)
BufferedInputStream(InputStream in, int size)
```

Конструкторы в первом параметре принимают объект потока ввода. Второй конструктор дополнительно позволяет указать размер буфера. Класс `BufferedInputStream` наследует все методы из класса `InputStream`, реализует их и не добавляет никаких новых методов. В частности, класс реализует методы для работы с маркерами, следовательно, мы можем пометить позицию маркером, потом прочитать данные, а затем вернуться к запомненной позиции и прочитать данные еще раз:

```
// import java.nio.file.*;
InputStream in = null;
BufferedInputStream buf = null;
try {
   byte[] bytes = "string1\nstring2".getBytes();
   Files.write(Paths.get("C:\\book\\test.txt"), bytes);
   in = new FileInputStream("C:\\book\\test.txt");
   buf = new BufferedInputStream(in);
   buf.mark(10); // Устанавливаем маркер
   for (int i = 0; i < 7; i++) {
      System.out.print( (char)buf.read() + " ");
   } // s t r i n g 1
   buf.reset(); // Возвращаемся к позиции маркера
   for (int i = 0; i < 7; i++) {
      System.out.print( (char)buf.read() + " ");
   } // s t r i n g 1
}
finally {
   if (buf != null) buf.close();
}
```

19.4. Класс *PushbackInputStream*

Класс `PushbackInputStream` позволяет возвращать байты в поток ввода после прочтения. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.PushbackInputStream;
```

Иерархия наследования:

```
Object - InputStream - FilterInputStream - PushbackInputStream
```

Класс `PushbackInputStream` реализует следующие интерфейсы:

```
Closeable, AutoCloseable
```

Создать объект позволяют следующие конструкторы класса `PushbackInputStream`:

```
PushbackInputStream(InputStream in)
PushbackInputStream(InputStream in, int size)
```

В первом параметре конструкторы принимают объект потока ввода. Второй формат позволяет дополнительно указать размер буфера. Если размер не указан, то вернуть в поток можно только один прочитанный до этого байт.

Класс `PushbackInputStream` наследует все методы из класса `InputStream`. Кроме того, он содержит дополнительный метод `unread()`, который позволяет вернуть байты в поток. Эти байты будут доступны для следующей операции чтения. Форматы метода:

```
public void unread(int b)                          throws IOException
public void unread(byte[] b, int off, int len) throws IOException
public void unread(byte[] b)                       throws IOException
```

Первый формат возвращает в поток один байт, второй формат — `len` байтов, начиная с позиции `off`, а третий формат — все байты из массива. Если буфер заполнен, то генерируется исключение:

```
try (
   InputStream in = new FileInputStream("C:\\book\\test.txt");
   PushbackInputStream push = new PushbackInputStream(in, 10);
)
{
   byte[] bytes = new byte[5];
   push.read(bytes);
   for (int i = 0; i < bytes.length; i++) {
      System.out.print( (char)bytes[i] + " " );
   } // s t r i n
   push.unread(bytes);  // Возвращаем байты в поток
   push.read(bytes);
   for (int i = 0; i < bytes.length; i++) {
      System.out.print( (char)bytes[i] + " " );
   } // s t r i n
}
```

19.5. Запись и чтение двоичных данных

Для записи и чтения файлов в двоичном формате предназначены классы `DataOutputStream` и `DataInputStream` из пакета `java.io`.

19.5.1. Класс *DataOutputStream*

Класс `DataOutputStream` позволяет записывать в поток данные в двоичном формате. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.DataOutputStream;
```

Иерархия наследования:

```
Object - OutputStream - FilterOutputStream - DataOutputStream
```

Класс `DataOutputStream` реализует следующие интерфейсы:

```
Closeable, Flushable, AutoCloseable, DataOutput
```

Создать объект позволяет следующий конструктор:

```
DataOutputStream(OutputStream out)
```

Класс `DataOutputStream` наследует все методы из класса `OutputStream`. Кроме того, он содержит методы из интерфейса `DataOutput` (см. *разд. 19.5.2*) и метод `size()`, возвращающий количество записанных байтов. Формат метода:

```
public final int size()
```

19.5.2. Интерфейс *DataOutput*

Интерфейс `DataOutput` содержит следующие методы:

```
public void writeBoolean(boolean v)
public void writeByte(int v)
public void write(int b)
public void write(byte[] b)
public void write(byte[] b, int off, int len)
public void writeShort(int v)
public void writeInt(int v)
public void writeLong(long v)
public void writeFloat(float v)
public void writeDouble(double v)
public void writeChar(int v)
public void writeBytes(String s)
public void writeChars(String s)
public void writeUTF(String s)
```

Большинство из этих методов не требуют дополнительных комментариев. Они записывают в поток данные элементарных типов, названия которых указаны в имени метода. Причем данные будут записаны в двоичном формате.

Метод `writeBytes()` преобразует строку в массив байтов, отбрасывая первые восемь битов из кода каждого символа. Учитывая, что русские буквы кодируются двумя байтами, при использовании этого метода будут возникать проблемы.

При использовании метода `writeChars()` к каждому символу в строке будет применен метод `writeChar()`.

Метод `writeUTF()` преобразует строку в массив байтов в модифицированной версии кодировки UTF-8 (подробности реализации смотрите в документации) и записывает его в поток. Русские буквы кодируются двумя байтами. Перед записью массива вставляются два байта, в которых с помощью метода `writeShort()` кодируется количество байтов в массиве.

Запишем данные различных типов в двоичный файл:

```
try (
   OutputStream out = new FileOutputStream("C:\\book\\data.txt");
   BufferedOutputStream buf = new BufferedOutputStream(out);
   DataOutputStream data = new DataOutputStream(buf);
)
{
   data.writeBoolean(false);
   data.writeByte(100);
   data.writeShort(45);
   data.writeInt(1254);
   data.writeLong(454L);
   data.writeFloat(42.2f);
   data.writeDouble(45.4);
   data.writeChar('Я');
   System.out.println(data.size());
}
```

19.5.3. Класс *DataInputStream*

Класс `DataInputStream` позволяет читать данные из двоичного файла. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.DataInputStream;
```

Иерархия наследования:

```
Object - InputStream - FilterInputStream - DataInputStream
```

Класс `DataInputStream` реализует следующие интерфейсы:

```
Closeable, DataInput, AutoCloseable
```

Создать объект позволяет следующий конструктор:

```
DataInputStream(InputStream in)
```

Класс `DataInputStream` наследует все методы из класса `InputStream`. Кроме того, он содержит методы из интерфейса `DataInput` (см. *разд. 19.5.4*) и статический метод

readUTF(), позволяющий читать данные в модифицированной версии кодировки UTF-8. Формат метода:

```
public static final String readUTF(DataInput in) throws IOException
```

Запишем строку в файл, а затем прочитаем ее из файла:

```
try (
   OutputStream out = new FileOutputStream("C:\\book\\test.txt");
   BufferedOutputStream buf = new BufferedOutputStream(out);
   DataOutputStream data = new DataOutputStream(buf);
)
{
   data.writeUTF("Строка1\nСтрока2");
   System.out.println(data.size());
}

try (
   InputStream in = new FileInputStream("C:\\book\\test.txt");
   BufferedInputStream buf = new BufferedInputStream(in);
   DataInputStream data = new DataInputStream(buf);
)
{
   System.out.println(DataInputStream.readUTF(data));
}
```

19.5.4. Интерфейс *DataInput*

Интерфейс DataInput содержит следующие методы:

```
public boolean readBoolean()
public byte readByte()
public void readFully(byte[] b)
public void readFully(byte[] b, int off, int len)
public int readUnsignedByte()
public short readShort()
public int readUnsignedShort()
public int readInt()
public long readLong()
public float readFloat()
public double readDouble()
public char readChar()
public String readLine()
public String readUTF()
public int skipBytes(int n)
```

Большинство из этих методов не требуют дополнительных комментариев. Они читают из потока данные элементарных типов, названия которых указаны в имени метода. Методы readUnsignedByte() и readUnsignedShort() возвращают значения без знака в диапазонах 0–255 и 0–65 535 соответственно. Метод readLine() объяв-

лен в классе DataInputStream как устаревший, вместо него нужно использовать символьные потоки. Метод readUTF() позволяет читать данные в модифицированной версии кодировки UTF-8 (подробности реализации смотрите в документации), а метод skipBytes() пропускает указанное количество байтов.

Прочитаем двоичные данные из файла data.txt, который мы записали в *разд. 19.5.2*:

```
try (
   InputStream in = new FileInputStream("C:\\book\\data.txt");
   BufferedInputStream buf = new BufferedInputStream(in);
   DataInputStream data = new DataInputStream(buf);
)
{
   System.out.println(data.readBoolean()); // false
   System.out.println(data.readByte());    // 100
   System.out.println(data.readShort());   // 45
   System.out.println(data.readInt());     // 1254
   System.out.println(data.readLong());    // 454
   System.out.println(data.readFloat());   // 42.2
   System.out.println(data.readDouble());  // 45.4
   System.out.println(data.readChar());    // Я
}
```

19.6. Сериализация объектов

В предыдущем разделе мы научились сохранять в файл значения элементарных типов. С помощью классов ObjectOutputStream и ObjectInputStream мы можем, помимо элементарных типов, сохранять и читать данные объектных типов. Процесс сохранения объектов называется *сериализацией*.

Однако сохранить можно не все объекты, а только объекты, классы которых реализуют интерфейс Serializable. Этот интерфейс не содержит объявлений методов, он является лишь маркером, означающим, что для объекта разрешена сериализация. Если бы можно было сохранять любые объекты, то значения приватных полей можно было бы перезаписать перед восстановлением объектов, кроме того, некоторые поля имели бы идентификаторы несуществующих устройств. Пример указания интерфейса Serializable:

```
class Class1 implements Serializable {
}
```

Следует учитывать, что статические члены класса принадлежат классу, а не объекту, и сериализации не подлежат. Если мы хотим ограничить сериализацию отдельных полей (например, содержащих значения дескрипторов файлов или ссылки на другие ресурсы, которые будут недействительными при восстановлении объекта), то при объявлении поля необходимо указать ключевое слово transient. Например, мы явно не хотим, чтобы пароль был сериализован и стал доступен всем:

```
private transient String password;
```

При восстановлении объекта такие поля получат значения по умолчанию. В нашем случае поле `password` получит значение `null`.

При сериализации происходит сохранение всей иерархии наследования классов. Если мы хотим предотвратить это, то следует внутри класса определить методы `writeObject()` и `readObject()`. Если методы определены, то они будут вызываться при сериализации и десериализации. Форматы методов:

```
private void writeObject(ObjectOutputStream out) throws IOException
private void readObject(ObjectInputStream in)
                        throws IOException, ClassNotFoundException
```

Внутри метода `writeObject()` мы можем записывать в поток данные элементарных типов из полей, используя методы из интерфейса `DataOutput`, а внутри метода `readObject()` — читать данные, используя методы из интерфейса `DataInput`, и записывать эти значения в поля класса.

Внутри класса, поддерживающего сериализацию, необходимо также объявить закрытую статическую константу `serialVersionUID`. Пример объявления константы:

```
private static final long serialVersionUID = 2350430129341501141L;
```

Это значение сохраняется при сериализации и проверяется при восстановлении объекта. Если значения не совпадают, то генерируется исключение. Если константа не существует в классе, то это значение генерируется автоматически. Однако на разных компьютерах это автоматическое значение может оказаться разным, что может привести к исключению даже при правильной работе процесса сериализации. При отсутствии константы редактор Eclipse подчеркивает название класса волнистой линией. Если навести указатель мыши на название класса, то во всплывающем окне будет предложено либо добавить константу со значением по умолчанию, либо сгенерировать ее значение автоматически, либо добавить аннотацию `@SuppressWarnings`:

```
@SuppressWarnings("serial")
```

19.6.1. Класс *ObjectOutputStream*

Класс `ObjectOutputStream` позволяет записывать в поток данные элементарных и объектных типов. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.ObjectOutputStream;
```

Иерархия наследования:

```
Object - OutputStream - ObjectOutputStream
```

Класс `ObjectOutputStream` реализует следующие интерфейсы:

```
Closeable, DataOutput, Flushable, ObjectOutput, ObjectStreamConstants,
AutoCloseable
```

Создать объект позволяет следующий конструктор:

```
ObjectOutputStream(OutputStream out)
```

Класс `ObjectOutputStream` наследует все методы из класса `OutputStream`. Кроме того, он содержит методы из интерфейсов `DataOutput` (см. *разд. 19.5.2*) и `ObjectOutput` (см. *разд. 19.6.2*).

19.6.2. Интерфейс *ObjectOutput*

Интерфейс `ObjectOutput` содержит следующие методы:

```
public void close()
public void flush()
public void write(int b)
public void write(byte[] b)
public void write(byte[] b, int off, int len)
public void writeObject(Object obj)
```

Все эти методы, кроме последнего, определены в базовом классе `OutputStream`. Метод `writeObject()` применяется для сохранения объектов, классы которых реализуют интерфейс `Serializable`.

19.6.3. Класс *ObjectInputStream*

Класс `ObjectInputStream` позволяет читать из потока данные элементарных и объектных типов, сохраненные ранее в двоичном формате. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.ObjectInputStream;
```

Иерархия наследования:

```
Object - InputStream - ObjectInputStream
```

Класс `ObjectInputStream` реализует следующие интерфейсы:

```
Closeable, DataInput, ObjectInput, ObjectStreamConstants, AutoCloseable
```

Создать объект позволяет следующий конструктор:

```
ObjectInputStream(InputStream in)
```

Класс `ObjectInputStream` наследует все методы из класса `InputStream`. Кроме того, он содержит методы из интерфейсов `DataInput` (см. *разд. 19.5.4*) и `ObjectInput` (см. *разд. 19.6.4*).

19.6.4. Интерфейс *ObjectInput*

Интерфейс `ObjectInput` содержит следующие методы:

```
public int available()
public void close()
public int read()
public int read(byte[] b)
public int read(byte[] b, int off, int len)
```

```
public long skip(long n)
public Object readObject()
```

Все эти методы, кроме последнего, определены в базовом классе `InputStream`. Метод `readObject()` применяется для восстановления объектов, классы которых реализуют интерфейс `Serializable`. После восстановления необходимо выполнить приведение типов.

В качестве примера запишем два объекта в файл, а затем прочитаем их из файла (листинг 19.4).

Листинг 19.4. Сериализация и десериализация объектов

```
package com.example.app;

import java.io.*;

public class MyClass {
   public static void main(String[] args) throws Exception {
      try (
         OutputStream out = new FileOutputStream("C:\\book\\test.txt");
         ObjectOutputStream data = new ObjectOutputStream(out);
      )
      {
         Class1 obj1 = new Class1(20);
         Class1 obj2 = new Class1(88);
         data.writeObject(obj1);
         data.writeObject(obj2);
      }

      try (
         InputStream in = new FileInputStream("C:\\book\\test.txt");
         ObjectInputStream data = new ObjectInputStream(in);
      )
      {
         Class1 obj3 = (Class1)data.readObject();
         Class1 obj4 = (Class1)data.readObject();
         System.out.println(obj3.getX());
         System.out.println(obj4.getX());
      }
   }
}

class Class1 implements Serializable {
   private static final long serialVersionUID =
                                 4159958489342586369L;
   private int x;
```

```
    public Classl(int x) {
        this.x = x;
    }
    public int getX() {
        return this.x;
    }
    public void setX(int x) {
        this.x = x;
    }
}
```

19.7. Файлы с произвольным доступом

Класс `RandomAccessFile` позволяет записывать в файл данные в двоичном формате и читать их. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.RandomAccessFile;
```

Иерархия наследования:

```
Object - RandomAccessFile
```

Класс `RandomAccessFile` реализует следующие интерфейсы:

```
Closeable, AutoCloseable, DataInput, DataOutput
```

Создать объект позволяют следующие конструкторы:

```
RandomAccessFile(String name, String mode)
RandomAccessFile(File file, String mode)
```

В первом параметре конструкторы принимают абсолютный или относительный путь к файлу в виде строки или объекта класса `File`. Во втором параметре задается режим открытия файла: `"r"`, `"rw"`, `"rws"` или `"rwd"`. Если указан режим `"r"`, то файл открывается только на чтение, и указатель устанавливается на начало файла. Если файл не существует, то генерируется исключение. Если указан режим `"rw"`, то файл открывается на чтение и запись. Если файл не существует, то он будет создан, и указатель в нем установится на начало файла. Если необходимо добавить данные к уже существующим, то не забудьте установить указатель на конец файла.

Обратите внимание: класс `RandomAccessFile` не наследует классы `OutputStream` и `InputStream`, но реализует интерфейсы `DataOutput` (см. *разд. 19.5.2*) и `DataInput` (см. *разд. 19.5.4*). Кроме того, он содержит несколько дополнительных методов:

- `length()` — возвращает длину файла в байтах. Формат метода:

  ```
  public long length()                    throws IOException
  ```

- `setLength()` — задает длину файла в байтах. Если значение меньше текущей длины файла, то он будет обрезан до указанного значения, а если больше — то будут добавлены новые байты с нулевым значением. Формат метода:

  ```
  public void setLength(long newLength) throws IOException
  ```

- `getFilePointer()` — возвращает текущую позицию указателя внутри файла. Формат метода:

```
public long getFilePointer()          throws IOException
```

- `seek()` — перемещает указатель в указанную позицию: `seek(0)` — в начало файла, `seek(file.length())` — в конец файла. Формат метода:

```
public void seek(long pos)            throws IOException
```

- `read()` — читает данные из потока. Форматы метода:

```
public int read()                     throws IOException
public int read(byte[] b, int off, int len)
                                      throws IOException
public int read(byte[] b)             throws IOException
```

Первый формат читает из потока один байт и возвращает его, если достигнут конец потока, метод вернет значение `-1`. Пример очистки файла, записи всех байтов из массива, перемещения в начало файла и чтения всего файла по одному байту:

```
try (RandomAccessFile file =
       new RandomAccessFile("C:\\book\\test.txt", "rw"))
{
   file.setLength(0);  // Очищаем файл
   byte[] bytes = "string".getBytes();
   file.write(bytes);  // Записываем байты
   file.seek(0);       // Перемещаемся в начало файла
   int c;
   while ((c = file.read()) != -1) {
      System.out.print( (char)c + " " );
   } // s t r i n g
}
```

Второй формат читает из потока `len` байтов и записывает их в массив `b`, начиная с индекса `off`. Метод возвращает количество прочтенных байтов или значение `-1`, если достигнут конец потока. Пример считывания по пять байтов:

```
try (RandomAccessFile file =
       new RandomAccessFile("C:\\book\\test.txt", "r"))
{
   byte[] buffer = new byte[5];
   int n;
   while ((n = file.read(buffer, 0, 5)) != -1) {
      for (int i = 0; i < n; i++) {
         System.out.print( (char)buffer[i] + " " );
      }
   } // s t r i n g
}
```

Третий формат читает из потока количество байтов, равное размеру массива `b` или меньшее его, и записывает их в массив `b`. Метод возвращает количество

прочтенных байтов или значение -1, если достигнут конец потока. Прочтем все доступные байты из файла в массив:

```
try (RandomAccessFile file =
       new RandomAccessFile("C:\\book\\test.txt", "r"))
{
   byte[] buffer = new byte[(int)file.length()];
   file.read(buffer);
   for (int i = 0; i < buffer.length; i++) {
      System.out.print( (char)buffer[i] + " " );
   } // s t r i n g
}
```

- `getFD()` — возвращает дескриптор потока. Формат метода:

```
public final FileDescriptor getFD() throws IOException
```

- `getChannel()` — возвращает `FileChannel` объект, связанный с потоком. Формат метода:

```
public final FileChannel getChannel()
```

- `close()` — закрывает файл. Формат метода:

```
// Интерфейсы Closeable и AutoCloseable
public void close()                 throws IOException
```

ГЛАВА 20

Символьные потоки ввода/вывода

В предыдущей главе мы познакомились с байтовыми потоками ввода/вывода. При работе с символами эти потоки не очень удобны, особенно при использовании кодировки UTF-8, в которой один символ может кодироваться несколькими байтами. Так, двумя байтами в кодировке UTF-8 (и в других Unicode-кодировках) кодируются русские символы. Поэтому для работы с текстовыми данными в различных кодировках в языке Java используются *символьные потоки* ввода/вывода.

20.1. Базовые классы символьных потоков ввода/вывода

Классы символьных потоков ввода/вывода организованы в виде иерархии. Во главе этой иерархии находятся абстрактные классы `Reader` (потоки ввода) и `Writer` (потоки вывода).

20.1.1. Класс *Writer*

Абстрактный класс `Writer` является базовым для символьных потоков вывода. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.Writer;
```

Класс `Writer` реализует следующие интерфейсы:

```
Closeable, Flushable, Appendable, AutoCloseable
```

В классе `Writer` объявлены следующие методы:

- `write()` — записывает символы в поток. Форматы метода:

```
public void write(int c)        throws IOException
public abstract void write(char[] cbuf, int off, int len)
                                throws IOException
public void write(char[] cbuf) throws IOException
public void write(String str)  throws IOException
```

```
public void write(String str, int off, int len)
                                  throws IOException
```

Первый формат записывает в поток один символ, второй — `len` символов из массива `cbuf`, начиная с индекса `off`, третий — все символы из массива `cbuf`, четвертый — все символы из строки, а пятый — `len` символов из строки `str`, начиная с индекса `off`;

- `append()` — записывает символы в поток и возвращает ссылку на поток. Форматы метода:

```
// Интерфейс Appendable
public Writer append(char c)            throws IOException
public Writer append(CharSequence csq, int start, int end)
                                        throws IOException
public Writer append(CharSequence csq) throws IOException
```

Первый формат записывает в поток один символ, второй — фрагмент от индекса `start` до индекса `end` (не включая символ с этим индексом) из объекта `csq`, третий — все символы из объекта `csq`. В качестве параметра `csq` можно указать объекты классов `String`, `StringBuilder`, `StringBuffer`, `CharBuffer` и `Segment`;

- `flush()` — сбрасывает данные из буфера (при его наличии) в файл. Формат метода:

```
// Интерфейс Flushable
public abstract void flush()            throws IOException
```

- `close()` — сбрасывает данные из буфера (при его наличии) в файл, закрывает поток и освобождает ресурсы. Дальнейшая работа с потоком становится невозможной. Формат метода:

```
// Интерфейсы Closeable и AutoCloseable
public abstract void close()            throws IOException
```

Как видно из форматов, все методы могут генерировать контролируемое исключение даже при закрытии потока. Поэтому необходимо предусмотреть обработку исключений и обязательно закрыть поток, даже если исключение возникло. Если поток не закрыть, то данные, помещенные в буфер (при его наличии), не будут сброшены в файл. Пример записи в файл с обработкой исключений приведен в листинге 20.1.

Листинг 20.1. Запись данных в файл с обработкой исключений

```
package com.example.app;

import java.io.*;

public class MyClass {
   public static void main(String[] args) {
      try (
```

```
            OutputStream out =
                  new FileOutputStream("C:\\book\\cp1251.txt");
            Writer file = new OutputStreamWriter(out, "cp1251");
         )
         {
            char[] chars = "слово2".toCharArray();
            file.write(chars, 0, 5);
            file.write(' ');
            file.write(chars);
            file.write(" слово3");
            file.write(" слово4444", 0, 7);
            file.append(' ');
            StringBuilder sb = new StringBuilder();
            sb.append("слово5");
            file.append(sb);
            file.append(" слово66666", 0, 7);
         }
         catch (UnsupportedEncodingException e) {
            System.err.println("Проблемы с кодировкой");
            System.exit(1);
         }
         catch (IOException e) {
            System.err.println("Не удалось записать в файл");
            System.exit(1);
         }
         System.out.println("Данные успешно записаны в файл");
      }
}
```

В дальнейших примерах мы будем сокращать код и помещать базовый класс `Exception` в заголовок метода `main()` после ключевого слова `throws`:

```
public static void main(String[] args) throws Exception {
   // Примеры
}
```

20.1.2. Класс *OutputStreamWriter*

Класс `OutputStreamWriter` является реализацией абстрактного класса `Writer` и позволяет записывать символы в поток вывода `OutputStream`. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.OutputStreamWriter;
```

Иерархия наследования:

```
Object - Writer - OutputStreamWriter
```

Класс `OutputStreamWriter` реализует следующие интерфейсы:

```
Closeable, Flushable, Appendable, AutoCloseable
```

Создать объект позволяют следующие конструкторы класса `OutputStreamWriter`:

```
OutputStreamWriter(OutputStream out)
OutputStreamWriter(OutputStream out, String charsetName)
                          throws UnsupportedEncodingException
OutputStreamWriter(OutputStream out, Charset cs)
OutputStreamWriter(OutputStream out, CharsetEncoder enc)
```

Первый конструктор использует кодировку, установленную в системе по умолчанию. Старайтесь избегать этого конструктора при работе с русскими символами, если не хотите получить множество проблем с кодировками. Во-первых, в разных операционных системах кодировка разная, а во-вторых, кодировку можно изменить из программы и из командной строки.

Второй формат во втором параметре позволяет указать кодировку в виде строки:

```
try (
   OutputStream out = new FileOutputStream("C:\\book\\test.txt");
   Writer file = new OutputStreamWriter(out, "utf-8");
)
{
   file.write("строка");
}
```

Третий формат во втором параметре принимает объект класса `Charset`:

```
// import java.nio.charset.Charset;
try (
   OutputStream out = new FileOutputStream("C:\\book\\test.txt");
   Writer file =
          new OutputStreamWriter(out, Charset.forName("cp1251"));
)
{
   file.write("строка");
}
```

Во втором параметре можно также указать следующие статические константы из класса `StandardCharsets`:

```
import java.nio.charset.StandardCharsets;
public static final Charset US_ASCII
public static final Charset ISO_8859_1
public static final Charset UTF_8
public static final Charset UTF_16BE
public static final Charset UTF_16LE
public static final Charset UTF_16
```

Пример записи строки в кодировке UTF-8:

```
try (
   OutputStream out = new FileOutputStream("C:\\book\\test.txt");
   Writer file =
             new OutputStreamWriter(out, StandardCharsets.UTF_8);
)
```

```
{
   file.write("строка");
}
```

Четвертый формат следует использовать, только если необходим полный контроль над процессом преобразования кодировок. Для этого надо реализовать абстрактный класс `CharsetEncoder`.

Класс `OutputStreamWriter` наследует все методы из класса `Writer` и реализует их. Кроме того, он содержит метод `getEncoding()`, который возвращает кодировку потока в виде строки. Формат метода:

```
public String getEncoding()
```

20.1.3. Класс *Reader*

Абстрактный класс `Reader` является базовым для символьных потоков ввода. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.Reader;
```

Класс `Reader` реализует следующие интерфейсы:

```
Closeable, AutoCloseable, Readable
```

В классе `Reader` объявлены следующие методы:

- `read()` — читает данные из потока. Форматы метода:

```
public int read()                    throws IOException
public abstract int read(char[] cbuf, int off, int len)
                                     throws IOException
public int read(char[] cbuf)         throws IOException
// Интерфейс Readable
public int read(CharBuffer target) throws IOException
```

Первый формат читает из потока один символ и возвращает его код. Если достигнут конец потока, метод вернет значение `-1`. Пример чтения всего файла по одному символу:

```
try (
   InputStream in = new FileInputStream("C:\\book\\cp1251.txt");
   Reader file = new InputStreamReader(in, "cp1251");
)
{
   int c;
   while ((c = file.read()) != -1) {
      System.out.print( (char)c );
   } // слово слово2 слово3 слово4 слово5 слово6
}
```

Второй формат читает из потока len символов и записывает их в массив cbuf, начиная с индекса off. Метод возвращает количество прочтенных символов или значение -1, если достигнут конец потока. Пример считывания по пять символов:

```
try (
   InputStream in = new FileInputStream("C:\\book\\cp1251.txt");
   Reader file = new InputStreamReader(in, "cp1251");
)
{
   char[] buffer = new char[5];
   int n;
   while ((n = file.read(buffer, 0, 5)) != -1) {
      for (int i = 0; i < n; i++) {
         System.out.print( (char)buffer[i] );
      }
   } // слово слово2 слово3 слово4 слово5 слово6
}
```

Третий формат читает из потока количество символов, равное размеру массива cbuf или меньшее его, и записывает их в массив cbuf. Метод возвращает количество прочтенных символов или значение -1, если достигнут конец потока. Пример считывания по десять символов с формированием строки:

```
try (
   InputStream in = new FileInputStream("C:\\book\\cp1251.txt");
   Reader file = new InputStreamReader(in, "cp1251");
)
{
   char[] buffer = new char[10];
   int n;
   StringBuilder sb = new StringBuilder();
   while ((n = file.read(buffer)) != -1) {
      sb.append(buffer, 0, n);
   }
   System.out.println(sb.toString());
   // слово слово2 слово3 слово4 слово5 слово6
}
```

Четвертый формат читает из потока количество символов, равное размеру объекта target или меньшее его, и записывает их в объект target. Метод возвращает количество прочтенных символов или значение -1, если достигнут конец потока. Пример считывания по десять символов с помощью объекта класса CharBuffer с формированием строки:

```
// import java.nio.CharBuffer;
try (
   InputStream in = new FileInputStream("C:\\book\\cp1251.txt");
   Reader file = new InputStreamReader(in, "cp1251");
)
```

```
{
   CharBuffer buffer = CharBuffer.allocate(10);
   int n;
   StringBuilder sb = new StringBuilder();
   while ((n = file.read(buffer)) != -1) {
      sb.append(buffer.array(), 0, n);
      buffer.clear();
   }
   System.out.println(sb.toString());
   // слово слово2 слово3 слово4 слово5 слово6
}
```

- `ready()` — возвращает значение `true`, если поток содержит символы, которые могут быть прочитаны методом `read()` без блокировки, и `false` — в противном случае. Формат метода:

  ```
  public boolean ready() throws IOException
  ```

- `skip()` — пропускает указанное количество символов и возвращает фактическое количество пропущенных символов. Формат метода:

  ```
  public long skip(long n) throws IOException
  ```

- `markSupported()` — возвращает значение `true`, если класс потока поддерживает возможность установки маркеров, и `false` — в противном случае. Формат метода:

  ```
  public boolean markSupported()
  ```

- `mark()` — устанавливает маркер на текущей позиции. После прочтения `readAheadLimit` символов поток может забыть о существовании маркера. Формат метода:

  ```
  public void mark(int readAheadLimit) throws IOException
  ```

- `reset()` — возвращается к позиции, помеченной маркером. Если класс не поддерживает возможность установки маркера (метод `markSupported()` возвращает `false`), то класс может генерировать исключение. Формат метода:

  ```
  public void reset() throws IOException
  ```

- `transferTo()` — читает все символы из потока ввода и записывает их в поток вывода. Метод доступен, начиная с Java 10. Формат метода:

  ```
  public long transferTo(Writer out) throws IOException
  ```

  Пример:

  ```
  try (
     InputStream in = new FileInputStream("C:\\book\\cp1251.txt");
     Reader file = new InputStreamReader(in, "cp1251");
     OutputStream out =
           new FileOutputStream("C:\\book\\test2.txt");
     Writer file2 = new OutputStreamWriter(out, "cp1251");
  )
  ```

```
    {
        file.transferTo(file2);
    }
```

- `close()` — закрывает поток и освобождает ресурсы. Формат метода:

```
// Интерфейсы Closeable и AutoCloseable
public abstract void close() throws IOException
```

Как видно из форматов, все методы могут генерировать контролируемое исключение даже при закрытии потока. Поэтому необходимо предусмотреть обработку исключений. Пример чтения из файла в кодировке windows-1251 с обработкой исключений приведен в листинге 20.2.

Листинг 20.2. Чтение из файла с обработкой исключений

```
package com.example.app;

import java.io.*;

public class MyClass {
    public static void main(String[] args) {
        try (
            InputStream in = new FileInputStream("C:\\book\\cp1251.txt");
            Reader file = new InputStreamReader(in, "cp1251");
        )
        {
            char[] buffer = new char[10];
            int n;
            while ((n = file.read(buffer)) != -1) {
                System.out.print(String.valueOf(buffer, 0, n));
            }
        }
        catch (UnsupportedEncodingException e) {
            System.err.println("Проблемы с кодировкой");
            System.exit(1);
        }
        catch (IOException e) {
            System.err.println("Не удалось прочитать файл");
            System.exit(1);
        }
    }
}
```

20.1.4. Класс *InputStreamReader*

Класс `InputStreamReader` является реализацией абстрактного класса `Reader` и позволяет читать символы из потока `InputStream`. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.InputStreamReader;
```

Иерархия наследования:

```
Object - Reader - InputStreamReader
```

Класс `InputStreamReader` **реализует следующие интерфейсы:**

```
Closeable, AutoCloseable, Readable
```

Создать объект позволяют следующие конструкторы класса `InputStreamReader`:

```
InputStreamReader(InputStream in)
InputStreamReader(InputStream in, String charsetName)
                  throws UnsupportedEncodingException
InputStreamReader(InputStream in, Charset cs)
InputStreamReader(InputStream in, CharsetDecoder dec)
```

Первый конструктор использует кодировку, установленную в системе по умолчанию. Старайтесь избегать этого конструктора при работе с русскими символами, если не хотите получить множество проблем с кодировками. Во-первых, в разных операционных системах кодировка разная, а во-вторых, кодировку можно изменить из программы и из командной строки.

Второй формат во втором параметре позволяет указать кодировку в виде строки:

```
try (
   InputStream in = new FileInputStream("C:\\book\\cp1251.txt");
   Reader file = new InputStreamReader(in, "cp1251");
)
{
   System.out.println( (char)file.read() );
}
```

Третий формат во втором параметре принимает объект класса `Charset`:

```
try (
   InputStream in = new FileInputStream("C:\\book\\cp1251.txt");
   Reader file =
          new InputStreamReader(in, Charset.forName("cp1251"));
)
{
   System.out.println( (char)file.read() );
}
```

Четвертый формат следует использовать, только если необходим полный контроль над процессом преобразования кодировок. Для этого нужно реализовать абстрактный класс `CharsetDecoder`.

Класс `InputStreamReader` **наследует все методы из класса** `Reader` **и реализует их (кроме методов для работы с маркерами). Помимо того, он содержит метод** `getEncoding()`, **который возвращает кодировку потока в виде строки. Формат метода:**

```
public String getEncoding()
```

20.2. Буферизованные символьные потоки ввода/вывода

Базовые символьные потоки ввода и вывода не содержат буфер. Добавить буфер позволяют классы `BufferedReader` (для потока ввода) и `BufferedWriter` (для потока вывода). Благодаря использованию буфера повышается производительность при вводе и выводе данных.

20.2.1. Класс *BufferedWriter*

Класс `BufferedWriter` добавляет буфер для потока вывода. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.BufferedWriter;
```

Иерархия наследования:

```
Object - Writer - BufferedWriter
```

Класс `BufferedWriter` реализует следующие интерфейсы:

```
Closeable, Flushable, Appendable, AutoCloseable
```

Создать объект позволяют следующие конструкторы класса `BufferedWriter`:

```
BufferedWriter(Writer out)
BufferedWriter(Writer out, int sz)
```

Конструкторы в первом параметре принимают объект потока вывода. Второй конструктор позволяет указать размер буфера, при достижении которого данные будут записаны в выходной поток. С помощью метода `flush()` можно сбросить данные в выходной поток, не дожидаясь заполнения буфера. При использовании буфера важно закрывать поток явным образом, иначе данные могут так и остаться в буфере.

Класс `BufferedWriter` наследует все методы из класса `Writer`, реализует их и добавляет метод `newLine()`, который выводит в поток разделитель строк в зависимости от используемой операционной системы (в Windows — `\r\n`). Формат метода:

```
public void newLine() throws IOException
```

Пример:

```
try (
   OutputStream out = new FileOutputStream("C:\\book\\test.txt");
   Writer writer = new OutputStreamWriter(out, "cp1251");
   BufferedWriter buf = new BufferedWriter(writer);
)
{
   buf.write("строка1");
   buf.newLine();        // Выводим разделитель строк
```

```
   buf.write("строка2");
   buf.flush();          // Сбрасываем содержимое буфера
}
```

Начиная с Java 7, для создания объекта класса `BufferedWriter` мы можем воспользоваться статическим методом `newBufferedWriter()` из класса `Files`. Форматы метода:

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.charset.Charset;
import java.nio.file.StandardOpenOption;
public static BufferedWriter newBufferedWriter(Path path,
                                    OpenOption... options)
                                       throws IOException
public static BufferedWriter newBufferedWriter(Path path,
                        Charset cs, OpenOption... options)
                                       throws IOException
```

В первом параметре указывается путь к файлу. Если параметр `options` не указан, то, если файл не существует, он будет создан, а если существует — перезаписан. Чтобы добавить строки к уже существующим в файле, необходимо явно указать опцию `StandardOpenOption.APPEND`. В параметре `cs` можно назначить кодировку файла. Если кодировка не задана, то используется кодировка UTF-8. Пример записи в файл:

```
try (
   BufferedWriter buf = Files.newBufferedWriter(
                        Paths.get("C:\\book\\utf8.txt"),
                        Charset.forName("utf-8"))
)
{
   buf.write("строка1");
   buf.newLine();        // Выводим разделитель строк
   buf.write("строка2");
   buf.flush();          // Сбрасываем содержимое буфера
}
```

Пример добавления строк в файл:

```
try (
   BufferedWriter buf = Files.newBufferedWriter(
                        Paths.get("C:\\book\\utf8.txt"),
                        Charset.forName("utf-8"),
                        StandardOpenOption.APPEND)
)
{
   buf.newLine();
   buf.write("строка3");
   buf.flush();
}
```

20.2.2. Класс *BufferedReader*

Класс `BufferedReader` добавляет буфер для символьного потока ввода. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.BufferedReader;
```

Иерархия наследования:

```
Object - Reader - BufferedReader
```

Класс `BufferedReader` реализует следующие интерфейсы:

```
Closeable, AutoCloseable, Readable
```

Создать объект позволяют следующие конструкторы класса `BufferedReader`:

```
BufferedReader(Reader in)
BufferedReader(Reader in, int sz)
```

Конструкторы в первом параметре принимают объект потока ввода. Второй конструктор дополнительно позволяет указать размер буфера.

Класс `BufferedReader` наследует все методы из класса `Reader`, реализует их и добавляет следующие методы:

- `readLine()` — позволяет читать из потока по одной строке. Если строк больше нет, метод возвращает значение `null`. Формат метода:

  ```
  public String readLine()                    throws IOException
  ```

  Пример чтения файла построчно:

  ```
  try (
     InputStream in = new FileInputStream("C:\\book\\test.txt");
     Reader reader = new InputStreamReader(in, "cp1251");
     BufferedReader buf = new BufferedReader(reader);
  )
  {
     String line = "";
     while ((line = buf.readLine()) != null) {
        System.out.println(line);
     }
  }
  ```

- `lines()` — возвращает поток `Stream<String>`, с помощью которого можно читать данные построчно. Метод доступен, начиная с Java 8. Формат метода:

  ```
  public Stream<String> lines()
  ```

  Пример:

  ```
  // import java.util.stream.Stream;
  try (
     InputStream in = new FileInputStream("C:\\book\\test.txt");
     Reader reader = new InputStreamReader(in, "cp1251");
  ```

```
      BufferedReader buf = new BufferedReader(reader);
   )
   {
      Stream<String> stream = buf.lines();
      stream.forEachOrdered( (line) -> System.out.println(line) );
   }
```

Класс `BufferedReader` реализует методы для работы с маркерами, следовательно, мы можем пометить позицию маркером, потом прочитать данные, а затем вернуться к запомненной позиции и прочитать данные еще раз:

```
try (
   InputStream in = new FileInputStream("C:\\book\\test.txt");
   Reader reader = new InputStreamReader(in, "cp1251");
   BufferedReader buf = new BufferedReader(reader);
)
{
   buf.mark(10); // Устанавливаем маркер
   System.out.println(buf.readLine());
   buf.reset();  // Возвращаемся к позиции маркера
   System.out.println(buf.readLine());
}
```

Начиная с Java 7, для создания объекта класса `BufferedReader` мы можем воспользоваться статическим методом `newBufferedReader()` из класса `Files`. Форматы метода:

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.charset.Charset;
public static BufferedReader newBufferedReader(Path path)
                                                  throws IOException
public static BufferedReader newBufferedReader(Path path,
                                 Charset cs) throws IOException
```

В первом параметре указывается путь к файлу, а во втором — кодировка. Если кодировка не указана, то используется кодировка UTF-8. Пример построчного чтения файла в кодировке windows-1251:

```
try (
   BufferedReader buf = Files.newBufferedReader(
                        Paths.get("C:\\book\\test.txt"),
                        Charset.forName("cp1251"))
)
{
   String line = "";
   while ((line = buf.readLine()) != null) {
      System.out.println(line);
   }
}
```

20.3. Класс *PushbackReader*

Класс `PushbackReader` позволяет возвращать символы в поток ввода после прочтения. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.io.PushbackReader;
```

Иерархия наследования:

```
Object - Reader - FilterReader - PushbackReader
```

Класс `PushbackReader` реализует следующие интерфейсы:

```
Closeable, AutoCloseable, Readable
```

Создать объект позволяют следующие конструкторы класса `PushbackReader`:

```
PushbackReader(Reader in)
PushbackReader(Reader in, int size)
```

В первом параметре конструкторы принимают объект потока ввода. Второй формат позволяет дополнительно указать размер буфера. Если размер не указан, то вернуть в поток можно только один прочитанный до этого символ.

Класс `PushbackReader` наследует все методы из класса `Reader`. Кроме того, он содержит дополнительный метод `unread()`, который позволяет вернуть символы в поток. Эти символы будут доступны для следующей операции чтения. Форматы метода:

```
public void unread(int c)                          throws IOException
public void unread(char[] cbuf, int off, int len)
                                                   throws IOException
public void unread(char[] cbuf)                    throws IOException
```

Первый формат возвращает в поток один символ, второй формат — `len` символов, начиная с позиции `off`, а третий формат — все символы из массива `cbuf`. Если буфер заполнен, то генерируется исключение:

```
try (
   InputStream in = new FileInputStream("C:\\book\\test.txt");
   Reader reader = new InputStreamReader(in, "cp1251");
   BufferedReader buf = new BufferedReader(reader);
   PushbackReader push = new PushbackReader(buf, 10);
)
{
   char[] chars = new char[6];
   push.read(chars);
   for (int i = 0; i < chars.length; i++) {
      System.out.print( chars[i] + " " );
   } // с т р о к а
   push.unread(chars);  // Возвращаем символы в поток
   push.read(chars);
```

```
   for (int i = 0; i < chars.length; i++) {
      System.out.print( chars[i] + " " );
   } // с т р о к а
}
```

Обратите внимание на комбинирование классов потоков. В этом примере мы создали цепочку из четырех классов потоков: чтение из файла, чтение символов из потока в определенной кодировке, буферизация и возможность вернуть символы в поток. В таких цепочках может быть еще больше классов.

20.4. Классы *PrintWriter* и *PrintStream*

Классы `PrintWriter` и `PrintStream` добавляют методы `print()`, `println()` и `printf()`, которые мы очень часто использовали в примерах для вывода результата с помощью объекта `System.out`. Оба класса позволяют работать с символами — различие лишь в параметрах метода `write()` и типе возвращаемого значения в методах `append()`, `printf()` и `format()`: в классе `PrintStream` метод `write()` работает с байтами, а в классе `PrintWriter` — с символами.

20.4.1. Класс *PrintWriter*

Прежде чем использовать класс `PrintWriter`, необходимо импортировать его с помощью инструкции:

```
import java.io.PrintWriter;
```

Иерархия наследования:

```
Object - Writer - PrintWriter
```

Класс `PrintWriter` реализует следующие интерфейсы:

```
Closeable, Flushable, Appendable, AutoCloseable
```

Создать объект позволяют следующие конструкторы класса `PrintWriter`:

```
PrintWriter(String fileName, String csn)
     throws FileNotFoundException, UnsupportedEncodingException
PrintWriter(String fileName, Charset csn)
     throws IOException
PrintWriter(File file, String csn)
     throws FileNotFoundException, UnsupportedEncodingException
PrintWriter(File file, Charset csn)
     throws IOException
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoFlush)
PrintWriter(OutputStream out, boolean autoFlush, Charset charset)
PrintWriter(Writer out)
PrintWriter(Writer out, boolean autoFlush)
```

Первый, второй, третий и четвертый конструкторы в первом параметре принимают путь к файлу (в виде строки или объекта класса `File`). Если файл не существует, то он будет создан, а если существует — перезаписан. Во втором параметре указывается кодировка файла в виде строки или объекта класса `Charset`. Если параметр `csn` вообще не указан, то применяется кодировка, используемая в системе по умолчанию (всегда указывайте кодировку, если не хотите получить множество проблем). Второй и четвертый конструкторы доступны, начиная с Java 10:

```
try (
   PrintWriter pw = new PrintWriter("C:\\book\\test.txt", "cp1251");
)
{
   pw.println("строка1");
   pw.println("строка2");
}
```

Пятый, шестой и седьмой конструкторы принимают в первом параметре объект потока вывода `OutputStream`. Если в параметре `autoFlush` указано значение `true`, то вызов методов `println()`, `printf()` и `format()` будет приводить к автоматическому сбросу буфера. Обратите внимание: при использовании пятого и шестого конструкторов данные будут преобразованы в кодировку, используемую в системе по умолчанию. Если не хотите проблем, то лучше пятый и шестой конструкторы не использовать. Седьмой конструктор, доступный с Java 10, позволяет указать кодировку в виде объекта класса `Charset`:

```
try (
   OutputStream out = new FileOutputStream("C:\\book\\test.txt");
   PrintWriter pw = new PrintWriter(out, true,
                            Charset.forName("utf-8"));
)
{
   pw.println("строка1");
   pw.println("строка2");
}
```

Восьмой и девятый конструкторы принимают в первом параметре объект потока вывода `Writer`. Если в параметре `autoFlush` указано значение `true`, то вызов методов `println()`, `printf()` и `format()` будет приводить к автоматическому сбросу буфера. Пример записи в файл в кодировке windows-1251:

```
try (
   OutputStream out = new FileOutputStream("C:\\book\\test.txt");
   Writer writer = new OutputStreamWriter(out, "cp1251");
   PrintWriter pw = new PrintWriter(writer, true);
)
{
   pw.println("строка1");
   pw.println("строка2");
}
```

Если необходимо добавить строки в файл, а не перезаписать его, то во втором параметре конструктора класса `FileOutputStream` следует указать значение `true`:

```
try (
   OutputStream out =
       new FileOutputStream("C:\\book\\test.txt", true);
   Writer writer = new OutputStreamWriter(out, "cp1251");
   PrintWriter pw = new PrintWriter(writer, true);
)
{
   pw.println("строка3"); // Добавляем строки в конец файла
   pw.println("строка4");
}
```

Класс `PrintWriter` наследует все методы из класса `Writer`, реализует их и добавляет следующие методы:

- `print()` — отправляет данные в поток вывода. Формат метода:

  ```
  public void print(<Данные>)
  ```

  В параметре `<Данные>` можно указать данные типов `boolean`, `int`, `long`, `float`, `double`, `char`, `char[]`, `String` и `Object`. Элементарные типы будут преобразованы в строку, а не записаны в двоичном формате. При указании объекта будет вызван метод `toString()`:

  ```
  try (
     PrintWriter pw =
         new PrintWriter("C:\\book\\test.txt", "cp1251");
  )
  {
     pw.print(10);
     pw.print(' ');
     pw.print("строка");
  } // 10 строка
  ```

- `println()` — отправляет данные в поток вывода и завершает текущую строку разделителем, используемым в операционной системе (`System.getProperty("line.separator")`). Если параметр `autoFlush` в конструкторе имеет значение `true`, то буфер будет автоматически сброшен. Формат метода:

  ```
  public void println([<Данные>])
  ```

  В параметре `<Данные>` можно указать данные типов `boolean`, `int`, `long`, `float`, `double`, `char`, `char[]`, `String` и `Object`. Если параметр не указан, то просто завершает текущую строку:

  ```
  try (
     PrintWriter pw =
         new PrintWriter("C:\\book\\test.txt", "cp1251");
  )
  ```

```
{
   pw.println(10);
   pw.println(88.2);
   pw.println("строка");
}
/*
10
88.2
строка
*/
```

- `printf()` — предназначен для форматированного вывода данных. Форматы метода:

```
public PrintWriter printf(String format, Object... args)
public PrintWriter printf(Locale locale, String format,
                                         Object... args)
```

В параметре `format` указывается строка специального формата, внутри которой с помощью спецификаторов задаются правила форматирования (эти спецификаторы мы уже рассматривали в *разд. 6.13*). В параметре `args` через запятую указываются различные значения. Параметр `locale` позволяет задать локаль. Настройки локали для разных стран различаются. Например, в одной стране принято десятичный разделитель вещественных чисел выводить в виде точки, а в другой — в виде запятой. В первом формате метода используются настройки локали по умолчанию. Метод возвращает объект потока вывода. Если параметр `autoFlush` в конструкторе имеет значение `true`, то буфер будет автоматически сброшен:

```
try (
   PrintWriter pw =
       new PrintWriter("C:\\book\\test.txt", "cp1251");
)
{
   pw.printf("%.2f", 10.5125484).println();
   pw.printf(new Locale("en", "US"), "%.2f", 10.5125484);
   pw.println();
   pw.printf("%d %s", 10, "строка");
}
/*
10,51
10.51
10 строка
*/
```

- `format()` — предназначен для форматированного вывода данных. Метод аналогичен методу `printf()`. Если параметр `autoFlush` в конструкторе имеет значение `true`, то буфер будет автоматически сброшен. Форматы метода:

```
public PrintWriter format(String format, Object... args)
public PrintWriter format(Locale locale, String format,
                                         Object... args)
```

- `checkError()` — сбрасывает данные из буфера и возвращает значение `true`, если произошла ошибка, или `false`, если ошибка не возникла. Формат метода:

```
public boolean checkError()
```

20.4.2. Класс *PrintStream*

Прежде чем использовать класс `PrintStream`, необходимо импортировать его с помощью инструкции:

```
import java.io.PrintStream;
```

Иерархия наследования:

```
Object - OutputStream - FilterOutputStream - PrintStream
```

Класс `PrintStream` реализует следующие интерфейсы:

```
Closeable, Flushable, Appendable, AutoCloseable
```

Создать объект позволяют следующие конструкторы класса `PrintStream`:

```
PrintStream(String fileName[, String csn])
     throws FileNotFoundException, UnsupportedEncodingException
PrintStream(String fileName, Charset csn)
     throws IOException
PrintStream(File file[, String csn])
     throws FileNotFoundException, UnsupportedEncodingException
PrintStream(File file, Charset csn)
     throws IOException
PrintStream(OutputStream out)
PrintStream(OutputStream out, boolean autoFlush)
PrintStream(OutputStream out, boolean autoFlush, String csn)
                        throws UnsupportedEncodingException
PrintStream(OutputStream out, boolean autoFlush, Charset csn)
```

Первый, второй, третий и четвертый конструкторы в первом параметре принимают путь к файлу (в виде строки или объекта класса `File`). Если файл не существует, то он будет создан, а если существует — перезаписан. Во втором параметре указывается кодировка файла в виде строки или объекта класса `Charset`. Если параметр `csn` вообще не указан, то применяется кодировка, используемая в системе по умолчанию. Второй и четвертый конструкторы доступны, начиная с Java 10:

```
try (
   PrintStream pw = new PrintStream("C:\\book\\test.txt", "cp1251");
)
{
   pw.println("строка1");
   pw.println("строка2");
}
```

Пятый, шестой, седьмой и восьмой конструкторы в первом параметре принимают объект класса OutputStream. Если в параметре autoFlush указано значение true, то вызов методов println(), printf() и format() будет приводить к автоматическому сбросу буфера. В параметре csn указывается кодировка файла в виде строки или объекта класса Charset. Если параметр csn вообще не указан, то применяется кодировка, используемая в системе по умолчанию. Восьмой конструктор доступен, начиная с Java 10. Пример добавления записей в файл в кодировке windows-1251:

```
try (
   OutputStream out =
      new FileOutputStream("C:\\book\\test.txt", true);
   PrintStream pw = new PrintStream(out, true, "cp1251");
)
{
   pw.println("строка3");
   pw.println("строка4");
}
```

Класс PrintStream наследует и реализует все методы из класса OutputStream, а также реализует все форматы метода append() (возвращают объект класса PrintStream) из интерфейса Appendable. Таким образом, класс PrintStream содержит все методы из класса Writer, но форматы метода write() работают с байтами, а не с символами.

Класс PrintStream дополнительно содержит методы print(), println(), printf(), format() и checkError(), которые аналогичны одноименным методам из класса PrintWriter. Различие только в методах printf() и format() — они возвращают объект класса PrintStream.

20.4.3. Перенаправление стандартных потоков вывода

Объекты System.out (стандартный поток вывода) и System.err (поток вывода сообщений об ошибках) являются экземплярами класса PrintStream. Поэтому мы можем использовать все методы из этого класса:

```
System.out.print("строка");
System.out.println();
System.out.println("строка");
System.out.printf("%d", 10).println();
System.out.format("%.2f", 10.54565).println();
System.out.append('c').append(" строка").println();
System.out.write("string\n".getBytes());
System.out.flush();
System.out.println(System.out.checkError());
```

По умолчанию объекты System.out и System.err связаны с окном консоли. В некоторых случаях необходимо перенаправить их вывод в файлы. Чтобы перенаправить

потоки из командной строки, достаточно после названия файла с классом указать следующую команду:

```
1> путь_к_файлу_для_out 2> путь_к_файлу_для_err
```

Как можно видеть, поток `System.out` имеет дескриптор `1`, а поток `System.err` — дескриптор `2`:

```
C:\book>java MyClass 1>out.txt 2>err.txt
```

В режиме `>` файлы будут постоянно перезаписываться, а чтобы записи добавлялись в конец файла, необходимо использовать режим `>>`:

```
C:\book>java MyClass 1>>out.txt 2>>err.txt
```

Для перенаправления потоков из программы предназначены методы `setOut()` и `setErr()` из класса `System`. Форматы методов:

```
public static void setOut(PrintStream out)
public static void setErr(PrintStream err)
```

Пример:

```
OutputStream out =
      new FileOutputStream("C:\\book\\out.txt", true);
OutputStream err =
      new FileOutputStream("C:\\book\\err.txt", true);
PrintStream ps_out = new PrintStream(out, true, "utf-8");
PrintStream ps_err = new PrintStream(err, true, "utf-8");
System.setOut(ps_out);
System.setErr(ps_err);
System.out.println("Обычное сообщение");
System.err.println("Сообщение об ошибке");
```

Если такое перенаправление лишь временное, то вначале следует сохранить ссылки на стандартные потоки, чтобы потом их можно было восстановить:

```
PrintStream ps_out_tmp = System.out;
PrintStream ps_err_tmp = System.err;
//...
System.setOut(ps_out_tmp);
System.setErr(ps_err_tmp);
```

20.5. Класс *Scanner*

Для чтения данных из потока ввода можно воспользоваться классом `Scanner`. Прежде чем использовать этот класс, необходимо импортировать его с помощью инструкции:

```
import java.util.Scanner;
```

Класс `Scanner` реализует следующие интерфейсы:

```
Closeable, AutoCloseable, Iterator<String>
```

Создать объект позволяют следующие основные конструкторы класса `Scanner`:

```
Scanner(File source[, String charsetName])
                          throws FileNotFoundException
Scanner(File source, Charset charset)
                          throws IOException
Scanner(Path source[, String charsetName])
                          throws IOException
Scanner(Path source, Charset charset)
                          throws IOException
Scanner(InputStream source[, String charsetName])
Scanner(InputStream source, Charset charset)
Scanner(Readable source)
Scanner(String source)
Scanner(ReadableByteChannel source[, String charsetName])
Scanner(ReadableByteChannel source, Charset charset)
```

Первый и второй конструкторы в первом параметре позволяют указать путь в виде объекта класса `File`, а во втором параметре — название кодировки в виде строки или объекта класса `Charset` (второй конструктор доступен, начиная с Java 10):

```
Scanner in = new Scanner(new File("C:\\book\\test.txt"), "cp1251");
```

Третий и четвертый конструкторы в первом параметре позволяют указать путь в виде объекта класса `Path`, а во втором параметре — название кодировки в виде строки или объекта класса `Charset` (четвертый конструктор доступен, начиная с Java 10):

```
Scanner in = new Scanner(Paths.get("C:\\book\\test.txt"), "cp1251");
```

Пятый и шестой конструкторы в первом параметре позволяют указать объект потока ввода `InputStream`, а во втором параметре — название кодировки в виде строки или объекта класса `Charset` (шестой конструктор доступен, начиная с Java 10):

```
InputStream fin = new FileInputStream("C:\\book\\test.txt");
Scanner in = new Scanner(fin, "cp1251");
```

Седьмой конструктор принимает в качестве параметра объект, реализующий интерфейс `Readable`:

```
InputStream fin = new FileInputStream("C:\\book\\test.txt");
Reader reader = new InputStreamReader(fin, "cp1251");
Scanner in = new Scanner(reader);
```

Восьмой конструктор создает объект на основе строки:

```
Scanner in = new Scanner("строка1\nстрока2\nстрока3");
```

Класс `Scanner` содержит следующие основные методы:

- `hasNextBoolean()`, `hasNextByte()`, `hasNextShort()`, `hasNextInt()`, `hasNextLong()`, `hasNextFloat()` и `hasNextDouble()` — возвращают значение `true`, если можно прочитать данные соответствующего элементарного типа, и `false` — в противном случае. Форматы методов:

```
public boolean hasNextBoolean()
public boolean hasNextByte()
public boolean hasNextByte(int radix)
public boolean hasNextShort()
public boolean hasNextShort(int radix)
public boolean hasNextInt()
public boolean hasNextInt(int radix)
public boolean hasNextLong()
public boolean hasNextLong(int radix)
public boolean hasNextFloat()
public boolean hasNextDouble()
```

Параметр `radix` задает систему счисления (`2`, `8`, `10`, `16`);

- `nextBoolean()`, `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()` и `nextDouble()` — получают данные элементарных типов. Форматы методов:

```
public boolean nextBoolean()
public byte nextByte()
public byte nextByte(int radix)
public short nextShort()
public short nextShort(int radix)
public int nextInt()
public int nextInt(int radix)
public long nextLong()
public long nextLong(int radix)
public float nextFloat()
public double nextDouble()
```

Параметр `radix` задает систему счисления (`2`, `8`, `10`, `16`). Если параметр не указан, то используется десятичная система:

```
Scanner in =
    new Scanner("false\n1001011\n077\n33\nFF\n2,5\n12,7");
System.out.println(in.nextBoolean()); // false
System.out.println(in.nextByte(2));   // 75
System.out.println(in.nextShort(8));  // 63
System.out.println(in.nextInt(10));   // 33
System.out.println(in.nextLong(16));  // 255
System.out.println(in.nextFloat());   // 2.5
System.out.println(in.nextDouble());  // 12.7
in.close();
```

- `hasNextLine()` — возвращает значение `true`, если можно прочитать строку, и `false` — в противном случае. Формат метода:

```
public boolean hasNextLine()
```

- `nextLine()` — позволяет читать построчно. Формат метода:

```
public String nextLine()
```

Пример:

```
Scanner in = new Scanner("строка 1\nстрока 2\nстрока 3");
System.out.println(in.nextLine()); // строка 1
System.out.println(in.nextLine()); // строка 2
System.out.println(in.nextLine()); // строка 3
in.close();
```

- `hasNext()` — возвращает значение `true`, если можно прочитать данные, и `false` — в противном случае. Форматы метода:

```
public boolean hasNext()
public boolean hasNext(String pattern)
public boolean hasNext(Pattern pattern)
```

Первый формат проверяет наличие строки, а второй и третий форматы — соответствие строки шаблону регулярного выражения;

- `next()` — возвращает следующий фрагмент. Форматы метода:

```
public String next()
public String next(String pattern)
public String next(Pattern pattern)
```

Первый формат возвращает следующий фрагмент, а второй и третий форматы — фрагмент, соответствующий шаблону регулярного выражения. Получим только слова, которые заканчиваются числами:

```
Scanner in = new Scanner("строка1 строка строка3");
Pattern p = Pattern.compile(".*[0-9]+");
while (in.hasNext()) {
   if (in.hasNext(p)) System.out.println(in.next(p));
   else in.next();
}
in.close();
```

- `useDelimiter()` — задает шаблон регулярного выражения, который будет использоваться для разделения фрагментов. Форматы метода:

```
public Scanner useDelimiter(String pattern)
public Scanner useDelimiter(Pattern pattern)
```

Пример разделения подстрок по любым пробельным символам:

```
Scanner in = new Scanner("1 2 3 4 5 \n6  7");
in.useDelimiter(Pattern.compile("\\s+"));
while (in.hasNext()) {
   System.out.print(in.next() + ";");
} // 1;2;3;4;5;6;7;
in.close();
```

- `delimiter()` — возвращает шаблон регулярного выражения, который используется для разделения фрагментов. Формат метода:

```
public Pattern delimiter()
```

Пример:

```
Scanner in = new Scanner("1 2 3 4 5 \n6  7");
System.out.println(in.delimiter());  // \p{javaWhitespace}+
in.useDelimiter(Pattern.compile("\\s+"));
System.out.println(in.delimiter());  // \s+
in.close();
```

- `useRadix()` — задает систему счисления для целых чисел. Формат метода:

```
public Scanner useRadix(int radix)
```

- `radix()` — возвращает систему счисления для целых чисел. Формат метода:

```
public int radix()
```

Пример:

```
Scanner in = new Scanner("1 2 3 4 5 \n6  7");
System.out.println(in.radix());  // 10
in.useRadix(8);
System.out.println(in.radix());  // 8
in.close();
```

- `useLocale()` — задает локаль. Формат метода:

```
public Scanner useLocale(Locale locale)
```

- `locale()` — возвращает используемую локаль. Формат метода:

```
public Locale locale()
```

Пример:

```
Scanner in = new Scanner("1 2 3 4 5 \n6  7");
System.out.println(in.locale());  // ru_RU
in.useLocale(new Locale("en", "US"));
System.out.println(in.locale());  // en_US
in.close();
```

- `reset()` — сбрасывает настройки разделителя, системы счисления и локали к настройкам по умолчанию. Формат метода:

```
public Scanner reset()
```

- `skip()` — пропускает фрагмент, соответствующий шаблону. Разделители фрагментов при этом игнорируются. Если фрагмент не найден, то генерируется исключение. Форматы метода:

```
public Scanner skip(String pattern)
public Scanner skip(Pattern pattern)
```

- `findInLine()` — выполняет поиск в текущей строке с помощью регулярного выражения, игнорируя разделитель фрагментов. Если совпадение найдено, то возвращает фрагмент, соответствующий шаблону, в противном случае — значение `null`. Форматы метода:

```
public String findInLine(String pattern)
public String findInLine(Pattern pattern)
```

- `findWithinHorizon()` — выполняет поиск во фрагменте, ограниченном количеством символов `horizon`, с помощью регулярного выражения, игнорируя разделитель фрагментов. Если совпадение найдено, то возвращает фрагмент, соответствующий шаблону, в противном случае — значение `null`. Если в параметре `horizon` указано значение `0`, то поиск будет без ограничений. Форматы метода:

```
public String findWithinHorizon(String pattern, int horizon)
public String findWithinHorizon(Pattern pattern, int horizon)
```

- `match()` — возвращает объект `MatchResult` с результатом предыдущего поиска. Формат метода:

```
public MatchResult match()
```

Пример получения текста между тегами и :

```
Scanner in = new Scanner(
   "text<b>Text 0</b>j<b>Text1</b> text\n<b>Text2\n5</b>");
Pattern p = Pattern.compile("<b>(.*?)</b>",
            Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE);
Scanner sc = null;
MatchResult m = null;
while (in.hasNextLine()) {
   sc = new Scanner(in.nextLine());
   while (sc.findInLine(p) != null) {
      m = sc.match();
      for (int i = 0; i < m.groupCount(); i++) {
         System.out.print(m.group(i + 1) + " ");
      }
   }
   sc.close();
}
in.close();
```

Результат:

```
Text 0 Text1
```

Как видно из результата, мы не получили текст между последними тегами, т. к. он расположен на нескольких строках. Метод `findWithinHorizon()` позволяет исправить эту проблему:

```
Scanner in = new Scanner(
   "text<b>Text 0</b>j<b>Text1</b> text\n<b>Text2\n5</b>");
Pattern p = Pattern.compile("<b>(.*?)</b>",
            Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE |
            Pattern.DOTALL);
MatchResult m = null;
```

```
while (in.findWithinHorizon(p, 0) != null) {
   m = in.match();
   for (int i = 0; i < m.groupCount(); i++) {
      System.out.print(m.group(i + 1) + " ");
   }
}
in.close();
```

Результат:

```
Text 0 Text1 Text2
5
```

- `findAll()` — возвращает поток с результатами поиска по шаблону регулярного выражения. Обратите внимание: поиск будет выполнен только после вызова терминального метода с использованием текущего состояния потока, а не сразу. Метод доступен, начиная с Java 9. Форматы метода:

```
import java.util.regex.MatchResult;
import java.util.stream.Stream;
public Stream<MatchResult> findAll(String pattern)
public Stream<MatchResult> findAll(Pattern pattern)
```

Пример получения всех чисел из потока:

```
Scanner in = new Scanner("1 2 3 str 14 5 \n6 str 77");
in.findAll("[0-9]+").forEachOrdered( m -> {
   System.out.print(m.group() + " ");
}); // 1 2 3 14 5 6 77
in.close();
```

- `tokens()` — возвращает поток с фрагментами между разделителем (задается с помощью метода `useDelimiter()`). Обратите внимание: операция будет выполнена только после вызова терминального метода с использованием текущего состояния потока, а не сразу. Метод доступен, начиная с Java 9. Формат метода:

```
import java.util.stream.Stream;
public Stream<String> tokens()
```

Пример:

```
Scanner in = new Scanner("1\r2\f3,4 5\n6\t7");
in.useDelimiter(Pattern.compile("[\\s,\\.]+"));
in.tokens().forEachOrdered( s -> {
   System.out.print(s + ";");
}); // 1;2;3;4;5;6;7;
in.close();
```

- `close()` — закрывает поток. Формат метода:

```
public void close()
```

В листинге 3.8 мы выполняли суммирование неопределенного количества целых чисел, введенных пользователем в окне консоли, но не выполняли никакой обра-

ботки ошибок. Если пользователь введет строку вместо числа, то возникнет исключение, и программа прекратит выполнение. Теперь же мы знаем о классе `Scanner` все необходимые сведения и можем обработать эти ошибки (листинг 20.3).

Листинг 20.3. Суммирование неопределенного количества целых чисел

```
package com.example.app;

import java.util.Scanner;

public class MyClass {
   public static void main(String[] args) {
      int x = 0, result = 0;
      Scanner in = new Scanner(System.in);
      System.out.println(
         "Введите число 0 для получения результата\n");
      while (true) {
         System.out.print("Введите число: ");
         if (in.hasNextInt()) {
            x = in.nextInt();
            if (x == 0) break;
            result += x;
            // Очищаем буфер
            if (in.hasNextLine()) in.nextLine();
         }
         else {
            if (in.hasNextLine()) {
               in.nextLine(); // Пропускаем ввод
               System.out.println("Необходимо ввести целое число!");
            }
            else break;
         }
      }
      System.out.println("Сумма чисел равна: " + result);
   }
}
```

Обратите внимание на инструкцию после комментария `Очищаем буфер`. Когда мы получаем число, в буфере остается фрагмент до символа перевода строки. Это может быть еще число, строка или вообще пустая строка. При вводе строки вместо числа мы пытаемся считать все содержимое буфера, чтобы пропустить неправильный ввод, и в этот момент мы вначале получим содержимое буфера и лишь при повторном пропуске — неправильный фрагмент. Рассмотрим это на примере.

Предположим, что инструкции очистки буфера нет. Пользователь вводит следующее значение:

```
10 20\n
```

После получения числа 10 в буфере останется следующий фрагмент:

```
20\n
```

Следующая операция ввода получит число 20, а не последующий ввод пользователя:

```
\n
строка\n
```

Заметьте, что в буфере все еще остается символ перевода строки. При следующем считывании не будет найдено число, и мы попытаемся пропустить ввод, вызвав метод nextLine(). Метод вначале найдет первый символ перевода строки и вернет пустую строку. В итоге в буфере останется следующий фрагмент:

```
строка\n
```

Опять число не найдено, и снова мы пропускаем ввод, вызвав метод nextLine(). В итоге буфер станет пустым.

Чтобы избежать получения данных из буфера и исключить подобные ситуации, после ввода числа мы сразу очищаем содержимое буфера до конца строки (инструкция после комментария Очищаем буфер). Теперь последовательность ввода будет следующей:

```
10 20\n
```

После получения числа 10 в буфере ничего не останется, и мы получим последующий ввод пользователя, а не число 20:

```
строка\n
```

В этом вводе не будет найдено число. После пропуска ввода буфер опять станет пустым.

20.6. Класс *Console*

Класс Console позволяет взаимодействовать с окном консоли: читать и выводить данные. Класс особенно полезен возможностью считывать пароли без отображения символов в консоли. Прежде чем использовать этот класс, необходимо выполнить его импорт с помощью инструкции:

```
import java.io.Console;
```

Создать объект позволяет метод console() из класса System. Формат метода:

```
public static Console console()
```

Если приложение не связано с консолью (при запуске из редактора Eclipse именно так и будет), то метод вернет значение null:

```
Console cons = System.console();
if (cons != null) {
   cons.printf("%s", "Текст, выводимый на консоль");
}
```

```
else {
   System.out.println("Консоль не подключена");
}
```

Класс `Console` содержит следующие методы:

- `printf()` и `format()` — предназначены для форматированного вывода данных. Форматы методов:

```
public Console printf(String format, Object... args)
public Console format(String format, Object... args)
```

  В параметре `format` указывается строка специального формата, внутри которой с помощью спецификаторов задаются правила форматирования (эти спецификаторы мы уже рассматривали в *разд. 6.13*). В параметре `args` через запятую указываются различные значения:

```
Console cons = System.console();
if (cons != null) {
   cons.printf("%d %s%n", 20, "Текст");
   cons.format("%.2f%n", 12.45464);
}
```

- `writer()` — возвращает объект класса `PrintWriter`, с помощью которого можно выводить данные на консоль. Формат метода:

```
public PrintWriter writer()
```

  Пример:

```
cons.writer().println(20);
```

- `flush()` — сбрасывает данные из буфера. Формат метода:

```
public void flush()
```

- `readLine()` — читает строку из консоли и возвращает ее. Форматы метода:

```
public String readLine()
public String readLine(String format, Object... args)
```

  Параметр `format` позволяет вывести форматированную подсказку пользователю:

```
Console cons = System.console();
if (cons != null) {
   String line = cons.readLine("%s", "Введите данные: ");
   cons.writer().println(line);
}
```

- `reader()` — возвращает объект класса `Reader`, с помощью которого можно вводить данные с консоли. Формат метода:

```
public Reader reader()
```

  Пример:

```
Console cons = System.console();
if (cons != null) {
   cons.printf("%s", "Введите данные: ");
```

```
    Scanner in = new Scanner(cons.reader());
    String line = in.nextLine();
    cons.writer().println(line);
  }
```

- `readPassword()` — позволяет ввести пароль без отображения символов в окне консоли. В целях безопасности пароль возвращается в виде символьного массива. После прочтения пароля необходимо перезаписать массив. Параметр `format` позволяет вывести форматированную подсказку пользователю. Форматы метода:

```
public char[] readPassword()
public char[] readPassword(String format, Object... args)
```

В качестве примера создадим программу проверки правильности ввода пароля (листинг 20.4).

Листинг 20.4. Проверка правильности ввода пароля

```
import java.io.Console;
import java.security.MessageDigest;
import java.util.Arrays;

public class MyClass {
   public static void main(String[] args) throws Exception {
      Console cons = System.console();
      if (cons != null) {
         char[] passwd = null;
         passwd = cons.readPassword("%s", "Введите пароль: ");
         char[] vp = {'e', '1', '0', 'a', 'd', 'c', '3', '9', '4',
                      '9', 'b', 'a', '5', '9', 'a', 'b', 'b', 'e',
                      '5', '6', 'e', '0', '5', '7', 'f', '2', '0',
                      'f', '8', '8', '3', 'e'};
         if ((passwd != null) &&
                     String.valueOf(vp).equals(md5(passwd))) {
            cons.printf("Пароль введен правильно%n");
            Arrays.fill(passwd, ' '); // Очищаем массив
         }
         else {
            cons.printf("Пароль введен неправильно%n");
            System.exit(1); // Останавливаем программу
         }
         // Пользователь авторизован
         cons.printf("Конфетка :)%n");
      }
      else {
         System.out.println("Консоль не подключена");
      }
   }
```

```
   public static String md5(char[] ch) {
      if (ch == null || ch.length == 0) return null;
      try {
         MessageDigest md5 = MessageDigest.getInstance("MD5");
         md5.update(String.valueOf(ch).getBytes("UTF-8"));
         byte[] arr = md5.digest();
         StringBuilder sb = new StringBuilder();
         for (int i = 0; i < arr.length; i++) {
            String s = Integer.toHexString(0xff & arr[i]);
            s = (s.length() == 1) ? "0" + s : s;
            sb.append(s);
         }
         return sb.toString();
      }
      catch (Exception e) {
         return null;
      }
   }
}
```

При работе с паролями не следует хранить их в строках. Лучше использовать массивы и в них хранить пароль в зашифрованном виде (массив `vp` в зашифрованном виде хранит пароль "`123456`"). В нашем примере мы используем алгоритм шифрования `MD5`. Пароль, закодированный этим алгоритмом, нельзя расшифровать, можно лишь выполнить шифрование введенного пароля и сравнить хеш-коды паролей. Для шифрования паролей с помощью алгоритма `MD5` используется класс `MessageDigest`. Конструктор класса принимает в качестве параметра название алгоритма шифрования в виде строки. Для добавления данных предназначен метод `update()`, а для получения зашифрованных данных — метод `digest()`, который возвращает массив байтов.

ГЛАВА 21

Работа с потоками данных: Stream API

Начиная с Java 8, в состав библиотеки языка входит пакет `java.util.stream`, который позволяет работать с *потоками данных*. Причем обработка данных может быть произведена либо последовательно (по умолчанию), либо в параллельных потоках, что позволяет увеличить скорость обработки. Схема работы с потоками данных выглядит следующим образом:

```
Источник -> Операция 1 -> ... -> Операция N -> Терминальная операция
```

В качестве источника данных могут выступать списки, множества и другие элементы коллекции, а также массивы, файлы и др. Над данными производятся различные *промежуточные операции* — например, фильтрация данных. Любая промежуточная операция возвращает поток, с которым можно выполнять другие промежуточные операции. Промежуточные операции обычно заканчиваются *терминальной операцией*. При выполнении терминальной операции работа с потоком прекращается. Следует учитывать, что промежуточные операции являются «ленивыми», т. е. все промежуточные операции выполняются только при выполнении терминальной операции.

Прежде чем использовать потоки данных, необходимо выполнить импорт пакета `java.util.stream` с помощью следующей инструкции:

```
import java.util.stream.*;
```

21.1. Создание потока данных

Как уже говорилось, в качестве источника данных могут выступать элементы коллекции, а также массивы, файлы и др. Давайте рассмотрим способы создания потока данных подробно.

21.1.1. Создание потока из коллекции

Для создания потока данных на основе коллекции предназначены следующие методы из интерфейса `Collection<E>` (напомню, что этот интерфейс реализуют списки, очереди и множества):

- `stream()` — создает последовательный поток. Формат метода:

```
default Stream<E> stream()
```

Выведем все элементы списка, значение которых меньше `4`:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5);
Stream<Integer> stream = arr.stream();
stream.filter( (x) -> x < 4 ).forEachOrdered(
       (x) -> System.out.print(x + " ") ); // 1 2 3
```

В этом примере метод `filter()` является промежуточной операцией, а метод `forEachOrdered()` — терминальной операцией. Как видно из примера, при обработке данных применяются лямбда-выражения. Вместо лямбда-выражений можно использовать ссылки на методы или объекты, классы которых реализуют требуемый функциональный интерфейс;

- `parallelStream()` — создает параллельный поток. Формат метода:

```
default Stream<E> parallelStream()
```

Параллельный поток может обрабатывать данные в нескольких потоках, что позволяет увеличить скорость обработки, если процессор является многоядерным. Получим размер пула потоков:

```
// import java.util.concurrent.ForkJoinPool;
ForkJoinPool pool = ForkJoinPool.commonPool();
System.out.println(pool.getParallelism()); // 3
```

Последовательный поток можно преобразовать в параллельный, вызвав метод `parallel()`:

```
Stream<Integer> stream = arr.stream().parallel();
```

При использовании параллельных потоков возможны конфликты между несколькими потоками при доступе к одному объекту, впрочем, как и в любых многопоточных приложениях. Используя упорядоченные и сортированные данные, следует учитывать, что для поддержания порядка следования элементов требуются дополнительные действия, на которые тратится время. Чтобы получить максимальный выигрыш от параллельной обработки, следует сделать данные неупорядоченными с помощью метода `unordered()`.

Давайте посмотрим, как данные распределяются по потокам, на примере умножения каждого элемента списка на `2` внутри метода `map()`:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3, 4, 5);
Stream<Integer> stream = arr.parallelStream();
List<Integer> arr2 = stream.unordered().map( (x) -> {
   System.out.println(
      Thread.currentThread().getName() + " x = " + x);
```

```
    return x * 2;
}).collect(Collectors.toList());
System.out.println(arr2.toString()); // [2, 4, 6, 8, 10]
```

Результат будет выглядеть примерно следующим образом:

```
ForkJoinPool.commonPool-worker-1 x = 2
ForkJoinPool.commonPool-worker-2 x = 5
ForkJoinPool.commonPool-worker-1 x = 4
main x = 3
ForkJoinPool.commonPool-worker-3 x = 1
[2, 4, 6, 8, 10]
```

Как видно из результата, каждому потоку достались отдельные данные. Если будет использован метод `stream()` вместо `parallelStream()`, то результат окажется таким:

```
main x = 1
main x = 2
main x = 3
main x = 4
main x = 5
[2, 4, 6, 8, 10]
```

В этом случае все данные обрабатывались последовательно внутри основного потока.

21.1.2. Создание потока из массива или списка значений

Для создания потока из массива предназначен статический метод `stream()` из класса `Arrays`. Форматы метода:

```
public static IntStream stream(int[] array)
public static IntStream stream(int[] array,int start, int end)
public static LongStream stream(long[] array)
public static LongStream stream(long[] array, int start, int end)
public static DoubleStream stream(double[] array)
public static DoubleStream stream(double[] array, int start, int end)
public static <T> Stream<T> stream(T[] array)
public static <T> Stream<T> stream(T[] array, int start, int end)
```

Параметр `start` задает начальный индекс, а параметр `end` — конечный индекс (элемент с этим индексом не попадет в поток). Если параметры не указаны, то будут использованы все элементы массива. Интерфейсы `IntStream`, `LongStream` и `DoubleStream` являются потоками данных соответствующих элементарных типов, а интерфейс `Stream<T>` — потоком данных объектных типов. Пример создания потока на основе целочисленного массива:

```
int[] arr = {1, 2, 3, 4, 5};
IntStream stream = Arrays.stream(arr).parallel();
int[] arr2 = stream.unordered().map( (x) -> x * 2).toArray();
```

```
System.out.println(Arrays.toString(arr));  // [1, 2, 3, 4, 5]
System.out.println(Arrays.toString(arr2)); // [2, 4, 6, 8, 10]
```

Обратите внимание на то, что потоки не изменяют данные источника.

Создать поток на основе массива, списка значений или отдельных значений позволяет также статический метод `of()` из интерфейсов потоков. Форматы метода:

```
public static IntStream IntStream.of(int t)
public static IntStream IntStream.of(int... values)
public static LongStream LongStream.of(long t)
public static LongStream LongStream.of(long... values)
public static DoubleStream DoubleStream.of(double t)
public static DoubleStream DoubleStream.of(double... values)
public static <T> Stream<T> Stream.of(T t)
public static <T> Stream<T> Stream.of(T... values)
```

Пример вывода только четных чисел:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5).parallel();
stream.unordered()
      .filter( (x) -> (x % 2) == 0 )
      .forEachOrdered(x -> System.out.print(x + " "));
// 2 4
```

Для создания потока на основе отдельных значений можно воспользоваться статическим методом `builder()` из интерфейсов потоков. Форматы метода:

```
public static IntStream.Builder builder()
public static LongStream.Builder builder()
public static DoubleStream.Builder builder()
public static <T> Stream.Builder<T> builder()
```

После создания вспомогательного объекта следует добавить элементы с помощью метода `add()`, а в конце вызвать метод `build()`, который возвратит поток:

```
IntStream stream = IntStream.builder().add(1).add(2).build();
stream.forEachOrdered(x -> System.out.print(x + " "));
// 1 2
Stream.builder().add("1").add("2").build()
   .forEachOrdered(x -> System.out.print(x + " "));
// 1 2
```

Статический метод `ofNullable()` создает поток из заданного значения. Если вместо значения указано `null`, то возвращает пустой поток. Метод доступен, начиная с Java 9. Формат метода:

```
public static <T> Stream<T> ofNullable(T t)
```

Пример:

```
Stream<Integer> stream = Stream.ofNullable(5);
stream.forEachOrdered(x -> System.out.println(x)); // 5
stream = Stream.<Integer>ofNullable(null);
System.out.println(stream.count());                // 0
```

21.1.3. Создание потока из строки

Создать поток символов из строки или из объекта, класс которого реализует интерфейс `CharSequence`, позволяют методы `chars()` и `codePoints()` из интерфейса `CharSequence`. Форматы методов:

```
default IntStream chars()
default IntStream codePoints()
```

Пример:

```
IntStream stream = "строка".chars();
stream.forEachOrdered( x -> System.out.print( (char)x ) );
// строка
StringBuilder sb = new StringBuilder();
sb.append("строка");
IntStream stream2 = sb.chars();
stream2.forEachOrdered( x -> System.out.print( (char)x ) );
// строка
```

Метод `splitAsStream()` из класса `Pattern` разбивает строку на подстроки по указанному разделителю, представленному в виде регулярного выражения, и возвращает поток. Формат метода:

```
import java.util.regex.Pattern;
public Stream<String> splitAsStream(CharSequence input)
```

Пример:

```
Pattern p = Pattern.compile("\\s+");
Stream<String> stream = p.splitAsStream("1 2 3");
stream.forEachOrdered( s -> System.out.println( s ));
/*
1
2
3
 */
```

Начиная с Java 9, в классе `Matcher` доступен метод `results()`, который позволяет получить поток данных с результатами поиска по шаблону регулярного выражения. Формат метода:

```
public Stream<MatchResult> results()
```

Пример:

```
Pattern p = Pattern.compile("[0-9]+");
Matcher m = p.matcher("10 20 30");
m.results().forEachOrdered(mr -> System.out.println(mr.group()));
```

Результат:

```
10
20
30
```

21.1.4. Создание потока из файла или каталога

Для создания потока строк из файла предназначен статический метод `lines()` из класса `Files`. Форматы метода:

```
import java.nio.charset.Charset;
import java.nio.file.*;
public static Stream<String> lines(Path path) throws IOException
public static Stream<String> lines(Path path, Charset cs)
                                                 throws IOException
```

В параметре `cs` можно указать кодировку данных, а если кодировка не указана, то используется кодировка UTF-8. Пример построчного чтения всего файла в кодировке windows-1251:

```
Path p = Paths.get("C:\\book\\cp1251.txt");
try (Stream<String> stream =
        Files.lines(p, Charset.forName("cp1251")))
{
   stream.forEachOrdered( (s) -> System.out.println(s) );
}
catch (IOException e) {
   e.printStackTrace();
}
```

Кроме того, можно воспользоваться методом `lines()` из класса `BufferedReader`, который позволяет читать файл построчно. Формат метода:

```
public Stream<String> lines()
```

Пример:

```
try (
   InputStream in = new FileInputStream("C:\\book\\test.txt");
   Reader reader = new InputStreamReader(in, "cp1251");
   BufferedReader buf = new BufferedReader(reader);
)
{
   Stream<String> stream = buf.lines();
   stream.forEachOrdered( (s) -> System.out.println(s) );
}
```

Для чтения содержимого каталога используется статический метод `list()` из класса `Files`. Формат метода:

```
public static Stream<Path> list(Path dir) throws IOException
```

Пример вывода всех элементов каталога и только файлов с расширением java:

```
Path p = Paths.get("C:\\book\\");
try (Stream<Path> stream = Files.list(p)) {
   stream.forEachOrdered(
      (obj) -> System.out.println(obj.toString())
   );
}
```

```
try (Stream<Path> stream = Files.list(p)) {
   stream.filter(
      (path) -> path.toString().endsWith(".java")
   ).forEachOrdered(
      (obj) -> System.out.println(obj.toString())
   );
}
```

С помощью метода `walk()` из класса `Files` можно создать поток для обхода всего дерева каталогов (пример использования метода `walk()` показан в листинге 18.5). Форматы метода:

```
public static Stream<Path> walk(Path start,
              FileVisitOption... options) throws IOException
public static Stream<Path> walk(Path start, int maxDepth,
              FileVisitOption... options) throws IOException
```

Для поиска элемента внутри дерева каталогов можно воспользоваться методом `find()`. Формат метода:

```
public static Stream<Path> find(Path start, int maxDepth,
              BiPredicate<Path, BasicFileAttributes> matcher,
              FileVisitOption... options) throws IOException
```

Пример поиска всех текстовых файлов:

```
Path p = Paths.get("C:\\book\\");
try (Stream<Path> stream = Files.find(p, 4, (f, attrs) -> {
         return f.toString().endsWith(".txt");
      }))
{
   stream.forEachOrdered(
      (path) -> System.out.println(path.toString())
   );
}
```

Примечание

В Java 9 в класс `Scanner` были добавлены методы `findAll()` и `tokens()`. Описание этих методов см. в *разд. 20.5*.

21.1.5. Создание потока с помощью итератора или генератора

Сгенерировать бесконечное количество элементов позволяют методы `iterate()` и `generate()` из интерфейсов потоков. Форматы метода `iterate()`:

```
public static IntStream iterate(int seed, IntUnaryOperator f)
public static IntStream iterate(int seed, IntPredicate hasNext,
                                IntUnaryOperator f)
```

```
public static LongStream iterate(long seed, LongUnaryOperator f)
public static LongStream iterate(long seed, LongPredicate hasNext,
                                 LongUnaryOperator f)
public static DoubleStream iterate(double seed, DoubleUnaryOperator f)
public static DoubleStream iterate(double seed, DoublePredicate hasNext,
                                   DoubleUnaryOperator f)
public static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
public static <T> Stream<T> iterate(T seed,
              Predicate<? super T> hasNext, UnaryOperator<T> f)
```

Параметр `seed` задает начальное значение, параметр `hasNext` (параметр доступен, начиная с Java 9) — условие окончания, а параметр `f` — приращение. Сгенерируем четные числа от `2` до `200`:

```
IntStream stream = IntStream.iterate(2, n -> n + 2).limit(100);
stream.forEachOrdered(x -> System.out.println(x));
```

В этом примере мы воспользовались методом `limit()`, который ограничивает количество элементов потока.

Пример ограничения с помощью параметра `hasNext`:

```
IntStream stream = IntStream.iterate(2, n -> n <= 200, n -> n + 2);
stream.forEachOrdered(x -> System.out.println(x));
```

Форматы метода `generate()`:

```
public static IntStream generate(IntSupplier s)
public static LongStream generate(LongSupplier s)
public static DoubleStream generate(DoubleSupplier s)
public static <T> Stream<T> generate(Supplier<? extends T> s)
```

Сгенерируем десять случайных чисел от `0` до `99`:

```
IntStream.generate(() -> (int)(Math.random() * 100))
         .limit(10)
         .forEachOrdered(x -> System.out.println(x));
```

При использовании целочисленных потоков можно сгенерировать диапазон чисел с помощью методов `range()` и `rangeClosed()`. Форматы методов:

```
public static IntStream range(int start, int end)
public static IntStream rangeClosed(int start, int end)
public static LongStream range(long start, long end)
public static LongStream rangeClosed(long start, long end)
```

Метод `range()` генерирует числа от `start` до `end` (не включая это число), а метод `rangeClosed()` — от `start` до `end` (включая это число). Шаг приращения `1`:

```
IntStream stream = IntStream.range(0, 3);
stream.forEachOrdered( n -> System.out.print(n + " ") );
// 0 1 2
stream = IntStream.rangeClosed(0, 3);
stream.forEachOrdered( n -> System.out.print(n + " ") );
// 0 1 2 3
```

21.1.6. Создание потока случайных чисел

Для создания потока случайных чисел используются следующие методы из класса `Random`:

- `ints()` — создает поток случайных целых чисел (тип `int`). Форматы метода:

```
public IntStream ints()
public IntStream ints(
       int randomNumberOrigin, int randomNumberBound)
public IntStream ints(long streamSize)
public IntStream ints(long streamSize,
       int randomNumberOrigin, int randomNumberBound)
```

Параметр `streamSize` задает количество чисел в потоке, а если размер не задан, то поток будет бесконечным. Если параметры `randomNumberOrigin` и `randomNumberBound` не заданы, то генерируются случайные значения в пределах значений для типа `int`. Пример генерации бесконечного потока целых чисел (значения типа `int`):

```
Random r = new Random();
IntStream stream = r.ints();
stream.limit(10).forEachOrdered(x -> System.out.println(x));
```

Пример генерации потока из десяти чисел от `0` до `100` (не включая это число):

```
Random r = new Random();
IntStream stream = r.ints(10L, 0, 100);
stream.forEachOrdered(x -> System.out.println(x));
```

- `longs()` — создает поток случайных целых чисел (тип `long`). Форматы метода:

```
public LongStream longs()
public LongStream longs(
       long randomNumberOrigin, long randomNumberBound)
public LongStream longs(long streamSize)
public LongStream longs(long streamSize,
       long randomNumberOrigin, long randomNumberBound)
```

Пример генерации потока из десяти целых чисел (значения типа `long`):

```
Random r = new Random();
LongStream stream = r.longs(10L);
stream.forEachOrdered(x -> System.out.println(x));
```

- `doubles()` — создает поток случайных вещественных чисел. Форматы метода:

```
public DoubleStream doubles()
public DoubleStream doubles(
       double randomNumberOrigin, double randomNumberBound)
public DoubleStream doubles(long streamSize)
public DoubleStream doubles(long streamSize,
       double randomNumberOrigin, double randomNumberBound)
```

Если параметры `randomNumberOrigin` и `randomNumberBound` не указаны, то генерируется диапазон чисел от `0.0` до `1.0` (не включая это значение):

```
Random r = new Random();
DoubleStream stream = r.doubles(10L, 0.0, 50.0);
stream.forEachOrdered(x -> System.out.println(x));
```

21.1.7. Создание пустого потока

Для создания пустого потока предназначен метод `empty()`. Форматы метода:

```
public static IntStream empty()
public static LongStream empty()
public static DoubleStream empty()
public static <T> Stream<T> empty()
```

21.2. Промежуточные операции

Как уже отмечалось ранее, после создания потока над данными могут производиться различные *промежуточные операции* — например, фильтрация данных. Любая промежуточная операция возвращает поток, с которым можно выполнять другие промежуточные операции. Следует учитывать, что промежуточные операции являются «ленивыми», т. е. все промежуточные операции выполняются только при выполнении терминальной операции.

21.2.1. Основные методы

Рассмотрим основные методы, выполняющие промежуточные операции:

- `filter()` — позволяет выполнить фильтрацию данных. Если метод `predicate` вернет значение `false`, то данные не попадут в результирующий поток. Форматы метода:

  ```
  public IntStream filter(IntPredicate predicate)
  public LongStream filter(LongPredicate predicate)
  public DoubleStream filter(DoublePredicate predicate)
  public Stream<T> filter(Predicate<? super T> predicate)
  ```

  Пример получения только нечетных чисел:

  ```
  IntStream.rangeClosed(1, 10)
           .filter( x -> (x % 2) != 0 )
           .forEachOrdered( n -> System.out.print(n + " ") );
  // 1 3 5 7 9
  ```

- `map()` — применяет метод `mapper` к каждому элементу потока. Внутри этого метода необходимо вернуть новое значение или старое. Форматы метода:

  ```
  public IntStream map(IntUnaryOperator mapper)
  public LongStream map(LongUnaryOperator mapper)
  ```

```
public DoubleStream map(DoubleUnaryOperator mapper)
public <R> Stream<R> map(Function<? super T,? extends R> mapper)
```

Умножим каждый элемент на 2:

```
IntStream.rangeClosed(1, 10)
         .map( x -> x * 2 )
         .forEachOrdered( n -> System.out.print(n + " ") );
// 2 4 6 8 10 12 14 16 18 20
```

- `limit()` — **позволяет ограничить поток указанным количеством первых элементов. Форматы метода:**

```
public IntStream limit(long maxSize)
public LongStream limit(long maxSize)
public DoubleStream limit(long maxSize)
public Stream<T> limit(long maxSize)
```

Выведем только первые пять элементов:

```
IntStream.rangeClosed(1, 10)
         .limit(5)
         .forEachOrdered( n -> System.out.print(n + " ") );
// 1 2 3 4 5
```

- `skip()` — **позволяет пропустить указанное количество первых элементов. Форматы метода:**

```
public IntStream skip(long n)
public LongStream skip(long n)
public DoubleStream skip(long n)
public Stream<T> skip(long n)
```

Выведем все элементы, кроме пяти первых:

```
IntStream.rangeClosed(1, 10)
         .skip(5)
         .forEachOrdered( n -> System.out.print(n + " ") );
// 6 7 8 9 10
```

- `distinct()` — **оставляет в потоке только уникальные элементы. Сравнение объектов производится с помощью метода** `equals()`, **а вещественных чисел — с помощью метода** `compare()` **из класса** `Double`. **Форматы метода:**

```
public IntStream distinct()
public LongStream distinct()
public DoubleStream distinct()
public Stream<T> distinct()
```

Пример:

```
IntStream.of(1, 1, 1, 2, 3, 3, 4)
         .distinct()
         .forEachOrdered( n -> System.out.print(n + " ") );
// 1 2 3 4
```

- `peek()` — применяет метод `action` к каждому элементу, при этом не изменяя элементы. Этот метод удобно использовать для отладки. Форматы метода:

```
public IntStream peek(IntConsumer action)
public LongStream peek(LongConsumer action)
public DoubleStream peek(DoubleConsumer action)
public Stream<T> peek(Consumer<? super T> action)
```

Пример вывода промежуточных результатов:

```
List<Integer> arr = Stream.of(1, 1, 1, 2, 3, 3, 4)
   .peek( n -> System.out.print(n + " ") )
   .distinct()
   .collect(Collectors.toList());
// 1 1 1 2 3 3 4
System.out.println(arr.toString());
// [1, 2, 3, 4]
```

- `sorted()` — позволяет отсортировать элементы. Форматы метода:

```
public IntStream sorted()
public LongStream sorted()
public DoubleStream sorted()
public Stream<T> sorted()
public Stream<T> sorted(Comparator<? super T> comparator)
```

В качестве параметра можно указать объект, реализующий интерфейс `Comparator<T>`. Если параметр не указан, то сортируемые объекты должны реализовывать интерфейс `Comparable<T>`. Пример сортировки в прямом и обратном порядке:

```
IntStream.of(9, 1, 3, 2, 5, 7, 6)
         .sorted()
         .forEachOrdered( n -> System.out.print(n + " ") );
// 1 2 3 5 6 7 9
Stream.of(9, 1, 3, 2, 5, 7, 6)
          .parallel()
          .sorted(Collections.reverseOrder())
          .forEachOrdered( n -> System.out.print(n + " ") );
// 9 7 6 5 3 2 1
```

- `parallel()` — преобразует последовательный поток в параллельный. Форматы метода:

```
public IntStream parallel()
public LongStream parallel()
public DoubleStream parallel()
public S parallel()
```

Пример:

```
IntStream.of(9, 1, 3, 2, 5, 7, 6)
         .parallel()
```

```
        .map( x -> x * 2 )
        .sorted()
        .forEachOrdered( n -> System.out.print(n + " ") );
// 2 4 6 10 12 14 18
```

- `sequential()` — преобразует параллельный поток в последовательный. Форматы метода:

```
public IntStream sequential()
public LongStream sequential()
public DoubleStream sequential()
public S sequential()
```

- `unordered()` — делает поток неупорядоченным. Упорядоченный поток требует дополнительных затрат ресурсов для поддержания порядка следования элементов. Если сделать поток неупорядоченным, то при использовании параллельных потоков можно на некоторых операциях получить прирост производительности:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 9, 1, 3, 2, 5, 7, 6);
arr.parallelStream()
   .unordered()
   .map( x -> x * 2 )
   .sorted()
   .forEachOrdered( n -> System.out.print(n + " ") );
// 2 4 6 10 12 14 18
```

- `takeWhile()` — применяет метод `predicate` к элементам потока до тех пор, пока `predicate` возвращает значение `true`. Элемент, для которого метод `predicate` вернул значение `false`, а также все последующие элементы до конца потока будут удалены. Обратите внимание: поток должен быть последовательным и упорядоченным. Метод доступен, начиная с Java 9. Форматы метода:

```
default IntStream takeWhile(IntPredicate predicate)
default LongStream takeWhile(LongPredicate predicate)
default DoubleStream takeWhile(DoublePredicate predicate)
default Stream<T> takeWhile(Predicate<? super T> predicate)
```

  Пример:

```
Stream<Integer> stream = Stream.of(0, 1, 2, 3, 4, 3, 2, 1, 0);
stream.takeWhile(n -> n <= 3)
      .forEachOrdered(x -> System.out.print(x + " "));
// 0 1 2 3
```

- `dropWhile()` — применяет метод `predicate` к элементам потока до тех пор, пока `predicate` возвращает значение `true`. В потоке останется элемент, для которого метод `predicate` вернул значение `false`, а также все последующие элементы до конца потока. Обратите внимание: поток должен быть последовательным и упорядоченным. Метод доступен, начиная с Java 9. Форматы метода:

```
default IntStream dropWhile(IntPredicate predicate)
default LongStream dropWhile(LongPredicate predicate)
default DoubleStream dropWhile(DoublePredicate predicate)
default Stream<T> dropWhile(Predicate<? super T> predicate)
```

Пример:

```
Stream<Integer> stream = Stream.of(0, 1, 2, 3, 4, 3, 2, 1, 0);
stream.dropWhile(n -> n <= 3)
      .forEachOrdered(x -> System.out.print(x + " "));
// 4 3 2 1 0
```

21.2.2. Преобразование типа потока

Поток одного типа можно преобразовать в поток другого типа с помощью следующих методов (все методы осуществляют промежуточные операции):

- `asLongStream()` — **преобразует поток** `IntStream` **в поток** `LongStream`. **Формат метода:**

  ```
  public LongStream asLongStream()
  ```

- `asDoubleStream()` — **преобразует потоки** `IntStream` **и** `LongStream` **в поток** `DoubleStream`. **Формат метода:**

  ```
  public DoubleStream asDoubleStream()
  ```

  Пример:

  ```
  IntStream.of(1, 2, 3, 4, 5)
           .asDoubleStream()
           .forEachOrdered( n -> System.out.print(n + " ") );
  // 1.0 2.0 3.0 4.0 5.0
  ```

- `boxed()` — **преобразует поток элементарных данных в поток объектных данных. Форматы метода:**

  ```
  public Stream<Integer> boxed()
  public Stream<Long> boxed()
  public Stream<Double> boxed()
  ```

  Пример преобразования массива целых чисел в список:

  ```
  int[] a = {1, 2, 3, 4, 5};
  List<Integer> arr = IntStream.of(a)
                               .boxed()
                               .collect(Collectors.toList());
  System.out.println(arr); // [1, 2, 3, 4, 5]
  ```

- `mapToInt()` — **преобразует поток в** `IntStream`, **используя метод** `mapper`. **Форматы метода:**

  ```
  public IntStream mapToInt(LongToIntFunction mapper)
  public IntStream mapToInt(DoubleToIntFunction mapper)
  public IntStream mapToInt(ToIntFunction<? super T> mapper)
  ```

Пример:

```
DoubleStream.of(1.2, 5.0, 4.3)
   .mapToInt( x -> (int)x )
   .forEachOrdered( n -> System.out.print(n + " ") );
//  1 5 4
```

- `mapToLong()` — **преобразует поток в** `LongStream`, **используя метод** `mapper`. **Форматы метода:**

```
public LongStream mapToLong(IntToLongFunction mapper)
public LongStream mapToLong(DoubleToLongFunction mapper)
public LongStream mapToLong(ToLongFunction<? super T> mapper)
```

Пример:

```
Stream.of(1.2, 5.0, 4.3)
   .mapToLong( x -> x.longValue() )
   .forEachOrdered( x -> System.out.print(x + " ") );
// 1 5 4
```

- `mapToDouble()` — **преобразует поток в** `DoubleStream`, **используя метод** `mapper`. **Форматы метода:**

```
public DoubleStream mapToDouble(IntToDoubleFunction mapper)
public DoubleStream mapToDouble(LongToDoubleFunction mapper)
public DoubleStream mapToDouble(
                        ToDoubleFunction<? super T> mapper)
```

Пример:

```
Stream.of(1, 5, 4)
   .mapToDouble( x -> x.doubleValue() )
   .forEachOrdered( x -> System.out.print(x + " ") );
// 1.0 5.0 4.0
```

- `mapToObj()` — **преобразует поток элементарных данных в объектный поток, используя метод** `mapper`. **Форматы метода:**

```
public <U> Stream<U> mapToObj(IntFunction<? extends U> mapper)
public <U> Stream<U> mapToObj(LongFunction<? extends U> mapper)
public <U> Stream<U> mapToObj(
                           DoubleFunction<? extends U> mapper)
```

Пример:

```
int[] a = {1, 2, 3, 4, 5};
List<Integer> arr = IntStream.of(a)
                        .mapToObj( x -> Integer.valueOf(x) )
                        .collect(Collectors.toList());
System.out.println(arr); // [1, 2, 3, 4, 5]
```

21.2.3. Объединение и добавление потоков

Объединить потоки данных позволяют следующие методы:

- `concat()` — объединяет два однотипных потока. Форматы метода:

```
public static IntStream concat(IntStream a, IntStream b)
public static LongStream concat(LongStream a, LongStream b)
public static DoubleStream concat(DoubleStream a,
                                  DoubleStream b)
public static <T> Stream<T> concat(Stream<? extends T> a,
                                   Stream<? extends T> b)
```

Пример:

```
IntStream s1 = IntStream.of(1, 2, 3, 4, 5);
IntStream s2 = IntStream.of(6, 7, 8, 9, 10);
IntStream.concat(s1, s2)
   .forEachOrdered( n -> System.out.print(n + " ") );
// 1 2 3 4 5 6 7 8 9 10
```

- `flatMap()` — позволяет на основе одного элемента добавить поток элементов. Например, мы читаем файл построчно, но нам нужно получить из каждой строки все слова по отдельности. На каждой итерации в качестве параметра метода `mapper` будет доступна текущая строка, а вернуть нужно поток слов. Форматы метода:

```
public IntStream flatMap(
    IntFunction<? extends IntStream> mapper)
public LongStream flatMap(
    LongFunction<? extends LongStream> mapper)
public DoubleStream flatMap(
    DoubleFunction<? extends DoubleStream> mapper)
public <R> Stream<R> flatMap(
    Function<? super T, ? extends Stream<? extends R>> mapper)
```

Пример:

```
Pattern p = Pattern.compile("\\s+");
List<String> arr = new ArrayList<String>();
arr.add("слово1 слово2 слово3");
arr.add("слово4 слово5 слово6");
Stream<String> stream = arr.stream();
stream.flatMap( str -> p.splitAsStream(str) )
   .forEachOrdered( str -> System.out.print(str + ";") );
// слово1;слово2;слово3;слово4;слово5;слово6;
```

- `flatMapToInt()`, `flatMapToLong()` и `flatMapToDouble()` — создают потоки элементарных данных на основе объектного потока. На каждой итерации в качестве параметра метода `mapper` будет доступен текущий объект, а вернуть нужно поток элементарных данных. Форматы методов:

```
public IntStream flatMapToInt(
       Function<? super T, ? extends IntStream> mapper)
public LongStream flatMapToLong(
       Function<? super T, ? extends LongStream> mapper)
public DoubleStream flatMapToDouble(
       Function<? super T, ? extends DoubleStream> mapper)
```

Пример:

```
List<String> arr = new ArrayList<String>();
Collections.addAll(arr, "1", "2", "3");
Stream<String> stream = arr.stream();
stream.flatMapToInt( s -> IntStream.of(Integer.parseInt(s)) )
   .forEachOrdered( x -> System.out.print(x + " ") );
```

21.3. Терминальные операции

Все промежуточные операции выполняются только после вызова *терминальной операции*, которая возвращает какое-либо значение, отличное от потока. Терминальная операция завершает работу с потоком.

21.3.1. Основные методы

Рассмотрим основные методы, выполняющие терминальную операцию:

- `forEach()` — перебирает все элементы потока. Обратите внимание: при использовании параллельного потока порядок следования элементов не гарантируется! Форматы метода:

```
public void forEach(IntConsumer action)
public void forEach(LongConsumer action)
public void forEach(DoubleConsumer action)
public void forEach(Consumer<? super T> action)
```

Пример:

```
IntStream.of(1, 2, 3, 4, 5)
   .parallel()
   .forEach( n -> System.out.print(n + " ") );
// 3 2 1 5 4
```

- `forEachOrdered()` — перебирает все элементы потока. Порядок следования элементов гарантируется. Форматы метода:

```
public void forEachOrdered(IntConsumer action)
public void forEachOrdered(LongConsumer action)
public void forEachOrdered(DoubleConsumer action)
public void forEachOrdered(Consumer<? super T> action)
```

Пример:

```
IntStream.of(1, 2, 3, 4, 5)
   .parallel()
```

```
    .forEachOrdered( n -> System.out.print(n + " ") );
// 1 2 3 4 5
```

- `count()` — **возвращает количество элементов в потоке. Формат метода:**

```
public long count()
```

Пример:

```
System.out.println(IntStream.of(1, 2, 3, 4, 5).count());
// 5
```

- `anyMatch()` — **возвращает значение** `true`**, если условие** `predicate` **выполняется хотя бы для одного элемента потока. Форматы метода:**

```
public boolean anyMatch(IntPredicate predicate)
public boolean anyMatch(LongPredicate predicate)
public boolean anyMatch(DoublePredicate predicate)
public boolean anyMatch(Predicate<? super T> predicate)
```

Пример проверки существования четного числа:

```
System.out.println(IntStream.of(1, 2, 3, 5)
    .anyMatch(x -> (x % 2) == 0));
// true
System.out.println(IntStream.of(1, 3, 5)
    .anyMatch(x -> (x % 2) == 0));
// false
```

- `noneMatch()` — **возвращает значение** `true`**, если условие** `predicate` **не выполняется ни для одного элемента потока. Форматы метода:**

```
public boolean noneMatch(IntPredicate predicate)
public boolean noneMatch(LongPredicate predicate)
public boolean noneMatch(DoublePredicate predicate)
public boolean noneMatch(Predicate<? super T> predicate)
```

Пример:

```
System.out.println(IntStream.of(1, 2, 3, 5)
    .noneMatch(x -> (x % 2) == 0));
// false
System.out.println(IntStream.of(1, 3, 5)
    .noneMatch(x -> (x % 2) == 0));
// true
```

- `allMatch()` — **возвращает значение** `true`**, если условие** `predicate` **выполняется для всех элементов потока. Форматы метода:**

```
public boolean allMatch(IntPredicate predicate)
public boolean allMatch(LongPredicate predicate)
public boolean allMatch(DoublePredicate predicate)
public boolean allMatch(Predicate<? super T> predicate)
```

Пример:

```
System.out.println(IntStream.of(1, 2, 3, 5)
    .allMatch(x -> (x % 2) != 0));
// false
System.out.println(IntStream.of(1, 3, 5)
    .allMatch(x -> (x % 2) != 0));
// true
```

- `findFirst()` — **возвращает первый элемент потока в виде объекта классов** `OptionalInt`, `OptionalLong`, `OptionalDouble` **или** `Optional<T>`. **Форматы метода:**

```
import java.util.*;
public OptionalInt findFirst()
public OptionalLong findFirst()
public OptionalDouble findFirst()
public Optional<T> findFirst()
```

Пример:

```
Optional<Integer> obj = Stream.of(1, 2, 3, 5).findFirst();
System.out.println(obj.get());
// 1
OptionalInt obj2 = IntStream.of(1, 2, 3, 5).findFirst();
System.out.println(obj2.getAsInt());
// 1
```

- `findAny()` — **возвращает произвольный элемент из потока в виде объекта классов** `OptionalInt`, `OptionalLong`, `OptionalDouble` **или** `Optional<T>`. **Форматы метода:**

```
import java.util.*;
public OptionalInt findAny()
public OptionalLong findAny()
public OptionalDouble findAny()
public Optional<T> findAny()
```

Пример:

```
Optional<Integer> obj =
   Stream.of(1, 2, 3, 5).parallel().findFirst();
System.out.println(obj.get());
// 1
Optional<Integer> obj2 =
   Stream.of(1, 2, 3, 5).parallel().findAny();
System.out.println(obj2.get());
// 3
```

- `min()` — **возвращает элемент с минимальным значением в виде объекта классов** `OptionalInt`, `OptionalLong`, `OptionalDouble` **или** `Optional<T>`. **Форматы метода:**

```
import java.util.*;
public OptionalInt min()
```

```
public OptionalLong min()
public OptionalDouble min()
public Optional<T> min(Comparator<? super T> comparator)
```

Пример:

```
System.out.println(
   LongStream.of(1, 2, 3, 4, 5).min().getAsLong()); // 1
Optional<Integer> obj =
   Stream.of(1, 2, 3, 5).min( (a, b) -> a.compareTo(b) );
System.out.println(obj.get()); // 1
```

- `max()` — **возвращает элемент с максимальным значением в виде объекта классов** `OptionalInt`, `OptionalLong`, `OptionalDouble` **или** `Optional<T>`. **Форматы метода:**

```
import java.util.*;
public OptionalInt max()
public OptionalLong max()
public OptionalDouble max()
public Optional<T> max(Comparator<? super T> comparator)
```

Пример:

```
System.out.println(
   IntStream.of(1, 2, 3, 4, 5).max().getAsInt()); // 5
Optional<Integer> obj =
   Stream.of(1, 2, 3, 5).max( (a, b) -> a.compareTo(b) );
System.out.println(obj.get()); // 5
```

- `sum()` — **возвращает сумму чисел для потоков со значениями элементарных типов. Форматы метода:**

```
public int sum()
public long sum()
public double sum()
```

Пример:

```
System.out.println(IntStream.of(1, 2, 3, 4, 5).sum()); // 15
```

- `average()` — **возвращает среднее арифметическое всех чисел для потоков со значениями элементарных типов. Формат метода:**

```
public OptionalDouble average()
```

Пример:

```
System.out.println(
IntStream.of(1, 2, 3, 4, 5).average().getAsDouble()); // 3.0
```

- `reduce()` — **применяет указанный метод к элементам и накапливает результат. Метод** `reduce()` **имеет несколько вариантов. Форматы первого варианта:**

```
public int reduce(int identity, IntBinaryOperator op)
public long reduce(long identity, LongBinaryOperator op)
```

```
public double reduce(double identity, DoubleBinaryOperator op)
public T reduce(T identity, BinaryOperator<T> op)
```

Параметр `identity` задает начальное значение, а параметр `op` — метод, в первом параметре принимающий промежуточное значение, а во втором параметре — значение текущего элемента. Получим сумму всех чисел внутри потока:

```
System.out.println(
   IntStream.of(1, 2, 3, 4, 5)
            .reduce(0, (sum, b) -> sum + b )); // 15
```

Форматы второго варианта:

```
public OptionalInt reduce(IntBinaryOperator op)
public OptionalLong reduce(LongBinaryOperator op)
public OptionalDouble reduce(DoubleBinaryOperator op)
public Optional<T> reduce(BinaryOperator<T> accumulator)
```

Второй формат не имеет начального значения и вместо определенного типа возвращает объекты `OptionalInt`, `OptionalLong`, `OptionalDouble` или `Optional<T>`:

```
Optional<Integer> obj =
   Stream.of(1, 2, 3, 4, 5).reduce((sum, b) -> sum + b );
System.out.println(obj.get()); // 15
```

Третий вариант есть только в интерфейсе `Stream<T>`:

```
public <U> U reduce(U identity,
             BiFunction<U, ? super T, U> accumulator,
             BinaryOperator<U> combiner)
```

Параметр `identity` задает начальное значение, а параметр `accumulator` — метод, в первом параметре принимающий промежуточное значение, а во втором параметре — значение текущего элемента. Метод `combiner`, указанный в третьем параметре, вызывается только при использовании параллельных потоков и комбинирует промежуточные значения. Рассмотрим последовательность вызова методов на примере:

```
Integer obj = Stream.of(1, 2, 3).parallel()
   .reduce(0,
    (a, b) -> {
       System.out.println("accumulator: a=" + a + " b=" + b);
       return a + b;
    },
    (a, b) -> {
       System.out.println("combiner: a=" + a + " b=" + b);
       return a + b; });
System.out.println(obj);
```

Результат при использовании параллельного потока:

```
accumulator: a=0 b=2
accumulator: a=0 b=1
accumulator: a=0 b=3
```

```
combiner: a=2 b=3
combiner: a=1 b=5
6
```

Если мы уберем вызов метода `parallel()`, то результат будет другим:

```
accumulator: a=0 b=1
accumulator: a=1 b=2
accumulator: a=3 b=3
6
```

Обратите внимание: при последовательной обработке первый параметр метода `accumulator` будет содержать сумму предыдущих значений, а при параллельной — значение параметра `identity`. Если параметр `identity` равен `0`, то результат получится одинаковым, а если параметр будет иметь другое значение, то при последовательной и параллельной обработке результат окажется разным:

```
System.out.println(IntStream.of(1, 2, 3)
        .reduce(6, Integer::sum)); // 12
System.out.println(IntStream.of(1, 2, 3).parallel()
        .reduce(6, Integer::sum)); // 24 !!!
```

- `close()` — закрывает поток. Формат метода:

```
// Интерфейс AutoCloseable
public void close()
```

Пример:

```
Stream<Integer> stream = Stream.of(0, 1, 2, 3);
stream.onClose(() -> System.out.println("Поток закрыт")).close();
// Поток закрыт
```

21.3.2. Класс *Optional<T>*

Многие методы, выполняющие терминальную операцию, возвращают объект класса `Optional<T>` (или `OptionalInt`, `OptionalLong`, `OptionalDouble`), т. к. поток может не содержать элементов, а возвращать значение `null` нельзя, ибо `null` — это одно из возможных значений потока.

Класс содержит следующие основные методы:

- `of()` — создает объект с указанным значением. Если значение равно `null`, то генерируется исключение. Форматы метода:

```
import java.util.*;
public static OptionalInt of(int value)
public static OptionalLong of(long value)
public static OptionalDouble of(double value)
public static <T> Optional<T> of(T value)
```

Пример:

```
OptionalInt obj = OptionalInt.of(10);
System.out.println(obj.getAsInt());       // 10
```

```
Optional<Integer> obj2 = Optional.of(10);
System.out.println(obj2.orElseThrow());   // 10
```

- `empty()` — создает пустой объект. Форматы метода:

```
public static OptionalInt empty()
public static OptionalLong empty()
public static OptionalDouble empty()
public static <T> Optional<T> empty()
```

Пример:

```
Optional<Integer> obj = Optional.<Integer>empty();
System.out.println(obj.isPresent());  // false
System.out.println(obj.orElse(10));   // 10
```

- `ofNullable()` — создает объект с указанным значением. Если значение равно `null`, то возвращает пустой объект. Формат метода:

```
public static <T> Optional<T> ofNullable(T value)
```

Пример:

```
Optional<Integer> obj = Optional.ofNullable(10);
System.out.println(obj.isPresent());      // true
System.out.println(obj.orElse(0));        // 10
obj = Optional.<Integer>ofNullable(null);
System.out.println(obj.isPresent());      // false
System.out.println(obj.orElse(0));        // 0
```

- `get()` — возвращает значение. Если значения нет, то генерируется исключение. Форматы методов:

```
public int getAsInt()
public long getAsLong()
public double getAsDouble()
public T get()
```

Пример:

```
System.out.println(
   IntStream.of(1, 2, 3).max().getAsInt()); // 3
Optional<Integer> obj = Stream.of(1, 2, 3).findFirst();
System.out.println(obj.get()); // 1
```

- `orElse()` — возвращает значение. Если значения нет, то возвращается значение, указанное в параметре метода. Форматы метода:

```
public int orElse(int other)
public long orElse(long other)
public double orElse(double other)
public T orElse(T other)
```

Пример:

```
System.out.println(
   IntStream.of().max().orElse(0));        // 0
```

```
System.out.println(
   IntStream.of(1, 2, 3).max().orElse(0)); // 3
```

- `orElseGet()` — возвращает значение. Если значения нет, то вызывается метод, указанный в качестве параметра, и возвращается результат его выполнения. Форматы метода:

```
public int orElseGet(IntSupplier other)
public long orElseGet(LongSupplier other)
public double orElseGet(DoubleSupplier other)
public T orElseGet(Supplier<? extends T> other)
```

Пример:

```
Optional<Integer> obj = Stream.of(1, 2, 3).findFirst();
System.out.println(obj.orElseGet(() -> 0)); // 1
```

- `orElseThrow()` — возвращает значение. Форматы метода:

```
public <X extends Throwable> int
orElseThrow(Supplier<? extends X> exceptionSupplier)
                        throws X extends Throwable
public <X extends Throwable> long
orElseThrow(Supplier<? extends X> exceptionSupplier)
                        throws X extends Throwable
public <X extends Throwable> double
orElseThrow(Supplier<? extends X> exceptionSupplier)
                        throws X extends Throwable
public <X extends Throwable> T
orElseThrow(Supplier<? extends X> exceptionSupplier)
                        throws X extends Throwable
```

Если значения нет, то вызывается метод, указанный в качестве параметра, и генерируется исключение, класс которого возвращается методом:

```
Optional<Integer> obj = Stream.of(1, 2, 3).findFirst();
System.out.println(
   obj.orElseThrow(NoSuchElementException::new)); // 1
```

В Java 10 были добавлены следующие форматы метода `orElseThrow()`:

```
public int orElseThrow()
public long orElseThrow()
public double orElseThrow()
public T orElseThrow()
```

Если значения нет, то генерируется исключение `NoSuchElementException`:

```
Optional<Integer> obj = Stream.of(1, 2, 3).findFirst();
System.out.println(obj.orElseThrow()); // 1
```

- `ifPresent()` — если значение есть, то вызывается метод, указанный в качестве параметра. Форматы метода:

```
public void ifPresent(IntConsumer consumer)
public void ifPresent(LongConsumer consumer)
public void ifPresent(DoubleConsumer consumer)
public void ifPresent(Consumer<? super T> consumer)
```

Пример:

```
IntStream.of(1, 2, 3).max()
         .ifPresent((x) -> System.out.println(x)); // 3
```

- `ifPresentOrElse()` — если значение есть, то вызывается метод `action`, в противном случае — метод `emptyAction`. Метод доступен, начиная с Java 9. Форматы метода:

```
public void ifPresentOrElse(IntConsumer action,
                            Runnable emptyAction)
public void ifPresentOrElse(LongConsumer action,
                            Runnable emptyAction)
public void ifPresentOrElse(DoubleConsumer action,
                            Runnable emptyAction)
public void ifPresentOrElse(Consumer<? super T> action,
                            Runnable emptyAction)
```

Функциональный интерфейс `Runnable` содержит описание метода `run()`:

```
public void run()
```

Пример:

```
IntStream.of(1, 2, 3)
         //.skip(3)
         .max().ifPresentOrElse(
                x -> System.out.println(x),
               () -> System.out.println("Нет значения")); // 3
```

Если удалить символы комментария перед методом `skip()`, то получим сообщение `Нет значения`;

- `isPresent()` — возвращает `true`, если значение есть, и `false` — в противном случае. Формат метода:

```
public boolean isPresent()
```

Пример:

```
Optional<Integer> obj = Stream.of(1, 2, 3).findFirst();
if (obj.isPresent()) {
   System.out.println(obj.get()); // 1
}
```

- `or()` — если значение есть, то метод возвращает его, а если нет, то вызывает метод `supplier`, который должен вернуть новое значение. Метод доступен, начиная с Java 9. Формат метода:

```
public Optional<T> or(
       Supplier<? extends Optional<? extends T>> supplier)
```

Пример:

```
Optional<Integer> obj = Stream.of(1)
                        .skip(1)
                        .findFirst().or( () -> Optional.of(0) );
System.out.println(obj.orElseThrow()); // 0
```

Если добавить символы комментария перед методом `skip()`, то получим значение `1`;

- `stream()` — создает поток на основе значения. Если значения нет, то метод возвращает пустой поток. Метод доступен, начиная с Java 9. Форматы метода:

```
public IntStream stream()
public LongStream stream()
public DoubleStream stream()
public Stream<T> stream()
```

Пример:

```
OptionalInt obj = OptionalInt.of(10);
System.out.println(obj.stream().count());  // 1
obj = OptionalInt.empty();
System.out.println(obj.stream().count());  // 0
```

- `filter()` — позволяет выполнить фильтрацию значения. Если метод `predicate` вернет значение `false`, то возвращается пустой объект. Формат метода:

```
public Optional<T> filter(Predicate<? super T> predicate)
```

Пример:

```
Optional<Integer> obj = Stream.of(1).findFirst()
                        .filter( n -> n < 0 );
System.out.println(obj.orElse(0)); // 0
```

Если изменить условие на `n > 0`, то получим значение `1`;

- `map()` — применяет метод `mapper` к значению. Внутри этого метода необходимо вернуть новое значение или старое. Формат метода:

```
public <U> Optional<U> map(
              Function<? super T, ? extends U> mapper)
```

Пример:

```
Optional<Integer> obj = Stream.of(1).findFirst()
                                .map( n -> n + 1 );
System.out.println(obj.orElse(0)); // 2
```

- `flatMap()` — применяет метод `mapper` к значению. Внутри этого метода необходимо вернуть новое значение в виде объекта `Optional<T>`. Формат метода:

```
public <U> Optional<U> flatMap(
        Function<? super T,
                 ? extends Optional<? extends U> > mapper)
```

Пример преобразования числового значения в строку:

```
Optional<String> obj = Stream.of(1, 2, 3).findFirst()
        .flatMap( n -> Optional.of(String.valueOf(n)) );
System.out.println(obj.orElse(""));  // 1
```

21.3.3. Преобразование потока в коллекцию, массив или в другой объект

После выполнения промежуточных операций поток можно преобразовать в коллекцию, массив или в другой объект. Для этого предназначены следующие методы:

- `toArray()` — возвращает массив. Форматы метода:

```
public int[] toArray()
public long[] toArray()
public double[] toArray()
public Object[] toArray()
public <A> A[] toArray(IntFunction<A[]> generator)
```

Пример:

```
int[] arr = IntStream.rangeClosed(1, 10)
                     .map( x -> x * 2 ).toArray();
System.out.print(Arrays.toString(arr));
// [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
Integer[] obj = Stream.of(1, 2, 3).toArray(Integer[]::new);
System.out.println(Arrays.toString(obj)); // [1, 2, 3]
```

- `collect()` — позволяет преобразовать поток в коллекцию или в другой объект. Форматы метода:

```
public <R> R collect(Supplier<R> supplier,
                     ObjIntConsumer<R> accumulator,
                     BiConsumer<R, R> combiner)
public <R> R collect(Supplier<R> supplier,
                     ObjLongConsumer<R> accumulator,
                     BiConsumer<R, R> combiner)
public <R> R collect(Supplier<R> supplier,
                     ObjDoubleConsumer<R> accumulator,
                     BiConsumer<R, R> combiner)
public <R,A> R collect(Collector<? super T, A, R> collector)
public <R> R collect(Supplier<R> supplier,
                     BiConsumer<R, ? super T> accumulator,
                     BiConsumer<R, R> combiner)
```

Пример:

```
List<Integer> obj = Stream.of(1, 2, 3)
                          .collect(Collectors.toList());
System.out.println(obj.toString()); // [1, 2, 3]
```

```
String s = Stream.of(1, 2, 3).collect(
   StringBuilder::new, StringBuilder::append,
   StringBuilder::append).toString();
System.out.println(s); // 123
```

В качестве параметра метод `collect()` может принимать объект, реализующий интерфейс `Collector<T, A, R>`. Класс `Collectors` содержит множество методов, возвращающих такой объект. Рассмотрим основные его методы (полный их список смотрите в документации):

- `toList()` — преобразует поток в список. Формат метода:

```
public static <T> Collector<T, ?, List<T>> toList()
```

Пример:

```
List<Integer> obj = Stream.of(1, 1, 1, 2, 3, 3, 4)
   .distinct().collect(Collectors.toList());
System.out.println(obj.toString()); // [1, 2, 3, 4]
```

- `toUnmodifiableList()` — преобразует поток в неизменяемый список. Метод доступен, начиная с Java 10. Формат метода:

```
public static <T> Collector<T, ?, List<T>> toUnmodifiableList()
```

Пример:

```
List<Integer> obj = Stream.of(1, 1, 1, 2, 3, 3, 4)
   .distinct().collect(Collectors.toUnmodifiableList());
System.out.println(obj.toString()); // [1, 2, 3, 4]
```

- `toSet()` — преобразует поток во множество. Формат метода:

```
public static <T> Collector<T, ?, Set<T>> toSet()
```

Пример:

```
Set<Integer> obj = Stream.of(1, 1, 1, 2, 3, 3, 4)
                         .collect(Collectors.toSet());
System.out.println(obj.toString()); // [1, 2, 3, 4]
```

- `toUnmodifiableSet()` — преобразует поток в неизменяемое множество. Метод доступен, начиная с Java 10. Формат метода:

```
public static <T> Collector<T, ?, Set<T>> toUnmodifiableSet()
```

Пример:

```
Set<Integer> obj = Stream.of(1, 1, 1, 2, 3, 3, 4)
                  .collect(Collectors.toUnmodifiableSet());
System.out.println(obj.toString()); // [1, 2, 3, 4]
```

- `joining()` — объединяет элементы потока в строку через указанный разделитель. Форматы метода:

```
public static Collector<CharSequence, ?, String> joining()
public static Collector<CharSequence, ?, String>
       joining(CharSequence delimiter)
```

```
public static Collector<CharSequence, ?, String>
       joining(CharSequence delimiter,
               CharSequence prefix, CharSequence suffix)
```

Параметр `delimiter` **задает разделитель, параметр** `prefix` **— фрагмент в начале строки, а** `suffix` **— фрагмент в конце строки:**

```
String obj = Stream.of("1", "2", "3")
           .collect(Collectors.joining());
System.out.println(obj.toString()); // 123
obj = Stream.of("1", "2", "3")
           .collect(Collectors.joining(", "));
System.out.println(obj.toString()); // 1, 2, 3
obj = Stream.of("1", "2", "3")
           .collect(Collectors.joining(" - ", "->", "<-"));
System.out.println(obj.toString()); // ->1 - 2 - 3<-
```

ГЛАВА 22

Взаимодействие с сетью Интернет

Для изучения взаимодействия с Интернетом и работы с базами данных нам понадобится установить на компьютер пакет XAMPP, который включает Web-сервер Apache, языки PHP и Perl, систему управления базами данных (СУБД) MySQL (MariaDB) и phpMyAdmin — приложение для администрирования СУБД MySQL.

Для загрузки этого пакета переходим на страницу **https://www.apachefriends.org/index.html** и скачиваем установочный EXE-файл (в моем случае он называется xampp-win32-7.2.0-0-VC15-installer.exe). Процесс установки очень прост и в комментариях не нуждается. На всех шагах мастера соглашаемся с настройками по умолчанию. Очень важно при установке разрешить запуск серверов Apache и MySQL. Если на компьютере установлены брандмауэр Windows или антивирусная программа, то эти серверы необходимо добавить в списки их исключений.

В итоге пакет будет установлен в папку C:\xampp. Запускаем **XAMPP Control Panel** (файл C:\xampp\xampp-control.exe) и пробуем запустить серверы. Для этого щелкаем на кнопках **Start** у пунктов **Apache** и **MySQL**. Результат успешного запуска показан на рис. 22.1. Если антивирусная программа или брандмауэр сообщают, что программа заблокирована, то необходимо обязательно разрешить ее запуск. Web-сервер Apache запускается на портах 80 и 443, а сервер MySQL использует порт 3306. Для проверки работоспособности открываем браузер и в адресной строке вводим: `http://localhost/` — должно отобразиться приветствие пакета XAMPP (рис. 22.2). Для проверки работоспособности PHP вводим: `http://localhost/dashboard/phpinfo.php` (результат показан на рис. 22.3), а для проверки phpMyAdmin — `http://localhost/phpmyadmin/` (результат показан на рис. 22.4).

Если серверы не запустились, то открываем командную строку (обязательно с привилегиями администратора!), набираем команду:

```
netstat -anb
```

и смотрим, какие программы занимают порты 80, 443 и 3306. Обычно эти порты занимают программы Skype и Web-сервер IIS. Перед запуском Apache эти программы нужно остановить.

Теперь попробуем написать приветствие на языке PHP. Для этого открываем файл C:\xampp\htdocs\index.php, с помощью текстового редактора (например, Notepad++)

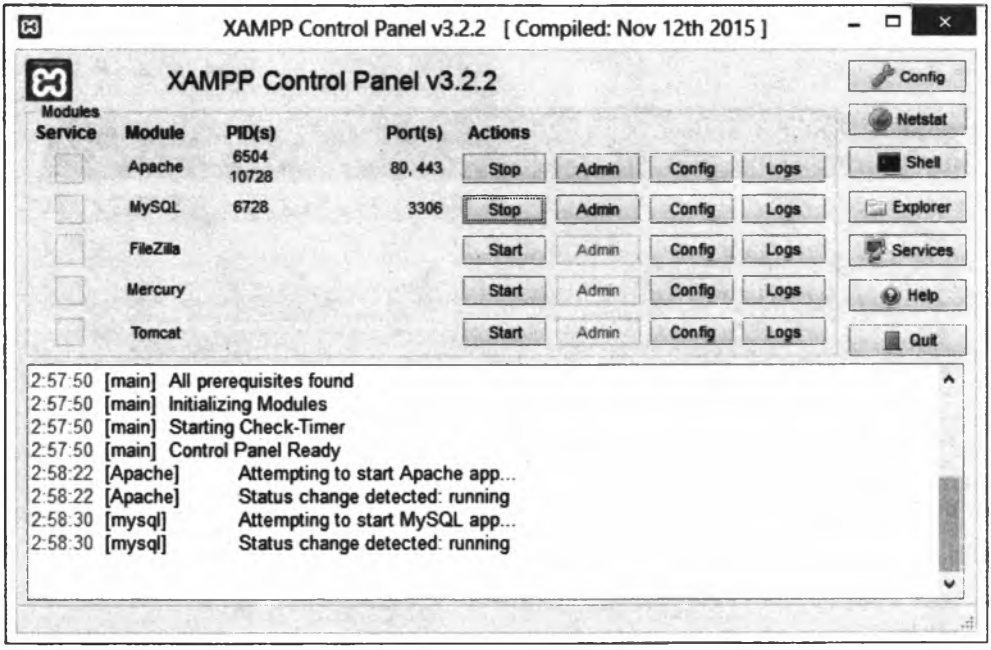

Рис. 22.1. Окно **XAMPP Control Panel**: серверы Apache и MySQL успешно запущены

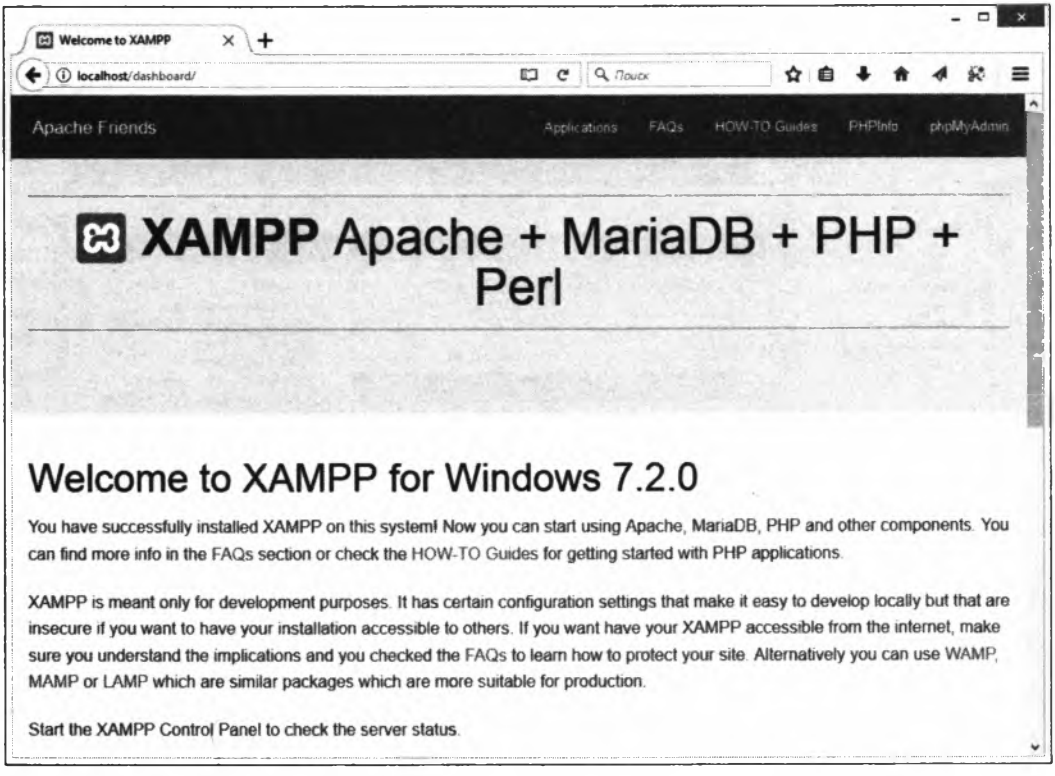

Рис. 22.2. Приветствие пакета XAMPP в окне Web-браузера

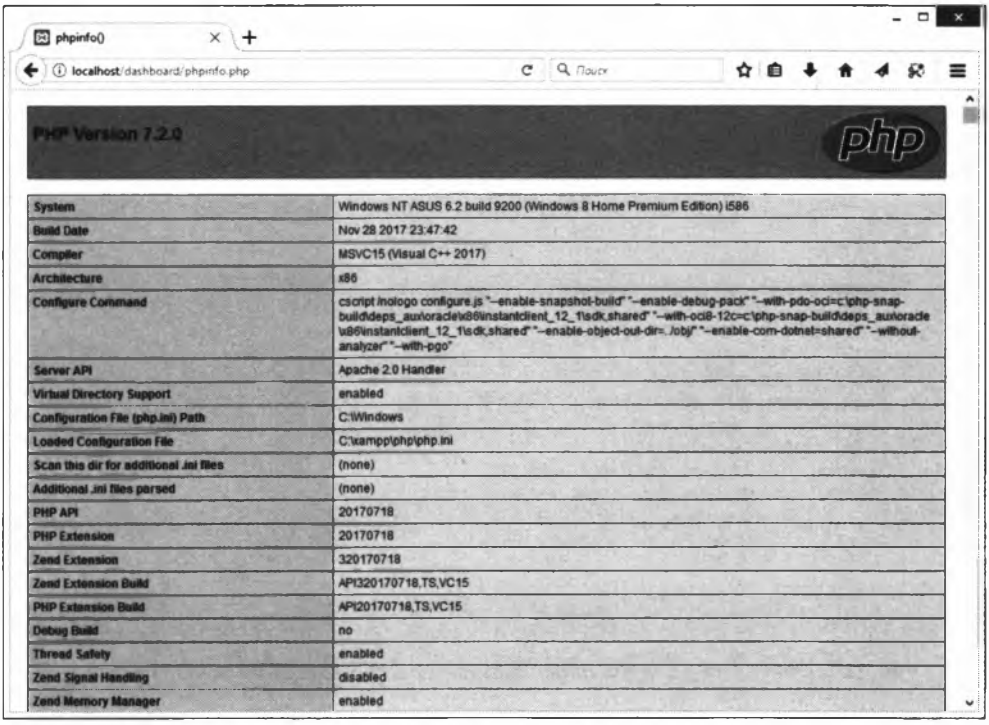

Рис. 22.3. Результат проверки работоспособности PHP

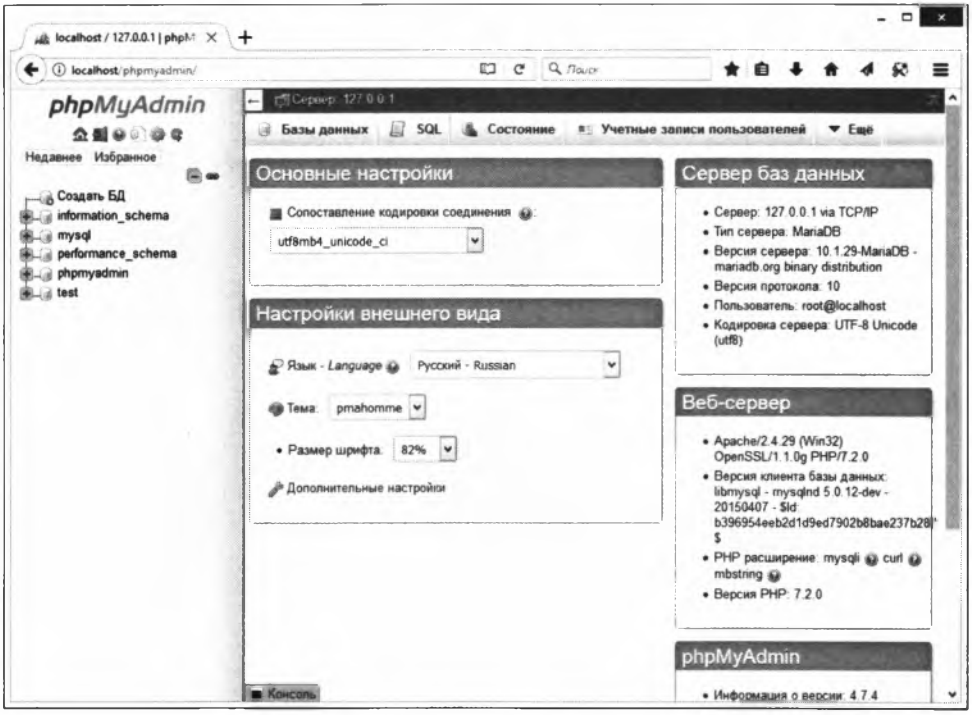

Рис. 22.4. Программа phpMyAdmin

стираем весь текст, набираем код из листинга 22.1 и сохраняем файл в кодировке UTF-8 (без BOM).

Листинг 22.1. Приветствие на языке PHP

```
<?php
echo "Привет, мир!";
?>
```

В адресной строке Web-браузера набираем: `http://localhost/index.php` или просто: `http://localhost/`. Если приветствие отобразилось, то вы готовы изучать язык PHP, — но не торопитесь, мы ведь еще не закончили изучение языка Java! Просто теперь мы готовы приступить к рассмотрению работы с Интернетом и базой данных MySQL.

22.1. Класс *URI*

Класс `URI` описывает идентификаторы в различных схемах, в том числе и URL-адреса. Прежде чем использовать этот класс, необходимо выполнить его импорт с помощью инструкции:

```
import java.net.URI;
```

Создать экземпляр класса позволяют следующие конструкторы:

```
URI(String str)                          throws URISyntaxException
URI(String scheme, String host, String path, String fragment)
                                         throws URISyntaxException
URI(String scheme, String authority, String path,
    String query, String fragment)
                                         throws URISyntaxException
URI(String scheme, String ssp, String fragment)
                                         throws URISyntaxException
URI(String scheme, String userInfo, String host,
    int port, String path, String query, String fragment)
                                         throws URISyntaxException
```

Первый конструктор принимает идентификатор в виде строки:

```
URI uri = new URI("http://localhost/index.php");
System.out.println(uri.toString());
// http://localhost/index.php
```

Второй конструктор позволяет указать по отдельности протокол, домен, путь и якорь:

```
URI uri = new URI("http", "localhost", "/index.php", "metka");
System.out.println(uri.toString());
// http://localhost/index.php#metka
```

Третий конструктор позволяет дополнительно указать строку запроса:

```
URI uri = new URI("http", "localhost", "/index.php",
                  "x=5&y=3", "metka");
System.out.println(uri.toString());
// http://localhost/index.php?x=5&y=3#metka
```

Четвертый конструктор удобно использовать для почтового идентификатора:

```
URI uri = new URI("mailto", "user@mail.ru", null);
System.out.println(uri.toString());
// mailto:user@mail.ru
```

Пятый конструктор позволяет указать FTP-идентификатор:

```
URI uri = new URI("ftp", "user:123", "localhost",
                   21, "/index.php", null, null);
System.out.println(uri.toString());
// ftp://user:123@localhost:21/index.php
```

Для создания объекта предназначен также статический метод `create()`. Формат метода:

```
public static URI create(String str)
```

Пример:

```
URI uri = URI.create("http://localhost/index.php");
System.out.println(uri.toString());
// http://localhost/index.php
```

Класс `URI` содержит следующие основные методы:

- `toString()` — возвращает идентификатор в виде строки. Формат метода:

  ```
  public String toString()
  ```

  Пример:

  ```
  URI uri =
      new URI("http://localhost:80/index.php?x=5&y=3#metka");
  System.out.println(uri.toString());
  // http://localhost:80/index.php?x=5&y=3#metka
  ```

- `toASCIIString()` — возвращает идентификатор в виде строки, кодируя при этом символы. Формат метода:

  ```
  public String toASCIIString()
  ```

  Пример:

  ```
  URI uri2 = new URI("/index.php?s=строка");
  System.out.println(uri2.toASCIIString());
  // /index.php?s=%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0
  ```

- `getScheme()` — возвращает схему или значение `null`. Формат метода:

  ```
  public String getScheme()
  ```

Пример:

```
System.out.println(uri.getScheme()); // http
```

- `getHost()` — **возвращает домен или значение** `null`. **Формат метода:**

```
public String getHost()
```

Пример:

```
System.out.println(uri.getHost()); // localhost
```

- `getPort()` — **возвращает номер порта или значение** `-1`. **Формат метода:**

```
public int getPort()
```

Пример:

```
System.out.println(uri.getPort()); // 80
```

- `getPath()` — **возвращает путь или значение** `null`. **Все escape-последовательности декодируются. Формат метода:**

```
public String getPath()
```

Пример:

```
System.out.println(uri.getPath()); // /index.php
URI uri3 = new URI("/%D1%84%D0%B0%D0%B9%D0%BB.php");
System.out.println(uri3.getPath()); // /файл.php
```

- `getRawPath()` — **возвращает путь или значение** `null`. **Формат метода:**

```
public String getRawPath()
```

Пример:

```
System.out.println(uri3.getRawPath());
// /%D1%84%D0%B0%D0%B9%D0%BB.php
```

- `getQuery()` — **возвращает строку запроса или значение** `null`. **Все escape-последовательности декодируются. Формат метода:**

```
public String getQuery()
```

Пример:

```
System.out.println(uri.getQuery()); // x=5&y=3
URI uri4 =
  new URI("/index.php?s=%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0");
System.out.println(uri4.getQuery()); // s=строка
```

- `getRawQuery()` — **возвращает строку запроса или значение** `null`. **Формат метода:**

```
public String getRawQuery()
```

Пример:

```
System.out.println(uri4.getRawQuery());
// s=%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0
```

- `getFragment()` — возвращает якорь или значение `null`. Все escape-последовательности декодируются. Формат метода:

```
public String getFragment()
```

Пример:

```
System.out.println(uri.getFragment()); // metka
URI uri5 =
   new URI("/index.php#%D0%BC%D0%B5%D1%82%D0%BA%D0%B0");
System.out.println(uri5.getFragment()); // метка
```

- `getRawFragment()` — возвращает якорь или значение `null`. Формат метода:

```
public String getRawFragment()
```

Пример:

```
System.out.println(uri5.getRawFragment());
// %D0%BC%D0%B5%D1%82%D0%BA%D0%B0
```

- `isAbsolute()` — возвращает значение `true`, если путь абсолютный (содержит название протокола), и `false` — в противном случае. Формат метода:

```
public boolean isAbsolute()
```

Пример:

```
System.out.println(uri.isAbsolute()); // true
URI uri6 = new URI("/index.php");
System.out.println(uri6.isAbsolute()); // false
```

- `normalize()` — выполняет нормализацию пути. Формат метода:

```
public URI normalize()
```

Пример:

```
URI uri = new URI("http://localhost/test/../index.php");
System.out.println(uri.normalize());
// http://localhost/index.php
```

- `resolve()` — разрешает идентификатор на основе указанного идентификатора. Форматы метода:

```
public URI resolve(URI uri)
public URI resolve(String str)
```

Пример:

```
URI uri = new URI("http://localhost/test/test/");
System.out.println(uri.resolve("index.php"));
// http://localhost/test/test/index.php
System.out.println(uri.resolve("../index.php"));
// http://localhost/test/index.php
System.out.println(uri.resolve("/index.php"));
// http://localhost/index.php
```

- `toURL()` — создает объект класса `URL` на основе текущего объекта. Формат метода:

```
public URL toURL() throws MalformedURLException
```

Пример:

```
URI uri = new URI("http://localhost/index.php");
URL url = uri.toURL();
System.out.println(url.toString());
// http://localhost/index.php
```

22.2. Класс *URL*

Класс `URL` описывает URL-адреса. Прежде чем использовать этот класс, необходимо выполнить его импорт с помощью инструкции:

```
import java.net.URL;
```

Создать экземпляр класса позволяют следующие основные конструкторы (полный их список смотрите в документации):

```
URL(String spec)    throws MalformedURLException
URL(String protocol, String host, String file)
                    throws MalformedURLException
URL(String protocol, String host, int port, String file)
                    throws MalformedURLException
URL(URL context, String spec)
                    throws MalformedURLException
```

Первый конструктор создает объект на основе строки:

```
URL url = new URL("http://localhost/index.php");
System.out.println(url.toString());
// http://localhost/index.php
```

Второй конструктор в первом параметре принимает протокол, во втором — название домена, а в третьем — путь:

```
URL url = new URL("http", "localhost", "/index.php");
System.out.println(url.toString());
// http://localhost/index.php
```

Третий конструктор позволяет дополнительно указать номер порта:

```
URL url = new URL("http", "localhost", 80, "/index.php");
System.out.println(url.toString());
// http://localhost:80/index.php
```

Четвертый конструктор разрешает путь `spec` на основе идентификатора `context`:

```
URL base_url = new URL("http://localhost/test/test/");
URL url = new URL(base_url, "index.php");
System.out.println(url.toString());
// http://localhost/test/test/index.php
```

```
url = new URL(base_url, "../index.php");
System.out.println(url.toString());
// http://localhost/test/index.php
url = new URL(base_url, "/index.php");
System.out.println(url.toString());
// http://localhost/index.php
```

22.2.1. Разбор URL-адреса

Класс `URL` содержит следующие основные методы, предназначенные для разбора URL-адреса:

- `toString()` и `toExternalForm()` — возвращают URL-адрес в виде строки. Форматы методов:

  ```
  public String toString()
  public String toExternalForm()
  ```

  Пример:

  ```
  URL url =
      new URL("http://localhost:80/index.php?x=5&y=3#metka");
  System.out.println(url.toString());
  // http://localhost:80/index.php?x=5&y=3#metka
  System.out.println(url.toExternalForm());
  // http://localhost:80/index.php?x=5&y=3#metka
  ```

- `getProtocol()` — возвращает протокол. Формат метода:

  ```
  public String getProtocol()
  ```

  Пример:

  ```
  System.out.println(url.getProtocol()); // http
  ```

- `getHost()` — возвращает домен. Формат метода:

  ```
  public String getHost()
  ```

  Пример:

  ```
  System.out.println(url.getHost()); // localhost
  ```

- `getPort()` — возвращает номер порта или значение `-1`. Формат метода:

  ```
  public int getPort()
  ```

  Пример:

  ```
  System.out.println(url.getPort()); // 80
  ```

- `getDefaultPort()` — возвращает номер порта по умолчанию для протокола или значение `-1`. Формат метода:

  ```
  public int getDefaultPort()
  ```

  Пример:

  ```
  System.out.println(url.getDefaultPort()); // 80
  ```

- `getPath()` — возвращает путь. Формат метода:

```
public String getPath()
```

Пример:

```
System.out.println(url.getPath()); // /index.php
```

- `getFile()` — возвращает путь со строкой запроса. Формат метода:

```
public String getFile()
```

Пример:

```
System.out.println(url.getFile()); // /index.php?x=5&y=3
```

- `getQuery()` — возвращает строку запроса или значение `null`. Формат метода:

```
public String getQuery()
```

Пример:

```
System.out.println(url.getQuery()); // x=5&y=3
```

- `getRef()` — возвращает якорь или значение `null`. Формат метода:

```
public String getRef()
```

Пример:

```
System.out.println(url.getRef()); // metka
```

- `getUserInfo()` — возвращает данные для авторизации или значение `null`. Формат метода:

```
public String getUserInfo()
```

Пример:

```
URL ftp =
   new URL("ftp://user:123@localhost:21/index.php");
System.out.println(ftp.getUserInfo()); // user:123
```

- `sameFile()` — сравнивает текущий объект с указанным без учета якоря. Формат метода:

```
public boolean sameFile(URL other)
```

Пример:

```
URL url1 =
    new URL("http://localhost/index.php#metka");
URL url2 =
    new URL("http://localhost/index.php");
System.out.println(url1.sameFile(url2)); // true
System.out.println(url1.equals(url2));   // false
```

- `toURI()` — создает объект класса `URI` на основе текущего объекта. Формат метода:

```
public URI toURI() throws URISyntaxException
```

Пример:

```
URL url = new URL("http://localhost/index.php");
URI uri = url.toURI();
System.out.println(uri.toASCIIString());
// http://localhost/index.php
```

22.2.2. Кодирование и декодирование строки запроса

Методы класса `URL` не производят никакого кодирования и декодирования. Для этого необходимо использовать класс `URI`:

```
URL url = new URL("http://localhost/?s=строка");
System.out.println(url.toString());
// http://localhost/?s=строка
URI uri = url.toURI();
System.out.println(uri.toASCIIString());
// http://localhost/?s=%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0
```

Для кодирования можно применить статический метод `encode()` из класса `URLEncoder`. Форматы метода:

```
import java.net.URLEncoder;
public static String encode(String s, String enc)
                    throws UnsupportedEncodingException
public static String encode(String s, Charset charset)
```

В первом параметре указывается строка, подлежащая кодированию, а во втором параметре — кодировка в виде строки или объекта класса `Charset` (начиная с Java 10):

```
System.out.println(URLEncoder.encode("строка", "utf-8"));
// %D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0
// import java.nio.charset.StandardCharsets;
System.out.println(URLEncoder.encode("строка",
                                     StandardCharsets.UTF_8));
// %D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0
```

Декодировать escape-последовательности позволяет статический метод `decode()` из класса `URLDecoder`. Форматы метода:

```
import java.net.URLDecoder;
public static String decode(String s, String enc)
                    throws UnsupportedEncodingException
public static String decode(String s, Charset charset)
```

Второй формат доступен, начиная с Java 10:

```
System.out.println(
   URLDecoder.decode("%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0", "utf-8"));
// строка
// import java.nio.charset.StandardCharsets;
```

```
System.out.println(
   URLDecoder.decode("%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0",
                     StandardCharsets.UTF_8));
// строка
```

22.2.3. Получение данных из сети Интернет

Класс `URL` позволяет не только разобрать URL-адрес на составляющие, но и выполнить подключение к ресурсу в Интернете. Для этого предназначены следующие методы:

- `openConnection()` — открывает соединение и возвращает объект, с помощью которого можно выполнить запрос и получить результат. Форматы метода:

```
public URLConnection openConnection() throws IOException
public URLConnection openConnection(Proxy proxy)
                                    throws IOException
```

- `openStream()` — открывает соединение и сразу вызывает метод `getInputStream()` объекта соединения. Формат метода:

```
public final InputStream openStream() throws IOException
```

- `getContent()` — открывает соединение и сразу вызывает метод `getContent()` объекта соединения. Форматы метода:

```
public final Object getContent() throws IOException
public final Object getContent(Class<?>[] classes)
                               throws IOException
```

Давайте попробуем получить текст приветствия, который выводится с помощью файла index.php (см. листинг 22.1). Предварительно запускаем Web-сервер Apache, а затем выполняем код из листинга 22.2.

Листинг 22.2. Получение данных из сети Интернет

```
package com.example.app;

import java.io.*;
import java.net.*;

public class MyClass {
   public static void main(String[] args) throws Exception {
      URL url = new URL("http://localhost/");
      URLConnection con = url.openConnection();
      try (
         Reader reader = new InputStreamReader(
                         con.getInputStream(), "utf-8");
         BufferedReader buf = new BufferedReader(reader);
      )
```

```
        {
            String line = "";
            while ((line = buf.readLine()) != null) {
                System.out.println(line);
            }
        }
    }
}
```

В результате в окне консоли должно отобразиться приветствие: **Привет, мир!**

22.3. Классы *URLConnection* и *HttpURLConnection*

Класс `URLConnection` описывает объект соединения. Прежде чем использовать этот класс, необходимо выполнить его импорт с помощью инструкции:

```
import java.net.URLConnection;
```

Создать объект позволяет метод `openConnection()` из класса `URL`:

```
URL url = new URL("http://localhost/");
URLConnection con = url.openConnection();
```

Класс `URLConnection` является абстрактным, и в результате этого действия мы получим экземпляр вспомогательного класса, в зависимости от используемого протокола: для HTTP — `HttpURLConnection`, для HTTPS — `HttpsURLConnectionImpl`, и т. д.:

```
System.out.println(con.getClass());
// class sun.net.www.protocol.http.HttpURLConnection
```

Класс `HttpURLConnection` является наследником класса `URLConnection` и наследует все методы из этого класса, а также добавляет несколько дополнительных методов. Чтобы использовать методы из класса `HttpURLConnection`, необходимо выполнить приведение типов:

```
// import java.net.HttpURLConnection;
URL url = new URL("http://localhost/");
HttpURLConnection con = (HttpURLConnection)url.openConnection();
```

22.3.1. Установка тайм-аута

Если ресурс в то или иное время недоступен или перегружен запросами, то поток, в котором открыто соединение, будет заблокирован до момента получения ответа. По этой причине необходимо, во-первых, открывать соединение в отдельном потоке (особенно при использовании оконных приложений), а во-вторых, ограничить время ожидания (установить тайм-аут для соединения и чтения). Для управления тайм-аутом предназначены следующие методы:

- `setConnectTimeout()` — задает тайм-аут для соединения (в миллисекундах). По истечении времени генерируется исключение. Формат метода:

  ```
  public void setConnectTimeout(int timeout)
  ```

- `getConnectTimeout()` — возвращает значение тайм-аута для соединения. Если ограничение не установлено, то метод вернет значение `0`. Формат метода:

  ```
  public int getConnectTimeout()
  ```

- `setReadTimeout()` — задает тайм-аут для чтения (в миллисекундах). По истечении времени генерируется исключение `SocketTimeoutException`. Формат метода:

  ```
  public void setReadTimeout(int timeout)
  ```

- `getReadTimeout()` — возвращает значение тайм-аута для чтения. Если ограничение не установлено, то метод вернет значение `0`. Формат метода:

  ```
  public int getReadTimeout()
  ```

Пример установки тайм-аута в пять секунд:

```
URL url = new URL("http://localhost/");
URLConnection con = url.openConnection();
con.setConnectTimeout(5000);
con.setReadTimeout(5000);
```

22.3.2. Получение заголовков ответа сервера

Для получения заголовков ответа сервера предназначены следующие основные методы:

- `getHeaderFields()` — возвращает словарь со всеми заголовками. Формат метода:

  ```
  public Map<String, List<String>> getHeaderFields()
  ```

  Пример:

  ```
  URL url = new URL("http://localhost/");
  URLConnection con = url.openConnection();
  Map<String, List<String>> map = con.getHeaderFields();
  for (String key: map.keySet()) {
     System.out.println(key + ": " + map.get(key).toString());
  }
  ```

  Примерный результат:

  ```
  Keep-Alive: [timeout=5, max=100]
  null: [HTTP/1.1 200 OK]
  Server: [Apache/2.4.29 (Win32) OpenSSL/1.1.0g PHP/7.2.0]
  Connection: [Keep-Alive]
  Content-Length: [21]
  Content-Language: [ru]
  Date: [Mon, 09 Apr 2018 04:18:40 GMT]
  Content-Type: [text/html; charset=UTF-8]
  X-Powered-By: [PHP/7.2.0]
  ```

- `getHeaderFieldKey()` — **возвращает название заголовка по индексу (нумерация с 0). Если индекс не существует, то возвращается значение** `null`. **Формат метода:**

```
public String getHeaderFieldKey(int n)
```

- `getHeaderField()` — **возвращает значение заголовка по индексу (нумерация с 0) или по ключу. Если индекс (или ключ) не существует, то возвращается значение** `null`. **Форматы метода:**

```
public String getHeaderField(int n)
public String getHeaderField(String name)
```

Пример:

```
URL url = new URL("http://localhost/");
URLConnection con = url.openConnection();
String header = "";
int n = 0;
while (true) {
   header = con.getHeaderField(n);
   if (header == null) break;
   System.out.print(con.getHeaderFieldKey(n));
   System.out.println(": " + header);
   n++;
}
```

Примерный результат:

```
null: HTTP/1.1 200 OK
Date: Mon, 09 Apr 2018 04:22:44 GMT
Server: Apache/2.4.29 (Win32) OpenSSL/1.1.0g PHP/7.2.0
X-Powered-By: PHP/7.2.0
Content-Length: 21
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

Пример получения заголовка по ключу:

```
System.out.println(con.getHeaderField("Content-Type"));
// text/html; charset=UTF-8
```

- `getContentType()` — **возвращает значение заголовка** `Content-Type` **или значение** `null`. **Формат метода:**

```
public String getContentType()
```

Пример:

```
System.out.println(con.getContentType());
// text/html; charset=UTF-8
```

- `getContentLength()` **и** `getContentLengthLong()` — **возвращают значение заголовка** `Content-Length` **или значение** –1. **Форматы методов:**

```
public int getContentLength()
public long getContentLengthLong()
```

Пример:

```
System.out.println(con.getContentLength());     // 21
System.out.println(con.getContentLengthLong()); // 21
```

- `getDate()` — возвращает значение заголовка `Date` (в виде количества миллисекунд, прошедших с 1 января 1970 года) или значение `0`. Формат метода:

```
public long getDate()
```

Пример:

```
System.out.println(new Date(con.getDate()));
// Mon Apr 09 07:27:14 MSK 2018
```

- `getExpiration()` — возвращает значение заголовка `Expires` (в виде количества миллисекунд, прошедших с 1 января 1970 года) или значение `0`. Формат метода:

```
public long getExpiration()
```

- `getLastModified()` — возвращает значение заголовка `Last-Modified` (в виде количества миллисекунд, прошедших с 1 января 1970 года) или значение `0`. Формат метода:

```
public long getLastModified()
```

В классе `HttpURLConnection` определено несколько дополнительных методов:

- `getResponseCode()` — возвращает код статуса запроса. Формат метода:

```
public int getResponseCode() throws IOException
```

Пример:

```
// import java.net.HttpURLConnection;
URL url = new URL("http://localhost/");
HttpURLConnection con = (HttpURLConnection)url.openConnection();
System.out.println(con.getResponseCode()); // 200
```

- `getResponseMessage()` — возвращает текст статуса запроса. Формат метода:

```
public String getResponseMessage() throws IOException
```

Пример:

```
System.out.println(con.getResponseMessage()); // OK
```

22.3.3. Отправка заголовков запроса

Для формирования и отправки заголовков запроса предназначены следующие методы:

- `setRequestProperty()` — задает заголовки запроса. Формат метода:

```
public void setRequestProperty(String key, String value)
```

Параметр `key` задает имя заголовка, а параметр `value` — его значение. Пример формирования и отправки заголовков запроса:

```
URL url = new URL("http://localhost/");
HttpURLConnection con = (HttpURLConnection)url.openConnection();
con.setRequestProperty("User-Agent", "MySpider/1.0");
con.setRequestProperty("Accept", "text/html, text/plain");
con.setRequestProperty("Accept-Language", "ru, ru-RU");
con.setRequestProperty("Referer", "/index.php");
con.connect();     // Отправляем запрос
// ... читаем ответ
con.disconnect(); // Закрываем соединение
```

Обратите внимание: если мы не формируем заголовки запроса, то используются значения заголовков по умолчанию. Например, значением заголовка `User-Agent` будет `"Java/10"`;

- `getRequestProperties()` — возвращает словарь со всеми установленными заголовками запроса. Использовать этот метод можно только до передачи запроса (до вызова метода `connect()`), в противном случае генерируется исключение. Формат метода:

  ```
  public Map<String, List<String>> getRequestProperties()
  ```

- `getRequestProperty()` — возвращает значение заголовка по его имени или значение `null`, если заголовок не установлен. Формат метода:

  ```
  public String getRequestProperty(String key)
  ```

  Пример:

  ```
  System.out.println(con.getRequestProperty("User-Agent"));
  ```

22.3.4. Отправка запроса методом *GET*

Отправка запроса возможна различными методами — например: `GET`, `POST`, `HEAD` и др. Для указания способа передачи предназначен метод `setRequestMethod()` из класса `HttpURLConnection`. Формат метода:

```
import java.net.HttpURLConnection;
public void setRequestMethod(String method) throws ProtocolException
```

Определить способ передачи позволяет метод `getRequestMethod()`. Формат метода:

```
public String getRequestMethod()
```

При использовании метода `GET` передаваемые данные указываются в составе URL-адреса после вопросительного знака. Формат данных:

```
<Имя1>=<Значение1>&...&<ИмяN>=<ЗначениеN>
```

Все специальные символы должны быть закодированы. Для кодирования можно использовать статический метод `encode()` из класса `URLEncoder` или методы класса `URI`.

Для получения ответа сервера используется метод `getInputStream()`, который возвращает объект потока ввода `InputStream`. Формат метода:

```
public InputStream getInputStream() throws IOException
```

Давайте изменим содержимое файла C:\xampp\htdocs\index.php следующим образом:

```
<?php
print_r($_GET);
?>
```

Теперь выполним запрос методом `GET` с указанием заголовков и обработаем результат, если код ответа сервера от `200` до `300` (не включая этот код) (листинг 22.3).

Листинг 22.3. Отправка запроса методом GET

```
package com.example.app;

import java.io.*;
import java.net.*;

public class MyClass {
   public static void main(String[] args) throws Exception {
      String s = "http://localhost/index.php?color1=" +
         URLEncoder.encode("красный", "utf-8") + "&color2=" +
         URLEncoder.encode("белый", "utf-8");
      URL url = new URL(s);
      HttpURLConnection con = (HttpURLConnection)url.openConnection();
      con.setConnectTimeout(5000);
      con.setReadTimeout(5000);
      con.setDoInput(true);
      con.setRequestMethod("GET");
      con.setRequestProperty("User-Agent", "MySpider/1.0");
      con.setRequestProperty("Accept", "text/html, text/plain");
      con.setRequestProperty("Accept-Language", "ru, ru-RU");
      con.connect();
      if (con.getResponseCode() >= 200 && con.getResponseCode() < 300) {
         try (
            Reader reader = new InputStreamReader(
                           con.getInputStream(), "utf-8");
            BufferedReader buf = new BufferedReader(reader);
         )
         {
            String line = "";
            while ((line = buf.readLine()) != null) {
               System.out.println(line);
            }
         }
      }
```

```
        else {
            System.out.println("Код: " + con.getResponseCode());
        }
        if (con != null) con.disconnect();
    }
}
```

Если все выполнено правильно, то получим следующий результат:

```
Array
(
    [color1] => красный
    [color2] => белый
)
```

22.3.5. Отправка запроса методом *POST*

При передаче данных методом `POST` необходимо выполнить дополнительные действия:

- вызвать метод `setDoOutput()` и передать ему значение `true`:

```
con.setDoOutput(true);
```

- вызвать метод `setRequestMethod()` и передать ему значение `"POST"`:

```
con.setRequestMethod("POST");
```

- закодировать данные и преобразовать их в массив байтов:

```
String s = "color1=" +
    URLEncoder.encode("красный", "utf-8") + "&color2=" +
    URLEncoder.encode("белый", "utf-8");
byte[] bytes = s.getBytes();
```

- добавить заголовки `Content-Type` (значение: `application/x-www-form-urlencoded`) и `Content-Length` (длина массива байтов):

```
con.setRequestProperty("Content-Type",
                       "application/x-www-form-urlencoded");
con.setRequestProperty("Content-Length",
                       String.valueOf(bytes.length));
```

- с помощью метода `getOutputStream()` получить ссылку на поток вывода `OutputStream` и записать данные в поток. Формат метода:

```
public OutputStream getOutputStream() throws IOException
```

Давайте изменим содержимое файла C:\xampp\htdocs\index.php следующим образом:

```
<?php
print_r($_POST);
?>
```

Теперь выполним запрос методом POST с указанием заголовков и обработаем результат (листинг 22.4).

Листинг 22.4. Отправка запроса методом POST

```
package com.example.app;

import java.io.*;
import java.net.*;

public class MyClass {
   public static void main(String[] args) throws Exception {
      String s = "color1=" +
         URLEncoder.encode("красный", "utf-8") + "&color2=" +
         URLEncoder.encode("белый", "utf-8");
      byte[] bytes = s.getBytes();
      URL url = new URL("http://localhost/index.php");
      HttpURLConnection con = (HttpURLConnection)url.openConnection();
      con.setConnectTimeout(5000);
      con.setReadTimeout(5000);
      con.setDoInput(true);
      con.setDoOutput(true);
      con.setRequestMethod("POST");
      con.setRequestProperty("User-Agent", "MySpider/1.0");
      con.setRequestProperty("Accept", "text/html, text/plain");
      con.setRequestProperty("Accept-Language", "ru, ru-RU");
      con.setRequestProperty("Content-Type",
                             "application/x-www-form-urlencoded");
      con.setRequestProperty("Content-Length",
                             String.valueOf(bytes.length));
      try (OutputStream out = con.getOutputStream()) {
         out.write(bytes);
      }
      con.connect();
      if (con.getResponseCode() >= 200 && con.getResponseCode() < 300) {
         try (
            Reader reader = new InputStreamReader(
                            con.getInputStream(), "utf-8");
            BufferedReader buf = new BufferedReader(reader);
         )
         {
            String line = "";
            while ((line = buf.readLine()) != null) {
               System.out.println(line);
            }
         }
      }
```

```
      else {
         System.out.println("Код: " + con.getResponseCode());
      }
      if (con != null) con.disconnect();
   }
}
```

Если все выполнено правильно, то получим следующий результат:

```
Array
(
    [color1] => красный
    [color2] => белый
)
```

22.3.6. Отправка файла методом *POST*

Отправка файла также осуществляется методом `POST`. Процесс отправки выглядит следующим образом:

```
POST /index.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64; rv:32.0)
Accept: text/html, application/xml;q=0.9,*/*;q=0.8
Content-Type: multipart/form-data; boundary=метка
Content-Length: длина запроса

--метка
Content-Disposition: form-data; name="имя поля"; filename="имя файла"
Content-Type: text/plain

Текст файла
--метка--
```

В заголовке `Content-Type` указывается значение `multipart/form-data`. Параметр `boundary` задает текстовую метку, которая будет отмечать начало и конец данных. В теле сообщения перед меткой указываются два символа тире, а после метки — символ перевода строки. Если метка является последней, то после нее указываются два символа тире. Между метками размещаются строки с заголовками `Content-Disposition` и `Content-Type`, а после пустой строки идет содержимое файла. Таких блоков, ограниченных метками, может быть несколько, следовательно, за один раз можно отправить сразу несколько файлов или других данных.

Чтобы можно было потренироваться и наглядно увидеть процесс загрузки файла, изменим содержимое файла C:\xampp\htdocs\index.php следующим образом:

```
<?php
if (isset($_FILES['file_name'])) {
   if ($_FILES['file_name']['error'] == 0 &&
       $_FILES['file_name']['size'] > 0) {
      $path = "C:\\xampp\\htdocs\\test_file.txt";
```

```
        if (@move_uploaded_file($_FILES['file_name']['tmp_name'], $path)) {
            echo "Файл успешно сохранен. Путь " . $path;
        }
        else {
            echo "Не удалось переместить файл";
        }
    }
    else {
        echo "Ошибка №" . $_FILES['file_name']['error'];
    }
}
else {
    echo "Нет файла";
}
?>
```

Теперь отправим этому скрипту файл (листинг 22.5).

Листинг 22.5. Отправка файла

```
package com.example.app;

import java.io.*;
import java.net.*;
import java.nio.file.*;

public class MyClass {
    public static void main(String[] args) throws Exception {
        Path p = Paths.get("C:\\book\\utf8.txt");
        byte[] bytes = Files.readAllBytes(p);
        URL url = new URL("http://localhost/index.php");
        HttpURLConnection con = (HttpURLConnection)url.openConnection();
        con.setConnectTimeout(5000);
        con.setReadTimeout(5000);
        con.setDoInput(true);
        con.setDoOutput(true);
        con.setRequestMethod("POST");
        con.setRequestProperty("User-Agent", "MySpider/1.0");
        con.setRequestProperty("Accept", "text/html, text/plain");
        con.setRequestProperty("Accept-Language", "ru, ru-RU");
        con.setRequestProperty("Content-Type",
        "multipart/form-data; boundary=---------------11487310131944");
        try (PrintStream out = new PrintStream(con.getOutputStream())) {
            out.print("-----------------11487310131944\n");
            out.print("Content-Disposition: form-data; " +
                      "name=\"file_name\"; filename=\"test.txt\"\n");
            out.print("Content-Type: text/plain\n\n");
            out.write(bytes);
            out.print("\n-----------------11487310131944--");
        }
```

```
      con.connect();
      if (con.getResponseCode() >= 200 && con.getResponseCode() < 300) {
         try (
            Reader reader = new InputStreamReader(
                           con.getInputStream(), "utf-8");
            BufferedReader buf = new BufferedReader(reader);
         )
         {
            String line = "";
            while ((line = buf.readLine()) != null) {
               System.out.println(line);
            }
         }
      }
      else {
         System.out.println("Код: " + con.getResponseCode());
      }
      if (con != null) con.disconnect();
   }
}
```

Если все сделано правильно, то отобразится сообщение:

```
Файл успешно сохранен. Путь C:\xampp\htdocs\test_file.txt
```

22.3.7. Обработка перенаправлений

Если на запрос сервер отвечает кодом 301, 302 и т. д., то такое перенаправление будет автоматически обработано, и мы получим страницу, указанную в заголовке Location. Например, сервер отвечает следующим образом:

```
HTTP/1.1 302 Found
Date: Mon, 09 Apr 2018 04:18:40 GMT
Server: Apache/2.4.29 (Win32) OpenSSL/1.1.0g PHP/7.2.0
Location: http://localhost/test.php
Content-Length: 0
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
```

В результате мы автоматически получим содержимое страницы test.php.

Для управления перенаправлением используются следующие методы из класса HttpURLConnection:

- setFollowRedirects() — разрешает (значение true) или запрещает (значение false) автоматическую обработку перенаправления. Формат метода:

  ```
  public static void setFollowRedirects(boolean set)
  ```

  Пример запрета автоматического перенаправления:

  ```
  HttpURLConnection.setFollowRedirects(false);
  ```

- `getFollowRedirects()` — возвращает значение `true`, если автоматическая обработка перенаправления разрешена, и `false` — если запрещена. Формат метода:

  ```
  public static boolean getFollowRedirects()
  ```

- `setInstanceFollowRedirects()` — разрешает (значение `true`) или запрещает (значение `false`) автоматическую обработку перенаправления для текущего соединения. Формат метода:

  ```
  public void setInstanceFollowRedirects(boolean set)
  ```

  Пример запрета автоматического перенаправления для текущего соединения:

  ```
  HttpURLConnection con = (HttpURLConnection)url.openConnection();
  con.setInstanceFollowRedirects(false);
  ```

- `getInstanceFollowRedirects()` — возвращает значение `true`, если автоматическая обработка перенаправления для текущего соединения разрешена, и `false` — если запрещена. Формат метода:

  ```
  public boolean getInstanceFollowRedirects()
  ```

22.3.8. Обработка кодов ошибок

Если на запрос сервер отвечает кодом ошибки `403`, `404` и т. д., то метод `getInputStream()` будет генерировать исключение. Получить текст сообщения об ошибке в этом случае позволяет метод `getErrorStream()`.

Формат метода:

```
public InputStream getErrorStream()
```

Пример:

```
if (con.getResponseCode() == 404) {
   try (
      Reader reader = new InputStreamReader(
                          con.getErrorStream(), "utf-8");
      BufferedReader buf = new BufferedReader(reader);
   )
   {
      String line = "";
      while ((line = buf.readLine()) != null) {
         System.out.println(line);
      }
   }
}
```

Классы `URLConnection` и `HttpURLConnection`, которые мы рассмотрели в этой главе, являются лишь удобными надстройками над низкоуровневыми операциями, выполняемыми с помощью *сокетов*. Для работы с сокетами в языке Java предназначен класс `Socket` из пакета `java.net`. С помощью этого класса можно организовать операции обмена данными практически по любому протоколу. За подробной информацией о работе с сокетами обращайтесь к документации.

ГЛАВА 23

Работа с базой данных MySQL

В предыдущей главе мы установили пакет XAMPP, который включает сервер MySQL и программу для управления базами данных MySQL — phpMyAdmin. Программа phpMyAdmin написана на языке PHP, поэтому, если вы хотите администрировать базы данных, вместе с сервером MySQL необходимо запустить и Web-сервер Apache. Для этого открываем **XAMPP Control Panel** (файл C:\xampp\xampp-control.exe) и щелкаем на кнопках **Start** у пунктов **Apache** и **MySQL**. Далее запускаем Web-браузер и в адресной строке вводим: `http://localhost/phpmyadmin/`. В итоге должна отобразиться стартовая страница программы phpMyAdmin.

23.1. Установка JDBC-драйвера

Чтобы получить доступ к базе данных MySQL из Java-программы, нам необходимо установить JDBC-драйвер. Для этого переходим на страницу **http://dev.mysql.com/downloads/connector/j/** и скачиваем архив с драйвером. В моем случае он называется mysql-connector-java-5.1.46.zip. Распаковываем архив и копируем файл mysql-connector-java-5.1.46-bin.jar, например, в папку C:\book.

Теперь попробуем подключить этот архив к нашему проекту в программе Eclipse. Для этого в меню **Project** выбираем пункт **Properties**. В открывшемся окне (рис. 23.1) слева в списке выделяем пункт **Java Build Path** и справа переходим на вкладку **Libraries**. Выделяем пункт **Classpath** и нажимаем кнопку **Add External JARs**. Находим файл с драйвером (по адресу C:\book\mysql-connector-java-5.1.46-bin.jar) и нажимаем кнопку **Открыть**. В итоге JAR-архив с драйвером будет добавлен в список. Нажимаем кнопку **Apply and Close** для сохранения настроек. Для проверки правильности установки драйвера набираем код из листинга 23.1 и запускаем программу. После запуска программы должна отобразиться надпись: **Подключение успешно выполнено.**

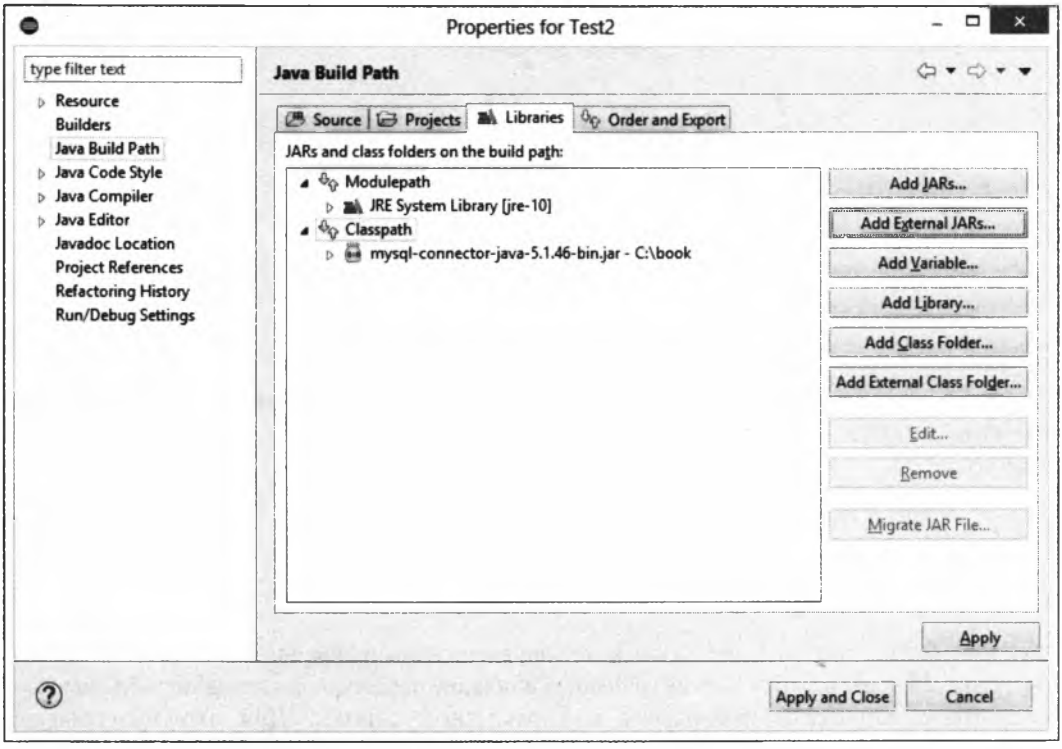

Рис. 23.1. Подключение JAR-архива с JDBC-драйвером к проекту

Листинг 23.1. Проверка правильности установки JDBC-драйвера

```
package com.example.app;

import java.sql.*;

public class MyClass {
   public static void main(String[] args) {
      Connection con = MyClass.connect("test");
      System.out.println("Подключение успешно выполнено");
      try {
         if (con != null) con.close();
      } catch (SQLException e) { }
   }

   public static Connection connect(String db) {
      Connection con = null;
      try {
         Class.forName("com.mysql.jdbc.Driver");
         con = DriverManager.getConnection(
               "jdbc:mysql://localhost:3306/" + db +
               "?characterEncoding=utf8", "root", "");
      }
```

```
        catch (ClassNotFoundException e) {
            System.out.println("Драйвер не найден");
            System.exit(1);
        }
        catch (SQLException e) {
            System.out.println("Ошибка: " + e.getMessage());
            System.exit(1);
        }
        return con;
    }
}
```

Пакет `java.sql`, классы из которого мы используем в программе, находится в одноименном модуле, поэтому в файле module-info нужно прописать зависимость от этого модуля:

```
module com.example.mymodule {
    exports com.example.app;
    requires java.sql;
}
```

Сохраняем этот код в файл C:\book\src\com.example.mymodule\module-info.java, а код из листинга 23.1 — в файл C:\book\src\com.example.mymodule\com\example\app\MyClass.java, и пробуем запустить программу из командной строки. Для этого открываем командную строку и набираем следующие команды:

```
C:\Users\Unicross>cd C:\book

C:\book>javac -encoding utf-8 -d /book/mods/com.example.mymodule
              /book/src/com.example.mymodule/module-info.java
              /book/src/com.example.mymodule/com/example/app/*.java

C:\book>java --module-path /book/mods
             -cp /book/mysql-connector-java-5.1.46-bin.jar
             --module com.example.mymodule/com.example.app.MyClass
Подключение успешно выполнено

C:\book>
```

Как видно из примера, при компиляции программ ничего дополнительно указывать не нужно, а вот при их выполнении необходимо добавить JAR-архив с драйвером в путь поиска классов.

23.2. Регистрация JDBC-драйвера и подключение к базе данных

Итак, прежде чем подключаться к базе данных, необходимо зарегистрировать JDBC-драйвер. Для этого используется метод `forName()` из класса `Class`, который в качестве параметра принимает путь к классу драйвера:

```
Class.forName("com.mysql.jdbc.Driver");
```

В результате выполнения этой инструкции загружается класс драйвера и внутри статического инициализационного блока осуществляется регистрация драйвера с помощью следующего кода:

```
java.sql.DriverManager.registerDriver(new Driver());
```

Если класс драйвера не удалось загрузить, то генерируется исключение `ClassNotFoundException`.

После регистрации драйвера можно произвести подключение к базе данных. Для этого используется статический метод `getConnection()` из класса `DriverManager`. Форматы метода:

```
import java.sql.DriverManager;
import java.sql.Connection;
public static Connection getConnection(String url)
                                        throws SQLException
public static Connection getConnection(String url, String user,
                        String password) throws SQLException
public static Connection getConnection(String url, Properties info)
                                        throws SQLException
```

В параметре `url` указывается строка подключения к базе данных, например:

```
jdbc:mysql://localhost:3306/test?characterEncoding=utf8
```

С помощью этой строки производится подключение к базе данных `test`, расположенной по адресу `localhost` (порт `3306`), и устанавливается кодировка соединения UTF-8. После знака вопроса через символ `&` можно указать различные параметры. Полный список параметров смотрите в документации.

Параметр `user` задает имя пользователя, а параметр `password` — пароль доступа. Мы будем работать с пользователем `root`, который по умолчанию не имеет пароля.

23.3. Создание базы данных

Для выполнения запросов к базе данных необходимо создать объект `Statement` с помощью метода `createStatement()` из интерфейса `Connection`. Формат метода:

```
import java.sql.Statement;
public Statement createStatement() throws SQLException
```

Выполнить SQL-запрос на создание базы данных позволяет метод `executeUpdate()` из интерфейса `Statement`. Формат метода:

```
public int executeUpdate(String sql) throws SQLException
```

Давайте создадим базу данных `catalog` (листинг 23.2).

Листинг 23.2. Создание базы данных

```
public static void main(String[] args) {
   Connection con = MyClass.connect("test");
   Statement stmt = null;
```

```
    try {
       stmt = con.createStatement();
       stmt.executeUpdate("CREATE DATABASE `catalog` " +
                          "DEFAULT CHARACTER SET utf8" +
                          " COLLATE utf8_general_ci");
       System.out.println("База данных создана");
    } catch (SQLException e) {
       System.out.println("Ошибка: " + e.getMessage());
    }
    finally {
       try { if (stmt != null) stmt.close(); }
       catch (SQLException e) {}
       try { if (con != null)  con.close(); }
       catch (SQLException e) {}
       stmt = null; con = null;
    }
}
```

23.4. Создание таблицы

Для создания таблицы можно также воспользоваться методом `executeUpdate()` из интерфейса `Statement`. Добавим в базу данных `catalog` три таблицы (листинг 23.3).

Листинг 23.3. Создание таблиц

```
public static void main(String[] args) {
   Connection con = MyClass.connect("catalog");
   Statement stmt = null;
   try {
      stmt = con.createStatement();
      stmt.executeUpdate("CREATE TABLE `user` ("
         + "`id_user` mediumint(9) auto_increment, "
         + "`email` char(50), "
         + "`passw` char(32), "
         + "PRIMARY KEY (`id_user`), "
         + "UNIQUE KEY (`email`) "
         + ") ENGINE=InnoDB DEFAULT CHARSET=utf8");
      stmt.executeUpdate("CREATE TABLE `rubr` ("
         + "`id_rubr` mediumint(6) auto_increment, "
         + "`name_rubr` char(150), "
         + "PRIMARY KEY (`id_rubr`) "
         + ") ENGINE=InnoDB DEFAULT CHARSET=utf8");
      stmt.executeUpdate("CREATE TABLE `site` ("
         + "`id_site` mediumint(9) auto_increment, "
         + "`id_user` mediumint(9), "
         + "`id_rubr` mediumint(6), "
         + "`url` char(255), "
```

```
            + "`title` char(80), "
            + "`msg` text, "
            + "`iq` tinyint, "
            + "PRIMARY KEY (`id_site`), "
            + "KEY (`id_rubr`) "
            + ") ENGINE=InnoDB DEFAULT CHARSET=utf8");
         System.out.println("Таблицы созданы");
      } catch (SQLException e) {
         System.out.println("Ошибка: " + e.getMessage());
      }
      finally {
         try { if (stmt != null) stmt.close(); }
         catch (SQLException e) {}
         try { if (con != null)  con.close(); }
         catch (SQLException e) {}
         stmt = null; con = null;
      }
   }
```

Обратите внимание: при создании соединения мы указали базу данных `catalog` вместо `test`. Если бы мы этого не сделали, то все таблицы создались бы в базе данных `test`.

23.5. Добавление записей

Добавить записи в таблицу можно несколькими способами. Начнем с уже знакомого нам метода `executeUpdate()` из интерфейса `Statement`. Добавим пользователя в таблицу `user`:

```
stmt.executeUpdate("INSERT INTO `user` (`email`, `passw`)"
                 + " VALUES ('userl@mail.ru', 'password')");
```

Если нам необходимо узнать, какой индекс был автоматически сгенерирован при добавлении записи (поле `id_user` в таблице `user` обозначено как `auto_increment`), то во втором параметре метода `executeUpdate()` следует указать значение `Statement.RETURN_GENERATED_KEYS`. Формат метода:

```
public int executeUpdate(String sql, int autoGeneratedKeys)
                                        throws SQLException
```

Чтобы получить индекс, необходимо после выполнения запроса вызвать метод `getGeneratedKeys()` и получить объект `ResultSet`. Формат метода:

```
import java.sql.ResultSet;
public ResultSet getGeneratedKeys() throws SQLException
```

Объект `ResultSet` содержит метод `next()`, который перемещает указатель внутри результирующего набора и возвращает значение `true`, если доступен результат, в противном случае — значение `false`. Формат метода:

```
public boolean next() throws SQLException
```

Если метод `next()` вернул значение `true`, то можно получить индекс с помощью метода `getInt()` из интерфейса `ResultSet`. В качестве параметра метод принимает индекс столбца (нумерация начинается с `1`). Формат метода:

```
public int getInt(int columnIndex) throws SQLException
```

Давайте добавим еще одного пользователя и получим сгенерированный индекс (листинг 23.4).

Листинг 23.4. Определение сгенерированного индекса

```
public static void main(String[] args) {
   Connection con = MyClass.connect("catalog");
   Statement stmt = null;
   try {
      stmt = con.createStatement();
      stmt.executeUpdate("INSERT INTO `user` (`email`, `passw`)"
         + " VALUES ('user2@mail.ru', 'пароль')",
         Statement.RETURN_GENERATED_KEYS);
      int index = -1;
      ResultSet rs = stmt.getGeneratedKeys();
      if (rs.next()) index = rs.getInt(1);
      System.out.println("Индекс новой записи: " + index);
      if (rs != null) rs.close();
   } catch (SQLException e) {
      System.out.println("Ошибка: " + e.getMessage());
   }
   finally {
      try { if (stmt != null) stmt.close(); }
      catch (SQLException e) {}
      try { if (con != null)  con.close(); }
      catch (SQLException e) {}
      stmt = null; con = null;
   }
}
```

В некоторых случаях в SQL-запрос необходимо подставлять данные, полученные от пользователя. Если эти данные не обработать и подставить в SQL-запрос как есть, то пользователь получит возможность видоизменить запрос и, например, зайти в закрытый раздел без ввода пароля. Чтобы предотвратить такой несанкционированный доступ, значения должны быть подставлены правильно, для чего необходимо использовать *подготовленные запросы*. Для создания подготовленного запроса предназначен метод `prepareStatement()` из интерфейса `Connection`. Формат метода:

```
import java.sql.PreparedStatement;
public PreparedStatement prepareStatement(String sql)
                                  throws SQLException
```

В качестве параметра метод принимает строку с SQL-запросом, внутри которого вместо значений следует указать символ вопроса:

```
"INSERT INTO `rubr` VALUES (?, ?)"
```

Чтобы подставить значения вместо знаков вопросов, необходимо вызвать метод из интерфейса `PreparedStatement`, который соответствует типу вставляемых данных. Например, чтобы вставить строку, следует вызвать метод `setString()`. Формат метода:

```
public void setString(int parameterIndex, String x)
                                throws SQLException
```

В первом параметре метод принимает порядковый индекс (нумерация с 1) вопросительного знака внутри подготовленного запроса, а во втором параметре — строку с данными. При этом все специальные символы внутри строки будут автоматически экранированы. После добавления всех данных следует вызвать метод `executeUpdate()`, который выполнит SQL-запрос на вставку записей.

Давайте добавим несколько рубрик в таблицу `rubr`:

```
PreparedStatement ps = con.prepareStatement(
    "INSERT INTO `rubr` (`name_rubr`) VALUES (?)");
String[] arr = {"Программирование", "Музыка",
                "Поисковые \' \" порталы", "Кино"};
for (String rubr: arr) {
   ps.setString(1, rubr);
   ps.executeUpdate();
}
```

Помимо метода `setString()`, интерфейс `PreparedStatement` содержит множество методов, предназначенных для вставки данных в различных форматах. Приведем наиболее часто используемые методы (полный список смотрите в документации):

```
public void setByte(int parameterIndex, byte x)
                                throws SQLException
public void setInt(int parameterIndex, int x)
                                throws SQLException
public void setLong(int parameterIndex, long x)
                                throws SQLException
public void setFloat(int parameterIndex, float x)
                                throws SQLException
public void setDouble(int parameterIndex, double x)
                                throws SQLException
public void setString(int parameterIndex, String x)
                                throws SQLException
public void setDate(int parameterIndex, Date x)
                                throws SQLException
public void setTime(int parameterIndex, Time x)
                                throws SQLException
public void setTimestamp(int parameterIndex, Timestamp x)
                                throws SQLException
```

Добавим два сайта в таблицу `site`:

```
PreparedStatement ps = con.prepareStatement(
   "INSERT INTO `site` (`id_user`, `id_rubr`, `url`,"
   + " `title`, `msg`, `iq`) VALUES (?, ?, ?, ?, ?, ?)");
ps.setInt(1, 1);
ps.setInt(2, 1);
ps.setString(3, "http://python.org");
ps.setString(4, "Python");
ps.setString(5, "Язык программирования Python");
ps.setByte(6, (byte)50);
ps.executeUpdate();
ps.setInt(1, 1);
ps.setInt(2, 3);
ps.setString(3, "http://google.ru");
ps.setString(4, "Гугль");
ps.setString(5, "Поисковый портал");
ps.setByte(6, (byte)80);
ps.executeUpdate();
```

23.6. Обновление и удаление записей

Для обновления и удаления записей используется уже знакомый нам метод `executeUpdate()` из интерфейса `Statement`. Можно также воспользоваться одноименным методом из интерфейса `PreparedStatement`. Метод возвращает количество измененных или удаленных записей. Изменим название рубрики с индексом `3` и удалим рубрику с индексом `4`:

```
stmt = con.createStatement();
int r = 0;
r = stmt.executeUpdate("UPDATE `rubr` "
         + "SET `name_rubr`='Поисковые порталы' "
         + "WHERE `id_rubr`=3");
System.out.println("Изменено записей: " + r);
r = stmt.executeUpdate("DELETE FROM `rubr` "
         + "WHERE `id_rubr`=4");
System.out.println("Удалено записей: " + r);
```

23.7. Получение записей

При использовании SQL-команды `SELECT` запросы выполняются с помощью метода `executeQuery()` из интерфейса `Statement`. Формат метода:

```
import java.sql.ResultSet;
public ResultSet executeQuery(String sql) throws SQLException
```

Метод `executeQuery()` возвращает объект `ResultSet`, с помощью которого можно получить результаты выполнения запроса. Интерфейс `ResultSet` содержит следующие основные методы (полный список смотрите в документации):

- `next()` — перемещает указатель на одну позицию и возвращает значение `true`, если запись доступна, или значение `false`. Формат метода:

```
public boolean next() throws SQLException
```

- `getString()` — возвращает строку по индексу поля (нумерация с `1`) или по названию поля (или псевдонима). Форматы метода:

```
public String getString(int columnIndex) throws SQLException
public String getString(String columnLabel)
                                        throws SQLException
```

Помимо метода `getString()`, интерфейс `ResultSet` содержит множество методов, предназначенных для получения данных в различных форматах. Например, `getByte()`, `getInt()`, `getLong()`, `getFloat()`, `getDouble()`, `getDate()`, `getBytes()` и др.

Для получения результата SQL-команды `SELECT` можно также воспользоваться методом `executeQuery()` из интерфейса `PreparedStatement`. Формат метода:

```
public ResultSet executeQuery() throws SQLException
```

В качестве примера получим все записи из таблицы `rubr` с сортировкой по имени рубрики (листинг 23.5). Кроме того, получим записи сразу из трех таблиц, выполнив всего один запрос.

Листинг 23.5. Получение записей

```
public static void main(String[] args) {
   Connection con = MyClass.connect("catalog");
   Statement stmt = null;
   PreparedStatement ps = null;
   try {
      stmt = con.createStatement();
      try (ResultSet rs = stmt.executeQuery(
         "SELECT * FROM `rubr` ORDER BY `name_rubr`")) {
         while (rs.next()) {
             System.out.print(rs.getInt(1) + " ");
            System.out.println(rs.getString("name_rubr"));
         }
      }
      ps = con.prepareStatement(
         "SELECT `site`.`url` AS `site_url`, "
         + "`site`.`title` AS `site_title`, "
         + "`rubr`.`name_rubr` AS `rubr_name`, "
         + "`user`.`email` AS `user_email` "
         + "FROM `site`, `rubr`, `user` "
         + "WHERE `site`.`id_rubr`=`rubr`.`id_rubr` "
         + "AND `site`.`id_user`=`user`.`id_user`");
```

```
            try (ResultSet rs = ps.executeQuery()) {
               while (rs.next()) {
                  System.out.print(rs.getString("site_url") + " ");
                  System.out.print(rs.getString("site_title") + " ");
                  System.out.print(rs.getString("rubr_name") + " ");
                  System.out.println(rs.getString("user_email"));
               }
            }
         } catch (SQLException e) {
            System.out.println("Ошибка: " + e.getMessage());
         }
         finally {
            try { if (ps != null) ps.close(); }
            catch (SQLException e) {}
            try { if (stmt != null) stmt.close(); }
            catch (SQLException e) {}
            try { if (con != null)  con.close(); }
            catch (SQLException e) {}
            stmt = null; con = null; ps = null;
         }
      }
```

Результат выполнения:

```
2 Музыка
3 Поисковые порталы
1 Программирование
http://python.org Python Программирование user1@mail.ru
http://google.ru Гугль Поисковые порталы user1@mail.ru
```

23.8. Метод *execute()*

Как вы уже знаете, для выполнения запросов, не требующих получения результирующего набора (`INSERT`, `UPDATE`, `DELETE` и др.), применяется метод `executeUpdate()`. Если нужно получить данные, то следует использовать метод `executeQuery()`, который возвращает объект `ResultSet`. Для выполнения любых запросов можно также воспользоваться методом `execute()`. Причем метод `execute()` позволяет получить множественный результат, состоящий из нескольких наборов. Формат метода:

```
public boolean execute(String sql) throws SQLException
```

Метод возвращает значение `true`, если доступен результирующий набор, и `false` — в противном случае (в этом случае может быть доступно количество изменений). Получить результат запроса позволяют следующие методы из интерфейса `Statement`:

- `getResultSet()` — возвращает объект `ResultSet`, через который можно получить результат, или значение `null`. Формат метода:

  ```
  public ResultSet getResultSet() throws SQLException
  ```

- `getUpdateCount()` — возвращает количество произведенных изменений или значение `-1`. Формат метода:

```
public int getUpdateCount() throws SQLException
```

- `getMoreResults()` — переходит к следующему результирующему набору и возвращает значение `true`, если доступен результирующий набор, и `false` — в противном случае (в этом случае может быть доступно количество изменений). Формат метода:

```
public boolean getMoreResults() throws SQLException
```

Добавим нового пользователя и выведем все записи из таблицы `user`, используя метод `execute()` (листинг 23.6).

Листинг 23.6. Метод `execute()`

```
public static void main(String[] args) {
   Connection con = MyClass.connect("catalog");
   Statement stmt = null;
   try {
      stmt = con.createStatement();
      if (!stmt.execute(
         "INSERT INTO `user` (`email`, `passw`)"
         + " VALUES ('user3@mail.ru', '123456')")) {
         System.out.println("Добавлено записей: " +
            stmt.getUpdateCount());
      }
      if (stmt.execute(
         "SELECT * FROM `user` ORDER BY `email`")) {
         try (ResultSet rs = stmt.getResultSet()) {
            while (rs.next()) {
               System.out.print(rs.getInt(1) + " ");
               System.out.println(rs.getString("email"));
            }
         }
      }
   } catch (SQLException e) {
      System.out.println("Ошибка: " + e.getMessage());
   }
   finally {
      try { if (stmt != null) stmt.close(); }
      catch (SQLException e) {}
      try { if (con != null)  con.close(); }
      catch (SQLException e) {}
      stmt = null; con = null;
   }
}
```

23.9. Получение информации о структуре набора *ResultSet*

Получить информацию о структуре результирующего набора `ResultSet` позволяет метод `getMetaData()`. Формат метода:

```
public ResultSetMetaData getMetaData() throws SQLException
```

Интерфейс `ResultSetMetaData` содержит следующие основные методы (полный список смотрите в документации):

- `getColumnCount()` — возвращает количество полей. Формат метода:

```
public int getColumnCount()              throws SQLException
```

- `getTableName()` — возвращает название таблицы. Формат метода:

```
public String getTableName(int column) throws SQLException
```

- `getColumnName()` — возвращает название поля. Формат метода:

```
public String getColumnName(int column)
                                         throws SQLException
```

- `getColumnLabel()` — возвращает псевдоним или название поля при отсутствии псевдонима. Формат метода:

```
public String getColumnLabel(int column)
                                         throws SQLException
```

- `getColumnClassName()` — возвращает название класса Java (с именем пакета), соответствующего типу данных внутри поля, — например: `java.lang.Integer`. Формат метода:

```
public String getColumnClassName(int column)
                                         throws SQLException
```

- `getColumnType()` — возвращает тип данных внутри поля (в виде значения константы из класса `java.sql.Types`). Формат метода:

```
public int getColumnType(int column)     throws SQLException
```

- `getColumnTypeName()` — возвращает название типа данных внутри поля в виде строки (например, `MEDIUMINT`). Формат метода:

```
public String getColumnTypeName(int column)
                                         throws SQLException
```

- `getPrecision()` — возвращает размер поля. Формат метода:

```
public int getPrecision(int column)      throws SQLException
```

Выведем структуру таблицы `site`:

```
stmt = con.createStatement();
if (stmt.execute("SELECT * FROM `site`")) {
   try (ResultSet rs = stmt.getResultSet()) {
      ResultSetMetaData r = rs.getMetaData();
```

```
      for (int i = 1; i <= r.getColumnCount(); i++) {
         System.out.print(r.getTableName(i) + " ; ");
         System.out.print(r.getColumnName(i) + " ; ");
         System.out.print(r.getColumnLabel(i) + " ; ");
         System.out.print(r.getColumnClassName(i) + " ; ");
         System.out.print(r.getColumnType(i) + " ; ");
         System.out.print(r.getColumnTypeName(i) + " ; ");
         System.out.println(r.getPrecision(i));
      }
   }
}
```

Результат:

```
site ; id_site ; id_site ; java.lang.Integer ; 4 ; MEDIUMINT ; 9
site ; id_user ; id_user ; java.lang.Integer ; 4 ; MEDIUMINT ; 9
site ; id_rubr ; id_rubr ; java.lang.Integer ; 4 ; MEDIUMINT ; 6
site ; url ; url ; java.lang.String ; 1 ; CHAR ; 255
site ; title ; title ; java.lang.String ; 1 ; CHAR ; 80
site ; msg ; msg ; java.lang.String ; -1 ; VARCHAR ; 21845
site ; iq ; iq ; java.lang.Integer ; -6 ; TINYINT ; 4
```

23.10. Транзакции

Очень часто несколько запросов выполняются последовательно. Например, при совершении покупки деньги списываются со счета клиента и сразу добавляются на счет магазина. Если во время добавления денег на счет магазина произойдет ошибка, то деньги будут списаны со счета клиента, но не попадут на счет магазина. Чтобы гарантировать успешное выполнение группы запросов, используется механизм *транзакций*. Запуск транзакции осуществляется методом `setAutoCommit()` из интерфейса `Connection` со значением параметра `false` (по умолчанию используется значение `true`, и каждый запрос автоматически завершает транзакцию). Формат метода:

```
public void setAutoCommit(boolean autoCommit) throws SQLException
```

После запуска транзакции группа запросов выполняется как единое целое. Если ошибки не произошло, то следует вызвать метод `commit()` из интерфейса `Connection`. Формат метода:

```
public void commit() throws SQLException
```

Метод `rollback()` из интерфейса `Connection` позволяет отменить все изменения в рамках транзакции. Обычно его вызывают внутри блока `catch` при возникновении исключения во время выполнения запросов. Форматы метода:

```
public void rollback()                      throws SQLException
public void rollback(Savepoint savepoint) throws SQLException
```

Обратите внимание!

При использовании базы данных MySQL следует учитывать, что транзакции поддерживаются только таблицами, имеющими тип `InnoDB`. Таблицы типа `MyISAM` транзакции не поддерживают.

Схема работы с транзакциями выглядит следующим образом:

```
try {
   con.setAutoCommit(false);        // Запуск транзакции
   stmt = con.createStatement();
   stmt.executeUpdate(...);         // Выполняем запросы
   stmt.executeUpdate(...);
   stmt.executeUpdate(...);
   con.commit();                    // Применяем запросы
} catch (SQLException e) {
   System.out.println("Ошибка: " + e.getMessage());
   try {
      con.rollback();               // Отменяем запросы
   }
   catch (SQLException e1) {
      System.out.println("Не удалось отменить запросы");
   }
}
```

С помощью метода `setSavepoint()` из интерфейса `Connection` можно пометить позицию, в которой начинается транзакция. Форматы метода:

```
public Savepoint setSavepoint() throws SQLException
public Savepoint setSavepoint(String name) throws SQLException
```

Чтобы при ошибке отменить все запросы после этой позиции, следует передать объект `Savepoint` в метод `rollback()`.

23.11. Получение информации об ошибках

При возникновении ошибки в процессе выполнения запроса генерируется исключение. Базовым классом для всех исключений внутри пакета `java.sql` является класс `SQLException`, наследующий класс `java.lang.Exception`. Получить информацию об ошибке позволяют следующие методы:

- `getMessage()` — возвращает описание ошибки. Формат метода:

  ```
  public String getMessage()
  ```

  Пример описания:

  ```
  Table 'catalog.test' doesn't exist
  ```

- `getErrorCode()` — возвращает код ошибки (например, `1146`). Формат метода:

  ```
  public int getErrorCode()
  ```

- `getSQLState()` — возвращает код статуса. Формат метода:

  ```
  public String getSQLState()
  ```

- `getNextException()` — возвращает объект следующего исключения или значение `null`. Формат метода:

  ```
  public SQLException getNextException()
  ```

Помимо ошибок могут быть получены и предупреждающие сообщения, описываемые классом `SQLWarning`. Этот класс является наследником класса `SQLException` и наследует все его методы. Получить объект класса `SQLWarning` позволяет метод `getWarnings()` из интерфейсов `Connection`, `Statement` и `ResultSet`. Если предупреждений нет, то метод возвращает значение `null`. Формат метода:

```
public SQLWarning getWarnings() throws SQLException
```

Получить следующее предупреждение позволяет метод `getNextWarning()`. Если предупреждений больше нет, то метод возвращает значение `null`. Формат метода:

```
public SQLWarning getNextWarning()
```

ГЛАВА 24

Многопоточные приложения

В ряде предыдущих глав мы познакомились с потоками ввода/вывода (*главы 19* и *20*) и потоками данных (*глава 21*). В этой главе мы рассмотрим еще один вид потоков — *потоки управления*. Благодаря потокам управления можно выполнять различные задачи параллельно. В случае, если процессор компьютера является одноядерным, между потоками будет осуществляться переключение, имитирующее параллельное выполнение.

Когда следует использовать потоки управления? Во-первых, при выполнении длительной операции. Например, получение данных из Интернета может блокировать основной поток, если сервер не отвечает. Во-вторых, когда необходимо ускорить выполнение операции. В этом случае операция разбивается на отдельные задачи, и эти задачи раздаются потокам. В-третьих, при использовании оконных приложений — операция не должна выполняться в потоке диспетчера обработки событий, иначе приложение не сможет выполнить перерисовку компонентов и перестанет реагировать на действия пользователя.

> ***Примечание***
>
> Не следует забывать, что потоки данных содержат механизм параллельной обработки данных, скрывающий детали запуска и управления потоками (см. *главу 21*).

24.1. Создание и прерывание потока

При запуске приложения создается главный поток управления, и в этом потоке запускается код из метода `main()`. Давайте получим название этого потока:

```
package com.example.app;

public class MyClass {
   public static void main(String[] args) {
      System.out.println(Thread.currentThread().getName()); // main
   }
}
```

Пользовательские потоки управления описываются классом `Thread` из пакета `java.lang`. Чтобы запустить поток, достаточно создать экземпляр класса `Thread`,

передав конструктору класса объект, реализующий интерфейс `Runnable`, а затем вызвать метод `start()`. Интерфейс `Runnable` содержит объявление метода `run()`:

```
public void run()
```

Обратите внимание: хотя код должен размещаться внутри метода `run()`, запускать поток необходимо вызовом метода `start()`. В этом случае метод `run()` будет выполнен в отдельном потоке. Если вызвать метод `run()` непосредственно, то код будет выполнен в текущем потоке, а не в отдельном.

В качестве примера создадим класс, реализующий интерфейс `Runnable`, и запустим четыре потока, внутри которых просто будем выводить данные в окно консоли (листинг 24.1).

Листинг 24.1. Создание потока

```
package com.example.app;

public class MyClass {
   public static void main(String[] args) {
      for (int i = 1; i < 5; i++) {
         (new Thread(new MyThread(i))).start();
      }
   }
}

class MyThread implements Runnable {
   private int id;
   MyThread(int id) {
      this.id = id;
   }
   public int getId() {
      return id;
   }
   @Override
   public void run() {
      for (int i = 0; i < 3; i++) {
         System.out.println("Поток " + getId() + " i=" + i +
            " " + Thread.currentThread().getName() +
            " " + Thread.currentThread().getId());
         try {
            Thread.sleep(1000); // Имитация выполнения задачи
         }
         catch (InterruptedException e) {}
      }
   }
}
```

В этом примере мы воспользовались следующими методами из класса `Thread`:

- `start()` — запускает поток и вызывает метод `run()`. Формат метода:

```
public void start()
```

- `sleep()` — прерывает выполнение потока на указанное время (значение параметра `millis` указывается в миллисекундах, а значение параметра `nanos` — в наносекундах в пределах `0` — `999 999`). Форматы метода:

```
public static void sleep(long millis)
                  throws InterruptedException
public static void sleep(long millis, int nanos)
                  throws InterruptedException
```

- `currentThread()` — возвращает ссылку на текущий поток. Формат метода:

```
public static Thread currentThread()
```

- `getName()` — возвращает название потока. Формат метода:

```
public final String getName()
```

- `getId()` — возвращает идентификатор потока. Формат метода:

```
public long getId()
```

Создать объект, реализующий интерфейс `Runnable`, можно также с помощью анонимного вложенного класса или лямбда-выражения:

```
(new Thread(new Runnable() {
   @Override
   public void run() {
      System.out.println("Привет из потока " +
         Thread.currentThread().getName());
   }
})).start();
(new Thread( () -> System.out.println("Привет из потока " +
                  Thread.currentThread().getName()))).start();
```

Потоки автоматически завершают свою работу после выхода из метода `run()` или при генерации исключения внутри метода `run()`. Если необходимо остановить работу потока досрочно, то следует послать сигнал, вызвав метод `interrupt()`. Формат метода:

```
public void interrupt()
```

Обратите внимание: метод не останавливает работу потока, а всего лишь сигнализирует о необходимости остановить работу. Поток должен периодически вызывать метод `isInterrupted()`, чтобы проверить наличие сигнала. Если метод вернет значение `true`, то сигнал получен. Формат метода:

```
public boolean isInterrupted()
```

Пример выполнения потока до момента поступления сигнала:

```
while (!Thread.currentThread().isInterrupted()) {
   // Что-то делаем
}
```

Для проверки наличия сигнала можно также воспользоваться статическим методом `interrupted()` из класса `Thread`. Если метод вернет значение `true`, то сигнал получен. Следует учитывать, что после получения информации о сигнале статус сигнала сбрасывается (а вот метод `isInterrupted()` статус не сбрасывает). Формат метода:

```
public static boolean interrupted()
```

Если во время посылки сигнала поток находится в режиме ожидания или прерывания (например, вызван метод `sleep()`), то генерируется исключение `InterruptedException`, информирующее о поступлении сигнала. После получения сигнала внутри потока можно решать что делать: остановить поток или продолжить выполнение. Например, нельзя останавливать работу потока, если какая-либо незавершенная операция может повредить данные. По этой причине метод `stop()`, явно прерывающий выполнение потока, объявлен устаревшим и не рекомендуется к использованию. Обратите внимание: после генерации исключения `InterruptedException` статус сигнала сбрасывается.

Создать поток можно также, создав класс, наследующий класс `Thread`. После создания экземпляра класса необходимо вызвать метод `start()`. Давайте рассмотрим это на примере, а заодно научимся посылать сигналы с помощью метода `interrupt()` (листинг 24.2).

Листинг 24.2. Создание и остановка потока

```
package com.example.app;

public class MyClass {
   public static void main(String[] args) {
      MyThread[] arr = new MyThread[4];
      for (int i = 0; i < 4; i++) {
         arr[i] = new MyThread();
         arr[i].start();
      }
      for (int i = 0; i < 4; i++) {
         try {
            Thread.sleep(2000);
         }
         catch (InterruptedException e) {}
         arr[i].interrupt();  // Посылаем сигнал
      }
   }
}
```

```
class MyThread extends Thread {
   @Override
   public void run() {
      for (int i = 0; i < 20; i++) {
         System.out.println("Поток " + getId() + " i=" + i +
            " " + getName());
         try {
            Thread.sleep(1000); // Имитация выполнения задачи
         }
         catch (InterruptedException e) {
            System.out.println("Выход. Поток " + getId());
            return;
         }
      }
   }
}
```

24.2. Потоки-демоны

По умолчанию потоки работают, пока не будет достигнут конец метода `run()` или не встретится оператор `return` внутри метода `run()`. При этом основной поток может уже прекратить работу. Если поток объявлен как *демон*, то, при завершении работы всех обычных потоков, поток-демон автоматически завершается, причем на любой стадии выполнения. Поэтому внутри потока-демона не следует работать с ресурсами (например, с файлами). Чтобы объявить поток демоном, достаточно перед запуском потока вызвать метод `setDaemon()` и передать ему значение `true`. Формат метода:

```
public final void setDaemon(boolean on)
```

Пример создания и запуска потока-демона:

```
Thread t = new Thread(new MyThread());
t.setDaemon(true);
t.start();
```

Метод `isDaemon()` возвращает значение `true`, если поток является демоном, и `false` — в противном случае. Формат метода:

```
public final boolean isDaemon()
```

24.3. Состояния потоков

Поток может пребывать в нескольких состояниях. Получить текущее состояние потока позволяет метод `getState()`. Формат метода:

```
public Thread.State getState()
```

Метод возвращает одну из констант из перечисления `Thread.State`:

- `NEW` — поток создан, но не запущен;
- `RUNNABLE` — поток является работоспособным;
- `BLOCKED` — поток блокирован;
- `WAITING` — поток ждет действий от других потоков без ограничения на время ожидания;
- `TIMED_WAITING` — поток ждет действий от других потоков до истечения тайм-аута;
- `TERMINATED` — поток завершил работу.

24.4. Приоритеты потоков

Каждый поток выполняется в соответствии с установленным приоритетом. Если существует поток с более высоким приоритетом, то выполнение потока с более низким приоритетом приостанавливается. Это означает, что поток с низким приоритетом может вообще не выполняться, ожидая завершения потоков с высоким приоритетом. Обратите внимание: в некоторых операционных системах все потоки имеют одинаковый приоритет.

Для установки приоритета предназначен метод `setPriority()`. Формат метода:

```
public final void setPriority(int newPriority)
```

В качестве значения параметра `newPriority` можно указать число от `1` до `10` или одно из значений: `MIN_PRIORITY`, `NORM_PRIORITY` или `MAX_PRIORITY`:

```
System.out.println(Thread.MIN_PRIORITY);  // 1
System.out.println(Thread.NORM_PRIORITY); // 5
System.out.println(Thread.MAX_PRIORITY);  // 10
```

Для получения приоритета потока предназначен метод `getPriority()`. Формат метода:

```
public final int getPriority()
```

24.5. Метод *join()*

Метод `join()` из класса `Thread` используется для приостановления выполнения текущего потока до момента, пока не закончит выполнение другой поток или не истечет время тайм-аута. Форматы метода:

```
public final void join()                throws InterruptedException
public final void join(long millis)     throws InterruptedException
public final void join(long millis, int nanos)
                                        throws InterruptedException
```

Первый формат приостанавливает выполнение текущего потока до завершения выполнения другого потока, а второй и третий — либо до завершения выполнения

другого потока, либо до истечения тайм-аута. Значение параметра `millis` указывается в миллисекундах, а значение параметра `nanos` — в наносекундах в пределах `0` — `999 999`. Если поток прерывается, то генерируется исключение `InterruptedException`.

В качестве примера завершим главный поток только после завершения работы другого потока (листинг 24.3).

Листинг 24.3. Метод `join()`

```
package com.example.app;

public class MyClass {
   public static void main(String[] args) {
      Thread t = new Thread(new MyThread());
      t.start();
      try {
         t.join();
      }
      catch (InterruptedException e) { }
      System.out.println("Выход. Поток " +
                         Thread.currentThread().getName());
   }
}

class MyThread implements Runnable {
   @Override
   public void run() {
      for (int i = 0; i < 10; i++) {
         System.out.println("i=" + i +
            " Поток " + Thread.currentThread().getName());
         try {
            Thread.sleep(1000);
         }
         catch (InterruptedException e) {
            Thread.currentThread().interrupt();
         }
      }
   }
}
```

24.6. Синхронизация

Если несколько потоков пытаются получить доступ к одному ресурсу, то результат этого действия может стать непредсказуемым. Рассмотрим следующий код:

```
public void print() {
   System.out.println("1. " + Thread.currentThread().getName());
```

```
    System.out.println("2. " + Thread.currentThread().getName());
    System.out.println("3. " + Thread.currentThread().getName());
}
```

Тело метода `print()` состоит из трех инструкций. При выполнении этого метода первый поток может быть прерван на любой из этих инструкций, второй поток получит доступ, но опять его прервет третий поток, и т. д. В итоге никакой атомарности при выполнении метода `print()` не будет, и результат его выполнения становится непредсказуемым. А если внутри этого метода мы меняем значения, то данные будут испорчены. Попробуйте выполнить метод `print()` из разных потоков:

```
package com.example.app;

public class MyClass {
    public static void main(String[] args) {
        Test obj = new Test();
        (new Thread(new MyThread(obj))).start();
        (new Thread(new MyThread(obj))).start();
        (new Thread(new MyThread(obj))).start();
    }
}
class Test {
    public void print() {
        System.out.println("1. " + Thread.currentThread().getName());
        System.out.println("2. " + Thread.currentThread().getName());
        System.out.println("3. " + Thread.currentThread().getName());
    }
}
class MyThread implements Runnable {
    private Test obj;
    public MyThread(Test obj) {
        this.obj = obj;
    }
    @Override
    public void run() {
        obj.print();
    }
}
```

Вот примерный результат при использовании трех потоков:

```
1. Thread-2
1. Thread-1
1. Thread-0
2. Thread-1
2. Thread-2
3. Thread-1
2. Thread-0
3. Thread-2
3. Thread-0
```

Чтобы не допустить проблем и избежать *состояния гонок*, необходимо выполнять *синхронизацию* критичных секций (секций, к которым имеют доступ несколько потоков). При доступе к синхронизированной секции поток запрашивает блокировку. Если блокировка получена, то поток изменяет что-то внутри синхронизированного блока. Если не удалось получить блокировку, то поток блокируется до момента получения разрешения. Таким образом, код внутри критичной секции в один момент времени выполняется только одним потоком как *атомарная операция*.

24.6.1. Ключевое слово *volatile*

Если потоки проверяют состояние какого-либо поля и иногда его изменяют, то можно обойтись без синхронизации, но при этом обязательно следует при объявлении поля указать ключевое слово `volatile`:

```
public volatile boolean isStop;
```

Проблема заключается в том, что компилятор и процессор могут выполнять различные оптимизации, и если ключевое слово `volatile` не указать, то значение поля может, например, кэшироваться. В результате разные потоки могут получить абсолютно разные значения. Ключевое слово `volatile` говорит компилятору, что оптимизации использовать не следует, так как к значению поля имеют доступ несколько потоков.

24.6.2. Ключевое слово *synchronized*

Чтобы к критичной секции, расположенной внутри метода, в один момент времени имел доступ только один поток, необходимо при объявлении метода указать ключевое слово `synchronized`. В этом случае поток вначале запрашивает блокировку. Если блокировка получена, то поток выполняет инструкции внутри метода. Если блокировку получить не удалось, то поток блокируется до момента получения разрешения на блокировку. Переделаем наш метод `print()`:

```
public synchronized void print() {
   System.out.println("1. " + Thread.currentThread().getName());
   System.out.println("2. " + Thread.currentThread().getName());
   System.out.println("3. " + Thread.currentThread().getName());
}
```

Теперь результат выполнения будет контролируемым, так как к инструкциям внутри метода в один момент имеет доступ только один поток:

```
1. Thread-0
2. Thread-0
3. Thread-0
1. Thread-1
2. Thread-1
3. Thread-1
1. Thread-2
2. Thread-2
3. Thread-2
```

24.6.3. Синхронизированные блоки

Вместо синхронизации кода всего метода можно использовать *синхронизированные блоки*. Схема объявления синхронизируемого блока:

```
synchronized (Объект) {
   // Критичная секция
}
```

Переделаем наш метод `print()` и используем синхронизированный блок:

```
public void print() {
   synchronized (this) {
      System.out.println("1. " + Thread.currentThread().getName());
      System.out.println("2. " + Thread.currentThread().getName());
      System.out.println("3. " + Thread.currentThread().getName());
   }
}
```

24.6.4. Методы *wait()*, *notify()* и *notifyAll()*

Все объекты в языке Java наследуют следующие методы из класса `Object`:

- `wait()` — переводит поток в режим ожидания. Форматы метода:

```
public final void wait()          throws InterruptedException
public final void wait(long timeout)
                                  throws InterruptedException
public final void wait(long timeout, int nanos)
                                  throws InterruptedException
```

  Первый формат ожидает вызова метода `notify()` или `notifyAll()` неограниченное количество времени. Второй и третий форматы позволяют указать тайм-аут ожидания. Значение параметра `timeout` указывается в миллисекундах, а значение параметра `nanos` — в наносекундах в пределах `0` — `999 999`. Если поток прерывается, то генерируется исключение `InterruptedException`;

- `notify()` — выводит из состояния ожидания один любой поток. Если один поток пишет, а второй читает, то вывод из ожидания потока-читателя при необходимости записи приведет к состоянию взаимной блокировки. Если потоки выполняют разные функции, то вместо этого метода необходимо вызвать метод `notifyAll()`. Формат метода:

```
public final void notify()
```

- `notifyAll()` — выводит из состояния ожидания все потоки. Формат метода:

```
public final void notifyAll()
```

Все эти методы должны вызываться только внутри синхронизированного метода или блока. Метод `wait()` необходимо вызывать внутри цикла `while`:

```
public synchronized String take()
                          throws InterruptedException {
```

```
    while (this.message == null) {
        this.wait();    // Ждем записи, прежде чем прочитать
    }
    String result = this.message; // Читаем
    this.message = null;          // Удаляем старое сообщение
    this.notifyAll();             // Оповещаем о прочтении
    return result;
}
```

На каждой итерации цикла проверяется условие. Если условие истинно, то можно выполнить инструкции внутри синхронизированного метода или блока. В противном случае поток переходит в режим ожидания. При переходе в режим ожидания освобождается блокировка объекта. При поступлении сигнала от метода `notify()` или `notifyAll()` блокировка восстанавливается.

В качестве примера создадим возможность обмена сообщениями между потоками (листинг 24.4). Поток-писатель (класс `MyThreadWriter`) будет записывать сообщения (при условии, что предыдущее сообщение прочитано), а поток-читатель (класс `MyThreadReader`) станет читать это сообщение (при условии, что сообщение существует) и выводить его в окно консоли. Если условие не соблюдается, то потоки переходят в режим ожидания. После записи и прочтения потоки будут оповещать друг друга об этом с помощью метода `notifyAll()`. После десяти секунд прервем выполнение потоков, отправив потокам сигнал с помощью метода `interrupt()`.

Листинг 24.4. Методы `wait()` и `notifyAll()`

```
package com.example.app;

public class MyClass {
    public static void main(String[] args)
                        throws InterruptedException {
        Thread[] arr = new Thread[2];
        Message m = new Message();
        arr[0] = new Thread(new MyThreadReader(m));
        arr[0].start();
        arr[1] = new Thread(new MyThreadWriter(m));
        arr[1].start();
        Thread.sleep(10000);  // 10 секунд на выполнение
        for (Thread t: arr) { // Завершаем работу потоков
            t.interrupt();
        }
        System.out.println("Выход. Поток main");
    }
}

class Message {
    private String message;
```

```
   public synchronized String take()
                              throws InterruptedException {
      while (this.message == null) {
         this.wait();    // Ждем записи, прежде чем прочитать
      }
      String result = this.message;
      this.message = null;
      this.notifyAll(); // Оповещаем о прочтении
      return result;
   }
   public synchronized void put(String message)
                             throws InterruptedException {
      while (this.message != null) {
         this.wait();    // Ждем прочтения, прежде чем записать
      }
      this.message = message;
      this.notifyAll(); // Оповещаем о записи
   }
}

class MyThreadReader implements Runnable { // Поток-читатель
   private Message m;
   MyThreadReader(Message m) {
      this.m = m;
   }
   @Override
   public void run() {
      while (true) {
         try {
            System.out.println("Прочитано " + m.take());
            Thread.sleep(500);
         }
         catch (InterruptedException e) { return; }
      }
   }
}

class MyThreadWriter implements Runnable { // Поток-писатель
   private Message m;
   MyThreadWriter(Message m) {
      this.m = m;
   }
   @Override
   public void run() {
      while (true) {
         try {
            m.put(String.valueOf(Math.random()));
```

```
            Thread.sleep(1000);
         }
         catch (InterruptedException e) { return; }
      }
   }
}
```

24.7. Пакет *java.util.concurrent.locks*

Для синхронизации критичных секций можно также воспользоваться классами и интерфейсами из пакета `java.util.concurrent.locks`. Код импорта всех идентификаторов из этого пакета:

```
import java.util.concurrent.locks.*;
```

24.7.1. Интерфейс *Lock*

Интерфейс `Lock` описывает объект блокировки критичной секции. Реализует этот интерфейс класс `ReentrantLock`. Форматы конструктора класса:

```
ReentrantLock()
ReentrantLock(boolean fair)
```

Если в параметре `fair` указано значение `true`, то предпочтение будет отдаваться потоку, ожидающему дольше всех. По умолчанию параметр имеет значение `false`. Пример создания объекта блокировки:

```
private final Lock myLock = new ReentrantLock();
```

Перед входом в критичную секцию необходимо вызвать метод `lock()`, а при выходе из секции — метод `unlock()`. Метод `unlock()` должен обязательно быть расположен внутри блока `finally`, иначе при возникновении исключения блокировка снята не будет, и секция окажется заблокированной навсегда. В качестве примера переделаем наш метод `print()`:

```
public void print() {
   this.myLock.lock();
   try {
      System.out.println("1. " + Thread.currentThread().getName());
      System.out.println("2. " + Thread.currentThread().getName());
      System.out.println("3. " + Thread.currentThread().getName());
   }
   finally {
      this.myLock.unlock();
   }
}
```

Метод `lock()` блокирует поток до момента получения доступа. Если необходимо избежать блокировки, то можно воспользоваться методом `tryLock()`. Форматы метода:

```
import java.util.concurrent.TimeUnit;
public boolean tryLock()
public boolean tryLock(long time, TimeUnit unit)
                    throws InterruptedException
```

Метод возвращает значение `true`, если доступ получен, и `false` — в противном случае. Первый формат сразу возвращает значение, а второй формат предварительно ждет указанное количество времени. В параметре `time` указывается значение тайм-аута, а в параметре `unit` — единица измерения (например, значения `TimeUnit.MILLISECONDS`, `TimeUnit.MICROSECONDS`, `TimeUnit.SECONDS` и др.).

24.7.2. Интерфейс *Condition*

Интерфейс `Condition` содержит аналоги методов `wait()`, `notify()` и `notifyAll()`. Называются они соответственно `await()`, `signal()` и `signalAll()`. Форматы методов:

```
public void await()  throws InterruptedException
public boolean await(long time, TimeUnit unit)
                     throws InterruptedException
public void signal()
public void signalAll()
```

Получить ссылку на объект `Condition` позволяет метод `newCondition()` объекта `Lock`. Формат метода:

```
public Condition newCondition()
```

Переделаем код класса `Message` из листинга 24.4 и используем методы `await()` и `signal()` вместо методов `wait()` и `notifyAll()` (листинг 24.5).

Листинг 24.5. Методы `await()` и `signal()`

```
// import java.util.concurrent.locks.*;
class Message {
   private String message;
   private final Lock myLock = new ReentrantLock();
   private final Condition write = myLock.newCondition();
   private final Condition read  = myLock.newCondition();

   public String take() throws InterruptedException {
      this.myLock.lock();
      try {
         while (this.message == null) {
            read.await(); // Ждем записи, прежде чем прочитать
         }
         String result = this.message;
         this.message = null;
         write.signal();  // Оповещаем о прочтении
         return result;
      }
```

```
      finally {
         this.myLock.unlock();
      }
   }
   public void put(String message)
                          throws InterruptedException {
      this.myLock.lock();
      try {
         while (this.message != null) {
            write.await(); // Ждем прочтения, прежде чем записать
         }
         this.message = message;
         read.signal();    // Оповещаем о записи
      }
      finally {
         this.myLock.unlock();
      }
   }
}
```

24.8. Пакет *java.util.concurrent*

Пакет `java.util.concurrent` содержит множество классов и интерфейсов, предназначенных для работы с многопоточными приложениями. В этом разделе мы рассмотрим лишь самые основные классы, которые будут полезны прикладному программисту. Если этих возможностей вам окажется недостаточно, то обращайтесь к документации по пакету.

Основные проблемы при работе с потоками возникают при доступе нескольких потоков к одному ресурсу. Как вы уже знаете, для решения этой проблемы необходимо выполнять синхронизацию критичных секций. Если синхронизация выполнена, то поток запрашивает блокировку, и если получает ее, то может производить изменения внутри критичной секции. Остальные потоки при доступе к этой же секции блокируются до момента получения разрешения на блокировку. Если что-то пошло не так, то потоки могут ждать друг друга вечно. Такая ситуация называется *взаимной блокировкой.*

Как ни крути, но потоки должны общаться между собой. Одни потоки должны раздавать задачи, а другие получать и выполнять эти задачи, а затем возвращать результат. В предыдущих разделах мы создали несколько вариантов класса `Message`, который, собственно, и предназначен для раздачи и получения задач. Но, во-первых, он работает с задачами в виде строки, во-вторых, в один момент времени может содержать только одну задачу, а в-третьих, не позволяет работать с приоритетами. Все эти проблемы решаются с помощью блокирующих очередей из пакета `java.util.concurrent`. Импортировать все идентификаторы из этого пакета позволяет следующая инструкция:

```
import java.util.concurrent.*;
```

24.8.1. Интерфейс *BlockingQueue<E>*: блокирующая односторонняя очередь

Интерфейс `BlockingQueue<E>` описывает блокирующую одностороннюю очередь «первым пришел, первым ушел» (обычная очередь, например, за билетами в кинотеатр — чем раньше встали в очередь, тем быстрее купите билет). Иерархия наследования:

```
Iterable<E> - Collection<E> - Queue<E> - BlockingQueue<E>
```

Приведем основные методы интерфейса `BlockingQueue<E>`:

- `put()` — добавляет элемент в конец очереди. Если очередь полна, то поток блокируется до момента освобождения места. Если получен сигнал прерывания, то метод генерирует исключение `InterruptedException`. Формат метода:

  ```
  public void put(E e)  throws InterruptedException
  ```

- `take()` — удаляет элемент из начала очереди и возвращает его. Если элементов нет, то поток блокируется до момента появления элементов в очереди. Если получен сигнал прерывания, то метод генерирует исключение `InterruptedException`. Формат метода:

  ```
  public E take()       throws InterruptedException
  ```

- `add()` — добавляет элемент в конец очереди. Метод возвращает значение `true`, если элемент был добавлен. В противном случае генерируется исключение `IllegalStateException`. Формат метода:

  ```
  public boolean add(E e)
  ```

- `offer()` — добавляет элемент в конец очереди. Метод возвращает значение `true`, если элемент был добавлен, и `false` — если очередь полна. Форматы метода:

  ```
  public boolean offer(E e)
  public boolean offer(E e, long timeout, TimeUnit unit)
                           throws InterruptedException
  ```

  Если очередь полна, то первый формат сразу вернет значение `false`, а второй формат будет предварительно ожидать освобождения места указанное количество времени и только по истечении этого срока вернет значение `false`. В параметре `timeout` указывается значение тайм-аута, а в параметре `unit` — единица измерения (например, значения `TimeUnit.MILLISECONDS`, `TimeUnit.MICROSECONDS`, `TimeUnit.SECONDS` и др.);

- `remove()` — находит в очереди первый элемент, соответствующий указанному объекту, и удаляет его. Метод возвращает значение `true`, если элемент был удален, и `false` — в противном случае. Формат метода:

  ```
  public boolean remove(Object o)
  ```

- `poll()` — удаляет первый элемент из очереди и возвращает его. Если очередь пуста, то метод возвращает значение `null`. Формат метода:

```
public E poll(long timeout, TimeUnit unit)
            throws InterruptedException
```

Если очередь пуста, то метод будет предварительно ожидать добавления элемента указанное количество времени и только по истечении этого срока вернет значение `null`. В параметре `timeout` указывается значение тайм-аута, а в параметре `unit` — единица измерения (например, значения `TimeUnit.MILLISECONDS`, `TimeUnit.MICROSECONDS`, `TimeUnit.SECONDS` и др.).

Интерфейс `BlockingQueue<E>` наследует интерфейс `Queue<E>`, поэтому можно использовать и методы из этого интерфейса, но следует учитывать, что многие методы не гарантируют атомарности операции.

Интерфейс `BlockingQueue<E>` реализуют классы `ArrayBlockingQueue<E>`, `LinkedBlockingDeque<E>`, `LinkedBlockingQueue<E>`, `LinkedTransferQueue<E>`, `PriorityBlockingQueue<E>`, `DelayQueue<E extends Delayed>` и `SynchronousQueue<E>`.

Класс `ArrayBlockingQueue<E>` строит очередь на основе массива фиксированного размера. Конструкторы класса:

```
ArrayBlockingQueue(int capacity)
ArrayBlockingQueue(int capacity, boolean fair)
ArrayBlockingQueue(int capacity, boolean fair,
                   Collection<? extends E> c)
```

Параметр `capacity` задает фиксированную емкость очереди. Если в параметре `fair` указано значение `true`, то предпочтение будет отдаваться потоку, ожидающему дольше всех. По умолчанию параметр имеет значение `false`. В параметре `c` можно указать коллекцию, элементы которой будут добавлены в очередь. Если коллекция имеет больше элементов, чем емкость очереди, то генерируется исключение `IllegalArgumentException`. Пример создания односторонней очереди из двух элементов, имеющих тип `String`:

```
BlockingQueue<String> queue = new ArrayBlockingQueue<String>(2);
```

Класс `LinkedBlockingQueue<E>` строит очередь на основе связанного списка. Конструкторы класса:

```
LinkedBlockingQueue()
LinkedBlockingQueue(int capacity)
LinkedBlockingQueue(Collection<? extends E> c)
```

Первый конструктор создает очередь с максимальной емкостью `Integer.MAX_VALUE`. Второй конструктор позволяет указать в параметре `capacity` фиксированную емкость очереди. Третий конструктор создает очередь с максимальной емкостью `Integer.MAX_VALUE` и добавляет в очередь элементы из коллекции `c`. Пример создания очереди из элементов, имеющих тип `String`:

```
BlockingQueue<String> queue = new LinkedBlockingQueue<String>();
```

Переделаем код из листинга 24.4 и используем блокирующую очередь из двух элементов вместо класса `Message` (листинг 24.6).

Листинг 24.6. Интерфейс BlockingQueue<E>

```
package com.example.app;

import java.util.concurrent.*;

public class MyClass {
   public static void main(String[] args)
                        throws InterruptedException {
      Thread[] arr = new Thread[3];
      BlockingQueue<String> queue = new LinkedBlockingQueue<String>(2);
      arr[0] = new Thread(new MyThreadReader(queue));
      arr[0].start();
      arr[1] = new Thread(new MyThreadWriter(queue));
      arr[1].start();
      arr[2] = new Thread(new MyThreadReader(queue));
      arr[2].start();
      Thread.sleep(10000);  // 10 секунд на выполнение
      for (Thread t: arr) { // Завершаем работу потоков
         t.interrupt();
      }
      System.out.println("Выход. Поток main");
   }
}

class MyThreadReader implements Runnable { // Поток-читатель
   private BlockingQueue<String> queue;
   MyThreadReader(BlockingQueue<String> queue) {
      this.queue = queue;
   }
   @Override
   public void run() {
      while (true) {
         try {
            System.out.println("Прочитано " + queue.take()
               + " " + Thread.currentThread().getName());
            Thread.sleep(1000);
         }
         catch (InterruptedException e) { return; }
      }
   }
}

class MyThreadWriter implements Runnable { // Поток-писатель
   private BlockingQueue<String> queue;
   MyThreadWriter(BlockingQueue<String> queue) {
      this.queue = queue;
   }
```

```
    @Override
    public void run() {
        while (true) {
            try {
                queue.put(String.valueOf(Math.random()));
                Thread.sleep(500);
            }
            catch (InterruptedException e) { return; }
        }
    }
}
```

24.8.2. Интерфейс *BlockingDeque<E>*: блокирующая двухсторонняя очередь

Интерфейс `BlockingDeque<E>` описывает блокирующую двухстороннюю очередь. Иерархия наследования:

```
Iterable<E> - Collection<E> - Queue<E> - BlockingQueue<E> -
                              Deque<E> - BlockingDeque<E>
```

Приведем основные методы интерфейса `BlockingDeque<E>`:

- `putFirst()` — добавляет элемент в начало очереди. Если очередь полна, то поток блокируется до момента освобождения места. Если получен сигнал прерывания, то метод генерирует исключение `InterruptedException`. Формат метода:

```
public void putFirst(E e)      throws InterruptedException
```

- `put()` и `putLast()` — добавляют элемент в конец очереди. Если очередь полна, то поток блокируется до момента освобождения места. Если получен сигнал прерывания, то метод генерирует исключение `InterruptedException`. Форматы методов:

```
public void put(E e)           throws InterruptedException
public void putLast(E e)       throws InterruptedException
```

- `take()` и `takeFirst()` — удаляют элемент из начала очереди и возвращают его. Если элементов нет, то поток блокируется до момента появления элементов в очереди. Если получен сигнал прерывания, то методы генерируют исключение `InterruptedException`. Форматы методов:

```
public E take()                throws InterruptedException
public E takeFirst()           throws InterruptedException
```

- `takeLast()` — удаляет элемент из конца очереди и возвращает его. Если элементов нет, то поток блокируется до момента появления элементов в очереди. Если получен сигнал прерывания, то метод генерирует исключение `InterruptedException`. Формат метода:

```
public E takeLast()            throws InterruptedException
```

- `addFirst()` и `push()` — добавляют элемент в начало очереди. Если очередь полна, генерируется исключение `IllegalStateException`. Форматы методов:

```
public void addFirst(E e)
public void push(E e)
```

- `add()` и `addLast()` — добавляют элемент в конец очереди. Метод `add()` возвращает значение `true`, если элемент был добавлен. Если очередь полна, генерируется исключение `IllegalStateException`. Форматы методов:

```
public boolean add(E e)
public void addLast(E e)
```

- `offerFirst()` — добавляет элемент в начало очереди. Метод возвращает значение `true`, если элемент был добавлен, и `false` — если очередь полна. Форматы метода:

```
public boolean offerFirst(E e)
public boolean offerFirst(E e, long timeout, TimeUnit unit)
                          throws InterruptedException
```

  Если очередь полна, то первый формат сразу вернет значение `false`, а второй формат будет предварительно ожидать освобождения места указанное количество времени и только по истечении этого срока вернет значение `false`. В параметре `timeout` указывается значение тайм-аута, а в параметре `unit` — единица измерения (например, значения `TimeUnit.MILLISECONDS`, `TimeUnit.MICROSECONDS`, `TimeUnit.SECONDS` и др.);

- `offer()` и `offerLast()` — добавляют элемент в конец очереди. Методы возвращают значение `true`, если элемент был добавлен, и `false` — если очередь полна. Форматы методов:

```
public boolean offer(E e)
public boolean offer(E e, long timeout, TimeUnit unit)
                          throws InterruptedException
public boolean offerLast(E e)
public boolean offerLast(E e, long timeout, TimeUnit unit)
                          throws InterruptedException
```

  Если очередь полна, то первый и третий форматы сразу вернут значение `false`, а второй и четвертый форматы будут предварительно ожидать освобождения места указанное количество времени и только по истечении этого срока вернут значение `false`. В параметре `timeout` указывается значение тайм-аута, а в параметре `unit` — единица измерения (например, значения `TimeUnit.MILLISECONDS`, `TimeUnit.MICROSECONDS`, `TimeUnit.SECONDS` и др.);

- `remove()` — находит в очереди первый элемент, соответствующий указанному объекту, и удаляет его. Метод возвращает значение `true`, если элемент был удален, и `false` — в противном случае. Формат метода:

```
public boolean remove(Object o)
```

- `poll()` и `pollFirst()` — удаляют первый элемент из очереди и возвращают его. Если очередь пуста, то методы возвращают значение `null`. Форматы методов:

```
public E poll(long timeout, TimeUnit unit)
              throws InterruptedException
public E pollFirst(long timeout, TimeUnit unit)
              throws InterruptedException
```

  Если очередь пуста, то методы будут предварительно ожидать добавления элемента указанное количество времени и только по истечении этого срока вернут значение `null`. В параметре `timeout` указывается значение тайм-аута, а в параметре `unit` — единица измерения (например, значения `TimeUnit.MILLISECONDS`, `TimeUnit.MICROSECONDS`, `TimeUnit.SECONDS` и др.);

- `pollLast()` — удаляет последний элемент из очереди и возвращает его. Если очередь пуста, то метод возвращает значение `null`. Формат метода:

```
public E pollLast(long timeout, TimeUnit unit)
                  throws InterruptedException
```

  Если очередь пуста, то метод будет предварительно ожидать добавления элемента указанное количество времени и только по истечении этого срока вернет значение `null`. В параметре `timeout` указывается значение тайм-аута, а в параметре `unit` — единица измерения (например, значения `TimeUnit.MILLISECONDS`, `TimeUnit.MICROSECONDS`, `TimeUnit.SECONDS` и др.).

Интерфейс `BlockingDeque<E>` реализует класс `LinkedBlockingDeque<E>`. Конструкторы класса:

```
LinkedBlockingDeque()
LinkedBlockingDeque(int capacity)
LinkedBlockingDeque(Collection<? extends E> c)
```

Первый конструктор создает очередь с максимальной емкостью `Integer.MAX_VALUE`. Второй конструктор позволяет указать в параметре `capacity` фиксированную емкость очереди. Третий конструктор создает очередь с максимальной емкостью `Integer.MAX_VALUE` и добавляет в очередь элементы из коллекции `c`. Пример создания очереди из двух элементов, имеющих тип `String`:

```
BlockingDeque<String> queue = new LinkedBlockingDeque<String>(2);
```

24.8.3. Класс *PriorityBlockingQueue<E>*: блокирующая очередь с приоритетами

Класс `PriorityBlockingQueue<E>` реализует интерфейс `BlockingQueue<E>` и описывает неограниченную размером блокирующую очередь с приоритетами. Из очереди первым возвращается элемент с наибольшим приоритетом (меньшим значением). Если очередь пуста, то поток блокируется до момента добавления элемента. Конструкторы класса:

```
PriorityBlockingQueue()
PriorityBlockingQueue(int initialCapacity)
```

```
PriorityBlockingQueue(Collection<? extends E> c)
PriorityBlockingQueue(int initialCapacity,
                      Comparator<? super E> comparator)
```

Первый конструктор создает очередь емкостью 11 элементов, второй конструктор позволяет указать начальную емкость, а третий конструктор создает очередь на основе другой коллекции. Во всех этих конструкторах используется определение приоритета по умолчанию (с помощью метода `compareTo()` из интерфейса `Comparable<T>`):

```
BlockingQueue<String> queue = new PriorityBlockingQueue<String>(11);
queue.put("1");
queue.put("2");
queue.put("3");
System.out.println(queue.take()); // 1
```

Четвертый конструктор позволяет указать начальную емкость и способ определения приоритета (объект, реализующий интерфейс `Comparator<T>`). Изменим приоритет на противоположный с помощью лямбда-выражения:

```
BlockingQueue<String> queue =
   new PriorityBlockingQueue<String>(11, (a, b) -> b.compareTo(a));
queue.put("1");
queue.put("2");
queue.put("3");
System.out.println(queue.take()); // 3
```

24.8.4. Интерфейсы *Callable<V>* и *Future<V>*

Очень часто мы создаем поток, чтобы выполнить в нем задание, и затем хотим получить результат его выполнения. Для этого существует интерфейс `Callable<V>` с единственным методом `call()`:

```
public V call() throws Exception
```

Метод ничего не принимает и возвращает обобщенный тип. Реальный возвращаемый тип указывается при реализации интерфейса. Обратите внимание: в отличие от метода `run()` из интерфейса `Runnable`, метод `call()` может генерировать контролируемые исключения.

Интерфейс `Future<V>` позволяет хранить и получать результат выполнения задания. Интерфейс содержит следующие методы:

- `get()` — возвращает результат выполнения задания. Форматы метода:

```
public V get()
         throws InterruptedException, ExecutionException
public V get(long timeout, TimeUnit unit)
         throws InterruptedException, ExecutionException,
                TimeoutException
```

Первый формат будет ждать завершения выполнения задачи, блокируя текущий поток. Второй формат будет ждать только указанное количество времени. В па-

раметре `timeout` указывается значение тайм-аута, а в параметре `unit` — единица измерения (например, значения `TimeUnit.MILLISECONDS`, `TimeUnit.MICROSECONDS`, `TimeUnit.SECONDS` и др.). Если тайм-аут истек, то генерируется исключение `TimeoutException`. Если задание было отменено, то метод генерирует исключение `CancellationException`. Если поток был прерван, то метод генерирует исключение `InterruptedException`. Если вычисление вызвало исключение, то метод генерирует исключение `ExecutionException`;

- `isDone()` — возвращает значение `true`, если выполнение задания завершено, и `false` — в противном случае. Метод не гарантирует нормального завершения и возвращает `true` даже при отмене задания. Формат метода:

```
public boolean isDone()
```

- `isCancelled()` — возвращает значение `true`, если задание было отменено, и `false` — в противном случае. Формат метода:

```
public boolean isCancelled()
```

- `cancel()` — позволяет отменить задание. Если задание успешно отменено, то метод возвращает значение `true`, в противном случае — `false`. Если задание было запущено на выполнение до вызова метода `cancel()`, то параметр `mayInterruptIfRunning` определяет, следует ли прервать поток. Формат метода:

```
public boolean cancel(boolean mayInterruptIfRunning)
```

Интерфейсы `Future<V>` и `Runnable` реализует класс `FutureTask<V>`. Конструкторы класса:

```
FutureTask(Callable<V> callable)
FutureTask(Runnable runnable, V result)
```

В качестве примера реализуем интерфейс `Callable<V>`, запустим задание на выполнение и получим результат (листинг 24.7).

Листинг 24.7. Интерфейс `Callable<V>`

```
package com.example.app;

import java.util.concurrent.*;

public class MyClass {
   public static void main(String[] args) {
      FutureTask<Integer> ft =
          new FutureTask<Integer>(new MyCallable(10));
      (new Thread(ft)).start();
      try {
         Integer result = ft.get();
         System.out.println("Результат: " + result);
      }
```

```
        catch (Exception e) {
            System.out.println("Ошибка");
        }
    }
}

class MyCallable implements Callable<Integer> {
    private Integer x;
    public MyCallable(Integer x) {
        this.x = x;
    }
    @Override
    public Integer call() throws Exception {
        System.out.println("Задание запущено");
        Thread.sleep(1000); // Имитация выполнения задания
        return this.x * 2;
    }
}
```

24.8.5. Пулы потоков

Вместо запуска отдельных потоков можно использовать *пулы потоков*. Пулы потоков содержат множество потоков в состоянии ожидания, готовых к выполнению множества заданий. Когда выполнение задания завершается, поток не уничтожается, а продолжает ожидать новые задания. Создать пул потоков позволяют статические методы `newFixedThreadPool()` и `newCachedThreadPool()` из класса `Executors`. Форматы методов:

```
public static ExecutorService newFixedThreadPool(int nThreads)
public static ExecutorService newCachedThreadPool()
```

Метод `newFixedThreadPool()` создает пул с фиксированным количеством потоков (значение параметра `nThreads`). Если заданий больше, чем потоков в пуле, то задания будут ожидать в очереди, пока какой-либо из потоков не освободится. Потоки в пуле будут существовать, пока не будет вызван метод `shutdown()`.

Метод `newCachedThreadPool()` создает пул, в котором потоки создаются по мере необходимости. Если существует ожидающий поток, то будет использован он, в противном случае создается новый поток. Если поток не был использован в течение 60 секунд, он завершает свою работу автоматически.

Начиная с Java 8, доступен также метод `newWorkStealingPool()`. Форматы метода:

```
public static ExecutorService newWorkStealingPool()
public static ExecutorService newWorkStealingPool(int parallelism)
```

Параметр `parallelism` задает максимальное количество потоков. Фактическое количество потоков может увеличиваться и уменьшаться динамически. Если пара-

метр не указан, то используется значение, возвращаемое методом `availableProcessors()` из класса `Runtime`:

```
System.out.println(Runtime.getRuntime().availableProcessors()); // 4
```

Интерфейс `ExecutorService` содержит следующие основные методы:

- `submit()` — добавляет задание в очередь. Форматы метода:

```
public <T> Future<T> submit(Callable<T> task)
public Future<?> submit(Runnable task)
public <T> Future<T> submit(Runnable task, T result)
```

- `shutdown()` — завершает выполнение заданий. Все задания, ожидающие в очереди, отменяются, а выполняющиеся — так и будут выполняться, пока не завершится выполнение задания. Формат метода:

```
public void shutdown()
```

- `shutdownNow()` — пытается завершить все задания: и ожидающие в очереди, и выполняющиеся. Формат метода:

```
public List<Runnable> shutdownNow()
```

- `awaitTermination()` — ожидает завершения потоков указанное количество времени и возвращает статус завершения (значение `true`, если потоки успешно завершены). Блокировка текущего потока снимается, если работа потоков завершена или истек тайм-аут или текущий поток прерывается. Формат метода:

```
public boolean awaitTermination(long timeout, TimeUnit unit)
                                throws InterruptedException
```

Запустим двадцать заданий на выполнение, используя пул потоков, и получим все результаты (листинг 24.8).

Листинг 24.8. Пулы потоков

```
package com.example.app;

import java.util.*;
import java.util.concurrent.*;

public class MyClass {
   public static void main(String[] args) {
      ExecutorService pool = Executors.newWorkStealingPool();
      List<Future<Integer>> list = new ArrayList<>();
      for (int i = 1; i <= 20; i++) {
         Future<Integer> future = pool.submit(new MyCallable(i));
         list.add(future);
      }
      for (Future<Integer> future: list) {
         try {
            Integer result = future.get();
```

```
            System.out.println("Результат: " + result);
         }
         catch (Exception e) {
            System.out.println("Ошибка");
         }
      }
      pool.shutdown();
      try {
         if (!pool.awaitTermination(10, TimeUnit.SECONDS)) {
            pool.shutdownNow();
            if (!pool.awaitTermination(10, TimeUnit.SECONDS))
               System.out.println("Ошибка при завершении");
         }
      }
      catch (InterruptedException e) {
         pool.shutdownNow();
         Thread.currentThread().interrupt();
      }
      System.out.println("Выход");
   }
}

class MyCallable implements Callable<Integer> {
   private Integer x;
   public MyCallable(Integer x) {
      this.x = x;
   }
   @Override
   public Integer call() throws Exception {
      System.out.println("x=" + this.x + " Поток:" +
                         Thread.currentThread().getName());
      Thread.sleep(1000); // Имитация выполнения задания
      return this.x * 2;
   }
}
```

24.9. Синхронизация коллекций

Все коллекции, которые мы рассматривали в *главах 14* и *15*, по умолчанию не синхронизированы и не могут использоваться для доступа из разных потоков. Исключения представляют унаследованные коллекции `Vector<E>` и `Hashtable<K, V>`, которые изначально являются синхронизированными. Однако существует способ выполнить синхронизацию коллекций. Для этого используются следующие статические методы из класса `Collections`:

```
public static <T> Collection<T>
                  synchronizedCollection(Collection<T> c)
```

```
public static <T> List<T> synchronizedList(List<T> list)
public static <K, V> Map<K, V> synchronizedMap(Map<K, V> m)
public static <K, V> NavigableMap<K, V>
                  synchronizedNavigableMap(NavigableMap<K, V> m)
public static <T> NavigableSet<T>
                  synchronizedNavigableSet(NavigableSet<T> s)
public static <T> Set<T> synchronizedSet(Set<T> s)
public static <K, V> SortedMap<K, V>
                  synchronizedSortedMap(SortedMap<K, V> m)
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)
```

Пример синхронизации списка:

```
List<String> arr = Collections.synchronizedList(
                                  new ArrayList<String>());
arr.add("1");
arr.add("2");
arr.add("3");
synchronized (arr) {
   for (String s: arr) {
      System.out.println(s);
   }
}
```

Обратите внимание: после синхронизации не должно остаться ссылок на не синхронизированную коллекцию. Кроме того, если необходимо выполнить несколько операций как единое целое, следует произвести дополнительную синхронизацию. В этом примере мы синхронизировали код перебора элементов коллекции с помощью синхронизированного блока.

Пакет `java.util.concurrent` **содержит дополнительные коллекции, созданные с учетом особенностей работы с многочисленными потоками. Некоторые из них могут оказаться лучшим решением, чем синхронизация коллекций. Обратите внимание на коллекции** `CopyOnWriteArrayList<E>`, `CopyOnWriteArraySet<E>`, `ConcurrentLinkedQueue<E>`, `ConcurrentLinkedDeque<E>`, `ConcurrentHashMap<K, V>`, `ConcurrentSkipListMap<K, V>` **и** `ConcurrentSkipListSet<E>`. **За подробной информацией обращайтесь к документации.**

ГЛАВА 25

Java SE 11

Язык Java стремительно развивается, в результате чего новые его версии выходят теперь каждые полгода. К сожалению, процесс написания, подготовки и печати книги не успевает за такими темпами развития языка — сразу же после выхода книги из печати она уже устаревает. Тем не менее, не все так плохо. Новые версии языка в большинстве случаев отличаются от старых версий лишь добавлением новой функциональности. Поэтому все примеры из предыдущих глав книги работают в Java 11 и, скорее всего, будут работать и в более новых версиях. В этой главе мы рассмотрим отличия Java 11 от Java 10, а также возможности, добавленные в новой версии языка.

25.1. Установка OpenJDK 11

Начиная с Java 11, компания Oracle изменила лицензионное соглашение. Теперь дистрибутив, называемый Oracle JDK, можно бесплатно загружать с сайта компании только для тестирования, а чтобы использовать его в коммерческих целях, нужно дополнительно приобрести лицензию. Поэтому для изучения возможностей Java 11 мы воспользуемся не Oracle JDK, а свободной реализацией JDK, которая имеет название *OpenJDK*.

OpenJDK представляет собой комплект разработчика Java Development Kit (сокращенно JDK), включающий компилятор javac.exe, стандартные библиотеки классов, исполнительную среду Java Runtime Environment (сокращенно JRE) и различные утилиты. Для загрузки дистрибутива переходим на страницу **http://jdk.java.net/11/** и скачиваем архив openjdk-11_windows-x64_bin.zip. Далее в корне диска C: создаем папку OpenJDK и копируем в нее папку jdk-11 из скачанного архива. Путь до компилятора должен быть таким: C:\OpenJDK\jdk-11\bin\javac.exe.

Затем необходимо добавить путь к папке C:\OpenJDK\jdk-11\bin в системную переменную PATH. Если вы устанавливали предыдущие версии Java, то нужно дополнительно удалить из этой переменной путь C:\ProgramData\Oracle\Java\javapath, иначе будет запускаться другая версия Java.

Чтобы изменить системную переменную, переходим в **Параметры | Панель управления | Система и безопасность | Система | Дополнительные параметры системы**. В результате откроется окно **Свойства системы**. На вкладке **Дополнительно** нажимаем кнопку **Переменные среды**. В открывшемся окне в списке **Системные переменные** выделяем строку с переменной **Path** и нажимаем кнопку **Изменить**. В открывшемся окне изменяем значение в поле **Значение переменной** — для этого переходим в конец существующей строки, ставим точку с запятой, а затем вводим путь к папке C:\OpenJDK\jdk-11\bin:

```
<Текущее значение>;C:\OpenJDK\jdk-11\bin
```

> ***Внимание!***
>
> Случайно не удалите существующее значение переменной PATH, иначе другие приложения перестанут запускаться.

Помимо добавления в переменную PATH пути к папке C:\OpenJDK\jdk-11\bin, во избежание проблем с настройками различных редакторов необходимо также прописать переменную окружения JAVA_HOME и присвоить ей путь к папке с установленным JDK. Делается это в том же окне **Переменные среды**, но в списке переменных для пользователя, хотя вы можете добавить ее и в список системных переменных. Нажимаем в этом окне кнопку **Создать**, в поле **Имя переменной** вводим значение JAVA_HOME, а в поле **Значение переменной** — C:\OpenJDK\jdk-11. Сохраняем все изменения.

Запускаем командную строку и проверяем установку переменной JAVA_HOME:

```
C:\book>set JAVA_HOME
JAVA_HOME=C:\OpenJDK\jdk-11
```

Чтобы проверить правильность установки Java 11, выполняем следующие команды:

```
java --version
javac --version
```

Результат должен быть примерно следующим:

```
C:\book>java --version
openjdk 11 2018-09-25
OpenJDK Runtime Environment 18.9 (build 11+28)
OpenJDK 64-Bit Server VM 18.9 (build 11+28, mixed mode)

C:\book>javac --version
javac 11
```

> ***Примечание***
>
> Если вы получили в результате другую версию Java или компилятора, то вначале перезапустите командную строку и повторите проверку. Если версия не изменилась, то, скорее всего, в переменной PATH прописан путь к предыдущей версии Java. Путь к старой версии Java нужно удалить.

25.2. Установка и настройка редактора Eclipse

Для загрузки редактора Eclipse переходим на страницу **http://www.eclipse.org/downloads/packages/** и скачиваем архив с программой из раздела **Eclipse IDE for Java Developers**. Распаковываем скачанный архив и копируем папку eclipse с файлами редактора в папку C:\JavaSE. Редактор не нуждается в установке, поэтому просто переходим в папку C:\JavaSE\eclipse и запускаем файл eclipse.exe.

При запуске редактор попросит указать папку с рабочим пространством. Указываем C:\JavaSE\projects и нажимаем кнопку **Launch**.

Редактор Eclipse использует встроенный компилятор, а не внешний компилятор из папки с установленным JDK. Причем доступны в нем компиляторы практически всех версий. Для выбора версии компилятора в меню **Window** редактора выбираем пункт **Preferences**. В открывшемся окне переходим на вкладку **Java | Compiler** (рис. 25.1) и выбираем нужную версию из списка **Compiler compliance level**. Мы будем изучать Java 11, поэтому в этом списке должен быть выбран пункт **11**.

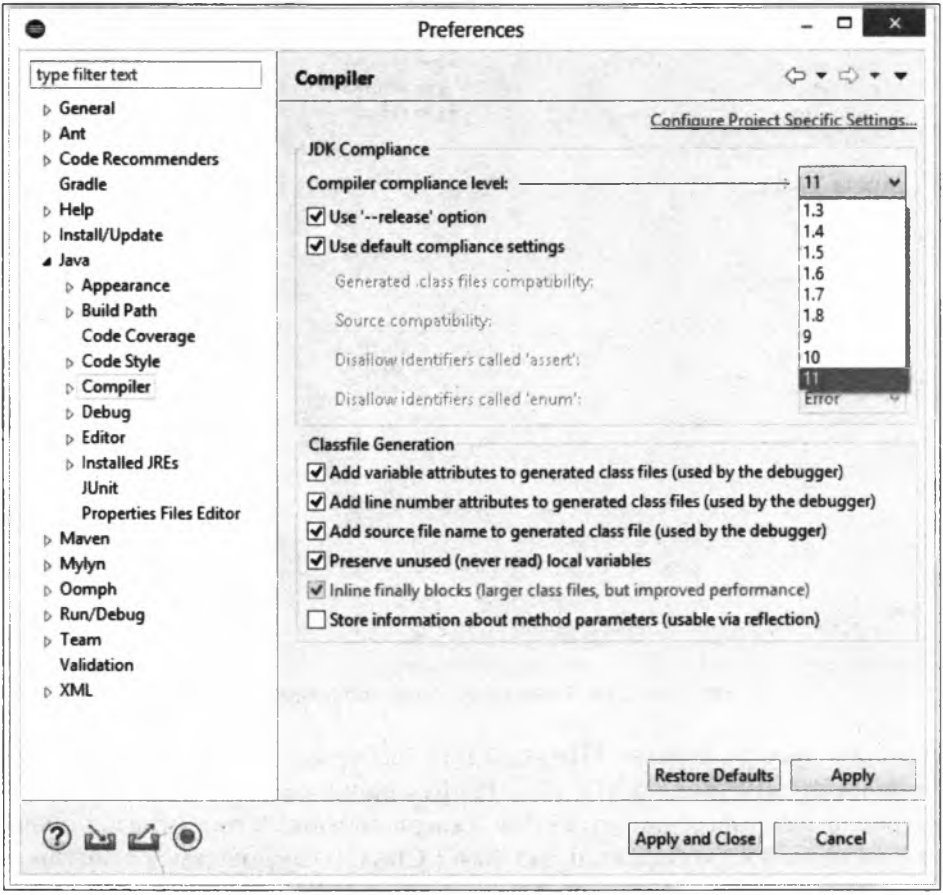

Рис. 25.1. Окно **Preferences**: вкладка **Compiler**

Если в списке **Compiler compliance level** нет пункта **11**, то в меню **Help** выбираем пункт **Eclipse Marketplace**. В открывшемся окне в поле **Find** вводим `Java 11` и нажимаем клавишу <Enter>. В списке находим пункт **Java 11 support for Eclipse** и нажимаем кнопку **Install**. После установки перезапускаем редактор, чтобы изменения вступили в силу.

Теперь добавим Java 11 в настройки редактора. Для этого в меню **Window** выбираем пункт **Preferences**. В открывшемся окне переходим на вкладку **Java | Installed JREs** (рис. 25.2). Если в списке нет пункта **jdk-11**, то нажимаем кнопку **Search** и указываем папку C:\OpenJDK. Все найденные версии отобразятся в списке. Устанавливаем флажок напротив пункта **jdk-11** и сохраняем настройки.

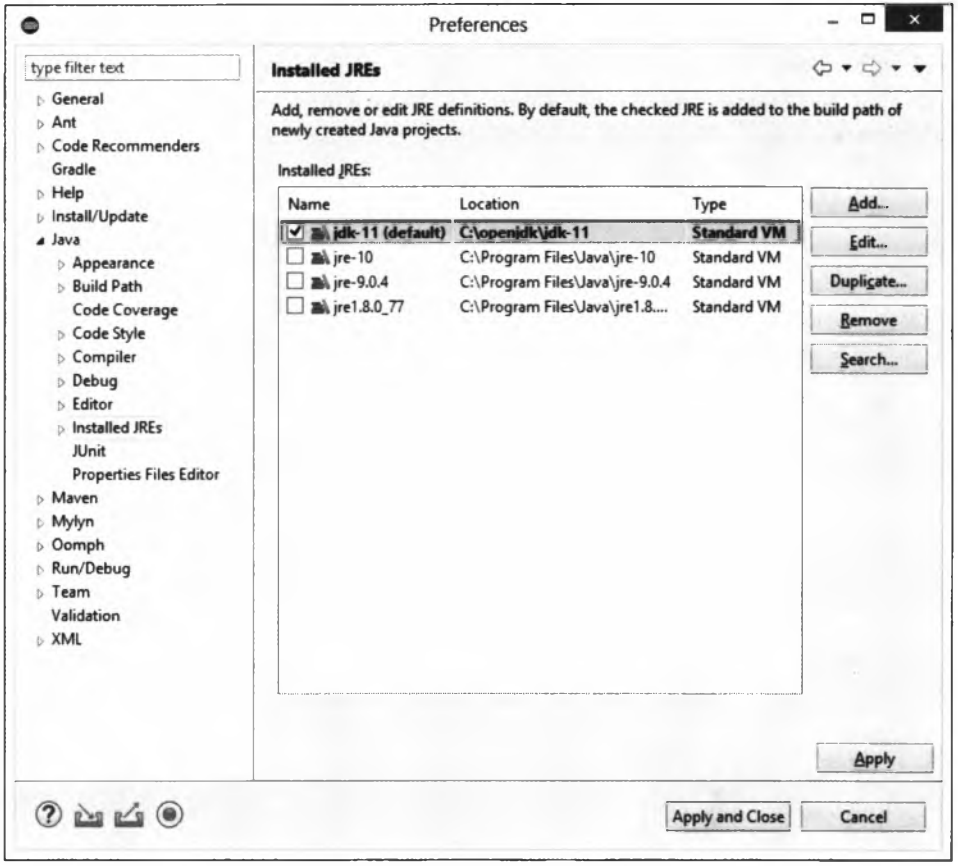

Рис. 25.2. Окно **Preferences**: вкладка **Installed JREs**

Для создания проекта в меню **File** редактора выбираем пункт **New | Java Project**. В открывшемся окне (рис. 25.3) в поле **Project name** вводим `HelloWorld`, выбираем версию JRE и нажимаем кнопку **Finish**. Теперь добавим в проект файл с классом. Для этого в меню **File** выбираем пункт **New | Class**. В открывшемся окне (рис. 25.4) в поле **Name** вводим `HelloWorld` и нажимаем кнопку **Finish**. Редактор создаст файл HelloWorld.java и откроет его для редактирования. Причем внутри файла уже будет

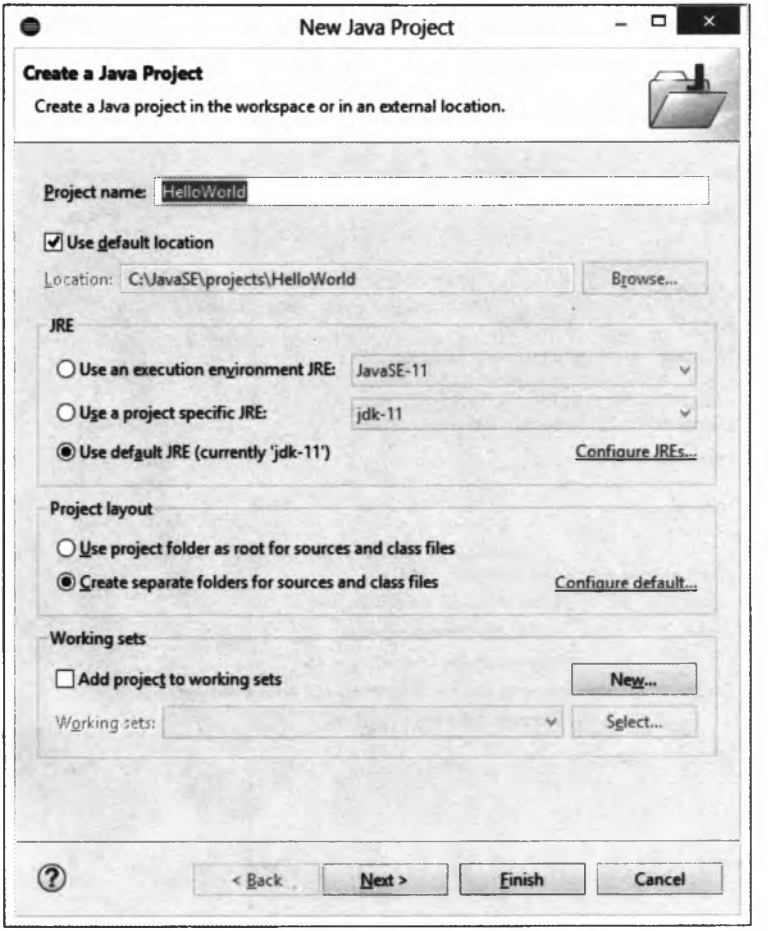

Рис. 25.3. Создание проекта

вставлен код. Вводим сюда код из листинга 1.1 и сохраняем проект. Для этого в меню **File** выбираем пункт **Save**.

Давайте теперь откроем папку C:\JavaSE\projects в Проводнике и посмотрим ее содержимое. В результате наших действий редактор создал папку HelloWorld и две папки внутри нее: src и bin. Внутри папки C:\JavaSE\projects\HelloWorld\src мы можем найти файл HelloWorld.java, который содержит код из листинга 1.1. Помимо этого редактор создал вспомогательные папки и файлы, названия которых начинаются с точки. В этих папках и файлах редактор сохраняет различные настройки. Не изменяйте и не удаляйте эти файлы.

Теперь попробуем скомпилировать программу и запустить ее на выполнение. Для этого командная строка нам не понадобится. Переходим в редактор и в меню **Run** выбираем пункт **Run** или нажимаем комбинацию клавиш <Ctrl>+<F11>. В результате в папке C:\JavaSE\projects\HelloWorld\bin будет создан файл HelloWorld.class с байт-кодом, а результат выполнения программы отобразится в окне **Console** редактора Eclipse. Все очень просто, быстро и удобно.

Рис. 25.4. Создание класса

Редактор позволяет получить справку при наведении указателя мыши на какое-либо название. Например, наведите указатель на метод `println()`, и вы получите краткую справку по этому методу. Это будет работать при активном подключении к Интернету или при подключении архива src.zip с исходным кодом (расположен в папке C:\OpenJDK\jdk-11\lib). В моем случае редактор подключил архив с исходным кодом автоматически (рис. 25.5), поэтому документацию скачивать отдельно не нужно, она входит в состав исходного кода.

Как уже было отмечено ранее, работать мы будем с кодировкой UTF-8, поэтому давайте настроим кодировку файлов по умолчанию. Для этого в меню **Window** выбираем пункт **Preferences**. В открывшемся окне переходим на вкладку **General | Workspace** (см. рис. 1.15). В группе **Text file encoding** устанавливаем флажок **Other** и из списка выбираем кодировку **UTF-8**. Сохраняем изменения. Если необходимо изменить кодировку уже открытого файла, в меню **Edit** выбираем пункт **Set Encoding**.

По умолчанию редактор вместо пробелов вставляет символы табуляции. Нас это не устраивает. Давайте изменим настройку форматирования кода. Для этого в меню

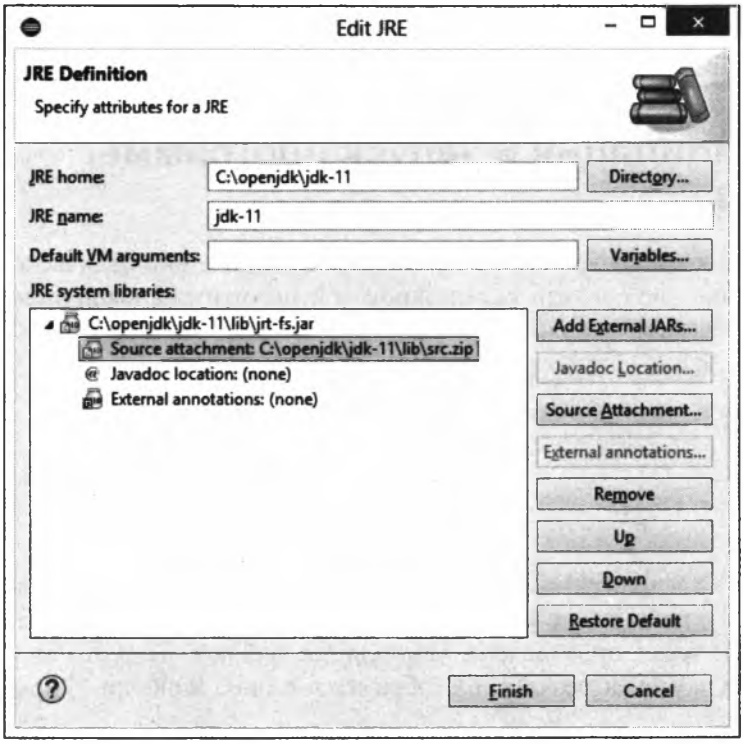

Рис. 25.5. Подключение архива с исходным кодом в окне Edit JRE

Window выбираем пункт **Preferences**. В открывшемся окне переходим на вкладку **Java | Code Style | Formatter** (см. рис. 1.16). Нажимаем кнопку **New**. В открывшемся окне (см. рис. 1.17) в поле **Profile name** вводим название стиля, например: `MyStyle`, а из списка выбираем пункт **Eclipse [built-in]**. Нажимаем кнопку **OK**. Откроется окно, в котором можно изменить настройки нашего стиля. На вкладке **Indentation** из списка **Tab policy** выбираем пункт **Spaces only**, а в поля **Indentation size** и **Tab size** вводим число 3. Сохраняем все изменения.

Если необходимо изменить размер шрифта, то в меню **Window** выбираем пункт **Preferences**. В открывшемся окне переходим на вкладку **General | Appearance | Colors and Fonts** (см. рис. 1.18). Из списка выбираем пункт **Java | Java Editor Text Font** и нажимаем кнопку **Edit**.

Редактор Eclipse хорошо работает на Java 11, но некоторые его дополнительные плагины могут работать некорректно. Если после установки плагина редактор не запускается, то нужно явным образом указать путь к более старой версии Java в настройках редактора. Чтобы узнать предпочтительную для редактора версию Java, открываем файл C:\JavaSE\eclipse\eclipse.ini и смотрим значение опции `requiredJavaVersion`. Устанавливаем на компьютер эту версию Java и указываем путь к ней с помощью опции `-vm`. Самый простой способ: создать ярлык для eclipse.exe, отобразить **Свойства** и в поле **Объект** указать такое значение (замените фрагмент `<Путь к JRE>` реальным значением):

```
C:\JavaSE\eclipse\eclipse.exe -vm "<Путь к JRE>\bin\server\jvm.dll"
```

Все способы указания значения подробно описаны на странице **http://wiki.eclipse.org/Eclipse.ini**.

25.3. Компиляция и запуск программы в Java 11

Процесс компиляции и запуска программ в Java 11 почти не отличается от процесса в Java 10. В качестве примера скомпилируем и запустим на выполнение программу из листинга 1.1:

```
C:\Users\Unicross>cd C:\book

C:\book>javac -encoding utf-8 HelloWorld.java

C:\book>java HelloWorld
Hello, world!
```

Начиная с Java 11, программе java.exe можно передать название файла с исходным кодом. В этом случае после названия файла указывается расширение java. Сначала будет автоматически произведена компиляция файла с исходным кодом, а затем запущена программа, и результат отобразится в окне консоли. Формат выполняемой команды:

```
java [<options>] <source file> [<args>]
```

Вот пример запуска программы из листинга 1.1 (предварительно удалите файл HelloWorld.class, созданный предыдущими командами, иначе получите сообщение об ошибке):

```
C:\book>java HelloWorld.java
Hello, world!
```

Новый формат очень удобен для тестирования, т. к. вместо двух команд используется только одна. Однако при этом мы не можем указать различные параметры — например, кодировку файла с исходным кодом. Так что файл с исходным кодом должен быть сохранен в кодировке, используемой в системе по умолчанию. В русской версии Windows по умолчанию используется кодировка windows-1251. Напомню, что в этой книге мы используем для файлов кодировку UTF-8, а не windows-1251.

Перед названием файла с исходным кодом можно указать различные опции, а после названия файла — аргументы, которые будут доступны через параметр `args` в методе `main()`. Вот пример передачи данных программе из листинга 1.4:

```
C:\book>java MyClass.java string1 string2
string1
string2
```

Чтобы увидеть список поддерживаемых опций, выполните такую команду:

```
java --help
```

При использовании нового формата следует учитывать, что файл с расширением class не создается. Если файл с расширением class уже существует, то будет выведено сообщение об ошибке:

```
C:\book>javac -encoding utf-8 MyClass.java

C:\book>java MyClass string1 string2
string1
string2

C:\book>java MyClass.java string1 string2
error: class found on application class path: MyClass
```

OpenJDK автоматически не прописывает ассоциацию для файлов с расширением jar. Если нужно, чтобы исполняемые JAR-архивы запускались путем двойного щелчка на значке архива, то сначала в папке C:\OpenJDK\jdk-11 создаем файл runJAR11.bat (кодировка файла должна быть windows-866) со следующим содержимым:

```
@echo off
start C:\OpenJDK\jdk-11\bin\javaw.exe -jar %1
```

Далее щелкаем правой кнопкой мыши на значке любого JAR-архива и из контекстного меню выбираем пункт **Открыть с помощью** (или **Открыть с помощью | Выбрать программу**). В открывшемся окне устанавливаем флажок **Использовать это приложение для всех файлов jar**. Переходим по ссылке **Дополнительно | Найти другое приложение на этом компьютере** и выбираем программу runJAR11.bat. Теперь при двойном щелчке на значке JAR-архива откроется черное окно, которое сразу закроется, а затем запустится приложение из JAR-архива. Приложение должно быть с оконным интерфейсом, например, как в листинге 16.4, иначе результат выполнения вы не увидите.

25.4. Инструкция *var* в лямбда-выражениях

Начиная с Java 11, слово `var` можно использовать не только для объявления типа локальных переменных (см. *разд. 2.9*), но и для объявления типа параметров в лямбда-выражениях. Это нововведение особенно полезно при использовании аннотаций.

Например, следующее лямбда-выражение:

```
x -> x * 2
```

можно записать так:

```
(var x) -> x * 2
```

Если лямбда-выражение принимает несколько параметров, то при объявлении одного параметра с помощью слова `var` все остальные параметры также должны быть объявлены с помощью слова `var`:

```
(var a, var b) -> a + b
```

Следующие объявления недопустимы:

```
(var a, b) -> a + b
// error: cannot mix 'var' and implicitly-typed parameters
(var a, Integer b) -> a + b
// error: cannot mix 'var' and explicitly-typed parameters
```

Пример использования слова `var` в лямбда-выражениях приведен в листинге 25.1.

Листинг 25.1. Использование слова `var` в лямбда-выражениях

```
import java.util.function.*;
import java.util.stream.IntStream;

public class MyClass {
   public static void main(String[] args) {
      IntStream.rangeClosed(1, 10)
               .map( (var x) -> x * 2 )
               .forEachOrdered( n -> System.out.print(n + " ") );
      // 2 4 6 8 10 12 14 16 18 20
      MyClass.test( (var a, var b) -> a + b, 10, 20 );  // 30
   }
   public static void test(
      BiFunction<Integer, Integer, Integer> f, int x, int y) {
      System.out.println(f.apply(x, y));
   }
}
```

Скомпилируем и запустим пример из командной строки:

```
C:\book>javac -encoding utf-8 MyClass.java

C:\book>java MyClass
2 4 6 8 10 12 14 16 18 20 30
```

25.5. Новые методы в классе *String*

Начиная с Java 11, класс `String` содержит следующие методы:

- `repeat()` — повторяет текущую строку указанное количество раз и возвращает новую строку. Формат метода:

  ```
  public String repeat(int count)
  ```

  Пример:

  ```
  System.out.println("12".repeat(5)); // 1212121212
  ```

- `strip()` — удаляет все пробельные символы из начала и конца строки. Символы проверяются с помощью статического метода `isWhitespace()` из класса `Character`. Формат метода:

  ```
  public String strip()
  ```

Пример:

```
System.out.println("'" + "\n\t\f\r str   ".strip() + "'");
// 'str'
```

- `stripLeading()` — удаляет все пробельные символы из начала строки. Символы проверяются с помощью статического метода `isWhitespace()` из класса `Character`. Формат метода:

```
public String stripLeading()
```

Пример:

```
System.out.println("'" + "   str   ".stripLeading() + "'");
// 'str   '
```

- `stripTrailing()` — удаляет все пробельные символы из конца строки. Символы проверяются с помощью статического метода `isWhitespace()` из класса `Character`. Формат метода:

```
public String stripTrailing()
```

Пример:

```
System.out.println("'" + "   str   ".stripTrailing() + "'");
// '   str'
```

- `isBlank()` — возвращает `true`, если строка пуста или состоит только из пробельных символов. Символы проверяются с помощью статического метода `isWhitespace()` из класса `Character`. Формат метода:

```
public boolean isBlank()
```

Пример:

```
System.out.println("".isBlank());          // true
System.out.println(" \n\t\r  ".isBlank()); // true
System.out.println("str".isBlank());       // false
```

- `lines()` — позволяет получить поток данных с подстроками. Символы-разделители (`\r` и `\n`) подстрок в поток не добавляются. Формат метода:

```
import java.util.stream.Stream;
public Stream<String> lines()
```

Пример:

```
String str = " подстрока1\n подстрока2\r\n подстрока3";
str.lines().forEachOrdered(System.out::println);
/*
 подстрока1
 подстрока2
 подстрока3
*/
```

25.6. Новый метод *of()* в интерфейсе *Path*

Начиная с Java 11, интерфейс `Path` содержит статический метод `of()`. Форматы метода:

```
import java.nio.file.Path;
public static Path of(String first, String... more)
import java.net.URI;
public static Path of(URI uri)
```

Первый формат позволяет указать отдельные компоненты пути в виде строк. Строки будут объединены через символ, используемый для разделения каталогов в операционной системе. Если создать путь не удалось, то генерируется исключение `InvalidPathException`. Пример:

```
Path p = Path.of("C:\\book\\file.txt");
System.out.println(p.toString()); // C:\book\file.txt
p = Path.of("C:", "book", "file.txt");
System.out.println(p.toString()); // C:\book\file.txt
```

Второй формат создает путь на основе объекта класса `URI`:

```
// import java.net.URI;
Path p = Path.of(URI.create("file:/C:/book/file.txt"));
System.out.println(p.toString()); // C:\book\file.txt
```

25.7. Новые методы в классе *Files*

Начиная с Java 11, класс `Files` содержит следующие статические методы:

- `writeString()` — записывает строку в файл. Форматы метода:

```
import java.nio.file.Files;
import java.nio.file.Path;
public static Path writeString(Path path, CharSequence csq,
       OpenOption... options) throws IOException
import java.nio.charset.Charset;
public static Path writeString(Path path, CharSequence csq,
       Charset cs, OpenOption... options) throws IOException
```

  Параметр `path` задает путь к файлу, параметр `csq` — записываемые данные, параметр `cs` — кодировку, а параметр `options` — опции открытия потока. По умолчанию создается новый файл (опция `CREATE`), и поток открывается на запись (опция `WRITE`). Если файл существует, то он будет перезаписан (опция `TRUNCATE_EXISTING`). Если параметр `cs` не указан, то используется кодировка UTF-8 (у меня возникли проблемы при работе с кодировкой UTF-8 в методе `writeString()` — текстовый редактор и метод `readString()` не смогли прочитать записанный этим методом файл).

Пример записи строки в файл в кодировке windows-1251:

```
Path p = Path.of("C:\\book\\file1.txt");
String str = "строка1\nстрока2\n";
System.out.println(
       Files.writeString(p, str, Charset.forName("cp1251")));
// C:\book\file1.txt
```

Если необходимо добавить строку в уже существующий файл, то в параметре `options` следует явным образом указать опцию `APPEND` (если файл не существует, то генерируется исключение):

```
// import java.nio.file.StandardOpenOption;
Path p = Path.of("C:\\book\\file1.txt");
String str = "строка3\nстрока4\n";
System.out.println(
       Files.writeString(p, str, Charset.forName("cp1251"),
                         StandardOpenOption.APPEND));
// C:\book\file1.txt
```

- `readString()` — читает данные из файла в строку. Форматы метода:

```
import java.nio.file.Files;
import java.nio.file.Path;
public static String readString(Path path) throws IOException
import java.nio.charset.Charset;
public static String readString(Path path, Charset cs)
                                    throws IOException
```

Параметр `path` задает путь к файлу, а параметр `cs` — кодировку. Если параметр `cs` не указан, то используется кодировка UTF-8.

Пример чтения из файла в кодировке windows-1251:

```
Path p = Path.of("C:\\book\\file1.txt");
System.out.println(
       Files.readString(p, Charset.forName("cp1251")));
/*
строка1
строка2
строка3
строка4 */
```

25.8. Новый формат метода *toArray()* в интерфейсе *Collection<E>*

В предыдущих главах мы много раз использовали метод `toArray()` из интерфейса `Collection<E>` для преобразования списка, очереди и множества в массив:

```
// import java.util.*;
ArrayList<Integer> arr = new ArrayList<Integer>();
```

```
Collections.addAll(arr, 1, 2, 3);
Integer[] arrInt = arr.toArray(new Integer[0]);
System.out.println(Arrays.toString(arrInt)); // [1, 2, 3]
```

Начиная с Java 11, в интерфейсе `Collection<E>` доступен новый формат метода `toArray()`:

```
default <T> T[] toArray(IntFunction<T[]> generator)
```

Предыдущий пример, используя новый формат, мы можем переписать так:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.addAll(arr, 1, 2, 3);
Integer[] arrInt = arr.toArray(Integer[]::new);
System.out.println(Arrays.toString(arrInt)); // [1, 2, 3]
```

25.9. Новый метод *not()* в интерфейсе *Predicate<T>*

Начиная с Java 11, функциональный интерфейс `Predicate<T>` содержит статический метод `not()`, который инвертирует значение. Формат метода:

```
import java.util.function.Predicate;
public static <T> Predicate<T> not(Predicate<? super T> target)
```

Напомню, что функциональный интерфейс `Predicate<T>` реализует следующий вызов:

```
boolean test(T t)
```

Пример использования метода `not()` приведен в листинге 25.2.

Листинг 25.2. Метод `not()` в интерфейсе `Predicate<T>`

```
package com.example.app;

import java.util.function.*;

public class MyClass {
   public static void main(String[] args) {
      test( n -> n != 0, 10 );                // true
      test( n -> n != 0, 0 );                 // false

      test(Predicate.not(n -> n != 0), 10 ); // false
      test(Predicate.not(n -> n != 0), 0 );  // true
   }
   public static void test(Predicate<Integer> f, int x) {
      System.out.println(f.test(x));
   }
}
```

25.10. Модуль *java.net.http*

Начиная с Java 11, в состав стандартной библиотеки входит модуль `java.net.http`, который содержит классы, позволяющие обмениваться данными по протоколам HTTP 1.1 и 2.0. Прежде чем использовать модуль, нужно прописать зависимость в файле module-info.java:

```
requires java.net.http;
```

Инструкция импорта всех классов:

```
import java.net.http.*;
```

25.10.1. Класс *HttpRequest*: описание параметров запроса

Класс `HttpRequest` описывает все параметры запроса: интернет-адрес (URL), метод отправки данных, версию протокола HTTP (1.1 или 2.0), тайм-аут на чтение и др. Инструкция импорта:

```
import java.net.http.HttpRequest;
```

Создать объект позволяет статический метод `newBuilder()`. Форматы метода:

```
public static HttpRequest.Builder newBuilder()
public static HttpRequest.Builder newBuilder(URI uri)
```

Метод возвращает объект, реализующий интерфейс `HttpRequest.Builder`. Почти все методы этого интерфейса возвращают ссылку на текущий объект. Это позволяет строить цепочку из методов, задающих различные параметры запроса. После указания всех параметров нужно вызвать метод `build()`, который возвращает объект класса `HttpRequest`:

```
HttpRequest request = HttpRequest.newBuilder()
                                 .uri(URI.create("http://localhost"))
                                 .GET()
                                 .build();
```

Пример использования класса `HttpRequest` приведен в листинге 25.3.

Листинг 25.3. Класс `HttpRequest`

```
package com.example.app;

import java.net.URI;
import java.net.http.*;
import java.time.Duration;
import java.util.Optional;

public class MyClass {
   public static void main(String[] args) {
```

```
        HttpRequest request = HttpRequest.newBuilder()
                              // URL-адрес
                              .uri(URI.create("http://localhost"))
                              // Тайм-аут на чтение
                              .timeout(Duration.ofSeconds(10))
                              // Версия протокола HTTP
                              .version(HttpClient.Version.HTTP_1_1)
                              // Добавление заголовков запроса
                              .header("User-Agent", "MySpider/1.0")
                              // Создание объекта
                              .build();
        // Получение значений
        System.out.println(request);          // http://localhost GET
        System.out.println(request.uri());    // http://localhost
        System.out.println(request.method()); // GET
        Optional<Duration> timeout = request.timeout();
        System.out.println(
               timeout.orElse(Duration.ZERO).toSeconds()); // 10
        Optional<HttpClient.Version> version = request.version();
        System.out.println(version.orElse(null)); // HTTP_1_1
        HttpHeaders headers = request.headers();
        System.out.println(headers);
        // java.net.http.HttpHeaders@a6b4f017
        // { {User-Agent=[MySpider/1.0]} }
        System.out.println(headers.allValues("User-Agent").get(0));
        // MySpider/1.0
    }
}
```

25.10.2. Класс *HttpClient*: отправка запроса и получение ответа сервера

Класс `HttpClient` описывает Web-браузер, позволяющий отправить запрос и получить ответ сервера. Можно настроить версию протокола HTTP (1.1 или 2.0), установить тайм-аут на соединение, задать способ обработки перенаправлений, указать параметры аутентификации, использовать прокси-сервер и др.

Инструкция импорта:

```
import java.net.http.HttpClient;
```

Создать объект с настройками по умолчанию позволяет статический метод `newHttpClient()`. Формат метода:

```
public static HttpClient newHttpClient()
```

Пример:

```
HttpClient client = HttpClient.newHttpClient();
```

Статический метод `newBuilder()` предназначен для создания объекта с произвольными параметрами. Формат метода:

```
public static HttpClient.Builder newBuilder()
```

Метод возвращает объект, реализующий интерфейс `HttpClient.Builder`. Почти все методы этого интерфейса возвращают ссылку на текущий объект. Это позволяет строить цепочку из методов, задающих различные параметры. После указания всех параметров нужно вызвать метод `build()`, который возвращает объект класса `HttpClient`:

```
HttpClient client = HttpClient.newBuilder()
                        .version(HttpClient.Version.HTTP_1_1)
                        .connectTimeout(Duration.ofSeconds(30))
                        .build();
```

Чтобы отправить запрос, следует воспользоваться методами `send()` (синхронный запрос) или `sendAsync()` (асинхронный запрос). При синхронном запросе текущий поток будет блокирован до момента получения ответа сервера. При использовании метода `sendAsync()` запрос выполняется в отдельном потоке. Форматы методов:

```
public abstract <T> HttpResponse<T> send(HttpRequest request,
                HttpResponse.BodyHandler<T> responseBodyHandler)
                       throws IOException, InterruptedException
public abstract <T> CompletableFuture<HttpResponse<T>> sendAsync(
                HttpRequest request,
                HttpResponse.BodyHandler<T> responseBodyHandler)
public abstract <T> CompletableFuture<HttpResponse<T>> sendAsync(
                HttpRequest request,
                HttpResponse.BodyHandler<T> responseBodyHandler,
                HttpResponse.PushPromiseHandler<T> pushPromiseHandler)
```

В первом параметре методы принимают объект запроса. Во втором параметре указывается объект, реализующий интерфейс `HttpResponse.BodyHandler<T>`, который определяет способ обработки тела ответа сервера. Готовые реализации этого интерфейса находятся в классе `HttpResponse.BodyHandlers`. Инструкция импорта:

```
import java.net.http.HttpResponse;
```

Если тело ответа должно быть преобразовано в строку (в объект класса `String`), то следует воспользоваться статическим методом `ofString()`. Форматы метода:

```
public static HttpResponse.BodyHandler<String> ofString()
public static HttpResponse.BodyHandler<String> ofString(Charset charset)
```

Если кодировка не указана, то будет использоваться кодировка из заголовка `Content-Type`, а при его отсутствии — UTF-8.

Если кодировка в заголовке ответа сервера не соответствует кодировке данных, то данные могут быть испорчены. Чтобы о кодировке заботиться самим, следует воспользоваться статическим методом `ofByteArray()`. Формат метода:

```
public static HttpResponse.BodyHandler<byte[]> ofByteArray()
```

Полученные данные могут иметь очень большой размер. Чтобы иметь возможность читать фрагментами — например, построчно — можно воспользоваться статическим методом `ofInputStream()`. Формат метода:

```
public static HttpResponse.BodyHandler<InputStream> ofInputStream()
```

Существуют и другие способы получения данных — например, в виде файла. Полный список методов смотрите в документации.

Пример использования класса `HttpClient` приведен в листинге 25.4. Предварительно следует запустить сервер Apache (см. *главу 22*) и добавить в файл C:\xampp\htdocs\index.php какой-либо текст или код. Например, для просмотра значений всех переменных сервера можно вставить в файл следующий код:

```
<?php
print_r($_SERVER);
?>
```

Среди полученных значений можно увидеть, какие данные отправляются серверу. Например, Web-браузер по умолчанию называет себя `Java-http-client/11`:

```
[HTTP_USER_AGENT] => Java-http-client/11
```

Чтобы использовать свое собственное название, нужно при создании объекта запроса добавить заголовок `User-Agent`:

```
HttpRequest request = HttpRequest.newBuilder()
                                 .uri(URI.create("http://localhost"))
                                 .header("User-Agent", "MySpider/1.0")
                                 .build();
```

Результат:

```
[HTTP_USER_AGENT] => MySpider/1.0
```

Листинг 25.4. Класс `HttpClient`

```
package com.example.app;

import java.io.IOException;
import java.net.URI;
import java.net.http.*;
import java.time.Duration;
import java.util.Optional;

public class MyClass {
   public static void main(String[] args) {
      HttpRequest request = HttpRequest.newBuilder()
          .uri(URI.create("http://localhost"))
          .build();
      // Создание объекта
      HttpClient client = HttpClient.newBuilder()
```

```
            // Версия протокола (по умолчанию HTTP_2)
            .version(HttpClient.Version.HTTP_1_1)
            // Обработка перенаправлений
            .followRedirects(HttpClient.Redirect.NORMAL)
            // Тайм-аут соединения
            .connectTimeout(Duration.ofSeconds(30))
            // Создание объекта
            .build();
        // Получение значений
        HttpClient.Version version = client.version();
        System.out.println(version); // HTTP_1_1
        HttpClient.Redirect redirect = client.followRedirects();
        System.out.println(redirect); // NORMAL
        Optional<Duration> timeout = client.connectTimeout();
        System.out.println(
               timeout.orElse(Duration.ZERO).toSeconds()); // 30
        // Отправка запроса и обработка ответа
        try {
           HttpResponse<String> response = client.send(
                        request, HttpResponse.BodyHandlers.ofString());
           if (response.statusCode()>=200 && response.statusCode()<300) {
              System.out.println("Ответ успешно получен");
              System.out.println(response.body());
              System.out.println(response.headers());
           }
        } catch (IOException e) {
           System.out.println("IOException: " + e.getMessage());
        } catch (InterruptedException e) {
           System.out.println("InterruptedException: " + e.getMessage());
        }
    }
}
```

25.10.3. Интерфейс *HttpResponse<T>*: обработка ответа сервера

Метод `send()` возвращает объект, реализующий интерфейс `HttpResponse<T>`. С помощью этого объекта можно получить заголовки ответа сервера, а также данные. Интерфейс `HttpResponse<T>` содержит следующие основные методы (полный список смотрите в документации):

- `statusCode()` — позволяет получить код возврата сервера. Формат метода:

  ```
  int statusCode()
  ```

- `body()` — возвращает данные. Тип данных зависит от параметра `responseBodyHandler` в методах `send()` и `sendAsync()`. Формат метода:

  ```
  T body()
  ```

- `headers()` — позволяет получить заголовки ответа сервера в виде объекта класса `HttpHeaders`. Формат метода:

```
import java.net.http.HttpHeaders;
HttpHeaders headers()
```

25.10.4. Отправка запроса методом *GET*

Отправка запроса возможна различными методами — например: `GET`, `POST` и др. Для указания способа передачи используются одноименные методы при создании объекта запроса. При отправке запроса методом `GET` можно указать метод `GET()`, однако это делать необязательно, т. к. метод `GET` используется по умолчанию. Формат метода:

```
HttpRequest.Builder GET()
```

Определить способ отправки запроса позволяет метод `method()` из класса `HttpRequest`. Формат метода:

```
public abstract String method()
```

Пример:

```
HttpRequest request = HttpRequest.newBuilder()
                                 .uri(URI.create("http://localhost"))
                                 .GET()
                                 .build();
System.out.println(request.method()); // GET
```

При использовании метода `GET` передаваемые данные указываются в составе URL-адреса после вопросительного знака. Формат данных:

```
<Имя1>=<Значение1>&...&<ИмяN>=<ЗначениеN>
```

Все специальные символы должны быть закодированы. Для кодирования можно использовать статический метод `encode()` из класса `URLEncoder` или методы класса `URI`.

Давайте изменим содержимое файла C:\xampp\htdocs\index.php следующим образом:

```
<?php
print_r($_GET);
?>
```

Теперь выполним запрос методом `GET` с указанием заголовков и обработаем результат, если код ответа сервера от `200` до `300` (не включая этот код) (листинг 25.5). Данные будем получать в виде массива байтов с последующим преобразованием в строку.

Листинг 25.5. Отправка запроса методом `GET`

```
package com.example.app;

import java.io.IOException;
import java.net.*;
```

```
import java.net.http.*;
import java.nio.charset.StandardCharsets;
import java.time.Duration;

public class MyClass {
   public static void main(String[] args) throws Exception {
      String uri = "http://localhost/index.php?color1=" +
            URLEncoder.encode("красный", "utf-8") + "&color2=" +
            URLEncoder.encode("белый", "utf-8");
      HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create(uri))
            .version(HttpClient.Version.HTTP_1_1)
            .timeout(Duration.ofSeconds(5))
            .header("User-Agent", "MySpider/1.0")
            .build();
      HttpClient client = HttpClient.newBuilder()
            .version(HttpClient.Version.HTTP_1_1)
            .followRedirects(HttpClient.Redirect.NORMAL)
            .connectTimeout(Duration.ofSeconds(5))
            .build();
      // Отправка запроса и обработка ответа
      try {
         HttpResponse<byte[]> response = client.send(
             request, HttpResponse.BodyHandlers.ofByteArray());
         if (response.statusCode()>=200 && response.statusCode()<300) {
            System.out.println(new String(response.body(),
                                          StandardCharsets.UTF_8));
         }
         else {
            System.out.println("Код: " + response.statusCode());
         }
      } catch (IOException e) {
         System.out.println("IOException: " + e.getMessage());
      } catch (InterruptedException e) {
         System.out.println("InterruptedException: " + e.getMessage());
      }
   }
}
```

Если все выполнено правильно, то получим следующий результат:

```
Array
(
    [color1] => красный
    [color2] => белый
)
```

25.10.5. Отправка запроса методом *POST*

При передаче данных методом `POST` необходимо выполнить дополнительные действия:

- при создании объекта запроса вызвать метод `POST()` и передать ему способ передачи данных в виде объекта, реализующего интерфейс `HttpRequest.BodyPublisher`. Формат метода:

```
HttpRequest.Builder POST(HttpRequest.BodyPublisher bodyPublisher)
```

  Различные реализации интерфейса `HttpRequest.BodyPublisher` можно найти в классе `HttpRequest.BodyPublishers`. Например, чтобы отправить массив байтов, нужно использовать метод `ofByteArray()`. Форматы метода:

```
public static HttpRequest.BodyPublisher ofByteArray(byte[] buf)
public static HttpRequest.BodyPublisher ofByteArray(byte[] buf,
                          int offset, int length)
```

- закодировать данные и преобразовать их в массив байтов:

```
String s = "color1=" +
       URLEncoder.encode("красный", "utf-8") + "&color2=" +
       URLEncoder.encode("белый", "utf-8");
byte[] bytes = s.getBytes();
```

- добавить заголовок `Content-Type` (заголовок `Content-Length` с длиной массива байтов будет добавлен автоматически). Для передачи данных формы нужно указать значение `application/x-www-form-urlencoded`:

```
.header("Content-Type", "application/x-www-form-urlencoded")
```

Давайте изменим содержимое файла C:\xampp\htdocs\index.php следующим образом:

```
<?php
print_r($_POST);
?>
```

Теперь выполним запрос методом `POST` с указанием заголовков и обработаем результат (листинг 25.6).

Листинг 25.6. Отправка запроса методом POST

```
package com.example.app;

import java.io.IOException;
import java.net.*;
import java.net.http.*;
import java.nio.charset.StandardCharsets;
import java.time.Duration;

public class MyClass {
   public static void main(String[] args) throws Exception {
```

```
        String s = "color1=" +
               URLEncoder.encode("красный", "utf-8") + "&color2=" +
               URLEncoder.encode("белый", "utf-8");
        byte[] bytes = s.getBytes();
        HttpRequest request = HttpRequest.newBuilder()
               .uri(URI.create("http://localhost"))
               .version(HttpClient.Version.HTTP_1_1)
               .POST(HttpRequest.BodyPublishers.ofByteArray(bytes))
               .timeout(Duration.ofSeconds(5))
               .header("Content-Type",
                       "application/x-www-form-urlencoded")
               .header("User-Agent", "MySpider/1.0")
               .build();
        HttpClient client = HttpClient.newBuilder()
               .version(HttpClient.Version.HTTP_1_1)
               .followRedirects(HttpClient.Redirect.NORMAL)
               .connectTimeout(Duration.ofSeconds(5))
               .build();
        // Отправка запроса и обработка ответа
        try {
           HttpResponse<byte[]> response = client.send(
               request, HttpResponse.BodyHandlers.ofByteArray());
           if (response.statusCode()>=200 && response.statusCode()<300) {
              System.out.println(new String(response.body(),
                                            StandardCharsets.UTF_8));
           }
           else {
              System.out.println("Код: " + response.statusCode());
           }
        } catch (IOException e) {
           System.out.println("IOException: " + e.getMessage());
        } catch (InterruptedException e) {
           System.out.println("InterruptedException: " + e.getMessage());
        }
    }
}
```

Если все выполнено правильно, то получим следующий результат:

```
Array
(
    [color1] => красный
    [color2] => белый
)
```

Заключение

Вот и закончилось наше путешествие в мир Java. Материал книги описывает лишь основы этой замечательной технологии. А здесь мы уточним, где найти дополнительную информацию, чтобы продолжить изучение языка.

Самым важным источником информации является официальный сайт компании Oracle: **http://www.oracle.com/technetwork/java/index.html**. На этом сайте вы найдете дистрибутивы, новости, а также ссылки на все другие ресурсы в Интернете, посвященные Java.

На сайте **http://docs.oracle.com/en/java/** расположена документация по Java, которая обновляется в режиме реального времени. Язык постоянно совершенствуется, появляются новые классы, изменяются параметры, добавляются пакеты и т. д. Регулярно посещайте этот сайт, и вы будете в курсе самых свежих новшеств Java.

На странице Википедии: **https://ru.wikipedia.org/wiki/Java** вы найдете историю развития языка Java и также действующие ссылки на все другие ресурсы в Интернете, посвященные этому языку.

Начиная с Java 8, в состав JDK входит поддержка JavaFX. С помощью этой технологии можно создавать оконные и мобильные приложения. В Java 11 библиотеку JavaFX удалили из JDK, но ее можно скачать с сайта **https://openjfx.io/**.

Не следует также забывать, что на языке Java можно разрабатывать мобильные приложения под Android, — достаточно установить на компьютер Android SDK. Подробную информацию по этому вопросу можно найти на сайте: **https://developer.android.com/**.

Если в процессе изучения языка у вас возникнут какие-либо недопонимания, то не следует забывать, что Интернет предоставляет множество ответов на самые разнообразные вопросы. Достаточно в строке запроса поискового портала (например, **http://www.google.com/**) набрать свой вопрос. Наверняка уже кто-то сталкивался с подобной проблемой и описал ее решение на каком-либо сайте.

Свои замечания и пожелания вы можете оставить на странице книги на сайте издательства «БХВ-Петербург»: **http://www.bhv.ru/**. Все замеченные опечатки и неточности прошу присылать на E-mail: **mail@bhv.ru** — не забудьте только указать название книги и имя автора.

ПРИЛОЖЕНИЕ

Описание электронного архива

Электронный архив к книге выложен на FTP-сервер издательства по адресу: **ftp://ftp.bhv.ru/9785977540124.zip**. Ссылка доступна и со страницы книги на сайте **www.bhv.ru**.

Структура архива представлена в табл. П.1.

Таблица П.1. *Структура электронного архива*

Файл	Описание
Listings.doc	Содержит все пронумерованные листинги из книги, а также некоторые полезные фрагменты кода
Readme.txt	Описание электронного архива

Предметный указатель

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

www.bhv.ru

Прохоренок Н.

OpenCV и Java. Обработка изображений и компьютерное зрение

Отдел оптовых поставок:
e-mail: opt@bhv.spb.su

Взгляни на мир глазами робота

- Загрузка изображения из файла
- Захват кадров с веб-камеры
- Трансформация изображения
- Применение фильтров
- Сегментация изображения
- Выделение границ объектов
- Поиск и сравнение контуров
- Поиск объекта по цвету или шаблону
- Поиск и сравнение особых точек

Книга знакомит с современными технологиями компьютерного зрения, позволяющими машинам, роботам, веб-камерам и другим устройствам распознавать изображения. Приведено описание библиотеки компьютерного зрения OpenCV применительно к языку программирования Java. Прочитав книгу, вы научитесь загружать и сохранять изображения в различных форматах, захватывать кадры с веб-камеры в режиме реального времени, выполнять обработку, трансформацию и сегментацию изображения, применять к изображению фильтры. На практических примерах рассматриваются алгоритмы компьютерного зрения, предназначенные для обнаружения, классификации и отслеживания объектов. Вы научитесь выделять границы и контуры объектов, выполнять поиск объектов по шаблону, особым точкам, цвету или обученному классификатору.

Пользоваться библиотекой OpenCV в Java просто и очень эффективно! Для понимания материала книги потребуется всего лишь владение основами языка программирования Java Standard Edition и знание математики на уровне средней школы.

Прохоренок Николай Анатольевич, профессиональный программист, автор книг «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Python 3. Самое необходимое», «Python 3 и PyQt 5. Разработка приложений», «Основы Java» и др.

www.bhv.ru

Прохоренок Н., Дронов В.

Python 3 и PyQt 5. Разработка приложений, 2-е изд.

Отдел оптовых поставок

E-mail: opt@bhv.spb.su

Быстрое создание приложений с графическим интерфейсом

- Описание языка Python
- Объектно-ориентированное программирование
- Работа с файлами и каталогами
- Создание оконных приложений
- Работа с базами данных
- Мультимедиа
- Печать и экспорт в формат PDF
- Взаимодействие с Windows
- Сохранение настроек приложений
- Работающий пример: приложение «Судоку»

Если вы хотите научиться программировать на языке Python 3 и создавать приложения с графическим интерфейсом, эта книга для вас. В первой части книги описан базовый синтаксис языка Python 3: типы данных, операторы, условия, циклы, регулярные выражения, функции, инструменты объектно-ориентированного программирования, часто используемые модули стандартной библиотеки. Вторая часть книги посвящена библиотеке PyQt 5, позволяющей создавать приложения с графическим интерфейсом на языке Python 3. Рассмотрены средства для обработки сигналов, управления свойствами окна, разработки многопоточных приложений, описаны основные компоненты (кнопки, текстовые поля, списки, таблицы, меню, панели инструментов и др.), варианты их размещения внутри окна, инструменты для работы с базами данных, мультимедиа, вывода документов на печать и экспорта их в формате Adobe PDF, взаимодействия с Windows и сохранения настроек приложений.

Книга содержит большое количество практических примеров, помогающих начать программировать на языке Python самостоятельно. А в конце книги описывается процесс разработки приложения, предназначенного для создания и решения головоломок судоку. Весь материал тщательно подобран, хорошо структурирован и компактно изложен, что позволяет использовать книгу как удобный справочник.

Прохоренок Николай Анатольевич, профессиональный программист, имеющий большой практический опыт создания и продвижения динамических сайтов с использованием HTML, JavaScript, PHP, Perl и MySQL. Автор книг «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Разработка Web-сайтов с помощью Perl и MySQL», «Python. Самое необходимое», «Python 3 и PyQt. Разработка приложений» и др.

Дронов Владимир Александрович, профессиональный программист, писатель и журналист, работает с компьютерами с 1987 года. Автор более 20 популярных компьютерных книг, в том числе «Django: практика создания Web-сайтов на Python», «Laravel. Быстрая разработка современных динамических Web-сайтов на PHP, MySQL, HTML и CSS», «Angular 4. Быстрая разработка сверхдинамических Web-сайтов на TypeScript и PHP» и книг по продуктам Adobe Flash и Adobe Dreamweaver различных версий. Его статьи публикуются в журналах «Мир ПК» и «ИнтерФейс» (Израиль) и интернет-порталах «IZ City» и «TheVista.ru».

Основы Java

2-е издание

Просто о сложном

Если вы хотите научиться программировать на языке Java, то эта книга для вас. В книге описан базовый синтаксис языка Java: типы данных, операторы, условия, циклы, регулярные выражения, лямбда-выражения, ссылки на методы, объектно-ориентированное программирование. Рассмотрены основные классы стандартной библиотеки, получение данных из сети Интернет, работа с базой данных MySQL. Во втором издании приводится описание большинства нововведений: модули, интерактивная оболочка JShell, инструкция var и др.

Прохоренок Николай Анатольевич, профессиональный программист, автор книг «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Python 3. Самое необходимое», «Python 3 и PyQt 5. Разработка приложений», «OpenCV и Java. Обработка изображений и компьютерное зрение» и др.

Книга содержит большое количество практических примеров, помогающих начать программировать на языке Java самостоятельно. Весь материал тщательно подобран, хорошо структурирован и компактно изложен, что позволяет использовать книгу как удобный справочник.

Примеры из книги можно скачать по ссылке ftp://ftp.bhv.ru/9785977540124.zip, а также со страницы книги на сайте www.bhv.ru.

ISBN 978-5-9775-4012-4

191036, Санкт-Петербург, Гончарная ул., 20

Тел.: (812) 717-10-50, 339-54-17, 339-54-28

E-mail: mail@bhv.ru

Internet: www.bhv.ru

bhv®