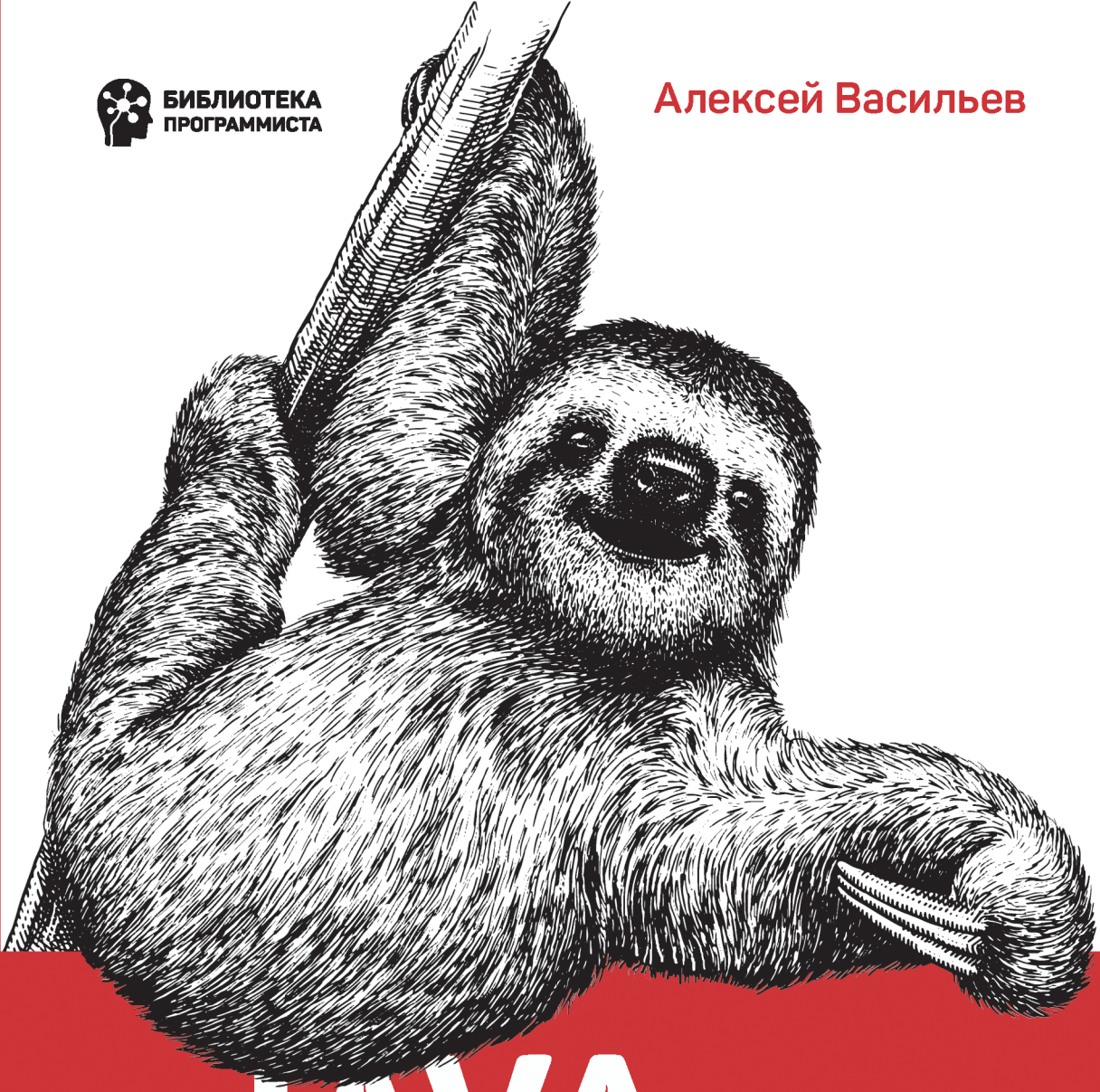
БИБЛИОТЕКА ПРОГРАММИСТА

Алексей Васильев



JAVA ДЛЯ ВСЕХ

КНИГА О ПОПУЛЯРНОМ ЯЗЫКЕ JAVA
ОТ АВТОРА КОМПЬЮТЕРНЫХ БЕСТСЕЛЛЕРОВ

Алексей Васильев

JAVA ДЛЯ ВСЕХ

КНИГА О ПОПУЛЯРНОМ ЯЗЫКЕ JAVA
ОТ АВТОРА КОМПЬЮТЕРНЫХ БЕСТСЕЛЛЕРОВ

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2020

ББК 32.973.2-018.1
УДК 004.43
В19

Васильев Алексей

В19 Java для всех. — СПб.: Питер, 2020. — 512 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1382-8

Java — один из самых популярных и востребованных языков в мире. Алексей Васильев — автор многочисленных компьютерных бестселлеров — познакомит вас со всем необходимым для эффективной работы с этим языком. Вы изучите базовые типы, управляющие инструкции, особенности описания классов и объектов в Java, создание интерфейсов, лямбда-выражения, обобщенные классы. Каждая глава содержит примеры кода, которые в свою очередь снабжены как построчными пояснениями, так и подробным разбором примера программы. Примеры, используемые в этой книге, пригодятся вам в дальнейшей работе с языком Java. Программирование — это нестрашно! Даже если у вас нет никакого опыта, вы с легкостью освоите Java, воспользовавшись уникальной методикой Алексея Васильева, и перейдете на профессиональный уровень.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-5-4461-1382-8

Оглавление

Вступление

О книге и не только

Не надо громких слов. Они потрясают воздух, но не собеседника.

из к/ф «Формула любви»

Вот уже многие годы язык Java входит в число самых популярных и востребованных. Он красивый, эффектный и, самое главное, очень производительный. Но, к сожалению, не самый простой. Именно поэтому спрос на Java-программистов неизменно высок. Язык Java — это бриллиант, который украсит багаж знаний любого программиста. А учить Java, как, я надеюсь, предстоит убедиться читателям этой книги, не только полезно, но и интересно. В основу книги положены курсы лекций, в разное время прочитанные мной для магистров физического факультета Киевского национального университета им. Тараса Шевченко, бакалавров медико-инженерного факультета Национального технического университета Украины «Киевский политехнический институт» и слушателей различных курсов по программированию. Материал книги и способ изложения адаптированы для всех, кто желает изучать Java не только в учебных заведениях, но и самостоятельно. Поэтому книга может использоваться и как самоучитель.

Объектно-ориентированное программирование и Java

— На что жалуемся?
— На голову жалуется.

из к/ф «Формула любви»

Есть несколько концепций программирования, которые в той или иной степени реализуются в разных языках программирования. Среди них особым образом выделяется концепция *объектно-ориентированного программирования* (ООП),

которая поддерживается всеми (или почти всеми) современными языками программирования. Концепция ООП — это реакция на значительное усложнение программ и увеличение объема кода. ООП преимущественно используют при написании больших и сложных программ.

НА ЗАМЕТКУ

Программа считается большой, если она содержит несколько тысяч строк кода. Понятно, что это очень условная оценка, которая, однако, позволяет составить представление о том, что такое много и мало.

Визитная карточка ООП — *классы* и *объекты*. Но это, если угодно, только вершина айсберга. В действительности ООП представляет собой стройную, элегантную концепцию, которая при умелом применении позволяет значительно облегчить работу над большими серьезными проектами. Вместе с тем изучение ООП требует известных усилий, в идеале — базирующихся на опыте программирования обучающегося. Поэтому обычно к изучению ООП приступают постепенно, отталкиваясь от традиционных подходов.

ПОДРОБНОСТИ

Концепция ООП базируется на трех фундаментальных принципах: *инкапсуляции*, *полиморфизме* и *наследовании*.

Инкапсуляция подразумевает объединение в одно целое данных и кода для обработки этих данных. Обычно инкапсуляция реализуется через использование классов и объектов. В разных языках программирования все происходит по-разному.

Полиморфизм базируется на использовании универсальных интерфейсов для решения однотипных задач. Как мы узнаем далее, этот механизм в Java реализуется за счет перегрузки и переопределения методов.

Наследование позволяет создавать новые классы на основе уже существующих классов, что значительно экономит усилия, сокращает объем кода и повышает его надежность.

Еще раз подчеркнем, что эти три принципа в той или иной степени присущи любому языку программирования, поддерживающему парадигму ООП.

Наиболее популярными объектно-ориентированными языками программирования сегодня принято считать, наряду с Java, языки C++ и C#. Исторически первым появился язык C++, ставший существенно усовершенствованной версией C. Усовершенствования касались главным образом поддержки парадигмы ООП. Именно C++ стал в известном смысле родительским для языков C# и Java. В этом несложно убедиться, если сравнить синтаксисы языков — они очень схожи. Но если в языке C++ можно создавать программы как с использованием классов, так и без них, то в языках Java и C# без ООП уже не обойтись.

Язык Java изначально предназначался для написания больших и сложных программ. Поэтому естественно, что Java в полной мере поддерживает концепцию ООП, являясь полностью объектно-ориентированным языком. Это означает, что любая программа, написанная на языке Java, должна содержать описание по крайней мере одного класса. Получается, что, изучая Java с нуля, приходится сразу окунаться в парадигму ООП, а это не так просто (особенно для тех, у кого нет опыта программирования). Пониманию парадигмы ООП поможет знакомство с наиболее характерными особенностями Java.

НА ЗАМЕТКУ

Что касается трудностей, то впадать в отчаяние не стоит: мы найдем способ обойти все подводные камни и решим все проблемы.

Особенности языка Java

Русская речь не сложнее других. Вон Маргадон — дикий человек, и то выучил.

из к/ф «Формула любви»

Язык Java был разработан (с 1991 по 1995 год) инженерами компании Sun Microsystems, которую затем поглотила корпорация Oracle (www.oracle.com). Именно Oracle теперь и отвечает за поддержку технологии Java.

Исторически стимулом к созданию Java (если точнее, то речь шла о разработке целой технологии) стало желание получить программные средства для работы с бытовыми приборами. Здесь на первый план выходит проблема универсальности программных кодов. И так совпало, что как раз в это время начали бурно развиваться интернет-технологии. Подход, использованный в Java, во многом идентичен подходу, примененному при разработке программ, предназначенных для работы во Всемирной паутине. Именно это обстоятельство и способствовало быстрому росту популярности Java (неформальный девиз Java звучит как «написано однажды — работает везде»).

НА ЗАМЕТКУ

Если вы создаете исходный код, заранее зная, на каком компьютере и с какой операционной системой он будет выполняться, то у вас есть возможность оптимизировать программу под параметры исполнительной среды. Если же вы пишете программу, предназначенную для использования в Интернете, то заранее не знаете ни тип процессора, ни тип операционной системы, которые использует конечный потребитель программного продукта. Особенностью интернет-среды является принципиальное разнообразие используемых операционных систем и аппаратного обеспечения. Отсюда — требование к универсальности кодов.

Проблема универсальности программных кодов решена в Java в рамках концепции *виртуальной Java-машины* (JVM, от *Java Virtual Machine*). Так, если обычно при компиляции программы (например, написанной на C++) на выходе мы получаем исполнительный машинный код, то в результате компиляции Java-программы получают промежуточный байт-код, который выполняется не операционной системой, а виртуальной Java-машиной, представляющей собой специальную программу. Разумеется, предварительно виртуальная Java-машина должна быть установлена на компьютер пользователя. С одной стороны, это позволяет создавать достаточно универсальные программы (в том смысле, что они могут использоваться с разными операционными системами). Но с другой стороны, платой за универсальность является снижение скорости выполнения программ.

Кроме того, следует четко понимать, что язык Java создавался именно для написания больших и сложных программ. Писать на Java консольные программы, которые выводят сообщения вроде "Hello, world!", — это все равно что на крейсере отправиться на ловлю карасей. Тем не менее Java позволяет решать и такие примитивные задачи. К слову, большинство примеров в книге — простые программные коды, и в данном случае это оправданно, поскольку в учебе хороши любые приемы — главное, чтобы они были эффективными.

Java и другие языки программирования

> Обо мне придумано столько небылиц, что я устаю их опровергать.
>
> *из к/ф «Формула любви»*

Несмотря на популярность, у языка Java есть конкуренты. Главные из них — языки C++ и C#. Далее следует сказать несколько слов о том, как эти языки возникли и развивались.

НА ЗАМЕТКУ

Кроме тройки лидеров (C++, C# и Java), есть и другие претенденты, которые периодически оказываются на вершине (или очень близко к вершине) списка популярности языков. На сегодняшний день очень быстро набирают высоту такие языки, как JavaScript и Python. Но у них немного иная концепция, и поэтому мы их обсуждать не будем. А вот C++, C# и Java относятся к семейству языков, которые берут свое начало от старого доброго C — одного из первых и самых удачных языков программирования.

На заре программирования законодателем моды был язык C — язык для программистов-профессионалов. Благодаря простому и лаконичному синтаксису он и сегодня имеет большую армию поклонников. Тем не менее время и технологии

бросили вызов даже такому продуманному и эффективному языку, как C. Объем программных кодов постоянно увеличивается. В какой-то момент стало понятно, что для эффективного программирования необходимо менять базовые подходы. Так на смену парадигме *процедурного программирования* пришла концепция ООП. Но в исходной своей версии язык C не был рассчитан на поддержку ООП. В результате на свет появился язык C++ — расширение языка C для поддержки парадигмы ООП. Затем был разработан язык Java, синтаксис которого внешне во многом напоминает синтаксис C++. Одновременно разработчики Java постарались учесть и устранить недостатки C++ (а недостатки есть у любого языка). Поскольку Java ориентирован на использование в Интернете, то упор был сделан на безопасность. Универсальность и безопасность — вот два вектора, определившие развитие языка.

Язык C# разрабатывался компанией Microsoft после появления языка Java. В каком-то смысле C# является ответом компании Microsoft на успех Java. В языке C# постарались учесть и исправить не только недоработки C++, но и изъяны Java. Как и Java, C# является полностью объектно-ориентированным языком. Как и в Java, при компиляции программ, написанных на C#, получают промежуточный код. Но между Java и C# есть и принципиальные отличия. Язык C# является базовым для технологии .Net Framework. В рамках этой технологии код, написанный на разных языках (не только C#), компилируется в промежуточный код, а на уровне промежуточного кода объединяется в один проект. Хотя внешне (и в плане синтаксиса, и на техническом уровне) языки похожи, идеологически они совершенно разные.

НА ЗАМЕТКУ

То, что базовые конструкции в языках C++, C# и Java однотипны, — не случайно и не удивительно. Дело в том, что при создании нового языка важно, чтобы он стал популярным. Один из ключевых факторов, определяющих легкость перехода на новый язык, — простота или узнаваемость синтаксиса. Поскольку язык C задал общий стандарт для языков программирования, то логично, что новые языки тяготеют к тому, чтобы сохранить знакомые большому количеству программистов синтаксические конструкции. И здесь важно понимать, что наличие таких сходств совершенно не означает одинаковости языков. Каждый язык уникален и имеет свои преимущества и недостатки.

Попробуем сделать общее сравнение языка Java с языками C++ и C#. Такое сравнение в первую очередь интересно для тех, кто программирует на C++ и/или C#.

Как отмечалось, основные синтаксические конструкции (базовые операторы и управляющие инструкции) во всех трех упоминаемых по тексту языках довольно схожи. Одно из первых проявлений непохожести языков — это реализация массивов. В отличие от C++, в Java (и C#) все массивы являются динамическими с автоматической проверкой на предмет выхода за пределы массива. В Java, если известно имя массива, можно узнать и его размер. Более того, в Java, в отличие

от C++ и C#, нет как таковых указателей. Скажем, в C++ имя массива является указателем на его первый элемент. В Java (и в C#) доступ к массиву получают через переменную, которая ссылается на массив. Таким образом, в Java переменная массива и сам массив — совсем не одно и то же. И хотя может показаться, что это неудобно, на практике все выглядит иначе и данный подход имеет неоспоримые преимущества.

Аналогичная ситуация имеет место и с объектами. Все объекты в Java и C# создаются динамически, а объектная переменная содержит ссылку на объект. При присваивании объектов ссылка с одного объекта перебрасывается на другой объект. Это обстоятельство, как увидят читатели, имеет важные последствия.

В языке C++ основной (но не единственный) способ реализации текстовых значений связан с использованием символьных массивов. В Java для работы с текстом есть несколько библиотечных классов, которые позволяют выполнять все текстовые операции быстро, просто и надежно (нечто подобное реализовано и в C#). А вот чего в Java нет, так это перегрузки операторов. И надо откровенно признать, что это одно из упущений разработчиков языка.

В Java, как и в C#, нет множественного наследования: если в языке C++ у класса при наследовании может быть несколько базовых классов, то в Java может наследоваться только один класс. Вместо множественного наследования в Java используются интерфейсы. Причем с учетом нововведений в версии Java 8 (наличие в интерфейсах кода по умолчанию для методов) потери сведены к минимуму.

Обобщенные типы в Java реализованы значительно скромнее по сравнению с C++ и даже C#. Зато в Java есть эффективная система поддержки многопоточного программирования и обработки ошибок. К неоспоримым преимуществам языка Java можно отнести и фундаментальную поддержку для разработки приложений с графическим интерфейсом.

Java — красивый, надежный и эффективный язык со своими преимуществами и недостатками, с которыми нам предстоит познакомиться.

Программное обеспечение

> От пальца не прикуривают, врать не буду.
> А искры из глаз летят.
>
> *из к/ф «Формула любви»*

В процессе изучения Java и работы над практическим материалом мы будем взаимодействовать с программным обеспечением (ПО), без которого обойтись крайне сложно. Важно знать, какие именно программные средства нам нужны, где их можно найти и как использовать.

Недостатка в программных средствах для работы с Java нет. Есть много интересных и полезных программных продуктов, большинство которых, кстати, распространяется на условиях лицензии с открытым кодом. Мы постараемся ограничиться разумным минимумом.

Что же нам понадобится? Во-первых, код где-то нужно набирать. Поэтому пригодится редактор — программа, которая позволит создать файл с набором команд на языке Java. Теоретически для этой цели сгодится и обычный текстовый редактор. Но, как мы узнаем далее, есть средства и лучше. Во-вторых, программу предстоит компилировать. Для этого понадобится специальная программа-компилятор. Такая программа создаст на основе файла с программой другой файл (или файлы) с промежуточным байт-кодом. Затем данный байт-код должен выполняться на компьютере. Это работа виртуальной машины. То есть нужна еще одна программа. Без всех этих программ нам как будто не обойтись?

На практике необходимо установить систему JDK (сокращение от *Java Development Kit*). Это набор средств разработки, в который входит, помимо прочего, компилятор и среда выполнения JRE (сокращение от *Java Runtime Environment*) — фактически та самая виртуальная машина. Загрузить систему JDK можно на сайте компании Oracle www.oracle.com. Соответствующее ПО распространяется свободно. Обновления также можно загружать с сайта поддержки языка Java www.java.com.

НА ЗАМЕТКУ

Как и где загружать ПО, а также методы работы с соответствующими программами описываются в приложении (в конце книги). Существует несколько платформ Java. Нас интересует стандартная платформа Java Standard Edition (сокращенно Java SE), рассчитанная на разработку обычных (не корпоративных) приложений.

Можно было бы ограничиться установкой системы JDK, но тогда пришлось бы набирать код в текстовом редакторе, а компилировать программы и запускать их — в командной строке. Это не самый лучший вариант, поэтому используются специальные программы, которые называются *средами разработки* (сокращенно IDE, от *Integrated Development Environment*).

Среда разработки позволяет в простом и удобном режиме решать все задачи сразу, не отвлекаясь на запуск разных вспомогательных программ, — набирать код, редактировать его, выполнять отладку, компиляцию и запускать скомпилированный проект. Поэтому использование среды разработки — вполне разумный ход. При этом для работы с Java существует много хороших и, что немаловажно, бесплатных сред разработки с сопоставимой функциональностью. Во многом выбор той или иной среды определяется личными предпочтениями и вкусами пользователя. На сегодня наиболее популярными средами разработки являются IntelliJ IDEA компании JetBrains (www.jetbrains.com), NetBeans (сайт https://netbeans.org) и Eclipse (www.

eclipse.org). Но есть и другие. Например, на сайте https://codenvy.io можно познакомиться с облачной средой разработки Codenvy, которая позволяет разрабатывать, кроме прочего, и программы на языке Java. При желании читатель без труда сможет найти и протестировать и другие подобные приложения.

ПОДРОБНОСТИ

Обычно до установки среды разработки следует установить систему JDK. Подчеркнем: устанавливаемая версия среды разработки должна поддерживать установленную версию JDK. Дело в том, что обычно версии JDK обновляются быстрее, чем версии сред разработки, поэтому не всегда та или иная среда разработки поддерживает последнюю версию JDK.

На момент написания книги актуальной (последней) является версия Java 12. Начиная с версии Java 9, поддерживаются только 64-разрядные операционные системы. Критически важными являются обновления, появившиеся в версии Java 8. Материал книги и представленные в книге примеры рассчитаны на то, что обучающимися используется версия не ниже Java 8.

Примеры, представленные в книге, тестировались в среде разработки IntelliJ IDEA.

Особенности книги

> Да, это от души. Замечательно. Достойно восхищения.
>
> *из к/ф «Формула любви»*

Книга имеет прикладную направленность и рассчитана на тех, кто изучает Java самостоятельно или в рамках учебного курса в университете (институте или на компьютерных курсах). Упор при подборе материала делался на то, чтобы даже читатель —новичок в программировании мог с самого начала писать программы, постепенно наращивая их сложность. Все программы, рассматриваемые в основной части книги, можно разбить на две группы: учебные и повышенной сложности. В последнем случае речь обычно идет о вычислительных задачах. Читателям не следует паниковать, если какие-то примеры вначале покажутся не очень понятными. Учебных программ вполне достаточно, чтобы освоить и понять основные приемы в работе с кодами. С обретением опыта и знаний можно вернуться к более сложным примерам.

Книга охватывает все основные концепции, необходимые для эффективного составления программ на Java. Вот некоторые из тем, которые рассматриваются в книге:

- Базовые типы и операторы.
- Управляющие инструкции (операторы цикла, условный оператор и оператор выбора).

- Работа с массивами.
- Описание классов и создание объектов.
- Наследование и использование интерфейсов.
- Лямбда-выражения и работа со ссылками на методы.
- Обработка исключительных ситуаций.
- Многопоточное программирование.
- Обобщенные классы и методы.
- Система ввода/вывода и работа с файлами.
- Создание приложений с графическим интерфейсом.

НА ЗАМЕТКУ

Начиная с версии Java 11, апплеты больше не поддерживаются, поэтому в книге они не обсуждаются.

В конце каждой главы приводится краткое резюме.

Полезные ресурсы

> Огонь тоже считался божественным, пока Прометей не выкрал его. Теперь мы кипятим на нем воду.
>
> *из к/ф «Формула любви»*

В книге представлен материал, необходимый для успешного освоения методов и приемов работы с Java. Тем не менее всегда полезно получать информацию из разных источников. Далее приводятся ссылки на ресурсы, которые могут быть полезны в процессе изучения Java.

- Много полезных сведений и важной документации представлено на сайте компании Oracle www.oracle.com.
- Справку по среде разработки (той, которую читатель решит использовать) можно получить на сайте поддержки соответствующей среды.
- Полезными могут быть многочисленные обучающие платформы, где можно изучить программирование (для работы с некоторыми платформами нужно знать английский язык): Coursera (www.coursera.org), edX (www.edx.org),

MIT Open Courseware (ресурс Массачусетского технологического института, https://ocw.mit.edu).

- В интернете есть огромное количество видеоуроков, которые легко найти с помощью поисковых систем. Оптимальный вариант — ориентироваться на официальные учебные курсы, которые читаются студентам. Большинство ведущих университетов выкладывают в свободный доступ записи лекций, в том числе и по программированию на Java. Обычно такие курсы готовятся профессионалами и адаптированы для аудиторий с разным уровнем подготовки.
- Никакое учебное пособие не может заменить живое (или через онлайн) общение с людьми, поэтому с вопросами или идеями полезно обращаться на форумы программистов. Например, один из самых известных ресурсов *Хабр* (https://habr.com) содержит много полезной информации разной направленности. Много информации можно найти на *Stack Overflow* (https://stackoverflow.com); обратите внимание, что обсуждения ведутся на английском языке. Есть и другие, которые читатель легко найдет в случае необходимости. Если возникают вопросы по поводу программных кодов и разных аспектов программирования, следует не лениться и не терять оптимизма, а искать. Какой бы ни была проблема, скорее всего, кто-то уже сталкивался с подобным и с высокой долей вероятности решение существует. Как говорится, кто ищет — тот всегда найдет.

Обратная связь

Слово лечит, разговор мысль отгоняет.

из к/ф «Формула любви»

Полезную информацию читатели могут найти на моем сайте www.vasilev.kiev.ua. Письма с пожеланиями и предложениями можно отправлять по адресу alex@vasilev.kiev.ua. Хотя, к сожалению, у меня нет возможности (в силу очевидных причин) ответить на все сообщения, это не приуменьшает благодарности читателям за их отзывы и замечания.

Благодарности

> Не умеете лгать, молодой человек. Все люди разделяются на тех, которым что-то надобно от меня, и на остальных, от которых что-то нужно мне.
>
> *из к/ф «Формула любви»*

Моя огромная благодарность — читателям, которые проявляют интерес к книгам, высказывают пожелания и замечания. Я искренне признателен своим студентам и слушателям курсов, которые стали той благодарной и терпеливой аудиторией, на которой апробировались и проверялись подходы и методы, положенные в основу книги.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Основы Java

Фандорин, у меня времени нет! Скажите по-человечески. Я не понимаю этого языка.

из к/ф «Статский советник»

Язык Java является полностью объектно-ориентированным. Это означает, что для составления даже самой простой программы необходимо описать класс.

Рассмотрение методов программирования в Java начнем с наиболее простых случаев. При этом нам все же придется использовать классы. Чтобы не загромождать начало книги довольно отвлеченными и не всегда понятными для новичков вопросами по созданию классов и объектов, постулируем некоторые базовые синтаксические конструкции как основу создания программы в Java, а затем, когда у нас уже будет небольшой опыт в программировании, обратимся к более подробному рассмотрению методов объектно-ориентированного программирования (далее — ООП). Такой подход позволит читателю, не знакомому с концепцией ООП, легче и быстрее усваивать новый материал, а затем плавно перейти к созданию реальных объектно-ориентированных программ в Java.

Простые программы

Отлично, отлично! Простенько и со вкусом!

из к/ф «Бриллиантовая рука»

Мы сразу начнем с небольшой программы, которая в процессе выполнения отображает сообщение в окне вывода среды разработки. Программа представлена в листинге 1.1.

Листинг 1.1. Отображение сообщения в окне вывода

```
class Demo{
   public static void main(String[] args){
      System.out.println("Приступаем к изучению Java!");
   }
}
```

После компиляции и запуска программы (например, в среде IntelliJ IDEA) в окне вывода появляется сообщение:

Результат выполнения программы (из листинга 1.1)

```
Приступаем к изучению Java!
```

Рассмотрим код подробнее. Сразу отметим, что фигурными скобками отмечаются блоки кода. Код размещается между открывающей (символ `{`) и закрывающей (символ `}`) фигурными скобками. В данном случае использовано две пары фигурных скобок. Первая, внешняя, пара нужна для определения кода *класса*, вторая — для определения *метода* этого класса.

НА ЗАМЕТКУ

Метод представляет собой именованный блок команд, которые выполняются при вызове метода. Другими словами, есть набор команд, и этот набор команд имеет имя.

Что такое класс, мы обсудим немного позже.

Создание программы в Java начинается с описания класса. Описание класса начинается с ключевого слова `class`. После этого следует уникальное имя класса. Мы его придумываем сами. В данном случае класс называется `Demo`. Код класса размещается в блоке из фигурных скобок.

ПОДРОБНОСТИ

Если мы назвали класс `Demo`, то файл, в котором описывается этот класс, должен называться `Demo.java`. В противном случае программа может не скомпилироваться.

Для описания класса используют следующую синтаксическую конструкцию:

```
class название_класса{
   // Описание класса
}
```

Код класса `Demo` состоит всего из одного метода с названием `main()` (здесь и далее названия методов будут указываться с круглыми скобками после имени, чтобы

отличать их от названий *переменных*). Это *главный метод программы*. Называться главный метод должен `main()` и никак иначе.

ПОДРОБНОСТИ

Каждая программа, написанная на Java, имеет один и только один главный метод. Этот метод обязательно называется `main()` и описывается по определенным правилам с определенными атрибутами. Класс, в котором описан главный метод, называется главным классом. Выполнение программы начинается с выполнения кода главного метода. Поэтому команды, которые размещаются в теле главного метода, — это именно те команды, которые выполняются в программе.

Главный метод описывается в соответствии с таким шаблоном:

```
public static void main(String[] args){
   // Команды в теле главного метода
}
```

Помимо того что метод должен называться `main()`, в его описании (перед названием метода) указываются ключевые слова `public`, `static` и `void`. Ключевое слово `public` означает, что метод является открытым и он доступен за пределами класса, в котором описан.

ПОДРОБНОСТИ

Методы (и поля, о которых позже), описываемые в классе, могут иметь разный уровень доступа. Уровень доступа определяется специальным ключевым словом (`public`, `private` или `protected`) или его отсутствием. Уровень доступа, в свою очередь, определяет, из какого места программы вы можете вызвать метод (или обратиться к полю). Самый высокий уровень доступа — когда метод описан с ключевым словом `public`. В таком случае доступ к методу есть практически из любого места кода. Поскольку выполнение программы начинается с вызова главного метода, то логично, что у него должен быть наивысший уровень доступа.

Ключевое слово `static` означает, что метод статический и для его вызова нет необходимости создавать объект класса.

ПОДРОБНОСТИ

Как отмечалось выше, метод — это именованный блок команд. Методы бывают статические и обычные (нестатические). Статический метод, хотя он и описывается в классе, существует сам по себе. Чтобы вызвать такой метод, достаточно знать, в каком классе он описан. Нестатический метод спрятан в объекте. Чтобы вызвать такой метод, сначала нужно создать объект. Объект создается на основе класса, подобно тому как дом (аналог объекта) строится на основе генерального плана (аналог класса). Если бы главный метод не был статическим, то для его вызова

на основе главного класса пришлось бы создать объект. Но чтобы создать объект, нужно запустить программу. А запуск программы — это выполнение главного метода. Получается замкнутый круг. Чтобы его разорвать, главный метод должен быть статическим.

Ключевое слово `void` означает, что метод не возвращает результат.

ПОДРОБНОСТИ

Методы могут возвращать результат. Если так, то после выполнения метода на память остается некоторое значение (результат метода). Его можно использовать в программе. Каждое значение относится к определенному *типу* (например, целое число, символ, текст). Если метод возвращает результат, то в описании метода указывается тип результата метода. Это специальный идентификатор, обозначающий определенный тип данных (например, `int` для целых чисел, `char` для символов или `String` для текста — хотя в последнем случае не все так просто). Но метод может и не возвращать результат. В таком случае при вызове метода просто выполняются команды, которые есть в этом методе. Если метод не возвращает результат, то идентификатором типа результата в описании метода указывается ключевое слово `void`.

То, что главный метод не возвращает результат, вполне логично. Поскольку главный метод — это фактически и есть программа, то возвращать результат (в рассматриваемом случае) особо некуда.

Это — обязательные атрибуты метода `main()`. Они должны указываться всегда, когда мы описываем главный метод. Но это еще не все. В круглых скобках после имени главного метода указана инструкция `String[] args`. Она описывает аргумент `args` главного метода. Аргументом является текстовый массив (набор значений текстового типа).

ПОДРОБНОСТИ

Текстовые значения в Java реализуются как объекты класса `String`. Пока не вдаваясь в подробности и с некоторым упрощением можно рассматривать класс `String` как текстовый тип данных. Массив, как мы узнаем немного позже, представляет собой набор значений определенного типа. Чтобы показать, что мы имеем дело с набором значений (массивом), а не с одним значением, к идентификатору типа (в данном случае это `String`) добавляют пустые квадратные скобки `[]`. В итоге получается инструкция `String[]`, обозначающая массив из текстовых значений.

Аргумент главного метода может понадобиться в случае, если при вызове программы ей передаются параметры. Тогда эти параметры считываются (в текстовом формате) и из них формируется массив. Ссылка на этот массив передается в главный метод через аргумент. Если параметры программе не передаются (а они обычно не передаются), то аргумент главного метода играет чисто декоративную

роль. Но описать его следует в любом случае. Поэтому инструкция `String[] args` в описании главного метода тоже является обязательной.

НА ЗАМЕТКУ

Название аргумента главного метода (допустим, `args`) выбирает пользователь. Но традиционно аргумент главного метода называют именно `args`.

Все это может поначалу выглядеть довольно запутанно. Но пугаться не стоит. Вот главное, что нужно запомнить: каждый раз при создании новой программы мы будем использовать определенный шаблон. Вот он:

```
class название_класса{
   public static void main(String[] args){
      // Команды в теле главного метода
   }
}
```

Практически любая наша программа будет содержать этот шаблон. Команды, которые формируют собственно программу, размещаются в теле главного метода `main()`.

Но вернемся к примеру (см. листинг 1.1). В теле главного метода — всего одна команда `System.out.println("Приступаем к изучению Java!")`. Команда заканчивается точкой с запятой — это стандарт для Java (все команды заканчиваются точкой с запятой). Командой с помощью встроенного метода `println()` в окне вывода печатается сообщение `"Приступаем к изучению Java!"`. Текст сообщения указан аргументом метода.

ПОДРОБНОСТИ

Текстовые значения (текстовые литералы) в Java заключаются в двойные кавычки.

Команда вызова метода дает нам пример использования классического точечного синтаксиса. Итак, есть библиотечный класс `System`. У этого класса есть статическое поле `out`, которое связано с потоком вывода. Поле `out` является объектом. И у этого объекта имеется метод `println()`, который нам нужно вызвать. В таком случае мы указываем полную иерархию: класс, поле, метод. Разделителем является точка. То есть инструкция вида `System.out.println()` означает, что нужно в классе `System` найти объект `out` и вызвать из него метод `println()`.

НА ЗАМЕТКУ

Важно понять: метод `println()` позволяет напечатать сообщение в окне вывода, и вызывается он в формате `System.out.println()`. Остальное (в рассматриваемом случае) не очень-то и важно.

Собственно, на этом анализ кода нашей первой программы мы заканчиваем.

НА ЗАМЕТКУ

Если у читателя возникли проблемы с компиляцией и запуском программы, можно обратиться к Приложению, в котором кратко описаны методы работы со средой разработки IntelliJ IDEA.

Далее мы рассмотрим еще одну простую программу (листинг 1.2), в которой тоже выводится сообщение, но уже не в окно вывода, а в специальное диалоговое окно, которое программой же и создается.

Листинг 1.2. Отображение сообщения в диалоговом окне

```
// Статический импорт метода:
import static javax.swing.JOptionPane.showMessageDialog;
class Demo{
   public static void main(String[] args){
      // Отображение сообщения в диалоговом окне:
      showMessageDialog(null,"Программируем на Java!");
   }
}
```

При запуске программы появляется окно, представленное на рис. 1.1.

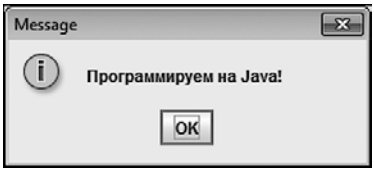


Рис. 1.1. Диалоговое окно с сообщением

Теперь постараемся разобраться, что происходит. Код программы — более чем простой. В теле главного метода — всего одна команда `showMessageDialog(null, "Программируем на Java!")`, посредством которой и отображается окно с сообщением. У метода — два аргумента, которые в команде вызова метода разделяются запятыми. Второй аргумент `"Программируем на Java!"` — это сообщение, которое появляется в диалоговом окне. А вот первым аргументом служит ключевое слово `null`, обозначающее пустую ссылку (отсутствие значения). Вообще, первый аргумент определяет окно, которое является родительским для данного диалогового окна. Инструкция `null` означает, что такого окна нет.

ПОДРОБНОСТИ

Есть такое понятие, как модальное окно. Допустим, имеется главное окно программы. Если появляется модальное окно, то пока вы его не закроете, доступ к главному окну будет заблокирован. Обычно окно с сообщением используют именно как модальное, которое при появлении должно заблокировать главное окно программы (для диалогового окна это главное окно является родительским).

В нашем случае такого главного окна нет, поэтому ничего блокировать не нужно. Как следствие, первым аргументом методу `showMessageDialog()` передаем ссылку `null`. Практически каждый раз, когда мы будем использовать этот метод, первым аргументом ему будет передаваться значение `null`.

Метод `showMessageDialog()` является статическим методом класса `JOptionPane`. Чтобы класс стал доступен в программе, его необходимо импортировать. Мы используем *статический импорт*, для чего в начале программы размещается инструкция `import static javax.swing.JOptionPane.showMessageDialog`. Она означает, что в программу импортируется статический метод `showMessageDialog`. Благодаря этому в программе метод можно использовать.

ПОДРОБНОСТИ

В Java все классы разбиваются по группам, которые называются *пакетами*. Когда в программу импортируется класс (или его статические методы), то указывается полный путь к классу в иерархии пакетов. Так, выражение `javax.swing.JOptionPane.showMessageDialog` в `import`-инструкции означает, что нас интересует метод `showMessageDialog()` из класса `JOptionPane`, который находится в пакете `swing`, а этот пакет, в свою очередь, находится в пакете `javax`.

Программа может содержать несколько `import`-инструкций. Причем это может быть обычный импорт или статический. Если импорт обычный, то после ключевого слова `import` указывается класс (вместе с иерархией пакетов), который импортируется в программу. В таком случае, если мы хотим использовать статический метод из этого класса, то при вызове метода следует указывать и имя класса. Если импорт статический, то после ключевого слова `import` указывается еще и ключевое слово `static`. В случае статического импорта при вызове статического метода имя класса можно не указывать. То есть статический импорт, по сравнению с обычным (нестатическим), удобнее. Недостаток статического импорта — в том, что он применим только к статическим полям и методам.

Классы (такие, например, как `System`), которые находятся в подпакете `lang` пакета `java`, импортировать не нужно — они доступны по умолчанию.

Кроме команд и уже знакомых нам инструкций, в программе есть еще и *комментарии* (начинаются с двойной косой черты, или двойного слеша, `//`). Комментарий — это текст, предназначенный для человека, а не для компилятора, который комментарии игнорирует. Хороший комментарий существенно улучшает читабельность кода и позволяет избежать многих неприятностей.

ПОДРОБНОСТИ

В Java существует три типа комментариев:

- Однострочный комментарий. Такой комментарий начинается с двойной косой черты (символ `//`). Все, что находится в строке кода справа от двойной косой черты, компилятором игнорируется.

- Многострочный комментарий. Такой комментарий начинается последовательностью символов `/*` и заканчивается последовательностью символов `*/`. Все, что находится между символами `/*` и `*/`, компилятором игнорируется.
- Многострочный комментарий документирования. Начинается последовательностью символов `/**` и заканчивается последовательностью символов `*/`. Обычно используется в случае, если планируется автоматическое генерирование документации.

Знакомство с переменными

Это мелочи. Но нет ничего важнее мелочей!

из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Переменная — это идентификатор (имя), с помощью которого можно получить доступ к памяти для записи значения или считывания значения. Переменную удобно представлять в виде ячейки с названием (имя переменной), и в эту ячейку можно что-то положить или посмотреть, что там находится. Данные, которые в принципе можно хранить с помощью переменных, разбиты на *типы*: например, целые числа, действительные числа, символы, логические значения. Каждая переменная предназначена для работы с данными только определенного типа. Соответственно, у каждой переменной есть тип.

Чтобы использовать переменную в программе, ее необходимо *объявить*. При объявлении переменной указывается тип переменной и ее название. Тип определяется с помощью специального идентификатора. Например, `int` означает целочисленный тип, `double` — тип для чисел с плавающей точкой (действительные числа), `char` — символьный тип (значением переменной может быть символ или буква). Для текста мы будем использовать тип `String`.

ПОДРОБНОСТИ

Описанная выше схема, когда переменная *содержит* значение, относится к базовым, или примитивным, типам (таким, как `int` или `char`). Но есть еще и ссылочные типы, переменные которых *ссылаются* на значение. Например, объектные переменные ссылаются на объект: значением такой переменной является адрес объекта в памяти. Текст реализуется как объект класса `String` и к базовым типам не относится. Переменная класса `String` ссылается на текстовый объект. Особенности работы с текстом мы еще будем обсуждать. Пока же, не вдаваясь в подробности, будем использовать для работы с текстом тип `String`.

Объявлять переменную можно в любом месте программы, но до того, как переменная впервые используется. Если объявляется несколько переменных одного типа,

то их можно перечислить через запятую после идентификатора типа. Одновременно с объявлением переменной ей можно присвоить значение (*инициализировать* переменную).

ПОДРОБНОСТИ

Если переменную объявить и инициализировать с ключевым словом `final`, то такая переменная становится *константой*. Изменить значение константы в процессе выполнения программы нельзя.

НА ЗАМЕТКУ

Область доступности переменных определяется блоком, в котором переменная объявлена. Блок, в свою очередь, выделяется парой фигурных скобок { и }.

Далее мы рассмотрим программу, в которой используются переменные. В процессе выполнения программы появляется диалоговое окно с полем ввода, в которое пользователь должен ввести текст. Текст считывается, появляется еще одно диалоговое окно с информацией о введенном тексте, количестве символов в тексте, указаны первая и последняя буквы в тексте. Обратимся к листингу 1.3.

Листинг 1.3. Знакомство с переменными

```
// Статический импорт:
import static javax.swing.JOptionPane.*;
class Demo{
   public static void main(String[] args){
      // Текстовые переменные:
      String txt,str;
      // Отображение окна с полем ввода:
      txt=showInputDialog("Введите текст:");
      // Целочисленная переменная:
      int size=txt.length();
      // Символьные переменные:
      char A=txt.charAt(0);
      char B=txt.charAt(size-1);
      // Формируется значение текстовой переменной:
      str="Текст: "+txt+"\n";
      str+="Символов в тексте: "+size+"\n";
      str+="Первый символ: "+A+"\n";
      str+="Последний символ: "+B;
      // Отображение диалогового окна:
      showMessageDialog(null,str);
   }
}
```

После запуска программы появляется окно (рис. 1.2).

В поле ввода окна мы набираем текст `Изучаем Java` и нажимаем кнопку OK. Окно закроется, но появится новое (рис. 1.3).

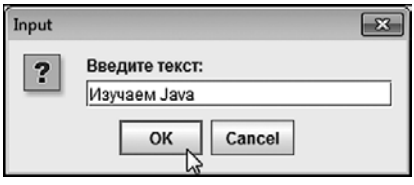


Рис. 1.2. В окне с полем ввода указан текст

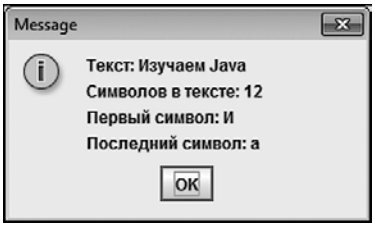


Рис. 1.3. Диалоговое окно с информацией о введенном тексте

Проанализируем код из примера. Мы используем статический импорт, причем в инструкции `import static javax.swing.JOptionPane.*` вместо имени импортируемого метода указана звездочка `*`. В таком случае из класса `JOptionPane` импортируются все статические методы. Нас интересуют два: уже знакомый нам метод `showMessageDialog()` для отображения диалогового окна с сообщением и новый для нас метод `showInputDialog()` для отображения диалогового окна с полем ввода.

В главном методе объявляются две текстовые (тип `String`) переменные, `txt` и `str`. Значение переменной `txt` вычисляется инструкцией `txt=showInputDialog("Введите текст:")`. Здесь вызывается метод `showInputDialog()`, которым отображается окно с полем ввода. Текст, переданный аргументом методу, появляется над полем ввода в диалоговом окне. Метод возвращает результат. Это текст, который пользователь ввел в поле ввода перед нажатием кнопки OK (см. рис. 1.2). Поскольку инструкция вызова метода присваивается переменной `txt`, в результате значением переменной становится тот текст, который ввел пользователь.

ПОДРОБНОСТИ

Оператором присваивания в Java является знак равенства =. Выражения вида `переменная=значение` обрабатывается так: `значение`, указанное справа от оператора присваивания, присваивается `переменной`, указанной слева от оператора присваивания.

Еще в программе объявляется целочисленная (тип `int`) переменная `size`. Значением такой переменной может быть целое число. Ей сразу присваивается значение. В рассматриваемом случае значением присваивается выражение `txt.length()`, результат которого — длина текста (количество символов), записанного в переменную `txt`.

ПОДРОБНОСТИ

Текстовая переменная `txt` является объектом. У текстовых объектов есть много интересных методов, включая метод `length()`, который результатом возвращает количество символов в текстовом объекте, из которого вызывается метод.

Метод `charAt()` позволяет прочитать символ из текста (в том объекте, из которого вызывается метод). Аргументом методу передается индекс символа (индексация символов начинается с нуля — то есть у первого символа в тексте нулевой индекс).

Также мы используем две символьные (тип `char`) переменные, `A` и `B`. В переменную `A` записывается первый символ из введенного пользователем текста (вычисляется выражением `txt.charAt(0)`), а в переменную `B` записывается последняя буква в тексте (вычисляется выражением `txt.charAt(size-1)`). В данном случае метод `charAt()` вызывается из текстового объекта `txt`. Результатом метод возвращает символ с указанным индексом.

ПОДРОБНОСТИ

Мы учли, что первый символ в тексте имеет нулевой индекс, а индекс последнего элемента на единицу меньше длины текста.

Значение переменной `str` формируется несколькими командами. Начальное значение переменной дается выражением `"Текст: "+txt+"\n"`, которым вычисляется сумма (использован оператор сложения `+`) трех текстовых значений. Если складываются текстовые значения, то выполняется объединение текста.

ПОДРОБНОСТИ

Здесь и далее мы использовали инструкцию перехода к новой строке `\n`. Если текст содержит такую инструкцию, то при попытке ее «напечатать» осуществляется переход к новой строке.

Затем к текущему текстовому значению переменной `str` дописываются новые блоки текста (команды `str+="Символов в тексте: "+size+"\n"`, `str+="Первый символ: "+A+"\n"` и `str+="Последний символ: "+B`).

ПОДРОБНОСТИ

Обратите внимание. Во-первых, мы использовали оператор `+=`. Это один из составных операторов присваивания. Выражение вида `X+=Y` эквивалентно выражению `X=X+Y`: то есть к текущему значению переменной `X` добавляется значение `Y`.

Во-вторых, если к тексту добавляется число или символ, то это число или символ автоматически преобразуются к текстовому формату и далее выполняется объединение текстовых строк.

После того как значение переменной `str` сформировано, `showMessageDialog (null,str)` отображает диалоговое окно с сообщением.

Базовые типы данных

Вы получите то, что желали, согласно намеченным контурам.

из к/ф «Формула любви»

В Java существует четыре группы базовых типов: для работы с целыми числами (четыре типа), для работы с действительными числами в формате с плавающей точкой (два типа), для использования символов (один тип) и логический тип (один). Всего получается восемь базовых типов (табл. 1.1).

Таблица 1.1. Базовые (простые) типы в Java

Тип данных (название)	Количество битов	Описание	Класс-оболочка
`byte`	8	Целые числа в диапазоне от –128 до 127	`Byte`
`short`	16	Целые числа в диапазоне от –32 768 до 32 767	`Short`
`int`	32	Целые числа в диапазоне от –2147483648 до 2147483647	`Integer`
`long`	64	Целые числа в диапазоне от –9223372036854775808 до 9223372036854775807	`Long`
`float`	32	Действительные числа. Минимальный шаг дискретности (точность) составляет величину $3{,}4 \times 10^{-38}$, максимальное значение (по модулю) — $3{,}4 \times 10^{38}$	`Float`
`double`	64	Действительные числа (двойной точности). Минимальный шаг дискретности (точность) составляет величину $1{,}7 \times 10^{-308}$, максимальное значение (по модулю) — $1{,}7 \times 10^{308}$	`Double`
`char`	16	Тип для представления символьных значений. Реализуются символы с кодами от 0 до 65 536	`Character`
`boolean`	Зависит от виртуальной машины	Логический тип данных. Переменная этого типа может принимать два значения: `true` (*истина*) и `false` (*ложь*)	`Boolean`

В этой же таблице приведены названия *классов-оболочек* для базовых типов, используемых, когда переменную соответствующего типа необходимо реализовать как объект. Далее изучим каждую группу базовых типов отдельно. В первую очередь стоит обратить внимание на целочисленные типы данных.

- В Java существует четыре типа целочисленных данных: `byte`, `short`, `int` и `long`. Отличаются типы количеством битов, выделяемых для записи значения соответствующего типа. Размер в битах увеличивается от 8 для типа `byte` до 64 для типа `long` (с шагом дискретности 8 бит). На практике, если нет крайней необходимости, для работы с целыми числами рекомендуется использовать тип `int`.
- Для работы с действительными числами используются типы `float` и `double` (на практике обычно используется тип `double`). С помощью этих типов реализуется формат числа с плавающей точкой.
- В Java для символьных данных (тип `char`) выделяется 16 бит, что позволяет охватить практически все имеющиеся и использующиеся на сегодня символы.
- Что касается логического типа `boolean`, то переменные этого типа могут принимать всего два значения: `true` (*истина*) и `false` (*ложь*). Размер (в битах) переменной типа `boolean` зависит от типа используемой виртуальной Java-машины.

НА ЗАМЕТКУ

Есть такое понятие, как *литерал*, — это значение, предназначенное для восприятия человеком, которое не может быть изменено в программе. Текстовые литералы (например, текст `"Изучаем Java"`) заключаются в двойные кавычки. Символьные литералы заключаются в одинарные кавычки (например, `'A'` или `'Z'`). Числовые литералы, кроме обычного десятичного представления (например, `123`), могут быть записаны в восьмеричной и шестнадцатеричной системах счисления. Восьмеричные литералы начинаются с нуля. Следующие цифры в позиционной записи восьмеричного литерала могут принимать значения в диапазоне от `0` до `7` включительно. Например, восьмеричный литерал `012` означает десятичное число `10`. Шестнадцатеричные литералы начинаются с префикса `0x`. Для позиционного представления шестнадцатеричного числа используются цифры от `0` до `9` и буквы от `A` до `F`. Например, шестнадцатеричный литерал `0x12` означает десятичное число `18`.

Также можно использовать числа в формате с плавающей точкой (например, `12.3`). Такие числа нередко представляют в научной (экспоненциальной) нотации, в которой указывается *мантисса* и, через символ `e` или `E`, *показатель степени*. Например, литерал `1.2E-3` соответствует числу $1{,}2 \times 10^{-3}$, а литерал `2.5e7` соответствует числу $2{,}5 \times 10^{7}$.

Приведение типов

— Узнаешь, Маргадон?
— Натюрлих, экселенц!
из к/ф «Формула любви»

Строгая типизация переменных вместе с очевидными преимуществами привносит и ряд не столь очевидных проблем. Поясним это на простом примере. Предположим, что в программе объявлены две числовые переменные: одна типа `int` и другая типа `double`. Переменным присвоены значения. Далее мы хотим к переменной типа `double` прибавить значение переменной типа `int` и результат записать в первую переменную. С формальной точки зрения здесь нет никакой проблемы, поскольку целые числа являются подмножеством множества действительных чисел. С точки зрения программной логики ситуация не такая простая, ведь складываются переменные разных типов. Понятно, что на самом деле проблемы не возникает и описанную операцию можно выполнить. Выполнение операций подобного рода базируется на *автоматическом приведении типов*. Другими словами, если нужно вычислить выражение, в которое входят переменные разных типов, то автоматически выполняется преобразование входящих в выражение переменных к общему формату. Процесс автоматического преобразования типов подчиняется нескольким базовым правилам:

- Типы переменных, входящих в выражение, должны быть совместимыми. Например, целое число можно преобразовать в формат действительного числа, чего не скажешь о текстовой строке.
- Целевой тип (тип, к которому выполняется приведение) должен быть шире исходного типа. Другими словами, преобразование должно выполняться без потери данных.
- Перед выполнением арифметических операций типы `byte`, `short` и `char` расширяются до типа `int`.
- Если в выражении есть операнды типа `long`, то расширение осуществляется до типа `long`.
- Если в выражении есть операнды типа `float`, то расширение осуществляется до типа `float`.
- Если в выражении есть операнды типа `double`, то расширение осуществляется до типа `double`.

К этим правилам следует добавить не менее важные правила интерпретации литералов. Действительно, как следует рассматривать, например, число (литерал) `2`?

Как значение типа `int`, типа `long` или, например, типа `double`? Ответы на подобные вопросы дают следующие правила:

- Литералы, обозначающие целые числа, интерпретируются как значения типа `int`.
- Литералы, обозначающие действительные числа, интерпретируются как значения типа `double`.

Хотя эти правила представляются логичными и простыми, нередко автоматическое приведение типов приводит к непредсказуемым, на первый взгляд, результатам и ошибкам. Например, представим, что объявляется несколько переменных типа `byte`:

```
byte x=1,y=2,z;
```

Если мы теперь вычислим сумму `x+y` и захотим присвоить ее в качестве значения переменной `z`, то возникнет ошибка. В чем причина? Хотя все три переменные относятся к типу `byte`, при вычислении выражения `x+y` выполняется автоматическое преобразование к типу `int`. В результате имеет место попытка присвоить значение типа `int` переменной типа `byte`. Поскольку в Java преобразования с возможной потерей значений не допускаются, то и программа с таким кодом не скомпилируется.

НА ЗАМЕТКУ

При этом если мы инициализируем переменную типа `byte` с помощью целочисленного литерала (который по умолчанию относится к типу `int`), то ошибки не возникает (при условии, что присваиваемое переменной значение не выходит за допустимый диапазон).

Или, например, мы объявляем переменную типа `float`, а затем пытаемся ей присвоить значение, заданное числовым литералом (например, `2.5`). И в этом случае получим ошибку, поскольку литерал `2.5` относится к типу `double`, в то время как переменная объявлена как относящаяся к типу `float`. Поскольку тип `double` шире, чем тип `float` (диапазон возможных значений для типа `double` больше, чем для типа `float`), то такая операция теоретически может привести к потере значения. Хотя в данном случае потери значения и не произошло бы, но операция все равно запрещена (поскольку она потенциально опасна) и возникает ошибка.

Для решения проблем подобного рода в Java предусмотрено *явное приведение типов*. Правило простое: для приведения выражения к нужному типу перед этим выражением указывается имя типа, заключенное в круглые скобки. Например, следующий код является корректным:

```
byte x=1,y=2,z;
// Использовано явное приведение типа:
z=(byte)(x+y);
```

Командой `(byte)(x+y)` вычисляется сумма значений переменных `x` и `y`, а результат преобразуется к типу `byte`. Получается, что переменной типа `byte` присваивается значение типа `byte` и проблем не возникает. Тем не менее следует понимать, что явное приведение типа потенциально опасно, поскольку может приводить к потере значения. Такие ситуации должен отслеживать программист — системой они не отслеживаются.

Как отмечалось ранее, каждый литерал автоматически относится к определенному типу. Но с помощью специальных суффиксов тип литерала можно изменить. Так, суффикс `L` (или `l`) у целочисленного литерала (например, `123L`) означает, что он принадлежит к типу `long`, а суффикс `F` (или `f`) у литерала, обозначающего действительное число (например, `2.5F`), означает, что этот литерал относится к типу `float`. Поэтому мы можем, например, объявить переменную типа `float`, а затем присвоить ей значение `2.5F`. Ошибки в таком случае не будет.

ПОДРОБНОСТИ

В Java разрешена *динамическая инициализация переменных*, когда значением переменной при объявлении присваивается выражение, содержащее другие переменные. Например:

```
int x=3,y=4;
int z=x*x+y*y;
```

В данном случае переменная `z` инициализируется выражением `x*x+y*y`, то есть получает значение `25`. При динамической инициализации все переменные, входящие в соответствующее выражение, должны быть предварительно объявлены и им должны быть присвоены значения.

Основные операторы

> Дядя Степан, помог бы ты им, а? Ну грех смеяться над убогими.
>
> *из к/ф «Формула любви»*

Все операторы Java можно разделить на четыре группы: *арифметические*, *логические*, *побитовые* и операторы *сравнения*. Рассмотрим последовательно каждую группу операторов. Начнем с арифметических. Эти операторы перечислены в табл. 1.2.

Таблица 1.2. Арифметические операторы Java

Оператор	Описание
`+`	Бинарный оператор *сложения*. Результатом команды вида `a+b` является сумма значений переменных `a` и `b`
`-`	Бинарный оператор *вычитания*. Результатом команды вида `a-b` является разность значений переменных `a` и `b`
`*`	Бинарный оператор *умножения*. Результатом команды `a*b` является произведение значений переменных `a` и `b`
`/`	Бинарный оператор *деления*. Результатом команды вида `a/b` является частное от деления значений переменных `a` и `b`. Для целочисленных операндов по умолчанию выполняется деление нацело
`%`	Бинарный оператор вычисления *остатка от деления*. Результатом команды вида `a%b` является остаток от целочисленного деления значений переменных `a` и `b`
`++`	Унарный оператор *инкремента*. В результате выполнения команды вида `a++` (или `++a`) значение переменной `a` увеличивается на единицу
`--`	Унарный оператор *декремента*. В результате выполнения команды вида `a--` (или `--a`) значение переменной `a` уменьшается на единицу

Эти операторы имеют некоторые особенности. В первую очередь обращаем внимание на оператор деления `/`. Если операндами являются целые числа, то деление выполняется нацело. Рассмотрим последовательность команд:

```
int a=7,b=2;
double x=a/b;
```

В данном примере переменная `x` получает значение `3.0`, а не `3.5`, как можно было бы ожидать. Дело в том, что сначала вычисляется выражение `a/b`. Поскольку операнды целочисленные, выполняется целочисленное деление (результат целочисленного деления `7` на `2` равен `3`). И только после этого полученное значение преобразуется к формату `double` и присваивается переменной `x`.

Для того чтобы при целочисленных операндах выполнялось обычное (не целочисленное) деление, перед выражением с оператором деления указывается в круглых скобках идентификатор типа `double`. Например, так:

```
int a=7,b=2;
double x=(double)a/b;
```

Теперь значение переменной `x` равно `3.5`.

В Java есть группа составных операторов присваивания. Если op — один из арифметических бинарных операторов, то соответствующий составной оператор присваивания выглядит как op=. Выражение вида x op=y является эквивалентом команды x=x op y.

НА ЗАМЕТКУ

Речь об операторах +=, -=, *=, /= и %=. Например, выражение a+=b эквивалентно выражению a=a+b, выражение a*=b эквивалентно выражению a=a*b, и так далее. Следует также отметить, что между командами вида x op=y и x=x op y разница все же есть. На самом деле команда x op=y эквивалентна команде x=(тип x)(x op y) — после вычисления выражения x op y оно еще и приводится к типу переменной x, после чего полученное значение присваивается переменной x. Иногда это бывает важно. Например, если переменная x относится к типу char и объявлена командой char x='A', то команда x+=1 вполне законна и в результате переменная x будет иметь значение 'B'. Как это происходит? К начальному значению 'A' переменной x прибавляется значение 1. В Java есть автоматическое преобразование типа char в тип int. Если к символу прибавляется число, то символ автоматически преобразуется в число (результат такого преобразования — код символа в кодовой таблице) и к этому числу прибавляется единица. То есть результатом выражения x+1 является целое число (код буквы 'B'). А переменная x относится к типу char. Тип char в тип int автоматически преобразуется, а тип int в тип char автоматически не преобразуется (но можно выполнить явное приведение типа). Поэтому переменной x нельзя просто присвоить значение выражением x+1. Следует использовать явное приведение типа как в команде x=(char)(x+1). Но если использована команда x+=1, то после вычисления выражения x+1 оно насильно преобразуется к типу char, и полученное символьное значение записывается в переменную x. Число в символ преобразуется так: число интерпретируется как код символа в кодовой таблице. Буквы в кодовой таблице расположены в соответствии с их расположением в алфавите.

Еще два исключительно полезных унарных оператора — операторы инкремента (++) и декремента (--). Действие оператора декремента сводится к увеличению на единицу значения операнда, а оператор декремента на единицу уменьшает значение операнда. Другими словами, команда x++ эквивалентна команде x=x+1, а команда x-- эквивалентна команде x=x-1.

НА ЗАМЕТКУ

Ситуация такая же, как с составными операторами присваивания. Команда вида x++ или x-- эквивалентна соответственно командам x=(тип x)(x+1) и x=(тип x)(x-1). То есть к текущему значению переменной x прибавляется единица (для оператора инкремента) или вычитается единица (для оператора декремента), и результат приводится к типу переменной x.

У операторов инкремента и декремента есть *постфиксная* форма (оператор следует после операнда: `x++` или `x--`) и *префиксная* форма (оператор располагается перед операндом: `++x` или `--x`). С точки зрения действия на операнд нет разницы в том, префиксная или постфиксная форма оператора использована. Однако если инструкция с оператором инкремента или декремента является частью более сложного выражения, то различие в префиксной и постфиксной форме операторов инкремента и декремента проявляется. Если использована префиксная форма оператора, то значением выражения является новое значение операнда. Если использована постфиксная форма оператора, то значением выражения является старое значение операнда. Рассмотрим небольшой пример:

```
int n,m;
n=100;
m=n++;
```

В этом случае после выполнения команд переменная `n` будет иметь значение `101`, а переменная `m` — значение `100`. При выполнении команды `m=n++` переменная `n` получает значение `101`, а поскольку использована постфиксная форма оператора инкремента, то значение выражения `n++` равно `100`. Такое значение получает переменная `m`.

Иной результат выполнения следующих команд:

```
int n,m;
n=100;
m=++n;
```

Обе переменные (`n` и `m`) в этом случае имеют значение `101`. При выполнении команды `m=++n` на единицу увеличивается значение переменной `n`, а поскольку в команде `m=++n` использована префиксная форма оператора инкремента, то значением выражения `++n` является число `101`.

Следующую группу образуют логические операторы (табл. 1.3). Операндами логических операторов являются переменные и литералы типа `boolean`.

Таблица 1.3. Логические операторы Java

Оператор	Описание
`&`	Бинарный оператор *логическое и*. Результатом операции вида `A&B` является значение `true`, если значения обоих операндов `A` и `B` равны `true`. В противном случае возвращается значение `false`
`&&`	Бинарный оператор *логические и* (сокращенная форма). Особенность оператора, по сравнению с оператором `&`, состоит в том, что если значение первого операнда равно `false`, то значение второго операнда не проверяется

Оператор	Описание
|	Бинарный оператор *логическое или*. Результатом операции вида `A|B` является значение `true`, если значение хотя бы одного из операндов `A` и `B` равно `true`. В противном случае возвращается значение `false`
||	Бинарный оператор *логическое или* (сокращенная форма). Особенность оператора, по сравнению с оператором `|`, состоит в том, что если значение первого операнда равно `true`, то значение второго операнда не проверяется
^	Бинарный оператор *исключающее или*. Результатом операции вида `A^B` является значение `true`, если значение одного и только одного операнда `A` или `B` равно `true`. В противном случае возвращается значение `false`
!	Унарный оператор *логическое отрицание*. Результатом выражения вида `!A` является значение `true`, если значение операнда `A` равно `false`. Если значение операнда `A` равно `true`, то результатом выражения `!A` является значение `false`

Логические операторы обычно используются в качестве условий в условных операторах и операторах цикла. Все операторы сравнения — бинарные (табл. 1.4).

Таблица 1.4. Операторы сравнения Java

Оператор	Описание
==	Оператор *равно*. Результатом выражения вида `A==B` является значение `true`, если операнды `A` и `B` имеют одинаковые значения. В противном случае значением является `false`
<	Оператор *не равно*. Результатом выражения вида `A<B` является значение `true`, если значение операнда `A` меньше значения операнда `B`. В противном случае значением является `false`
<=	Оператор *меньше или равно*. Результатом выражения вида `A<=B` является значение `true`, если значение операнда `A` не больше значения операнда `B`. В противном случае значением является `false`
>	Оператор *больше*. Результатом выражения вида `A>B` является значение `true`, если значение операнда `A` больше значения операнда `B`. В противном случае значением является `false`
>=	Оператор *больше или равно*. Результатом выражения вида `A>=B` является значение `true`, если значение операнда `A` не меньше значения операнда `B`. В противном случае значением является `false`
!=	Оператор *не равно*. Результатом выражения вида `A!=B` является значение `true`, если операнды `A` и `B` имеют разные значения. В противном случае значением является `false`

Операторы сравнения обычно используются совместно с логическими операторами.

Для понимания принципов работы побитовых операторов (табл. 1.5) необходимо иметь хотя бы элементарные познания о двоичном представлении чисел. Напомним читателю некоторые основные моменты.

- В двоичном представлении позиционная запись числа содержит нули и единицы.
- Старший бит (самый первый слева) определяет знак числа. Для положительных чисел старший бит равен нулю, для отрицательных — единице.

Перевод из двоичной системы счисления положительного числа с позиционной записью $b_n b_{n-1} ... b_2 b_1 b_0$, где параметры $b_k (k = 0, 1, ..., n)$ могут принимать значения 0 или 1, выполняется так: $b_n b_{n-1} ... b_2 b_1 b_0 = b_0 2^0 + b_1 2^1 + b_2 2^2 + ... + b_{n-1} 2^{n-1} + b_n 2^n$.

Для перевода отрицательного двоичного числа в десятичное представление производится побитовое инвертирование кода (все нули меняются на единицы, а единицы меняются на нули), полученное двоичное число переводится в десятичную систему, к нему прибавляется единица и добавляется знак «минус».

Чтобы определить двоичный код для отрицательного числа (заданного в десятичной системе), модуль числа переводим в двоичный код, инвертируем этот код и прибавляем единицу.

Таблица 1.5. Побитовые операторы Java

Оператор	Описание
`&`	Бинарный оператор *побитовое и*. Результатом выражения вида `a&b` с целочисленными операндами `a` и `b` является целое число. Биты в этом числе вычисляются сравнением соответствующих битов в представлении значений операндов `a` и `b`. Если каждый из двух сравниваемых битов равен `1`, то результатом (значение бита) является `1`. В противном случае соответствующий бит равен `0`
`|`	Бинарный оператор *побитовое или*. Результатом выражения вида `a|b` с целочисленными операндами `a` и `b` является целое число. Биты в этом числе вычисляются сравнением соответствующих битов в представлении значений операндов `a` и `b`. Если хотя бы один из двух сравниваемых битов равен `1`, то результатом (значение бита) является `1`. В противном случае результат (значение бита) равен `0`
`^`	Бинарный оператор *побитовое исключающее или*. Результатом выражения вида `a^b` с целочисленными операндами `a` и `b` является целое число. Биты в числе-результате вычисляются сравнением соответствующих битов в представлении значений операндов `a` и `b`. Если один и только один из двух сравниваемых битов равен `1`, то результатом (значением бита) является `1`. В противном случае результат (значение бита) равен `0`

Оператор	Описание
~	Унарный оператор *побитовая инверсия*. Результатом выражения вида `~a` является целое число, бинарный код которого получается заменой в бинарном коде операнда `a` нулей на единицы и единиц на нули
>>	Бинарный оператор *сдвиг вправо*. Результатом выражения вида `a>>n` является число, получаемое сдвигом вправо в бинарном представлении первого операнда `a` на количество битов, определяемых вторым операндом `n`. Исходное значение первого операнда при этом не меняется. Младшие биты теряются, а старшие заполняются значением знакового бита
<<	Бинарный оператор *сдвиг влево*. Результатом выражения вида `a<<n` является число, получаемое сдвигом влево в бинарном представлении первого операнда `a` на количество битов, определяемых вторым операндом `n`. Исходное значение первого операнда при этом не меняется. Младшие биты заполняются нулями, а старшие теряются
>>>	Бинарный оператор *беззнаковый сдвиг вправо*. Результатом выражения вида `a>>>n` является число, получаемое сдвигом вправо в позиционном представлении первого операнда `a` на количество битов, определяемых вторым операндом `n`. Исходное значение первого операнда при этом не меняется. Младшие биты теряются, а старшие заполняются нулями

НА ЗАМЕТКУ

Для бинарных побитовых операторов (как и для арифметических) существуют составные операторы присваивания. Так, выражение вида `a&=b` является эквивалентом (с поправкой на приведение типа) команды `a=a&b`, выражение `a^=b` эквивалентно команде `a=a^b`, и так далее.

Помимо перечисленных выше операторов, в Java есть единственный *тернарный оператор* (у оператора три операнда). Формально оператор обозначается как `?:`. Синтаксис вызова этого оператора следующий:

```
условие?значение:значение
```

Первым операндом указывается некоторое `условие` (выражение, возвращающее логическое значение). Если условие истинно (значение `true`), тернарный оператор результатом возвращает `значение`, указанное после вопросительного знака. Если условие ложно (значение `false`), тернарный оператор результатом возвращает `значение` после двоеточия.

ПОДРОБНОСТИ

Несколько замечаний по поводу оператора присваивания `=`. В Java оператор присваивания возвращает значение. Команда вида `x=y` выполняется следующим образом: вычисляется значение `y` и оно записывается в переменную `x`. Это же значение является значением всего выражения `x=y`. Поэтому допустимой является,

например, команда вида x=y=z. В этом случае значение переменной z присваивается сначала переменной y, а затем значение выражения y=z (оно же значение переменной y) присваивается переменной x.

В табл. 1.6 приведены данные о приоритете различных операторов в Java.

Таблица 1.6. Приоритеты операторов в Java

Приоритет	Операторы
1	*Круглые скобки* ( ), *квадратные скобки* [ ] и оператор *«точка»*
2	*Инкремент* ++, *декремент* --, *побитовая инверсия* ~ и *логическое отрицание* !
3	*Умножение* *, *деление* / и *вычисление остатка* %
4	*Сложение* + и *вычитание* -
5	Побитовые *сдвиги* >>, << и >>>
6	*Больше* >, *больше или равно* >=, *меньше или равно* <= и *меньше* <
7	*Равно* == и *не равно* !=
8	*Побитовое и* &
9	*Побитовое исключающее или* ^
10	*Побитовое или* |
11	*Логическое и* &&
12	*Логическое или* ||
13	*Тернарный оператор* ?:
14	*Присваивание* = и *составные операторы* вида op=

Операторы равных приоритетов (за исключением присваивания) выполняются слева направо. В случаях, когда возникают сомнения в приоритете операторов и последовательности вычисления выражений, рекомендуется использовать круглые скобки.

Использование основных операторов

Товарищ, там человек говорит, что он инопланетянин. Надо что-то делать.

из к/ф «Кин-дза-дза»

Далее рассмотрим некоторые задачи, которые иллюстрируют возможности Java и специфику синтаксиса этого языка.

Полет тела, брошенного под углом к горизонту

Составим программу для вычисления координат тела, брошенного под углом к горизонту. Полагаем, что известны масса тела m, начальная скорость V, угол α, под которым тело брошено к горизонту. Кроме того, считаем, что на тело действует сила сопротивления воздуха, по модулю пропорциональная скорости тела и направленная противоположно к направлению полета тела. Коэффициент пропорциональности для силы сопротивления воздуха γ также считаем известным.

В программе используем известное аналитическое решение для зависимостей координат тела от времени. В частности, для горизонтальной координаты (расстояние от точки бросания до тела вдоль горизонтали) имеем зависимость:

$$x(t) = \frac{Vm\cos(\alpha)}{\gamma}\left(1 - \exp\left(-\frac{\gamma t}{m}\right)\right).$$

Для вертикальной координаты (высота тела над горизонтальной поверхностью) зависимость следующая:

$$y(t) = \frac{m(V\sin(\alpha)\gamma + mg)}{\gamma^2}\left(1 - \exp\left(-\frac{\gamma t}{m}\right)\right) - \frac{mgt}{\gamma}.$$

Здесь через g обозначено ускорение свободного падения. Этими соотношениями воспользуемся при создании программы (листинг 1.4).

Листинг 1.4. Вычисление координат тела

```
// Статический импорт:
import static java.lang.Math.*;
class Demo{
   public static void main(String args[]){
      // Ускорение свободного падения:
      final double g=9.8;
      // Угол к горизонту (в градусах):
      double alpha=30;
      // Масса тела (в килограммах):
      double m=0.1;
      // Коэффициент сопротивления воздуха (Н*с/м):
      double gamma=0.1;
      // Скорость тела (м/с):
      double V=100.0;
      // Время (в секундах):
      double t=1.0;
      // Координаты тела (в метрах):
      double x,y;
      // Перевод градусов в радианы:
      alpha/=180/PI;
      // Вычисление координат:
```

```
      x=V*m*cos(alpha)/gamma*(1-exp(-gamma*t/m));
      y=m*(V*sin(alpha)*gamma+m*g)/gamma/gamma*
        (1-exp(-gamma*t/m))-m*g*t/gamma;
      // Вывод информации на экран:
      System.out.println(
         "Координаты тела для t="+t+" сек:\nx="+x+
         " м\ny="+y+" м");
      System.out.println("Параметры:");
      System.out.println(
         "Угол alpha="+alpha/PI*180+" градусов"
      );
      System.out.println("Скорость V="+V+" м/с");
      System.out.println(
         "Коэффициент сопротивления gamma="+gamma+" Н*с/м"
      );
      System.out.println("Масса тела m="+m+" кг");
   }
}
```

В результате выполнения программы появляются такие сообщения:

Результат выполнения программы (из листинга 1.4)

```
Координаты тела для t=1.0 с:
x=54.74324621999467 м
y=28.000809417947753 м
Параметры:
Угол alpha=30.0 градусов
Скорость V=100.0 м/с
Коэффициент сопротивления gamma=0.1 Н*с/м
Масса тела m=0.1 кг
```

Математические функции для вычисления синуса, косинуса и экспоненты реализованы как статические методы в классе `Math`. Мы используем статический импорт методов из данного класса.

ПОДРОБНОСТИ

Класс `Math` доступен в программе и без импорта. Статический импорт мы используем для того, чтобы не указывать название класса при вызове методов. Другими словами, если бы мы не использовали статический импорт, то методы из класса нам пришлось бы вызывать в формате `Math.sin()`, `Math.cos()` и `Math.exp()`.

Кроме статических методов, в классе `Math` описана константа `PI` со значением числа $\pi \approx 3{,}141592$.

Программа простая: объявляется несколько переменных, которым при объявлении сразу присваиваются значения (начальная скорость `V`, угол в градусах `alpha`, под которым брошено тело, коэффициент сопротивления `gamma`, а также масса тела `m`). Ускорение свободного падения реализуется через константу `g` (описана с ключевым

словом `final`). Переменная `t` определяет момент времени, для которого вычисляются координаты тела. Переменные `x` и `y` предназначены для записи в них значений координат тела. После присваивания этим переменным значения результаты вычислений выводятся на экран вместе с дополнительной информацией о массе тела, начальной скорости и так далее.

НА ЗАМЕТКУ

Здесь и далее для удобства восприятия информации некоторые слишком длинные команды разбиты на несколько строк. Подобное допустимо, код при этом остается рабочим.

Вычисление скорости на маршруте

В следующей программе вычисляется скорость движения автомобиля на маршруте, если известно, что автомобиль движется с постоянной известной скоростью между пунктами *А* и *Б*, расстояние между которыми тоже известно. Далее автомобиль движется от пункта *Б* до пункта *В* (расстояние между пунктами известно) с постоянной, но неизвестной скоростью. Ее необходимо вычислить, если известна средняя скорость движения автомобиля на маршруте от пункта *А* до пункта *В* (через пункт *Б*).

Если расстояние между пунктами *А* и *Б* обозначить через S_1, расстояние между пунктами *Б* и *В* — через S_2, скорость движения на этих участках — соответственно через V_1 и V_2, среднюю скорость движения на маршруте — через V, то неизвестную скорость V_2 движения на маршруте от *Б* до *В* можно вычислить по формуле:

$$V_2 = \frac{S_2}{\frac{S_1 + S_2}{V} - \frac{S_1}{V_1}}.$$

Проблема, однако, в том, что вычисленное по данной формуле значение для скорости может оказаться отрицательным. На самом деле это означает невозможность для автомобиля иметь указанную среднюю скорость. Другими словами, даже если бы автомобиль мгновенно переместился из пункта *Б* в пункт *В*, он настолько медленно проехал первый участок, что средняя скорость никак не может оказаться равной указанному значению. Учтем это обстоятельство при написании кода.

Некоторые замечания касаются самого процесса вычисления скорости. Удобно разбить процесс вычислений на несколько этапов. В частности, разумно предварительно вычислить время движения автомобиля по всему маршруту $T = (S_1 + S_2)/V$, а также время движения по первому участку $t = S_1/V_1$. Затем искомую скорость можно рассчитать по формуле $V_2 = S_2/(T - t)$. В листинге 1.5 приведен код для вычисления скорости движения автомобиля.

Листинг 1.5. Вычисление скорости автомобиля

```
class Demo{
   public static void main(String args[]){
      // Расстояние между объектами (км):
      double S1=100;
      double S2=200;
      // Скорость на первом участке (км/ч):
      double V1=80;
      // Средняя скорость (км/ч):
      double V=48;
      /* Скорость на втором участке, общее время движения
      и время движения на первом участке:*/
      double V2,T,t;
      // Общее время движения (ч):
      T=(S1+S2)/V;
      // Время движения на первом участке (ч):
      t=S1/V1;
      // Скорость движения на втором участке (км/ч):
      V2=T>t?(S1+S2)/(T-t):-1;
      System.out.println("Скорость на втором участке:");
      // Результат:
      System.out.println(
         V2<0?"Это невозможно!":V2+" км/ч"
      );
   }
}
```

Результат выполнения программы имеет следующий вид.

Результат выполнения программы (из листинга 1.5)

```
Скорость на втором участке:
60.0 км/ч
```

Если изменить значение средней скорости (переменная `V`) на `240` или большее (при неизменных прочих параметрах), получим иное сообщение:

Результат выполнения программы (из листинга 1.5)

```
Скорость на втором участке:
Это невозможно!
```

Значение скорости на втором участке в программе определяется с помощью тернарного оператора командой `V2=T>t?(S1+S2)/(T-t):-1`. Тернарный оператор здесь необходим исключительно с одной целью: предотвратить возможное деление на ноль при условии, что значения переменных `T` и `t` совпадают. Если общее время движения превышает время движения по первому участку, значение скорости автомобиля на втором участке вычисляется по приведенной выше формуле. Если данное условие не выполняется, то переменной `V2` для скорости на втором участке присваивается формальное отрицательное значение `-1`. В зависимости от значения перемен-

ной V2 либо выводится информация о вычисленном значении скорости на втором участке, либо появляется сообщение "Это невозможно!". Мы используем команду System.out.println(V2<0?"Это невозможно!":V2+" км/ч"), в которой аргументом методу println() передано выражение V2<0?"Это невозможно!":V2+" км/ч". В нем использован тернарный оператор. При отрицательном значении переменной V2 возвращается текстовое значение "Это невозможно!", а в противном случае возвращается текстовое значение, которое получается объединением (и преобразованием к текстовому формату) значения скорости и надписи " км/ч".

Орбита спутника

Следующая задача иллюстрирует работу с большими числами. Состоит она в вычислении высоты орбиты спутника над поверхностью Земли, если известны масса и радиус Земли, а также период обращения спутника вокруг Земли. В частности, используем такие значения универсальной гравитационной постоянной, массы Земли и радиуса Земли:

Универсальная гравитационная постоянная: $G \approx 6{,}672 \times 10^{-11}$ (Н м^2/кг^2).

Масса Земли: $M \approx 5{,}96 \times 10^{24}$ (кг).

Радиус Земли: $R \approx 6{,}37 \times 10^{6}$ (м).

Если через T обозначить период обращения спутника (в секундах), то высоту H спутника над поверхностью Земли можно вычислить по формуле:

$$H = \sqrt[3]{\frac{GMT^2}{4\pi^2}} - R.$$

Соответствующий код приведен в листинге 1.6.

Листинг 1.6. Орбита спутника

```
// Статический импорт:
import static java.lang.Math.*;
class Demo{
   public static void main(String args[]){
      // Гравитационная постоянная (Нм^2/кг^2):
      final double G=6.672E-11;
      // Масса Земли (кг):
      final double M=5.96e24;
      // Радиус Земли:
      final double R=6.37E6;
      // Период обращения спутника (ч):
      double T=1.5;
      // Высота над поверхностью:
      double H;
      // Перевод в секунды:
      T*=3600;
```

```
      // Высота в метрах:
      H=pow(G*M*T*T/4/PI/PI,(double)1/3)-R;
      // Высота в километрах с точностью до тысячных:
      H=(double)round(H)/1000;
      // Вывод результата на экран:
      System.out.println(
         "Высота орбиты спутника: "+H+" км"
      );
   }
}
```

В результате выполнения программы получаем следующее:

Результат выполнения программы (из листинга 1.5)

```
Высота орбиты спутника: 277.271 км
```

При инициализации переменных, определяющих параметры Земли и значение гравитационной постоянной, используется формат представления чисел в виде мантиссы и значения показателя степени (после литеры `E` или `e`). Поскольку время обращения спутника (переменная `T`) задается в часах, для перевода в секунды используем команду `T*=3600`. Высота вычисляется с помощью команды `H=pow(G*M*T*T/4/PI/PI,(double)1/3)-R`. Здесь мы использовали статический метод `pow()` из класса `Math` для возведения числа в степень. Первым аргументом указывается возводимое в степень число, вторым — показатель степени. Поскольку в программе использован статический импорт методов класса `Math`, то при вызове метода `pow()` (как и прочих статических методов из этого класса) можно явно не указывать класс `Math`. Также использована константа `PI` (полная ссылка на константу имеет вид `Math.PI`) для числа π. Кроме того, при вычислении показателя степени (второй аргумент метода `pow()` со значением 1/3) делятся два целых числа, а по умолчанию такое деление выполняется нацело. Чтобы деление выполнялось как обычное (не целочисленное), использована инструкция `(double)`.

После вычисления значения переменной `H` получаем высоту орбиты в метрах. Затем с помощью метода `round()` из класса `Math` это значение округляем (до целочисленного) и делим на `1000` для вычисления значения высоты орбиты в километрах. Поскольку методом `round()` возвращается целое число, при делении этого числа на `1000` по умолчанию также выполняется деление нацело. Поэтому перед выражением указана инструкция `(double)`, в результате чего значение переменной `H` получаем в километрах с точностью до сотых, то есть точность орбиты вычисляется с точностью до метра.

Комплексные числа

Рассмотрим программу, в которой вычисляется целочисленная степень комплексного числа. Напомним, что комплексным называется число вида $z = x + iy$, где x и y — действительные числа, а мнимая единица $i^2 = -1$. Величина $\mathrm{Re}(z) = x$ назы-

вается действительной частью комплексного числа, а величина $\mathrm{Im}(z) = y$ — мнимой. Модулем комплексного числа называется действительная величина $r = \sqrt{x^2 + y^2}$. Каждое комплексное число может быть представлено в тригонометрическом виде $z = r\exp(i\varphi) = r\cos(\varphi) + ir\sin(\varphi)$, где модуль комплексного числа r и аргумент φ связаны с действительной x и мнимой y частями комплексного числа соотношениями $x = r\cos(\varphi)$ и $y = r\sin(\varphi)$.

Если комплексное число $z = x + iy$ необходимо возвести в целочисленную степень n, результатом является комплексное число $z^n = r^n\exp(in\varphi) = r^n\cos(n\varphi) + ir^n\sin(n\varphi)$. Этим соотношением воспользуемся в программе для вычисления целочисленной степени комплексного числа. Программа представлена в листинге 1.7.

Листинг 1.7. Возведение комплексного числа в степень

```
import static java.lang.Math.*;
class Demo{
   public static void main(String args[]){
      double x=1.0,y=-1.0;
      int n=5;
      double r,phi;
      double Re,Im;
      r=sqrt(x*x+y*y);
      phi=atan2(y,x);
      Re=pow(r,n)*cos(n*phi);
      Im=pow(r,n)*sin(n*phi);
      System.out.println("Re="+Re);
      System.out.println("Im="+Im);
   }
}
```

В программе на основании действительной (переменная `x`) и мнимой (переменная `y`) части исходного комплексного числа вычисляется модуль (переменная `r`) и аргумент (переменная `phi`) этого числа. Затем вычисляется действительная (переменная `Re`) и мнимая (переменная `Im`) часть комплексного числа, которое получается возведением в целочисленную степень (переменная `n`) исходного числа. Для вычисления квадратного корня использован статический метод `sqrt()` из класса `Math`. Аргумент комплексного числа вычисляется с помощью статического метода `atan2()` из этого же класса. Аргументами методу `atan2()` передаются ордината и абсцисса точки на плоскости, а результатом метод возвращает угол между осью абсцисс и лучом в направлении на точку. Это означает, что если мы передадим методу мнимую и действительную часть комплексного числа, то результатом будет аргумент комплексного числа.

НА ЗАМЕТКУ

Комплексное число можно представить как точку на плоскости. Действительная часть числа определяет горизонтальную координату точки (абсцисса). Мнимая часть числа определяет вертикальную координату точки (ордината). Расстояние

от начала координат до точки определяет модуль числа, а аргумент числа равен углу между осью абсцисс (горизонтальная координатная ось) и лучом, который выходит из начала координат в направлении на точку.

Возведение в целочисленную степень выполняется с помощью статического метода `pow()` (первый аргумент — возводимое в степень число, второй аргумент — степень, в которую возводится число).

После выполнения всех необходимых расчетов получаем:

Результат выполнения программы (из листинга 1.7)

```
Re=-4.000000000000003
Im=4.000000000000001
```

Справедливости ради следует отметить, что для работы с комплексными числами все же лучше использовать классы и объекты.

Прыгающий мячик

Упругий мячик бросают под углом к горизонту с некоторой начальной скоростью. При падении мячика на ровную горизонтальную поверхность происходит упругое отбивание, так что горизонтальная составляющая скорости мячика не меняется, а вертикальная меняется на противоположную. Представленная далее программа позволяет вычислить положение (координаты) мячика в произвольный момент времени.

Мы воспользуемся тем, что если в начальный момент времени (при $t = 0$) скорость мячика по модулю равна V, а угол к горизонту равен α, то закон движения для горизонтальной координаты имеет вид

$$x(t) = tV\cos(\alpha).$$

Для вертикальной координаты соответствующая зависимость может быть записана так:

$$y(t) = (t - T)V\sin(\alpha) - g(t - T)^2/2.$$

Здесь через T обозначено время последнего на данный момент удара о землю. Поскольку время между ударами может быть определено как $T_0 = 2V\sin(\alpha)/g$, $T = T_0[t/T_0]$. В данном случае квадратные скобки означают вычисление целой части от внутреннего выражения. Соответствующий код представлен в листинге 1.8.

Листинг 1.8. Прыгающий мячик

```
import static java.lang.Math.*;
class Demo{
   public static void main(String args[]){
      // Ускорение свободного падения, м/с^2:
      final double g=9.8;
```

```
        // Начальная скорость, м/с:
        double V=10;
        // Угол в градусах:
        double alpha=30;
        // Время в секундах:
        double t=5;
        // Расчетные параметры:
        double T0,T,x,y;
        // Перевод угла в радианы
        alpha=toRadians(alpha);
        // Время между ударами о поверхность:
        T0=2*V*Math.sin(alpha)/g;
        // Момент последнего удара о поверхность:
        T=T0*floor(t/T0);
        // Горизонтальная координата:
        x=V*cos(alpha)*t;
        // Высота над поверхностью:
        y=V*sin(alpha)*(t-T)-g*(t-T)*(t-T)/2;
        // Округление значений:
        x=round(100*x)/100.0;
        y=round(100*y)/100.0;
        // Вывод результатов:
        System.out.println("x("+t+")="+x+" м");
        System.out.println("y("+t+")="+y+" м");
    }
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 1.8)

```
x(5.0)=43.3 м
y(5.0)=0.46 м
```

В начале программы задаются значения ускорения свободного падения (константа `g`), начальная скорость мячика (переменная `V`), угол в градусах, под которым тело брошено к горизонту (переменная `alpha`), и момент времени, для которого вычисляются координаты мячика (переменная `t`). Переменные `T0` и `T` используются для записи в них значений времени полета мячика между ударами о поверхность и времени последнего удара соответственно. В переменные `x` и `y` записываются значения координат мячика (эти значения и нужно вычислить в программе).

Поскольку угол задан в градусах, его необходимо перевести в радианы. В данном случае — посредством команды `alpha=toRadians(alpha)`, которой вызывается статический метод `toRadians()` (из класса `Math`), предназначенный именно для этих целей.

Время полета между двумя последовательными ударами мячика о поверхность вычисляется командой `T0=2*V*sin(alpha)/g`. Момент времени для последнего удара определяется с помощью команды `T=T0*floor(t/T0)`. Здесь использован статический метод `floor()` из класса `Math`, который результатом возвращает наибольшее целое число, не превышающее аргумент функции. Координаты мячика вычисляются

командами `x=V*cos(alpha)*t` и `y=V*sin(alpha)*(t-T)-g*(t-T)*(t-T)/2`. Округление этих значений до сотых выполняется инструкциями `x=round(100*x)/100.0` и `y=round(100*y)/100.0`. Для округления использован статический метод `round()`, который округляет до ближайшего целого значения. После округления результат вычислений отображается в окне вывода.

Решение тригонометрического уравнения

В следующем примере программными методами решается уравнение вида

$$a\cos(x) + b\sin(x) = c.$$

Это уравнение можно записать так:

$$\sin(x+\alpha) = \frac{c}{\sqrt{a^2+b^2}},$$

где $\sin(\alpha) = a / \sqrt{a^2+b^2}$. Поэтому формально решение исходного уравнения для любого целого n дается выражением

$$x = -\arcsin\left(\frac{a}{\sqrt{a^2+b^2}}\right) + (-1)^n \arcsin\left(\frac{c}{\sqrt{a^2+b^2}}\right) + \pi n.$$

В программе, представленной далее, по значениям параметров a, b и c вычисляется решение (разумеется, если оно существует) уравнения для случая $n = 0$:

$$x = \arcsin\left(\frac{c}{\sqrt{a^2+b^2}}\right) - \arcsin\left(\frac{a}{\sqrt{a^2+b^2}}\right).$$

При этом проверяется условие существования решения $a^2+b^2 \geq c^2$. Если данное условие не выполняется, то уравнение решений не имеет.

НА ЗАМЕТКУ

Экзотический случай, когда $a = b = c = 0$ (при таких условиях решением является любое значение для x), в программе не отслеживается.

Рассмотрим код из листинга 1.9.

Листинг 1.9. Вычисление корня уравнения

```
import static java.lang.Math.*;
class Demo{
   public static void main(String args[]){
      // Параметры уравнения:
```

```
      double a=5;
      double b=3;
      double c=1;
      // Вспомогательная переменная:
      double alpha;
      // Логическая переменная – критерий наличия решений:
      boolean state;
      // Значение вспомогательной переменной:
      alpha=asin(a/sqrt(a*a+b*b));
      // Вычисление критерия:
      state=a*a+b*b>=c*c;
      // Вывод на экран значений исходных параметров:
      System.out.println("Уравнение a*cos(x)+b*sin(x)=c");
      System.out.println("Параметры:");
      System.out.println("a="+a);
      System.out.println("b="+b);
      System.out.println("c="+c);
      System.out.print("Решение для x: ");
      // Вычисление решения уравнения и вывод на экран:
      System.out.println(
         state?asin(c/sqrt(a*a+b*b))-alpha:"решений нет!"
      );
   }
}
```

Основное место в программе — использование тернарного оператора в последней команде вывода значения для корня уравнения (предварительно выводится справочная информация о значениях параметров уравнения), где аргументом метода `println()` указано выражение `state?asin(c/sqrt(a*a+b*b))-alpha:"решений нет!"`. В тернарном операторе проверяемым условием является логическая переменная `state`. Значение этой переменной присваивается командой `state=a*a+b*b>=c*c`. Значение равно `true`, если уравнение имеет решения, и `false` — если не имеет. Если значение переменной `state` равно `true`, то тернарным оператором в качестве результата возвращается числовое значение `asin(c/sqrt(a*a+b*b))-alpha`, где переменной `alpha` предварительно присвоено значение командой `alpha=asin(a/sqrt(a*a+b*b))`. В этих выражениях использованы статические методы `asin()` и `sqrt()` из класса `Math` для вычисления арксинуса и квадратного корня соответственно. Таким образом, при истинном условии значением возвращается решение уравнения. Если условие (переменная `state`) равно `false`, то результатом возвращается текст `"решений нет!"`.

НА ЗАМЕТКУ

При разных значениях условия тернарный оператор возвращает не просто разные значения, а значения разного типа. Но проблем в данном случае не возникает, поскольку выражение на основе тернарного оператора передано аргументом методу `println()`. В этом случае результат тернарного оператора, каким бы он ни был, автоматически преобразуется в текстовый формат.

Результат выполнения программы будет таким:

Результат выполнения программы (из листинга 1.9)

```
Уравнение a*cos(x)+b*sin(x)=c
Параметры:
a=5.0
b=3.0
c=1.0
Решение для x: -0.8580262366249893
```

Если поменять значения исходных параметров уравнения (переменной `c` присвоить значение `10`), получим следующее:

Результат выполнения программы (из листинга 1.9)

```
Уравнение a*cos(x)+b*sin(x)=c
Параметры:
a=5.0
b=3.0
c=10.0
Решение для x: решений нет!
```

Хотя использование тернарного оператора может быть достаточно эффективным, обычно такие задачи решаются с помощью условного оператора и оператора выбора, которые описываются в следующей главе.

Кодирование символов

Рассмотрим простую иллюстративную программу, в которой для записи сразу двух символьных значений типа `char` используется одна переменная целочисленного типа `int`.

В программе учитывается то обстоятельство, что тип `int` в Java имеет размер 32 бита, а для записи основных символов достаточно 16 бит. Таким образом, объем памяти, выделяемой переменной типа `int`, достаточен для записи по меньшей мере двух символов (значений типа `char`). Принцип записи символьных значений в виде числа следующий: начальные 16 битов числа (младшие биты) будут содержать код первого символа, а следующие 16 битов числа (старшие биты) — код второго символа (листинг 1.10).

Листинг 1.10. Кодирование символов

```
class Demo{
   public static void main(String args[]){
      // Кодовое число:
      int num;
      // Исходные буквы для кодирования:
      char A='А',B='ы';
      // Буквы после декодирования:
```

```
        char X,Y;
        // Вычисление кода:
        num=((int)B<<16)+((int)A);
        // Вывод исходных данных и кода:
        System.out.println(
           "Исходные буквы: \'"+A+"\' и \'"+B+"\'."
        );
        System.out.println("Кодовое число: "+num);
        // Декодирование:
        Y=(char)(num>>>16);
        X=(char)(num^((int)Y<<16));
        // Вывод результата декодирования:
        System.out.println("Обратное преобразование:");
        System.out.println("Буквы \'"+X+"\' и \'"+Y+"\'.");
    }
}
```

Целочисленная переменная `num` предназначена для записи в нее числового кода, который формируется на основе значений переменных `A` и `B` типа `char`. После того как код создан и записан в переменную `num`, выполняется обратная операция: на основании значения переменной `num` восстанавливаются исходные символы, а результат записывается в переменные `X` и `Y` типа `char`.

Значение переменной `num` вычисляется командой `num=((int)B<<16)+((int)A)`. Правая часть является суммой двух слагаемых. Первое слагаемое `(int)B<<16` представляет собой смещенный влево на `16` битов числовой код символа, записанного в переменную `B`. Для получения кода символа использована инструкция `(int)` явного приведения типов. То есть инструкцией `(int)B` получаем код символа, после чего с помощью оператора сдвига `<<` смещаем код на `16` позиций влево с заполнением младших `16` битов нулями. В эти биты записывается код символа из переменной `A`. Для этого к полученному на первом этапе коду прибавляется значение `(int)A`, то есть код первого символа.

ПОДРОБНОСТИ

Символьное значение типа `char` запоминается в виде кода из 16 битов. Если мы символ преобразуем в тип `int`, то результат записывается с помощью 32 битов. В таком случае 16 младших битов — это код символа, а 16 старших битов содержат нули. Значение выражения `(int)A` — это число, записанное 32 битами. Младшие 16 битов содержат код символа из переменной `A`, а старшие 16 битов заполнены нулями.

Исходные символы и полученный на их основе числовой код отображаются в окне вывода. Затем начинается обратная процедура по извлечению символов из числового кода. Для этого командой `Y=(char)(num>>>16)` считывается второй символ и записывается в переменную `Y`. Действительно, результатом инструкции `num>>>16` является смещенный вправо на 16 битов код переменной `num` (с заполнением старших битов нулями), то есть код второго символа (того, что записан

в переменную B). Первый символ считывается немного сложнее. В частности, используется команда X=(char)(num^((int)Y<<16)). Здесь следует учесть, что результатом инструкции (int)Y<<16 является код уже считанного второго символа, смещенный влево на 16 битов. По сравнению с кодом, записанным в переменную num, он отличается тем, что его младшие 16 битов — нулевые, в то время как в переменной num эти биты содержат код первого символа (из переменной A). Старшие 16 битов в обоих значениях совпадают. Указанные два кода являются операндами в выражении на основе оператора ^ (*побитовое исключающее или*). В результате такой операции сравниваются соответствующие биты. Результатом (значение бита) является единица, если один и только один из двух сравниваемых битов равен единице. Для совпадающих старших битов это означает полное обнуление, а младшие единичные биты выживают, поэтому на выходе получаем код, записанный в младшие 16 битов (то есть код первого символа). Сам символ получаем с помощью явного приведения к символьному типу. После выполненного декодирования символы отображаются в окне вывода. Результат выполнения программы такой:

Результат выполнения программы (из листинга 1.10)

```
Исходные буквы: 'А' и 'ы'.
Кодовое число: 72025104
Обратное преобразование:
Буквы 'А' и 'ы'.
```

Видим, что исходные символы совпадают с теми, что получены в результате декодирования целого числа.

Расчет параметров цепи

Далее представлена программа, имеющая отношение к вычислению сопротивления участка электрической цепи. Предположим, что участок цепи состоит из двух блоков, в каждом из которых располагаются два параллельно соединенных резистора. Блоки между собой соединены последовательно. Имеется три резистора известного сопротивления, которые можно свободно переставлять между блоками, и один основной резистор, который должен находиться во втором блоке. Необходимо определить, какой резистор вставить во второй блок в дополнение к основному, чтобы общее сопротивление участка цепи было минимальным.

Если сопротивления трех переставляемых резисторов обозначить как R_1, R_2 и R_3, а сопротивление основного резистора как R, то при условии, что первые два резистора подключаются в первый блок, а третий резистор — во второй, общее сопротивление участка цепи будет составлять величину

$$r = \frac{R_1 R_2}{R_1 + R_2} + \frac{R R_3}{R + R_3}.$$

Для определения оптимального способа подключения резисторов нужно проверить три варианта, когда каждый из трех резисторов включается во второй блок, и выбрать тот вариант подключения, при котором общее сопротивление минимально. Соответствующий код представлен в листинге 1.11.

Листинг 1.11. Оптимальное подключение резисторов

```
class Demo{
   public static void main(String args[]){
      // Сопротивление резисторов (Ом):
      double R1=3,R2=5,R3=2,R=1;
      // Расчетные значения для сопротивления
      // участка цепи (Ом):
      double r1,r2,r3;
      // Логические значения для определения
      // способа подключения:
      boolean A,B;
      // Вычисление сопротивления участка цепи
      // для разных способов подключения:
      r1=R2*R3/(R2+R3)+R1*R/(R1+R);
      r2=R1*R3/(R1+R3)+R2*R/(R2+R);
      r3=R2*R1/(R2+R1)+R3*R/(R3+R);
      // Вычисление критериев для способа подключения:
      A=(r1<=r2)&&(r1<=r3);
      B=(r2<=r1)&&(r2<=r3);
      // Вывод начальных значений:
      System.out.println(
         "Значения сопротивлений резисторов:"
      );
      System.out.println("Первый  R1="+R1+" Ом");
      System.out.println("Второй  R2="+R2+" Ом");
      System.out.println("Третий  R3="+R3+" Ом");
      System.out.println("Основной R="+R3+" Ом");
      // Вычисление и вывод результата:
      System.out.print("Во второй блок подключается ");
      System.out.print(A?"первый":B?"второй":"третий");
      System.out.println(" резистор!");
   }
}
```

В результате выполнения программы получаем следующие сообщения:

Результат выполнения программы (из листинга 1.11)

```
Значения сопротивлений резисторов:
Первый R1=3.0 Ом
Второй R2=5.0 Ом
Третий R3=2.0 Ом
Основной R=2.0 Ом
Во второй блок подключается второй резистор!
```

В программе объявляются и инициализируются переменные `R1`, `R2`, `R3`, `R` типа `double`, определяющие сопротивления трех переставляемых резисторов и основного

резистора соответственно. Переменные r1, r2 и r3 типа double предназначены для вычисления и записи в них значения сопротивления участка цепи для каждого из трех возможных способов подключения резисторов. Также в программе объявляются две логические переменные A и B (типа boolean). Значения этих переменных определяются командами A=(r1<=r2)&&(r1<=r3) и B=(r2<=r1)&&(r2<=r3). Значение переменной A равно true, если при первом способе подключения резисторов (во втором блоке размещается первый резистор) общее сопротивление цепи не превышает сопротивление цепи для второго и третьего способов подключения резисторов. Значение переменной B равно true, если при втором способе подключения резисторов (во втором блоке размещается второй резистор) общее сопротивление цепи не превышает сопротивление цепи для первого и третьего способов подключения резисторов. Понятно, что если обе эти переменные равны false, то оптимальным является третий способ подключения резисторов (во втором блоке размещается третий резистор).

После вычисления значений переменных A и B выполняется вывод результата. Сначала серией команд отображаются текущие значения для сопротивлений резисторов. Затем выводится начало фразы о способе подключения резисторов. Номер резистора (в текстовом формате) определяется непосредственно в аргументе метода println() командой A?"первый":B?"второй":"третий", в которой использованы вложенные тернарные операторы. Если значение переменной A (условие внешнего тернарного оператора) равно true, то возвращается второй операнд внешнего тернарного оператора — текстовое значение "первый". В противном случае вычисляется третий операнд внешнего тернарного оператора. Третьим операндом является тернарный оператор B?"второй":"третий". Если значение переменной B равно true, то возвращается текст "второй", в противном случае — текст "третий". После того как нужное слово (название резистора) выведено на экран, следующими командами завершается выведение финальной фразы.

Резюме

> Только быстро. А то одна секунда здесь — это полгода там.
>
> *из к/ф «Кин-дза-дза»*

- Java является полностью объектно-ориентированным. Для создания даже самой простой программы необходимо описать по крайней мере один класс. Этот класс содержит главный метод со стандартным названием main(). Выполнение программы отождествляется с выполнением главного метода.
- В методе main() можно объявлять переменные. Для объявления переменной указывается тип переменной и ее имя. Переменной одновременно с объявлением можно присвоить значение (инициализировать переменную). Переменная должна быть инициализирована до ее первого использования.

- Существует несколько базовых типов данных. При вычислении выражений выполняется автоматическое преобразование типов. Особенность автоматического преобразования типов в Java состоит в том, что оно осуществляется без потери значений. Также можно выполнять явное приведение типов, для чего перед выражением в круглых скобках указывается идентификатор соответствующего типа.
- Основные операторы Java делятся на арифметические, логические, побитовые и операторы сравнения. Арифметические операторы предназначены для выполнения таких операций, как сложение, вычитание, деление и умножение. Логические операторы предназначены для работы с логическими операндами и позволяют выполнять операции *отрицания*, *или*, *и*, *исключающего или*. Операторы сравнения используются, как правило, при сравнении (на предмет равенства или неравенства) числовых значений. Результатом сравнения является логическое значение (значение логического типа). Побитовые операторы служат для выполнения операций на уровне битовых представлений чисел.
- В Java существуют составные операторы присваивания. Команда вида `x =x op y` может быть записана как `x op=y`, где через `op` обозначен арифметический или побитовый оператор.
- В Java есть тернарный оператор, который представляет собой упрощенную форму условного оператора. Первым его операндом указывается логическое выражение (условие). В зависимости от значения условия результатом возвращается значение второго или третьего операнда.

2 Управляющие инструкции Java

Пошли, Скрипач, в открытый космос.

из к/ф «Кин-дза-дза»

В этой главе мы рассмотрим операторы цикла, условный оператор и оператор выбора. Все эти синтаксические конструкции позволяют создавать в программе точки ветвления и многократно выполнять блоки команд. Обычно их называют *управляющими инструкциями*. Без этих инструкций попросту невозможно создавать эффективные программные коды.

Условный оператор if

Хорошо, допустим, Земля вращается вокруг Солнца.

из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Как отмечалось в предыдущей главе, кроме тернарного оператора в Java имеется более функциональная конструкция. Речь об *условном операторе* со следующим синтаксисом:

```
if(условие){
   // Команды
}
else{
   // Команды
}
```

После ключевого слова `if` в круглых скобках указывается условие.

НА ЗАМЕТКУ

Условие, указанное в операторе `if`, представляет собой выражение, возвращающее результатом значение логического типа `boolean`. Это же замечание относится к условиям, используемым в операторах цикла.

Выполнение условного оператора начинается с проверки этого условия. Если условие истинно (значение выражения в круглых скобках после ключевого слова `if` равно `true`), выполняются команды, указанные в фигурных скобках сразу после инструкции `if(условие)`. Если условие ложно (значение выражения в круглых скобках равно `false`), то выполняются команды, размещенные в блоке (выделенном фигурными скобками) после ключевого слова `else`. После выполнения условного оператора управление передается следующей после него команде (рис. 2.1).

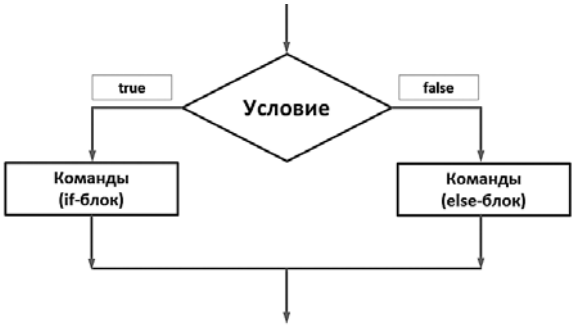


Рис. 2.1. Схема работы условного оператора

НА ЗАМЕТКУ

Условный оператор работает по следующей схеме: есть два блока команд и есть условие. Если условие истинно, то выполняется один блок команд. Если условие ложно, то выполняется другой блок команд.

Если любой из двух блоков команд (в `if`-блоке или в `else`-блоке) состоит из одной команды, то фигурные скобки для соответствующего блока можно не использовать. Тем не менее фигурные скобки улучшают читабельность программы.

У условного оператора есть упрощенная форма, в которой не используется `else`-блок (проще говоря, в условном операторе `else`-блок не является обязательным). Синтаксис упрощенной формы условного оператора имеет следующий вид:

```
if(условие){
   // Команды
}
```

В этом случае сначала проверяется на истинность условие, указанное в скобках после ключевого слова `if`. Если условие истинно, то выполняется расположенный далее блок команд. Если условие ложно, то ничего не происходит (рис. 2.2).

НА ЗАМЕТКУ

Принцип работы упрощенной формы условного оператора следующий: есть блок команд, которые выполняются, только если истинно некоторое условие.

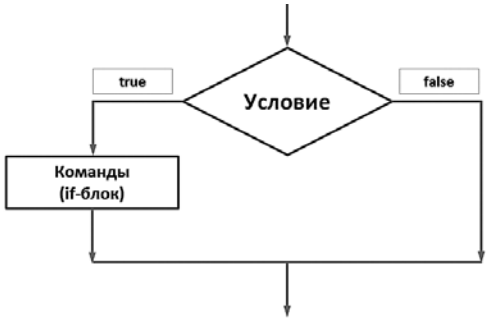


Рис. 2.2. Принцип работы упрощенной формы условного оператора

На практике нередко используются вложенные `if`-операторы. С точки зрения синтаксиса языка Java такая ситуация проста: в `else`-блоке условного оператора размещается другой условный оператор, а в его `else`-блоке размещается еще один условный оператор, и так далее. Но при этом вся конструкция выглядит достаточно элегантно:

```
if(условие){
   // Команды
}else if(условие){
   // Команды
}else if(условие){
   // Команды
}
// Продолжение
else if(условие){
   // Команды
}else{
   // Команды
}
```

Принцип выполнения вложенных условных операторов следующий. Сначала проверяется условие в первом (внешнем) `if`-операторе. Если оно истинно, то выполняются соответствующие команды. Если условие ложно, то проверяется условие во втором (внутреннем) условном операторе в `else`-блоке. При истинном условии в этом операторе выполняются команды в его `if`-блоке. Если условие ложно, то начинает выполняться третий условный оператор в `else`-блоке второго оператора, и так далее. Получается, что условия проверяются последовательно, одно за другим, если все предыдущие условия оказались ложными. Последний `else`-блок (если он есть) выполняется, если все условия оказались ложными.

В листинге 2.1 представлен пример несложной программы, в которой используется условный оператор. В программе генерируется случайное число (в диапазоне значений от `1` до `5` включительно). Пользователю предлагается угадать это число (указывается в поле ввода диалогового окна). В зависимости от того, угадал пользователь или нет, появляется сообщение соответствующего содержания.

Листинг 2.1. Знакомство с условным оператором

```
import static javax.swing.JOptionPane.*;
import static java.lang.Integer.parseInt;
import static java.lang.Math.random;
class Demo{
   public static void main(String[] args){
      int num,ans,icon;
      String txt;
      // Случайное целое число (от 1 до 5):
      num=(int)(5*random())+1;
      // Считывание целого числа:
      ans=parseInt( // Преобразование текста в число
         showInputDialog(null, // Родительское окно
            // Текст над полем ввода:
            "Угадайте число (от 1 до 5):",
            "Число", // Заголовок окна
            QUESTION_MESSAGE // Тип пиктограммы
         )
      );
      // Условный оператор:
      if(ans==num){
         txt="Вы угадали! Это число "+num+"!";
         icon=INFORMATION_MESSAGE;
      }else{
         txt="Вы не угадали! Это число "+num+"!";
         icon=ERROR_MESSAGE;
      }
      showMessageDialog(null, // Родительское окно
         txt, // Текст сообщения
         "Результат", // Заголовок окна
         icon // Тип пиктограммы
      );
   }
}
```

При запуске программы появляется окно (рис. 2.3).

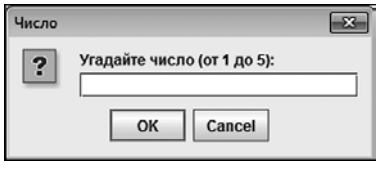


Рис. 2.3. Окно с полем для ввода числа

Пользователь должен ввести число в поле диалогового окна и нажать кнопку OK. Если пользователь угадал число, то появится диалоговое окно (рис. 2.4).

Если пользователь не угадал число, то окно будет иметь иной вид (рис. 2.5).

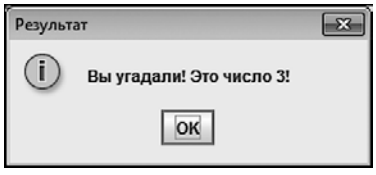


Рис. 2.4. Окно появляется, если пользователь угадал число

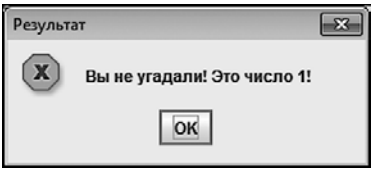


Рис. 2.5. Окно появляется, если пользователь не угадал число

Теперь проанализируем код, выполнение которого приводит к таким результатам.

В программе генерируется целое случайное число, и это число записывается в целочисленную переменную `num`. Мы использовали команду `num=(int)(5*random())+1`, задействовав статический метод `random()` из класса `Math` (для импорта метода использован статический импорт). Метод возвращает результатом случайное действительное число от `0` (включительно) до `1` (строго меньше). Умножив результат вызова метода на `5`, получим случайное действительное число от `0` (включительно) до `5` (строго меньше). После приведения этого значения к целочисленному формату (инструкция `(int)`) получаем целое число в диапазоне от `0` до `4` включительно. После прибавления единицы получаем число в диапазоне значений от `1` до `5`.

Число, которое вводит пользователь, записывается в целочисленную переменную `ans`. Для отображения диалогового окна с полем ввода мы вызываем статический метод `showInputDialog()` из класса `JOptionPane`. Но на этот раз методу передается не один (как было раньше), а четыре аргумента. Первый аргумент `null` является пустой ссылкой на родительское окно (такого окна нет). Второй текстовый аргумент определяет надпись над полем ввода. Третий текстовый аргумент задает заголовок окна. Наконец, четвертым аргументом передана статическая константа `QUESTION_MESSAGE` из класса `JOptionPane`. Эта константа определяет тип пиктограммы, отображаемой в окне (в данном случае — пиктограмма со знаком вопроса). Но это не все. Даже если пользователь вводит в поле диалогового окна целое число, оно все равно считывается как текст. Это так называемое текстовое представление числа, то есть речь о тексте (например, `"3"`), который содержит число. Чтобы извлечь число из текста, используем статический метод `parseInt()` из класса-оболочки `Integer`. Инструкция с вызовом метода `showInputDialog()` указана аргументом метода `parseInt()`.

НА ЗАМЕТКУ

Для использования статических методов и констант из класса `JOptionPane` использован статический импорт всех статических полей и методов этого класса. Также мы использовали статический импорт для метода `parseInt()` из класса `Integer`.

Далее в игру вступает условный оператор. В нем проверяется условие `ans==num`, которое истинно в случае, если введенное пользователем число равно сгенерирован-

ному случайному числу. При истинном условии значение текстовой переменной `txt` присваивается командой `txt="Вы угадали! Это число "+num+"!"`, а целочисленная переменная `icon` получает значение `INFORMATION_MESSAGE` (статическая константа из класса `JOptionPane`). Если условие ложно, то переменной `txt` присваивается значение `"Вы не угадали! Это число "+num+"!"`, а переменной `icon` присваивается значение `ERROR_MESSAGE` (статическая константа из класса `JOptionPane`).

ПОДРОБНОСТИ

Класс `JOptionPane`, кроме статических методов, содержит еще и статические поля — константы, используемые для определения вида пиктограмм в диалоговых окнах: `INFORMATION_MESSAGE` (информационная пиктограмма), `QUESTION_MESSAGE` (пиктограмма со знаком вопроса), `WARNING_MESSAGE` (пиктограмма предупреждения), `ERROR_MESSAGE` (пиктограмма ошибки), `PLAIN_MESSAGE` (пиктограмма отсутствует). Поскольку константы целочисленные, то их можно присваивать значениями целочисленным переменным.

После определения значений переменных `txt` и `icon` они используются как аргументы при вызове метода `showMessageDialog()`. Методу передаются такие аргументы: пустая ссылка `null` на родительское окно (его нет), текст сообщения (определяется значением переменной `txt`), текст для заголовка окна, тип пиктограммы (определяется значением переменной `icon`).

НА ЗАМЕТКУ

В зависимости от истинности или ложности условия в условном операторе поразному определяются значения переменных, определяющих текст сообщения и тип пиктограммы. После этого переменные используются при отображении сообщения.

Пример использования упрощенной формы условного оператора приведен в листинге 2.2.

Листинг 2.2. Упрощенная форма условного оператора

```
// Статический импорт:
import static javax.swing.JOptionPane.*;
// Импорт класса:
import javax.swing.ImageIcon;
class Demo{
   public static void main(String[] args){
      int res;
      // Отображается окно подтверждения:
      res=showConfirmDialog(null, // Родительское окно
         "Хотите увидеть красную панду?", // Сообщение
         "Вопрос", // Заголовок окна
         YES_NO_OPTION // Кнопки
      );
```

```
        // Упрощенная форма условного оператора:
        if(res==YES_OPTION){
            // Полный путь к файлу с изображением:
            String file="d:/Pictures/Animals/panda.png";
            // Создание объекта изображения:
            ImageIcon img=new ImageIcon(file);
            // Отображается окно с картинкой:
            showMessageDialog(null, // Родительское окно
                img, // Изображение
                "Красная панда", // Заголовок окна
                PLAIN_MESSAGE // Пиктограмма отсутствует
            );
        }
    }
}
```

При запуске программы появляется *окно подтверждения*: сообщение в окне содержит вопрос, и у окна есть две кнопки (Yes и No), как показано на рис. 2.6.

Если пользователь нажимает кнопку No, то окно закрывается и на этом выполнение программы завершается. Но если пользователь нажимает кнопку Yes, то появляется еще одно окно (рис. 2.7).

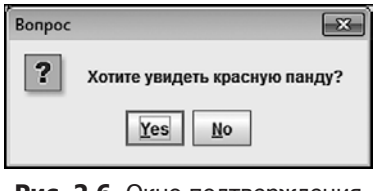


Рис. 2.6. Окно подтверждения с двумя кнопками

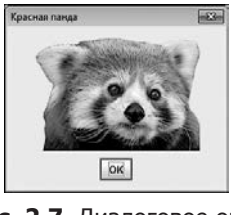


Рис. 2.7. Диалоговое окно с изображением красной панды

Окно содержит изображение красной панды.

Что касается самой программы, то она небольшая. Для отображения окна подтверждения мы используем статический метод `showConfirmDialog()` из класса `JOptionPane`. Аргументами методу при вызове передаются:

- Ссылка на родительское окно (указано значение `null`, поскольку такого окна нет).
- Текст сообщения `"Хотите увидеть красную панду?"`, которое отображается в диалоговом окне.
- Текст `"Вопрос"`, определяющий заголовок окна.
- Статическая константа `YES_NO_OPTION` из класса `JOptionPane`, определяющая количество кнопок в диалоговом окне.

ПОДРОБНОСТИ

Константа YES_NO_OPTION означает, что в окне отображаются кнопки Yes и No. Также можно использовать следующие константы: DEFAULT_OPTION (набор кнопок по умолчанию — отображается одна кнопка OK), YES_NO_CANCEL_OPTION (отображаются кнопки Yes, No и Cancel) или OK_CANCEL_OPTION (кнопки OK и Cancel).

Результат вызова метода записывается в целочисленную переменную res. Значение, возвращаемое методом showConfirmDialog(), зависит от того, как было закрыто окно. Если пользователь закрывает окно нажатием кнопки Yes, то результатом метод возвращает значение константы YES_OPTION.

ПОДРОБНОСТИ

Результат NO_OPTION возвращается, если пользователь нажал кнопку No. Результат CANCEL_OPTION возвращается, если была нажата кнопка Cancel. Результат OK_OPTION означает, что была нажата кнопка OK. А если окно закрыто нажатием системной пиктограммы, то результат определяется константой CLOSED_OPTION.

После того как первое окно закрыто, начинается выполнение условного оператора (в упрощенной форме — у этого оператора отсутствует else-блок). Проверяется условие res==YES_OPTION, истинность которого означает, что окно было закрыто нажатием кнопки Yes. Если так, то объявляется текстовая переменная file. Значением ей присваивается текст "d:/Pictures/Animals/panda.png". Это полный путь к файлу с изображением, которое мы собираемся показать в диалоговом окне. Но предварительно на основе изображения нужно создать объект. Объект изображения создается командой ImageIcon img=new ImageIcon(file).

ПОДРОБНОСТИ

Объект создается на основе класса, подобно тому как дом строится на основе проекта. Объект изображения создается на основе класса ImageIcon, который импортируется в программу (инструкция import javax.swing.ImageIcon). Для создания объекта используется инструкция new, после которой указывается название класса и, в круглых скобках, параметры, необходимые для создания объекта. В данном случае класс — ImageIcon, а параметр один, и это полный путь к файлу с изображением. Результатом выражения на основе инструкции new является ссылка на созданный объект. Эта ссылка записывается в объектную переменную (в нашем случае объектная переменная называется img). Объектная переменная объявляется как обычная переменная, но только типом переменной указывается класс, на основе которого создается объект.

Классы и объекты обсуждаются немного позже. Что касается самого изображения, то для корректной работы программы необходимо разместить файл panda.png в директорию D:\Pictures\Animals (или изменить в программе значение текстовой переменной file в соответствии с тем, где именно находится файл с картинкой). Сам графический файл сохранен в формате, позволяющем исполь-

зование прозрачной подложки (прозрачный фон), поэтому в диалоговом окне при отображении изображения создается впечатление, что оно наложено на фон окна. Вообще же использовано прямоугольное изображение шириной 200 пикселов и высотой 140 пикселов.

После создания объекта изображения отображается диалоговое окно с картинкой. Особенность соответствующей команды на основе метода `showMessageDialog()` в том, что вторым аргументом вместо текстового значения передается переменная `img` (объект изображения). То есть вместо текста мы указываем графический объект. Тип пиктограммы в окне определяется константой `PLAIN_MESSAGE` из класса `JOptionPane`, что на самом деле означает отсутствие пиктограммы у окна (две картинки в одном окне выглядят не очень эстетично).

Пример использования нескольких вложенных условных операторов представлен в листинге 2.3.

Листинг 2.3. Вложенные условные операторы

```
import java.util.Scanner;
class Demo{
   public static void main(String[] args){
      int a;
      // Создание объекта класса Scanner:
      Scanner input=new Scanner(System.in);
      System.out.print("Введите целое число: ");
      // Считывание целого числа:
      a=input.nextInt();
      if(a==0){ // Если введен ноль
         System.out.println("Вы ввели ноль!");
      }else if(a==1){ // Если введена единица
         System.out.println("Вы ввели единицу!");
      }else if(a%2==0){ // Если введено четное число
         System.out.println("Вы ввели четное число!");
      }else{ // В прочих случаях
         System.out.println("Вы ввели нечетное число!");
      }
      System.out.println("Спасибо!");
   }
}
```

В программе считывается целое число, введенное пользователем, и в зависимости от значения этого числа выводится то или иное сообщение. Для считывания числа мы используем консольный ввод.

НА ЗАМЕТКУ

Консольный ввод в Java реализован не самым простым образом. Причина в том, что язык Java разрабатывался не для создания консольных программ, хотя и такие программы успешно создаются.

Для считывания значений, которые пользователь вводит через окно вывода, нам нужен объект класса `Scanner`. Класс импортируется в программу инструкцией `import java.util.Scanner`. Объект класса `Scanner` создается командой `Scanner input=new Scanner(System.in)`.

ПОДРОБНОСТИ

Мы используем ту же схему, что и при создании объекта изображения, а именно: объект класса `Scanner` создается с помощью инструкции `new`, после которой указано название класса. Параметром, необходимым для создания объекта, является ссылка `System.in` на объект стандартного потока ввода (реализуется через статическое поле `in` класса `System`). Ссылка на созданный объект записывается в объектную переменную `input`.

Командой `System.out.print("Введите целое число: ")` в окно выводится сообщение.

НА ЗАМЕТКУ

В отличие от метода `println()`, методом `print()` выводится сообщение, но курсор в области вывода в новую строку не переводится — он остается в той же строке.

Командой `a=input.nextInt()` считывается целочисленное значение, введенное пользователем, которое записывается в переменную `a`.

ПОДРОБНОСТИ

Из объекта `input` вызывается метод `nextInt()`. Метод результатом возвращает целочисленное значение, которое ввел пользователь.

Происходит все так: появляется сообщение `Введите целое число:`, и курсор находится в области вывода. Пользователь вводит число, нажимает клавишу <Enter>, и программа считывает введенное пользователем значение.

Для считанного значения отслеживаются варианты действий: пользователь ввел ноль; пользователь ввел единицу; пользователь ввел четное число (делится без остатка на `2`); пользователь ввел нечетное число (не делится без остатка на `2`). Перебор всех возможных вариантов реализован через блок вложенных условных операторов. Перечисленные ранее условия проверяются по очереди, до первого выполненного условия. Если ни одно из условий не является истинным, то выполняются команды в последнем `else`-блоке вложенных условных операторов. Результат выполнения программы варьируется и может быть таким (здесь и далее жирным шрифтом выделено введенное пользователем значение):

Результат выполнения программы (из листинга 2.3)

```
Введите целое число: 0
Вы ввели ноль!
Спасибо!
```

Таким:

Результат выполнения программы (из листинга 2.3)

```
Введите целое число: 1
Вы ввели единицу!
Спасибо!
```

Таким:

Результат выполнения программы (из листинга 2.3)

```
Введите целое число: 8
Вы ввели четное число!
Спасибо!
```

Или таким:

Результат выполнения программы (из листинга 2.3)

```
Введите целое число: 9
Вы ввели нечетное число!
Спасибо!
```

Хотя с помощью блоков вложенных условных операторов можно реализовать практически любой алгоритм с точками ветвления, нередко более эффективным представляется использование оператора выбора `switch`.

Оператор выбора switch

— Утром деньги — вечером стулья. Вечером деньги — утром стулья.

— А можно так: утром стулья — вечером деньги?

— Можно. Но деньги вперед.

из к/ф «Двенадцать стульев»

Оператор выбора `switch` позволяет выполнять разные блоки команд в зависимости от значения, которое принимает некоторое выражение (в этом смысле он немного напоминает условный оператор). Синтаксис вызова оператора `switch` следующий:

```
switch(условие){
   case значение:
      // Команды
      break;
   case значение:
      // Команды
```

```
      break;
   // Другие case-блоки
   case значение:
      // Команды
      break;
   default:
      // Команды
}
```

После ключевого слова `switch` в круглых скобках указывается выражение, значение которого проверяется (тип выражения — целочисленный, символьный или текстовый). Тело оператора составляют `case`-блоки. Каждый такой блок начинается ключевым словом `case` и обычно (но не всегда) заканчивается инструкцией `break`. В каждом блоке после ключевого слова `case` указывается контрольное значение (значение, которое может принимать проверяемое выражение). Контрольное значение должно быть константой или литералом. После контрольного значения ставится двоеточие. Последним в операторе `switch` может быть блок, помеченный ключевым словом `default` (блок не является обязательным).

Согласно алгоритму выполнения оператора `switch`, сначала вычисляется выражение в `switch`-инструкции. Вычисленное значение последовательно сравнивается, до первого совпадения, с контрольными значениями, указанными после инструкций `case`. Если совпадение найдено, то начинают выполняться команды соответствующего блока. Выполняются они до первой инструкции `break`, а если таковая не встретится — то до конца оператора выбора. Если при сравнении значения выражения с контрольными значениями совпадения нет, то выполняются команды в блоке `default` (если он есть).

НА ЗАМЕТКУ

В конце `case`-блока выполнение команд автоматически не прекращается. Нужна инструкция `break`. Если ее нет, то после выполнения команд в одном `case`-блоке начнут выполняться команды в следующем блоке.

Схема выполнения оператора выбора показана на рис. 2.8.

Пример использования оператора `switch` приведен в листинге 2.4.

Листинг 2.4. Использование оператора выбора

```
import java.util.Scanner;
class Demo{
   public static void main(String[] args){
      Scanner input=new Scanner(System.in);
      String name,txt;
      System.out.print("Введите имя: ");
      name=input.nextLine();
      // Оператор выбора:
      switch(name){
```

```
            case "Матроскин":
               txt="Это Кот "+name+"!";
               break;
            case "Шарик":
               txt="Это Пес "+name+"!";
               break;
            case "Федор":
               txt="Это Дядя "+name+"!";
               break;
            default:
               txt="Это неизвестный персонаж!";
         }
         System.out.println(txt);
      }
}
```

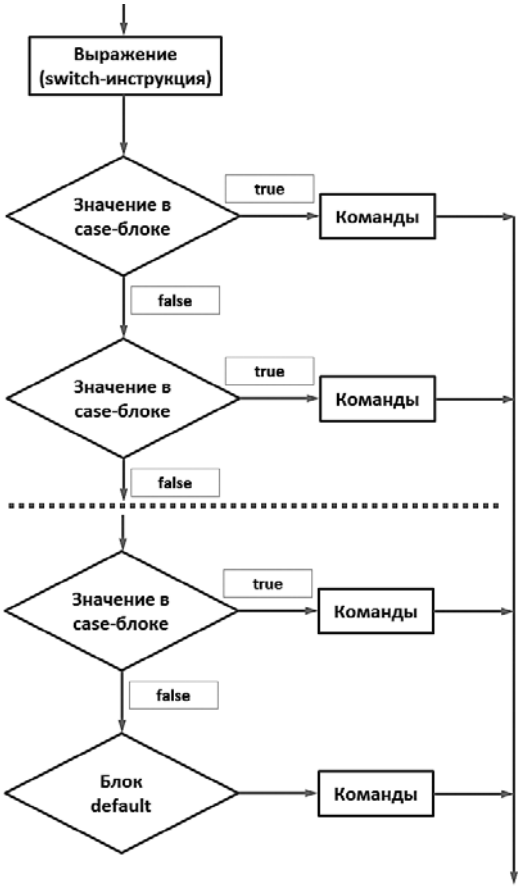


Рис. 2.8. Схема выполнения оператора выбора

При запуске программы пользователь должен ввести имя, а программа на это имя отреагирует (отобразит сообщение). Возможный результат выполнения программы представлен ниже (жирным шрифтом здесь и далее выделен текст, который вводит пользователь):

Результат выполнения программы (из листинга 2.4)

```
Введите имя: Матроскин
Это Кот Матроскин!
```

Еще один возможный результат выполнения программы:

Результат выполнения программы (из листинга 2.4)

```
Введите имя: Шарик
Это Пес Шарик!
```

Результат может быть таким:

Результат выполнения программы (из листинга 2.4)

```
Введите имя: Федор
Это Дядя Федор!
```

И еще результат может быть таким:

Результат выполнения программы (из листинга 2.4)

```
Введите имя: Печкин
Это неизвестный персонаж!
```

Как и в предыдущем примере, мы для реализации консольного ввода создаем объект `input` класса `Scanner`. Но на этот раз мы собираемся считывать не число, а текст, поэтому из объекта `input` вызывается метод `nextLine()`, который результатом возвращает текстовую строку, введенную пользователем. Считанное значение записывается в текстовую переменную `name` (команда `name=input.nextLine()`). Значение переменной `name` проверяется в `case`-блоках оператора выбора `switch`. В зависимости от значения переменной `name` формируется значение переменной `txt`. После выполнения оператора выбора значение переменной `txt` отображается в окне вывода.

НА ЗАМЕТКУ

Блок `default` будет задействован, только если значение переменной `name` не совпало ни с одним из контрольных значений, указанных в `case`-блоках.

На практике возможны ситуации, когда некоторые `case`-блоки остаются пустыми. Такой подход бывает полезен, если необходимо один и тот же набор команд выполнять для разных контрольных значений проверяемого выражения (листинг 2.5).

Листинг 2.5. Пустые блоки в операторе выбора

```
import java.util.Scanner;
class Demo{
   public static void main(String[] args){
      Scanner input=new Scanner(System.in);
      String name,txt;
      System.out.print("Введите имя: ");
      name=input.nextLine();
      switch(name){
         case "Матроскин":
         case "Барсик":
            txt="Это Кот "+name+"!";
            break;
         case "Шарик":
         case "Тузик":
         case "Бобик":
            txt="Это Пес "+name+"!";
            break;
         case "Федор":
         case "Вася":
            txt="Это Дядя "+name+"!";
            break;
         default:
            txt="Это неизвестный персонаж!";
      }
      System.out.println(txt);
   }
}
```

Данный пример является вариацией предыдущего. Принципиальное его отличие в том, что в операторе выбора использованы пустые `case`-блоки с дополнительными контрольными значениями для переменной `name`. В результате имена `Матроскин` и `Барсик` опознаются как имена котов, имена `Шарик`, `Тузик` и `Бобик` интерпретируются как принадлежащие собакам, а также программа узнает людей с именами `Федор` и `Вася`. Как это работает? Например, пользователь вводит имя `Тузик`:

Результат выполнения программы (из листинга 2.5)

```
Введите имя: Тузик
Это Пес Тузик!
```

При переборе `case`-блоков находится совпадение, начинают выполняться команды от места совпадения до первой инструкции `break` (или конца оператора выбора). Но поскольку `case`-блок с контрольным значением `"Тузик"` пуст и в нем нет `break`-инструкции, то выполняться будут команды вплоть до инструкции `break` в `case`-блоке с контрольным значением `"Бобик"`. Те же команды выполняются, если мы введем имя `Шарик` или `Бобик`. По аналогичной схеме работают и прочие блоки оператора выбора.

Оператор цикла for

Все кончается, рано или поздно.

из к/ф «Гараж»

Для выполнения многократно повторяющихся действий используют *операторы цикла*. В Java существует несколько операторов цикла. Мы начнем с оператора цикла `for`. Его синтаксис представлен ниже:

```
for(команды;условие;команды){
   // Команды
}
```

В круглых скобках после ключевого слова `for` указываются три блока команд. Блоки разделяются точкой с запятой. Инструкции в пределах одного блока разделяются запятыми. Первый блок обычно называется блоком *инициализации*. Как правило, там размещаются команды, которыми присваиваются начальные значения переменным. Второй блок — это блок *условия*. Во втором блоке указывается выражение логического типа (пока оно истинно, оператор продолжает работу). Третий блок обычно содержит команды, которыми изменяются значения переменных, используемых в операторе цикла. Третий блок обычно называют блоком *инкремента* или *декремента*. Основное тело с командами оператора цикла выделяется фигурными скобками (но если тело оператора цикла состоит из одной команды, фигурные скобки можно не использовать). Алгоритм выполнения оператора цикла `for` следующий.

- Все начинается с выполнения команд в первом блоке (блоке инициализации). Эти команды выполняются один и только один раз в самом начале работы оператора цикла.
- Затем проверяется условие во втором блоке. Если оно истинно (значение `true`), то выполняются команды в теле оператора цикла (команды в фигурных скобках). Далее выполняются команды в третьем блоке в круглых скобках.
- Проверяется условие во втором блоке. Если условие истинно, то выполняются команды в теле оператора цикла и в третьем блоке, и затем снова проверяется условие, и так далее.
- Если при проверке условия окажется, что оно ложно (значение `false`), то выполнение оператора цикла на этом завершается.

Схема выполнения оператора цикла `for` показана на рис. 2.9.

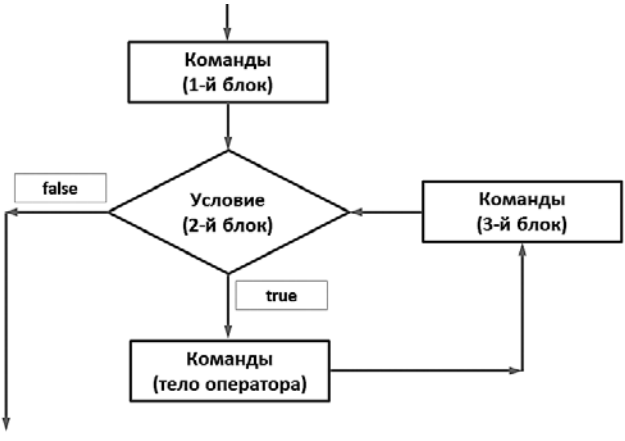


Рис. 2.9. Схема выполнения оператора цикла for

Пример использования оператора цикла представлен в листинге 2.6.

Листинг 2.6. Вычисление суммы чисел

```
class Demo{
   public static void main(String[] args){
      // Индексная переменная и верхняя
      // граница суммы:
      int i,n=100;
      // Переменная для записи значения суммы:
      int sum=0;
      // Оператор цикла:
      for(i=1;i<=n;i++){
         sum+=i;
      }
      System.out.println(
         "Сумма чисел от 1 до "+n+": "+sum
      );
   }
}
```

Программой вычисляется сумма натуральных чисел от `1` до `100` (значение верхней границы для суммы записано в переменную `n`). В программе имеется целочисленная переменная `sum`, которая инициализируется с начальным нулевым значением — в эту переменную записывается значение суммы. Вычисление суммы осуществляется посредством оператора цикла. В нем используется целочисленная индексная переменная `i`. Объявляется переменная до запуска оператора цикла. В первом блоке (блок инициализации) индексной переменной присваивается значение `1`. Проверяется условие `i<=n`, то есть оператор цикла выполняется до тех пор, пока значение индексной переменной не превысит значение переменой `n` (в данном случае это `100`). В теле оператора цикла значение переменной `sum` увеличивается на текущее

значение индексной переменной `i`. В третьем блоке оператора цикла командой `i++` значение индексной переменной увеличивается на единицу.

Последней командой программы отображается значение суммы. В результате выполнения программы мы получаем такое сообщение:

Результат выполнения программы (из листинга 2.6)

```
Сумма чисел от 1 до 100: 5050
```

Данную программу легко модифицировать, чтобы она вычисляла, например, сумму нечетных чисел в указанном диапазоне. Для этого в третьем блоке оператора цикла следует команду `i++` заменить на команду `i+=2`. Кроме того, индексную переменную можно объявить прямо в первом блоке оператора цикла.

ПОДРОБНОСТИ

Переменная доступна в том блоке, в котором она объявлена. Границы блока определяются парой фигурных скобок. Если мы объявляем переменную в операторе цикла, то такая переменная доступна только в операторе цикла.

Пример измененной программы для вычисления суммы нечетных чисел можно найти в листинге 2.7.

Листинг 2.7. Вычисление суммы нечетных чисел

```
class Demo{
   public static void main(String[] args){
      // Переменная для вычисления суммы
      // и верхняя граница для суммы:
      int sum=0,n=100;
      // Оператор цикла:
      for(int i=1;i<=n;i+=2){
         sum+=i;
      }
      System.out.println(
         "Сумма нечетных чисел от 1 до "+n+": "+sum
      );
   }
}
```

Результат выполнения программы такой (вычисляется сумма нечетных чисел, которые не превышают значение `100`):

Результат выполнения программы (из листинга 2.7)

```
Сумма нечетных чисел от 1 до 100: 2500
```

При работе с оператором цикла у нас достаточно много степеней свободы: блоки в операторе цикла могут содержать несколько команд, но могут быть пустыми.

НА ЗАМЕТКУ

Пустой второй блок условия эквивалентен истинному условию. Для преждевременного завершения оператора цикла используют инструкцию `break`. С помощью инструкции `continue` можно завершить текущую итерацию для оператора цикла.

В листинге 2.8 представлена программа для вычисления суммы нечетных чисел, в которой команды инициализации переменных и команда из основного тела оператора цикла включены соответственно в первый и третий блоки.

Листинг 2.8. Несколько команд в блоках

```
class Demo{
   public static void main(String[] args){
      int sum,i,n;
      // Оператор цикла:
      for(sum=0,i=1,n=100;i<=n;sum+=i,i+=2);
      System.out.println(
         "Сумма нечетных чисел от 1 до "+n+": "+sum
      );
   }
}
```

Индексная переменная `i`, а также переменные `sum` и `n` объявляются вне оператора цикла. Инициализируются все три переменные в первом блоке. Условие выполнения оператора цикла не изменилось. Третий блок состоит из двух команд: команды `sum+=i`, предназначенной для увеличения значения переменной `sum` на величину `i`, и команды `i+=2`, изменяющей значение индексной переменной. Тело оператора цикла не содержит команд, поэтому после закрывающей круглой скобки стоит точка с запятой. Результат выполнения программы — такой же, как и в предыдущем случае. Отметим несколько принципиальных моментов.

- Если переменные (`sum`, `n` и `i`) объявить в операторе цикла, доступными они будут только в пределах этого оператора. С индексной переменной `i` в этом случае проблем не возникает, а вот последняя команда программы, в которой имеется ссылка на переменные `sum` и `n`, окажется некорректной.
- Имеет значение порядок следования команд в третьем блоке оператора цикла. Если команды `sum+=i` и `i+=2` поменять местами, то сначала будет изменяться значение индексной переменной `i`, а затем на это новое значение будет увеличиваться значение переменной `sum`. В результате сумма будет вычисляться не от `1`, а от `3`.
- Хотя тело оператора цикла не содержит команд, точку с запятой все равно ставить нужно (или пустые фигурные скобки). Если этого не сделать, то телом оператора цикла станет следующая после `for`-инструкции команда — в данном случае речь о команде `System.out.println("Сумма нечетных чисел от 1 до`

"+n+": "+sum). При этом с формальной точки зрения синтаксис программы будет корректным.

Блоки оператора цикла могут быть пустыми. В листинге 2.9 приведен пример, в котором отсутствуют команды в первом и третьем блоках. Но программа работает корректно, сумма нечетных чисел вычисляется правильно.

Листинг 2.9. Пустые блоки в операторе цикла

```
class Demo{
   public static void main(String[] args){
      int sum=0,i=1,n=100;
      // Оператор цикла с пустыми блоками:
      for(;i<=n;){
         sum+=i;
         i+=2;
      }
      System.out.println(
         "Сумма нечетных чисел от 1 до "+n+": "+sum
      );
   }
}
```

Переменные i, sum и n инициализируются при объявлении до вызова оператора цикла. Поэтому в первом блоке инициализации ничего инициализировать не нужно, а сам блок остается пустым (хотя точка с запятой все равно ставится). Второй блок с условием остался неизменным. Третий блок — также пустой. Команда i+=2, которой изменяется значение индексной переменной, вынесена в тело оператора цикла.

Ситуацию можно усугубить, что называется, до предела, видоизменив программу так, чтобы все три блока оператора цикла стали пустыми. Пример представлен в листинге 2.10.

Листинг 2.10. В операторе цикла все блоки пустые

```
class Demo{
   public static void main(String[] args){
      int sum=0,i=1,n=100;
      // В операторе цикла все блоки пустые:
      for(;;){
         sum+=i;
         i+=2;
         // Завершение оператора цикла:
         if(i>n) break;
      }
      System.out.println(
         "Сумма нечетных чисел от 1 до "+n+": "+sum
      );
   }
}
```

Поскольку второй блок пустой, то формально мы имеем дело с бесконечным циклом. Необходимо предусмотреть возможность выхода из такого оператора. Для этого использована команда `if(i>n) break` с условным оператором (в упрощенной форме, без `else`-блока). При истинности условия `i>n` инструкцией `break` завершается работа оператора цикла.

Оператор цикла while

> — Ну, барин, ты задачи ставишь! За десять дён одному не справиться. Тут помощник нужен — хомо сапиенс.
>
> — Бери помощников, но чтобы не раньше!
>
> *из к/ф «Формула любви»*

Для организации циклов кроме оператора `for` часто используется оператор `while` (есть еще оператор `do-while`, который рассматривается в следующем разделе). Синтаксис вызова оператора `while` такой:

```
while(условие){
   // Команды
}
```

После ключевого слова `while` в круглых скобках указывается условие. Оно проверяется в начале выполнения оператора цикла `while`. Если условие истинно, то выполняются команды в теле оператора цикла — они заключаются в фигурные скобки. После этого снова проверяется условие. Если оно истинно, то вновь выполняются команды и проверяется условие, и так далее. Работа оператора цикла завершается, если при очередной проверке условия оно окажется ложным. Схема выполнения оператора `while` показана на рис. 2.10.

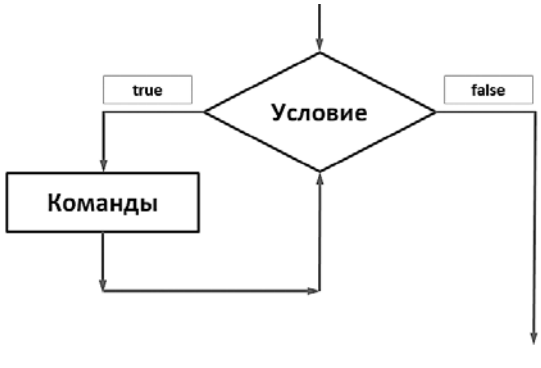


Рис. 2.10. Схема выполнения оператора цикла while

В листинге 2.11 приведен пример программы для вычисления суммы нечетных натуральных чисел с использованием оператора цикла `while`.

Листинг 2.11. Вычисление суммы с помощью оператора while

```
class Demo{
   public static void main(String[] args){
      int sum=0,i=1,n=100;
      // Оператор цикла:
      while(i<=n){
         sum+=i;
         i+=2;
      }
      System.out.println(
         "Сумма нечетных чисел от 1 до "+n+": "+sum
      );
   }
}
```

Результат выполнения программы — такой же, как в предыдущих примерах. Хочется верить, что пояснения здесь не нужны.

Оператор цикла do-while

— Что же делать?
— Ждать.
— Чего?
— Пока не похудеет.

из м/ф «Вини-Пух идет в гости»

Оператор цикла `do-while` напоминает оператор цикла `while`. Синтаксис его вызова такой:

```
do{
   // Команды
}while(условие);
```

Выполнение оператора начинается с блока команд, размещенных в фигурных скобках после ключевого слова `do`. Затем проверяется условие, указанное в круглых скобках после ключевого слова `while`. Если условие истинно, то снова выполняются команды и затем проверяется условие, и так далее. Схема использования оператора цикла `do-while` показана на рис. 2.11.

Получается, что хотя бы один раз команды в теле оператора цикла будут выполнены — в этом главное отличие оператора `do-while` от оператора `while`. В листинге 2.12

приведен пример использования оператора `do-while` для вычисления суммы нечетных натуральных чисел.

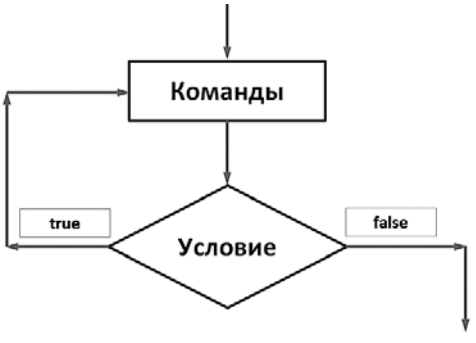


Рис. 2.11. Схема выполнения оператора цикла do-while

Листинг 2.12. Использование оператора do-while

```
class Demo{
   public static void main(String[] args){
      int sum=0,i=1,n=100;
      // Оператор цикла:
      do{
         sum+=i;
         i+=2;
      }while(i<=n);
      System.out.println(
         "Сумма нечетных чисел от 1 до "+n+": "+sum
      );
   }
}
```

Результат выполнения этой программы — точно такой же, как и в рассмотренных ранее примерах.

НА ЗАМЕТКУ

Операторы цикла `do-while`, `while` и `for` в общем-то эквивалентны: все, что можно запрограммировать с помощью одного оператора, можно сделать и воспользовавшись двумя другими. Вопрос лишь в синтаксисе. Но идеологические отличия все же есть. В операторе `while` сначала проверяется условие, и если при первой проверке оно окажется ложным, то команды в теле оператора не будут выполнены ни разу. В операторе `for` ситуация схожая: если при первой проверке условие окажется ложным, то команды в теле оператора и в третьем блоке не выполняются. Однако команды в первом блоке будут выполнены (они выполняются перед проверкой условия). В операторе цикла `do-while` сначала выполняются команды в теле оператора, и только после этого первый раз проверяется условие.

Использование управляющих инструкций

> Ален ноби, ностра алис! Что означает — ежели один человек построил, другой завсегда разобрать может.
>
> *из к/ф «Формула любви»*

Далее рассматриваются некоторые программы, в которых используются управляющие инструкции.

Вычисление экспоненты

В классе `Math` есть статический метод `exp()`, предназначенный для вычисления экспоненты: для числового аргумента x вычисляется значение e^x (или $\exp(x)$), где $e \approx 2{,}718281828$ — постоянная Эйлера. В действительности для вычисления экспоненты используется следующая сумма (разложение функции $\exp(x)$ в ряд Тейлора):

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots .$$

Разумеется, на практике вычислить бесконечную сумму невозможно, поэтому ограничиваются вычислением суммы для конечного количества слагаемых, и чем их больше, тем выше точность. В листинге 2.13 приведен пример программы, в которой вычисляется значение экспоненты.

Листинг 2.13. Вычисление экспоненты

```
class Demo{
   public static void main(String args[]){
      // Верхняя граница для суммы
      // и индексная переменная:
      int n=100,k;
      // Аргумент экспоненты, переменная для записи суммы
      // и итерационная добавка:
      double x=1,s=0,q=1;
      // Вычисление экспоненты:
      for(k=0;k<=n;k++){
         s+=q;
         q*=x/(k+1);
      }
      // Результат вычислений:
      System.out.println("exp("+x+")="+s);
   }
}
```

Поскольку значением для аргумента экспоненты указана единица, в результате выполнения программы получаем приближенное значение для постоянной Эйлера (причем с довольно неплохой точностью):

Результат выполнения программы (из листинга 2.13)

```
exp(1.0)=2.7182818284590455
```

Основу программы составляет оператор цикла. В нем индексная переменная `k` пробегает значения от `0` до `n` (значение этой переменной задано равным `100`). В теле цикла всего две команды. Командой `s+=q` значение переменной `s` (начальное значение равно нулю) увеличивается на величину `q` (единичное начальное значение), после чего командой `q*=x/(k+1)` изменяется значение переменной-добавки. Переменная `q` умножается на `x` и делится на `(k+1)`. Изменение переменной-добавки выполняется так, чтобы на следующем шаге эта добавка давала правильное приращение для ряда Тейлора. Действительно, в программе вычисляется сумма $\exp(x) \approx \sum_{k=0}^{n} \frac{x^k}{k!}$, поэтому приращение суммы для k-го индекса равняется $q_k = x^k/k!$. Для $(k + 1)$-го индекса добавка равняется $q_{k+1} = x^{k+1}/(k + 1)!$. В соответствии с соотношением $q_{k+1}/q_k = x/(k + 1)$ на основе добавки на k-м шаге для следующей итерации добавку необходимо умножить на x и разделить на $(k + 1)$.

Числа Фибоначчи

Числами Фибоначчи называется последовательность натуральных чисел, первые два из которых равны единице, а каждое следующее число в последовательности равняется сумме двух предыдущих. В листинге 2.14 приведен пример программы, в которой выводятся числа из последовательности Фибоначчи.

Листинг 2.14. Числа Фибоначчи

```
class Demo{
   public static void main(String args[]){
      // Количество чисел в последовательности,
      // начальные члены и индексная переменная:
      int n=15,a=1,b=1,i;
      System.out.println("Числа Фибоначчи:");
      // Вывод на экран двух первых
      // членов последовательности:
      System.out.print(a+" "+b);
      // Вычисление последовательности Фибоначчи:
      for(i=3;i<=n;i++){
         b=a+b;
         a=b-a;
         System.out.print(" "+b);
      }
   }
}
```

В результате выполнения программы получаем следующее:

Результат выполнения программы (из листинга 2.14)

```
Числа Фибоначчи:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

В программе объявляются целочисленные переменные `n` (количество вычисляемых чисел в последовательности Фибоначчи), а также переменные `a` и `b`, которые предназначены для записи предпоследнего и последнего вычисленных на данный момент чисел в последовательности. Этим переменным присвоены начальные единичные значения, поэтому они сразу отображаются в окне вывода. Далее в операторе цикла выполняется вычисление с отображением в окне вывода последующих членов. В частности, для вычисления следующего числа в последовательности, если известны последнее (переменная `b`) и предпоследнее (переменная `a`) значения, используется команда `b=a+b` — каждое новое число равняется сумме двух предыдущих. После этого необходимо в переменную `a` записать значение, которое до этого было записано в переменную `b`. Поскольку значение `b` уже изменилось и содержит сумму старого значения переменной `b` и текущего значения переменной `a`, от текущего значения переменной `b` необходимо отнять текущее значение переменной `a` и записать результат в переменную `a`. Поэтому после первой упомянутой команды в операторе цикла выполняется команда `a=b-a`. Новое вычисленное значение `b` отображается в окне вывода командой `System.out.print(" "+b)`.

Вычисление числа π

Воспользуемся модифицированным методом Монте-Карло для вычисления значения числа π. В частности, проведем следующий мысленный эксперимент. Впишем круг в квадрат с единичной стороной. Площадь такого квадрата равна, очевидно, единице. Радиус круга равен 1/2, а площадь — $\pi/4$. Эксперимент состоит в том, что внутри квадрата случайным образом выбираются точки. Точек много, и они равномерно распределены по квадрату. Некоторые из них попадают внутрь круга, другие — нет. Вероятность попадания точки внутрь круга равна отношению площади круга к площади квадрата, то есть $\pi/4$. В то же время если точек достаточно много, то отношение числа попавших внутрь круга точек к общему числу точек внутри квадрата должно быть близко к вероятности попадания точки внутрь круга. Чем больше выбрано точек, тем точнее совпадение. Поэтому для расчета числа $\pi \approx 3{,}14159265$ случайным (или не очень случайным) образом выбираем внутри квадрата какое-то количество точек (чем больше — тем лучше), подсчитываем, сколько из них попадает внутрь круга, находим отношение количества точек внутри круга к общему количеству точек, умножаем полученное значение на 4 и вычисляем, таким образом, значение для числа π.

Для решения этой задачи нужен хороший генератор случайных чисел — такой, чтобы генерировал случайное число с постоянной плотностью распределения на

интервале значений от `0` до `1`. Этого не так просто добиться, как может показаться на первый взгляд. Поэтому вместо генерирования случайных чисел покроем область квадрата сеткой, узлы которой будут играть роль случайных точек. Чем меньше размер ячейки сетки, тем выше точность вычислений. В листинге 2.15 приведен код, в котором решается эта задача.

Листинг 2.15. Вычисление числа π

```
class Demo{
   public static void main(String args[]){
      // Количество базовых линий сетки:
      int n=100000;
      // Индексные переменные:
      int i,j;
      // Счетчик для попавших в круг точек:
      long count=0;
      // Координаты точек и число "пи":
      double x,y,Pi;
      // Перебор точек:
      for(i=0;i<=n;i++){
         for(j=0;j<=n;j++){
            // Координаты точки:
            x=(double)i/n;
            y=(double)j/n;
            // Если точка попадает внутрь круга:
            if((x-0.5)*(x-0.5)+(y-0.5)*(y-0.5)<=0.25){
               count++;
            }
         }
      }
      // Число "пи":
      Pi=4.0*count/(n+1)/(n+1);
      // Результат вычислений:
      System.out.println(
         "Вычисление значения по "+(long)(n+1)*(n+1)+
         " точкам:"
      );
      System.out.println(Pi);
   }
}
```

Хотя в примере используется достаточно большое количество точек, результат, откровенно говоря, оставляет желать лучшего:

Результат выполнения программы (из листинга 2.15)

```
Вычисление значения по 10000200001 точкам:
3.141529585494137
```

В программе инициализируется целочисленная переменная `n`, которая определяет количество базовых линий сетки по каждой из координат. Общее количество точек в этом случае внутри квадрата равно `(n+1)*(n+1)`. Это число может быть достаточно

большим. Сравнимое с ним число — количество точек, которые попадают внутрь круга. Поэтому переменная `count`, которая предназначена для подсчета количества попавших внутрь круга точек, объявляется как принадлежащая типу `long`. Кроме целочисленных индексных переменных `i` и `j`, в программе объявляются переменные `x` и `y` для вычисления координат точек, а переменная `Pi` предназначена для записи вычисляемого значения числа π.

Внутри вложенных операторов цикла командами `x=(double)i/n` и `y=(double)j/n` вычисляются координаты точки, находящейся в узле на пересечении `i`-й и `j`-й линий. При делении оба операнда — целочисленные, поэтому для вычисления результата в формате с плавающей точкой используется инструкция `(double)`. Поскольку центр вписанного в квадрат круга имеет координаты (0,5; 0,5), а радиус круга равен 0,5, то критерий того, что точка с координатами (x, y) попадает внутрь круга (или на его границу), имеет вид $(x - 0{,}5)^2 + (y - 0{,}5)^2 \le 0{,}5^2$. Именно это условие и проверяется в условном операторе; если условие истинно, то значение переменной `count` увеличивается на единицу.

Число π вычисляется командой `Pi=4.0*count/(n+1)/(n+1)`. Чтобы избежать целочисленного деления, мы использовали литерал `4.0` типа `double`. Вычисленное значение `Pi` выводится на экран. При выводе на экран значения `(n+1)*(n+1)`, определяющего общее количество точек, для приведения к соответствующему формату использована команда `(long)`. Как уже отмечалось, даже если значение переменной `n` не выходит за допустимые границы диапазона для типа `int`, это может произойти при вычислении значения `(n+1)*(n+1)`.

Следует отметить, что предложенный способ вычисления числа π не является оптимальным. Дело в том, что для достижения мало-мальски приемлемой точности приходится использовать достаточно большое количество точек, что существенно сказывается на времени выполнения программы.

Альтернативный метод вычисления числа π, который мы здесь рассмотрим, базируется на применении ряда Фурье. В частности, можно воспользоваться тем, что на интервале от 0 до 2π имеет место разложение в ряд Фурье:

$$x = 4\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n}\sin\left(\frac{nx}{2}\right).$$

Если в этом разложении $x = \pi$, то получим $\pi = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$. В коде, представленном в листинге 2.16, для получения значения числа π вычисляется соответствующая сумма.

Листинг 2.16. Вычисление числа π на основе ряда Фурье

```
class Demo{
   public static void main(String args[]){
      // Количество слагаемых и индексная переменная:
      int n=5000000,k;
```

```
      // Начальное значение и добавка:
      double Pi=0,q=4;
      // Вычисление числа "пи":
      for(k=0;k<=n;k++){
         Pi+=q/(2*k+1);
         q*=(-1);
      }
      // Результат вычислений:
      System.out.println(
         "Вычислено по "+n+" слагаемым:"
      );
      System.out.println(Pi);
   }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 2.16)

```
Вычислено по 5000000 слагаемым:
3.1415928535897395
```

Точность по сравнению с предыдущим способом вычисления выше, хотя количество слагаемых в сумме, которые при этом пришлось учесть, достаточно велико.

Следующий способ получения значения для числа π основан на вычислении произведения. В частности, используем такое соотношение:

$$\pi = 2 \cdot \frac{2}{\sqrt{2}} \cdot \frac{2}{\sqrt{2+\sqrt{2}}} \cdot \frac{2}{\sqrt{2+\sqrt{2+\sqrt{2}}}} \cdot \ldots$$

В листинге 2.17 представлен код, в котором данное произведение используется для вычисления значения числа π.

Листинг 2.17. Вычисление числа π на основе произведения

```
class Demo{
   public static void main(String args[]){
      // Количество множителей и индексная переменная:
      int n=20,k;
      // Начальное значение и итерационный множитель:
      double Pi=2,q=Math.sqrt(2);
      // Вычисление числа "пи":
      for(k=1;k<=n;k++){
         Pi*=2/q;
         q=Math.sqrt(2+q);
      }
      // Результат вычислений:
      System.out.println(
         "Вычислено по "+n+" множителям:"
      );
```

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

Получаем такое значения для числа π:

Результат выполнения программы (из листинга 2.17)

```
Вычислено по 20 множителям:
3.1415926535886207
```

Отметим, что достаточно неплохой по точности результат получен на основе относительно малого количества множителей. Что касается непосредственно алгоритма вычисления значения числа π, то его основу составляет оператор цикла, в котором вычисляется произведение, используемое как оценка для числа π. Результат записывается в переменную `Pi`, начальное значение которой равно `2`. При вычислении произведения учтено то свойство, что каждый новый множитель представляет собой дробь. В числителе дроби — двойка, а знаменатель дроби может быть получен на основе знаменателя предыдущего множителя, если к этому знаменателю добавить `2` и извлечь из результата квадратный корень. Для записи значения знаменателя на каждом из итерационных шагов используется переменная `q` с начальным значением $\sqrt{2}$. В теле оператора цикла всего две команды. Командой `Pi*=2/q` на основе данного значения множителя изменяется значение переменной — результата `Pi`, а затем командой `q=Math.sqrt(2+q)` изменяется знаменатель для следующего множителя.

НА ЗАМЕТКУ

Поскольку статический импорт класса `Math` в программе не выполнялся, то при вызове метода `sqrt()` необходимо указывать имя класса `Math`.

Метод последовательных итераций

В следующей программе методом последовательных итераций решается алгебраическое уравнение. Для уравнения вида $x = f(x)$, решаемого относительно переменной x, применение метода последовательных итераций подразумевает выполнение следующих действий. Для переменной x задается начальное приближение x_0, то есть $x = x_0$. Каждое следующее приближение вычисляется на основе предыдущего. Если на n-м шаге для корня уравнения вычислено приближение x_n, то приближение x_{n+1} на следующем шаге вычисляется как $x_{n+1} = f(x_n)$.

Для того чтобы соответствующая итерационная процедура сходилась к корню уравнения, необходимо, чтобы в области поиска корня выполнялось условие $|df(x)/dx| < 1$. В листинге 2.18 представлен код, с помощью которого методом последовательных итераций решается уравнение $x = (x^2 + 10)/7$ с заданным начальным приближением. Корнями этого квадратного уравнения являются значения $x = 2$ и $x = 5$. В данном случае уравнение представлено в виде $x = f(x)$, где функция

$f(x) = (x^2 + 10)/7$. Поскольку $df(x)/dx = 2x/7$, для такого представления уравнения методом последовательных итераций можно искать корень, попадающий в интервал значений $-2{,}5 < x < 2{,}5$, то есть корень $x = 2$.

Листинг 2.18. Метод последовательных итераций

```
class Demo{
   public static void main(String args[]){
      // Начальное приближение:
      double x0=0;
      // Переменные для корня и функции:
      double x,f;
      // Погрешность:
      double epsilon=1E-10;
      // Ограничение на количество итераций:
      int Nmax=1000;
      // Итерационная переменная:
      int n=0;
      // Начальное значение для функции:
      f=x0;
      // Вычисление корня:
      do{
         // Изменение индексной переменной:
         n++;
         // Новое приближение для корня:
         x=f;
         // Новое значение для функции (корня):
         f=(x*x+10)/7;
      }while((n<=Nmax)&&(Math.abs(x-f)>epsilon));
      // Последняя "итерация":
      x=f;
      // Результат вычислений:
      System.out.println("Решение уравнения:");
      System.out.println("x="+x);
      System.out.println("Количество итераций: "+(n+1));
   }
}
```

Используемая в программе итерационная процедура выполняется до тех пор, пока не будет достигнута необходимая точность вычислений либо общее количество итераций не превысит установленную верхнюю границу. Верхняя граница для значения итерационной переменой определяется значением переменной `Nmax`. Погрешность корня задается значением переменной `epsilon`. В принципе, можно применять и более точные оценки погрешности вычисляемого корня. Здесь в качестве оценки для погрешности учитывается разница между значениями корней на разных итерациях $|x_{n+1} - x_n|$. Другими словами, это не точность корня, а значение приращения для корня (по абсолютной величине). Чтобы контролировать эту величину в программе, необходимо иметь две переменные: текущее значение корня (переменная `x`) и следующее значение корня (переменная `f`). В теле оператора цикла каждое новое значение переменной `x` вычисляется командой `x=f`, а следующее значение для корня, записываемое

в переменную `f`, — командой `f=(x*x+10)/7`. В операторе цикла проверяется условие `(n<=Nmax)&&(Math.abs(x-f)>epsilon)`. Значение этого выражения равно `true`, если индексная переменная `n` не превышает значение `Nmax` и если разность `x-f` по абсолютной величине превышает значение переменной `epsilon`.

НА ЗАМЕТКУ

Для вычисления модуля использован статический метод `abs()` из класса `Math`.

Для приведенных в программном коде значений (для начального приближения корня и точности вычислений) получаем следующий результат:

Результат выполнения программы (из листинга 2.18)

```
Решение уравнения:
x=1.9999999999205635
Количество итераций: 42
```

В данном случае работа программы завершена из-за того, что приращение корня стало меньше значения переменной `epsilon`.

Решение квадратного уравнения

Рассмотрим программу, в которой решается квадратное уравнение, то есть уравнение вида $ax^2 + bx + c = 0$. В известном смысле задача банальная — корнями уравнения являются значения $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ при условии, что соответствующие значения вычислимы. Рассмотрим наиболее общую ситуацию, когда параметры a, b и c могут принимать любые действительные значения. Можно выделить следующие особые ситуации, в которых формальное применение приведенных решений невозможно.

- Параметр $a = 0$. В этом случае уравнение не является квадратным — отсутствует слагаемое с x^2. Значит, мы имеем дело с линейным уравнением вида $bx + c = 0$. Несмотря на кажущуюся простоту, это уравнение также имеет свои особенности.
- Если параметр b отличен от нуля (при условии, что $a = 0$), то уравнение имеет решение $x = -c/b$. Если же $b = 0$, то возможны два варианта: отсутствие решения при $c \neq 0$ или решение — любое число, если $c = 0$.

В случае, если параметр $a \neq 0$, выделим три ситуации, определяемые знаком дискриминанта $D = b^2 - 4ac$. При $D < 0$ квадратное уравнение на множестве действительных чисел решений не имеет. Если $D = 0$, квадратное уравнение имеет единственный корень $x = -\frac{b}{2a}$. Наконец, при $D > 0$ уравнение имеет два корня — это $x = -\frac{b \pm \sqrt{D}}{2a}$.

Все перечисленные варианты обрабатываются в программе, представленной в листинге 2.19.

Листинг 2.19. Решение квадратного уравнения

```
import java.util.Scanner;
class Demo{
   public static void main(String args[]){
      Scanner input=new Scanner(System.in);
      // Параметры уравнения:
      double a,b,c;
      // Корни и дискриминант:
      double x1,x2,D;
      System.out.println("Параметры уравнения:");
      // Определение параметров уравнения:
      System.out.print("a=");
      a=input.nextDouble();
      System.out.print("b=");
      b=input.nextDouble();
      System.out.print("c=");
      c=input.nextDouble();
      // Поиск решения:
      if(a==0){ // Если a равно 0
         System.out.println("Линейное уравнение!");
         if(b!=0){ // Если a равно 0 и b не равно 0
            System.out.println("Решение x="+(-c/b)+".");
         }else{
            if(c==0){ // Если a, b, и c равны нулю
               System.out.println(
                  "Решение – любое число."
               );
            }else{ // Если a и b равны нулю, а c – нет
               System.out.println("Решений нет!");
            }
         }
      }else{ // Если a не равно 0
         System.out.println("Квадратное уравнение!");
         // Дискриминант (значение):
         D=b*b-4*a*c;
         if(D<0){ // Отрицательный дискриминант
            System.out.println(
               "Действительных решений нет!"
            );
         }else{ // Нулевой дискриминант
            if(D==0){
               System.out.println("Решение x="+(-b/2/a));
            }else{ // Положительный дискриминант
               x1=(-b-Math.sqrt(D))/2/a;
               x2=(-b+Math.sqrt(D))/2/a;
               System.out.println(
                  "Два решения: x="+x1+" и x="+x2+"."
               );
```

```
            }
         }
      }
      // Завершение работы программы:
      System.out.println("Работа программы завершена.");
   }
}
```

Значения параметров уравнения вводятся с клавиатуры. Для этого создается объект `input` класса `Scanner`. Поскольку считываются значения типа `double`, то из объекта `input` на этот раз вызывается метод `nextDouble()`. Также в программе использованы вложенные условные операторы, в которых последовательно проверяются условия наличия у уравнения решений. Результат вычислений зависит от значений, которые вводит пользователь. Например, результат может быть таким (здесь и далее жирным шрифтом выделены значения, введенные пользователем):

Результат выполнения программы (из листинга 2.19)

```
Параметры уравнения:
a=2
b=-3
c=1
Квадратное уравнение!
Два решения: x=0.5 и x=1.0.
Работа программы завершена.
```

А вот случай, когда квадратное уравнение имеет одно решение:

Результат выполнения программы (из листинга 2.19)

```
Параметры уравнения:
a=1
b=-2
c=1
Квадратное уравнение!
Решение x=1.0
Работа программы завершена.
```

Если же квадратное уравнение не имеет решений на множестве действительных чисел, то результат будет таким:

Результат выполнения программы (из листинга 2.19)

```
Параметры уравнения:
a=2
b=-3
c=6
Квадратное уравнение!
Действительных решений нет!
Работа программы завершена.
```

Если значение переменной `a` равно нулю, то получаем линейное уравнение:

Результат выполнения программы (из листинга 2.19)

```
Параметры уравнения:
a=0
b=-3
c=6
Линейное уравнение!
Решение x=2.0.
Работа программы завершена.
```

Линейное уравнение может не иметь решений:

Результат выполнения программы (из листинга 2.19)

```
Параметры уравнения:
a=0
b=0
c=1
Линейное уравнение!
Решений нет!
Работа программы завершена.
```

Решением может быть и любое число:

Результат выполнения программы (из листинга 2.19)

```
Параметры уравнения:
a=0
b=0
c=0
Линейное уравнение!
Решение — любое число.
Работа программы завершена.
```

Последний случай достаточно экзотический и реализуется, когда все параметры уравнения равны нулю, — если так, то уравнение превращается в тождество.

Полет в атмосфере

Рассмотрим еще одну задачу, в которой вычисляется траектория движения тела, брошенного с известной начальной скоростью под углом к горизонту при условии, что на тело, кроме силы тяжести, действует еще и сила сопротивления воздуха. Предположим, что в атмосфере, в зависимости от высоты над поверхностью, сила сопротивления воздуха различна: в первом нижнем слое она пропорциональна квадрату скорости тела (и направлена против вектора скорости), во втором, центральном слое, сила сопротивления воздуха пропорциональна скорости тела, а в третьем, верхнем слое атмосферы, сила сопротивления воздуха отсутствует.

В программе задаются начальная скорость тела V, угол к горизонту α, под которым тело брошено, масса тела m, высота H_1 (на этой высоте заканчивается первый слой),

высота H_2 (на данной высоте заканчивается второй слой), ускорение свободного падения g, коэффициенты пропорциональности γ_1 и γ_2 для силы сопротивления воздуха в первой и второй зонах соответственно. По этим параметрам рассчитываются максимальная высота подъема тела H_{max}, дальность полета тела S_{max} и время полета тела T_{max}. Для вычислений используется дискретная модель — самый простой ее вариант.

Исходными являются дифференциальные уравнения второго порядка (уравнения Ньютона), описывающие движение тела по каждой из координатных осей (вдоль горизонтали и вертикали). Не вдаваясь в физические детали, отметим, что это уравнения вида $m\ddot{x} = -F_x(\dot{x},\ \dot{y})$ и $m\ddot{y} = -mg - F_y(\dot{x},\ \dot{y})$. Точка означает производную по времени, x и y — координаты тела как функции времени, $F_x(\dot{x}, \dot{y})$ и $F_y(\dot{x}, \dot{y})$ — модули проекции силы сопротивления воздуха, которая, в силу условия, зависит только от скорости тела и неявно — от высоты тела, то есть от координаты y.

В рамках модели предполагаем, что время изменяется дискретно с интервалом Δt. В этом случае моменты времени можно нумеровать целыми числами. Для n-го момента времени $t_n = n\Delta t$ обозначим координаты тела как x_n и y_n, а проекции скорости на координатные оси — соответственно как V_n и U_n. Задача состоит в том, чтобы по известным значениям для координат и скорости на n-м шаге определить эти параметры на $(n + 1)$-м шаге. Для этого можно воспользоваться соотношениями:

$$x_{n+1} = x_n + V_n\Delta t,$$

$$y_{n+1} = y_n + U_n\Delta t,$$

$$V_{n+1} = V_n - \frac{F_x(V_n, U_n)}{m}\Delta t,$$

$$U_{n+1} = U_n - g\Delta t - \frac{F_y(V_n, U_n)}{m}\Delta t.$$

В начальный момент, то есть при $n = 0$, имеем $x_n = 0$, $y_n = 0$, $V_0 = V\cos(\alpha)$ и $U_0 = V\sin(\alpha)$, где V — модуль вектора начальной скорости, а α — угол к горизонту, под которым брошено тело.

Что касается проекций силы сопротивления воздуха, то для первой воздушной зоны (первый слой, который определяется условием $y < H_1$) проекции силы сопротивления воздуха определяются соотношениями:

$$F_x = \gamma_1 V_n\sqrt{V_n^2 + U_n^2},$$

$$F_y = \gamma_1 U_n\sqrt{V_n^2 + U_n^2}.$$

Для второй зоны (второй слой определяется условием $H_1 \le y < H_2$) проекции силы сопротивления воздуха определяются соотношениями $F_x = \gamma_2 V_n$ и $F_y = \gamma_2 U_n$. Наконец,

для третьей зоны (третий слой определяется условием $H_2 \le y$) $F_x = 0$ и $F_y = 0$. Этой информации вполне достаточно для составления программы (листинг 2.20).

Листинг 2.20. Полет тела в атмосфере

```
import static java.lang.Math.*;
class Demo{
   public static void main(String args[]){
      // Ускорение свободного падения (м/с^2):
      final double g=9.8;
      // Масса (кг):
      double m=0.1;
      // Начальная скорость (м/с):
      double V=100;
      // Угол в градусах:
      double alpha=60;
      // Уровни воздушных зон (м):
      double H1=100,H2=300;
      // Коэффициенты для силы сопротивления
      // воздуха (Нс/м) и (Нс^2/м^2):
      double gamma1=0.0001,gamma2=0.0001;
      // Интервал времени (с):
      double dt=1E-6;
      // Координаты (м) и скорость (м/с)
      double Xn=0,Yn=0,Vn,Un;
      // Проекция силы сопротивления (Н):
      double Fx,Fy;
      // Время полета (с), дальность (м) и высота (м):
      double Tmax,Smax,Hmax=0;
      // Индикатор высоты (номер зоны):
      int height;
      // Перевод угла в радианы:
      alpha=toRadians(alpha);
      // Проекции начальной скорости:
      Vn=V*cos(alpha);
      Un=V*sin(alpha);
      for(int n=1;true;n++){
         // Координата по вертикали:
         Yn+=Un*dt;
         // Критерий завершения вычислений
         // и расчетные параметры:
         if(Yn<0){
            Tmax=round((n-1)*dt*100)/100.0;
            Smax=round(Xn*100)/100.0;
            Hmax=round(Hmax*100)/100.0;
            break;
         }
         // Координата по горизонтали:
         Xn+=Vn*dt;
         // Максимальная высота:
         if(Yn>Hmax) Hmax=Yn;
         // Вычисление номера зоны:
```

```
            height=Yn<H1?1:Yn<H2?2:3;
            // Сила сопротивления:
            switch(height){
               // Первая зона:
               case 1:
                  Fx=gamma1*Vn*sqrt(Vn*Vn+Un*Un);
                  Fy=gamma1*Un*sqrt(Vn*Vn+Un*Un);
                  break;
               // Вторая зона:
               case 2:
                  Fx=gamma2*Vn;
                  Fy=gamma2*Un;
                  break;
               // Третья зона:
               default:
                  Fx=0;
                  Fy=0;
            }
            // Проекция скорости по горизонтали:
            Vn+=-Fx*dt/m;
            // Проекция скорости по вертикали:
            Un+=-g*dt-Fy*dt/m;
         }
         // Результаты вычислений:
         System.out.println(
            "Время полета тела Tmax="+Tmax+" секунд."
         );
         System.out.println(
            "Дальность полета тела Smax="+Smax+" метров."
         );
         System.out.println(
            "Максимальная высота подъема тела Hmax="+
            Hmax+" метров."
         );
      }
}
```

В результате выполнения программы получаем следующее:

Результат выполнения программы (из листинга 2.20)

```
Время полета тела Tmax=15.97 секунды.
Дальность полета тела Smax=705.95 метра.
Максимальная высота подъема тела Hmax=312.31 метра.
```

Назначение переменных, объявленных и использованных в программе, описано в табл. 2.1.

Общая идея, положенная в основу алгоритма вычисления параметров траектории тела, достаточно проста. На начальном этапе координатам и проекциям скорости на координатные оси, исходя из начальных условий, присваиваются значения. Затем запускается оператор цикла, в рамках которого последовательно в соответствии

Таблица 2.1. Назначение переменных программы

Переменная	Назначение
`g`	Константа, содержащая значение для ускорения свободного падения
`m`	Масса тела
`V`	Начальная скорость тела (модуль)
`alpha`	Угол к горизонту в градусах, под которым брошено тело
`H1`	Высота, на которой заканчивается первая воздушная зона. Ниже этой высоты сила сопротивления пропорциональна квадрату скорости
`H2`	Высота, на которой заканчивается вторая воздушная зона. Ниже этой высоты (но выше первой) сила сопротивления воздуха пропорциональна скорости тела. Выше этого уровня сила сопротивления воздуха отсутствует
`gamma1`	Коэффициент пропорциональности в выражении для силы сопротивления воздуха в первой воздушной зоне
`gamma2`	Коэффициент пропорциональности в выражении для силы сопротивления воздуха во второй воздушной зоне
`dt`	Интервал дискретности по времени. Чем меньше значение этой переменной, тем точнее дискретная модель. С другой стороны, это же приводит к увеличению времени расчетов
`Xn`	Координата тела вдоль горизонтали. Она же определяет расстояние, которое пролетело тело на данный момент времени. В начальный момент времени координата равна нулю
`Yn`	Координата тела по вертикали. Она же определяет высоту, на которой находится тело в данный момент времени. В начальный момент значение равно нулю. Критерием прекращения вычислений является отрицательность значения этой координаты (вычисления прекращаются, когда координата становится меньше нуля)
`Vn`	Переменная, в которую записывается проекция скорости тела на горизонтальную ось в данный момент времени
`Un`	Переменная, в которую записывается проекция скорости тела на вертикальную ось в данный момент времени
`Fx`	Переменная, в которую записывается проекция силы сопротивления воздуха на горизонтальную ось в данный момент времени
`Fy`	Переменная, в которую записывается проекция силы сопротивления воздуха на вертикальную ось в данный момент времени
`height`	Целочисленная переменная, в которую записывается номер воздушной зоны, в которой в данный момент находится тело

Переменная	Назначение
`Tmax`	Переменная, в которую записывается значение времени полета тела
`Hmax`	Переменная, в которую записывается значение максимальной высоты подъема тела
`Smax`	Переменная, в которую записывается дальность полета тела

с приведенными соотношениями изменяются значения для координат и проекций скорости тела. Оператор цикла выполняется до тех пор, пока вертикальная координата не станет отрицательной. При этом каждый раз при вычислении вертикальной координаты `Yn` она сравнивается с текущим значением переменной `Hmax`, в которую записано значение максимальной высоты подъема. Если вычисленная координата `Yn` больше текущего значения максимальной высоты подъема, то вычисленная координата заменяет значение высоты подъема. Поскольку для вычисления новых значений координат и проекций скорости необходимо знать силу сопротивления воздуха, которая зависит от того, в какой зоне находится объект и с какой скоростью движется, на каждой итерации выполняется вычисление проекций силы сопротивления воздуха на координатные оси. По завершении вычислений результат отображается в окне вывода.

Основу программы составляет оператор цикла. В данном случае он формально бесконечный, поскольку в качестве условия указано значение `true`. Первой командой в операторе цикла на основе текущего значения проекции скорости по вертикальной оси вычисляется новая вертикальная координата. После этого проверяется условие ее отрицательности. Это критерий завершения работы оператора цикла. Если условие истинно, то вычисляются характеристики траектории и командой `break` завершается работа оператора цикла. В частности, для времени полета `Tmax` используется предыдущий момент, когда вертикальная координата еще была неотрицательной. В качестве дальности полета учитывается текущее значение горизонтальной координаты `Xn` (это значение еще осталось старым в отличие от измененного значения `Yn`). Также применяется текущее значение переменной `Hmax`. Все три переменные округляются до сотых значений: значение умножается на `100`, округляется с помощью метода `round()`, а затем снова делится на `100.0` (литерал типа `double`, чтобы не выполнялось целочисленное деление).

Если координата `Yn` неотрицательна, то работа оператора цикла продолжается. В частности, вычисляется новое значение горизонтальной координаты и с помощью условного оператора проверяется необходимость изменения значения переменной `Hmax`. Командой `height=Yn<H1?1:Yn<H2?2:3` вычисляется номер воздушной зоны, в которой находится тело. Далее по номеру зоны с помощью оператора выбора `switch` определяются проекции силы сопротивления воздуха на координатные оси. Разумеется, всю эту процедуру можно реализовать и с помощью вложенных условных операторов без непосредственного вычисления номера воздушной зоны, но примененный метод нагляднее.

После вычисления компонентов вектора для силы сопротивления воздуха вычисляются новые значения для проекций скорости на каждую из координатных осей.

Резюме

— Мы договорились?
— Да, принц!
— Значит, я ставлю ультиматум...
— Да. А я захожу сзади.

из к/ф «Формула любви»

- Для создания точек ветвления и многократного повторения блоков команд в программе используют управляющие инструкции: операторы цикла (`for`, `while` и `do-while`), условный оператор `if` и оператор выбора `switch`.
- Синтаксис вызова условного оператора `if` таков: после ключевого слова `if` в круглых скобках указывается условие (выражение, результатом которого является значение типа `boolean`). Если условие истинно (значение `true`), то выполняется блок команд, указанный далее в фигурных скобках, если условие ложно — блок команд, размещенный после ключевого слова `else`. Эта часть условного оператора не является обязательной.
- В операторе выбора `switch` после ключевого слова `switch` в круглых скобках указывается выражение, значением которого может быть число, символ или текст. В зависимости от значения этого выражения выполняется один из `case`-блоков оператора. Такой блок начинается с ключевого слова `case` и контрольного значения, которое может принимать выражение. Если найдено совпадение значения выражения и контрольного значения в `case`-блоке, то выполняются команды, начиная с этого блока и вплоть до конца оператора выбора или до первой инструкции `break`. В операторе выбора может использоваться необязательный блок `default`, команды которого выполняются, если при сравнении значения выражения с контрольными значениями совпадений не найдено.
- Синтаксис вызова оператора цикла `for` — следующий. В круглых скобках после ключевого слова `for` указывается три блока команд. Блоки разделяются точкой с запятой. В первом блоке (блок инициализации) размещаются команды, запускаемые один раз в начале выполнения оператора цикла. Второй блок содержит условие. Оператор цикла выполняется, пока истинно условие. В третьем блоке обычно размещаются команды для изменения переменных, используемых в операторе цикла. После этого в фигурных скобках указывается блок команд тела оператора цикла. Первый и третий блоки могут содержать по несколько команд. Команды в блоке разделяются запятыми. Допускается

использование пустых блоков. Выполняется оператор цикла по следующей схеме: сначала один раз выполняются команды первого блока, затем проверяется условие, выполняются команды в теле оператора цикла, выполняются команды третьего блока, проверяется условие во втором блоке, и так далее.

- Оператор цикла `while` работает по следующей схеме. Сначала проверяется условие, указанное в круглых скобках после ключевого слова `while`. Если условие истинно (значение `true`), то выполняются команды в теле оператора цикла (в фигурных скобках). Затем снова проверяется условие, и так далее. Если при проверке условия оказывается, что оно ложно (значение `false`), то выполнение оператора цикла завершается.
- При вызове оператора `do-while` используется следующий синтаксис. После ключевого слова `do` в фигурных скобках указывается блок команд. Затем указывается ключевое слово `while` и, в круглых скобках, проверяемое условие. Заканчивается инструкция точкой с запятой. Принцип выполнения оператора — такой же, как оператора `while`, с той лишь разницей, что сначала выполняются команды в теле оператора цикла, а затем проверяется условие.

3 Массивы

Ну и что вы скажете обо всем этом, Ватсон?

из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Под *массивом* подразумевают набор однотипных значений (переменных), к которым можно обращаться по общему имени. Именно массивам посвящена эта глава.

Переменные, относящиеся к одному массиву, называются *элементами* этого массива. Чтобы однозначно идентифицировать элемент массива, необходимо знать имя массива и позицию (размещение) элемента в массиве. Позиция элементов в массиве определяется с помощью целочисленных *индексов*. Количество индексов, необходимых для идентификации элемента массива, называется *размерностью массива*. Одномерный массив — это такой массив, в котором идентификация элементов осуществляется с помощью одного индекса.

Одномерные массивы

Это дело очень интересное. И простое!

из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Одномерный массив удобно себе представлять в виде упорядоченной цепочки или последовательности переменных (одного типа). Для объявления одномерного массива необходимо задать тип, к которому относятся элементы массива, название массива, а также количество элементов, входящих в массив. Синтаксис объявления одномерного массива такой:

```
тип[] имя=new тип[размер];
```

Вначале указывается `тип` элементов массива, а после идентификатора типа следуют пустые квадратные скобки. Далее указывается `имя` массива, оператор

присваивания, инструкция (оператор) `new`, снова `тип` элементов массива и в квадратных скобках `размер` массива (количество элементов в массиве). Например, командой `int nums=new int[20]` объявляется целочисленный массив `nums` из `20` элементов.

Строго говоря, представленная здесь команда объявления массива является симбиозом двух команд: команды `int[] nums` объявления переменной `nums` типа «целочисленный массив» (*переменная массива*) и инструкции `new int[20]`, которой, собственно, и создается массив. Данная инструкция присваивается значением переменной `nums`, и в результате ссылки на массив записывается в переменную `nums`. Другими словами, процесс создания массива можно выполнить двумя командами:

```
int[] nums;
nums=new int[20];
```

Причем эти команды могут быть разнесены в программном коде — то есть мы можем объявить переменную массива и лишь затем, в другом месте кода, создать массив (и записать ссылку на этот массив в переменную массива).

ПОДРОБНОСТИ

Ранее мы в основном имели дело с базовыми, или примитивными, типами (такими, как `int`, `char` или `double`). Переменная базового типа *хранит* значение. Технически это выглядит так: под переменную выделяется место в памяти, и значение переменной записывается именно в это место. Но есть другой способ работы с данными, при котором переменная *ссылается* на данные. Так происходит с объектами, и так реализуются массивы. Есть собственно массив, но доступ к нему мы получаем не напрямую, а с помощью посредника, которым является переменная массива. Значением переменной массива является не массив, а адрес массива. Поэтому создание переменной массива не означает создания массива. Массив создается отдельно. Мы будем описывать данную ситуацию как такую, при которой переменная массива ссылается на массив. Каждый раз, когда нам нужно будет получить доступ к массиву, мы будем обращаться к переменной массива, которая ссылается на данный массив.

Хотя на первый взгляд такая схема может показаться излишней, тем не менее она имеет свои преимущества. И мы в этом убедимся.

В случаях, когда не возникает недоразумений, мы будем переменную массива отождествлять с массивом.

При объявлении переменной массива допускается указывать квадратные скобки либо после идентификатора типа, либо после имени массива. Например, вместо команды `int[] nums` можно использовать команду `int nums[]`.

Обращение к элементу одномерного массива осуществляется через имя массива с указанием в квадратных скобках индекса элемента. Индексация элементов

массива начинается с нуля. Таким образом, ссылка на первый элемент массива `nums` будет иметь вид `nums[0]`. Если в массиве `20` элементов, то последний элемент массива имеет индекс `19`, то есть инструкция обращения к элементу выглядит как `nums[19]`.

Длину массива можно узнать с помощью свойства `length`: указывается имя массива и, через точку, свойство `length`. Например, чтобы узнать длину массива `nums`, можно воспользоваться инструкцией `nums.length`. Тогда ссылка на последний элемент массива может быть записана как `nums[nums.length-1]`.

НА ЗАМЕТКУ

В Java используется автоматическая проверка на предмет выхода за пределы массива. Поэтому если в коде выполняется обращение к несуществующему элементу массива, то возникает ошибка.

При объявлении массива для него выделяется память. В Java элементы массива автоматически инициализируются с нулевыми значениями — выделенные ячейки обнуляются, а значения этих обнуленных ячеек интерпретируются в зависимости от типа массива. Но на такую автоматическую инициализацию полагаться не стоит. Разумно инициализировать элементы массива в явном виде. Для этого используют оператор цикла или задают список значений элементов при объявлении массива.

Для инициализации массива списком значений при объявлении переменной массива после нее указывается (через оператор присваивания) заключенный в фигурные скобки список значений. Например:

```
int[] data={3,8,1,7};
```

Здесь объявляется переменная `data` для целочисленного массива, создается массив, и ссылка на него записывается в эту переменную. Размер массива и значения элементов определяются автоматически в соответствии с количеством элементов в списке значений. В данном случае создается целочисленный массив из четырех элементов со значениями элементов `3`, `8`, `1` и `7`. Того же результата можно добиться, воспользовавшись, например, следующими командами:

```
int[] data;
data=new int[]{3,8,1,7};
```

Первой командой `int[] data` объявляется переменная массива. Инструкцией `new int[]{3,8,1,7}` создается массив из четырех целых чисел, а ссылка на этот массив присваивается переменной `data` (имеется в виду команда `data=new int[]{3,8,1,7}`).

Пример объявления, инициализации и использования массивов приведен в листинге 3.1.

Листинг 3.1. Знакомство с одномерными массивами

```
class Demo{
   public static void main(String[] args){
      // Целочисленные переменные:
      int i,n;
      // Объявление переменной массива:
      int[] data;
      // Первый массив:
      data=new int[]{3,8,1,7};
      // Размер массива:
      n=data.length;
      // Второй массив:
      int[] nums=new int[n];
      // Заполнение второго массива:
      for(i=0;i<nums.length;i++){
         nums[i]=2*data[i]-3;
         System.out.println("nums["+i+"]="+nums[i]);
      }
   }
}
```

В программе объявляется и инициализируется массив `data` из четырех элементов. Длина массива вычисляется инструкцией `data.length`. Это значение записывается в целочисленную переменную `n` (команда `n=data.length`). Далее командой `int[] nums=new int[n]` объявляется еще один целочисленный массив `nums`. Количество элементов в этом массиве определяется значением переменной `n`, поэтому совпадает с размером массива `data`. Заполнение массива `nums` выполняется с помощью оператора цикла. Значения элементов массива `nums` вычисляются на основе значений элементов массива `data` (команда `nums[i]=2*data[i]-3`). Для отображения значений элементов массива `nums` использована команда `System.out.println("nums["+i+"]="+nums[i])`.

Ниже показано, как выглядит результат выполнения программы:

Результат выполнения программы (из листинга 3.1)

```
nums[0]=3
nums[1]=13
nums[2]=-1
nums[3]=11
```

Еще раз отметим, что индексация элементов массива начинается с нуля. Поэтому в операторе цикла индексная переменная `i` инициализируется с начальным нулевым значением, а в проверяемом условии `i<nums.length` использован оператор строгого неравенства.

Немаловажно и то обстоятельство, что при создании массива `nums` его размер определяется с помощью переменной `n`, значение которой вычисляется в процессе выполнения программы.

Двумерные и многомерные массивы

— Вы хотите сказать, что вам уже все ясно?

— Не хватает некоторых деталей. Но не в этом суть.

из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Размерность массива может быть больше единицы. Но на практике массивы размерности выше второй используют редко. Далее рассмотрим способы объявления, инициализации и использования двумерных массивов, в которых доступ к элементу массива осуществляется с помощью двух индексов. Если представлять двумерный массив как таблицу, то первый индекс элемента определяет строку, в которой находится элемент, а второй индекс определяет столбец, в котором находится элемент.

ПОДРОБНОСТИ

Хотя двумерный массив и удобно представлять как таблицу, реализуется он совершенно иначе. На самом деле двумерный массив в Java — это одномерный массив, элементами которого являются переменные массива. Каждая такая переменная ссылается на одномерный массив. При использовании этой конструкции возникает иллюзия, что мы имеем дело с таблицей.

Создаются двумерные массивы так же просто, как и одномерные. Но при объявлении переменной для двумерного массива после идентификатора типа указываются две пары пустых квадратных скобок, а в инструкции создания двумерного массива — в отдельных квадратных скобках — указывается размер по каждому из индексов (количество строк и столбцов в массиве). Синтаксис объявления двумерного массива выглядит так:

```
тип[][] имя=new тип[размер][размер];
```

Как и в случае одномерного массива, данная команда может быть разбита на две:

```
тип[][] имя;
имя=new тип[размер][размер];
```

Первой командой объявляется переменная для двумерного массива. Второй командой создается собственно двумерный массив с указанными размерами, и ссылка на этот массив присваивается в качестве значения переменной массива. Например, командой `double[][] data=new double[3][4]` создается двумерный массив с элементами типа `double`. В массиве `3` строки и `4` столбца, а ссылка на массив записывается в переменную `data`. К тому же результату приведет выполнение следующих команд:

```
double[][] data;
data=new double[3][4];
```

Обращение к элементам двумерного массива выполняется в следующем формате: указывается имя массива, в квадратных скобках — первый индекс элемента, в других квадратных скобках — второй индекс элемента массива. Индексация по всем размерностям начинается с нуля. Например, ссылка `data[0][3]` является обращением к элементу массива `data` с индексами `0` и `3`, и это элемент в первой строке и в четвертом столбце.

Для инициализации двумерного массива используют вложенные операторы цикла или список, состоящий из списков значений. Каждый такой внутренний список определяет значения элементов массива в строке. Ниже приведены примеры инициализации двумерного массива с помощью списка:

```
double[][] data={{0.1,0.2,0.3},{0.4,0.5,0.6}};
int nums[][]={{1,2,3},{4,5}};
```

Первой командой создается и инициализируется двумерный массив `data` размером `2` на `3` (две строки и три столбца). Количество строк определяется количеством элементов во внешнем списке. Таких элементов два — это списки `{0.1,0.2,0.3}` и `{0.4,0.5,0.6}`. В каждом из этих списков по три элемента. Отсюда и получается массив размерами `2` на `3`. Список `{0.1,0.2,0.3}` определяет значения элементов в первой строке, список `{0.4,0.5,0.6}` определяет значения элементов во второй строке. Например, элемент `data[0][0]` получает значение `0.1`, элемент `data[0][2]` — значение `0.3`, элемент `data[1][0]` — значение `0.4`, а элемент `data[1][2]` — значение `0.6`.

Второй командой создается целочисленный массив `nums`, который состоит из двух строк (поскольку внутри присваиваемого списка два элемента — списки `{1,2,3}` и `{4,5}`). Однако в первой строке созданного массива содержится три элемента (поскольку в списке `{1,2,3}` три значения), а во второй строке массива — два элемента (поскольку в списке `{4,5}` два значения). В созданном массиве элемент `nums[0][0]` имеет значение `1`, элемент `nums[0][1]` — значение `2`, элемент `nums[0][2]` — значение `3`, элемент `nums[1][0]` — значение `4`, а элемент `nums[1][1]` — значение `5`.

ПОДРОБНОСТИ

Проблемы в том, что массив `nums` в разных строках содержит разное количество элементов, нет. Технически все реализуется более чем просто. Переменная двумерного массива `nums` ссылается на самом деле на одномерный массив из двух элементов (количество строк в двумерном массиве). Но элементы этого массива не целые числа, а переменные, которые могут ссылаться на одномерные целочисленные массивы (условно говоря, элементы относятся к типу `int[]`). Первая переменная ссылается на одномерный массив из трех элементов (`1`, `2` и `3`), а вторая переменная ссылается на одномерный массив из двух элементов (`4` и `5`). Когда мы индексируем (одним индексом!) переменную `nums`, то получаем доступ к элементу одномерного массива, на который ссылается эта переменная. Например, `nums[0]` — первый элемент, а `nums[1]` — второй элемент упомянутого массива из переменных массива. И данные элементы являются ссылками на массивы.

Их можно индексировать. Поэтому, скажем, `nums[0][1]` — это второй элемент в массиве, на который ссылается первый элемент `nums[0]` в массиве, на который ссылается переменная `nums`. Так все происходит на самом деле. А интерпретируем мы инструкцию `nums[0][1]` как обращение к элементу в первой строке и во втором столбце двумерного массива.

В листинге 3.2 приведен пример программы, в которой создается двумерный массив, заполняемый с помощью вложенных операторов цикла.

Листинг 3.2. Создание двумерного массива

```
class Demo{
   public static void main(String[] args){
      int i,j,n=3,val=1;
      // Создание двумерного массива:
      int[][] nums=new int[n-1][n];
      // Вложенные операторы цикла:
      for(i=0;i<n-1;i++){ // Перебор строк массива
         for(j=0;j<n;j++){ // Перебор столбцов массива
            // Заполнение элементов массива:
            nums[i][j]=val++;
            // Отображение значения элемента:
            System.out.print(nums[i][j]+" ");
         }
         // Переход к новой строке:
         System.out.println();
      }
   }
}
```

Командой `int[][] nums=new int[n-1][n]` создается целочисленный массив `nums`, в котором `n-1` строк и `n` столбцов. Переменной `n` предварительно присвоено значение `3`. Заполняется массив с помощью вложенных операторов цикла. Значение элементу массива (при заданных индексах `i` и `j`) присваивается командой `nums[i][j]=val++`. Здесь элементу `nums[i][j]` присваивается текущее значение переменной `val`, а сразу после этого значение переменной `val` увеличивается на единицу.

НА ЗАМЕТКУ

В результате выполнения инструкции `val++` значение переменной `val` увеличивается на единицу. Но поскольку использована постфиксная форма оператора инкремента, то значением выражения `val++` является старое (до увеличения на единицу) значение переменной `val`.

После вычисления значения элемента оно отображается в области вывода. В результате выполнения программы получаем:

Результат выполнения программы (из листинга 3.2)

```
1 2 3
4 5 6
```

В листинге 3.3 приведен код программы, в которой создается двумерный массив со строками разной длины.

Листинг 3.3. Массив со строками разной длины

```
class Demo{
   public static void main(String[] args){
      int i,j,val=1;
      // Создание массива (второй размер не указан):
      int[][] nums=new int[4][];
      // Цикл для создания треугольного массива:
      for(i=0;i<nums.length;i++){
         // Создание строки в массиве:
         nums[i]=new int[i+1];
      }
      // Заполнение массива:
      for(i=0;i<nums.length;i++){
         for(j=0;j<nums[i].length;j++){
            // Значение элемента массива:
            nums[i][j]=val++;
            // Отображение значения элемента:
            System.out.print(nums[i][j]+" ");
         }
         // Переход к новой строке:
         System.out.println();
      }
   }
}
```

Командой `int[][] nums=new int[4][]` создается двумерный целочисленный массив `nums`. Этот массив состоит из четырех строк. Однако размер строк не указан — вторая пара квадратных скобок в конце команды пуста. Точнее, строк еще нет. Их нужно создать. Строки создаются в операторе цикла. В этом операторе цикла индексная переменная `i` пробегает значения от `0` до `nums.length-1` включительно. Командой `nums[i]=new int[i+1]` создается строка двумерного массива с индексом `i`. Такая строка содержит `i+1` элемент.

ПОДРОБНОСТИ

Технически все происходит так: инструкцией `new int[i+1]` создается одномерный массив (строка для двумерного массива) и ссылка на этот массив записывается в переменную `nums[i]`. При этом `nums[i]` можно интерпретировать как ссылку на строку с индексом `i`.

> Смысл инструкции `nums.length` станет понятен, если вспомнить, что на самом деле двумерный массив представляет собой одномерный массив, элементы которого ссылаются на одномерные массивы. В таком случае `nums.length` дает значение для количества элементов в массиве, на который ссылается переменная `nums`, — то есть это количество строк в двумерном массиве.

В результате мы получаем двумерный массив треугольного вида: в первой строке массива один элемент, во второй — два элемента, и так далее, вплоть до четвертой строки массива.

Заполняется созданный массив с помощью вложенных операторов цикла. Индексная переменная `i` во внешнем операторе цикла принимает значения от `0` до `nums.length-1` и определяет строку в двумерном массиве `nums`. Для индексной переменной `j` верхняя граница диапазона изменения определяется инструкцией `nums[i].length`, которая возвращает количество элементов в строке с индексом `i` (переменная `j` изменяется от значения `0` до значения `nums[i].length-1` включительно).

ПОДРОБНОСТИ

Следует принять во внимание, что `nums[i]`, по сути, является переменной, которая ссылается на одномерный массив, формирующий строку с индексом `i`. Количество элементов в этом массиве (строке) определяется выражением `nums[i].length`.

Значение элементам массива присваивается командой `nums[i][j]=val++`. Ниже показано, как выглядит результат выполнения программы:

Результат выполнения программы (из листинга 3.3)

```
1
2 3
4 5 6
7 8 9 10
```

Как и ожидалось, мы получили массив, у которого в разных строках содержится разное количество элементов.

НА ЗАМЕТКУ

Массивы, размерность которых больше двух, создаются аналогично. При объявлении переменной для многомерного массива после идентификатора типа указываются пары пустых скобок. Количество таких пар определяется размерностью массива. В команде создания массива в отдельных квадратных скобках указывается размер по каждому из индексов.

Символьные и текстовые массивы

— Готовы ли вы ответствовать?
— Вопрошайте.
— Готовы ли вы сказать нам всю правду?
— Ну, всю не всю... А что вас интересует?

из к/ф «Формула любви»

В Java символьные и текстовые массивы имеют некоторые особенности — не то чтобы серьезные, но все же.

НА ЗАМЕТКУ

Одномерный символьный массив представляет собой набор символов. Текст по своей сути это тоже набор символов. И хотя в Java текст реализуется как объект класса `String`, но естественная связь между символьными массивами и текстовыми значениями есть. Как минимум, на практике нередко приходится преобразовывать символьные массивы в текст и на основе текста создавать символьные массивы.

Мы начнем с несложного примера символьного массива (листинг 3.4).

Листинг 3.4. Символьные массивы

```
import java.util.Arrays;
class Demo{
   public static void main(String[] args){
      // Символьный массив:
      char[] symbs={'J','a','v','a'};
      // Текстовое представление для массива:
      String str=Arrays.toString(symbs);
      // Создание текста на основе массива:
      String txt=new String(symbs);
      // Отображение результата:
      System.out.println(symbs);
      System.out.println(str);
      System.out.println(txt);
      // Создание массива на основе текста:
      char[] word="Pascal".toCharArray();
      // Отображение результата:
      System.out.println(Arrays.toString(word));
   }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 3.4)

```
Java
[J, a, v, a]
Java
[P, a, s, c, a, l]
```

В программе мы создаем символьный массив командой `char[] symbs={'J', 'a','v','a'}`. С одной стороны, это самый обычный массив. С другой стороны, если передать символьный массив аргументом методу `println()`, как в команде `System.out.println(symbs)`, то будет отображаться содержимое символьного массива, причем без знаков разделения, как текст. Это специфика работы метода `println()`. Увидеть символьный массив в виде массива можно, воспользовавшись статическим методом `toString()` из класса `Arrays` (класс импортируется в программу инструкцией `import java.util.Arrays`). Методу аргументом передается переменная массива (причем это не обязательно должен быть символьный массив), а результатом метод возвращает текстовое представление для массива, в котором элементы разделяются запятыми, а вся конструкция заключена в квадратные скобки. Мы используем данный метод в команде `String str=Arrays.toString(symbs)`. В результате в текстовую переменную `str` записывается текстовое представление для массива `symbs`. Для создания текста на основе символьного массива можно воспользоваться командой `String txt=new String(symbs)`. Обратная задача, когда на основе текста создается символьный массив, решается с помощью метода `toCharArray()`, который вызывается из текстового объекта и результатом возвращает символьный массив, сформированный из букв в тексте. Примером служит команда `char[] word="Pascal".toCharArray()`.

ПОДРОБНОСТИ

Отметим несколько обстоятельств. Во-первых, в классе `Arrays` есть много полезных и эффективных методов для работы с массивами — не только символьными. Во-вторых, текст мы будем обсуждать в одной из следующих глав. Пока же напомним, что в действительности текстовые значения реализуются как объекты класса `String`. Причем текстовые литералы также являются объектами. У объектов есть методы. Среди методов текстовых объектов имеется метод `toCharArray()`. Этот метод позволяет на основе текста создать символьный массив. Текст считывается из того объекта, из которого вызывается метод.

Что касается создания текстовых объектов, то ранее мы поступали так: объявляли переменную класса `String` и значением переменной присваивали текстовый литерал. Но есть и другие способы создания текстовых объектов. В частности, можно создавать объекты с помощью инструкции `new`. Так, например, поступают, когда создают текст на основе символьного массива.

Еще один пример, который мы рассмотрим, иллюстрирует работу с текстовыми массивами (листинг 3.5).

Листинг 3.5. Использование текстового массива

```
import static javax.swing.JOptionPane.*;
import javax.swing.ImageIcon;
class Demo{
   public static void main(String[] args){
      // Путь к директории с файлами:
      String path="D:/Pictures/Animals/";
      // Текстовый массив с названиями хищников:
      String[] names={"Лев","Тигр","Медведь"};
      // Текстовый массив с названиями файлов:
      String[] files={"lion.png","tiger.png","bear.png"};
      // Отображение окна с раскрывающимся списком:
      String animal=(String)showInputDialog(
         null, // Ссылка на родительское окно
         "Какого хищника показать?", // Текст над списком
         "Хищники", // Заголовок окна
         QUESTION_MESSAGE, // Пиктограмма
         null, // Изображение для пиктограммы
         names, // Массив определяет содержимое списка
         names[0] // Выбранный элемент списка
      );
      // Индекс выбранного элемента:
      int index=0;
      // Определение индекса:
      for(int k=1;k<names.length;k++){
         if(animal.equals(names[k])){
            index=k;
            break;
         }
      }
      // Создание объекта изображения:
      ImageIcon img=new ImageIcon(path+files[index]);
      // Окно с изображением:
      showMessageDialog(null,img,animal,PLAIN_MESSAGE);
   }
}
```

При запуске программы появляется окно (рис. 3.1).

Оно напоминает окно с полем ввода, но вместо поля ввода в данном случае используется раскрывающийся список, в котором можно выбрать название хищника (рис. 3.2).

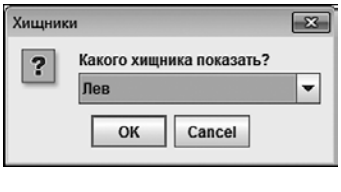


Рис. 3.1. Окно с раскрывающимся списком

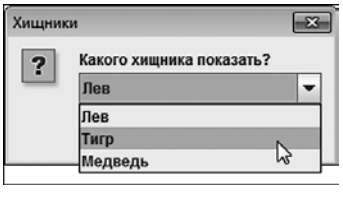


Рис. 3.2. Выбор пункта в раскрывающемся списке

Пользователю нужно выбрать одну из позиций в раскрывающемся списке и нажать кнопку OK. После этого появляется окно с изображением соответствующего зверя. Если пользователь в списке выбирает пункт Лев, то следующее окно будет таким, как показано на рис. 3.3.

Если выбран пункт Тигр, то окно с изображением будет таким, как показано на рис. 3.4.

Наконец, на рис. 3.5 показано, как выглядит окно с изображением медведя, которое появляется, если пользователь в раскрывающемся списке выбирает пункт Медведь.

Рис. 3.3. Окно с изображением льва

Рис. 3.4. Окно с изображением тигра

Рис. 3.5. Окно с изображением медведя

ПОДРОБНОСТИ

Для корректной работы программы в директории `D:\Picture\Animals` размещаются файлы `lion.png`, `tiger.png` и `bear.png` размером 180 пикселов в ширину и 180 пикселов в высоту. В них содержатся картинки, которые отображаются в диалоговом окне в процессе выполнения программы.

В текстовую переменную `path` записывается путь к директории, в которой хранятся графические файлы (значение `"D:/Pictures/Animals/"`). Командой `String[] names={"Лев","Тигр","Медведь"}` создается текстовый массив из трех элементов, определяющих названия хищников. Мы используем данный массив для формирования раскрывающегося списка.

НА ЗАМЕТКУ

Поскольку текст реализуется в виде объекта класса `String`, то создание текстового массива — это, строго говоря, создание массива объектов. Массив объектов в общем случае создается так: создается массив из объектных переменных; эти переменные ссылаются на объекты. Создать именно текстовый массив относительно просто. Для этого достаточно объявить переменную для текстового массива и в качестве значения присвоить ей список из текстовых литералов, как мы и поступили в программе.

Еще один текстовый массив `files` содержит названия файлов с картинками (это файлы `lion.png`, `tiger.png` и `bear.png`). Между названиями животных (массив

names) и названиями файлов (массив files) должно быть строгое однозначное соответствие.

Для отображения окна с раскрывающимся списком мы вызываем метод showInputDialog(), который ранее использовали для отображения окна с полем ввода. Но на этот раз аргументы методу мы передаем немного иначе. Первым аргументом передается пустая ссылка null на родительское окно (которого нет). Второй аргумент (литерал "Какого хищника показать?") определяет текст, который будет отображаться над раскрывающимся списком. Третий аргумент (литерал "Хищники") определяет заголовок окна. Четвертый аргумент, которым указана константа QUESTION_MESSAGE, определяет тип пиктограммы в окне. Если бы мы этим ограничились, то получили бы окно с полем ввода. Но мы передаем методу еще три аргумента. Пятый аргумент в общем случае является ссылкой на объект изображения. Это изображение используется в качестве пиктограммы в окне. В таком случае предыдущий, четвертый аргумент (который определяет тип пиктограммы) игнорируется. Но мы в данном случае указали значение null. Это пустая ссылка, которая означает, что картинки для пиктограммы нет. В таком случае пиктограмма определяется предыдущим, четвертым аргументом. Шестым аргументом указан массив names. На основе элементов этого массива сформируется раскрывающийся список. Причем когда окно отображается, какой-то элемент уже должен быть выбран в списке. Этот по умолчанию выбранный элемент определяется седьмым аргументом. Мы указали седьмым аргументом инструкцию names[0] (первый элемент в массиве names), поэтому в самом начале выбран первый элемент из списка.

Версия метода showInputDialog(), которую мы в данном случае вызываем, результатом возвращает ссылку на элемент, который был выбран в раскрывающемся списке на момент нажатия кнопки OK в окне. Эта ссылка относится к классу Object. Класс Object — особый. Он находится в вершине иерархии наследования и обладает одним важным свойством: переменная класса Object может ссылаться на значение любого типа. Фактически это означает, что метод возвращает ссылку и тип этой ссылки не конкретизирован. Но мы знаем, что для формирования раскрывающегося списка использовали текстовый массив, и, как следствие, в списке будет выбран текстовый элемент. Поэтому ссылку, которую возвращает метод showInputDialog(), мы приводим к типу String, для чего перед командой вызова метода размещаем инструкцию (String). Результат записываем в текстовую переменную animal. В итоге в эту переменную будет записано название животного, которое выбрал пользователь в раскрывающемся списке. Нас интересует еще и индекс соответствующего элемента в массиве names. Для этого запускается оператор цикла, в котором перебираются все элементы массива names, начиная со второго. Ищется совпадение значения элемента массива и значения переменной animal. Текстовые значения сравниваются с помощью метода equals(), который вызывается из одного текстового объекта, а аргументом методу передается другой текстовый объект. Если совпадение есть, то значение индекса элемента, на котором найдено совпадение, записывается в переменную index, и инструкцией break завершается

выполнение оператора цикла. Если совпадений не найдено, то переменная `index` остается со своим исходным нулевым значением.

После того как индекс выбранного в раскрывающемся списке элемента определен, командой `ImageIcon img=new ImageIcon(path+files[index])` создается объект изображения. В этой команде используется элемент массива `files` с тем же индексом `index`, что и у элемента, выбранного в раскрывающемся списке. Окно с картинкой отображается командой `showMessageDialog(null,img,animal,PLAIN_MESSAGE)`.

У задачи имеется и другое, более простое решение: можно вместо окна с раскрывающимся списком показать окно с кнопками, названия которых совпадают с названиями животных. Нажатие кнопки приводит к отображению окна с соответствующей картинкой. Рассмотрим код в листинге 3.6.

Листинг 3.6. Текстовый массив и выбор изображения

```
import static javax.swing.JOptionPane.*;
import javax.swing.ImageIcon;
class Demo{
   public static void main(String[] args){
      // Путь к директории с файлами:
      String path="D:/Pictures/Animals/";
      // Текстовый массив с названиями хищников:
      String[] names={"Лев","Тигр","Медведь"};
      // Текстовый массив с названиями файлов:
      String[] files={"lion.png","tiger.png","bear.png"};
      // Отображение окна с набором кнопок:
      int index=showOptionDialog(
         null, // Ссылка на родительское окно
         "Выберите хищника", // Текст сообщения
         "Хищники", // Заголовок окна
         DEFAULT_OPTION, // Тип окна по умолчанию
         WARNING_MESSAGE, // Пиктограмма
         null, // Изображение для пиктограммы
         names, // Массив определяет набор кнопок
         names[0] // Активная по умолчанию кнопка
      );
      // Создание объекта изображения:
      ImageIcon img=new ImageIcon(path+files[index]);
      // Окно с изображением:
      showMessageDialog(
         null,img,names[index],PLAIN_MESSAGE
      );
   }
}
```

При запуске программы появляется окно (рис. 3.6).

Окно содержит кнопки с названиями Лев, Тигр и Медведь. После нажатия одной из кнопок появляется окно с картинкой — окна такие же, как и в предыдущем примере (см. рис. 3.3–3.5). Теперь проанализируем этот код, а именно то место, которое

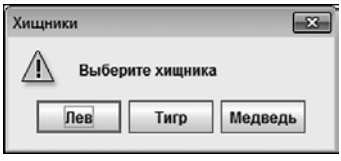


Рис. 3.6. Окно с набором кнопок для выбора хищника

принципиально отличается от кода в листинге 3.5. Для отображения окна с набором кнопок мы вызываем статический метод `showOptionDialog()` из класса `JOptionPane`. У метода довольно много аргументов. Первый аргумент (значение `null`) является ссылкой на родительское окно (его в данном случае нет). Второй аргумент (значение `"Выберите хищника"`) определяет текст сообщения в окне. Третий аргумент (значение `"Хищники"`) задает заголовок окна. Четвертый аргумент (константа `DEFAULT_OPTION`) определяет стандартный набор кнопок в окне. Но этот аргумент вступает в игру, только если два последних аргумента равны `null`. В нашем случае это не так, поэтому конкретное значение четвертого аргумента не влияет на вид окна. Пятый аргумент (константа `WARNING_MESSAGE`) определяет пиктограмму для окна. Но если бы шестым аргументом не была указана пустая ссылка `null`, то предыдущий аргумент был бы не важен (поскольку в общем случае шестым аргументом передается изображение для пиктограммы). Седьмым аргументом указан массив `names`, элементы которого определяют набор кнопок в окне. Восьмым аргументом указан первый элемент этого массива (значение `names[0]`). Этот аргумент определяет кнопку, которой будет передан фокус при отображении окна (в области названия этой кнопки отображается рамка фокуса — см. рис. 3.6). Результатом метод возвращает индекс кнопки, которую нажимает пользователь. Значение записывается в переменную `index` и затем используется при создании объекта изображения.

Присваивание и сравнение массивов

Простые вещи разучились делать!

из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Если тип элементов и тип размерности совпадает, то одной переменной массива в качестве значения можно присвоить другую переменную массива. Но копирования массивов при этом не происходит. Чтобы понять почему, имеет смысл вспомнить технические подробности реализации массивов в Java.

В первую очередь следует учесть, что объявление переменной массива не означает создания массива. Другими словами, непосредственно массив и переменная массива — это далеко не одно и то же. В этом смысле показательным является двухэтапный (двумя командами) процесс создания массива. Например:

```
int[] nums;
nums=new int[]{1,2,3,4};
```

В данном случае команда `int[] nums` есть не что иное, как объявление переменной `nums`. Тип этой переменной — «массив целых чисел». Значением переменной может быть ссылка (адрес) на какой-нибудь массив, состоящий из целых чисел.

Оператор `new` в общем случае служит для динамического выделения памяти под различные объекты, включая массивы. Командой `new int[]{1,2,3,4}` в памяти выделяется место для целочисленного массива из четырех элементов, элементам массива присваиваются соответствующие значения. У этого вновь созданного массива есть адрес (ссылка на массив). В качестве значения оператор `new` возвращает ссылку на созданный объект. В данном случае возвращается ссылка на массив, присваиваемая переменной `nums` как значение. Теперь несложно догадаться, каким будет результат выполнения следующих команд:

```
int[] nums,data;
nums=new int[]{1,2,3,4};
data=nums;
```

Первой командой `int[] nums,data` объявляются две переменные массива — `nums` и `data`. Второй командой `nums=new int[]{1,2,3,4}` создается массив, а ссылка на него присваивается в качестве значения переменной `nums`. Далее командой `data=nums` значение переменной `nums` присваивается переменной `data`. Однако значение переменой `nums` — это ссылка на массив. Поэтому после присваивания переменная `data` ссылается на тот же массив! Например, элемент `data[1]` имеет такое же значение, что и `nums[1]` (значение 2). Точнее, это один и тот же элемент. Более того, если теперь изменить какой-нибудь элемент массива `data` (например, командой `data[3]=-1`), автоматически изменится и соответствующий элемент массива `nums`. Причина все та же — массив на самом деле один, просто на него ссылаются две переменные.

Если попытаться сравнивать массивы с помощью операторов равно == и не равно != (например, `nums==data` или `nums!=data`), будут сравниваться значения переменных массива, а не элементы в этих массивах. Поэтому результатом выражения `nums==data` является значение `true`, если обе переменные, `nums` и `data`, ссылаются на один и тот же массив.

НА ЗАМЕТКУ

Для сравнения двух физически разных массивов на предмет совпадения значений элементов нужно проверить, одинаковое ли количество элементов в массивах, и затем сравнить значения всех соответствующих элементов.

Такой способ реализации массивов (с использованием переменной массива) хотя и не самый простой, но позволяет эффективно решать многие задачи. Например, после создания массива его размер изменить нельзя. Но зато можно создать иллюзию того, что размер массива изменился. Допустим, что имеется некоторый массив

и мы хотим удалить из него определенный элемент. Для этого создаем новый массив, количество элементов в котором на один меньше, чем в исходном. Копируем из исходного массива значения всех элементов, за исключением удаляемого. Затем ссылку на новый массив записываем в переменную массива, которая вначале ссылалась на исходный массив. Поскольку доступ к массиву мы получаем через переменную массива, то создается впечатление, что из массива был удален элемент. Данный подход реализован в программе в листинге 3.7.

Листинг 3.7. Удаление элемента из массива

```
import java.util.*;
class Demo{
   public static void main(String[] args){
      // Исходный массив:
      int[] nums={10,20,30,40,50};
      System.out.println("Исходный массив:");
      // Отображается содержимое массива:
      System.out.println(Arrays.toString(nums));
      Scanner input=new Scanner(System.in);
      System.out.print("Индекс элемента для удаления: ");
      // Считывается индекс элемента (для удаления):
      int index=input.nextInt();
      // Если индекс попадает в допустимый диапазон:
      if((index>=0)&&(index<nums.length)){
         // Создание нового массива:
         int[] m=new int[nums.length-1];
         // Заполнение созданного массива:
         for(int k=0;k<index;k++){
            m[k]=nums[k];
         }
         for(int k=index+1;k<nums.length;k++){
            m[k-1]=nums[k];
         }
         // Переменной nums присваивается новое значение:
         nums=m;
      }
      System.out.println("После удаления элемента:");
      // Отображается содержимое массива:
      System.out.println(Arrays.toString(nums));
   }
}
```

В программе создается целочисленный массив `nums`, его содержимое отображается в окне вывода, после чего пользователю предлагается ввести индекс элемента, который должен быть удален. Введенное число записывается в переменную `index`. Процедура удаления выполняется, только если указанный индекс попадает в допустимый диапазон значений (условие `(index>=0)&&(index<nums.length)` в условном операторе). В условном операторе командой `int[] m=new int[nums.length-1]` создается новый массив `m`, размер которого на единицу меньше исходного массива `nums`. Затем с помощью двух операторов цикла массив `m` заполняется значениями

из массива nums, причем элемент с индексом index не копируется (собственно, именно поэтому использовано два оператора цикла). После заполнения массива m командой nums=m в переменную nums записывается ссылка на созданный массив. В итоге переменная nums ссылается на новый массив, у которого, по сравнению с исходным, отсутствует один из элементов. Результат выполнения программы может быть следующим (полужирным шрифтом выделено значение, введенное пользователем):

Результат выполнения программы (из листинга 3.7)

```
Исходный массив:
[10, 20, 30, 40, 50]
Индекс элемента для удаления: 2
После удаления элемента:
[10, 20, 40, 50]
```

В данном случае удален элемент с индексом 2.

ПОДРОБНОСТИ

В программе использованы классы Scanner и Arrays из подпакета util пакета java, поэтому код начинается инструкцией import java.util.*.

Еще одно замечание касается переменной m. Напомним, что область доступности переменной определяется блоком, в котором она объявлена. Данная переменная объявлена в условном операторе и потому доступна только в условном операторе. После того как условный оператор завершит выполнение, доступа к переменной больше не будет. Далее, в Java работает система автоматической сборки мусора: все объекты, на которые нет ссылок в программе, удаляются. Переменная m ссылается на массив. Если бы это была единственная переменная, ссылающаяся на массив, то после того, как она станет недоступной, автоматически бы удалялся и массив, на который переменная раньше ссылалась. Но в нашем случае после выполнения команды nums=m на массив ссылаются две переменные (m и nums). И когда переменная m уходит со сцены, переменная nums по-прежнему ссылается на массив, который остается доступным.

В двух последовательных операторах цикла использована индексная переменная k, причем она объявляется в каждом из операторов цикла. Причина двойного объявления — в том, что объявленная в операторе цикла переменная доступна только в этом операторе цикла. Поэтому эти переменные на самом деле разные, так как у них разная область доступности.

Процедура сравнения двух символьных массивов показана в листинге 3.8.

Листинг 3.8. Сравнение массивов

```
import java.util.*;
class Demo{
```

```
   public static void main(String[] args){
      Scanner input=new Scanner(System.in);
      // Переменные массива:
      char[] A,B;
      // Считывание текста:
      System.out.print("Первый массив: ");
      A=input.nextLine().toCharArray();
      System.out.print("Второй массив: ");
      B=input.nextLine().toCharArray();
      // Отображается содержимое массивов:
      System.out.println("A="+Arrays.toString(A));
      System.out.println("B="+Arrays.toString(B));
      // Сравнение переменных массива:
      System.out.print("Переменные массива: ");
      if(A==B) System.out.println("A==B");
      else System.out.println("A!=B");
      // Сравнение массивов:
      System.out.print("Сравнение массивов: ");
      boolean state=true;
      if(A.length!=B.length){
         state=false;
      }else{
         for(int k=0;k<A.length;k++){
            if(A[k]!=B[k]){
               state=false;
               break;
            }
         }
      }
      // Результат сравнения:
      if(state) System.out.println("A==B");
      else System.out.println("A!=B");
   }
}
```

При запуске программы пользователю предлагается ввести два текстовых значения, на основе которых создаются и далее сравниваются символьные массивы. Массивы создаются так: методом `nextLine()` считывается введенный пользователем текст, из которого вызывается метод `toCharArray()`, которым на основе текста и формируется символьный массив.

ПОДРОБНОСТИ

Мы используем объект `input` класса `Scanner`. Из этого объекта вызывается метод `nextLine()`. Значением инструкции `input.nextLine()` является ссылка на текстовый объект. У текстового объекта есть вызываемый метод `toCharArray()`. Результатом выражения `input.nextLine().toCharArray()` является ссылка на массив, который создан на основе текста, введенного пользователем.

После создания массивов мы выполняем два вида сравнений: сравниваем переменные массива A и B, а также массивы, на которые ссылаются переменные. Первая операция сводится к вычислению значения выражения A==B.

НА ЗАМЕТКУ

Поскольку переменные A и B ссылаются на физически разные массивы (даже если они одинаковые), то результатом выражения A==B всегда является значение false.

Для сравнения массивов используем логическую переменную state с начальным значением true. Сначала проверяем, совпадают ли их размеры (проверяется условие A.length!=B.length). Если размеры разные, то переменной state присваивается значение false. Если же размеры одинаковые, то начинается сравнение элементов двух массивов. Как только найдены элементы с разными значениями, переменной state присваивается значение false, а инструкцией break прекращается работа оператора цикла, в котором перебираются элементы массивов.

НА ЗАМЕТКУ

Если значение переменной state осталось равным true, то массивы одинаковые. Если значение переменной state стало равным false, то массивы разные.

Если пользователь вводит разные текстовые значения, то результат выполнения программы может быть таким (здесь и далее полужирным шрифтом выделены введенные пользователем значения):

Результат выполнения программы (из листинга 3.8)

```
Первый массив: Alpha
Второй массив: Bravo
A=[A, l, p, h, a]
B=[B, r, a, v, o]
Переменные массива: A!=B
Сравнение массивов: A!=B
```

Так выглядит результат выполнения программы, если пользователь вводит два совпадающих текстовых значения:

Результат выполнения программы (из листинга 3.8)

```
Первый массив: Alpha
Второй массив: Alpha
A=[A, l, p, h, a]
B=[A, l, p, h, a]
Переменные массива: A!=B
Сравнение массивов: A==B
```

А теперь переходим к рассмотрению более сложных примеров.

Использование массивов

> Значит, такое предложение: сейчас мы нажимаем на контакты и перемещаемся к вам. Но если эта машинка не сработает, тогда уж вы с нами переместитесь, куда мы вас переместим!
>
> *из к/ф «Кин-дза-дза»*

В этом разделе рассматриваются некоторые программы, в которых в том или ином виде используются массивы.

Кодирование и декодирование текста

Начнем с примера, в котором кодируется и декодируется текст с использованием символьных массивов (листинг 3.9).

Листинг 3.9. Кодирование и декодирование текста

```
import java.util.Scanner;
class Demo{
   public static void main(String[] args){
      Scanner input=new Scanner(System.in);
      System.out.print("Текст для кодирования: ");
      // Считывается текст:
      String text=input.nextLine();
      // Преобразование текста в символьный массив:
      char[] symbs=text.toCharArray();
      // Кодирование символов:
      for(int k=0;k<symbs.length;k++){
         symbs[k]=(char)(symbs[k]+k+1);
      }
      // Создание текста на основе массива:
      text=new String(symbs);
      System.out.println("После кодирования:");
      // Закодированный текст:
      System.out.println(text);
      // Создание массив на основе
      // закодированного текста:
      symbs=text.toCharArray();
      // Декодирование символов:
      for(int k=0;k<symbs.length;k++){
         symbs[k]=(char)(symbs[k]-k-1);
      }
      // Создание текста на основе массива:
      text=new String(symbs);
      System.out.println("После декодирования:");
      // Декодированный текст:
```

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

Результат выполнения программы может быть таким (полужирным шрифтом выделен введенный пользователем текст):

Результат выполнения программы (из листинга 3.9)

```
Текст для кодирования: Сообщение
После кодирования:
Трсеюлфро
После декодирования:
Сообщение
```

Мы считываем текст и записываем его в текстовую переменную `text`. На основе этого текста создается символьный массив `symbs`. Затем с помощью оператора цикла мы перебираем все символы в массиве и изменяем их. Массив используется для создания текста. Текст отображается в области вывода. После этого выполняется обратное преобразование (декодирование текста).

ПОДРОБНОСТИ

Для кодирования символов использована команда `symbs[k]=(char)(symbs[k]+k+1)` в операторе цикла (при заданном индексе `k`). Преобразование выполняется так: к текущему коду символа прибавляется порядковый номер символа в массиве; полученное значение обратно преобразуется в символ. Декодируются символы командой `symbs[k]=(char)(symbs[k]-k-1)`, которая из текущего кода символа вычитает порядковый номер символа в массиве, и полученное значение преобразует в символ.

Умножение векторов

С помощью массивов удобно реализовывать операции с векторами. Рассмотрим программу, в которой вычисляется скалярное и векторное произведения векторов. Под вектором будем подразумевать набор из трех чисел (координаты вектора). Допустим, $\vec{a}$ и $\vec{b}$ — векторы. Скалярным произведением этих векторов называется такое число:

$$\vec{a} \cdot \vec{b} = \sum_{k=1}^{3} a_k b_k.$$

Здесь через a_k и b_k ($k = 1,2,3$) обозначены компоненты векторов. Векторным произведением двух векторов и называется вектор $\vec{c} = [\vec{a} \times \vec{b}]$ с координатами $c_1 = a_2 b_3 - a_3 b_2$, $c_2 = a_3 b_1 - a_1 b_3$ и $c_3 = a_1 b_2 - a_2 b_1$.

При составлении программы векторы реализуются в виде массивов, состоящих из трех элементов. При вычислении скалярного произведения получаем число. При вычислении векторного произведения создается и заполняется еще один массив из трех элементов. Элементы c_k этого массива могут быть вычислены на основе элементов a_k и b_k исходных массивов по формуле

$$c_k = a_{k+1}b_{k+2} - a_{k+2}b_{k+1}.$$

Причем здесь выполняется циклическая перестановка индексов: если индекс выходит за верхнюю допустимую границу диапазона индексации, то индекс циклически перемещается в начало диапазона. Например, индексы компонентов вектора изменяются от 1 до 3, поэтому $a_4 = a_1$ и $a_5 = a_2$.

НА ЗАМЕТКУ

Поскольку индексы массива изменяются от `0` до `2`, то элементу с индексом `3` соответствует элемент с индексом `0`, а элементу с индексом `4` — элемент с индексом `1`.

В листинге 3.10 представлен код программы для вычисления скалярного и векторного произведений двух векторов.

Листинг 3.10. Произведение векторов

```
import java.util.Arrays;
class Demo{
   public static void main(String args[]){
      // Массивы для реализации векторов:
      double[] a={1,2,-1};
      double[] b={3,-1,2};
      double[] c=new double[3];
      // Переменная для записи скалярного произведения:
      double s=0;
      // Вычисление скалярного и векторного произведения:
      for(int k=0;k<3;k++){
         s+=a[k]*b[k];
         c[k]=a[(k+1)%3]*b[(k+2)%3]-a[(k+2)%3]*b[(k+1)%3];
      }
      // Отображение результата:
      System.out.println("a="+Arrays.toString(a));
      System.out.println("b="+Arrays.toString(b));
      System.out.println("a.b="+s);
      System.out.println("c=[a.b]="+Arrays.toString(c));
   }
}
```

В программе создается три массива, `a`, `b` и `c`, с элементами типа `double`. Для массивов `a` и `b` сразу задаются значения элементов, а элементы массива `c` рассчитываются в соответствии с правилом вычисления векторного произведения. Значение для скалярного произведения записывается в переменную `s` типа `double`. Основные

вычисления происходят в операторе цикла. Там индексная переменная k принимает значения от 0 до 2 включительно. В теле оператора цикла командой s+=a[k]*b[k] к переменной, определяющей скалярное произведение, добавляется произведение соответствующих элементов массивов a и b. Командой c[k]=a[(k+1)%3]*b[(k+2)%3]-a[(k+2)%3]*b[(k+1)%3] вычисляется элемент для массива, в который записывается результат векторного произведения.

ПОДРОБНОСТИ

Индексы элементов массивов вычисляются как остаток от деления на 3. Это позволяет реализовать процедуру циклической перестановки индексов (если значение индекса выходит за допустимые границы).

Вычисленные значения отображаются в области вывода. Результат выполнения программы имеет такой вид:

Результат выполнения программы (из листинга 3.9)

```
a=[1.0, 2.0, -1.0]
b=[3.0, -1.0, 2.0]
a.b=-1.0
c=[a.b]=[3.0, -5.0, -7.0]
```

Следует отметить, что подобные операции с векторами лучше все же реализовывать с помощью специальных классов или, по крайней мере, создать для этого несколько методов.

Числа Фибоначчи

Мы уже сталкивались с вычислением чисел Фибоначчи. Рассмотрим программу, в которой числами Фибоначчи заполняется массив (листинг 3.11).

Листинг 3.11. Числа Фибоначчи

```
class Demo{
   public static void main(String args[]){
      // Размер массива:
      int n=15;
      // Массив для чисел Фибоначчи:
      int[] Fib=new int[n];
      // Первые два числа последовательности:
      Fib[0]=1;
      Fib[1]=1;
      // Отображение первых двух значений:
      System.out.print(Fib[0]+" "+Fib[1]);
      // Вычисление последовательности
      // и отображение результата:
      for(int k=2;k<n;k++){
         // Вычисляется значение элемента массива:
```

```
         Fib[k]=Fib[k-1]+Fib[k-2];
         // Отображение значения элемента:
         System.out.print(" "+Fib[k]);
      }
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 3.11)

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

Обратим внимание на способ заполнения массива. Значения (единичные) для первых двух элементов задаются в явном виде. Затем в операторе цикла вычисляются прочие элементы массива. Значение очередного элемента вычисляется как сумма значений двух предыдущих элементов.

Работа с полиномами

Массивы могут быть полезны и при работе с выражениями полиномиального вида. Напомним, что полиномом степени n называется функция вида

$$P_n(x) = a_0x^0 + a_1x^1 + a_2x^2 + ... + a_nx^n.$$

Степенью полинома называется наибольшая степень аргумента x, входящая в полиномиальное выражение. Основная информация о полиноме заложена в коэффициентах a_k, $k = 0,1,...,n$. Фактически, для того чтобы вычислить значение полинома для аргумента x, нужно знать массив коэффициентов a_k.

При известном массиве коэффициентов полинома можно вычислить и производную от полинома. Производная от полинома также является полиномом и задается соотношением

$$P_n'(x) = a_1 + 2a_2x^1 + 3a_3x^2 + ... + na_nx^{n-1}.$$

Это — полином степени $n - 1$, а коэффициенты полинома-производной определяются на основе коэффициентов исходного полинома. Коэффициент полинома-производной равен $b_k = (k + 1)a_{k+1}$ для $k = 0,1,...,n - 1$ и $b_n = 0$. В листинге 3.12 представлен код, в котором по массиву коэффициентов полинома для заданного аргумента вычисляется значение полинома и значение производной от этого полинома в этой же точке.

Листинг 3.12. Полином и его производная

```
class Demo{
   public static void main(String args[]){
      // Коэффициенты полинома:
      double[] a=new double[]{1,-3,2,4,1,-1};
      // Массив коэффициентов для производной:
```

```
      double[] b=new double[a.length-1];
      // Аргумент и множитель:
      double x=2.0,q=1;
      // Значения полинома и производной:
      double P=0,Q=0;
      // Вычисление результата:
      for(int k=0;k<b.length;k++){
         // Полином:
         P+=a[k]*q;
         // Коэффициент производной:
         b[k]=(k+1)*a[k+1];
         // Производная:
         Q+=b[k]*q;
         // Изменение множителя:
         q*=x;
      }
      // Последнее слагаемое полинома:
      P+=a[a.length-1]*q;
      // Отображение результата:
      System.out.println("Полином P(x)="+P);
      System.out.println("Производная P'(x)="+Q);
   }
}
```

В программе командой `double[] a=new double[]{1,-3,2,4,1,-1}` создается и инициализируется массив с коэффициентами для полинома. Командой `double[] b=new double[a.length-1]` создается массив для записи коэффициентов полинома-производной. Размер этого массива на единицу меньше размера первого массива. Здесь учитывается, что степень полинома-производной на единицу меньше степени исходного (дифференцируемого) полинома. Размер первого полинома вычисляется инструкцией `a.length`.

Переменная `x` типа `double` содержит значение для аргумента полинома, а переменная `q` того же типа представляет собой степенной множитель, используемый в дальнейшем при вычислении значения полинома и значения производной. Начальное значение этой переменной равно `1`. Переменные `P` и `Q` типа `double` с нулевыми начальными значениями нужны для записи значения полинома и значения производной соответственно.

В операторе цикла переменная `k` пробегает значения от `0` до `b.length-1` включительно, что соответствует диапазону индексации элементов массива `b` с коэффициентами для полинома-производной. За каждую итерацию выполняется несколько команд. Первой командой `P+=a[k]*q` изменяется значение полинома: к текущему значению добавляется полиномиальный коэффициент `a[k]`, умноженный на аргумент в соответствующей степени. Последнее значение вычисляется и записывается в переменную `q`. Командой `b[k]=(k+1)*a[k+1]` вычисляется коэффициент для полинома-производной. Значение производной формируется командой `Q+=b[k]*q`, в которой использован вычисленный на предыдущем этапе коэффициент полинома-производной, а аргумент в нужной степени записан, как и ранее, в перемен-

ную q. Наконец, значение переменной q изменяется командой q*=x (на следующей итерации показатель степени аргумента увеличивается на единицу).

После завершения оператора цикла значение производной вычислено, а в полиноме не учтено еще одно слагаемое. Поэтому командой P+=a[a.length-1]*q оно прибавляется к текущему значению для полинома. Здесь используется последний элемент из массива коэффициентов полинома и переменная q, которая после выполнения оператора цикла содержит значение аргумента в нужной степени.

После вычисления значений для полинома и производной результат отображается в области вывода:

Результат выполнения программы (из листинга 3.12)

```
Полином P(x)=19.0
Производная P'(x)=5.0
```

Сортировка массива

Существует несколько алгоритмов сортировки массивов. Достаточно популярным и простым, хотя и не вполне оптимальным, является *метод пузырька*. Идея метода достаточно проста. Перебираются все элементы массива, причем каждый раз сравниваются два соседних элемента. Если элемент с меньшим индексом больше элемента с большим индексом, то элементы меняются местами (обмениваются значениями). После перебора всех элементов самый большой элемент оказывается последним. Если указанную процедуру проделать еще раз, то на правильном месте оказывается второй по величине элемент, и так далее. Это позволяет упорядочить элементы массива в порядке возрастания. Если нужно отсортировать массив в порядке убывания, то при переборе и сравнении массива элементы меняются местами (если элемент с меньшим индексом меньше элемента с большим индексом).

В листинге 3.13 приведен пример программы, в которой для целочисленного массива выполняется сортировка методом пузырька.

Листинг 3.13. Сортировка массива

```
import static java.lang.Math.random;
class Demo{
   public static void main(String args[]){
      // Размер массива и переменная:
      int n=15,s;
      // Создание массива:
      int[] nums=new int[n];
      System.out.println("Исходный массив:");
      // Заполнение массива
      // и отображение значений элементов:
      for(int k=0;k<nums.length;k++){
         // Значения элементов — случайные числа:
         nums[k]=(int)(5*n*random());
         System.out.print(nums[k]+" ");
```

```
      }
      // Сортировка массива:
      for(int m=1;m<nums.length;m++){
         for(int k=0;k<nums.length-m;k++){
            if(nums[k]>nums[k+1]){
               // Обмен значениями элементов:
               s=nums[k];
               nums[k]=nums[k+1];
               nums[k+1]=s;
            }
         }
      }
      // Отображение результата:
      System.out.println("\nМассив после сортировки:");
      for(int k=0;k<nums.length;k++){
         System.out.print(nums[k]+" ");
      }
      // Переход к новой строке:
      System.out.println();
   }
}
```

В программе объявляется целочисленный массив `nums`, с помощью оператора цикла элементы массива заполняются случайными целыми числами. Для генерирования случайных чисел использован статический метод `random()` из класса `Math`, который возвращает действительное случайное число в диапазоне от `0` до `1`. Это число умножается на `5` и на размер массива (переменная `n`), после чего с помощью инструкции `(int)` выполняется приведение к целочисленному типу (в действительном числе отбрасывается дробная часть). Как только значение элементу присвоено, оно отображается в области вывода (значения отображаются через пробел в одной строке).

Сортировка элементов массива выполняется с помощью вложенных операторов цикла. Индексная переменная `m` нумерует проходы — полные циклы перебора и сравнения двух соседних элементов. После каждого такого прохода по меньшей мере один элемент оказывается на правильном месте. При этом нужно учесть, что когда предпоследний элемент занимает свою позицию, то последний автоматически тоже оказывается в нужном месте. Поэтому количество проходов на единицу меньше количества элементов в массиве. Внутренняя индексная переменная `k` нумерует элементы массива. Она изменяется в пределах от `0` до `nums.length-m-1` включительно. Здесь мы учли, во-первых, что при фиксированном значении `k` сравниваются элементы с индексами `k` и `k+1`. Поэтому индекс последнего проверяемого элемента на единицу больше верхней границы диапазона изменения индекса `k`. Во-вторых, если какое-то количество проходов уже выполнено, то такое же количество последних элементов массива можно не проверять.

После того как сортировка массива выполнена, с помощью оператора цикла содержимое массива отображается в области вывода. Результат выполнения программы (с поправкой на то, что используется оператор цикла) может быть следующим:

Результат выполнения программы (из листинга 3.13)

```
Исходный массив:
73 51 72 30 73 74 44 33 59 47 8 25 47 70 4
Массив после сортировки:
4 8 25 30 33 44 47 47 51 59 70 72 73 73 74
```

Поскольку массив заполняется случайными числами, от запуска к запуску результаты (значения элементов массива) могут быть разными. Неизменным остается одно — после сортировки элементы массива располагаются в порядке возрастания.

Транспонирование квадратной матрицы

Транспонирование матрицы подразумевает взаимную замену строк и столбцов матрицы. Для простоты рассмотрим процедуру транспонирования квадратной матрицы, реализованной в виде двумерного массива. Результат транспонирования записывается в тот же массив.

Если элементами исходной квадратной матрицы A ранга n являются значения a_{ij}, где индексы $i,j = 1, 2, ..., n$, то транспонированная матрица A^T состоит из элементов $a_{ij}^T = a_{ji}$. Образно выражаясь, для того чтобы транспонировать квадратную матрицу, необходимо зеркально отобразить ее относительно главной диагонали. В листинге 3.14 приведен пример программы, в которой выполнена такая процедура.

Листинг 3.14. Транспонирование матрицы

```
import static java.lang.Math.random;
class Demo{
   public static void main(String args[]){
      // Ранг матрицы:
      int n=4;
      // Двумерный массив:
      int[][] A=new int[n][n];
      // Переменные:
      int i,j,tmp;
      System.out.println("Матрица до транспонирования:");
      // Заполнение матрицы случайными числами:
      for(i=0;i<n;i++){
         for(j=0;j<n;j++){
            A[i][j]=(int)(10*random());
            System.out.print(A[i][j]+"  ");
         }
         // Переход к новой строке:
         System.out.println();
      }
      // Транспонирование матрицы:
      for(i=0;i<n;i++){
         for(j=i+1;j<n;j++){
            tmp=A[i][j];
            A[i][j]=A[j][i];
```

```
            A[j][i]=tmp;
         }
      }
      // Отображение результата:
      System.out.println(
         "Матрица после транспонирования:"
      );
      for(i=0;i<n;i++){
         for(j=0;j<n;j++){
            System.out.print(A[i][j]+"  ");
         }
         // Переход к новой строке:
         System.out.println();
      }
   }
}
```

Результат выполнения программы может иметь следующий вид:

Результат выполнения программы (из листинга 3.14)

```
Матрица до транспонирования:
8 1 5 7
9 8 2 8
4 9 6 7
3 7 6 9
Матрица после транспонирования:
8 9 4 3
1 8 9 7
5 2 6 6
7 8 7 9
```

В программе командой `int[][] A=new int[n][n]` создается двумерный массив `A`, который мы отождествляем с исходной матрицей. Заполнение массива и отображение значений элементов массива выполняется с помощью вложенных операторов цикла. Случайное число, которое в качестве значения присваивается элементу массива, вычисляется командой `(int)(10*random())`. Результатом является целое число в диапазоне от `0` до `9` включительно. Значение элемента отображается командой `System.out.print(A[i][j]+" ")`. По завершении внутреннего оператора цикла выполняется переход к новой строке. Поэтому элементы двумерного массива выводятся построчно.

В следующем блоке из вложенных операторов цикла выполняется транспонирование матрицы. Однако в данном случае перебираются не все элементы, а только те, что лежат выше главной диагонали. Поэтому первый индекс `i`, с помощью которого перебираются строки матрицы, изменяется в пределах от `0` до `n-1`, а второй индекс, связанный с нумерацией столбцов, изменяется от `i+1` до `n-1`. Тело внутреннего оператора цикла состоит из трех команд, с помощью которых меняются местами элементы, расположенные симметрично относительно главной диагонали (эти элементы отличаются порядком индексов).

Третий блок из вложенных операторов цикла служит для отображения значений элементов матрицы (массива) после транспонирования.

Произведение квадратных матриц

Если A и B — квадратные матрицы ранга n с элементами a_{ij} и b_{ij} соответственно (индексы $i,j = 1, 2, ..., n$), то произведением этих матриц является матрица $C = AB$ с элементами:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + ... + a_{in-1}b_{n-1j} + a_{in}b_{nj}.$$

В листинге 3.15 представлена программа, в которой вычисляется произведение квадратных матриц.

Листинг 3.15. Произведение квадратных матриц

```
import static java.lang.Math.random;
class Demo{
   public static void main(String args[]){
      // Ранг квадратных матриц:
      int n=4;
      // Массивы для реализации матриц:
      int[][] A=new int[n][n];
      int[][] B=new int[n][n];
      int[][] C=new int[n][n];
      // Индексные переменные:
      int i,j,k;
      // Заполнение матрицы A:
      System.out.println("Матрица A:");
      for(i=0;i<n;i++){
         for(j=0;j<n;j++){
            A[i][j]=(int)(20*random()-9);
            System.out.printf("%4d",A[i][j]);
         }
         System.out.println();
      }
      // Заполнение матрицы B:
      System.out.println("Матрица B:");
      for(i=0;i<n;i++){
         for(j=0;j<n;j++){
            B[i][j]=(int)(20*random()-9);
            System.out.printf("%4d",B[i][j]);
         }
         System.out.println();
      }
      // Вычисление произведения матриц:
      System.out.println("Матрица C=AB:");
      for(i=0;i<n;i++){
         for(j=0;j<n;j++){
            C[i][j]=0;
            for(k=0;k<n;k++){
```

```
                C[i][j]+=A[i][k]*B[k][j];
              }
              System.out.printf("%4d",C[i][j]);
           }
           System.out.println();
        }
     }
}
```

Результат выполнения программы может иметь следующий вид:

Результат выполнения программы (из листинга 3.15)

```
Матрица A:
   6   0  -4   0
  -5   4  -3   5
   9  -6   5  10
  -3  -7   9  -2
Матрица B:
  -8   4   5   9
   4   7  -5   7
  -7   9  -3   4
   8  -7  -8  -2
Матрица C=AB:
 -20 -12  42  38
 117 -54 -76 -39
 -51 -31 -20  39
 -83  34   9 -36
```

В программе создается три двумерных массива. Первые два заполняются случайными числами. Третий массив заполняется в блоке из трех вложенных операторов цикла. Для отображения значений элементов массивов использовался метод `printf()`. Первым аргументом методу передается текстовая строка с инструкцией форматирования `"%4d"`. Метод отображает эту строку, но вместо инструкции форматирования `%4d` вставляется значение, указанное вторым аргументом метода.

ПОДРОБНОСТИ

В инструкции `%4d` символ `%` является индикатором инструкции форматирования. Буква `d` означает, что отображается целочисленное значение. Число `4` определяет минимальное количество позиций, выделяемых для отображения целого числа.

Задача перколяции

Проблема перколяции в том или ином виде имеет отношение к решению целого ряда прикладных задач. У нее есть несколько формулировок. Рассмотрим один из наиболее простых вариантов. Имеется сетка из полых трубочек, которая может

пропускать жидкость. Случайным образом выбирают какое-то количество узлов сетки и перекрывают их. Необходимо определить, поступит ли жидкость, поданная с одного края сетки, на другой ее край.

Воспользуемся при составлении программы следующим подходом. Создадим в программе квадратную матрицу (двумерный массив), соответствующую сетке, на которой изучается перколяция. Элементы матрицы соответствуют узлам сетки. На начальном этапе элементы могут принимать два значения: `0` — если узел может пропускать жидкость и `1` в противном случае. Перекрытые узлы выбираются случайным образом. В частности, генерируется случайное число в диапазоне от `0` до `1`. Если это число больше определенного значения p (вероятность того, что случайно выбранный узел пропускает жидкость), то элемент матрицы получает значение `1`. В противном случае значением элемента является `0`.

После заполнения перколяционной матрицы начинается заливка жидкости. Узел, в который попала жидкость, мы будем отмечать в перколяционной матрице значением `2`. На начальном этапе перебираются элементы первого столбца матрицы; если значение элемента равняется `0`, оно меняется на значение `2`.

В основной части программы выполняется последовательный перебор всех элементов перколяционной матрицы. Если значение элемента равно `2` (в этом узле уже есть жидкость), то соседним элементам, если их текущее значение равно `0` (узел может пропускать жидкость), присваивается значение `2`. После перебора всех элементов процедура повторяется до тех пор, пока за весь перебор элементов перколяционной матрицы ни один из элементов не изменит свое состояние.

Для проверки того, достигла ли жидкость конечной части сетки, просматривается последний столбик перколяционной матрицы. Если значение хотя бы одного элемента в этом столбце равно `2`, имеет место протекание жидкости.

Вся описанная процедура позволяет определить, возможно ли протекание жидкости для данной конфигурации выведенных из строя узлов. В общем случае желательно иметь более надежные и объективные показатели. Обычно изучают зависимость вероятности того, что сетка пропускает жидкость, от вероятности того, что выбранный случайным образом узел сетки пропускает жидкость (упоминавшийся ранее параметр p). Для вычисления такой зависимости необходимо провести статистическое усреднение: при заданном фиксированном значении p проделать описанную процедуру несколько раз (чем больше — тем лучше) и вычислить вероятность пропускания сеткой жидкости как относительное значение случаев, когда сетка пропускала жидкость, к общему количеству случаев исследования сетки на предмет пропускания жидкости.

В программе, представленной в листинге 3.16 для нескольких значений параметра p (вероятность пропускания жидкости отдельным случайно выбранным узлом сетки), вычисляется вероятность пропускания жидкости всей сеткой. Полученные значения заносятся в массив. В результате выполнения программы данные из этого массива в виде импровизированной таблицы отображаются в области вывода.

Листинг 3.16. Задача о перколяции

```
class Demo{
   public static void main(String args[]){
      // Количество запусков для усреднения:
      int N=100;
      // Количество точек вычисления вероятности:
      int M=5;
      // Размер сетки:
      int n=200;
      // Переменная-счетчик:
      int count;
      // Начальная вероятность и ее приращение:
      double q=0.57,dq=0.01;
      // Перколяционная матрица:
      int[][] A=new int[n][n];
      // Массив со значениями вероятностей:
      double[][] P=new double[2][M+1];
      // Индексные переменные:
      int i,j,k,m;
      // Заполнение массива вероятностей:
      for(m=0;m<=M;m++) P[0][m]=q+dq*m;
      // Вычисление вероятностей протекания:
      for(m=0;m<=M;m++){
         // Начальное значение вероятности протекания:
         P[1][m]=0;
         for(k=1;k<=N;k++){
            for(i=0;i<n;i++){
               for(j=0;j<n;j++){
                  // Заполнение перколяционной матрицы:
                  if(Math.random()>P[0][m]) A[i][j]=1;
                  else A[i][j]=0;
               }
            }
            // "Заливка" жидкости в сетку
            // (заполнение первого столбца):
            for(i=0;i<n;i++){
               if(A[i][0]==0) A[i][0]=2;
            }
            // Определение протекания:
            do{
               // Начальное состояние счетчика:
               count=0;
               // Изменение состояния узлов сетки:
               for(i=0;i<n;i++){
                  for(j=0;j<n;j++){
                     if(A[i][j]==2){
                        if(i>0&&A[i-1][j]==0){
                           A[i-1][j]=2;
                           count++;
                        }
                        if(i<n-1&&A[i+1][j]==0){
                           A[i+1][j]=2;
                           count++;
```

```
                        }
                        if(j<n-1&&A[i][j+1]==0){
                           A[i][j+1]=2;
                           count++;
                        }
                        if(j>0&&A[i][j-1]==0){
                           A[i][j-1]=2;
                           count++;
                        }
                     }
                  }
               }
            }while(count>0);
            // Проверка последнего столбца
            // перколяционной матрицы:
            for(i=0;i<n;i++){
               if(A[i][n-1]==2){
                  P[1][m]+=(double)1/N;
                  break;
               }
            }
         }
      }
      // Отображение результата:
      System.out.print("Протекание узла \t");
      for(m=0;m<=M;m++){
         System.out.printf("%7.2f",
            Math.round(P[0][m]*100)/100.0
         );
      }
      System.out.print("\nПротекание сетки\t");
      for(m=0;m<=M;m++){
         System.out.printf("%7.2f",
            Math.round(P[1][m]*100)/100.0
         );
      }
      System.out.println();
   }
}
```

Переменные, использованные в программе, описаны в табл. 3.1.

Таблица 3.1. Переменные в задаче о перколяции

Переменная	Описание
N	Целочисленная переменная, определяющая количество измерений, на основе которых вычисляется оценка для вероятности пропускания сетки при данном значении вероятности пропускания узла. При увеличении значения этой переменной точность оценок повышается, равно как и время расчетов

Таблица 3.1 (*окончание*)

Переменная	Описание
`M`	Целочисленная переменная, определяющая количество значений (их `M+1`) вероятности пропускания узлов, для которых вычисляется вероятность пропускания сетки
`n`	Целочисленная переменная, определяющая размер перколяционной сетки и, соответственно, размер матрицы `A`, в которую записывается состояние узлов сетки
`count`	Целочисленная переменная-счетчик. Используется для подсчета количества элементов матрицы `A`, которым при определении пропускания сетки было присвоено значение `2`. При определении протекания сетки запускается оператор цикла, значение переменной `count` обнуляется, после чего проверяются все элементы матрицы `A`. При изменении значения элемента матрицы значение переменной `count` увеличивается на единицу. После перебора всех элементов значение `count` снова обнуляется, перебираются все элементы матрицы `A`, и так далее. Процесс продолжается до тех пор, пока после перебора всех элементов матрицы значение переменой `count` не изменится (останется нулевым)
`q`	Первое из набора значений для вероятности протекания узла. В программе вычисляется вероятность протекания сетки для нескольких значений (точнее, их `M+1`) вероятности протекания выбранного случайно узла. Первое из этих значений равно `q`, последнее — `q+M*dq`
`dq`	Переменная, которая определяет интервал дискретности для вероятности протекания узла
`A`	Целочисленный двумерный массив, соответствующий перколяционной сетке. Элементы массива (матрицы) `A` могут принимать значения `0` (узел пропускает жидкость) и `1` (узел не пропускает жидкость). В процессе выполнения программы элемент, имеющий значение `0`, может получить значение `2`. Это означает, что узел заполнен жидкостью
`P`	Двумерный массив размером `2` на `M+1` с элементами типа `double`. Строка `P[0]` содержит значения для вероятностей пропускания узлов перколяционной сетки. Строка `P[1]` содержит вычисленные вероятности для пропускания сетки, соответствующие значениям из строки `P[0]`
`i`	Целочисленная индексная переменная. Используется в операторе цикла
`j`	Целочисленная индексная переменная. Используется в операторе цикла
`k`	Целочисленная индексная переменная. Используется в операторе цикла
`m`	Целочисленная индексная переменная. Используется в операторе цикла

После объявления всех переменных и массивов в программе запускается оператор цикла, в теле которого командой `P[0][m]=q+dq*m` (индексная переменная `m` получает

значения от 0 до M включительно) заполняется первая строка P[0] массива P значениями вероятности протекания узлов, для которых затем вычисляются значения вероятностей протекания сетки. Значения вероятностей протекания сетки вычисляются в следующем операторе цикла (переменная m изменяется в тех же пределах) и после вычисления записываются в строку P[1]. Для начала эти элементы командой P[1][m]=0 для надежности обнуляются (хотя этого можно и не делать — при создании массива его элементы уже получили нулевые значения). Затем все в том же операторе цикла вызывается еще один оператор цикла (индексная переменная k изменяется от 1 до N включительно), которым соответствующее количество раз выполняется проверка протекания перколяционной сетки при фиксированном значении вероятности протекания узлов. На основании результатов работы этого оператора цикла определяется оценка для вероятности протекания сетки. В начале цикла случайным образом, в соответствии с текущим значением вероятности протекания узлов сетки, элементы массива A заполняются нулями и единицами. Для этого используются вложенные операторы цикла. Затем с помощью еще одного оператора цикла элементам массива A со значением 0 в первом столбце присваивается значение 2 — это означает, что в соответствующие узлы поступила жидкость. На следующем этапе начинается реализация процесса заполнения сетки жидкостью. В частности, запускается оператор цикла do-while. Там вначале переменной-счетчику count присваивается нулевое значение. Проверяемым условием является count>0. В операторе цикла do-while перебираются все элементы массива A. Если значение элемента равно 2, то соседним элементам, имеющим нулевые значения, также присваивается значение 2. Соседние элементы — это те, у которых один и только один индекс отличается на единицу от индексов текущего элемента. При этом нужно учесть, что текущий узел может находиться не в центре сетки, а на ее границе. Поэтому для соответствующего элемента операция смещения индекса на одну позицию может привести к выходу за пределы массива A. В силу этого обстоятельства проверяемое условие состоит не только в том, что значение элемента, расположенного рядом с текущим, равно 0, но и в том, что текущий элемент не является граничным и операция обращения к соседнему элементу корректна.

ПОДРОБНОСТИ

При проверке условий для оператора *логическое и* используется сокращенная версия &&. В таком случае вычисляется первый операнд, и если он равен false, то второй операнд не вычисляется. Благодаря этому код выполняется без ошибки.

Если хотя бы для одного элемента массива A значение изменено на 2, то значение переменной count увеличивается на единицу. Таким образом, цикл do-while выполняется до тех пор, пока при переборе всех элементов массива A ни одно значение не будет изменено.

После этого необходимо проверить результат — есть или нет протекание сетки. Наличие протекания означает, что жидкость дошла до правого края сетки. А раз так, то последний столбец массива A содержит хотя бы одно значение 2. Поиск такого

значения осуществляется в еще одном операторе цикла. Если значение 2 найдено, вероятность P[1][m] увеличивается на величину 1/N, после чего работа оператора цикла заканчивается (инструкцией break). На этом основная, расчетная часть программы заканчивается. Далее результаты с помощью двух операторов цикла отображаются в области вывода.

ПОДРОБНОСТИ

При отображении результатов использована инструкция перехода к новой строке \n и инструкция выполнения табуляции \t. Также был задействован метод printf(). Первым аргументом методу передавалась текстовая строка с инструкцией форматирования %7.2f, которая означает, что в соответствующем месте будет отображаться действительное число, под него отводится не менее 7 позиций, причем после десятичной точки/запятой выделяется не менее 2 позиций. В зависимости от настроек системы и среды разработки при выводе может отображаться десятичная точка или запятая.

При вызове метода printf() отображается текстовая строка (первый аргумент), в которую вместо инструкции форматирования вставляется значение второго аргумента метода.

В частности, результат выполнения программы может быть таким:

Результат выполнения программы (из листинга 3.16)

```
Протекание узла    0,57   0,58   0,59   0,60   0,61   0,62
Протекание сетки   0,05   0,08   0,38   0,74   0,94   0,99
```

С учетом того, что в программе используется процедура генерирования случайных чисел, от запуска к запуску результат может изменяться. Однако если количество запусков, на основании которых выполняется усреднение, достаточно большое, то результаты должны изменяться незначительно.

Резюме

> Это лирическое отступление пора бы заканчивать.
>
> *из к/ф «Гараж»*

- Массивом называется совокупность переменных одного типа, к которым можно обращаться по общему имени и индексу (или индексам).
- Создание массива можно условно разделить на два этапа. Во-первых, объявляется переменная массива, которой впоследствии в качестве значения присваивается ссылка на массив. Во-вторых, с помощью оператора new для массива

выделяется место (создается массив). Результат (ссылка на массив) обычно записывается в переменную массива.

- При объявлении переменной массива после идентификатора для типа элементов указываются пустые квадратные скобки. Количество пар пустых скобок соответствует размерности массива. При создании массива после оператора `new` указывается тип элементов массива и в квадратных скобках — размер массива по каждой размерности.
- При создании массива его элементы можно инициализировать (по умолчанию элементы созданного массива обнуляются). Список значений элементов массива (список инициализации) указывается в фигурных скобках через запятую. Этот список может размещаться в команде объявления переменной массива после имени переменной (через оператор присваивания). Можно также указать список значений сразу за квадратными скобками после идентификатора типа в инструкции создания массива (с оператором `new`). В этом случае в квадратных скобках размер массива не указывается (он определяется автоматически по количеству значений в списке инициализации).
- Обращение к элементу массива выполняется в следующем формате: имя массива и в квадратных скобках индекс элемента. Индекс по каждой из размерностей указывается в отдельных квадратных скобках. Индексация элементов массива всегда начинается с нуля.
- В Java выполняется проверка на предмет выхода индекса элемента массива за допустимые границы. Длину массива (количество элементов) можно получить с помощью свойства `length` (указывается через точку после имени массива).
- Если одной переменной массива значением присвоить другую переменную массива, то массивы копироваться не будут. Вместо этого происходит копирование ссылок на массив. Для создания копии массива необходимо создать массив такого же размера и один за другим скопировать значения элементов исходного массива.
- Если аргументом метода `print()` или `println()` указано имя символьного массива (массива, элементом которого являются символы), отображается все содержимое символьного массива. Текстовые массивы можно создавать, объявив переменную для текстового массива и присвоив ей значением список инициализации с текстовыми литералами.

4 Классы и объекты

Ну, Ватсон, это уж такая простая дедукция!
Могли бы сами догадаться!

из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Java является особым языком по нескольким причинам. Да, для успешной работы необходимо как минимум неплохо знать синтаксис языка, но все же не это самое главное. Успешное использование Java на практике невозможно без глубокого понимания принципов *объектно-ориентированного программирования* (сокращенно ООП). Основные идеи, заложенные в ООП, раскрываются далее на простых примерах из повседневной жизни.

Знакомство с ООП

Все правильно и очень просто — после того, как вы мне объяснили.

из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Java является полностью объектно-ориентированным языком. Это означает, что программа, написанная на Java, должна строго соответствовать парадигме объектно-ориентированного программирования. Следует понимать, что принципы ООП не просто определяют структуру программы. Это некий фундаментальный подход, если угодно, философия программирования, на сути которой имеет смысл остановиться подробнее.

НА ЗАМЕТКУ

Особенности ООП во многом объясняются причинами, приведшими к появлению ООП как такового. Не вдаваясь в детали, отметим лишь, что в свое время в развитии программных технологий наступил момент, когда сложность прикладных

программ достигла уровня, критического для понимания программистами. Традиционный подход, который получил название *процедурного программирования*, оказался неприменим (или малоприменим) для составления больших и сложных программ.

Любая программа связана с данными и методами для обработки этих данных. Если данных и методов много, в рамках одной программы сложно разумно структурировать код. Такие коды программисты называют спагетти-кодами, поскольку отдельные ветви алгоритма переплетаются, образуя запутанный клубок, в котором невозможно разобраться. Проблема в какой-то момент стала принципиальной и острой — необходимо было искать выход. Выход нашли в рамках нового подхода, который получил название *объектно-ориентированного программирования*.

Решение проблемы упорядочивания кода базируется на объединении данных, с одной стороны, и методов для обработки этих данных, с другой стороны, в одно целое. Это «одно целое» в ООП называется *объектом*. Такой, на первый взгляд, искусственный прием позволяет четко разграничить область применимости методов. Вся программа при этом имеет блочную структуру, что существенно упрощает анализ кода.

НА ЗАМЕТКУ

Представим себе автомастерскую. Если это небольшое предприятие, то там, как правило, работает лишь несколько специалистов. У них есть начальник, который лично знает каждого сотрудника и каждому раздает задания. Это аналог обычной, не объектно-ориентированной программы. Все это хорошо работает, если размеры предприятия не очень большие. Если предприятие расширяется, то такая схема организации производственного процесса начнет давать сбои. Во-первых, одному начальнику не справиться — если подчиненных слишком много, организовать их работу и уследить за ее качеством сложно. Во-вторых, наверняка начнутся разнообразные проблемы — кто-то не сделал вовремя свою работу, кто-то не заказал запчасти, кто-то забрал чьи-то инструменты (список потенциальных проблем можно расширить). Как с этим бороться? Реорганизовать производственный процесс. Например, разнести работу по разным цехам, где будут выполняться работы определенного вида. В каждом цехе поставить своего начальника, для каждого цеха организовать свою службу обеспечения и отдельные склады. Возможно, в реальности все не так — но в данном случае важен принцип. А принцип заключается в том, чтобы максимально структурировать все производство, разбить его на самодостаточные блоки, которыми можно управлять и которые можно контролировать. Это аналог объектно-ориентированной программы.

Каждый объект определяется общим шаблоном, который называется *классом*. В рамках класса задается общая структура, на основе которой затем создаются объекты. Данные, относящиеся к классу, называются *полями* класса, а код для их обработки — *методами* класса.

НА ЗАМЕТКУ

Если мы, например, строим дом, то дом строится на основе проекта. Проект дома является аналогом класса. Сам дом является аналогом объекта. Если на основе одного проекта построено несколько домов, то эти дома будут однотипными. Аналогично, однотипными будут объекты, созданные на основе одного класса. Если мы берем разные проекты, то на их основе получаем разные дома. Если объекты создаются на основе разных классов, то это не просто разные объекты — они еще и разного типа. Класс отличается от объекта так же, как проект дома отличается от собственно дома.

В программе на основе того или иного класса создается экземпляр класса (объект), в котором указываются конкретные значения полей и выполняются необходимые действия над ними. Различие между классом и объектом поясним на простом примере, не имеющем никакого отношения к программированию. Поговорим о домашних животных, таких как коты и собаки. Проводя аналогию с программированием, можем определить класс *Кот* и класс *Собака*. Определение класса производится через указание полей (данных) и методов класса. Для класса *Кот* в качестве полей укажем *имя* (кличку кота) и *окрас* (цвет). Для класса *Собака* задаем поля *имя* (кличка собаки), *окрас* и *породу*. Помимо полей, определим методы для этих классов. По большому счету, метод — это то, что может делать объект соответствующего класса (или что можно делать с объектом). Коты будут мяукать и ловить мышей, а собаки — лаять и вилять хвостом. Отсюда методами класса *Кот* являются *мяукать* и *ловить мышей*, а класса *Собака* — *лаять* и *вилять хвостом*. Мы определили шаблоны, на основании которых впоследствии будут создаваться экземпляры классов или объекты. Разница между классом и объектом такая же, как между абстрактным понятием и реальным объектом. При создании объекта класса задаем конкретные значения для полей. Когда мы говорим о *собаке* вообще, как о понятии, мы имеем в виду домашнее животное, у которого есть имя, окрас, порода и которое умеет лаять и вилять хвостом. Точно так же понятие *кот* означает, что он мяукает и ловит мышей, к нему можно обратиться по имени и шубка у него может быть какого-то цвета. Это — абстрактные понятия, которые соответствуют классу. А вот если речь идет о конкретном Шарике или Мурзике, то это уже объекты, экземпляры класса.

Представим, что во дворе живут три собаки и два кота: Шарик (дворняжка коричневого окраса), Джек (рыжий спаниель), Ричард (черная немецкая овчарка), Мурзик (белый и пушистый кот) и Барсик (черный кот с белой манишкой). Каждый из пяти этих друзей представляет собой объект. В то же время они относятся к двум классам: Шарик, Джек и Ричард являются объектами класса *Собака*, а Мурзик и Барсик — объектами класса *Кот*. Каждый объект в пределах класса характеризуется одинаковым набором полей и методов. Одновременно с этим каждый объект уникален. Хотя Шарик, Джек и Ричард являются объектами одного класса, они уникальны, поскольку у них разные имена, породы и окрасы. Лают и виляют хвостом они тоже по-разному. Но даже если бы мы смогли клонировать, например, Шарика и назвать пса тем же именем, у нас, несмотря на полную тождественность обоих Шариков, было бы два объекта класса *Собака*. Каждый из них уникален,

причем не в силу каких-то физических различий, а по причине того, что один пес существует независимо от другого.

НА ЗАМЕТКУ

Когда мы говорим о полях и методах (обычных, не статических), то делается это в привязке к объекту. Например, если мы хотим, чтобы кот поймал мышь, то нужно указать, какой именно кот. Абстрактный кот ловить мышей не будет (разве что в наших фантазиях). Реальное действие может производить только реальный объект. Опять же, если нас интересует окрас собаки, то нужно указать, какой конкретно собаки. Аналогично, чтобы работать с полем, необходимо указать, к какому объекту это поле относится. Нет объекта — нет поля.

Объектно-ориентированный подход основан на нескольких достаточно простых и прагматичных принципах: *инкапсуляции*, *полиморфизма* и *наследования*.

Инкапсуляция позволяет объединить данные и код для обработки этих данных (методы) в одно целое. Фактически речь идет об объектах. Объект является именно той конструкцией, через которую реализуется механизм инкапсуляции. Забегая вперед, отметим, что данные и код внутри объекта могут быть открытыми, доступными вне объекта, и закрытыми. В последнем случае доступ к данным и коду может осуществляться только в рамках объекта.

С точки зрения указанного подхода класс является базовой единицей инкапсуляции. Класс задает формат объекта, определяя тем самым новый тип данных в широком смысле этого термина, включая и методы. Методы и поля, описанные в классе, называются *членами класса*.

Полиморфизм позволяет использовать один и тот же интерфейс или шаблон для выполнения различных действий. Здесь действует принцип «один интерфейс — много методов». Благодаря полиморфизму программы становятся менее сложными, так как для выполнения однотипных действий используют однотипные подходы.

НА ЗАМЕТКУ

Принцип полиморфизма можно проиллюстрировать на примере с вождением автомобиля. Например, если у автомобиля механическая коробка передач, то принцип переключения передач достаточно универсален, хотя в разных машинах коробка передач может быть устроена по-разному. Водителю всю эту внутреннюю механику знать не обязательно. Ему достаточно знать общие правила работы с коробкой передач. Даже если поменять автомобиль, правила останутся теми же.

Или педали в автомобиле: для большинства автомобилей (с механической коробкой передач) педалей три, причем идут они в строго определенном порядке (сцепление, тормоз, газ). Делается это для того, чтобы облегчить жизнь конечному потребителю (водителю) и ему не нужно было переучиваться при смене автомобиля. При этом тормозная система автомобиля или система подачи топлива является индивидуальной — для каждой машины своя. Но за счет «автомобильного полиморфизма» все эти особенности спрятаны и не создают проблем.

С полиморфизмом мы столкнемся при изучении *перегрузки* и *переопределения* методов.

Наследование — это механизм, с помощью которого один класс можно создать на основе другого, уже существующего класса. Например, если мы решим создать новый класс *Породистая собака*, который от класса *Собака* отличается наличием поля *награды на выставках*, то в общем случае придется заново создавать класс, описывая в явном виде все его поля и методы. В рамках ООП с помощью механизма наследования можно создать новый класс *Породистая собака* на основе уже существующего класса *Собака*, добавив в описание класса только новые свойства — старые наследуются автоматически.

НА ЗАМЕТКУ

Наследование — удобный и полезный механизм, который очень часто используется на практике.

Применение концепции ООП существенно расширило возможности в составлении сложных программных кодов. Основные преимущества ООП перечислены ниже.

- Благодаря механизму наследования можно многократно использовать созданный ранее код. Это позволяет экономить время и усилия при создании нового кода.
- Как правило, объектно-ориентированные программы хорошо структурированы, что повышает их читабельность. Работать с таким кодом удобно и приятно.
- Объектно-ориентированные программы легко редактировать и тестировать, поскольку работа может выполняться с отдельными блоками программы.
- Объектно-ориентированные программы в случае необходимости легко дорабатываются и расширяются. Данная особенность крайне важна при создании больших проектов.

Вместе с тем следует четко понимать, что концепция ООП эффективна лишь в том случае, когда речь идет о больших и сложных программах. Для создания простых программ лучше использовать простые приемы.

НА ЗАМЕТКУ

У концепции ООП есть не только преимущества, но и недостатки. И конечно, у ООП есть не только приверженцы, но и заядлые критики. Истина, скорее всего, где-то посредине. Вместе с тем серьезной альтернативы ООП на сегодня нет, и многие популярные и перспективные языки программирования реализуют этот подход.

Создание классов и объектов

Ну кто так строит?! Кто так строит?!

из к/ф «Чародеи»

Теперь перейдем к прикладной стороне проблемы: рассмотрим способы создания классов и объектов в Java.

Описание класса начинается с ключевого слова `class`. После этого следует имя класса и, в фигурных скобках, — тело класса. Тело класса состоит из описания членов класса: *полей* и *методов*. Таким образом, синтаксис описания класса имеет следующий вид:

```
class имя_класса{
   // Тело класса
}
```

Поля класса — это, если смотреть в корень, переменные. Что касается методов, то это — специально выделенные блоки команд, описанные в теле класса.

НА ЗАМЕТКУ

Концепция методов берет свое начало от функций и процедур, используемых, в том или ином виде, практически во всех языках программирования. Речь идет о блоке команд. Этот блок описан один раз, но его можно выполнять много раз в разных местах программы. Для этого у блока команд есть имя, и каждый раз, когда нужно выполнить команды из блока, просто указывается имя этого блока. Метод — это блок, который описывается в классе. У блока есть имя, по которому мы вызываем метод. Также у метода могут быть аргументы, и метод может возвращать результат.

Описание метода состоит из сигнатуры и тела метода. Сигнатура метода, в свою очередь, состоит из ключевого слова, которое обозначает тип возвращаемого методом результата, имени метода и списка аргументов в круглых скобках после имени метода. Аргументы разделяются запятыми, для каждого аргумента перед формальным именем аргумента указывается его тип.

Тело метода заключается в фигурные скобки и содержит код, определяющий функциональность метода. Методы могут возвращать значения простых (базовых) или ссылочных типов (объекты). Если метод не возвращает результат, в качестве идентификатора типа указывается ключевое слово `void`. Синтаксис объявления метода имеет такой вид:

```
тип_результата имя_метода(аргументы){
   // Тело метода
}
```

В теле метода описываются команды, которые будут выполняться при вызове метода. Если метод возвращает значение, то в теле метода значение, возвращаемое методом, указывается после инструкции `return`.

ПОДРОБНОСТИ

Вообще, выполнение инструкции `return` приводит к завершению работы метода (в котором эта инструкция выполняется). Если после инструкции `return` указано некоторое выражение, то значение этого выражения возвращается результатом метода. Тип возвращаемого значения должен совпадать с типом, указанным в сигнатуре метода.

Если метод возвращает значение, то инструкция вызова метода отождествляется с этим значением. Проще говоря, такую инструкцию можно указывать в выражениях, и при вычислении значения соответствующего выражения вместо инструкции вызова метода будет подставляться возвращаемое методом значение.

Если метод не возвращает значение (в таком случае идентификатором типа результата указывается ключевое слово `void`), то при вызове метода просто выполняются команды в теле метода. Когда выполняется последняя команда, работа метода завершается.

Отметим также, что сигнатура метода может содержать и другие ключевые слова (например, спецификатор уровня доступа), но о них речь будет идти позже.

Все программы, которые рассматривались до этого, состояли из одного класса — главного класса программы. В этом классе описывался всего один метод — главный метод программы `main()`. Никаких дополнительных полей и/или методов в главном классе мы не использовали (хотя это допустимо, и нередко так и поступают). Теперь кроме главного класса мы будем использовать и другие классы. Проще говоря, в наших программах будет несколько классов. Но мало описать класс в программе. На его основе предстоит создавать объекты. Как же они создаются? В принципе, ничего сложного. Объект создается в два этапа, которые обычно объединяют. На первом этапе объявляется *объектная переменная* — переменная, которая может ссылаться на объект. Второй этап подразумевает собственно создание объекта (выделение в памяти места под объект). Ссылка на созданный объект записывается в объектную переменную.

НА ЗАМЕТКУ

С технической точки зрения объектная переменная содержит в качестве значения адрес объекта. Эта ситуация очень похожа на способ, которым в Java реализуются массивы.

Синтаксис объявления объектной переменной мало отличается от объявления переменной базового типа: разница лишь в том, что в качестве типа переменной указывается имя класса, для которого планируется создавать объект. Создание

объекта (выделение памяти под объект) выполняется с помощью оператора new, после которого указывается имя класса с пустыми круглыми скобками.

ПОДРОБНОСТИ

На самом деле в выражении на основе инструкции new вызывается *конструктор* класса, а в круглых скобках в общем случае передаются аргументы для конструктора. Имя конструктора совпадает с именем класса, а пустые круглые скобки означают, что аргументы конструктору не передаются. Но пока что нас эти подробности не интересуют — обратимся к их рассмотрению немного позже.

Синтаксис команд для создания объекта имеет вид:

```
Класс переменная;       // Объектная переменная
переменная=new Класс(); // Создание объекта
```

Эти две команды можно объединить:

```
Класс переменная=new Класс();
```

Теперь мы теоретически уже знаем, как описать класс и как на его основе создать объект. Настало время переходить к действиям. Например, так выглядит объявление класса, содержащего два поля и два метода:

```
class MyClass{
   // Поля:
   String name;
   int number;
   // Методы:
   void set(String txt,int num){
      name=txt;
      number=num;
   }
   void show(){
      System.out.println(name+": "+number);
   }
}
```

Класс называется MyClass и у него есть поля name (типа String) и number (типа int), а также методы set() и show(). Методы не возвращают результат, поэтому в сигнатуре методов в качестве типа возвращаемого результата указано ключевое слово void. У метода set() два аргумента: один типа String и второй типа int. Первый аргумент присваивается в качестве значения полю name, второй определяет значение поля number. У метода show() нет аргументов. При вызове метода в окне вывода отображается сообщение, содержащее значения полей объекта.

НА ЗАМЕТКУ

Если в методе выполняется обращение к полям, то имеются в виду поля того объекта, из которого будет вызываться метод.

Важно понимать, что описание класса к созданию объектов не приводит. Другими словами, описывающий класс код — это всего лишь шаблон, по которому впоследствии можно создавать объекты, а можно и не создавать. В данном случае команды по созданию объекта класса `MyClass` могут выглядеть так:

```
MyClass obj;       // Объектная переменная
obj=new MyClass(); // Создание объекта
```

Или так:

```
MyClass obj=new MyClass();
```

В последнем случае объединены две команды: команда объявления объектной переменной и команда создания объекта.

НА ЗАМЕТКУ

Как уже упоминалось, в Java программа может состоять из нескольких классов. Классы можно описывать в разных файлах, но каждый класс должен быть описан только в одном файле (то есть нельзя разбивать описание класса на несколько файлов).

Поскольку все объекты класса создаются по единому шаблону (на основе класса), очевидно, что они имеют одинаковый набор полей и методов. Если в программе используется несколько объектов одного класса, необходимо как-то различать, поле или метод какого именно объекта используется, — ведь только по названию метода или поля этого не сделать. В Java, как и в прочих объектно-ориентированных языках, применяют так называемый *точечный синтаксис*. Основная его идея состоит в том, что при обращении к полю или методу объекта сначала указывается имя этого объекта, затем ставится точка и после этого следует имя поля или метода (с круглыми скобками и, если необходимо, аргументами).

НА ЗАМЕТКУ

Кроме обычных, существуют так называемые *статические* члены класса. Статический член класса — один для всех объектов класса. Для использования статического члена класса объект создавать не нужно. К статическому члену также обращаются с использованием точечного синтаксиса, но вместо имени объекта указывается имя класса.

В листинге 4.1 приведен пример программы, в которой, кроме главного класса, описан еще один класс (тот, который рассматривался выше). В этом классе объявляются несколько полей и методов. В главном методе на основе класса создаются объекты, и затем выполняются некоторые операции с этими объектами.

Листинг 4.1. Знакомство с классами и объектами

```
class MyClass{
   // Поля:
   String name;
```

```
   int number;
   // Методы:
   void set(String txt,int num){
      name=txt;
      number=num;
   }
   void show(){
      System.out.println(name+": "+number);
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание первого объекта:
      MyClass A=new MyClass();
      // Создание второго объекта:
      MyClass B;
      B=new MyClass();
      // Присваивание значений полям объектов:
      A.set("Объект A",100);
      B.name="Объект B";
      B.number=200;
      // Отображение значений полей объектов:
      A.show();
      B.show();
      // Изменение значения поля:
      A.number=300;
      // Отображение значений полей:
      A.show();
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 4.1)

```
Объект A: 100
Объект B: 200
Объект A: 300
```

В программе описывается класс `MyClass`, который имеет два поля (текстовое поле `name` и целочисленное поле `number`) и два метода (метод `set()` для присваивания значений полям и метод `show()` для отображения значений полей).

В методе `main()` класса `Demo` создаются два объекта класса `MyClass`. Первый объект создаем командой `MyClass A=new MyClass()`, а при создании второго объекта сначала командой `MyClass B` объявляется объектная переменная класса `MyClass`, а затем командой `B=new MyClass()` создается объект и ссылка на него записывается в переменную `B`. После создания объектов их полям присваиваются значения. Поля объекта `A` получают значение при выполнении команды `A.set("Объект A",100)`. Полям другого объекта значения присваиваются в индивидуальном порядке (команды `B.name="Объект B"` и `B.number=200`). Для отображения значений полей объектов вызываем метод `show()`

(команды `A.show()` и `B.show()`). Также в программе есть пример изменения значения поля `number` объекта `A` (команда `A.number=300`). Проверка с помощью вызова метода `show()` из объекта `A` подтверждает, что поле действительно изменило значение.

НА ЗАМЕТКУ

Еще раз обращаем внимание, что методы обращаются к полям того объекта, из которого они вызываются.

Статические поля и методы

— Что-то еще, джентльмены?

— Одну сигару на всех, сэр!

из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Как уже отмечалось, помимо обычных полей и методов (которые мы рассматривали выше) в классе могут быть *статические* члены. От нестатических членов статические принципиально отличаются тем, что они общие для всех объектов данного класса. Например, если речь идет о нестатическом поле, то у каждого объекта класса это поле имеет свое уникальное значение. Если поле является статическим, то у всех объектов значение этого поля одно и то же. Более того, статические поля существуют вне зависимости от того, создавались объекты данного класса или нет. Примерно такая же ситуация и со статическими методами: они не привязаны к объектам и существуют как бы сами по себе.

С формальной точки зрения статические поля и методы объявляются просто: достаточно в соответствующей инструкции указать идентификатор `static`. Статическому полю (да и нестатическому тоже) можно сразу (в описании) присвоить значение (которое впоследствии может быть изменено).

НА ЗАМЕТКУ

Если в описании нестатического поля ему сразу присвоить значение, то при создании объектов класса соответствующее поле будет иметь указанное значение.

Обращение к статическим полям и методам вне пределов класса, в котором они описаны, может выполняться обычным способом (через объект). Однако правильно делать это через класс.

ПОДРОБНОСТИ

В описании статического метода нельзя обращаться к нестатическим полям и методам класса. Причина в том, что статический метод не связан с конкретным объектом. А если нет объекта, то непонятно, о каких нестатических полях и методах может идти речь.

В листинге 4.2 приведен пример использования статических полей и методов.

Листинг 4.2. Использование статических полей и методов

```
import javax.swing.*;
// Класс со статическими членами:
class MyCalcs{
   // Статическое поле:
   static int number;
   // Статический метод:
   static String getCode(){
      String res="";
      int n=number;
      do{
         res=(n&1)+res;
         n>>>=1;
      }while(n>0);
      return res;
   }
}
// Главный класс:
class Demo{
   // Статические поля:
   static ImageIcon pict=new ImageIcon(
      "D:/Pictures/pict.png"
   );
   static String[] btns={"Далее","Закрыть"};
   // Статический метод:
   static void show(String txt){
      int res=JOptionPane.showOptionDialog(
         null, // Ссылка на родительское окно
         txt, // Текст сообщения
         "Сообщение", // Заголовок окна
         JOptionPane.DEFAULT_OPTION, // Кнопки
         JOptionPane.PLAIN_MESSAGE, // Пиктограмма
         pict, // Изображение для пиктограммы
         btns, // Набор кнопок
         btns[0] // Активная кнопка
      );
      // Завершение выполнения программы:
      if(res!=0) System.exit(0);
   }
   // Главный метод:
   public static void main(String[] args){
      // Отображается сообщение:
      show("Добрый день!\nПоказать бинарный код?");
      // Статическому полю присваивается значение:
      MyCalcs.number=25;
      // Формируется текст сообщения:
      String txt="Число: "+MyCalcs.number;
      txt+="\nБинарный код: "+MyCalcs.getCode();
      // Отображается сообщение:
      show(txt);
   }
}
```

При запуске программы появляется диалоговое окно (рис. 4.1).

Окно содержит кнопки Далее и Закрыть. Если нажать кнопку Закрыть, то выполнение программы на этом завершается. А если нажать кнопку Далее, то появится еще одно окно (рис. 4.2).

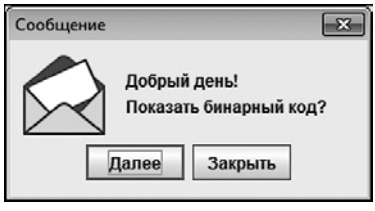


Рис. 4.1. Окно с двумя кнопками появляется при запуске программы

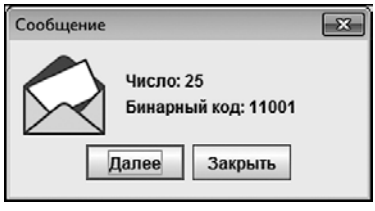


Рис. 4.2. Окно появляется, если в предыдущем окне нажата кнопка Далее

В окне представлен бинарный код (это `11001`) для числа `25`. Теперь проанализируем код программы.

В программе описан класс `MyCalcs` со статическим целочисленным полем `number` и статическим методом `getCode()`, который результатом возвращает текст с бинарным кодом числа, являющегося значением поля `number`.

ПОДРОБНОСТИ

Вычисление бинарного кода производится следующим образом. Соответствующее числовое значение копируется из поля `number` в локальную переменную `n`. В операторе цикла `do-while` инструкцией `n&1` определяется последний бит (`0` или `1`) в бинарном коде для текущего значения переменной `n`. Здесь использован оператор *побитовое и* `&`, а значение переменной `n` сравнивается со значением `1`. У числа `1` код простой — все нули и последняя единица. Поэтому результатом выражения `n&1` является `0`, если младший бит в значении переменной `n` нулевой. А если младший бит единичный, то результатом инструкции является значение `1`. Считанное значение дописывается в начало текста из переменной `res`, а командой `n>>>=1` битовое представление для значения переменной `n` сдвигается вправо на одну позицию. Предпоследний бит становится последним, а старший бит заполняется нулем.

Сформированное текстовое значение (переменная `res`) возвращается результатом метода.

НА ЗАМЕТКУ

Использованный алгоритм позволяет вычислять бинарные коды не только для положительных, но и для отрицательных чисел.

В главном классе `Demo`, помимо главного метода, описано два статических поля, причем им сразу присвоены значения. Статическое поле `pict` является ссылкой на объект класса `ImageIcon` (объект изображения). Статическое поле `btns` является ссылкой на текстовый массив (из двух элементов).

НА ЗАМЕТКУ

Для корректной работы программы в папку `D:\Pictures` помещается файл `pict.png` с картинкой. Она используется в качестве пиктограммы в диалоговых окнах. Элементы массива `btns` определяют названия кнопок.

Статический метод `show()` предназначен для отображения диалогового окна. Метод не возвращает результат, имеет один текстовый аргумент (текст сообщения в диалоговом окне).

ПОДРОБНОСТИ

Для отображения окна вызывается метод `showOptionDialog()` из класса `JOptionPane`. Аргументы у метода такие: `null` (ссылка на родительское окно, которого нет), `txt` (аргумент метода `show()` — текст сообщения), `"Сообщение"` (заголовок окна), `JOptionPane.DEFAULT_OPTION` (стандартный набор кнопок, но этот аргумент игнорируется, поскольку указаны отличные от `null` два последних аргумента), `JOptionPane.PLAIN_MESSAGE` (отсутствие пиктограммы, но этот аргумент игнорируется, поскольку задан отличный от `null` следующий аргумент), `pict` (ссылка на изображение для пиктограммы), `btns` (массив, определяющий набор кнопок в диалоговом окне), `btns[0]` (кнопка, которой передан фокус при отображении окна).

Метод `showOptionDialog()` результатом возвращает индекс кнопки, которую нажимает пользователь (если окно закрывают нажатием системной пиктограммы, то результатом метода является значение `-1`). Результат вызова метода записывается в переменную `res`, и если значение переменной отлично от нуля (то есть пользователь нажал не первую кнопку), то командой `System.exit(0)` завершается выполнение программы.

В главном методе командой `show("Добрый день!\nПоказать бинарный код?")` отображается окно с первым сообщением. Следующая команда `MyCalcs.number=25`, которой статическому полю присваивается значение, выполняется только в том случае, если пользователь в первом окне нажал кнопку Далее (в противном случае метод `show()` завершит выполнение программы). Затем формируется текст для следующего сообщения. Текст записывается в переменную `txt` и содержит значение поля `number` и результат вызова статического метода `getCode()`. Командой `show(txt)` отображается второе диалоговое окно с сообщением.

НА ЗАМЕТКУ

В принципе, к статическим полям и методам можно обращаться как к обычным, нестатическим членам, указывая вместо имени класса название объекта. Но это не самый удачный стиль, поскольку создается иллюзия, что выполняется обращение к полю или методу, относящимся к объекту, что на самом деле не так.

Если мы используем статическое поле или метод в классе, в котором они описаны, то имя класса можно не указывать.

Закрытые члены класса

Нормальные герои всегда идут в обход!

из к/ф «Айболит-66»

Поля и методы могут быть *закрытыми*. В таком случае доступ к ним есть только внутри кода класса. Чтобы поле или метод были закрытыми, они описываются с ключевым словом `private`.

НА ЗАМЕТКУ

Члены класса могут быть открытыми (описываются со спецификатором доступа `public` или вообще без спецификатора доступа — как мы поступали ранее), закрытыми (описываются со спецификатором доступа `private`) и защищенными (описываются со спецификатором доступа `protected`). Защищенные члены класса будут обсуждаться после того, как мы познакомимся с наследованием.

Пример в листинге 4.3 иллюстрирует разницу между открытыми и закрытыми членами класса.

Листинг 4.3. Закрытые и открытые члены класса

```
class MyClass{
   // Закрытое поле:
   private int number;
   // Закрытый метод:
   private int reverse(){
      int n=number;
      int res=0;
      do{
         res=res*10+(n%10);
         n/=10;
      }while(n>0);
      return res;
   }
   // Открытый метод:
   public void show(){
      // Обращение к закрытым членам:
      System.out.println("Исходное число: "+number);
      System.out.println("Инверсия числа: "+reverse());
   }
   // Открытый метод:
   public void set(int n){
      // Обращение к закрытому полю:
      number=n;
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
```

```
        // Создание объекта:
        MyClass obj=new MyClass();
        // Вызов открытых методов:
        obj.set(12345);
        obj.show();
    }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 4.3)

```
Исходное число: 12345
Инверсия числа: 54321
```

В программе в классе `MyClass` объявлено закрытое целочисленное поле `number`. Поскольку поле закрытое, присвоить ему значения вне класса прямым обращением нельзя. Поэтому для присваивания значения полю предусмотрен метод `set()`. У метода целочисленный аргумент, значение которого присваивается полю. Обращаем внимание, что метод `set()` имеет доступ ко всем членам класса, в том числе и к закрытому полю, поскольку метод описан в том же классе, что и поле.

Еще один открытый метод `show()` предназначен для отображения значения поля, и, кроме этого, метод `show()` отображает значение поля в обратном порядке (то есть в позиционном представлении числа разряды идут слева направо). Чтобы получить такое инвертированное число, вызывается закрытый метод `reverse()`.

ПОДРОБНОСТИ

Результат метода `reverse()` вычисляется так. Значение поля `number` записывается в локальную переменную `n`. Еще одна локальная переменная `res` (которая потом возвращается результатом метода) имеет нулевое начальное значение. Затем в операторе цикла `do-while` инструкцией `n%10` вычисляется последняя цифра в представлении значения переменной `n`. Это число прибавляется к значению переменной `res`, умноженному на `10` (команда `res=res*10+(n%10)`). Когда мы умножаем на `10` некоторое число, то все цифры в позиционном представлении этого числа сдвигаются на одну позицию влево, а последней становится цифра `0`. Если теперь к этому выражению прибавить число не большее `10` (остаток от деления на `10` удовлетворяет этому условию), то такое число просто допишется вместо последней цифры `0`. После того как изменено значение переменной `res`, командой `n/=10` переменной `n` присваивается результат целочисленного деления текущего значения переменной на `10`. Эта операция на практике означает отбрасывание последней цифры в значении переменной `n`. Последней станет предпоследняя цифра, которая на следующей итерации будет добавлена в конец значения, записанного в переменную `res`.

В главном методе программы создается объект `obj` класса `MyClass` и из него вызываются открытые методы `set()` и `show()`.

НА ЗАМЕТКУ

Методы `set()` и `show()` описаны со спецификатором доступа `public`. Необходимости в этом нет, и можно было бы не указывать спецификатор доступа совсем — в этом случае методы были бы открытыми. То есть в Java по умолчанию поля и методы являются открытыми. Мы использовали ключевое слово `public`, чтобы лишний раз подчеркнуть, что речь в данном случае идет об открытых методах.

Вновь подчеркнем, что в методе `main()` нельзя обратиться к полю `number` или методу `reverse()` объекта `obj`. Причина в том, что эти члены класса `MyClass` являются закрытыми (описаны со спецификатором `private`). В программе использован следующий подход: доступ к закрытым членам осуществляется через открытые методы. На первый взгляд, данный способ реализации класса может показаться нелогичным и неудобным. Представим, однако, ситуацию, когда необходимо ограничить и четко регламентировать операции, допустимые с полями класса. Самый надежный способ для этого — сделать поля закрытыми, а для допустимых операций над полями предусмотреть открытые методы. В качестве аналогии можно привести черный ящик: внутреннее содержимое ящика — закрытые члены класса. Открытые методы — рычажки, которые позволяют запускать внутренние механизмы. Что не предусмотрено конструкцией, выполнено быть не может. Такой подход позволяет создавать безопасные и надежные коды и широко используется на практике.

Ключевое слово this

> Дальше следует непереводимая игра слов с использованием местных идиоматических выражений.
>
> *из к/ф «Бриллиантовая рука»*

Нередко в процессе описания методов в классе необходимо явно сослаться на объект, из которого будет вызываться метод. Проблема в том, что когда описывается метод, объекта еще не существует в принципе. Именно в таких случаях и используется ключевое слово `this`, которое является стандартной ссылкой на объект, из которого вызывается метод.

НА ЗАМЕТКУ

Есть еще один способ использования ключевого слова `this` — в случае, когда в теле одной версии конструктора нужно вызвать другую версию конструктора.

В предыдущих примерах ссылки на члены класса в теле методов этого же класса выполнялись простым указанием имени соответствующего члена. Но в действительности это упрощенная форма синтаксиса. К нестатическим членам нужно об-

ращаться с указанием объекта. Просто когда мы описывали методы в классе, такого объекта не было и мы пошли простым (но законным) путем. Однако так поступать удается не всегда. Например, возможна следующая ситуация: аргумент метода имеет такое же название, что и поле. Это само по себе корректно. Но в Java есть правило: в случае совпадения названий полей и локальных переменных локальные переменные имеют приоритет. Аргументы имеют силу локальных переменных. Поэтому если в теле метода указан соответствующий идентификатор, то он будет отождествляться с аргументом метода, а не с полем. Чтобы обратиться к полю, придется использовать ключевое слово `this`. Пример приведен в листинге 4.4.

Листинг 4.4. Использование ссылки this

```
class MyClass{
   // Поле класса:
   char symb;
   void set(char symb){
      // Использование ссылки this:
      this.symb=symb;
   }
   void show(){
      System.out.println("Символ: "+symb);
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объекта:
      MyClass obj=new MyClass();
      // Полю присваивается значение:
      obj.set('A');
      // Отображение значения поля:
      obj.show();
      // Новое значение поля:
      obj.symb='B';
      // Отображение значения поля:
      obj.show();
   }
}
```

Ниже представлен результат выполнения этой программы:

Результат выполнения программы (из листинга 4.4)

```
Символ: A
Символ: B
```

В классе `MyClass` есть поле `symb` типа `char` и два метода. Метод `show()` не имеет аргументов и не возвращает результат. При вызове метода `show()` отображается значение поля `symb` объекта, из которого вызывается метод. А вот у метода `set()` есть аргумент, причем называется он так же, как и поле. Поэтому в теле метода (при присвоении полю значения) мы используем команду `this.symb=symb`. Здесь

инструкция `symb` означает обращение к аргументу метода, а инструкция `this.symb` является ссылкой на поле.

В главном методе программы создается объект класса `MyClass`, полю объекта присваивается значение как с помощью метода `set()`, так и через прямую ссылку, а также отображается значение поля.

Внутренние классы

> А если надо будет — снова пойдем кривым путем.
>
> *из к/ф «Айболит-66»*

Внутренний класс — это класс, описанный внутри другого класса. Класс, в котором объявлен внутренний класс, называется *внешним*.

Внутренний класс имеет несколько особенностей. Во-первых, члены внутреннего класса доступны только в пределах внутреннего класса и недоступны во внешнем классе (даже если они открытые). Во-вторых, во внутреннем классе можно обращаться к членам внешнего класса напрямую.

НА ЗАМЕТКУ

Здесь имеется в виду следующее. Если мы описываем метод во внутреннем классе, то из этого метода можно обращаться к полям, объявленным во внешнем классе. Если мы описываем метод во внешнем классе, то прямого доступа к полям внутреннего класса он не имеет (но можно создать объект внутреннего класса и через него получить доступ к членам внутреннего класса).

Пример использования внутреннего класса приведен в листинге 4.5.

Листинг 4.5. Использование внутреннего класса

```
// Внешний класс:
class Alpha{
   // Поле внешнего класса:
   int number=123;
   // Метод внешнего класса:
   void show(){
      // Создание объекта внутреннего класса:
      Bravo B=new Bravo();
      // Вызов метода из объекта внутреннего класса:
      B.display();
   }
   // Внутренний класс:
   class Bravo{
      // Метод внутреннего класса:
```

```
      void display(){
         System.out.println("Поле number: "+number);
      }
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объекта внешнего класса:
      Alpha A=new Alpha();
      // Вызов метода объекта внешнего класса:
      A.show();
   }
}
```

Результат выполнения программы таков:

Результат выполнения программы (из листинга 4.5)

```
Поле number: 123
```

В программе описаны три класса: внешний класс `Alpha`, описанный в нем внутренний класс `Bravo`, а также главный класс `Demo`. В классе `Demo` описан метод `main()`, в котором создается объект `A` внешнего класса `Alpha`, и из этого объекта вызывается метод `show()`.

Во внешнем классе `Alpha` есть поле `number`, метод `show()` и еще описывается внутренний класс `Bravo`. У внутреннего класса есть метод `display()`, который вызывается в методе `show()` внешнего класса. Для вызова метода `display()` в методе `show()` создается объект `B` внутреннего класса `Bravo`.

НА ЗАМЕТКУ

Хотя метод `display()` формально описан внутри класса `Alpha` (но во внутреннем классе `Bravo`), вызывать метод `display()` напрямую нельзя — члены внутреннего класса во внешнем классе недоступны.

Методом `display()` отображается сообщение со значением поля внешнего класса `number`.

НА ЗАМЕТКУ

Поскольку во внутреннем классе допускается непосредственное обращение к членам внешнего класса, обращение к полю `number` выполняется простым указанием его имени.

Отметим, что в главном методе программы можно создать объект внешнего класса, но нельзя создать объект внутреннего класса — за пределами внешнего класса внутренний класс недоступен.

Анонимные объекты

Какая встреча! И какая неприятная!

из к/ф «Айболит-66»

Как уже отмечалось, при создании объектов с помощью оператора `new` возвращается ссылка на вновь созданный объект. Прелесть ситуации состоит в том, что эту ссылку необязательно присваивать в качестве значения переменной. В таких случаях создается *анонимный объект*. Другими словами, объект есть, а переменной, которая содержала бы ссылку на этот объект, нет. С практической точки зрения такая возможность представляется сомнительной, но это только на первый взгляд. На самом деле потребность в анонимных объектах возникает довольно часто — обычно в тех ситуациях, когда объект используется лишь один раз. Простой пример применения анонимного объекта приведен в листинге 4.6.

Листинг 4.6. Анонимный объект

```
class MyClass{
   void show(String msg){
      System.out.println(msg);
   }
}
class Demo{
   public static void main(String[] args){
      // Использование анонимного объекта:
      new MyClass().show("Анонимный объект");
   }
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 4.6)

```
Анонимный объект
```

В классе `MyClass` описан всего один метод `show()` с текстовым аргументом, значение которого отображается в области вывода.

В методе `main()` в главном классе `Demo` имеется всего одна команда `new MyClass().show("Анонимный объект")`. Ее можно условно разбить на две части. Инструкцией `new MyClass()` создается новый объект класса `MyClass`, а сама инструкция в качестве значения возвращает ссылку на созданный объект. Поскольку ссылка никакой переменной не присваивается, созданный объект является анонимным. Однако это все равно объект класса `MyClass`, поэтому у него есть метод `show()`. Именно он вызывается с аргументом `"Анонимный объект"`. Для этого после инструкции `new MyClass()` ставится точка и указывается имя метода с соответствующим аргументом.

НА ЗАМЕТКУ

Мы рассмотрели искусственный пример. Его основное назначение — проиллюстрировать концепцию в целом. Мы еще вернемся к анонимным объектам.

Работа с классами и объектами

Ты ничего не сможешь сделать с нами,
потому что ты хочешь все сделать сразу.

из к/ф «Айболит-66»

Далее рассматриваются некоторые программы, в которых, кроме главного класса программы (класса, содержащего метод `main()`), описываются и используются другие классы.

Схема Бернулли

Схемой Бернулли называется серия независимых испытаний, в каждом из которых может быть только один из двух случайных результатов — их принято называть *успехом* и *неудачей*. Есть два важных параметра, которые определяют все прочие свойства серии опытов: это вероятность успеха в одном опыте p и количество опытов в серии n. Величина $q = 1 - p$ называется вероятностью неудачи в одном опыте. Достаточно часто на практике используется случайная величина (назовем ее ξ), которая определяется как число успехов в схеме Бернулли. Математическое ожидание этой случайной величины $M\xi = np$, а дисперсия равна $D\xi = npq$. Среднее значение для количества успехов в схеме Бернулли является оценкой математического ожидания, поэтому для схемы с большим количеством испытаний с высокой вероятностью количество успехов в схеме Бернулли близко к математическому ожиданию. Корень квадратный из дисперсии определяет характерную область разброса количества успехов по отношению к математическому ожиданию.

В листинге 4.7 представлен код программы, в которой для реализации схемы Бернулли создается специальный класс.

Листинг 4.7. Схема Бернулли

```
import static java.lang.Math.*;
class Bernoulli{
   // Количество опытов (испытаний) в схеме:
   private int n;
   // Вероятность успеха:
   private double p;
   // Результат испытаний:
   private boolean[] test;
   // Метод для определения параметров схемы:
```

```
   public void setAll(int n,double p){
      if(n>=0) this.n=n;
      else n=0;
      if(p>=0&&p<=1) this.p=p;
      else this.p=0;
      test=new boolean[n];
      for(int i=0;i<n;i++){
         if(random()<=p) test[i]=true;
         else test[i]=false;
      }
   }
   // Подсчет количества успехов:
   private int getVal(){
      int count,i;
      for(i=0,count=0;i<n;i++){
         if(test[i]) count++;
      }
      return count;
   }
   // Отображение основных характеристик:
   public void show(){
      System.out.println("СТАТИСТИКА ДЛЯ СХЕМЫ БЕРНУЛЛИ");
      System.out.println("Испытаний: "+n);
      System.out.println("Вероятность успеха: "+p);
      System.out.println("Успехов: "+getVal());
      System.out.println("Неудач: "+(n-getVal()));
      System.out.println("Мат. ожидание: "+n*p);
      System.out.println("Отклонение: "+sqrt(n*p*(1-p)));
   }
}
class Demo{
   public static void main(String[] args){
      // Создание объекта:
      Bernoulli obj=new Bernoulli();
      // Определение количества испытаний
      // и вероятности успеха:
      obj.setAll(10000,0.36);
      // Отображение результата:
      obj.show();
   }
}
```

В классе `Bernoulli` объявляется несколько полей и методов. Закрытое целочисленное поле `n` предназначено для записи количества испытаний в схеме Бернулли. Закрытое поле `p` типа `double` содержит значение вероятности успеха в одном испытании. Поля специально объявлены как закрытые, поскольку их изменение влечет за собой изменение всей статистики, связанной со схемой Бернулли. Чтобы нельзя было изменять значения полей независимо от других параметров, эти поля и были закрыты. Это же замечание относится к закрытому полю `test`. Поле является ссылкой на массив с элементами типа `boolean` (значение `true` соответствует успеху в соответствующем опыте, а неудаче соответствует значение `false`). Данный

массив представляет собой результат серии испытаний. Его размер определяется значением поля n. Заполнение осуществляется с учетом значения поля p (поэтому при изменении хотя бы одного из этих полей должен был бы измениться и массив, на который ссылается переменная массива test).

Создание и заполнение массива реализованы через метод setAll(), у которого два аргумента: первый целочисленный аргумент определяет значение поля n, а второй аргумент типа double определяет значение поля p. При присваивании значений полям проверяется условие, попадают ли переданные методу аргументы в диапазон допустимых значений. Для количества испытаний значение должно быть неотрицательным, а для вероятности успеха в одном опыте значение должно быть неотрицательным и не должны превышать единицу. Если данные критерии не соблюдаются, то соответствующему полю присваивается нулевое значение.

После того как значения полям n и p присвоены, командой test=new boolean[n] создается массив нужного размера и ссылка на него присваивается в качестве значения полю test. Заполнение массива реализовано посредством оператора цикла. При заполнении элементов массива использован метод для генерирования случайных чисел random() из класса Math. Метод возвращает псевдослучайное число в диапазоне от 0 до 1. Правило заполнения элементов массива следующее: если сгенерированное число не превышает значения поля p, элементу присваивается значение true, в противном случае — значение false. Поскольку генерируемые методом random() значения равномерно распределены в интервале от 0 до 1, элемент массива test принимает значение true с вероятностью p и значение false с вероятностью 1-p, чего мы и добивались.

Возвращаемым значением метода getVal() является целое число — количество успехов в схеме Бернулли. Подсчет выполняется по элементам массива test. Результат записывается в локальную переменную count. Перебираются все элементы массива test; если значение элемента равно true, то значение переменной count увеличивается на единицу. После завершения оператора цикла значение переменной count возвращается в качестве результата метода.

Метод getVal() закрытый и вызывается в открытом методе show(). Методом show() выводится практически вся полезная информация относительно схемы Бернулли. В частности, отображается количество испытаний, вероятность успеха в одном испытании, подсчитывается количество успехов в серии, вычисляется количество неудач, а также математическое ожидание и стандартное отклонение.

В главном методе программы командой Bernoulli obj=new Bernoulli() создается объект obj класса Bernoulli. Командой obj.setAll(10000,0.36) заполняются поля объекта, а командой obj.show() отображается результат — например, так:

Результат выполнения программы (из листинга 4.7)

```
СТАТИСТИКА ДЛЯ СХЕМЫ БЕРНУЛЛИ
Испытаний: 10000
Вероятность успеха: 0.36
```

```
Успехов: 3577
Неудач: 6423
Мат. ожидание: 3600.0
Отклонение: 48.0
```

Единственный способ получить доступ к полям класса `Bernoulli` для изменения их значений состоит в вызове метода `setAll()`. Такой подход делает невозможным несанкционированное изменение значений полей.

Математические функции

В Java основные математические функции реализованы с помощью статических методов класса `Math`. Недостающие функции можно описать самостоятельно (листинг 4.8).

Листинг 4.8. Математические функции

```
// Класс с реализацией математических функций:
class MyMath{
   // Граница интервала для разложения в ряд Фурье:
   static double L=Math.PI;
   // Экспонента:
   static double Exp(double x,int N){
      int i;
      double s=0,q=1;
      for(i=0;i<N;i++){
         s+=q;
         q*=x/(i+1);
      }
      return s+q;
   }
   // Синус:
   static double Sin(double x,int N){
      int i;
      double s=0,q=x;
      for(i=0;i<N;i++){
         s+=q;
         q*=(-1)*x*x/(2*i+2)/(2*i+3);
      }
      return s+q;
   }
   // Косинус:
   static double Cos(double x,int N){
      int i;
      double s=0,q=1;
      for(i=0;i<N;i++){
         s+=q;
         q*=(-1)*x*x/(2*i+1)/(2*i+2);
      }
      return s+q;
   }
```

```
   // Функция Бесселя:
   static double BesselJ(double x,int N){
      int i;
      double s=0,q=1;
      for(i=0;i<N;i++){
         s+=q;
         q*=(-1)*x*x/4/(i+1)/(i+1);
      }
      return s+q;
   }
   // Ряд Фурье по синусам:
   static double FourSin(double x,double[] a){
      int i,N=a.length;
      double s=0;
      for(i=0;i<N;i++){
         s+=a[i]*Math.sin(Math.PI*x*(i+1)/L);
      }
      return s;
   }
   // Ряд Фурье по косинусам:
   static double FourCos(double x,double[] a){
      int i,N=a.length;
      double s=0;
      for(i=0;i<N;i++){
         s+=a[i]*Math.cos(Math.PI*x*i/L);
      }
      return s;
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      System.out.println("Примеры вычислений:");
      // Вычисление экспоненты:
      System.out.println(
         "exp(1)="+MyMath.Exp(1,30)
      );
      // Вычисление синуса:
      System.out.println(
         "sin(pi)="+MyMath.Sin(Math.PI,100)
      );
      // Вычисление косинуса:
      System.out.println(
         "cos(pi/2)="+MyMath.Cos(Math.PI/2,100)
      );
      // Вычисление функции Бесселя:
      System.out.println(
         "J0(mu1)="+MyMath.BesselJ(2.404825558,100)
      );
      // Вычисление коэффициентов для ряда Фурье
      // для функции y(x)=x:
      int m=1000;
      double[] a=new double[m];
```

```
      double[] b=new double[m+1];
      b[0]=MyMath.L/2;
      for(int i=1;i<=m;i++){
         a[i-1]=(2*(i%2)-1)*2*MyMath.L/Math.PI/i;
         b[i]=-4*(i%2)*MyMath.L/Math.pow(Math.PI*i,2);
      }
      // Вычисление значения функции y(x)=x
      // через синус-ряд Фурье:
      System.out.println("2.0->"+MyMath.FourSin(2.0,a));
      // Вычисление значения функции y(x)=x
      // через косинус-ряд Фурье:
      System.out.println("2.0->"+MyMath.FourCos(2.0,b));
   }
}
```

В программе описывается класс `MyMath`, в котором объявлено несколько статических методов. С их помощью реализуются математические функции для вычисления экспоненты, косинуса и синуса, а также функции Бесселя нулевого индекса. Во всех перечисленных случаях для вычисления значений функций используется ряд Тейлора. Каждый метод имеет по два аргумента: первый аргумент типа `double` определяет непосредственно аргумент математической функции, а второй целочисленный аргумент определяет количество слагаемых в ряде Тейлора, на основании которых вычисляется функция. Для экспоненты использован ряд

$$\exp(x) \approx \sum_{k=0}^{N} \frac{x^k}{k!}.$$

Синус и косинус вычисляются соответственно по формулам

$$\sin(x) \approx \sum_{k=0}^{N} \frac{(-1)^k x^{2k+1}}{(2k+1)!},$$

$$\cos(x) \approx \sum_{k=0}^{N} \frac{(-1)^k x^{2k}}{(2k)!}.$$

Для функции Бесселя нулевого индекса использован ряд

$$J_0(x) \approx \sum_{k=0}^{N} \frac{(-1)^k \left(\frac{x}{2}\right)^{2k}}{\left(k!\right)^2}.$$

Кроме этих функций, в классе `MyMath` определены методы для вычисления рядов Фурье по базовым функциям $\sin(\pi nx/L)$ (метод `FourSin()`) и $\cos(\pi nx/L)$ (метод `FourCos()`). В частности, у метода `FourSin()` два аргумента: первый — это переменная `x` типа `double`, второй — массив `a` с элементами типа `double`. В качестве результата методом возвращается сумма вида

$$\sum_{n=1}^{N} a_{n-1} \sin\left(\frac{\pi n x}{L}\right).$$

Здесь через a_n обозначены элементы массива `a`, верхняя граница суммы N определяется по количеству элементов этого массива (команда `N=a.length`), а сама сумма является разложением в ряд (берется конечное количество слагаемых ряда) на интервале от `0` до `L` некоторой функции с коэффициентами разложения, записанными в массив `a`. Параметр `L` объявлен в классе `MyMath` как статическое поле со значением `Math.PI`.

Напомним, что разложением функции $y(x)$ в ряд Фурье по синусам на интервале от 0 до L называется бесконечная сумма

$$y(x) = \sum_{n=1}^{\infty} y_n \sin\left(\frac{\pi n x}{L}\right).$$

Коэффициенты разложения рассчитываются так:

$$y_n = \frac{2}{L}\int_0^L y(x)\sin\left(\frac{\pi n x}{L}\right)dx.$$

Этот ряд дает значение функции $y(x)$ при $0 < x < L$. Аналогично, рядом Фурье по косинусам для функции $y(x)$ на интервале от 0 до L называется бесконечная сумма

$$y(x) = y_0 + \sum_{n=1}^{\infty} y_n \cos\left(\frac{\pi n x}{L}\right).$$

Коэффициенты разложения при $n > 0$ такие:

$$y_n = \frac{2}{L}\int_0^L y(x)\cos\left(\frac{\pi n x}{L}\right)dx.$$

При $n = 0$ коэффициент вычисляется так:

$$y_0 = \frac{1}{L}\int_0^L y(x)dx.$$

Этот ряд также дает значение функции $y(x)$ при $0 < x < L$.

НА ЗАМЕТКУ

Представление функции в виде ряда Фурье часто используется в математической физике, например, при решении задач теплопроводности. С практической точки зрения задача сводится к вычислению коэффициентов разложения. Если коэффициенты разложения известны, то, вычислив ряд по базисным функциям, получаем значение для функции (приближенное, поскольку ряд вычисляется по конечному количеству слагаемых).

В главном методе программы в классе `Demo` приводятся примеры вычислений с использованием описанных в классе `MyMath` статических методов. Приводим один из примеров:

Результат выполнения программы (из листинга 4.8)

```
Примеры вычислений:
exp(1)=2.7182818284590455
sin(pi)=2.4790606536130346E-16
cos(pi/2)=4.590388303752165E-17
J0(mu1)=-1.5793881580131606E-10
2.0->1.9996438367829905
2.0->1.999999349096476
```

Экспонента вычисляется с единичным первым аргументом, что позволяет получить оценку для постоянной Эйлера $e \approx 2{,}7182818284590452$.

НА ЗАМЕТКУ

Вычисленное в программе значение для постоянной Эйлера отличается от точного лишь в последнем знаке.

Для вычисления синуса в качестве первого аргумента указано значение `Math.PI`, а при вычислении косинуса первым аргументом соответствующего метода — значение `Math.PI/2`. Поэтому как синусам, так и косинусам должно возвращаться нулевое значение. Точный результат обеспечивается в данном случае с достаточно неплохой точностью (до 15 знаков после запятой).

Для вычисления функции Бесселя в качестве первого аргумента указано значение первого нуля функции Бесселя $\mu_1 \approx 2{,}404825558$.

НА ЗАМЕТКУ

Нулями μ_n (n = 1, 2, ...) функции Бесселя нулевого индекса $J_0(x)$ называются неотрицательные решения уравнения $J_0(\mu_n) = 0$. Поэтому для указанного аргумента значение функции должно быть нулевым (в пределах точности вычислений) — что мы и наблюдаем по результатам выполнения программы.

Методы `FourSin()` и `FourCos()` используются для вычисления значения функции $y(x) = x$ в точке $x = 2$. Для этого предварительно формируются массивы `a` и `b` с коэффициентами разложения функции $y(x) = x$ в ряд по синусам и косинусам соответственно. Мы использовали известные аналитические выражения для коэффициентов разложения этой функции в ряд Фурье. Так, для синус-разложения коэффициенты разложения $y_n = (-1)^{n+1}\dfrac{2L}{\pi n}$, а для косинус-разложения коэффициенты $y_n = \dfrac{2L((-1)^n - 1)}{\pi^2 n^2}$ при $n > 0$ и $y_0 = L/2$.

Для заполнения массивов a и b соответствующими значениями в главном методе программы используется оператор цикла. После заполнения массивов командами MyMath.FourSin(2.0,a) и MyMath.FourCos(2.0,b) вычисляются два различных ряда Фурье для функции $y(x) = x$ при значении $x = 2$, то есть в обоих случаях точным результатом является значение 2.

НА ЗАМЕТКУ

Желающие могут сопоставить точность вычислений и количество слагаемых (размеры массивов a и b), оставленных в рядах Фурье для вычисления этого результата.

Динамический список из объектов

Следующий пример связан с созданием динамического списка из объектов. Каждый объект в этом списке, кроме числового поля, содержит ссылку на следующий объект, а самый последний объект ссылается на самый первый. При этом объекты имеют метод, возвращающий в качестве результата значение поля объекта, отстоящего от текущего объекта (из которого вызывается метод) на определенное количество позиций. Соответствующий код представлен в листинге 4.9.

Листинг 4.9. Динамический список объектов

```
class MyClass{
   // Поле для нумерации объектов:
   int number=0;
   // Ссылка на следующий объект:
   MyClass next=this;
   // Метод для создания списка объектов:
   void create(int n){
      int i;
      MyClass A=this;
      MyClass B;
      // Создание списка:
      for(i=1;i<=n;i++){
         B=new MyClass();
         A.next=B;
         B.number=A.number+1;
         A=B;
      }
      // Последний объект списка
      // ссылается на начальный:
      A.next=this;
   }
   // Метод для получения номера объекта в списке:
   int get(int k){
      int i;
      MyClass obj=this;
      for(i=1;i<=k;i++){
         obj=obj.next;
```

```
      }
      return obj.number;
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Исходный объект:
      MyClass obj=new MyClass();
      // Создание списка из четырех объектов
      // (начальный и еще три объекта):
      obj.create(3);
      // Проверка содержимого списка:
      System.out.println("Проверка значения поля number");
      System.out.println(
         "2-й после начального:  "+obj.get(2)
      );
      System.out.println(
         "4-й после начального:  "+obj.get(4)
      );
      System.out.println(
         "2-й объект после 1-го: "+obj.next.get(2)
      );
   }
}
```

В программе описывается класс `MyClass`, у которого всего два поля: целочисленное поле `number` с нулевым значением по умолчанию и объектная переменная `next` класса `MyClass`. В эту переменную записывается ссылка на следующий объект списка. По умолчанию полю присваивается ссылка `this` (поле ссылается на свой собственный объект).

Для создания списка объектов предусмотрен метод `create()`. Метод не возвращает результат, а аргументом ему передается целое число, которое определяет количество объектов, добавляемых в список. В методе объявляется локальная целочисленная индексная переменная `i`, локальная объектная переменная `A` класса `MyClass` с начальным значением `this` (ссылка на объект, из которого вызывается метод `create()`), а также объектная переменная `B` класса `MyClass`. Затем запускается оператор цикла, в котором индексная переменная пробегает значения от `1` до `n` (аргумент метода `create()`). Командой `B=new MyClass()` в операторе цикла создается объект класса `MyClass`, и ссылка на этот объект присваивается значением переменной `B`. Командой `A.next=B` в поле `next` объекта `A` записывается ссылка на объект `B` (в поле `next` текущего объекта списка записывается ссылка на следующий объект списка). Далее командой `B.number=A.number+1` полю `number` следующего объекта списка присваивается значение, на единицу большее значения поля `number` текущего объекта списка. Наконец, командой `A=B` переменной `A` присваивается ссылка на следующий объект списка. На очередной итерации новое значение получит и переменная `B`. После завершения оператора цикла переменные `A` и `B` будут ссылаться на последний объект списка. Полю `number` этого объекта значение уже присвоено (при выполнении оператора цикла).

Осталось только присвоить значение его полю next (по умолчанию поле next ссылается на свой же объект). Новое значение полю присваивается командой A.next=this. В данном случае this — это ссылка на объект, из которого вызывался метод create(), то есть ссылка на начальный объект списка. Таким образом, список создан.

Метод get() возвращает в качестве результата целочисленное значение поля number объекта, который отстоит в списке на указанное количество позиций (аргумент метода) по отношению к объекту, из которого вызывается метод. Поскольку объекты в списке ссылаются друг на друга циклически (последний объект ссылается на первый), аргумент метода get() может быть больше, чем количество объектов в списке. Алгоритм выполнения метода достаточно прост: в методе запускается оператор цикла, в котором командой obj=obj.next в локальную объектную переменную obj записывается ссылка на следующий объект списка (напомним, эта ссылка хранится в поле next). После завершения оператора цикла переменная obj ссылается на нужный объект. В качестве результата возвращается поле number этого объекта.

В главном методе программы в классе Demo командой MyClass obj=new MyClass() создается базовый начальный объект. Затем командой obj.create(3) создается список объектов (всего четыре объекта — один начальный и еще три к нему добавляются). Поле number начального объекта имеет по умолчанию значение 0, а у следующих в списке объектов значения полей number равны 1, 2 и 3. После этого несколькими командами выполняется проверка свойств созданной структуры объектов. В результате выполнения программы получаем следующее:

Результат выполнения программы (из листинга 4.9)

```
Проверка значения поля number
2-й после начального: 2
4-й после начального: 0
2-й объект после 1-го: 3
```

В частности, командой obj.get(2) возвращается значение поля number объекта, смещенного от начального объекта на две позиции, то есть значение 2. Командой obj.get(4) возвращается значение поля number объекта, отстоящего от начального на 4 позиции. На 3 позиции от начального размещен последний объект в списке. Этот объект ссылается на начальный объект. Поэтому в результате выполнения команды obj.get(4) возвращается значение поля number начального объекта, то есть значение 0. Наконец, командой obj.next.get(2) возвращается значение поля number объекта, смещенного на две позиции от объекта, ссылка на который записана в поле next объекта obj (начальный объект). Это третий объект после начального. Соответственно, результатом команды является значение 3.

Работа с матрицами

В листинге 4.10 приведен простой пример программы, в которой для работы с квадратными матрицами создается специальный класс. В этом классе преду-

смотрены методы для выполнения таких операций, как вычисление детерминанта (определителя), транспонирование матрицы, вычисление следа матрицы (сумма диагональных элементов), заполнение матрицы числами в различных режимах, вывод значений матрицы на экран.

Листинг 4.10. Работа с квадратными матрицами

```
class Matrix{
   // Размер матрицы:
   private int n;
   // Ссылка на двумерный массив:
   private int[][] matrix;
   // Метод для создания матрицы:
   void create(int n){
      this.n=n;
      matrix=new int[n][n];
   }
   // Метод для заполнения матрицы
   // случайными числами:
   void rand(){
      int i,j;
      for(i=0;i<n;i++){
         for(j=0;j<n;j++){
            matrix[i][j]=(int)(Math.random()*10);
         }
      }
   }
   // Метод для заполнения матрицы одним числом:
   void value(int a){
      int i,j;
      for(i=0;i<n;i++){
         for(j=0;j<n;j++){
            matrix[i][j]=a;
         }
      }
   }
   // Метод для циклического заполнения матрицы
   // последовательностью цифр:
   void digits(){
      int i,j;
      for(i=0;i<n;i++){
         for(j=0;j<n;j++){
            matrix[i][j]=(i*n+j)%9+1;
         }
      }
   }
   // Метод для заполнения единичной матрицы:
   void unit(){
      int i;
      value(0);
      for(i=0;i<n;i++){
         matrix[i][i]=1;
      }
```

```
}
// Метод для вычисления следа матрицы:
int trace(){
   int i,s=0;
   for(i=0;i<n;i++){
      s+=matrix[i][i];
   }
   return s;
}
// Метод для вычисления определителя матрицы:
int det(){
   int D=0;
   switch(n){
      // Матрица размером 1 на 1:
      case 1:
         D=matrix[0][0];
         break;
      // Матрица размером 2 на 2:
      case 2:
         D=matrix[0][0]*matrix[1][1]-
           matrix[0][1]*matrix[1][0];
         break;
      // Прочие случаи:
      default:
         int i,j,k,sign=1;
         Matrix m;
         for(k=0;k<n;k++){
            m=new Matrix();
            m.create(n-1);
            for(i=1;i<n;i++){
               for(j=0;j<k;j++){
                  m.matrix[i-1][j]=matrix[i][j];
               }
               for(j=k+1;j<n;j++){
                  m.matrix[i-1][j-1]=matrix[i][j];
               }
            }
            D+=sign*matrix[0][k]*m.det();
            sign*=(-1);
         }
   }
   return D;
}
// Метод для транспонирования матрицы:
void trans(){
   int i,j,s;
   for(i=0;i<n;i++){
      for(j=i+1;j<n;j++){
         s=matrix[i][j];
         matrix[i][j]=matrix[j][i];
         matrix[j][i]=s;
      }
   }
```

```
   }
   // Метод для отображения матрицы:
   void show(){
      int i,j;
      for(i=0;i<n;i++){
         for(j=0;j<n;j++){
            System.out.print(matrix[i][j]+ " ");
         }
         System.out.println();
      }
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объекта:
      Matrix obj=new Matrix();
      // Создание матрицы:
      obj.create(3);
      // Единичная матрица:
      obj.unit();
      System.out.println("Единичная матрица:");
      obj.show();
      // Заполнение одним и тем же числом:
      obj.value(5);
      System.out.println("Все элементы одинаковые:");
      obj.show();
      // Заполнение случайными числами:
      obj.rand();
      System.out.println("Случайные числа:");
      obj.show();
      System.out.println("После транспонирования:");
      // Транспонирование матрицы:
      obj.trans();
      obj.show();
      // Вычисление следа матрицы:
      System.out.println("След матрицы: "+obj.trace());
      // Вычисление определителя матрицы:
      System.out.println(
         "Определитель матрицы: "+obj.det()
      );
      // Новая матрица:
      obj.create(5);
      // Заполнение последовательностью цифр:
      obj.digits();
      System.out.println("Последовательность цифр:");
      obj.show();
      // Вычисление определителя матрицы:
      System.out.println(
         "Определитель матрицы: "+obj.det()
      );
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 4.10)

```
Единичная матрица:
1 0 0
0 1 0
0 0 1
Все элементы одинаковые:
5 5 5
5 5 5
5 5 5
Случайные числа:
6 2 8
3 1 5
3 5 7
После транспонирования:
6 3 3
2 1 5
8 5 7
След матрицы: 14
Определитель матрицы: -24
Последовательность цифр:
1 2 3 4 5
6 7 8 9 1
2 3 4 5 6
7 8 9 1 2
3 4 5 6 7
Определитель матрицы: 0
```

Основу программы составляет класс `Matrix`. Класс реализует квадратную матрицу: закрытое поле `matrix` является ссылкой на двумерный массив, который отождествляется с матрицей. Для удобства полагаем, что элементы матрицы — целые числа. Еще одно закрытое целочисленное поле `n` определяет размер квадратной матрицы (двумерного массива). Значения поля `n` и размер массива, на который ссылается поле `matrix`, должны, очевидно, изменяться синхронно. Поэтому данные поля закрыты.

Создание двумерного массива происходит при вызове метода `create()`. Метод имеет целочисленный аргумент, который определяет размер массива `matrix` по каждому из двух индексов.

ПОДРОБНОСТИ

Поскольку формальное название аргумента метода совпадает с полем класса `n`, присваивание значения полю выполняется командой `this.n=n` с явной ссылкой на объект класса `this`. Командой `matrix=new int[n][n]` создается двумерный целочисленный массив, а ссылка на этот массив в качестве значения присваивается полю `matrix`. Таким образом, только после вызова метода `create()` поле `matrix` объекта оказывается связанным с реальным двумерным массивом.

Несколько открытых методов класса позволяют заполнять матрицу числовыми значениями. В частности, метод `rand()` предназначен для заполнения матрицы случайными целыми числами. Для генерирования случайного целого числа предназначен метод `random()` из класса `Math`. Метод возвращает псевдослучайное действительное неотрицательное число, не превышающее единицу. Для вычисления на его основе целого числа использована команда `(int)(Math.random()*10)` — полученное в результате вызова метода `random()` случайное число умножается на `10`, после чего дробная часть отбрасывается (благодаря команде явного приведения типов).

Метод `digits()` предназначен для построчного циклического заполнения элементов матрицы последовательностью цифр от `1` и до `9`. Элементы массива `matrix` заполняются с помощью вложенных операторов цикла с использованием команды `matrix[i][j]=(i*n+j)%9+1`.

Метод `value()` позволяет всем элементам массива `matrix` присвоить одно и то же значение (аргумент метода). Вызов этого метода с нулевым аргументом происходит в теле метода `unit()`. В результате массив, на который ссылается поле `matrix`, заполняется нулями, после чего диагональным элементам (элементы с одинаковым значением индексов) присваиваются единичные значения — получается единичная матрица.

Метод `trace()` в качестве результата возвращает целое число, равное следу матрицы. След матрицы определяется как сумма всех диагональных элементов (элементов с одинаковыми индексами).

Метод `trans()` не имеет аргументов и не возвращает результат. Он используется для транспонирования матрицы. При выполнении метода перебираются элементы матрицы, размещенные над главной диагональю (второй индекс у соответствующих элементов массива больше первого индекса). При этом выполняется взаимный обмен значений элементов, расположенных симметрично относительно главной диагонали (у этих элементов индексы отличаются порядком следования).

Самый значительный и по объему кода, и по сложности метод `det()` предназначен для вычисления определителя матрицы. Основу метода составляет оператор выбора `switch()` — в нем проверяется значение поля `n` объекта, из которого вызывается метод. Выделяются три случая: когда значение поля `n` равно `1`, `2` и все прочие значения. Результат метода записывается в переменную `D` — именно эта переменная в итоге возвращается как результат метода.

Если поле `n` равно `1`, мы имеем дело фактически со скаляром (числом). В этом случае под определителем подразумевают сам скаляр. Поэтому для первого случая в операторе выбора переменной `D` присваивается значение `matrix[0][0]` — единственный элемент массива `matrix`.

Если размер матрицы по каждому из индексов равен `2`, то определитель вычисляется командой `D=matrix[0][0]*matrix[1][1]-matrix[0][1]*matrix[1][0]`. Здесь уместно напомнить, что для матрицы A размером 2 на 2 и элементами a_{ij} (индексы i,

$j = 1, 2$) определитель вычисляется как $\det(A) = a_{11}a_{22} - a_{12}a_{21}$, то есть как разность произведений диагональных и недиагональных элементов.

Если размер матрицы превышает значение 2, то определитель вычисляется по правилу Лапласа. В частности, если мы имеем дело с квадратной матрицей A размера n с элементами $i, j = 1, 2, ..., n$, то выражение для вычисления определителя матрицы можно записать в следующем виде:

$$\det(A) = \sum_{j=1}^{n} (-1)^{i+j} a_{ij} \det(M^{(ij)}).$$

Здесь матрица $M^{(ij)}$ получается из матрицы A вычеркиванием i-й строки и j-го столбца. Матрицы $M^{(ij)}$ являются квадратными матрицами размера $n - 1$. Таким образом, задача вычисления определителя матрицы размера n сводится к вычислению n определителей матриц размера $n - 1$. Эту процедуру можно продолжить и по той же схеме записать выражения для определителей матриц $M^{(ij)}$ через определители матриц размера $n - 2$ и так далее — пока все не сведется к вычислению определителя квадратной матрицы размера 2.

Данная схема реализована в блоке `default` оператора выбора `switch()`. Разложение всегда выполняется по первой строке, то есть в приведенной выше формуле значение индекса i зафиксировано на значении `1` (для массива `matrix` это означает, что первый индекс равен нулю).

В блоке `default` объявляются целочисленные индексные переменные `i`, `j` и `k`, а также переменная `sign` с начальным значением `1`. Переменная `sign` служит для запоминания знака (множитель `1` или `-1`), с которым соответствующее слагаемое входит в выражение для определителя (он поочередно меняется). Затем объявляется объектная переменная `m` класса `Matrix`. Эта переменная используется при вычислении определителей матриц меньшего ранга (размера), чем исходная. Основу блока `default` составляет оператор цикла. Индексная переменная `k` пробегает значения от `0` до `n-1`. В операторе цикла команда `m=new Matrix()` создает новый объект класса `Matrix`, и ссылка на этот объект присваивается в качестве значения переменной `m`. Команда `m.create(n-1)` ранг соответствующей матрицы устанавливает на единицу меньшим, чем ранг матрицы объекта, из которого вызывается метод. Затем запускаются вложенные операторы цикла для заполнения матрицы вновь созданного объекта. Элементы матрицы объекта `m` (то есть массив `m.matrix`) получаются из матрицы `matrix`, если вычеркнуть строку с нулевым индексом и столбец с индексом `k` (значение индексной переменной внешнего цикла).

Индексная переменная `i` первого оператора цикла пробегает значения от `1` до `n-1` и перебирает строки массива `matrix` (то есть массива `this.matrix`), используемые при заполнении массива `m.matrix`. При этом строки массивов смещены на одну позицию: строка с индексом `1` массива `matrix` служит для заполнения строки с индексом `0` массива `m.matrix`. На основе строки с индексом `2` массива `matrix` заполняется строка с индексом `1` массива `m.matrix`, и так далее. Что касается столбцов, то до `k`-го

имеет место однозначное соответствие: `j`-му столбцу массива `matrix` соответствует `j`-й столбец массива `m.matrix`. Далее, `k`-й столбец массива `matrix` в расчет не принимается, а начиная с `(k+1)`-го столбца, `j`-му столбцу массива `matrix` соответствует `(j-1)`-й столбец массива `m.matrix`. Перебор столбцов (при фиксированном индексе строки `i`) осуществляется в два этапа: переменная `j` сначала пробегает значения от `0` до `k-1`, а затем от `k+1` до `n-1`.

После заполнения массива `m.matrix` командой `D+=sign*matrix[0][k]*m.det()` изменяется значение переменной `D`. Эта команда содержит вызов метода `det()`, но уже из объекта `m`, а не из текущего объекта. Другими словами, в определении метода `det()` происходит вызов этого же метода, но из другого объекта. Таким образом, имеет место *рекурсия*. Кроме того, командой `sign*=(-1)` меняется значение используемой в качестве знакового множителя переменной `sign`.

НА ЗАМЕТКУ

Под рекурсией подразумевают ситуацию, когда метод в процессе выполнения вызывает сам себя.

Метод `show()` не имеет аргументов, не возвращает результат и предназначен для поэлементного отображения массива `matrix`. Для этого используются вложенные операторы цикла.

В главном методе программы в классе `Demo` командой `Matrix obj=new Matrix()` создается объект класса `Matrix`. Затем с этим объектом, с помощью разработанных методов, выполняются несложные манипуляции.

НА ЗАМЕТКУ

Желающие могут поэкспериментировать с методами класса `Matrix`. В частности, можно проверить, что происходит, если создается матрица размером 2 на 2 или матрица, состоящая из одного элемента.

Резюме

Хочешь поговорить — плати еще чатл.

из к/ф «Кин-дза-дза»

- Описание класса начинается с ключевого слова `class`, после чего следуют имя класса и, в фигурных скобках, описание класса. В состав класса могут входить поля и методы, которые называются членами класса. При описании поля указываются его тип и имя. При описании метода указывается тип результата, имя метода, в круглых скобках — список аргументов и в фигурных скобках — тело метода.

- Объекты создаются с помощью оператора `new`. После оператора `new` указывается имя класса и круглые скобки. Результатом такой инструкции является ссылка на созданный объект. Обычно она присваивается в качестве значения объектной переменной, которую отождествляют с объектом. Для объявления объектной переменной указывают имя соответствующего класса и имя этой переменной. На практике команды объявления объектной переменной и создания объекта (с присваиванием переменной ссылки на объект) объединяют в одну команду. Если ссылка на созданный объект не присваивается никакой объектной переменной, говорят об анонимном объекте.
- Обычные поля и методы существуют только в привязке к объекту. В отличие от них, статические члены класса существуют сами по себе и к конкретному объекту не привязаны. Статический член описывается в классе с идентификатором `static`.
- В описании членов класса могут использоваться спецификаторы доступа `public` (открытые члены), `private` (закрытые члены) и `protected` (защищенные члены). По умолчанию члены класса считаются открытыми — они доступны не только в самом классе, но и вне его.
- Доступ к членам класса осуществляется с использованием точечного синтаксиса: после имени объекта указывается, через точку, имя поля или метода этого объекта. Вне класса доступны только открытые члены. Доступ к статическим членам осуществляется через имя класса — оно указывается вместо имени объекта.
- Ключевое слово `this` является ссылкой на объект, из которого вызывается метод. Это ключевое слово используется при описании методов класса.
- В Java один класс может объявляться внутри другого класса. В этом случае говорят о внутреннем классе. Особенность внутреннего класса в том, что он имеет доступ к полям и методам содержащего его класса (внешнего класса). Напротив, члены внутреннего класса во внешнем классе недоступны.

5 Методы и конструкторы

— Героическая у вас работа! С вами будет спокойнее.

— За беспокойство не беспокойтесь.

из к/ф «Полосатый рейс»

В этой главе мы продолжим знакомство с принципами ООП и их реализацией в Java. Нас будет интересовать использование *методов* и *конструкторов*. А начнем мы с такого интересного механизма, как *перегрузка методов*.

Перегрузка методов

А это как бы премия оптовому покупателю от фирмы.

из к/ф «Полосатый рейс»

Поясним на простом примере необходимость в перегрузке методов. Допустим, имеется класс с двумя числовыми полями и методом, с помощью которого задаются значения этих полей. Метод имеет два аргумента — по одному для каждого поля. Мы хотим, чтобы в случае, если полям присваиваются одинаковые значения, можно было вызывать метод с одним аргументом. В отсутствие возможности перегрузить метод пришлось бы описывать новый. Это неудобно, поскольку для одного и того же по сути действия пришлось бы использовать два разных метода. Намного удобнее вызывать один и тот же метод, но с разным количеством аргументов в зависимости от ситуации. Благодаря перегрузке методов такая возможность существует.

При перегрузке методов создается несколько методов с одинаковыми именами, но разным количеством и/или типом аргументов. При вызове метода решение о том, какую его версию следует использовать, принимается на основе контекста команды вызова (в соответствии со списком аргументов, переданных методу).

НА ЗАМЕТКУ

Технически речь идет о разных методах, но поскольку все они имеют одинаковые названия, обычно говорят о разных версиях одного метода или вообще об одном методе.

Что касается функциональности различных версий перегруженного метода, то как таковых ограничений нет. Общепринятым является принцип, согласно которому перегруженные версии метода должны реализовывать один общий алгоритм. В этом смысле неприемлема ситуация, когда одна версия метода, например, отображает сообщение со значением аргумента, а другая — выполняет поиск наибольшего значения среди переданных методу аргументов. Хотя с технической точки зрения это возможно.

В листинге 5.1 представлен пример программы с перегруженным методом.

Листинг 5.1. Перегрузка метода

```
// Класс с перегруженным методом:
class MyClass{
   // Закрытые поля:
   private int first;
   private int second;
   // Метод с двумя аргументами
   // для присваивания значений полям:
   void set(int a,int b){
      first=a;
      second=b;
   }
   // Метод с одним аргументом
   // для присваивания значений полям:
   void set(int n){
      first=n;
      second=n;
   }
   // Метод без аргументов
   // для присваивания значений полям:
   void set(){
      first=100;
      second=200;
   }
   // Метод без аргументов для отображения
   // значений полей:
   void show(){
      System.out.println("Поле first: "+first);
      System.out.println("Поле second: "+second);
   }
   // Метод с одним логическим аргументом
   // для отображения значения поля:
   void show(boolean t){
      if(t){
         System.out.println("Поле first: "+first);
```

```
      }else{
         System.out.println("Поле second: "+second);
      }
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объекта:
      MyClass obj=new MyClass();
      // Операции с объектом:
      obj.set();
      obj.show();
      obj.set(111,222);
      obj.show(false);
      obj.show(true);
      obj.set(123);
      obj.show();
   }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 5.1)

```
Поле first: 100
Поле second: 200
Поле second: 222
Поле first: 111
Поле first: 123
Поле second: 123
```

В программе описывается класс `MyClass` с двумя закрытыми целочисленными полями (`first` и `second`). В классе описаны три версии метода `set()`, предназначенного для присваивания значений полям (без аргументов, с одним аргументом и с двумя аргументами). Также в классе описаны две версии метода `show()`. Если метод вызывается без аргументов, то он отображает значения полей объекта. Если метод вызывается с одним логическим аргументом (значение аргумента `true` или `false`), то отображается значение лишь одного поля (в зависимости от значения аргумента).

ПОДРОБНОСТИ

Если метод `set()` вызывается с двумя аргументами, то эти аргументы задают значения полей объекта, из которого вызывается метод. Если метод `set()` вызывается с одним аргументом, то аргумент задает значения обоих полей объекта. Если метод `set()` вызывается без аргументов, то первое поле объекта, из которого вызывается метод, получает значение `100`, а второе поле получает значение `200`.

В главном методе программы создается объект `obj` класса `MyClass`, и с его помощью иллюстрируется работа перегруженных методов.

Еще раз подчеркнем, что вызов нужного варианта метода осуществляется в зависимости от количества и типа аргументов, переданных методу при вызове. Поэтому перегрузка методов должна осуществляться так, чтобы по команде вызова можно было однозначно определить версию метода. Учитывая, что есть такой механизм, как автоматическое приведение типов, задача может быть нетривиальной. Пример приведен в листинге 5.2.

Листинг 5.2. Перегрузка метода и приведение типов

```
class MyClass{
   // Закрытые поля:
   private int number;
   private char symbol;
   // Метод с одним аргументом для присваивания
   // значения целочисленному полю:
   void set(int n){
      number=n;
   }
   // Метод с одним аргументом для присваивания
   // значения символьному полю:
   void set(char s){
      symbol=s;
   }
   // Метод с двумя аргументами для присваивания
   // значений полям:
   void set(int n,char s){
      // Вызов версии метода с одним аргументом:
      set(n);
      set(s);
   }
   // Метод для отображения значений полей:
   void show(){
      System.out.println("Поля объекта");
      System.out.println("Число:  "+number);
      System.out.println("Символ: "+symbol);
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объекта:
      MyClass obj=new MyClass();
      // Операции с объектом:
      obj.set(100,'A');
      obj.show();
      obj.set(200);
      obj.show();
      obj.set('B');
      obj.show();
      // Используется автоматическое
      // приведение типа:
      obj.set('A','D');
```

```
        obj.show();
    }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 5.2)

```
Поля объекта
Число: 100
Символ: A
Поля объекта
Число: 200
Символ: A
Поля объекта
Число: 200
Символ: B
Поля объекта
Число: 65
Символ: D
```

В классе `MyClass` описаны два закрытых поля (целочисленное `number` и символьное `symbol`). В классе есть метод `show()`, при вызове которого отображаются значения полей. Также в классе описаны три версии метода `set()`. Если метод вызывается с целочисленным аргументом, то он определяет значение целочисленного поля. Если метод вызывается с символьным аргументом, то он определяет значение символьного поля. При вызове метода с двумя аргументами (целочисленным и символьным) новое значение получают оба поля. Версия метода `set()` с двумя аргументами (целочисленным `n` и символьным `s`) описана так: в теле метода вызываются версии этого же метода с одним аргументом (команды `set(n)` и `set(s)`). Но необычного здесь ничего нет: то, что мы называем разными версиями метода, на самом деле разные методы. А вызов в теле одного метода другого метода — обычное дело.

НА ЗАМЕТКУ

Рекурсии в данном случае нет, поскольку одна версия метода вызывает не себя, а другую версию метода, то есть, по сути, вызывается другой метод.

В главном методе создается объект `obj` класса `MyClass` и затем из этого объекта вызывается метод `show()` и метод `set()` с разными аргументами. Нас интересует команда `obj.set('A','D')`, в которой метод `set()` вызывается с двумя целочисленными аргументами. Интрига в том, что мы не описывали версию метода `set()` с двумя целочисленными аргументами. Но команда рабочая, и объяснение кроется в автоматическом приведении типов. Поскольку версии метода с двумя аргументами нет, то начинается поиск версии, которая подошла бы, если выполнить автоматическое приведение типов. В данном случае первый аргумент `'A'` типа `char` автоматически трансформируется в тип `int` (получается код `65` символа `'A'`), и на самом деле вызывается версия метода `set()` с двумя аргументами: первый `65`, а второй `'D'`.

НА ЗАМЕТКУ

Если мы попытаемся вызвать метод `set()` с двумя целочисленными аргументами, то возникнет ошибка. Причина в том, что в Java есть автоматическое приведение типа `char` в тип `int`, но автоматического приведения типа `int` в тип `char` нет. А поскольку мы не описали версию метода `set()` с двумя целочисленными аргументами, то получается, что мы вызываем несуществующую версию метода.

Более того, если мы опишем в классе `MyClass` версию метода `set()` с символьным и целочисленным аргументами (то есть первый аргумент — символьный, а второй — целочисленный), то команда `obj.set('A','D')` станет некорректной. Причина в том, что для команды `obj.set('A','D')` будет невозможно определить, какая из двух версий метода (с двумя аргументами) должна вызываться (потому что формально подходят обе).

Конструктор

Какие мы все хорошие! Давайте все дружить, а?

из к/ф «Айболит-66»

Конструктор — это метод, который вызывается автоматически при создании объекта. Кроме того что конструктор вызывается автоматически, от обычного метода, объявленного в классе, конструктор отличается тем, что:

- имя конструктора совпадает с именем класса;
- конструктор не возвращает результат, и идентификатор типа для конструктора не указывается;
- как и обычный метод, конструктор может иметь аргументы и его можно перегружать.

До этого мы использовали классы, в которых конструкторов не было (мы их там не описывали). В таких случаях (то есть когда конструктор в классе явно не описан) используется *конструктор по умолчанию*. Конструктор по умолчанию не имеет аргументов, и при его вызове никакие дополнительные действия (кроме создания объекта) не выполняются. Именно этот конструктор вызывался в рассмотренных ранее примерах при создании объектов класса.

НА ЗАМЕТКУ

Конструкторы можно перегружать, поэтому в классе может быть описано несколько версий конструктора. Если хотя бы один конструктор в классе описан, то конструктор по умолчанию больше не доступен.

В листинге 5.3 приведен пример кода с классом, в котором описан конструктор.

Листинг 5.3. Класс с конструктором

```
class MyClass{
   // Поля класса:
   int number;
   char symbol;
   // Метод для отображения значений полей:
   void show(){
      System.out.println("Число: "+number);
      System.out.println("Символ: "+symbol);
   }
   // Конструктор:
   MyClass(){
      // Присваивание значений полям:
      number=100;
      symbol='A';
      // Отображение значения полей:
      show();
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      MyClass obj;
      System.out.println("Создается объект");
      obj=new MyClass();
   }
}
```

Результат выполнения программы будет таким:

Результат выполнения программы (из листинга 5.3)

```
Создается объект
Число: 100
Символ: A
```

В данном случае основной интерес вызывает конструктор. Имя конструктора `MyClass()` совпадает с именем класса `MyClass`. Тип результата для конструктора не указывается. Аргументы конструктору также не передаются — после имени конструктора идут пустые круглые скобки. Код в фигурных скобках — это тело конструктора, полям `number` и `symbol` в котором присваиваются значения соответственно `100` и `'A'`, а затем вызывается метод `show()`, который отображает значения полей. Таким образом, каждый раз при создании объекта поля этого объекта автоматически получают значения `100` и `'A'`, а в области вывода отображается сообщение со значениями полей созданного объекта. Именно это и происходит при создании объекта класса `MyClass` в главном методе программы.

Как и обычному методу, конструктору можно передавать аргументы. В таком случае аргументы, предназначенные для конструктора, указываются в круглых скобках

после имени класса в команде создания объекта. Если в классе описано несколько конструкторов, то решение об использовании той или иной версии конструктора принимается исходя из типа и количества аргументов, переданных конструктору. Пример программы с использованием нескольких конструкторов представлен в листинге 5.4.

Листинг 5.4. Перегрузка конструктора

```
class MyClass{
   int number;
   char symbol;
   void show(){
      System.out.println("Число:  "+number);
      System.out.println("Символ: "+symbol);
   }
   // Конструктор с двумя аргументами:
   MyClass(int n,char s){
      System.out.println("Создается объект");
      // Полям присваиваются значения:
      number=n;
      symbol=s;
   }
   // Конструктор без аргументов:
   MyClass(){
      // Вызов конструктора с двумя аргументами:
      this(100,'A');
      System.out.println("Объект создан");
   }
   // Конструктор с целочисленным аргументом:
   MyClass(int n){
      // Вызов конструктора с двумя аргументами:
      this(n,'B');
   }
   // Конструктор с символьным аргументом:
   MyClass(char s){
      System.out.println("Новый объект");
      // Полям присваиваются значения:
      number=300;
      symbol=s;
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      MyClass A=new MyClass();
      A.show();
      MyClass B=new MyClass(200);
      B.show();
      MyClass C=new MyClass('C');
      C.show();
      MyClass D=new MyClass(400,'D');
      D.show();
   }
}
```

При выполнении программы получаем результат:

Результат выполнения программы (из листинга 5.4)

```
Создается объект
Объект создан
Число:  100
Символ: A
Создается объект
Число:  200
Символ: B
Новый объект
Число:  300
Символ: C
Создается объект
Число:  400
Символ: D
```

В классе `MyClass` описаны четыре версии конструктора: без аргументов, с двумя аргументами и две версии конструктора с одним аргументом (символьным или целочисленным). Код конструкторов обычно устроен просто: полям объекта присваиваются значения, а затем отображаются некоторые сообщения. Но используется и новый для нас прием: мы задействовали инструкцию `this` с круглыми скобками и аргументами для вызова в одной версии конструктора другой версии конструктора. Например, в теле конструктора без аргументов команда `this(100,'A')` вызывает версию конструктора с двумя аргументами (`100` и `'A'`).

ПОДРОБНОСТИ

Ранее мы использовали ключевое слово `this` в описании методов как ссылку на объект, из которого вызывается метод. Если в теле конструктора после ключевого слова `this` указать круглые скобки (с аргументами или без), то это будет инструкция вызова соответствующей версии конструктора. Другими словами, у нас есть возможность в одной версии конструктора вызвать другую версию конструктора. Вызываемая версия определяется по количеству и типу аргументов, указанных после ключевого слова `this`. Эта инструкция может быть не единственной в теле конструктора, но должна быть там первой командой.

Во всем остальном — без сюрпризов.

НА ЗАМЕТКУ

Если бы в классе `MyClass` мы не описали версию конструктора без аргументов, то у нас бы не было возможности создавать объекты без передачи аргументов конструктору. Причина в том, что как только в классе описана хотя бы одна версия конструктора, конструктор по умолчанию больше не доступен.

Объект как аргумент и результат метода

Эх, люблю подлить масла в огонь!

из к/ф «Айболит-66»

Для передачи объекта методу используют объектную переменную (значением которой является ссылка на объект). Если метод возвращает объект, то речь идет о том, что возвращается ссылка на объект.

ПОДРОБНОСТИ

Другими словами, если объект нужно передать для обработки в метод, то на самом деле в метод передается адрес этого объекта. Адрес объекта содержится в объектной переменной, которая ссылается на объект. Аналогично, если есть некоторый объект, который метод должен вернуть результатом, то достаточно вернуть адрес этого объекта. Если такой адрес записать в объектную переменную, получим доступ к объекту. Объект, ссылка на который возвращается методом, обычно создается в процессе работы метода.

Формально в случае, если аргументом методу передается объект, то соответствующий аргумент описывается как объектная переменная: типом аргумента указывается имя класса, а при вызове метода соответствующим аргументом методу передается переменная объекта.

Если результатом метода является объект (точнее, ссылка на объект), то в качестве идентификатора типа результата указывается соответствующий класс. Результат вызова такого метода можно записать в объектную переменную, класс которой соответствует классу объекта.

В листинге 5.5 приведен пример кода, в котором иллюстрируется передача объектов аргументами методам, а также показано, как объект может возвращаться методом.

Листинг 5.5. Объект как аргумент и результат метода

```
// Класс с методами:
class MyClass{
   // Целочисленное поле:
   int code;
   // Метод для отображения значения поля:
   void show(){
      System.out.println("Поле: "+code);
   }
   // Конструктор с одним аргументом:
   MyClass(int n){
      code=n;
   }
   // Конструктор создания копии:
```

```
   MyClass(MyClass obj){
      code=obj.code;
   }
   // Результат метода и аргумент – объекты:
   MyClass get(MyClass obj){
      // Создание локального объекта:
      MyClass tmp=new MyClass(code);
      // Уточнение значения поля:
      tmp.code+=obj.code;
      // Результат метода:
      return tmp;
   }
}
// Главный класс:
class Demo{
   // Метод для создания объекта:
   static MyClass create(int n){
      // Локальный объект:
      MyClass tmp=new MyClass(n);
      // Результат метода:
      return tmp;
   }
   // Главный метод:
   public static void main(String[] args){
      // Создание объектов:
      MyClass A=new MyClass(100);
      MyClass B=new MyClass(A);
      MyClass C=create(200);
      MyClass D=A.get(C);
      // Изменение значения поля:
      A.code--;
      // Отображение значения поля code:
      A.show();
      B.show();
      C.show();
      D.show();
   }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 5.5)

```
Поле: 99
Поле: 100
Поле: 200
Поле: 300
```

В классе `MyClass` есть целочисленное поле `code`, метод `show()`, предназначенный для отображения значения поля, а также две версии конструктора: с целочисленным аргументом и с аргументом, являющимся объектом касса `MyClass`. Если кон-

структору аргументом передается целое число, то оно становится значением поля объекта. Если аргументом конструктору передается объект класса `MyClass`, то поле `code` создаваемого объекта получает такое же значение, что и у поля `code` объекта, переданного аргументом конструктору.

НА ЗАМЕТКУ

В этом случае создаваемый объект является копией объекта, переданного аргументом конструктору.

Однако у класса `MyClass` есть еще один поучительный метод, у которого аргументом является объект класса `MyClass`, а результатом — объект класса `MyClass`. Речь о методе `get()`, в теле которого командой `MyClass tmp=new MyClass(code)` создается локальный объект `tmp` и у поля `code` которого значение такое же, как и у объекта, из которого вызывается метод. Затем командой `tmp.code+=obj.code` это значение увеличивается на значение поля `code` объекта `obj`, переданного аргументом методу. После этого командой `return tmp` ссылка на созданный объект возвращается результатом метода.

НА ЗАМЕТКУ

Метод `get()` вызывается из объекта, и аргументом ему передается объект. Результатом является еще один объект, причем значение поля `code` этого объекта равно сумме значений полей двух первых объектов (того, из которого вызывался метод, и того, который передан методу аргументом).

В главном классе `Demo`, кроме главного метода, описан еще и статический метод `create()`. Аргументом методу передается целое число, а метод возвращает ссылку на новый созданный объект. Значение поля `code` этого объекта равно значению аргумента метода.

В главном методе программы разными способами создается несколько объектов. При создании объекта `A` вызывается конструктор с целочисленным аргументом. Объект `B` создается (на основе объекта `A`) вызовом конструктора создания копии.

НА ЗАМЕТКУ

Поскольку объект `B` является копией объекта `A`, то при изменении значения поля `code` объекта `A` (например, командой `A.code--`) значение поля `code` объекта `B` не меняется.

Для создания объекта `C` вызывается метод `create()` с аргументом `200` (значение поля `code` объекта `C`). Объект `D` создается командой `A.get(C)`, поэтому значение поля `code` объекта `D` равно сумме значений полей объектов `A` и `C`.

Механизм передачи аргументов

> Ребят, как же это вы без гравицапы пепелац выкатываете из гаража? Это непорядок.
>
> *из к/ф «Кин-дза-дза»*

В Java при передаче аргументов методам на самом деле передаются не те переменные, которые фактически указаны аргументами в команде вызова метода, а их копии (это называется передачей аргументов *по значению*). Копии создаются автоматически. После того как метод завершит свою работу, безымянные копии аргументов автоматически удаляются. Если мы не планируем изменять значения аргументов метода в процессе его выполнения, то механизм передачи аргументов не столь важен. Но в некоторых случаях его все же нужно принимать во внимание. В листинге 5.6 приведен пример программы, в которой иллюстрируются особенности механизма передачи аргументов методам.

Листинг 5.6. Механизм передачи аргументов методам

```
// Класс с полем:
class MyClass{
   int code;
}
// Главный класс:
class Demo{
   // Перегруженный метод:
   static void change(int n){
      System.out.println(
         "Исходное значение переменной: "+n
      );
      // Попытка изменить значение аргумента:
      n++;
      System.out.println(
         "Конечное значение переменной: "+n
      );
   }
   static void change(MyClass obj){
      System.out.println(
         "Исходное значение поля: "+obj.code
      );
      // Попытка изменить значение поля:
      obj.code++;
      System.out.println(
         "Конечное значение поля: "+obj.code
      );
   }
   // Главный метод:
   public static void main(String[] args){
      // Переменная:
      int code=100;
```

```
      // Объект:
      MyClass obj=new MyClass();
      obj.code=200;
      // Попытка изменить аргумент метода:
      System.out.println("До вызова метода: code="+code);
      change(code);
      System.out.println(
         "После вызова метода: code="+code
      );
      System.out.println(
         "До вызова метода: obj.code="+obj.code
      );
      change(obj);
      System.out.println(
         "После вызова метода: obj.code="+obj.code
      );
      System.out.println("Еще одна попытка");
      change(obj.code);
      System.out.println(
         "После вызова метода: obj.code="+obj.code
      );
   }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 5.6)

```
До вызова метода: code=100
Исходное значение переменной: 100
Конечное значение переменной: 101
После вызова метода: code=100
До вызова метода: obj.code=200
Исходное значение поля: 200
Конечное значение поля: 201
После вызова метода: obj.code=201
Еще одна попытка
Исходное значение переменной: 201
Конечное значение переменной: 202
После вызова метода: obj.code=201
```

Мы описали в программе очень простой класс `MyClass`, у которого всего одно целочисленное поле `code`. В главном классе описан перегруженный статический метод `change()`, которому аргументом можно передавать целое число или объект класса `MyClass`. Если аргументом методу передается числовое значение, то в теле метода отображается исходное значение аргумента, затем предпринимается попытка увеличить значение аргумента на единицу, и снова отображается значение аргумента. Если аргументом методу передается объект класса `MyClass`, то все перечисленные операции выполняются с полем `code` объекта.

В главном методе программы объявляется целочисленная переменная `code` со значением `100`, а также создается объект `obj` класса `MyClass`, и его полю `code` присваивает-

ся значение 200. Далее проверяется значение переменной code, которая передается аргументом методу change(), после чего снова проверяется значение переменной. То же происходит с полем obj.code и с объектом obj (в последнем случае проверяется значение поля code объекта). О чем говорят результаты выполнения программы? Начнем с переменной code. В теле метода change() проверка значения аргумента после его изменения свидетельствует о том, что значение аргумента увеличилось на единицу. Но при проверке значения переменной после завершения работы метода оказывается, что значение переменной не изменилось. Почему? Ответ связан с механизмом передачи аргументов. Когда переменная code передается аргументом методу change(), то на самом деле метод получает не оригинал, а копию переменной. Именно значение копии увеличивается на единицу, а когда метод завершает работу, копия удаляется. С переменной code ничего не происходит.

После передачи аргументом методу change() поля obj.code происходит то же самое — для поля создается копия, значение которой увеличивается. Само поле при этом остается неизменным. Но если передать методу change() сам объект obj, а не его поле, то значение поля code объекта obj увеличится на единицу, поскольку и в этом случае для аргумента создается копия объектной переменной obj. Значением объектной переменной является адрес объекта. У копии значение — такое же, как у оригинала. Поэтому копия переменной obj ссылается на тот же объект, что и переменная obj. При получении через копию доступа к полю code и изменении его значения изменяется поле именно того объекта, на который ссылается переменная obj.

Применение методов на практике

> Показывай свою гравицапу. Если фирменная вещь — возьмем.
>
> *из к/ф «Кин-дза-дза»*

Далее рассматриваются примеры, в которых используются конструкторы и методы.

Интерполяционный полином

Важной задачей прикладного числового анализа является проблема интерполяции и аппроксимации зависимостей, заданных в табулированном виде, то есть в виде массивов узловых точек и значений некоторой, обычно неизвестной, функции в этих точках. Рассмотрим пример, позволяющий получить общее представление о способах решения этой задачи методами объектно-ориентированного программирования. В программе из листинга 5.7 создается класс, содержащий двумерный массив со значениями узловых точек аргумента и значениями функции в этих узловых точках. В классе описывается несколько методов, позволяющих вычис-

лять на основе упомянутых данных интерполяционный полином, а также строить простую регрессионную модель.

Листинг 5.7. Интерполяция и аппроксимация

```
class Optimizer{
   // Размер массива:
   private int n;
   // Параметры регрессионной модели:
   private double a,b;
   // Массив данных:
   double[][] data;
   // Метод для вычисления значений базовых функций
   // в схеме Лагранжа:
   private double psi(int k,double x){
      int i;
      double s=1;
      for(i=0;i<k;i++){
         s*=(x-data[0][i])/(data[0][k]-data[0][i]);
      }
      for(i=k+1;i<=n;i++){
         s*=(x-data[0][i])/(data[0][k]-data[0][i]);
      }
      return s;
   }
   // Метод для вычисления параметров
   // регрессионной модели:
   private void set(){
      double Sxy=0,Sx=0,Sy=0,Sxx=0;
      for(int i=0;i<=n;i++){
         Sx+=data[0][i];
         Sy+=data[1][i];
         Sxx+=data[0][i]*data[0][i];
         Sxy+=data[0][i]*data[1][i];
      }
      a=((n+1)*Sxy-Sx*Sy)/((n+1)*Sxx-Sx*Sx);
      b=Sy/(n+1)-a/(n+1)*Sx;
   }
   // Метод для вычисления значения
   // регрессионной функции:
   double approx(double x){
      return a*x+b;
   }
   // Метод для вычисления значения
   // интерполяционного полинома:
   double interp(double x){
      double s=0;
      for(int i=0;i<=n;i++){
         s+=data[1][i]*psi(i,x);
      }
      return s;
   }
   // Конструктор:
```

```
   Optimizer(int n){
      this.n=n;
      data=new double[2][n+1];
      for(int i=0;i<=n;i++){
         data[0][i]=Math.PI*i/n/2;
         data[1][i]=Math.sin(data[0][i]);
      }
      set();
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      double x;
      int i,n=4,N=2*n;
      Optimizer obj=new Optimizer(n);
      System.out.printf("%60s","Таблица значений:\n");
      System.out.printf("%25s","Аргумент x");
      System.out.printf("%25s","Функция y=sin(x)");
      System.out.printf("%25s","Интерп. полином L(x)");
      System.out.printf("%25s","Регр. функция f(x)\n");
      for(i=0;i<=N;i++){
         x=i*Math.PI/N/2;
         System.out.printf("%25s",x);
         System.out.printf("%25s",Math.sin(x));
         System.out.printf("%25s",obj.interp(x));
         System.out.printf("%25s",obj.approx(x)+"\n");
      }
   }
}
```

Приведем результат выполнения программы:

Результат выполнения программы (из листинга 5.7)

```
Таблица значений:
         Аргумент x      Функция y=sin(x)  Интерп. полином L(x)  Регр. функция f(x)
                0.0                   0.0                   0.0  0.09449472918334523
0.19634954084936207   0.19509032201612825  0.19489713092476776  0.22155453419065513
0.39269908169872414    0.3826834323650898   0.3826834323650898  0.34861433919796503
 0.5890486225480862    0.5555702330196022   0.5556486374880385  0.47567414420527493
 0.7853981633974483    0.7071067811865475   0.7071067811865475   0.6027339492125848
 0.9817477042468103    0.8314696123025452   0.8313962000794116   0.7297937542198947
 1.1780972450961724    0.9238795325112867   0.9238795325112867   0.8568535592272046
 1.3744467859455345    0.9807852804032304   0.9809437185526901   0.9839133642345145
 1.5707963267948966                   1.0                   1.0   1.1109731692418245
```

Основу программы составляет класс `Optimizer`, который содержит двумерный массив с исходными данными для интерполяции и аппроксимации, методы для выполнения соответствующих оценок, а также ряд вспомогательных полей и методов. Узловые точки и значения табулируемой в этих точках функции при создании объекта класса `Optimizer` заносятся в двумерный массив, ссылка на который со-

держится в поле `data`. По первому индексу массив имеет размер `2` (индекс `0` соответствует узловым точкам, индекс `1` — значениям табулируемой функции в этих точках), а по второму индексу количество элементов определяется закрытым полем `n` (массив содержит `n+1` элемент). Второй индекс определяет номер узловой точки. Конструктору класса `Optimizer` аргументом передается значение поля `n`. После присвоения полю `n` значения командой `data=new double[2][n+1]` создается массив, ссылка на который записывается в поле `data`, и начинается заполнение массива. В операторе цикла индексная переменная `i` принимает значения от `0` до `n+1` включительно. Команды `data[0][i]=Math.PI*i/n/2` и `data[1][i]=Math.sin(data[0][i])` присваивают значения элементам массива исходных данных. Несложно догадаться, что в данном случае табулируется функция $y(x) = \sin(x)$ на интервале значений аргумента $0 \le x \le \pi/2$.

После заполнения массива вызывается метод `set()`. Это закрытый метод, позволяющий на основе данных из массива `data` вычислить параметры линейной регрессионной модели. Эти параметры записываются в закрытые поля `a` и `b`. Чтобы разобраться в методике расчета параметров, кратко остановимся на способах построения линейной регрессионной модели.

Предположим, необходимо выполнить аппроксимацию по наборам данных $\{x_k\}$ и $\{y_k\}$ (индекс $k = 0, 1, 2, ..., n$), которые представляют собой соответственно значения аргумента x в узловых точках и некоторой функции $y(x)$ в этих же точках (то есть по определению $y(x_k) = y_k$). Задача состоит в нахождении параметров a и b, при которых функция $f(x) = ax + b$ (регрессионная функция) наилучшим образом описывала бы эту зависимость. Для определения этого «наилучшего образа» обычно используют метод наименьших квадратов, в соответствии с которым указанные параметры вычисляются так, чтобы сумма квадратов отклонений табличных значений функции y_k и регрессионной функции $f(x_k)$ в узловых точках была минимальна. Другими словами, необходимо найти такие значения параметров a и b, чтобы была минимальной сумма

$$\sum_{k=0}^{n}(y_k - f(x_k))^2 = \sum_{k=0}^{n}(y_k - ax_k - b)^2.$$

Легко показать, что в этом случае параметры a и b вычисляются следующим образом:

$$a = \frac{(n+1)\sum_{k=0}^{n} x_k y_k - \sum_{k=0}^{n} x_k \sum_{k=0}^{n} y_k}{(n+1)\sum_{k=0}^{n} x_k^2 - \left(\sum_{k=0}^{n} x_k\right)^2},$$

$$b = \frac{1}{n+1}\sum_{k=0}^{n} y_k - \frac{a}{n+1}\sum_{k=0}^{n} x_k.$$

В методе `set()` в операторе цикла вычисляются суммы $\sum_{k=0}^{n} x_k$ (переменная `Sx`), $\sum_{k=0}^{n} y_k$ (переменная `Sy`), $\sum_{k=0}^{n} x_k y_k$ (переменная `Sxy`) и $\sum_{k=0}^{n} x_k^2$ (переменная `Sxx`), а затем на основе вычисленных значений определяются параметры регрессионной модели. Метод `approx()`, предназначенный для вычисления значения регрессионной функции, достаточно прост и особых комментариев не требует. Несколько сложнее вычисляется интерполяционный полином в методе `interp()`. При вычислении значения интерполяционного полинома вызывается закрытый метод `psi()`, предназначенный для расчета базисных функций интерполяционного полинома в схеме Лагранжа. Вот краткое изложение использованного подхода.

Для создания интерполяционного полинома на основе наборов данных $\{x_k\}$ и $\{y_k\}$ (индекс $k = 0, 1, 2, ..., n$) необходимо вычислить параметры (коэффициенты) полинома $L(x)$ степени n по аргументу x такого, чтобы в узловых точках значения полинома совпадали со значениями табулированной функции. Должно выполняться условие $L(x_k) = y_k$ для всех $k = 0, 1, 2, ..., n$. При создании интерполяционного полинома по схеме Лагранжа соответствующее полиномиальное выражение находится в виде

$$L(x) = \sum_{k=0}^{n} y_k \psi_k(x).$$

Базисные функции здесь такие:

$$\psi_k(x) = \prod_{\substack{m=0,\\ m\neq k}}^{n} \frac{x - x_k}{x_m - x_k}.$$

Особенность базисной функции состоит в том, что $\psi_k(x_m) = \delta_{km}$, где символ Кронекера δ_{km} равен нулю для разных индексов и единице — для одинаковых.

Именно схема Лагранжа реализована при вычислении интерполяционного полинома. Значение базисной функции вычисляется методом `psi()`, а непосредственно значение полинома — методом `interp()`.

Что касается главного метода программы, то в нем создается объект класса `Optimizer`. Затем на основе этого объекта для нескольких значений аргумента отображается: непосредственно аргумент, значение исходной функции в соответствующей точке, значения в этой точке интерполяционного полинома и регрессионной функции. Данные отображаются в виде импровизированной таблицы.

НА ЗАМЕТКУ

Для отображения данных использован метод `printf()`, позволяющий задавать способ форматирования. Первый аргумент этого метода является текстовой строкой. Инструкция `"%25s"` означает, что для вывода данных (в данном случае текстового представления числа) используется 25 позиций.

Что касается непосредственно результатов выполнения программы, то получено довольно неплохое совпадение для синуса, рассчитанного на основе встроенной функции и интерполяционного полинома. Причина в том, что на выбранном интервале исходная табулированная функция гладкая, поэтому интерполяция эффективна. Хуже результаты для аппроксимации, поскольку в данном случае линейная регрессия не является оптимальной для описания исходной функциональной зависимости.

Геометрические фигуры

В листинге 5.8 приведен код программы, в которой реализован класс `Figures` для работы с графическими объектами на плоскости (речь в данном случае — о правильных многоугольниках). Класс содержит методы для определения координат вершин фигур и вычисления периметра и площади.

В классе `Figures` целочисленное закрытое поле `n` предназначено для записи количества вершин. Многоугольник определяется фактически набором точек на плоскости. Для реализации объекта «точка» в классе `Figures` описывается внутренний класс `Point`, имеющий символьное (тип `char`) поле `name` для записи названия точки (латинская буква). Поля `x` и `y` типа `double` предназначены для записи и хранения координат точки.

Конструктору внутреннего класса в качестве аргументов передается символ (буква) названия точки, а также две ее координаты. Метод `dist()` в качестве результата возвращает расстояние от начала координат до точки, реализованной через объект, из которого вызывается метод. Расстояние вычисляется как корень квадратный из суммы квадратов координат точки. Наконец, методом `show()` отображается название точки с ее координатами (в круглых скобках). При отображении координат точки в десятичной части остается не более двух цифр. Для округления используется метод `round()` из класса `Math`.

Идея, реализованная в программе, такова. На основе начальной точки создается правильный многоугольник с указанным количеством вершин. Конструктор класса имеет три аргумента: количество вершин многоугольника и координаты первой точки. Прочие точки находятся на таком же расстоянии от начала координат, что и первая точка. Каждая следующая получается смещением точки против часовой стрелки на один и тот же угол. Каждая точка — объект класса `Point`. Ссылки на эти объекты записываются в закрытое поле `points`. Поле `points` объявляется как переменная массива, элементами которого являются объектные переменные класса `Point` (соответствующая инструкция имеет вид `Point[] points`). Размер массива определяется значением поля `n`. Поля `n` и `points` объявлены как закрытые для предотвращения их несанкционированного или несинхронного изменения.

В классе также описан метод `perimeter()` для вычисления периметра и метод `square()` для вычисления площади многоугольника. Методу `dist()`, описанному в классе `Figures`, аргументами передаются два объекта класса `Point`, а результатом

метод возвращает расстояние между соответствующими точками (вычисляется как корень квадратный из суммы квадратов разностей соответствующих координат точек).

Все основные вычисления происходят при вызове конструктора. В первую очередь там создается начальная точка (объект `p` класса `Point`). Для этого используется команда `Point p=new Point('A',x,y)`. Первая точка имеет имя `A`, а ее координаты определяются вторым и третьим аргументами конструктора класса `Figures`. Командой `this.n=n` полю `n` класса присваивается значение, переданное первым аргументом конструктору. После этого командой `points=new Point[n]` создается массив для записи ссылок на объекты точек. Угол `phi0` на начальную точку и расстояние `r` до нее вычисляются соответственно командами `phi0=Math.atan2(y,x)` и `r=p.dist()`. Напомним, что методом `atan2()` (из класса `Math`), если он вызван с аргументами `y` и `x`, возвращается угол на точку с координатами `x` и `y`. Угол поворота определяется как `phi=2*Math.PI/n`. Далее для расчета следующих точек запускается оператор цикла цикл. Одновременно с определением имен и координат точек методом `show()` осуществляется отображение данных в области вывода. Каждая новая вершина (ссылка на соответствующий объект) записывается в переменную `p`. Предварительно старая ссылка заносится в массив, на который ссылается поле `points`. При вычислении имени новой точки к имени старой точки (ссылка `p.name`) прибавляется единица. Результатом является код следующего после `p.name` символа. Это значение преобразуется к типу `char` с помощью явного приведения типов. В результате получаем букву, следующую после буквы `p.name`. При вычислении координат вершины использовано то свойство, что точка, находящаяся на расстоянии r от начала координат в направлении угла α, имеет координаты $x = r\cos(\alpha)$ и $y = r\sin(\alpha)$. В то же время для k-й точки (по порядку, а не по индексу) угол определяется как $\alpha = \varphi_0 + (k-1)\,\varphi$, где φ_0 — угол в направлении на начальную точку, а $\varphi = 2\pi/n$ — угол поворота. Именно эти соотношения использованы при вычислении вершин многоугольника. После вычисления вершин многоугольника вычисляются и отображаются значения для периметра и площади многоугольника.

Листинг 5.8. Геометрические фигуры

```
class Figures{
   // Количество вершин:
   private int n;
   // Точки (вершины):
   private Point[] points;
   // Конструктор класса:
   Figures(int n,double x,double y){
      // Угол на начальную точку, угол приращения
      // и расстояние до начальной точки:
      double phi0,phi,r;
      // Начальная точка:
      Point p=new Point('A',x,y);
      // Индексная переменная:
      int i;
      // Значение для количества вершин:
```

```
   this.n=n;
   // Массив объектных переменных для "точек":
   points=new Point[n];
   // Вычисление угла на начальную точку:
   phi0=Math.atan2(y,x);
   // Вычисление угла приращения:
   phi=2*Math.PI/n;
   // Расстояние от начала координат
   // до начальной точки:
   r=p.dist();
   System.out.print(
      "Правильный "+n+"-угольник с вершинами в точках "
   );
   // Заполнение массива "точек"
   // и отображение результата:
   for(i=0;i<n-1;i++){
      p.show();
      System.out.print(i==n-2?" и ":", ");
      points[i]=p;
      // "Вычисление" вершин:
      p=new Point(
         (char)(p.name+1),
         r*Math.cos(phi0+(i+1)*phi),
         r*Math.sin(phi0+(i+1)*phi)
      );
   }
   // "Последняя" вершина:
   points[n-1]=p;
   p.show();
   System.out.println();
   // Периметр фигуры:
   System.out.println("Периметр:\t"+perimeter());
   // Площадь фигуры:
   System.out.println("Площадь:\t"+square());
}
// Метод для вычисления расстояния между точками:
double dist(Point A,Point B){
   return Math.sqrt(
      (A.x-B.x)*(A.x-B.x)+(A.y-B.y)*(A.y-B.y)
   );
}
// Метод для вычисления периметра:
double perimeter(){
   double P=0;
   int i;
   for(i=0;i<n-1;i++){
      P+=dist(points[i],points[i+1]);
   }
   P+=dist(points[n-1],points[0]);
   return P;
}
// Метод для вычисления площади:
double square(){
```

```
      double r=points[0].dist();
      double phi=2*Math.PI/n;
      double s=r*r*Math.sin(phi)/2;
      return s*n;
   }
   // Внутренний класс для "точек":
   class Point{
      // Название точки:
      char name;
      // Координаты точки:
      double x,y;
      // Конструктор внутреннего класса:
      Point(char name,double x,double y){
         // Название точки:
         this.name=name;
         // Координаты точки:
         this.x=x;
         this.y=y;
      }
      // Метод для вычисления расстояния
      // от начала координат до точки:
      double dist(){
         return Math.sqrt(x*x+y*y);
      }
      // Метод для отображения названия точки
      // и ее координат:
      void show(){
         System.out.print(
            name+"("+Math.round(x*100)/100.0+
            ","+Math.round(y*100)/100.0+")"
         );
      }
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание квадрата:
      new Figures(4,1,1);
   }
}
```

Создание многоугольника означает создание объекта класса `Figures`: в главном методе создается анонимный объект этого класса. Результат выполнения программы будет иметь следующий вид:

Результат выполнения программы (из листинга 5.8)

```
Правильный 4-угольник с вершинами в точках A(1.0,1.0), B(-1.0,1.0), C(-1.0,-1.0)
и D(1.0,-1.0)
Периметр:     8.0
Площадь:      4.000000000000001
```

Стоит заметить, что для правильного многоугольника периметр и площадь можно вычислить на основе информации о количестве вершин и расстоянии от начала координат до первой точки. Здесь вычисления в иллюстративных целях проводились, что называется, в лоб. Например, для расчета периметра вычислялись расстояния между соседними точками, причем для последней точки соседней является первая точка. При расчете площади вычислялась площадь одного сегмента (треугольник с вершиной в начале координат и основанием — отрезком, соединяющим соседние вершины многоугольника), а затем полученное значение умножалось на количество таких сегментов (равное количеству вершин).

Матричная экспонента

В следующем примере представлена программа, с помощью которой вычисляется матричная экспонента. Известно, что экспоненциальная функция от действительного аргумента x вычисляется в виде ряда

$$\exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!}.$$

Если речь идет о матричной экспоненте, то ее можно вычислять по тому же принципу. В частности, если A — некоторая квадратная матрица, то по определению матричной экспонентой от этой матрицы называется матрица $\exp(A)$, которая вычисляется так:

$$\exp(A) = \sum_{k=0}^{\infty} \frac{A^k}{k!}.$$

Здесь опять же по определению полагают, что матрица A в нулевой степени равняется единичной матрице E (по диагонали единицы, все остальные элементы — нули), то есть $A^0 = E$.

На практике вместо вычисления бесконечного ряда для матричной экспоненты ограничиваются конечным числом слагаемых. Что касается непосредственно ряда, то для его вычисления нужно уметь проделывать как минимум три операции с матрицами: складывать матрицы, умножать матрицу на матрицу и умножать матрицу на действительное число. Именно такие действия (методы для их реализации) предусмотрены в программе в листинге 5.9. Там мы описываем класс `MatrixExp` с полем `matrix`, являющимся ссылкой на двумерный массив (предназначенный для реализации квадратной матрицы). Таким образом, матрица упакована в объект класса `MatrixExp`. Операции с матрицами, в том числе и такие, как вычисление экспоненты, реализуются через методы класса.

Листинг 5.9. Матричная экспонента

```
// Класс для реализации матрицы и
// вычисления матричной экспоненты:
class MatrixExp{
```

```
// Количество слагаемых в ряде для экспоненты:
private final static int N=100;
// Размер матрицы:
private int n;
// Ссылка на матрицу (двумерный массив):
private double[][] matrix;
// Конструктор (размер матрицы и диапазон
// случайных значений):
MatrixExp(int n,double Xmin,double Xmax){
   double x=Math.abs(Xmax-Xmin);
   int i,j;
   this.n=n;
   matrix=new double[n][n];
   for(i=0;i<n;i++){
      for(j=0;j<n;j++){
         matrix[i][j]=x*Math.random()+Xmin;
      }
   }
}
// Конструктор (на основе существующей матрицы):
MatrixExp(double[][] matrix){
   this.n=matrix[0].length;
   this.matrix=new double[n][n];
   int i,j;
   for(i=0;i<n;i++){
      for(j=0;j<n;j++){
         this.matrix[i][j]=matrix[i][j];
      }
   }
}
// Конструктор (единичная матрица):
MatrixExp(int n){
   this.n=n;
   matrix=new double[n][n];
   int i,j;
   for(i=0;i<n;i++){
      for(j=0;j<i;j++){
         matrix[i][j]=0;
      }
      matrix[i][i]=1;
      for(j=i+1;j<n;j++){
         matrix[i][j]=0;
      }
   }
}
// Конструктор (заполнение одним числом):
MatrixExp(int n,double a){
   this.n=n;
   matrix=new double[n][n];
   int i,j;
   for(i=0;i<n;i++){
      for(j=0;j<n;j++){
         matrix[i][j]=a;
```

```
      }
   }
}
// Метод для отображения матрицы:
void show(){
   int i,j;
   for(i=0;i<n;i++){
      for(j=0;j<n-1;j++){
         System.out.print(
            Math.round(1000*matrix[i][j])/1000.0+"\t"
         );
      }
      System.out.print(
         Math.round(1000*matrix[i][n-1])/1000.0+"\n"
      );
   }
}
// Метод для вычисления суммы матриц:
MatrixExp sum(MatrixExp B){
   MatrixExp t=new MatrixExp(n,0);
   int i,j;
   for(i=0;i<n;i++){
      for(j=0;j<n;j++){
         t.matrix[i][j]=matrix[i][j]+B.matrix[i][j];
      }
   }
   return t;
}
// Метод для вычисления произведения матрицы на число:
MatrixExp prod(double x){
   MatrixExp t=new MatrixExp(matrix);
   int i,j;
   for(i=0;i<n;i++){
      for(j=0;j<n;j++){
         t.matrix[i][j]*=x;
      }
   }
   return t;
}
// Метод для вычисления произведения матриц:
MatrixExp prod(MatrixExp B){
   MatrixExp t=new MatrixExp(n,0);
   int i,j,k;
   for(i=0;i<n;i++){
      for(j=0;j<n;j++){
         for(k=0;k<n;k++){
            t.matrix[i][j]+=matrix[i][k]*B.matrix[k][j];
         }
      }
   }
   return t;
}
// Метод для вычисления матричной экспоненты:
```

```
   MatrixExp mExp(){
      MatrixExp t,q;
      // Начальное значение – единичная матрица:
      t=new MatrixExp(n);
      // Начальная добавка:
      q=new MatrixExp(matrix);
      int i;
      // Вычисление ряда для экспоненты:
      for(i=1;i<=N;i++){
         t=t.sum(q);
         q=q.prod(this).prod(1.0/(i+1));
      }
      return t;
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Исходная матрица:
      MatrixExp A=new MatrixExp(3,0,1);
      System.out.println("Матрица A:");
      // Отображение исходной матрицы:
      A.show();
      System.out.println("Матрица exp(A):");
      // Вычисление матричной экспоненты
      // и отображение результата:
      A.mExp().show();
   }
}
```

Кроме поля `matrix` в классе `MatrixExp` есть еще два закрытых поля: статическое целочисленное `final`-поле `N` определяет количество слагаемых при вычислении ряда для экспоненты, а целочисленное поле `n` задает размер матрицы. В классе имеется и несколько конструкторов для создания объекта, соответствующего:

- единичной матрице;
- матрице, заполненной случайными числами;
- матрице, созданной на основе уже существующей матрицы;
- матрице, заполненной одним и тем же числом.

Конструктор класса, в котором массив заполняется случайными числами, имеет три аргумента: целочисленный размер матрицы и границы диапазона, в котором генерируются случайные числа для заполнения. При создании объекта на основе уже существующей матрицы ссылка на соответствующий двумерный массив передается аргументом конструктору. На основе ссылки вычисляется размер массива (по первой строке, при этом предполагается, что массив по каждому из индексов имеет одинаковый размер) и выполняется поэлементное копирование. При создании объекта для реализации единичной матрицы конструктору передается целое число,

которое определяет размер матрицы. Наконец, для создания объекта с массивом, заполненным одним и тем же числом, кроме размера матрицы конструктору нужно передать еще и число для заполнения.

У класса `MatrixExp` есть метод `show()` для отображения матрицы (ссылка на которую записана в поле `matrix`). У метода нет аргументов, и он не возвращает результат. Элементы матрицы отображаются построчно, причем предварительно соответствующее значение округляется так, чтобы в дробной части было не больше трех цифр.

Метод `sum()` предназначен для вычисления суммы матриц. Аргументом ему передается объект класса `MatrixExp`. В теле метода создается локальный объект класса `MatrixExp`. Матрица, реализованная этим объектом, вычисляется как сумма матриц, представленных объектом, из которого вызывается метод, и объектом, переданным аргументом методу. В процессе вычислений использованы вложенные операторы цикла. Данный локальный объект возвращается как результат метода.

Перегруженный метод `prod()` нужен для вычисления произведения матриц и умножения матрицы на число. В частности, методу `prod()` аргументом можно передавать число или объект класса `MatrixExp`. В первом случае на основе объекта, из которого вызывается метод, создается локальная копия и затем все элементы матрицы, реализованной через этот локальный объект, умножаются на число, переданное аргументом методу. После этого локальный объект возвращается результатом метода.

Если методу `prod()` передать аргументом объект класса `MatrixExp`, то вычисляется произведение матриц. Первая матрица в произведении определяется объектом, из которого вызывается метод. Вторая матрица реализуется объектом, переданным аргументом методу. Произведение матриц вычисляется как сумма попарных произведений значений элементов строк первой матрицы и столбцов второй матрицы.

ПОДРОБНОСТИ

Допустим, есть две матрицы, для которых требуется вычислить их произведение. Результатом будет матрица. Фактически нам нужно вычислить элементы этой матрицы. Допустим, нас интересует элемент в третьей строке и пятом столбце. Тогда нужно из первой матрицы взять третью строку, а из второй матрицы — пятый столбец. Элементы из третьей строки и пятого столбца попарно перемножаются и суммируются. Результат отражает значение элемента.

Результатом метода является объект класса `MatrixExp`, через который реализуется матрица, полученная при вычислении произведения матриц.

НА ЗАМЕТКУ

При создании локального объекта класса `MatrixExp`, который затем (после выполнения всех необходимых вычислений) возвращается методом, использован конструктор заполнения матрицы одинаковыми числами (в данном случае нулями). Для вычисления произведения матриц использованы три вложенных оператора

цикла (две внешние индексные переменные определяют индексы элемента, а третья индексная переменная нужна для вычисления суммы попарных произведений значений элементов).

Метод `mExp()` используется для вычисления матричной экспоненты. В методе `mExp()` объявляются две объектные переменные `t` и `q` класса `MatrixExp`. Значения этим переменным присваиваются командами `t=new MatrixExp(n)` и `q=new MatrixExp(matrix)` соответственно. В первом случае создается единичная матрица и ссылка на нее записывается в объектную переменную `t`. Это — начальное значение для вычисления матричной экспоненты. Второй командой создается копия исходной матрицы (поле `matrix`) и ссылка на результат записывается в переменную `q`. Это —добавка, то есть следующее после текущего слагаемое в ряде для матричной экспоненты. На первом итерационном шаге эта добавка должна соответствовать исходной матрице. В общем случае на k-м итерационном шаге добавка q_k определяется как $q_k = A^k/k!$, где через A обозначена исходная матрица. Принимая во внимание, что $q_{k+1}/q_k = A/(k+1)$, приходим к выводу, что для вычисления добавки для следующего итерационного шага текущее значение переменной `q` нужно умножить (по правилу умножения матриц) на исходную матрицу, а полученный результат умножить на число $1/(k+1)$ (через k обозначено текущее значение индексной переменной). На следующем итерационном шаге значение переменной `q` прибавляется (по правилу сложения матриц) к матрице, реализованной через объект, на который ссылается переменная `t`. Ссылка на полученный в результате таких действий объект записывается в переменную `t`. Вся операция реализуется командой `t=t.sum(q)` (первая команда в операторе цикла в методе `mExp()`). Значение переменной `q` изменяется командой `q=q.prod(this).prod(1.0/(i+1))`. Интерес представляет правая часть этого выражения. Она формально состоит из двух частей. Инструкцией `q.prod(this)` вычисляется произведение матриц (той, что реализована через объект, на который ссылается переменная `q`, и той, что реализована через объект, из которого вызывается метод `mExp()`, — а ссылку на него можно получить с помощью ключевого слова `this`). Ссылка на созданный объект никуда не записывается, то есть объект является анонимным. Из анонимного объекта вызывается метод `prod()` с числовым аргументом (инструкция `q.prod(this).prod(1.0/(i+1))`). Как следствие, на основе анонимного объекта создается еще один объект. Элементы матрицы, реализованной в этом объекте, получаются умножением элементов матрицы из анонимного объекта на число, переданное аргументом методу `prod()`. Ссылка на результат записывается в переменную `q`.

НА ЗАМЕТКУ

При передаче числового аргумента методу `prod()` единица в числителе указывается как десятичный литерал, чтобы избежать целочисленного деления.

После выполнения всех необходимых операций в качестве значения метода `mExp()` возвращается объект, на который ссылается переменная `t`.

В главном методе программы в классе `Demo` командой `MatrixExp A=new MatrixExp(3,0,1)` создается объект класса `MatrixExp`. Через этот объект реализуется матрица ранга `3`, заполненная случайными числами в диапазоне от `0` до `1`. Матрица отображается командой `A.show()`. Командой `A.mExp().show()` вычисляется матричная экспонента, и результат отображается в области вывода. В данном случае также используется анонимный объект — результат инструкции `A.mExp()`. Поскольку метод `mExp()` возвращает ссылку на объект класса `MatrixExp` с вычисленной матричной экспонентой, то из этого объекта (имеется в виду объект `A.mExp()`) можно вызвать метод `show()`, что и происходит. Результат выполнения программы (с учетом того, что используется генератор случайных чисел) может быть таким:

Результат выполнения программы (из листинга 5.9)

```
Матрица A:
0.128    0.166    0.196
0.824    0.639    0.945
0.985    0.168    0.388
Матрица exp(A):
1.404    0.295    0.401
2.052    2.203    1.844
1.511    0.433    1.791
```

Более наглядный результат можно получить, выбрав в качестве начальной матрицы `A` единичную. В итоге получим матрицу, на главной диагонали которой размещены константы Эйлера (значение `2.718`).

НА ЗАМЕТКУ

Отметим, что использованный нами подход позволяет вычислять не только матричные экспоненты, но и матричные синусы, косинусы и так далее.

Операции с векторами

Следующая программа служит иллюстрацией к созданию класса для реализации векторов в трехмерном пространстве и выполнения основных операций с ними. Код программы представлен в листинге 5.10.

Листинг 5.10. Операции с векторами

```
// Класс для работы с векторами:
class Vector{
   // Поле (ссылка на массив):
   private double[] vect=new double[3];
   // Метод с тремя аргументами для определения
   // компонентов вектора:
   void set(double x,double y,double z){
      vect[0]=x;
      vect[1]=y;
      vect[2]=z;
```

```
}
// Метод для определения компонентов вектора
// (аргумент – ссылка на массив):
void set(double[] params){
   for(int i=0;i<3;i++){
      vect[i]=params[i];
   }
}
// Метод без аргументов для определения
// компонентов вектора:
void set(){
   set(0,0,0);
}
// Метод для отображения компонентов вектора
// (без перехода к новой строке):
void show(){
   double[] x=new double[3];
   for(int i=0;i<3;i++){
      x[i]=Math.round(vect[i]*100)/100.0;
   }
   System.out.print("<"+x[0]+"|"+x[1]+"|"+x[2]+">");
}
// Метод для отображения компонентов вектора
// (с переходом к новой строке):
void showln(){
   show();
   System.out.println();
}
// Метод для вычисления суммы векторов:
Vector plus(Vector b){
   Vector t=new Vector();
   for(int i=0;i<3;i++){
      t.vect[i]=vect[i]+b.vect[i];
   }
   return t;
}
// Метод для вычисления разности векторов:
Vector minus(Vector b){
   Vector t=new Vector();
   for(int i=0;i<3;i++){
      t.vect[i]=vect[i]-b.vect[i];
   }
   return t;
}
// Метод для вычисления произведения
// вектора на число:
Vector prod(double x){
   Vector t=new Vector();
   for(int i=0;i<3;i++){
      t.vect[i]=vect[i]*x;
   }
   return t;
}
```

```
// Метод для вычисления скалярного
// произведения векторов:
double prod(Vector b){
   double x=0;
   for(int i=0;i<3;i++){
      x+=vect[i]*b.vect[i];
   }
   return x;
}
// Метод для вычисления векторного
// произведения векторов:
Vector vprod(Vector b){
   Vector t=new Vector();
   for(int i=0;i<3;i++){
      t.vect[i]=vect[(i+1)%3]*b.vect[(i+2)%3]-
         vect[(i+2)%3]*b.vect[(i+1)%3];
   }
   return t;
}
// Метод для вычисления смешанного
// произведения векторов:
double mprod(Vector b,Vector c){
   return vprod(b).prod(c);
}
// Метод для деления вектора на число:
Vector div(double x){
   Vector t=new Vector();
   for(int i=0;i<3;i++){
      t.vect[i]=vect[i]/x;
   }
   return t;
}
// Метод для вычисления модуля вектора:
double module(){
   return Math.sqrt(prod(this));
}
// Метод для вычисления угла
// между векторами (в радианах):
double ang(Vector b){
   double z;
   z=prod(b)/module()/b.module();
   return Math.acos(z);
}
// Метод для вычисления угла между
// векторами (в градусах):
double angDeg(Vector b){
   return Math.toDegrees(ang(b));
}
// Метод для вычисления площади параллелограмма:
double square(Vector b){
   Vector t;
   t=vprod(b);
   return t.module();
```

```
   }
   // Конструктор (аргумент – ссылка на массив):
   Vector(double[] params){
      set(params);
   }
   // Конструктор (три аргумента):
   Vector(double x,double y,double z){
      set(x,y,z);
   }
   // Конструктор (без аргументов):
   Vector(){
      set();
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объектов:
      Vector a=new Vector(1,0,0);
      Vector b=new Vector(new double[]{0,1,0});
      // Объектная переменная:
      Vector c;
      // Выполнение операций с векторами:
      System.out.println("Векторное произведение:");
      (c=a.vprod(b)).showln();
      System.out.println(
         "Смешанное произведение: "+a.mprod(b,c)
      );
      System.out.println("Линейная комбинация векторов:");
      a.prod(3).plus(b.div(2)).minus(c).showln();
      a.set(4,0,-3);
      b.set(0,10,0);
      System.out.println(
         "Угол между векторами (в градусах): "+a.angDeg(b)
      );
      System.out.println(
         "Площадь параллелограмма: "+a.square(b)
      );
   }
}
```

Для работы с векторами предлагается класс `Vector`. У класса есть закрытое поле `vect`, которое является ссылкой на массив из трех элементов. Перегруженный метод `set()` предназначен для присваивания значения элементам этого массива. Методу аргументами могут передаваться три числа типа `double` (значения трех элементов массива `vect`) либо один аргумент — ссылка на массив из трех элементов. Существует и версия метода без аргументов (в этом случае создается вектор с нулевыми компонентами).

ПОДРОБНОСТИ

Версия метода `set()` без аргументов реализована на основе вызова версии метода `set()` с тремя нулевыми аргументами.

В соответствии с версиями метода `set()` в классе создаются и версии конструктора.

ПОДРОБНОСТИ

Конструктору могут передаваться те же аргументы, что и методу `set()`, причем реализованы разные версии конструктора на основе вызова соответствующих версий метода `set()`.

Методы `show()` и `showln()` предназначены для отображения значений компонентов вектора. Отличие между методами в том, что во втором случае выполняется переход к новой строке.

Есть несколько методов, предназначенных для выполнения операций с векторами. Так, с помощью метода `plus()` вычисляется сумма двух векторов. Результатом является объект класса `Vector`, через который реализуется сумма векторов. Разность двух векторов вычисляется методом `minus()`. Если аргументом методу `prod()` передается действительное число, то вычисляется произведение вектора на это число. Если же передать аргументом методу объект класса `Vector`, то методом `prod()` вычисляется скалярное произведение векторов. Для деления вектора на число предназначен метод `div()`.

ПОДРОБНОСТИ

При вычислении суммы векторов результатом является вектор, компоненты которого вычисляются как сумма компонентов суммируемых векторов. При вычислении разности векторов результатом также является вектор, но на этот раз речь идет о вычислении разности компонентов векторов.

Если вектор умножается или делится на число, то каждый компонент вектора умножается или делится на число.

Результатом скалярного произведения двух векторов является число. Оно определяется как сумма попарных произведений компонентов перемножаемых векторов.

Для вычисления векторного произведения используется метод `vprod()`. Результатом векторного произведения двух векторов (допустим, a и b) является вектор (назовем его c). Компоненты вектора $c = a \times b$ вычисляются на основе компонентов векторов a и b по формуле $c_k = a_{k+1}b_{k+2} - a_{k+2}b_{k+1}$, в которой подразумевается циклическая перестановка индексов — следующим после последнего индекса является первый.

НА ЗАМЕТКУ

Именно для реализации правила циклической перестановки индексов при индексации элементов в методе `vprod()` используется оператор `%` вычисления остатка от целочисленного деления.

Смешанное произведение — это векторное произведение векторов a и b, скалярно умноженное на вектор c. Смешанное произведение векторов вычисляется мето-

дом `mprod()`. Результатом является скаляр (значение типа `double`). Векторы *b* и *c* в виде объектов класса `Vector` передаются аргументами методу `mprod()`. Вектор *a* отождествляется с объектом, из которого вызывается метод. При вычислении смешанного произведения использован метод вычисления векторного произведения `vprod()`. При этом результат вычисляется инструкцией `vprod(b).prod(c)`, где `b` и `c` — аргументы метода. Здесь использовано то свойство, что результатом инструкции `vprod(b)` является объект класса `Vector`, соответствующий векторному произведению вектора, реализованного через объект, из которого вызывается метод, и вектора, реализованного через объект `b`, переданный аргументом методу. Для вычисления скалярного произведения из объекта `vprod(b)` вызывается метод `prod()` с аргументом `c` (второй аргумент метода `mprod()`).

Методом `module()` в качестве результата возвращается модуль вектора — корень квадратный из скалярного произведения вектора на самого себя (для ссылки на объект, из которого вызывается метод, использовано ключевое слово `this`).

Методами `ang()` и `angDeg()` возвращается угол между векторами (в радианах и градусах соответственно). Косинус угла между векторами равен отношению скалярного произведения векторов к произведению их модулей. Наконец, методом `square()` возвращается модуль векторного произведения двух векторов — число, которое равняется площади параллелограмма, образованного двумя данными векторами.

В главном методе программы проверяется функциональность созданного класса и его методов. Результат выполнения программы имеет следующий вид:

Результат выполнения программы (из листинга 5.10)

```
Векторное произведение:
<0.0|0.0|1.0>
Смешанное произведение: 1.0
Линейная комбинация векторов:
<3.0|0.5|-1.0>
Угол между векторами (в градусах): 90.0
Площадь параллелограмма: 50.0
```

Желающие могут изменить значения компонентов исходных векторов и посмотреть, как это скажется на результате выполнения программы.

НА ЗАМЕТКУ

Обратите внимание на команды `(c=a.vprod(b)).showln()` и `a.prod(3).plus(b.div(2)).minus(c).showln()` в главном методе программы. В первом случае объекту `c` в качестве значения присваивается результат операции `a.vprod(b)`, и поскольку оператор присваивания возвращает значение, для объекта-результата вызывается метод `showln()`. Второй командой для векторов, представленных объектами `a`, `b` и `c`, вычисляется линейная комбинация $3a + b/2 - c$.

Операции с полиномами

В следующей программе (листинг 5.11) для работы с выражениями полиномиального типа создается класс `Polynom`. В нем описаны методы для сложения, вычитания, умножения полиномов, умножения и деления полинома на число и вычисления производной от полинома.

Листинг 5.11. Операции с полиномами

```
// Класс для реализации полиномов:
class Polynom{
   // Полином степени n-1:
   private int n;
   // Коэффициенты полинома:
   private double[] a;
   // Метод для определения коэффициентов полинома
   // на основе массива:
   void set(double[] a){
      this.n=a.length;
      this.a=new double[n];
      int i;
      for(i=0;i<n;i++){
         this.a[i]=a[i];
      }
   }
   // Метод для определения коэффициентов полинома
   // (аргументы – размер массива и число, которым
   // заполняется массив):
   void set(int n,double z){
      this.n=n;
      this.a=new double[n];
      int i;
      for(i=0;i<n;i++){
         this.a[i]=z;
      }
   }
   // Метод для определения коэффициентов полинома
   // (аргумент – размер массива, при этом
   // массив заполняется нулями):
   void set(int n){
      set(n,0);
   }
   // Метод для вычисления значения полинома в точке:
   double value(double x){
      double z=0,q=1;
      for(int i=0;i<n;i++){
         z+=a[i]*q;
         q*=x;
      }
      return z;
   }
```

```
// Метод для отображения коэффициентов полинома:
void show(){
   int i;
   System.out.print("Степень x:  ");
   for(i=0;i<n-1;i++){
      System.out.printf("%6d",i);
   }
   System.out.printf("%6d\n",(n-1));
   System.out.print("Коэффициент:");
   for(i=0;i<n-1;i++){
      System.out.printf("%6.1f",a[i]);
   }
   System.out.printf("%6.1f\n",a[n-1]);
}
// Метод для отображения значения полинома в точке:
void show(double x){
   System.out.println("Значение аргумента   x="+x);
   System.out.println(
      "Значение полинома P(x)="+value(x)
   );
}
// Метод для вычисления производной от полинома:
Polynom diff(){
   Polynom t=new Polynom(n-1);
   for(int i=0;i<n-1;i++){
      t.a[i]=a[i+1]*(i+1);
   }
   return t;
}
// Метод для вычисления производной (порядка k)
// от полинома:
Polynom diff(int k){
   if(k>=n) return new Polynom(1);
   if(k>0) return diff().diff(k-1);
   else return new Polynom(a);
}
// Метод для вычисления суммы полиномов:
Polynom plus(Polynom Q){
   Polynom t;
   int i;
   if(n>=Q.n){
      t=new Polynom(a);
      for(i=0;i<Q.n;i++){
         t.a[i]+=Q.a[i];
      }
   }else{
      t=new Polynom(Q.a);
      for(i=0;i<n;i++){
         t.a[i]+=a[i];
      }
   }
   return t;
}
```

```
   // Метод для вычисления разности полиномов:
   Polynom minus(Polynom Q){
      return plus(Q.prod(-1));
   }
   // Метод для вычисления результата деления
   // полинома на число:
   Polynom div(double z){
      return prod(1/z);
   }
   // Метод для вычисления результата умножения
   // полинома на число:
   Polynom prod(double z){
      Polynom t=new Polynom(a);
      for(int i=0;i<n;i++){
         a[i]*=z;
      }
      return t;
   }
   // Метод для вычисления произведения полиномов:
   Polynom prod(Polynom Q){
      int N=n+Q.n-1;
      Polynom t=new Polynom(N);
      for(int i=0;i<n;i++){
         for(int j=0;j<Q.n;j++){
            t.a[i+j]+=a[i]*Q.a[j];
         }
      }
      return t;
   }
   // Конструктор (аргумент – ссылка на массив):
   Polynom(double[] a){
      set(a);
   }
   // Конструктор (два числовых аргумента):
   Polynom(int n,double z){
      set(n,z);
   }
   // Конструктор (один числовой аргумент):
   Polynom(int n){
      set(n);
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Коэффициенты для полинома:
      double[] coefs=new double[]{3,-2,-1,0,1};
      // Создание объекта для реализации полинома:
      Polynom P=new Polynom(coefs);
      System.out.println(
         "\tКоэффициенты исходного полинома:"
      );
```

```
        // Коэффициенты полинома:
        P.show();
        System.out.println("\tЗначение полинома в точке:");
        // Значение полинома для единичного аргумента:
        P.show(1);
        System.out.println("\tВторая производная:");
        // Вычисление второй производной для полинома:
        Polynom Q=P.diff(2);
        // Результат вычисления производной (коэффициенты):
        Q.show();
        System.out.println("\tСумма полиномов:");
        // Сумма полиномов (результат):
        Q.plus(P).show();
        System.out.println("\tПроизведение полиномов:");
        // Произведение полиномов (результат):
        Q.prod(P).show();
    }
}
```

В классе `Polynom` закрытое целочисленное поле `n` определяет степень полинома (степень полинома равняется `n-1`, и такой полином определяется набором из `n` коэффициентов), а закрытое поле `a` является ссылкой на массив типа `double`. Этот массив предназначен для записи коэффициентов полинома.

Перегруженный метод `set()` предназначен для заполнения массива с коэффициентами полинома. Предусмотрена возможность передавать методу ссылку на массив, заполнять массив одинаковыми числами (передав аргументом методу размер массива и значение, которым заполняется массив), а также описан частный случай, когда аргументом методу передается только размер массива (при этом массив заполняется нулями). Реализация последней версии метода `set()` базируется на вызове варианта этого же метода с первым аргументом — размером массива и нулевым вторым аргументом.

У метода `value()` один аргумент типа `double`, а в качестве результата метод возвращает значение полинома в точке, которая определяется аргументом метода.

Перегруженный метод `show()` имеет две версии: без аргументов и с одним аргументом типа `double`. В первом случае вместе с дополнительной информацией отображаются значения коэффициентов полинома. Если методу `show()` передается аргумент (значение типа `double`), то отображается сообщение с информацией о значении полинома в этой точке.

В классе `Polynom` предусмотрена возможность вычислять производные, причем произвольного порядка. Результатом вычисления производной от полинома также является полином. Задача по вычислению производной от полинома сводится, по большому счету, к расчету на основе коэффициентов исходного полинома коэффициентов полинома-производной. Поскольку для степенной функции $y(x)=x^n$ производная dy/dx определяется как $dy/dx=nx^{n-1}$, а производная суммы функций равняется сумме производных, то производной для полинома $P(x)=\sum_{k=0}^{n} a_k x^k$ является функция-полином

$$P'(x)=\sum_{k=1}^{n}ka_kx^{k-1}=\sum_{k=0}^{n-1}(k+1)a_{k+1}x^k\equiv\sum_{k=0}^{n-1}b_kx^k.$$

Коэффициенты полинома-производной b_k связаны с коэффициентами a_k исходного полинома соотношением $b_k=(k+1)a_{k+1}$ для $k=0, 1, ..., n-1$, а старший коэффициент $b_n=0$. Это означает фактически, что производная является полиномом степени, на единицу меньшей, чем исходный полином. Данное обстоятельство, а также соотношения между коэффициентами исходного полинома и полинома-производной, нашли отображение в коде метода `diff()`, которым в качестве результата возвращается объект класса `Polynom`. Это первая производная для полинома, реализованного через объект, из которого вызывается метод. В теле метода командой `Polynom t=new Polynom(n-1)` создается объект `t`, соответствующий полиному степени, на единицу меньшей, чем исходный. При создании объекта использован конструктор с одним аргументом (размер массива), поэтому коэффициенты поля `a` этого объекта заполняются нулями (хотя в данном случае это непринципиально). В операторе цикла с индексной переменной `i` командой `t.a[i]=a[i+1]*(i+1)` выполняется последовательное вычисление коэффициентов полинома-производной. После этого объект `t` возвращается как результат метода.

Метод `diff()` перегружается. Если этому методу передать целочисленный аргумент, то результатом возвращается производная соответствующего порядка. При этом если аргумент `k` (порядок производной) превышает степень полинома (равняется `n-1`), то посредством команды `if(k>=n) return new Polynom(1)` создается анонимный объект, соответствующий полиному нулевой степени (то есть это число с единственным нулевым коэффициентом), и этот полином (объект) возвращается в качестве результата: если порядок производной превышает показатель степенной функции, то производная тождественно равна нулю. В противном случае при положительном (больше нуля) значении порядка производной в качестве результата возвращается объект `diff().diff(k-1)`. Иначе возвращается анонимный объект, который создается командой `new Polynom(a)`. Это — копия исходного полинома. Здесь реализовано формальное правило, гласящее, что производная нулевого порядка по определению совпадает с исходной (дифференцируемой) функцией.

ПОДРОБНОСТИ

Несколько замечаний относительно команды `diff().diff(k-1)`. В ней использован рекурсивный вызов метода `diff()` одновременно с вызовом версии этого метода без аргумента. В частности, команда `diff().diff(k-1)` реализует правило, согласно которому производная порядка `k` — это производная порядка `k-1` от первой производной. Первая производная вычисляется инструкцией `diff()`. Результатом инструкции является объект (в данном случае анонимный), который соответствует полиному-производной. Из этого анонимного объекта инструкцией `diff(k-1)` снова вызывается метод для вычисления производной, но уже меньшего порядка.

Методом `plus()` в качестве результата вычисляется объект класса `Polynom`, соответствующий сумме двух полиномов: один из них реализован через объект, из которого вызывается метод, второй — соответствует объекту, переданному аргументом методу. Общий принцип вычисления суммы полиномов состоит в том, что нужно сложить коэффициенты, соответствующие одинаковым показателям степени переменной полинома. Главная проблема в данном случае связана с тем, что складываемые полиномы могут иметь разную степень и, как результат, разные размеры массивов с коэффициентами. Поэтому на начальном этапе проверяется, у какого из объектов размер массива больше. Для этого объекта создается локальная копия. Далее перебираются элементы массива второго объекта (того, у которого размер массива меньше), и значения этих элементов прибавляются к значениям соответствующих элементов массива из локального объекта. Затем локальный объект возвращается как результат метода.

Разность полиномов вычисляется методом `minus()`. С помощью метода `prod()` полином, реализованный объектом, переданным аргументом методу `minus()`, умножается на `-1`. Полученный в результате полином прибавляется (с помощью метода `plus()`) к исходному полиному (представленному объектом, из которого вызывается метод `minus()`). Вся процедура по вычислению разности двух полиномов реализована инструкцией `plus(Q.prod(-1))`, где `Q` — объект класса `Polynom`, переданный аргументом методу `minus()`. Что касается метода `prod()`, то он позволяет умножать полином (объект класса `Polynom`) на число и на другой полином. При умножении полинома на число аргументом методу `prod()` передается значение типа `double`. На это число умножается каждый коэффициент полинома. Если же аргументом методу `prod()` передать объект класса `Polynom`, то вычисляется произведение полиномов (один из них представлен объектом, из которого вызывается метод, другой — объектом, переданным аргументом методу). Результатом также является объект класса `Polynom`. При вычислении его параметров (массива с коэффициентами) приняты во внимание следующие обстоятельства. Во-первых, если умножаются полиномы степени n и m, то результатом будет полином степени $n + m$. Поскольку степень полинома на единицу меньше количества коэффициентов, которыми однозначно определяется полином, то размер массива для объекта-результата на единицу меньше, чем сумма размеров массивов для исходных объектов. Это правило реализовано в команде `int N=n+Q.n-1`, где `Q` является объектом класса `Polynom`, который передается методу `prod()`. Переменная `N` таким образом определяет размер массива для объекта, являющегося результатом метода `prod()`. Локальный объект, возвращаемый в последующем в качестве результата, создается командой `Polynom t=new Polynom(N)`. Начальные значения элементов массива для локального объекта `t` равны нулю. Дальше запускаются вложенные операторы цикла. Индексная переменная в одном операторе цикла перебирает элементы массива первого объекта, а индексная переменная в другом операторе цикла перебирает элементы массива для второго объекта. Изменение значений элементов массива для локального объекта выполняется командой `t.a[i+j]+=a[i]*Q.a[j]`. В данном случае принято во внимание, что элемент в массиве коэффициентов полинома с индексом `i` соот-

ветствует слагаемому с переменной полинома в степени `i`. Если мы перемножаем два полинома, $P(x)=\sum_{i=0}^{n}a_i x^i$ и $Q(x)=\sum_{j=0}^{m}b_j x^j$, то произведение элементов массивов с индексами `i` и `j` дает в полиноме-результате вклад в слагаемое, соответствующее переменной в степени `i+j`:

$$R(x)\equiv P(x)Q(x)=\sum_{i=0}^{n}a_i x^i\sum_{j=0}^{m}b_j x^j=\sum_{\substack{0\le i\le n,\\ 0\le j\le m}}a_i b_j x^{i+j}.$$

Другими словами, чтобы рассчитать коэффициент для полинома-результата с индексом k, необходимо найти сумму $\sum_{i+j=k}a_i b_j$ всех попарных произведений коэффициентов исходных полиномов таких, что сумма их индексов равняется k. Именно этот принцип и реализован в программе.

Конструкторы класса `Polynom` реализованы на основе различных версий метода `set()`, которые уже описывались и, думается, особых комментариев не требуют.

Что касается главного метода программы в классе `Demo`, то там проверяется работа некоторых методов, описанных в классе `Polynom`.

НА ЗАМЕТКУ

Обращаем внимание на способ вызова методов из анонимных объектов. Например, командой `Polynom Q=P.diff(2)` на основе определенного ранее объекта `P` вычисляется объект для второй производной от исходного полинома и ссылка на этот объект записывается в объектную переменную `Q`. Аналогично, командой `Q.plus(P).show()` вычисляется анонимный объект для суммы полиномов (инструкция `Q.plus(P)`) и из этого объекта вызывается метод `show()`.

Результат выполнения программы имеет следующий вид:

Результат выполнения программы (из листинга 5.11)

```
    Коэффициенты исходного полинома:
Степень x:       0     1     2     3     4
Коэффициент:   3,0  -2,0  -1,0   0,0   1,0
    Значение полинома в точке:
Значение аргумента   x=1.0
Значение полинома P(x)=1.0
    Вторая производная:
Степень x:       0     1     2
Коэффициент:  -2,0   0,0  12,0
    Сумма полиномов:
Степень x:       0     1     2     3     4
Коэффициент:   1,0  -2,0  11,0   0,0   1,0
    Произведение полиномов:
Степень x:       0     1     2     3     4     5     6
Коэффициент:  -6,0   4,0  38,0 -24,0 -14,0   0,0  12,0
```

Желающие могут проверить, что все коэффициенты и значения вычислены корректно.

НА ЗАМЕТКУ

Отображение результатов в данном случае реализовано с помощью специального метода `show()` из класса `Polynom`. На практике существует более простой, надежный и эффективный способ обеспечить приемлемый способ отображения информации об объекте. Состоит он в переопределении метода `toString()`. Подробнее этот механизм будет обсуждаться после того, как мы познакомимся с наследованием.

Бинарное дерево

Рассмотрим в качестве элементарного примера программу, в которой конструкторами создается бинарное дерево объектов — каждый объект имеет по две ссылки на объекты того же класса. Каждый объект, кроме прочего, имеет несколько полей. Символьное (типа `char`) поле `Level` определяет уровень объекта: например, в вершине иерархии находится объект уровня `A`, который ссылается на два объекта уровня `B`, которые, в свою очередь, ссылаются в общей сложности на четыре объекта уровня `C`, и так далее. Объекты нумеруются, для чего используется целочисленное поле `Number`. Нумерация выполняется в пределах одного уровня. Например, на верхнем уровне `A` — всего один объект с номером `1`. На втором уровне `B` — два объекта с номерами `1` и `2`. На третьем уровне `C` — четыре объекта с номерами от `1` до `4` включительно, и так далее. Описанная структура объектов представлена на рис. 5.1.

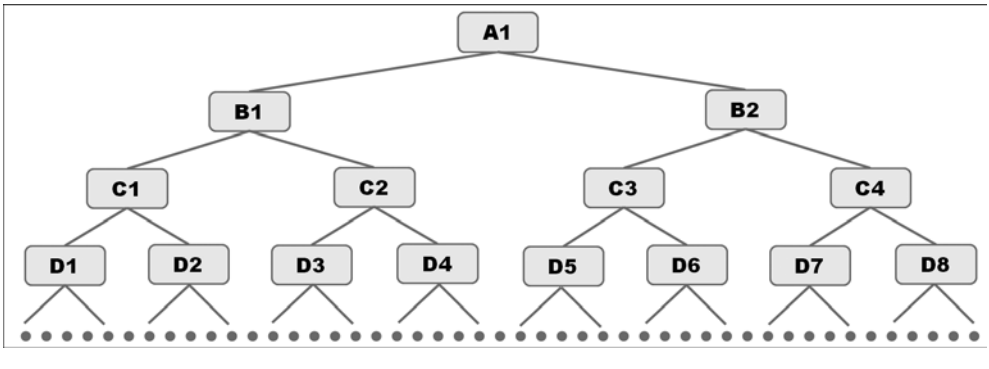


Рис. 5.1. Структура бинарного дерева объектов

Кроме метки уровня и номера объекта на уровне, каждый объект имеет еще и свой идентификационный код. Этот код генерируется случайным образом при создании объекта и состоит по умолчанию из восьми цифр (количество цифр в коде определяется закрытым статическим целочисленным полем `IDnum`). Что касается

самого кода, то он записывается в целочисленный массив, на который ссылается переменная массива `ID`, — закрытое поле класса `ObjectTree`. Каждая цифра кода записывается отдельным элементом соответствующего массива. Для генерирования (создания) кода объекта используется метод `getID()`. Для этого в теле метода командой `ID[i]=(int)(Math.random()*10)` в рамках оператора цикла генерируются случайные целые числа (инструкцией `(int)(Math.random()*10)`) и присваиваются элементам массива `ID`.

Для отображения кода объекта используется закрытый метод `showID()`. Методом последовательно отображаются значения элементов массива `ID`, при этом в качестве разделителя используется вертикальная черта. Сам метод вызывается в методе `show()`, который, в свою очередь, предназначен для отображения параметров объекта: уровня объекта в структуре, порядкового номера объекта на уровне и идентификационного кода объекта.

Открытые поля `FirstRef` и `SecondRef` являются объектными переменными класса `ObjectTree` и предназначены для записи ссылок на объекты следующего уровня. Присваивание значений этим переменным выполняется при вызове конструктора. А теперь обратимся к листингу 5.12.

Листинг 5.12. Бинарное дерево объектов

```
// Класс для реализации бинарного дерева:
class ObjectTree{
   // Количество цифр в ID-коде объекта:
   private static int IDnum=8;
   // Уровень объекта (буква):
   private char Level;
   // Номер объекта на уровне:
   private int Number;
   // Код объекта (массив цифр):
   private int[] ID;
   // Ссылка на первый объект:
   ObjectTree FirstRef;
   // Ссылка на второй объект:
   ObjectTree SecondRef;
   // Метод для генерирования ID-кода объекта:
   private void getID(){
      ID=new int[IDnum];
      for(int i=0;i<IDnum;i++){
         ID[i]=(int)(Math.random()*10);
      }
   }
   // Метод для отображения ID-кода объекта:
   private void showID(){
      for(int i=0;i<IDnum;i++){
         System.out.print("|"+ID[i]);
      }
      System.out.println("|");
   }
```

```
   // Метод для отображения параметров объекта:
   void show(){
      System.out.println("Уровень объекта: \t"+Level);
      System.out.println("Номер на уровне: \t"+Number);
      System.out.print("ID-код объекта: \t");
      showID();
   }
   // Конструктор для создания бинарного дерева:
   ObjectTree(int k,char L,int n){
      System.out.println("\tСоздан новый объект!");
      Level=L;
      Number=n;
      getID();
      show();
      if(k==1){
         FirstRef=null;
         SecondRef=null;
      }else{
         // Рекурсивный вызов конструктора:
         FirstRef=new ObjectTree(k-1,(char)(L+1),2*n-1);
         SecondRef=new ObjectTree(k-1,(char)(L+1),2*n);
      }
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Дерево объектов:
      ObjectTree tree=new ObjectTree(4,'A',1);
      System.out.println("\tПроверка дерева объектов");
      // Проверка структуры дерева объектов:
      tree.FirstRef.SecondRef.FirstRef.show();
   }
}
```

Обратим особое внимание на код конструктора, поскольку именно при его вызове создается вся структура бинарного дерева. У конструктора три аргумента: первый определяет количество уровней в структуре, начиная с текущего объекта, второй задает метку для объекта, а третий определяет номер объекта. При вызове конструктора выводится сообщение о создании объекта, после чего на основе значений аргументов конструктора присваиваются значения полям `Level` и `Number`. Затем с помощью метода `getID()` генерируется идентификационный код объекта и методом `show()` выводится информация о созданном объекте. Вторая часть кода конструктора реализована с помощью условного оператора. Там первый аргумент конструктора проверяется на предмет равенства единице. Если первый аргумент равен единице (это означает, что после текущего объекта других объектов нет), полям `FirstRef` и `SecondRef` в качестве значений присваиваются нулевые ссылки (значение `null`). Это означает, что текущий объект не имеет ссылок на другие объекты. В противном случае (то есть если аргумент кон-

структора отличен от единицы) командами `FirstRef=new ObjectTree(k-1,(char)(L+1),2*n-1)` и `SecondRef=new ObjectTree(k-1,(char)(L+1),2*n)` создаются два новых объекта. Ссылки на созданные объекты записываются в поля `FirstRef` и `SecondRef` текущего объекта. При создании новых объектов значение первого аргумента конструктора на единицу меньше, чем значение первого аргумента этого конструктора (при выполнении которого создается два новых объекта). Уровень новых создаваемых объектов определяется на основе текущего значения `L` для уровня текущего объекта как `(char)(L+1)`. Для первого из двух создаваемых объектов номер объекта вычисляется на основе номера текущего объекта `n` как `2*n-1`. Второй объект получает номер `2*n`. Принцип нумерации рассчитан так, что если в вершине иерархии объект имеет номер `1`, то на всех прочих уровнях объекты нумеруются последовательностью натуральных чисел. Таким образом, код конструктора класса организован по рекурсивному принципу: в конструкторе вызывается конструктор, но с другими аргументами. Для создания бинарного дерева вызывается конструктор (а происходит это при создании объекта), первым аргументом которому передается количество уровней в бинарном дереве, имя (буква) и номер для первого объекта.

В главном методе программы командой `ObjectTree tree=new ObjectTree(4,'A',1)` создается дерево из четырех уровней, после чего выполняется проверка созданной структуры через систему последовательных ссылок посредством команды `tree.FirstRef.SecondRef.FirstRef.show()` с вызовом метода `show()` для отображения параметров одного из объектов. Результат выполнения программы может иметь (с учетом использования генератора случайных чисел) следующий вид:

Результат выполнения программы (из листинга 5.12)

```
Создан новый объект!
Уровень объекта:    A
Номер на уровне:    1
ID-код объекта:     |7|8|3|3|5|0|3|6|
    Создан новый объект!
Уровень объекта:    B
Номер на уровне:    1
ID-код объекта:     |2|3|8|0|2|3|9|3|
    Создан новый объект!
Уровень объекта:    C
Номер на уровне:    1
ID-код объекта:     |9|4|8|4|2|2|9|2|
    Создан новый объект!
Уровень объекта:    D
Номер на уровне:    1
ID-код объекта:     |3|2|2|2|5|7|4|5|
    Создан новый объект!
Уровень объекта:    D
Номер на уровне:    2
ID-код объекта:     |2|8|2|0|8|0|7|4|
    Создан новый объект!
Уровень объекта:    C
```

```
Номер на уровне:    2
ID-код объекта:     |1|0|0|4|1|5|5|1|
    Создан новый объект!
Уровень объекта:    D
Номер на уровне:    3
ID-код объекта:     |0|0|2|0|9|5|6|8|
    Создан новый объект!
Уровень объекта:    D
Номер на уровне:    4
ID-код объекта:     |3|0|1|6|9|8|4|4|
    Создан новый объект!
Уровень объекта:    B
Номер на уровне:    2
ID-код объекта:     |4|0|3|4|8|1|0|8|
    Создан новый объект!
Уровень объекта:    C
Номер на уровне:    3
ID-код объекта:     |2|4|5|5|7|5|9|7|
    Создан новый объект!
Уровень объекта:    D
Номер на уровне:    5
ID-код объекта:     |3|3|1|8|6|9|8|9|
    Создан новый объект!
Уровень объекта:    D
Номер на уровне:    6
ID-код объекта:     |4|9|1|5|2|4|7|2|
    Создан новый объект!
Уровень объекта:    C
Номер на уровне:    4
ID-код объекта:     |0|0|4|7|8|2|7|1|
    Создан новый объект!
Уровень объекта:    D
Номер на уровне:    7
ID-код объекта:     |0|2|4|0|2|9|0|0|
    Создан новый объект!
Уровень объекта:    D
Номер на уровне:    8
ID-код объекта:     |6|6|0|9|0|4|2|2|
    Проверка дерева объектов
Уровень объекта:    D
Номер на уровне:    3
ID-код объекта:     |0|0|2|0|9|5|6|8|
```

Сначала сверху вниз (см. рис. 5.1) создаются объекты, имеющие номер `1`. Затем создается объект последнего уровня с номером `2`. Далее создается объект предпоследнего уровня с номером `2`, после чего объект последнего уровня с номером `3`, объект последнего уровня с номером `4` и так далее. Командой `tree.FirstRef.SecondRef.FirstRef.show()` метод `show()` вызывается из объекта уровня `D` с номером `3`. Отображаются параметры именно этого объекта.

Резюме

Все-все, конец фильма! Дальше без меня, дальше неинтересно.

из к/ф «Айболит-66»

- Методы могут перегружаться. В этом случае создается несколько версий одного метода. Все они имеют одинаковое название, но отличаются количеством и/или типом аргументов. Также перегруженные версии методов могут возвращать результат разного типа (или не возвращать совсем). Версия метода для вызова определяется по контексту команды, в которой вызывается метод (по типу и количеству фактически переданных методу аргументов).
- Конструктор — это метод, который вызывается автоматически при создании объекта. Имя конструктора совпадает с именем класса. Конструктор не возвращает результат, идентификатор типа результата для него не указывается. Конструктору можно передавать аргументы; конструктор можно перегружать. Аргументы, которые передаются конструктору при создании объекта, указываются в круглых скобках после имени класса в инструкции создания объекта.
- Объекты могут передаваться методам в качестве аргументов, а также возвращаться методами в качестве результата. Если в метод нужно передать объект, то методу (через объектную переменную) передается ссылка на объект. Если метод возвращает результатом объект, то в действительности речь идет о том, что возвращается ссылка на объект. Сам объект обычно создается в процессе выполнения метода.
- Аргументы передаются по значению. Это означает, что на самом деле в метод передается копия переменной, указанной аргументом. В некоторых случаях эту особенность нужно принимать в расчет.

6 Наследование

— Где ваша родина?

— Не знаю. Я родился на корабле, но куда он плыл и откуда — никто не помнит.

из к/ф «Формула любви»

Одним из фундаментальных механизмов, лежащих в основе любого объектно-ориентированного языка, в том числе и Java, является *наследование*. Наследование позволяет создавать новые классы на основе уже существующих. Это экономит силы и средства на создание новых кодов, одновременно повышая надежность программ. Именно с наследованием нам предстоит познакомиться в данной главе.

Знакомство с наследованием

Чтобы продать что-нибудь ненужное, надо сначала купить что-нибудь ненужное. А у нас денег нет!

из м/ф «Трое из Простоквашино»

Итак, мы планируем использовать наследование при создании классов. Существующий класс, на основе которого создается новый класс, называется *суперклассом*. Класс, который создается на основе суперкласса, называется *подклассом*. Поясним, что в подклассе нет необходимости описывать код из суперкласса: все (не закрытые) поля и методы суперкласса автоматически оказываются в подклассе. Так что в подклассе описываются только дополнительные члены класса, которые мы хотим добавить в подкласс помимо полей и методов, описанных в суперклассе.

НА ЗАМЕТКУ

Если в описании класса указать ключевое слово `final`, то на основе такого класса нельзя будет создать подкласс.

Процесс создания подкласса практически не отличается от процесса создания обычного класса, только при создании подкласса необходимо указать суперкласс, на основе которого создается подкласс. Для этого в описании подкласса после имени класса указывается ключевое слово `extends` и имя суперкласса. Во всем остальном описание подкласса не отличается от описания обычного класса (то есть класса, который создается с нуля). Синтаксис описания подкласса имеет вид:

```
class Bravo extends Alpha{
   // Описание подкласса
}
```

В данном случае подкласс `Bravo` создается на основе суперкласса `Alpha`. В результате подкласс `Bravo` получает (наследует) открытые и защищенные члены класса `Alpha`.

НА ЗАМЕТКУ

Члены класса могут быть описаны без спецификатора уровня доступа (открытые) или со спецификаторами `public` (открытые), `private` (закрытые) или `protected` (защищенные). Наследуются члены суперкласса, которые описаны без спецификатора уровня доступа или со спецификаторами `public` и `protected`. Что происходит с закрытыми членами суперкласса (описаны со спецификатором `private`), мы обсудим позже.

В Java, в отличие от некоторых других языков, подкласс класс может создаваться на основе только одного суперкласса. Ситуация, когда новый класс можно создавать на основе сразу нескольких классов, называется множественным наследованием. В Java множественного наследования нет. Зато есть многоуровневое, когда на основе суперкласса можно создать подкласс, а на основе этого подкласса создать еще один подкласс, и так далее. То есть подкласс может быть суперклассом для другого класса. Благодаря многоуровневому наследованию можно создавать целые цепочки связанных механизмом наследования классов.

В листинге 6.1 приведен пример создания подкласса на основе суперкласса.

Листинг 6.1. Создание подкласса на основе суперкласса

```
// Суперкласс:
class Alpha{
   // Целочисленное поле:
   int number;
   // Метод для отображения значения поля:
   void show(){
      System.out.println("Поле number: "+number);
   }
}
// Подкласс:
class Bravo extends Alpha{
   // Целочисленное поле:
   int value;
   // Метод для отображения значения поля:
```

```
   void display(){
      System.out.println("Поле value: "+value);
   }
   // Метод для вычисления суммы полей:
   void sum(){
      // Обращение к унаследованному полю:
      System.out.println("Сумма: "+(number+value));
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Объект суперкласса:
      Alpha A=new Alpha();
      System.out.println("Объект A");
      A.number=100;
      A.show();
      // Объект подкласса:
      Bravo B=new Bravo();
      System.out.println("Объект B");
      B.number=200;
      B.value=300;
      B.show();
      B.display();
      B.sum();
   }
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 6.1)

```
Объект A
Поле number: 100
Объект B
Поле number: 200
Поле value: 300
Сумма: 500
```

В классе `Alpha` есть целочисленное поле `number` и метод `show()`, при вызове которого отображается значение поля. На основе класса `Alpha` наследованием создается класс `Bravo`. Поэтому автоматически в классе `Bravo` появляется поле `number` и метод `show()` (хотя мы их там и не описывали). Дополнительно мы описываем в классе `Bravo` еще одно целочисленное поле `value`, метод `display()` (отображает значение поля `value`), а также метод `sum()`. Методом `sum()` вычисляется сумма значений полей `number` и `value`, и полученное значение отображается в области вывода.

ПОДРОБНОСТИ

Таким образом, в классе `Bravo` есть поля `number` и `value`, а также методы `show()`, `display()` и `sum()`. Ко всем этим полям и методам мы можем обращаться в про-

граммном коде класса, а если на основе класса будет создан объект, то вызывать методы и обращаться к полям через этот объект.

Также хочется обратить внимание на инструкцию `"Сумма: "+(number+value)`, указанную аргументом метода `println()`. А именно скобки при вычислении суммы полей `number` и `value` использованы не случайно. Здесь мы, по сути, имеем дело с суммой трех слагаемых, первое из которых является текстом (литерал `"Сумма: "`), а два других — числа (значения полей `number` и `value`). Если не использовать скобки, то выражение будет вычисляться так: к тексту дописывается второе слагаемое, а к полученной текстовой строке дописывается третье слагаемое. Поскольку мы использовали скобки, то сначала вычисляется сумма полей, и затем уже полученное значение дописывается к первому текстовому слагаемому.

В главном методе создаются два объекта: объект `A` класса `Alpha` и объект `B` класса `Bravo`. Там же есть примеры обращения к полям и методам этих объектов.

НА ЗАМЕТКУ

То, что класс `Bravo` является подклассом суперкласса `Alpha`, никаких ограничений на объекты `A` и `B` не накладывает. Объекты между собой никак не связаны, и каждый живет своей жизнью.

Наследование и закрытые члены

> Варварская игра, дикая местность — меня тянет на родину.
>
> *из к/ф «Формула любви»*

Пришло время выяснить, какова судьба закрытых (описанных со спецификатором доступа `private`) членов суперкласса при наследовании. Обычно утверждают, что закрытые члены суперкласса не наследуются. С этим утверждением можно согласиться. Важно только понимать, что означает выражение «не наследуются». Рассмотрим в этой связи пример, представленный в листинге 6.2.

Листинг 6.2. Закрытые члены суперкласса и наследование

```
// Суперкласс с закрытым полем:
class Alpha{
   // Закрытое символьное поле:
   private char symbol;
   // Открытый метод для отображения значения поля:
   void show(){
      System.out.println("Символ: "+symbol);
   }
   // Открытый метод для присваивания значения полю:
   void set(char s){
```

```
      symbol=s;
   }
}
// Подкласс:
class Bravo extends Alpha{}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Объект подкласса:
      Bravo B=new Bravo();
      // Вызов унаследованных методов:
      B.set('B');
      B.show();
   }
}
```

Ниже показано, как выглядит результат выполнения программы:

Результат выполнения программы (из листинга 6.2)

```
Символ: B
```

Мы описали класс `Alpha` с закрытым символьным полем `symbol` и двумя открытыми методами — `show()` и `set()`. Метод `show()` нужен для отображения значения поля, а метод `set()` предназначен для присваивания значения полю (значение передается аргументом методу). На основе класса `Alpha` наследованием создается класс `Bravo`. Причем в теле класса `Bravo` вообще ничего не описывается (в фигурных скобках с описанием класса нет ни одной команды). Поэтому в классе `Bravo` есть только те члены, которые унаследованы из класса `Alpha`. Из класса `Alpha` в классе `Bravo` наследуются открытые методы `show()` и `set()`. Но интрига в том, что при вызове эти методы обращаются к полю `symbol`, которое как бы не наследуется. В главном методе мы создаем объект `B` класса `Bravo` и вызываем из этого объекта методы `set()` и `show()`. О чем свидетельствует результат выполнения программы? Все происходит так, как если бы у класса `B` было поле `symbol`: метод `set()` при вызове присваивает значение этому полю, а метод `show()` считывает значение поля. Данное противоречие (между тем, что поле `symbol` не наследуется, с одной стороны, и как бы существует — с другой) разрешается, если уточнить, какой смысл мы вкладываем в понятие «не наследуется». А именно: закрытое поле или метод из суперкласса не наследуются в подклассе в том смысле, что в подклассе эти поля и методы недоступны. Мы не можем к ним обратиться напрямую даже в теле класса. Например, в теле класса `Bravo` использование инструкции `symbol` приведет к ошибке. Класс про такое поле ничего не знает (а объект `B` — тем более). Но физически это поле существует, и методы `set()` и `show()`знают, где его искать.

ПОДРОБНОСТИ

Наследование членов суперкласса подразумевает, что эти поля доступны в подклассе. Другими словами, подкласс знает о существовании наследуемых членов,

и к этим членам можно обращаться так, как если бы они были описаны в самом подклассе. Если же член суперкласса подклассом не наследуется, то о таком члене подкласс ничего не знает, и, соответственно, попытка обратиться к такому неизвестному для подкласса члену напрямую ведет к ошибке. Однако технически ненаследуемые члены в подклассе существуют, о чем свидетельствует хотя бы приведенный пример. Причина такого положения дел кроется в способе создания объектов подкласса: при создании объекта подкласса сначала вызывается конструктор суперкласса, а затем уже непосредственно выполняются команды, связанные с созданием объекта подкласса. Конструктором суперкласса выделяется в памяти место для всех членов объекта, в том числе и ненаследуемых.

Пакеты и защищенные члены

Господи, прости, от страха все слова повыскакивали.

из к/ф «Формула любви»

Нам осталось расставить все точки над *i* в вопросе с *защищенными* членами суперкласса (теми, что описаны со спецификатором доступа `protected`). Но перед этим имеет смысл познакомиться с *пакетами*.

В известном смысле пакет — это контейнер для классов, их пространство имен. Современные тенденции программирования таковы, что обычно приходится создавать объемные проекты, имея дело с большим количеством классов. В принципе, каждый из этих классов должен иметь уникальное имя. Хотя гипотетически придумать названий для классов можно бесконечно много, наступает момент, когда подобный подход означает выход за рамки разумного.

В соответствии с концепцией пакетов все классы разбиваются по группам, которые и называются пакетами. Имя класса должно быть уникальным в пределах пакета. При этом неважно, есть ли в другом пакете класс с таким же именем.

Для определения пакета необходимо в файле с описанием класса (включаемого в пакет) первой командой указать инструкцию `package` и имя пакета, например:

```
package mypack;
```

В данном случае `mypack` — это имя пакета. Если пакет с таким именем уже существует, то соответствующий класс (или классы) из файла добавляется в этот пакет. Если такого пакета нет, то он создается. Таким образом, одна и та же инструкция `package` может использоваться в нескольких файлах. Однако в файле может быть или только одна инструкция `package`, или ее может не быть вовсе. В последнем случае классы попадают в так называемый *пакет по умолчанию*.

Пакет, кроме классов, может содержать *интерфейсы* (описываются позже), а также *подпакеты* (то есть другие пакеты). При указании имени подпакета (пакета, нахо-

дящегося в другом пакете) используется точечный синтаксис — имени подпакета предшествует имя пакета, а в качестве разделителя используется точка.

ПОДРОБНОСТИ

Иерархия пакетов строго соответствует структуре файловой системы и размещению в ней пакетов и классов. Например, инструкция `package java.awt.image` означает, что файлы подпакета `image` размещены в каталоге `java/awt/image`.

При работе со средой разработки IntelliJ IDEA создание проекта с новым пакетом имеет свои особенности. Так, после создания нового пустого проекта следует в контекстном меню New папки src выбрать команду Package и создать пакет. В результате будет создана папка с названием пакета. Файлы, относящиеся к соответствующему пакету, размещаются в этой папке.

Если в программе выполняется обращение к классам, размещенным во внешних пакетах, то необходимо указывать полное имя класса: через точку перечисляется вся иерархия пакетов, где размещен нужный класс. Например, если класс `MyClass` находится в подпакете `subpack` пакета `mypack`, то обращение к этому классу будет иметь вид `mypack.subpack.MyClass`.

Чтобы можно было ссылаться на классы внешних пакетов в упрощенной форме, прибегают к *импорту* пакетов. При этом используется ключевое слово `import` (соответствующая команда размещается в начале файла после команды подключения пакета). Файл может содержать несколько инструкций импорта. Допускается подключать (импортировать) отдельные классы пакета или весь пакет. В частности, для импорта класса после ключевого слова `import` указывают полное имя класса (с учетом иерархии пакетов), например:

```
import mypack.subpack.MyClass;
```

Для импорта всего пакета после имени пакета ставят звездочку *, например, так:

```
import mypack.subpack.*;
```

Для использования статических членов класса без ссылки на класс можно использовать *статический импорт*. В этом случае после ключевого слова `import` указывается ключевое слово `static`, а затем — полная иерархия к импортируемому статическому члену класса. Например:

```
import static mypack.subpack.MyClass.name;
```

В этом случае импортируется статическое поле или метод с именем `name` из класса `MyClass`, который находится в подпакете `subpack` из пакета `mypack`. Если вместо названия статического члена класса указать звездочку *, то импортируются все статические члены класса:

```
import static mypack.subpack.MyClass.*;
```

В данном случае импортируются все статические члены из класса `MyClass`.

НА ЗАМЕТКУ

Существуют некоторые ограничения, накладываемые на импорт пакетов.

- Импортировать можно только открытые классы (в описании класса перед ключевым словом `class` должен быть указан спецификатор `public`).
- Пакет `java.lang` (базовая библиотека) можно не импортировать — он доступен по умолчанию.
- Имя файла должно совпадать с именем открытого класса, если такой класс есть в файле.
- Если в пакете несколько открытых классов, они должны размещаться в разных файлах (в одном файле может быть только один открытый класс).

Теперь вернемся к вопросу доступности или недоступности членов класса. Если мы пытаемся обратиться к полю или методу, то исходим из того, что это поле (метод) описано в каком-то классе. Код, в котором мы размещаем инструкцию для получения доступа к полю или методу, может находиться в том же классе либо в другом классе. Если речь идет не об одном классе, а о разных, то они могут быть как в одном, так и в разных пакетах. А еще класс, из которого мы получаем доступ, может быть связан наследованием с классом, в котором находится поле/метод. Таким образом, есть пять качественно разных ситуаций:

- Доступ к полю/методу выполняется в том же классе, в котором находится поле/метод.
- Мы пытаемся получить доступ к полю/методу из другого класса в том же пакете, и классы наследованием не связаны.
- Класс, из которого мы пытаемся получить доступ к полю/методу, находится в другом пакете и не связан наследованием с классом, в котором находится поле/метод.
- Нужно получить доступ к полю/методу суперкласса из подкласса в том же пакете.
- Доступ к полю/методу суперкласса выполняется из подкласса, который находится в другом пакете.

Кроме этого, член класса, к которому мы получаем доступ, может быть описан со спецификатором доступа `public`, `private`, `protected` или без такового. Как видим, получается достаточно много комбинаций (табл. 6.1; символ + означает наличие доступа, символ – означает отсутствие доступа).

Таблица 6.1. Доступность членов класса

Место получения доступа	private	нет	protected	public
Тот же класс	+	+	+	+
Подкласс в том же пакете	-	+	+	+
Обычный класс в том же пакете	-	+	+	+
Подкласс в другом пакете	-	-	+	+
Обычный класс в другом пакете	-	-	-	+

На основе данных, представленных в таблице, можно сформулировать несколько правил:

- Открытые члены, описанные со спецификатором `public`, доступны везде.
- Закрытые члены, описанные со спецификатором `private`, доступны только в пределах класса, в котором они объявлены.
- Открытые члены класса, описанные без спецификатора доступа, доступны в пределах своего пакета.
- Защищенные члены класса, описанные со спецификатором `protected`, доступны в пределах своего пакета, а также в подклассах, даже если они находятся в другом пакете.

Нужно, кроме того, помнить, что у классов также есть уровни доступа. Так, в описании класса можно указать спецификатор `public` (а можно и не указывать). Если класс объявлен со спецификатором `public`, то он доступен отовсюду. Если спецификатор `public` не указан, то класс доступен только в пределах своего пакета.

Конструкторы и наследование

— А нормального входа в этот универсам нет?

— За нормальный чатлами надо платить, родной.

из к/ф «Кин-дза-дза»

Создание объектов подкласса имеет свои особенности. Фактически объект подкласса создается частями: сначала создается та часть, которая определяется суперклассом, а затем достраивается часть, связанная собственно с подклассом. В прикладном плане все сводится к тому, что при создании объекта подкласса

сначала автоматически вызывается конструктор суперкласса. Если в суперклассе и подклассе используются конструкторы по умолчанию (то есть ни в суперклассе, ни в подклассе конструкторы не описаны), то вся эта кухня особого значения не имеет. Собственно, мы и до этого создавали объекты подклассов без особых проблем. Но ситуация меняется, если конструктору суперкласса необходимо передавать аргументы.

ПОДРОБНОСТИ

Если в суперклассе все конструкторы такие, что им нужно передавать аргументы, то при создании объектов подкласса возникает проблема следующего рода. Поскольку при создании объекта подкласса сначала автоматически вызывается конструктор суперкласса, то в этот конструктор как-то нужно передать аргументы, даже если непосредственно конструктор подкласса может без них обойтись. Отсюда накладывается ряд ограничений на способ описания конструктора подкласса (да и на саму необходимость такого описания). Эти ограничения сводятся к тому, что в конструкторе подкласса необходимо предусмотреть передачу аргументов конструктору суперкласса (разумеется, если такая передача аргументов вообще требуется).

Решение проблемы сводится к тому, что в таких случаях нужно описывать конструктор подкласса и в теле этого конструктора должна быть инструкция вызова конструктора суперкласса (с указанием аргументов, которые ему передаются). Для вызова конструктора суперкласса используется ключевое слово `super`, после которого в круглых скобках указываются аргументы, передаваемые конструктору суперкласса. Инструкция вызова конструктора суперкласса указывается первой командой в теле конструктора подкласса. Общий синтаксис объявления конструктора подкласса выглядит так:

```
Конструктор(аргументы){
   // Вызов конструктора суперкласса:
   super(аргументы);
   // Команды
}
```

Если в теле конструктора подкласса инструкцию `super` не указать вовсе, в качестве конструктора суперкласса вызывается конструктор без аргументов. Пример использования конструкторов при наследовании — в листинге 6.3.

Листинг 6.3. Конструкторы и наследование

```
// Суперкласс:
class Alpha{
   // Целочисленное поле:
   int number;
   // Метод для отображения значения поля:
   void show(){
      System.out.println("Поле number: "+number);
```

```
   }
   // Конструктор без аргументов:
   Alpha(){
      // Присваивание значения полю:
      number=100;
      // Отображение значения поля:
      show();
   }
   // Конструктор с целочисленным аргументом:
   Alpha(int n){
      // Присваивание значения полю:
      number=n;
      // Отображение значения поля:
      show();
   }
}
// Подкласс:
class Bravo extends Alpha{
   // Символьное поле:
   char symbol;
   // Метод для отображения значения поля:
   void display(){
      System.out.println("Поле symbol: "+symbol);
   }
   // Конструктор без аргументов:
   Bravo(){
      // Вызов конструктора суперкласса:
      super();
      // Присваивание значения символьному полю:
      symbol='A';
      // Отображение значения символьного поля:
      display();
   }
   // Конструктор с двумя аргументами:
   Bravo(int n,char s){
      // Вызов конструктора суперкласса:
      super(n);
      // Присваивание значения символьному полю:
      symbol=s;
      // Отображение значения символьного поля:
      display();
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      System.out.println("Первый объект:");
      Bravo A=new Bravo();
      System.out.println("Второй объект:");
      Bravo B=new Bravo(200,'B');
   }
}
```

В результате выполнения этой программы получаем такое:

Результат выполнения программы (из листинга 6.3)

```
Первый объект:
Поле number: 100
Поле symbol: A
Второй объект:
Поле number: 200
Поле symbol: B
```

Программа состоит из трех классов. В первом классе `Alpha` описано целочисленное поле `number`, метод `show()` для отображения значения этого поля, а также два варианта конструкторов: без аргументов и с одним аргументом. В конструкторе без аргументов полю `number` присваивается значение `100`. В конструкторе с целочисленным аргументом полю присваивается значение аргумента. В обоих случаях с помощью метода `show()` значение поля `number` отображается в области вывода.

На основе класса `Alpha` создается подкласс `Bravo`. Непосредственно в классе описывается поле `symbol` типа `char` и метод `display()` для отображения значения этого поля.

В подклассе определяются два конструктора: без аргументов и с двумя аргументами. В каждом из этих конструкторов с помощью инструкции `super` вызывается конструктор суперкласса. В конструкторе подкласса без аргументов командой `super()` вызывается конструктор суперкласса без аргументов. Если при создании объекта подкласса конструктору передаются два аргумента (типа `int` и типа `char`), то аргумент типа `int` передается конструктору суперкласса (командой `super(n)` в теле конструктора подкласса с двумя аргументами).

В главном методе программы создаются два объекта подкласса `Bravo`. В первом случае вызывается конструктор без аргументов, во втором — конструктор с двумя аргументами.

Переопределение методов

> Удивляюсь вашей принципиальности. То вы за правление, то против.
>
> *из к/ф «Гараж»*

Может сложиться ситуация, требующая изменения унаследованного из суперкласса метода в подклассе. Другими словами, иногда случается, что метод унаследован, но нас его функциональность в силу каких-то причин не устраивает. Тогда можно прибегнуть к *переопределению* метода. Идея переопределения методов очень проста: унаследованный из суперкласса и переопределяемый в подклассе метод просто

заново описывается. В таком случае, если мы будем вызывать метод из объекта суперкласса, используется версия метода, описанная в суперклассе. Если же метод вызывается из объекта подкласса, то вызывается метод, описанный в подклассе.

ПОДРОБНОСТИ

При переопределении метода в подклассе унаследованная версия метода не пропадает бесследно. Она остается доступной в подклассе. Чтобы вызывать версию метода, описанную в суперклассе и переопределенную в подклассе, перед инструкцией вызова метода указывают (через точку) ключевое слово `super`.

Также стоит учесть, что если в суперклассе в описании метода использовать ключевое слово `final`, то в подклассе такой метод переопределить нельзя.

Для повышения надежности кода перед переопределяемым методом можно (но не обязательно) размещать аннотацию (инструкцию) `@Override`. Аннотация информирует компилятор о переопределении метода из суперкласса. Это позволяет избежать ошибки, связанной с тем, что программист случайно вместо переопределения унаследованного метода описал в подклассе новый метод (или новую версию уже существующего метода).

Переопределение методов можно использовать одновременно с перегрузкой.

НА ЗАМЕТКУ

Между переопределением и перегрузкой методов существует принципиальное отличие. При перегрузке методы имеют одинаковые названия, но разные аргументы. При переопределении совпадают не только названия методов, но и тип результата и аргументы.

Переопределение реализуется при наследовании. Для перегрузки в наследовании необходимости нет. Если наследуется перегруженный метод, то переопределение выполняется для каждой его версии в отдельности. Если в подклассе какая-то версия перегруженного метода не описана, эта версия наследуется из суперкласса. Может сложиться и более хитрая ситуация. Допустим, в суперклассе определен некий метод, а в подклассе определяется метод с таким же именем, но другими аргументами. В этом случае в подклассе будут доступны обе версии метода: и версия, унаследованная из суперкласса, и версия, описанная в подклассе. То есть имеет место перегрузка метода, причем одна версия метода описана в суперклассе, а другая — в подклассе.

В листинге 6.4 приведен пример программы, в которой используется переопределение методов.

Листинг 6.4. Переопределение метода

```
// Суперкласс:
class Alpha{
   // Закрытое целочисленное поле:
   private int number;
```

```
   // Метод для отображения значения поля:
   void show(){
      System.out.println("Поле number: "+number);
   }
   // Метод для присваивания значения полю:
   void set(int n){
      number=n;
   }
   // Конструктор:
   Alpha(int n){
      set(n);
   }
}
// Подкласс:
class Bravo extends Alpha{
   // Закрытое символьное поле:
   private char symbol;
   // Метод для отображения значения полей:
   void show(){
      // Вызов версии метода из суперкласса:
      super.show();
      // Отображение значения символьного поля:
      System.out.println("Поле symbol: "+symbol);
   }
   // Метод для присваивания значений полям:
   void set(int n,char s){
      // Вызов версии метода с одним аргументом:
      set(n);
      // Присваивание значения символьному полю:
      symbol=s;
   }
   // Конструктор:
   Bravo(int n,char s){
      // Вызов конструктора суперкласса:
      super(n);
      // Присваивание значения символьному полю:
      symbol=s;
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объекта суперкласса:
      Alpha A=new Alpha(10);
      // Вызов методов из объекта суперкласса:
      A.show();
      A.set(20);
      A.show();
      // Создание объекта подкласса:
      Bravo B=new Bravo(100,'B');
      // Вызов методов из объекта подкласса:
      B.show();
      B.set(200,'Z');
```

```
        B.show();
        B.set(300);
        B.show();
    }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 6.4)

```
Поле number: 10
Поле number: 20
Поле number: 100
Поле symbol: B
Поле number: 200
Поле symbol: Z
Поле number: 300
Поле symbol: Z
```

Разберем код и результат его выполнения. В программе описывается класс `Alpha` (суперкласс), на основе которого создается подкласс `Bravo`. Класс `Alpha` имеет закрытое целочисленное поле `number`. Кроме этого, в классе описан метод `set()` с одним аргументом для присваивания значения полю `number`, а также метод `show()` для отображения значения поля `number`. У класса есть конструктор с одним аргументом, который определяет значение поля `number` (в теле конструктора для присваивания значения полю `number` вызывается метод `set()`).

Подкласс `Bravo` создается на основе суперкласса `Alpha`. В подклассе `Bravo` объявляется закрытое символьное поле `symbol`. У конструктора класса два аргумента: первый — типа `int` для определения значения поля `number`, второй — типа `char` для определения значения поля `symbol`. Код конструктора класса `Bravo` состоит всего из двух команд: команды вызова конструктора суперкласса `super(n)` и команды присваивания значения символьному полю `symbol=s` (`n` и `s` — аргументы конструктора).

Метод `show()` в классе `Bravo` переопределяется. Название, тип результата (он не возвращается) и список аргументов (их нет) описанного в классе `Bravo` метода `show()` такие же, как у метода `show()`, описанного в классе `Alpha`. Если в классе `Alpha` методом `show()` отображается значение поля `number`, то в классе `Bravo` метод `show()` отображает еще и значение поля `symbol`. При этом в переопределенном методе `show()` вызывается также прежняя (исходная) версия метода из класса `Alpha`. Для этого используется инструкция `super.show()`.

Метод `set()` в классе `Bravo` перегружается. Хотя в классе `Alpha` есть метод с таким же названием, аргументы у методов в суперклассе и подклассе разные. В суперклассе у метода `set()` имеется один целочисленный аргумент, а в подклассе у этого же метода имеются два аргумента: целочисленный и символьный. Поэтому и в классе `Bravo` имеется два варианта метода `set()` — с одним и с двумя аргументами. Первый наследуется из суперкласса `Alpha`, а второй определен непосредственно в подклассе `Bravo`.

НА ЗАМЕТКУ

В описании метода `set()` в классе `Bravo` вызывается версия этого же метода с одним аргументом. Это та версия, которая наследуется в подклассе из суперкласса.

В главном методе программы создается объект `A` суперкласса `Alpha`, а также объект `B` подкласса `Bravo`. Из объектов вызываются методы `show()` и `set()`. Поскольку метод `show()` перегружен, то при выполнении команды `A.show()` вызывается метод `show()`, описанный в суперклассе `Alpha`, а при выполнении команды `B.show()` вызывается метод `show()`, описанный в подклассе `Bravo`. Поэтому для объекта `A` отображается значение одного (и единственного) поля, а для объекта `B` — значения двух полей.

Командой `B.set(300)` метод `set()` вызывается из объекта `B`. Поскольку в данном случае методу передан всего один аргумент, вызывается версия метода, описанная в классе `Alpha`. В результате меняется значение поля `number` (которое не наследуется, но физически существует), а поле `symbol` остается неизменным. При выполнении команды `B.set(200,'Z')` будет вызван тот вариант метода `set()`, который описан в классе `Bravo`.

У Java-методов есть важное свойство, которое проявляется при наследовании и называется *виртуальностью*. Рассмотрим виртуальность на немного экзотическом, но показательном примере (листинг 6.5):

Листинг 6.5. Виртуальность методов

```
// Суперкласс:
class Alpha{
   // Целочисленное поле:
   int number;
   // Конструктор без аргументов:
   Alpha(){
      set();
   }
   // Метод для отображения значения поля:
   void show(){
      System.out.println("Класс Alpha");
      System.out.println("Поле number: "+number);
   }
   // Метод для присваивания значения полю:
   void set(){
      number=100;
   }
   // В методе вызывается другой метод:
   void display(){
      show();
   }
}
// Подкласс:
class Bravo extends Alpha{
   // Символьное поле:
   char symbol;
```

```
   // Переопределение метода для отображения
   // значения полей:
   void show(){
      System.out.println("Класс Bravo");
      System.out.println("Поле number: "+number);
      System.out.println("Поле symbol: "+symbol);
   }
   // Переопределение метода для присваивания
   // значений полям:
   void set(){
      number=200;
      symbol='B';
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объекта суперкласса:
      Alpha A=new Alpha();
      // Вызов метода из объекта суперкласса:
      A.display();
      // Создание объекта подкласса:
      Bravo B=new Bravo();
      // Вызов метода из объекта подкласса:
      B.display();
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 6.5)

```
Класс Alpha
Поле number: 100
Класс Bravo
Поле number: 200
Поле symbol: B
```

Проанализируем код примера и постараемся понять, что в нем необычного. В суперклассе `Alpha` описано целочисленное поле `number`, конструктор и методы `set()`, `show()` и `display()` (все три — без аргументов и не возвращают результат). При вызове метода `set()` полю `number` присваивается значение `100`. Метод `show()` предназначен для отображения значения поля `number`. Метод `display()` дублирует работу метода `show()` (точнее, при вызове метода `display()` вызывается метод `show()`). В теле конструктора вызывается метод `set()`. То есть код класса `Alpha` более чем простой. На основе суперкласса `Alpha` создается подкласс `Bravo`. В подклассе `Bravo` мы описываем символьное поле `symbol`, переопределяем метод `show()` (теперь метод отображает значения двух полей), а также переопределяем метод `set()` (полю `number` присваивается значение `200`, а полю `symbol` присваивается значение `'B'`).

НА ЗАМЕТКУ

Хотя в классе `Alpha` конструктор описан, в классе `Bravo` мы конструктор не описывали. В таком случае для класса `Bravo` используется конструктор по умолчанию. При вызове этого конструктора автоматически вызывается конструктор суперкласса без аргументов. Поскольку в классе `Alpha` мы описали именно конструктор без аргументов, то он и вызывается.

В главном методе создается объект `A` суперкласса `Alpha` и объект `B` подкласса `Bravo`. Из каждого объекта вызывается метод `display()`. Что происходит в этом случае? Чтобы дать ответ, имеет смысл локализовать интригу, которая связана с двумя обстоятельствами. Метод `display()` из класса `Alpha` наследуется в классе `Bravo`. Но в этом методе, в свою очередь, вызывается метод `show()`, переопределенный в классе `Bravo`. Отсюда возникает дилемма, связанная с тем, какая версия метода `show()` будет вызвана унаследованным методом `display()` при выполнении команды `B.display()`. Нечто похожее касается и конструктора: конструктор по умолчанию класса `Bravo` вызывает конструктор класса `Alpha`, в котором вызывается метод `set()`. Но метод `set()` переопределяется в классе `Bravo`. Какая же версия этого метода будет вызвана при создании объекта класса `Bravo`? Ответ прост: в обоих случаях вызывается переопределенная версия метода. Это и есть проявление виртуальности методов.

НА ЗАМЕТКУ

Фактически версия метода определяется классом объекта, из которого вызывается метод.

Замещение полей при наследовании

> Мы вам ничего не позволим показывать.
> Мы вам сами все покажем.
>
> *из к/ф «Гараж»*

При наследовании могут возникать неоднозначные ситуации. Например, совпадение названия наследуемого подклассом поля с названием поля, описанного непосредственно в подклассе. С формальной точки зрения все выглядит так, как если бы у подкласса было два поля с одним и тем же именем: одно поле собственно подкласса и одно — полученное по наследству. Технически так и есть. Но тогда возникает вопрос о способе обращения к данным полям. По умолчанию если обращение выполняется в обычном формате, через указание имени поля, то используется поле, описанное непосредственно в подклассе. Для обращения к полю с таким же именем, унаследованным из суперкласса, перед именем поля указывают (через точку) ключевое слово `super`. Рассмотрим пример из листинга 6.6.

Листинг 6.6. Замещение полей при наследовании

```
// Суперкласс:
class Alpha{
   // Целочисленное поле:
   int code;
}
// Подкласс:
class Bravo extends Alpha{
   // Поле с тем же именем, что и в суперклассе:
   int code;
   // Метод для присваивания значений полям:
   void set(int m,int n){
      // Полю из суперкласса присваивается значение:
      super.code=m;
      // Полю из подкласса присваивается значение:
      code=n;
   }
   // Метод для отображения значений полей:
   void show(){
      // Отображение значения поля из суперкласса:
      System.out.println("Alpha: "+super.code);
      // Отображение значения поля из подкласса:
      System.out.println("Bravo: "+code);
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объекта подкласса:
      Bravo obj=new Bravo();
      // Присваивание значений полям:
      obj.set(100,200);
      // Отображение значений полей:
      obj.show();
      // Присваивание значения полю:
      obj.code=300;
      // Отображение значений полей:
      obj.show();
   }
}
```

В суперклассе `Alpha` есть целочисленное поле `code`. В подклассе `Bravo` объявляется поле такого же типа и с таким же именем. Поле `code`, объявленное в подклассе `Bravo`, замещает поле `code`, унаследованное из суперкласса `Alpha`. Но унаследованное поле `code` физически существует. Если мы в классе `Bravo` используем инструкцию `code`, то это — обращение к полю, объявленному в подклассе. Если же нужно получить доступ к полю `code`, унаследованному из суперкласса, то используем инструкцию `super.code`. Примеры таких инструкций есть в описании методов `set()` (используется для присваивания значений полям) и `show()` (используется для отображения значений полей). В главном методе создается объект `obj` класса `Bravo` и из объекта вызываются методы `set()` и `show()`. Результат выполнения программы такой:

Результат выполнения программы (из листинга 6.6)

```
Alpha: 100
Bravo: 200
Alpha: 100
Bravo: 300
```

Стоит заметить, что если мы обращаемся к полю `code` через объект `obj`, то обращение выполняется к тому полю, которое было описано в подклассе.

Многоуровневое наследование

> Сердце подвластно разуму. Чувства подвластны сердцу. Разум подвластен чувствам. Круг замкнулся.
>
> *из к/ф «Формула любви»*

Хотя множественное наследование (наследование сразу нескольких классов) в Java не допускается, может использоваться *многоуровневое наследование*. В этом случае подкласс становится суперклассом для другого подкласса (листинг 6.7).

Листинг 6.7. Многоуровневое наследование

```
// Суперкласс:
class Alpha{
   int alpha;
   Alpha(int a){
      alpha=a;
      System.out.println("Поле alpha:   "+alpha);
   }
}
// Подкласс класса Alpha:
class Bravo extends Alpha{
   int bravo;
   Bravo(int a,int b){
      super(a);
      bravo=b;
      System.out.println("Поле bravo:   "+bravo);
   }
}
// Подкласс класса Bravo:
class Charlie extends Bravo{
   int charlie;
   Charlie(int a,int b,int c){
      super(a,b);
      charlie=c;
      System.out.println("Поле charlie: "+charlie);
   }
}
```

```
// Главный класс:
class Demo{
   public static void main(String[] args){
      Charlie obj=new Charlie(1,2,3);
   }
}
```

Как видим, класс `Alpha` является суперклассом для подкласса `Bravo`. Класс `Bravo`, в свою очередь, является суперклассом для подкласса `Charlie`. Таким образом, получается своеобразная иерархия классов. В классе `Alpha` — всего одно числовое поле `alpha` и конструктор с одним аргументом. Аргумент определяет значение поля создаваемого объекта. Кроме того, отображается сообщение о значении поля объекта.

В классе `Bravo` наследуется поле `alpha` из класса `Alpha` и появляется еще одно поле `bravo`. Соответственно, конструктор имеет два аргумента. Первый передается конструктору суперкласса (класс `Alpha`), а второй определяет значение нового поля `bravo`. Также отображается сообщение о значении этого поля, однако прежде сообщение о значении поля `alpha` отображается конструктором суперкласса.

Два поля, `alpha` и `bravo`, наследуются в классе `Charlie`. Там же описано числовое поле `charlie`. Первые два аргумента конструктора передаются конструктору суперкласса (класса `Bravo`), а третий присваивается в качестве значения полю `charlie`. В конструкторе класса `Charlie` имеется команда для отображения в области вывода значения этого поля. Значения полей `alpha` и `bravo` отображаются при выполнении конструктора суперкласса.

В главном методе программы командой `Charlie obj=new Charlie(1,2,3)` создается объект класса `Charlie`. Результат представлен ниже:

Результат выполнения программы (из листинга 6.7)

```
Поле alpha:   1
Поле bravo:   2
Поле charlie: 3
```

С помощью многоуровневого наследования можно создавать достаточно сложные иерархические структуры классов. Особенно эффективным механизм многоуровневого наследования становится при одновременном использовании перегрузки и переопределения методов (листинг 6.8).

Листинг 6.8. Перегрузка и переопределение методов

```
// Суперкласс:
class Alpha{
   void show(){
      System.out.println("Метод класса Alpha");
   }
}
// Подкласс класса Alpha:
class Bravo extends Alpha{
```

```
   void show(String msg){
      System.out.println(msg);
   }
}
// Подкласс класса Bravo:
class Charlie extends Bravo{
   void show(){
      System.out.println("Метод класса Charlie");
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      Alpha A=new Alpha();
      Bravo B=new Bravo();
      Charlie C=new Charlie();
      A.show();
      B.show();
      B.show("Класс Bravo");
      C.show();
      C.show("Класс Charlie");
   }
}
```

Как и в предыдущем примере, создается иерархическая цепочка из трех классов: в вершине находится суперкласс `Alpha`, на основе которого создается подкласс `Bravo`, в свою очередь являющийся суперклассом для подкласса `Charlie`. При этом классами наследуется, перегружается или переопределяется описанный в классе `Alpha` метод `show()`. В частности, метод `show()` из класса `Alpha` не имеет аргументов и отображает сообщение `Метод класса Alpha`. В классе `Bravo` этот метод наследуется. Кроме того, в классе `Bravo` метод `show()` перегружен с текстовым аргументом так, что он отображает значение аргумента. В классе `Charlie` версия метода `show()` без аргументов переопределяется, а версия этого метода с текстовым аргументом наследуется из класса `Bravo`.

В главном методе программы создаются три объекта — по объекту для каждого из классов. Затем из каждого объекта вызывается метод `show()` (с аргументами или без, в зависимости от того, из какого именно объекта вызывается метод). В результате мы получаем следующее:

Результат выполнения программы (из листинга 6.8)

```
Метод класса Alpha
Метод класса Alpha
Класс Bravo
Метод класса Charlie
Класс Charlie
```

Из объекта класса `Alpha` вызывается версия метода без аргументов. Из объекта класса `Bravo` метод вызывается без аргументов (версия метода из класса `Alpha`)

и с текстовым аргументом (версия метода, описанная в классе `Bravo`). Вызываемая из объекта класса `Charlie` версия метода без аргументов описана в классе `Charlie`, а версия метода с текстовым аргументом наследуется из класса `Bravo`.

Объектные переменные суперкласса

> Нет, это не Жазель. Жазель была брунетка.
> А эта вся белая.
>
> *из к/ф «Формула любви»*

В наследовании было бы мало пользы, если бы не одно важное и интересное свойство объектных переменных суперкласса: они могут ссылаться на объекты подкласса. Но здесь есть и важное ограничение: через объектную переменную суперкласса можно ссылаться только на те члены подкласса, которые наследуются из суперкласса (при этом унаследованные члены могут переопределяться в подклассе).

ПОДРОБНОСТИ

Объектная переменная — это переменная, значением которой является ссылка на объект (адрес объекта). Объектная переменная объявляется так же, как обычная переменная, но в качестве типа переменной указывается имя класса. А объект создается с помощью инструкции `new` и конструктора класса. Раньше у нас всегда тип (класс) объектной переменной совпадал с классом объекта, на который переменная ссылалась. Но может быть и так, что объектная переменная относится к суперклассу, а объект, на который переменная ссылается, создается на основе подкласса этого суперкласса.

Пример ссылки объектной переменной суперкласса на объект подкласса представлен в листинге 6.9.

Листинг 6.9. Объектная переменная суперкласса

```
// Суперкласс:
class Alpha{
   char symb;
   void set(char s){
      symb=s;
   }
   void show(){
      System.out.println("Класс Alpha");
      System.out.println("Символ: "+symb);
   }
}
// Подкласс:
class Bravo extends Alpha{
```

```
   int num;
   void set(char s,int n){
      symb=s;
      num=n;
   }
   void show(){
      System.out.println("Класс Bravo");
      System.out.println("Символ: "+symb);
      System.out.println("Число: "+num);
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Объектная переменная суперкласса:
      Alpha A;
      // Создание объекта подкласса:
      Bravo B=new Bravo();
      // В объектную переменную суперкласса
      // записывается ссылка на объект подкласса:
      A=B;
      // Вызов методов через переменную подкласса:
      B.set('B',100);
      B.show();
      // Вызов методов через переменную суперкласса:
      A.set('A');
      A.show();
   }
}
```

В данном случае описывается суперкласс `Alpha`, на основе которого создается подкласс `Bravo`. В суперклассе `Alpha` объявлено символьное поле `symb` и методы `set()` и `show()`. Метод `show()` не имеет аргументов и отображает сообщение с названием класса и значением поля `symb`. Метод `set()` имеет один аргумент, который присваивается в качестве значения полю `symb`.

Поле `symb` наследуется в классе `Bravo`. В этом классе также описывается целочисленное поле `num`. Метод `set()` перегружается так, чтобы ему можно было передавать два аргумента, определяющих значения полей `symb` и `num`. Перегружается и метод `show()` — так, чтобы отображались значения обоих полей.

В главном методе программы командой `Alpha A` объявляется объектная переменная `A` класса `Alpha`. Командой `Bravo B=new Bravo()` создается объект класса `Bravo`, и ссылка на этот объект присваивается в качестве значения объектной переменной `B` класса `Bravo`. Затем командой `A=B` ссылка на тот же объект присваивается в качестве значения объектной переменной `A`. Таким образом, в результате и объектная переменная `A`, и объектная переменная `B` ссылаются на один и тот же объект. То есть переменных две, а объект один. Тем не менее ссылка на объект через переменную `A` является ограниченной — через нее можно обращаться не ко всем членам объекта класса `Bravo`.

Командой `B.set('B',100)` полям `symb` и `num` объекта присваиваются значения `'A'` и `100` соответственно. Командой `B.show()` значения полей объекта отображаются в области вывода. Для этого вызывается версия метода `show()`, описанная в классе `Bravo`. Командой `A.set('A')` меняется значение поля `symb`. Для этого вызывается версия метода `set()`, описанная в классе `Alpha` и наследуемая в классе `Bravo`. Вызвать через объектную переменную `A` версию метода `set()` с двумя аргументами не получится — эта версия не описана в суперклассе `Alpha`, поэтому через объектную переменную суперкласса версия метода с двумя аргументами недоступна. Однако командой `A.show()` вызывается переопределенный в классе `Bravo` метод `show()`. Результат выполнения программы такой:

Результат выполнения программы (из листинга 6.9)

```
Класс Bravo
Символ: B
Число: 100
Класс Bravo
Символ: A
Число: 100
```

Отметим также, что в силу отмеченных выше причин через объектную переменную `A` можно обратиться к полю `symb` объекта подкласса, но нельзя обратиться к полю `num`.

ПОДРОБНОСТИ

При вызове метода версия метода определяется не типом объектной переменной, через которую вызывается метод, а типом объекта, на который ссылается переменная.

Хотя возможность ссылаться на объекты подклассов через объектные переменные суперклассов может показаться не очень полезной, она на самом деле более чем важна в наследовании. Рассмотрим еще один небольшой пример (листинг 6.10).

Листинг 6.10. Использование объектной переменной

```
// Суперкласс:
class Alpha{
   void show(){
      System.out.println("Класс Alpha");
   }
}
// Подкласс класса Alpha:
class Bravo extends Alpha{
   void show(){
      System.out.println("Класс Bravo");
   }
}
// Подкласс класса Alpha:
class Charlie extends Alpha{
```

```
   void show(){
      System.out.println("Класс Charlie");
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объектов:
      Alpha A=new Alpha();
      Bravo B=new Bravo();
      Charlie C=new Charlie();
      // Переменная суперкласса:
      Alpha R;
      // Переменная ссылается на объект класса Alpha:
      R=A;
      R.show();
      // Переменная ссылается на объект класса Bravo:
      R=B;
      R.show();
      // Переменная ссылается на объект класса Charlie:
      R=C;
      R.show();
   }
}
```

В программе описывается суперкласс `Alpha`, на основе которого создаются два класса — `Bravo` и `Charlie`. В классе `Alpha` описан метод `show()`, действие которого сводится к отображению сообщения `Класс Alpha`. В каждом из классов `Bravo` и `Charlie` этот метод переопределяется. Версия метода `show()` из класса `Bravo` отображает сообщение `Класс Bravo`, а версия этого же метода из класса `Charlie` отображает сообщение `Класс Charlie`.

В главном методе программы создаются объекты `A`, `B` и `C` классов `Alpha`, `Bravo` и `Charlie` соответственно, а также объявляется объектная переменная `R` класса `Alpha`. Далее этой объектной переменной последовательно в качестве значений присваиваются ссылки на объекты `A`, `B` и `C` (командами `R=A`, `R=B` и `R=C`). Поскольку класс `Alpha` является суперклассом и для класса `Bravo`, и для класса `Charlie`, данные операции возможны. Причем после каждого такого присваивания через объектную переменную `R` командой `R.show()` вызывается метод `show()`. Результат выполнения программы имеет такой вид:

Результат выполнения программы (из листинга 6.10)

```
Класс Alpha
Класс Bravo
Класс Charlie
```

Хотя формально во всех трех случаях команда `R.show()` вызова метода `show()` одна и та же, результат разный в зависимости от того, на какой объект в данный момент ссылается объектная переменная `R`.

НА ЗАМЕТКУ

Все классы в Java связаны через наследование в иерархическую структуру, в вершине которой находится класс `Object`. Все классы, которые создаются без явного указания суперкласса, автоматически являются подклассами класса `Object`. Поэтому даже в тех классах, которые мы в программе создаем с нуля, методов на самом деле немножко больше, чем мы фактически описали. Но не это главное. Главное то, что поскольку класс `Object` является суперклассом (прямым или через цепочку наследования) для любого класса, то объектная переменная класса `Object` может ссылаться на объект любого класса. Но такая переменная ничего не знает о фактическом типе объекта, на который она ссылается. Поэтому обычно прибегают к явному приведению типов.

Абстрактные классы

Статуя здесь ни при чем. Она тоже женщина несчастная. Она графа любит.

из к/ф «Формула любви»

Методы и классы могут быть *абстрактными*. Абстрактный метод — это метод, тело которого в классе не описано. Для абстрактного метода объявляется только сигнатура (тип результата, имя и список аргументов). В объявлении абстрактного метода указывается ключевое слово `abstract`. Класс, который содержит хотя бы один абстрактный метод, тоже называется абстрактным. Описание абстрактного класса начинается с ключевого слова `abstract`.

Абстрактный класс в силу очевидных причин не может использоваться для создания объектов. Поэтому абстрактные классы обычно являются суперклассами для подклассов. При этом в подклассе абстрактные методы абстрактного суперкласса должны быть определены в явном виде (иначе подкласс тоже будет абстрактным).

НА ЗАМЕТКУ

Хотя на основе абстрактного класса нельзя создать объект, допускается объявлять объектные переменные абстрактного класса. Такие переменные могут ссылаться на объекты подклассов, наследующих абстрактный суперкласс.

Пример использования абстрактного класса (посредством описания с последующим созданием подкласса) приведен в листинге 6.11.

Листинг 6.11. Абстрактный класс

```
// Абстрактный суперкласс:
abstract class Alpha{
   // Абстрактный метод:
   abstract void first();
```

```
   // Обычный метод:
   void second(){
      System.out.println("Второй метод");
   }
}
// Подкласс:
class Bravo extends Alpha{
   // Определение наследуемого абстрактного метода:
   void first(){
      System.out.println("Первый метод");
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Объект подкласса:
      Bravo B=new Bravo();
      // Вызов методов:
      B.first();
      B.second();
      // Объектная переменная абстрактного класса:
      Alpha A;
      // Ссылка на объект подкласса:
      A=B;
      // Вызов методов:
      A.first();
      A.second();
   }
}
```

Суперкласс `Alpha` содержит абстрактный метод `first()` и обычный (не абстрактный) метод `second()`. Оба метода наследуются в классе `Bravo`. Но поскольку метод `first()` — абстрактный, он описан в классе `Bravo`.

Методом `first()` отображается сообщение `Первый метод`, а методом `second()` — сообщение `Второй метод`. В главном методе программы создается объект `B` подкласса `Bravo`, а затем из этого объекта последовательно вызываются оба метода. Кроме того, объявляется объектная переменная `A` абстрактного класса `Alpha` и в качестве значения переменной присваивается ссылка на созданный ранее объект класса `Bravo`. Через эту переменную также вызываются методы `first()` и `second()`. Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 6.11)

```
Первый метод
Второй метод
Первый метод
Второй метод
```

В данном случае важно то, что через переменную `A` мы имеем доступ к методу `first()`, который создавался фактически в классе `Bravo`. Но поскольку объявлен

метод (пускай даже как абстрактный) в классе `Alpha`, то через объектную переменную класса `Alpha` к этому методу можно получить доступ.

Что касается практического использования абстрактных классов, то обычно абстрактный класс служит своеобразным шаблоном, определяющим, что должно быть в подклассах. Конкретная же реализация методов выносится в подклассы. Это нередко позволяет избежать ошибок, поскольку, будь у суперкласса только неабстрактные наследуемые методы, было бы сложнее контролировать их переопределение в подклассах. Если же не определить в подклассе абстрактный метод, то при компиляции неизбежна ошибка.

Анонимные классы

> За это вам наша искренняя сердечная благодарность.
>
> *из к/ф «Формула любви»*

Мы уже знаем, что бывают анонимные объекты. Это одноразовые объекты, ссылки на которые мы не записываем в объектные переменные. Потребность в анонимном объекте может появиться, когда объект используется один раз в одном месте. Но аналогичная ситуация может возникнуть и в отношении класса. Тогда можно прибегнуть к помощи *анонимного класса*. Фактически анонимный класс — это класс, у которого нет названия. При этом на основе анонимного класса можно создать объект. Чтобы легче было понять принцип использования анонимных классов, мы сразу перейдем к практической стороне вопроса.

Итак, анонимный класс появляется в команде создания объекта — объекта, который создается на основе этого анонимного класса. Другими словами, анонимный класс создается одновременно с созданием объекта на основе этого класса. Анонимный класс создается наследованием суперкласса — обычно абстрактного. Поэтому ссылку на созданный объект можно записать в объектную переменную абстрактного суперкласса. Сам объект создается традиционно с использованием оператора `new`, но поскольку имени у класса, на основе которого создается объект, нет, то указывается имя суперкласса и список аргументов, которые передаются конструктору анонимного класса. Затем следует блок из фигурных скобок, в которых описываются абстрактные методы из абстрактного класса (хотя в этом блоке можно переопределить и обычные методы из суперкласса). Общий синтаксис команды создания объекта на основе анонимного класса таков:

```
Суперкласс переменная=new Суперкласс(аргументы){
   // Описание методов
};
```

При выполнении команды такого вида создается объект, ссылка на который записывается в объектную `переменную` типа `Суперкласс`. Объект создается на основе

анонимного (у него нет имени) класса, который является подклассом `Суперкласса`, и абстрактные (как правило) методы из `Суперкласса` в этом анонимном классе реализованы так, как они описаны в команде создания объекта.

ПОДРОБНОСТИ

У анонимного класса есть имя — технический идентификатор, который автоматически генерируется программой. Но нам он неизвестен (да и не нужен).

Теперь рассмотрим небольшой пример (листинг 6.12).

Листинг 6.12. Анонимный класс

```
// Абстрактный класс:
abstract class MyClass{
   // Целочисленное поле:
   int code;
   // Конструктор:
   MyClass(int n){
      code=n;
   }
   // Абстрактный метод:
   abstract void show();
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Объект создается на основе анонимного класса:
      MyClass A=new MyClass(100){
         void show(){
            System.out.println("Объект A: "+code);
         }
      };
      // Объект создается на основе анонимного класса:
      MyClass B=new MyClass(200){
         void show(){
            System.out.println("Объект B: "+code);
         }
      };
      // Вызов метода:
      A.show();
      B.show();
      // Изменение значения поля:
      A.code=150;
      B.code=250;
      // Вызов метода:
      A.show();
      B.show();
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 6.12)

```
Объект A: 100
Объект B: 200
Объект A: 150
Объект B: 250
```

Как видим, в программе описан абстрактный класс `MyClass`, у которого есть целочисленное поле `code`, конструктор с одним целочисленным аргументом (определяет значение поля), а также объявлен абстрактный метод `show()`. В главном методе программы создается два объекта. Для создания первого объекта использована следующая инструкция:

```
MyClass A=new MyClass(100){
   void show(){
      System.out.println("Объект A: "+code);
   }
};
```

В данном случае объект создается на основе класса, который является подклассом класса `MyClass`. В этом подклассе метод `show()` реализован так:

```
void show(){
   System.out.println("Объект A: "+code);
}
```

При создании объекта конструктору передается значение `100`, а ссылка на созданный объект записывается в объектную переменную `A` класса `MyClass`.

Аналогичным образом создается еще один объект:

```
MyClass B=new MyClass(200){
   void show(){
      System.out.println("Объект B: "+code);
   }
};
```

В переменную `B` записывается ссылка на объект, который создается на основе анонимного класса, который является подклассом класса `MyClass`. Метод `show()` в этом подклассе определен следующим образом:

```
void show(){
   System.out.println("Объект B: "+code);
}
```

При создании объекта конструктору передается аргумент `200`.

После создания объектов командами `A.show()` и `B.show()` вызывается метод `show()`, при этом отображаются те значения, которые переданы конструктору при созда-

нии объектов. Если изменить значения полей объектов (командами `A.code=150` и `B.code=250`), то при вызове метода `show()` образуются новые значения.

НА ЗАМЕТКУ

На несколько обстоятельств хочется обратить внимание. Во-первых, переменные `A` и `B` ссылаются на объекты, у которых одинаковый набор полей и методов, но методы при этом определены по-разному. Во-вторых, хотя переменные `A` и `B` относятся к классу `MyClass`, объекты, на которые они ссылаются, относятся к разным классам. Оба этих класса анонимные, оба являются подклассами класса `MyClass`. Но это разные классы.

Анонимный класс может создаваться не только на основе абстрактного. Другими словами, суперклассом для анонимного класса может быть и обычный класс. Вот небольшой пример.

НА ЗАМЕТКУ

Речь идет о событиях, которые обрабатываются в программах с графическим интерфейсом. Подробнее эта тема обсуждается немного позже. Здесь же — первая ласточка.

Чтобы легче было понять код, начнем с результатов выполнения программы. При запуске программы появляется окно с сообщением (рис. 6.1).

Во-первых, текст сообщения реализован с помощью такого графического компонента, как *текстовая метка*. Текст с сообщением в области окна отображается курсивным шрифтом Arial синего цвета. Во-вторых, эта метка реагирует на действия пользователя: если навести курсор мыши на область метки, то текст сообщения изменится и отображаться будет не синим, а красным цветом (рис. 6.2).

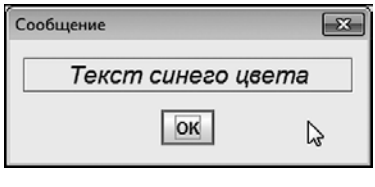


Рис. 6.1. Окно с текстовой меткой появляется при запуске программы

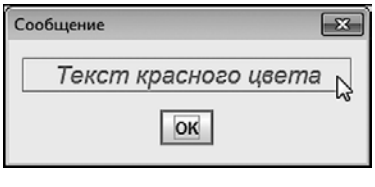


Рис. 6.2. При наведении курсора мыши на область метки меняется текст сообщения и цвет шрифта

Если убрать курсор мыши из области метки, то все вернется к исходному состоянию (текст сообщения и цвет шрифта — см. рис. 6.1). Такая нехитрая функциональность реализуется за счет обработки событий, а они, в свою очередь, обрабатываются с использованием объекта, созданного на основе анонимного класса. Чтобы понять, как именно это происходит, рассмотрим код, представленный в листинге 6.13.

Листинг 6.13. Активная текстовая метка

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static javax.swing.JOptionPane.*;
class Demo{
   public static void main(String[] args){
      // Текст для метки:
      String msg="Текст синего цвета";
      String txt="Текст красного цвета";
      // Создание текстовой метки:
      JLabel label=new JLabel(msg);
      // Режим выравнивания текста в метке по центру:
      label.setHorizontalAlignment(JLabel.CENTER);
      // Синий цвет для текста метки:
      label.setForeground(Color.blue);
      // Создание объекта шрифта:
      Font F=new Font("Arial",Font.ITALIC,18);
      // Применение шрифта к метке:
      label.setFont(F);
      // Рамка вокруг метки:
      label.setBorder(BorderFactory.createEtchedBorder());
      // Создание обработчика для метки:
      MouseAdapter handler=new MouseAdapter(){
         // Метод вызывается при наведении курсора
         // на область метки:
         public void mouseEntered(MouseEvent e){
            // Текст для метки:
            label.setText(txt);
            // Красный цвет шрифта для метки:
            label.setForeground(Color.red);
         }
         // Метод вызывается при выходе курсора
         // за область метки:
         public void mouseExited(MouseEvent e){
            // Текст для метки:
            label.setText(msg);
            // Синий цвет шрифта для метки:
            label.setForeground(Color.blue);
         }
      };
      // Регистрация обработчика в метке:
      label.addMouseListener(handler);
      // Отображение сообщения:
      showMessageDialog(null, // Родительское окно
         label, // Отображается метка
         "Сообщение", // Название окна
         PLAIN_MESSAGE // Нет пиктограммы
      );
   }
}
```

Все происходит в главном методе программы. Но анализ кода лучше начать с последней команды, которой вызывается метод `showMessageDialog()`. Мы подобных команд уже видели много. Необычность представленной команды состоит в том, что вторым аргументом методу передана ссылка `label` на объект метки. То есть если раньше мы отображали в диалоговом окне текст или картинки (изображения), то теперь отображается текстовая метка. Причем вся функциональность, связанная с изменением текста и параметров шрифта метки, спрятана непосредственно в метке.

Объект метки создается командой `JLabel label=new JLabel(msg)`. Аргументом конструктору класса `JLabel` передается текстовая переменная `msg`, которая определяет текст, отображаемый в метке.

ПОДРОБНОСТИ

Для использования того или иного графического компонента создается объект. Для реализации текстовых меток предназначен класс `JLabel` из библиотеки Swing (пакет `javax.swing`). Сама текстовая метка представляет собой прямоугольную область, предназначенную для отображения текста (а также изображений).

После создания объекта метки выполняется настройка параметров метки. Для этого из объекта вызываются специальные методы (с разными аргументами). Например, командой `label.setHorizontalAlignment(JLabel.CENTER)` устанавливается режим, при котором содержимое метки выравнивается вдоль горизонтали по центру.

ПОДРОБНОСТИ

Способ выравнивания содержимого метки вдоль горизонтали определяется методом `setHorizontalAlignment()`. Аргументом методу передается статическая целочисленная константа из класса `JLabel`, которая, собственно, и определяет способ выравнивания. Константа `CENTER` означает, что выравнивание выполняется по центру. Для выравнивания содержимого по левому краю используют константу `LEFT`. Константа `RIGHT` означает выравнивание по правому краю.

Цвет шрифта, которым отображается текст в метке, задается командой `label.setForeground(Color.blue)`. Аргументом методу `setForeground()` передается константа `blue` (означает использование синего цвета) из класса `Color` (пакет `java.awt`).

Для текста в метке мы задаем шрифт (сами определяем такие параметры шрифта, как тип, стиль и размер). Для этого создается объект класса `Font` (пакет `java.awt`). Аргументами конструктору класса `Font` передаются: тип шрифта `"Arial"`, константа `ITALIC` класса `Font`, означающая, что используется курсивный стиль, а также целочисленное значение `18`, определяющее размер шрифта. Для применения шрифта к метке используем команду `label.setFont(F)`.

ПОДРОБНОСТИ

Для определения стиля шрифта используется одна из целочисленных констант класса `Font`: `PLAIN` (обычный стиль), `BOLD` (жирный стиль), `ITALIC` (курсивный

стиль). Если нужно использовать жирный курсивный стиль, используют инструкцию `Font.BOLD|Font.ITALIC`.

Первым аргументом конструктору класса `Font` можно передавать текст с названием (типом) шрифта. Еще один вариант — передать текстовую константу (`MONOSPACED`, `DIALOG`, `DIALOG_INPUT`, `SERIF`, `SANS_SERIF`) класса `Font`, определяющую логический тип шрифта.

Мы добавляем рамку вокруг метки (хотя этого можно было и не делать — в данном случае рамка нужна для того, чтобы обозначить, где начинается область метки). Чтобы у метки появилась рамка, из объекта рамки `label` вызывается метод `setBorder()`. Аргументом методу в общем случае передается объект, который определяет тип и параметры рамки. Для получения такого объекта используется статический метод `createEtchedBorder()` из класса `BorderFactory` (пакет `java.awt`). Объект, переданный аргументом методу `setBorder()`, соответствует рамке, отображаемой с использованием эффекта гравировки.

Все описанные команды используются для выполнения таких декоративных настроек. Наиболее важная часть кода относится к обработке событий, связанных с движением курсора мыши. Схема обработки предусматривает создание объекта, предназначенного для обработки события (объект-обработчик), и его последующую регистрацию в компоненте (в данном случае речь о метке).

ПОДРОБНОСТИ

Все события, которые могут произойти с компонентами, разбиты по группам или классам. Если событие происходит, на основе соответствующего класса создается объект. Событиям, связанным с действиями мыши, соответствует класс `MouseEvent` (пакет `java.awt.event`). Чтобы обрабатывать события, нужен специальный объект (объект-обработчик), и этот объект должен иметь определенный набор методов. Если такой объект зарегистрирован в качестве обработчика для компонента, то при возникновении события вызывается определенный метод объекта-обработчика. Объект-обработчик создается на основе класса — но не любого, а такого, что реализует определенный *интерфейс* (интерфейсы обсуждаются позже). Но нередко (здесь все зависит от класса обрабатываемого события) есть и другой способ. Состоит он в том, что класс для объекта-обработчика создается наследованием специального *класса-адаптера*. Этот класс содержит все необходимые методы для обработки событий, но эти методы реализованы с пустым телом (то есть при вызове методов ничего не происходит). Поэтому, создавая производный класс на основе класса-адаптера, мы описываем методы, которые вызываются при возникновении события. Тем самым определяется реакция компонента на обрабатываемое событие. Для событий класса `MouseEvent` классом-адаптером является класс `MouseAdapter` (пакет `java.awt.event`).

Объект обработчика создается на основе анонимного класса, который, в свою очередь, создается наследованием класса `MouseAdapter`. Ссылка на созданный объект записывается в объектную переменную `handler` (класса `MouseAdapter`). В команде создания

объекта-обработчика на основе анонимного класса `MouseAdapter` описываются два метода. Метод `mouseEntered()` вызывается в случае, если курсор наводится на область компонента (в котором зарегистрирован обработчик). Метод `mouseExited()` вызывается в случае, если курсор покидает область компонента (в котором зарегистрирован обработчик). У обоих методов есть аргумент класса `MouseEvent` — объект события, который передается для обработки. Мы этот объект не используем, но описать обязаны, поскольку именно так объявлены методы. По этой же причине в описании обоих методов должен быть указан и спецификатор доступа `public`.

Оба метода описаны однотипно: с помощью метода `setText()` для метки задается текст (текстовое значение определяется переменными `txt` и `msg` — в зависимости от метода). Цвет шрифта задается методом `setForeground()`, аргумент которого непосредственно определяет применяемый цвет (константа `red` из класса `Color` соответствует красному цвету, а константа `blue` — синему цвету). После того как объект для обработки событий класса `MouseEvent` создан, этот объект регистрируется в компоненте, для чего используется команда `label.addMouseListener(handler)`.

ПОДРОБНОСТИ

Для регистрации обработчиков событий разных классов используются разные методы. Обработчики для событий класса `MouseEvent` регистрируются методом `addMouseListener()`. Метод вызывается из объекта компонента, для которого регистрируется обработчик. Объект обработчика передается аргументом методу.

Получается, что обработка событий, связанных с перемещением курсора мыши, сводится к следующему: если курсор мыши наводится на область метки, то для метки применяется определенный текст и цвет шрифта. Если курсор мыши покидает область метки, то для метки применяется другой текст и цвет шрифта.

Отметим, что анонимный класс, на основе которого создается объект обработчика, находится в главном методе программы. Поэтому в описании методов этого класса можно обращаться к объектной переменной `label` (которая является локальной переменной в главном методе). Альтернативный подход мог бы состоять в том, чтобы описать внутренний класс в методе `main()` (классы можно описывать непосредственно в методах), создать на его основе объект-обработчик и затем зарегистрировать этот обработчик в метке. Как выглядел бы такой код, показано в листинге 6.14 (для удобства основная часть комментариев удалена).

Листинг 6.14. Обработчик на основе внутреннего класса

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static javax.swing.JOptionPane.*;
class Demo{
   public static void main(String[] args){
      String msg="Текст синего цвета";
      String txt="Текст красного цвета";
      JLabel label=new JLabel(msg);
```

```
      label.setHorizontalAlignment(JLabel.CENTER);
      label.setForeground(Color.blue);
      Font F=new Font("Arial",Font.ITALIC,18);
      label.setFont(F);
      label.setBorder(BorderFactory.createEtchedBorder());
      // Внутренний класс:
      class Handler extends MouseAdapter{
         public void mouseEntered(MouseEvent e){
            label.setText(txt);
            label.setForeground(Color.red);
         }
         public void mouseExited(MouseEvent e){
            label.setText(msg);
            label.setForeground(Color.blue);
         }
      };
      // Создание объекта-обработчика:
      Handler handler=new Handler();
      // Регистрация обработчика в метке:
      label.addMouseListener(handler);
      showMessageDialog(
              null,label,"Сообщение",PLAIN_MESSAGE
      );
   }
}
```

Результат выполнения программы — такой же, как и в предыдущем случае.

НА ЗАМЕТКУ

Если бы мы попытались реализовать обработчик на основе обычного (не внутреннего) класса, то пришлось бы изменить структуру программы из-за необходимости организовать доступ к объекту метки из методов в классе обработчика.

Резюме

Я тебя полюбил — я тебя научу.

из к/ф «Кин-дза-дза»

- В Java на основе одних классов можно создавать другие. Такой механизм называется наследованием. При наследовании поля и методы исходного класса, который называется суперклассом, наследуются классом, создаваемым на его основе. Этот второй класс называется подклассом.
- При создании подкласса после его имени через ключевое слово `extends` указывается имя суперкласса, на основе которого создается подкласс. Наследование или ненаследование членов суперкласса в подклассе, а также уровень доступа

унаследованных из суперкласса членов определяются уровнем доступа членов в суперклассе и взаимным размещением суперкласса и подкласса (с учетом наличия пакетов).

- Чтобы защитить класс от наследования, используют ключевое слово `final`.
- При создании объекта подкласса сначала вызывается конструктор суперкласса. Инструкция вызова конструктора суперкласса в явном виде указывается в конструкторе подкласса. Для этого используется ключевое слово `super`, в круглых скобках после которого перечисляются аргументы, передаваемые конструктору суперкласса. Команда вызова конструктора суперкласса должна быть первой в теле конструктора подкласса. Если инструкцию вызова конструктора суперкласса в конструкторе подкласса не указать, то автоматически вызывается конструктор суперкласса без аргументов.
- Наследуемые методы можно переопределять. В этом случае в подклассе описывается метод с таким же названием, типом результата и списком аргументов. Старая (исходная) версия метода также доступна с помощью ключевого слова `super`. Переопределение и перегрузка методов могут использоваться одновременно. Чтобы метод нельзя было переопределить, в его описании используют ключевое слово `final`.
- В Java множественное наследование (когда подкласс создается на основе нескольких суперклассов) не поддерживается, но есть многоуровневое наследование. В этом случае подкласс служит суперклассом для другого подкласса.
- Объектные переменные суперкласса могут ссылаться на объекты подкласса. В этом случае через такие переменные доступны только члены, объявленные в суперклассе.
- В Java существуют абстрактные методы и абстрактные классы. Абстрактным является класс, который содержит хотя бы один абстрактный метод. Абстрактный метод в классе не описывается. Класс содержит только сигнатуру абстрактного метода. В описании абстрактных классов и методов используют ключевое слово `abstract`. На основе абстрактного класса нельзя создавать объекты (но можно объявить объектную переменную). Обычно абстрактные классы используются в качестве суперклассов. В таком случае в подклассе должны быть описаны все абстрактные методы из абстрактного суперкласса.
- Объекты можно создавать на основе анонимных классов (классы без имени). При создании объекта на основе анонимного класса в команде создания объекта после оператора `new` указывается конструктор суперкласса (обычно абстрактного), наследованием которого создается анонимный класс, а также в фигурных скобках описываются методы, которые реализуются в анонимном классе.

7 Интерфейсы и лямбда-выражения

История, которую мы хотим рассказать, не опирается на факты. Она настолько невероятна, что в нее просто нельзя не поверить.

из к/ф «О бедном гусаре замолвите слово»

В данной главе речь пойдет об *интерфейсах*, *лямбда-выражениях* и *ссылках на методы*. Читатели также узнают, что такое *функциональные интерфейсы* и как они используются на практике. Сразу следует отметить, что на использовании интерфейсов (и лямбда-выражений) базируется большое количество механизмов и приемов, в том числе и создание приложений с графическим интерфейсом (в части обработки событий).

Знакомство с интерфейсами

— Крупное дело?

— Давно такого не было!

из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Напомним, что в Java запрещено множественное наследование, так как оно может вызвать различные проблемы. Однако множественное наследование открывает широкие перспективы для составления эффективных программных кодов и значительно повышает гибкость программ. Выход был найден в использовании *интерфейсов*.

НА ЗАМЕТКУ

Начиная с версии Java 8, концепция интерфейсов достаточно сильно изменилась. Мы будем отталкиваться от классической концепции интерфейсов, комментируя при этом относящиеся к интерфейсам новшества, которые появились в языке программирования Java в последних версиях.

Интерфейсы в исходном своем классическом варианте во многом напоминают полностью абстрактные классы: интерфейс содержит только сигнатуры методов без описания, а также поля-константы (поля, значения которых постоянны и не могут изменяться). В последних версиях Java для методов в интерфейсах можно указывать не только сигнатуры, но и полное описание методов, как в обычных классах. В этом смысле интерфейсы стали еще больше похожи на классы.

Описание интерфейса аналогично описанию класса, только вместо ключевого слова `class` используется ключевое слово `interface`. Методы в интерфейсе или просто объявляются (без описания тела метода), или, если тело метода все же описано, в сигнатуре метода используется ключевое слово `default`. Все методы, объявленные и/или описанные в интерфейсе, автоматически считаются открытыми с уровнем доступа `public` (хотя спецификатор доступа `public` при этом не указывается).

НА ЗАМЕТКУ

Если метод в интерфейсе содержит тело, то про такие методы говорят, что они имеют код по умолчанию.

Также следует учесть, что, начиная с версии Java 9, методы, объявленные и описанные в интерфейсе, могут быть статическими или закрытыми.

Объявленные в интерфейсе поля по умолчанию считаются статическими и константными (как если бы они были описаны с ключевыми словами `final` и `static`, но соответствующие ключевые слова не указываются в объявлении полей). Общая схема объявления интерфейса выглядит так:

```
interface название{
   // Объявление полей и методов
}
```

Практическое использование интерфейса подразумевает его *реализацию* в классе. Эта процедура напоминает наследование абстрактных классов. Если класс реализует интерфейс, то он должен содержать описание всех методов из интерфейса. Если метод в интерфейсе не только объявлен, но и описан (имеет код по умолчанию), то в классе такой метод можно не описывать — в таком случае для метода используется код по умолчанию из интерфейса. Причем один и тот же класс может реализовать одновременно несколько интерфейсов (равно как один и тот же интерфейс может реализовываться несколькими классами). Это фактически является компенсацией за отсутствие механизма множественного наследования. За счет использования интерфейсов в одном классе можно объединить сразу несколько групп методов, как при множественном наследовании.

НА ЗАМЕТКУ

Проблемы, возникающие при множественном наследовании классов, как правило, связаны не с самой структурой наследования, а скорее со спецификой наследуемых методов при попытке реализации конкретного кода. За счет использования

интерфейсов проблема конечного кода сводится к минимуму, поскольку интерфейсы содержат объявления методов, а конкретная реализация этих методов выполняется в классе.

ПОДРОБНОСТИ

Если в классе, который реализует интерфейс, метод из интерфейса не описан и этот метод не имеет кода по умолчанию, то такой класс будет абстрактным и должен описываться с ключевым словом `abstract`.

Методы из интерфейса в классе описываются с ключевым словом `public` (хотя при объявлении методов в интерфейсе это ключевое слово не используется).

Если класс реализует интерфейс, то в описании этого класса после его названия указывается ключевое слово `implements` и затем — имя реализуемого интерфейса:

```
class имя implements интерфейс{
   // Описание класса
}
```

Если класс реализует сразу несколько интерфейсов, то в описании класса названия этих интерфейсов перечисляются через запятую после ключевого слова `implements`. Более того, класс может не только реализовывать интерфейсы, но еще и наследовать суперкласс. Если так, то в описании класса после его имени и ключевого слова `extends` указывают имя наследуемого суперкласса, а затем через ключевое слово `implements` перечисляются реализуемые в классе интерфейсы:

```
class имя extends суперкласс implements интерфейсы{
   // Описание класса
}
```

В листинге 7.1 приведен пример программы, в которой показано использование интерфейсов.

Листинг 7.1. Знакомство с интерфейсами

```
// Интерфейс:
interface Alpha{
   // Константное поле:
   int NUMBER=5;
   // Объявление методов:
   void set(int n);
   String get();
   // Метод с кодом по умолчанию:
   default void show(){
      System.out.println("Результат: "+get());
   }
}
// Класс реализует интерфейс Alpha:
class MyClass implements Alpha{
   // Закрытое целочисленное поле:
```

```
   private int code;
   // Описание методов из интерфейса.
   // Метод для присваивания значения полю:
   public void set(int n){
      if(n>=0) code=n;
      else code=-n;
      System.out.println("Число: "+code);
   }
   // Метод для перевода значения поля в другую
   // систему счисления:
   public String get(){
      String res="|";
      int num=code;
      do{
         res="|"+(num%NUMBER)+res;
         num/=NUMBER;
      }while(num>0);
      return res;
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объекта:
      MyClass obj=new MyClass();
      // Отображение значения поля:
      System.out.println("NUMBER: "+Alpha.NUMBER);
      // Вызов методов из объекта:
      obj.set(116);
      obj.show();
   }
}
```

Результат выполнения программы таков:

Результат выполнения программы (из листинга 7.1)

```
NUMBER: 5
Число: 116
Результат: |4|3|1|
```

В программе описан интерфейс `Alpha` и класс `MyClass`, в котором реализован этот интерфейс. В интерфейсе `Alpha` объявлено константное целочисленное поле `NUMBER` со значением `5`. Также в интерфейсе объявляются два метода — `set()` (с целочисленным аргументом, не возвращает результат) и `get()` (без аргументов, возвращает текстовый результат). Кроме этих методов, в интерфейсе описан метод `show()` с кодом по умолчанию (поэтому в описании метода использовано ключевое слово `default`).

ПОДРОБНОСТИ

В коде по умолчанию для метода `show()` вызывается метод `get()`, который в интерфейсе только объявлен. Когда интерфейс реализуется в классе, метод `get()`

будет описан, и код по умолчанию метода `show()` (если он не переопределяется в классе) будет иметь смысл.

В классе `MyClass`, как отмечалось, реализуется интерфейс `Alpha`. Поэтому в классе описываются методы `set()` и `get()` из интерфейса. В частности, в классе объявлено закрытое целочисленное поле `code`. Метод `set()` определен так, что при вызове он задает значение этого поля.

ПОДРОБНОСТИ

Метод `set()` описан так, что значением полю `code` присваивается модуль аргумента, переданного методу. В результате поле `code` получает неотрицательное значение.

Метод `get()` предназначен для перевода значения поля `code` в другую систему счисления. Основа для системы счисления определяется значением поля `NUMBER` (это поле наследуется в классе `MyClass` из интерфейса `Alpha`).

НА ЗАМЕТКУ

Если число записывается в системе счисления с основой n, то позиционное представление числа задается цифрами в диапазоне от 0 до $n - 1$ включительно. Если известно представление $a_m a_{m-1}...a_2 a_1 a_0$ числа в такой системе (параметры $a_k = 0,1,2,...,n - 1$), то в десятичную систему данное число можно перевести по формуле $a_m a_{m-1}...a_2 a_1 a_0 = a_0 n^0 + a_1 n^1 + a_2 n^2 + ... + a_{m-1} n^{m-1} + a_m n^m$.

В процессе выполнения метода `get()` формируется (и затем возвращается в качестве результата) текстовая строка с кодом числа. В качестве разделителей (для удобства восприятия) используются вертикальные черточки.

ПОДРОБНОСТИ

Текст с результатом формируется так. Значение поля `code` записывается в локальную переменную `num`. Это значение задано в десятичной системе. Нас интересует представление числа в системе счисления с основой `NUMBER`. Последняя цифра в таком представлении вычисляется командой `num%NUMBER` как остаток от деления числа на значение поля `NUMBER` (основа системы счисления). Затем при выполнении команды `num/=NUMBER` (целочисленное деление) в представлении числа (из переменной `num`) отбрасывается последняя цифра (и последней становится предпоследняя цифра).

При вычислениях мы учли следующее. Допустим, число записано в некоторой системе счисления. Если вычислить остаток от деления этого числа на основу системы счисления, получим последнюю цифру в представлении числа. Деление нацело числа на основу системы счисления приводит к отбрасыванию последней цифры в представлении числа.

В главном методе создается объект `obj` класса `MyClass`, а затем из этого объекта вызываются методы `set()` и `show()`.

НА ЗАМЕТКУ

К полю `NUMBER` мы обращаемся с указанием названия интерфейса `Alpha`. Но с таким же успехом вместо имени интерфейса мы могли бы использовать название класса `MyClass` (поскольку поле наследуется в классе).

Метод `show()` в классе `MyClass` не описан, поэтому для метода используется код по умолчанию. Но в случае необходимости мы могли бы переопределить метод `show()` в классе `MyClass`, описав там этот метод заново.

Что касается результатов выполнения программы, то число с кодом 431 в системе счисления с основанием 5 в десятичную систему переводится так: $4 \cdot 5^2 + 3 \cdot 5^1 + 1 \cdot 5^0 = 116$, что соответствует значению, которое передается методу `set()` при его вызове.

Интерфейсные переменные

Хотите обмануть мага? Боже, какая детская непосредственность. Я же вижу вас насквозь.

из к/ф «31 июня»

Подобно абстрактным классам, на основе интерфейсов нельзя создавать объекты. Но можно объявить *интерфейсную переменную*. Это переменная, типом которой указывается название интерфейса. Такая переменная может ссылаться на объект класса, в котором реализуется данный интерфейс. Как и в случае с объектными переменными суперкласса, через интерфейсную переменную можно сослаться не на все члены объекта: доступны лишь методы, объявленные в интерфейсе. С учетом того, что класс может реализовать несколько интерфейсов, а один и тот же интерфейс может быть реализован в разных классах, ситуация представляется достаточно пикантной. В листинге 7.2 приведен пример программы, в которой используются интерфейсные переменные.

Листинг 7.2. Интерфейсные переменные

```
// Интерфейс:
interface Calculator{
   int calc(int n);
}
// Класс реализует интерфейс:
class Alpha implements Calculator{
   // Метод для вычисления двойного факториала числа:
   public  int calc(int n){
      if(n==1||n==2) return n;
      else return n*calc(n-2);
```

```
   }
}
// Класс реализует интерфейс:
class Bravo implements Calculator{
   // Метод для вычисления факториала числа:
   public int calc(int n){
      if(n<1) return 1;
      else return n*calc(n-1);
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Интерфейсная переменная:
      Calculator R;
      // Создание объекта:
      R=new Alpha();
      // Проверка работы метода:
      System.out.println("5!! = "+R.calc(5));
      // Создание объекта:
      R=new Bravo();
      // Проверка работы метода:
      System.out.println("5!  = "+R.calc(5));
   }
}
```

Результат выполнения программы таков:

Результат выполнения программы (из листинга 7.2)

```
5!! = 15
5!  = 120
```

В интерфейсе `Calculator` объявлен всего один метод с названием `calc()`, целочисленным аргументом и целочисленным результатом. Классы `Alpha` и `Bravo` реализуют интерфейс `Calculator`, причем каждый по-своему. В классе `Alpha` метод `calc()` описан так, что он вычисляет двойной факториал числа, переданного аргументом методу.

НА ЗАМЕТКУ

По определению двойной факториал числа n есть произведение натуральных чисел до n той же самой четности, что и n, то есть $n!! = n(n-2)(n-4)\ldots$

В классе `Bravo` методом `calc()` вычисляется факториал числа, переданного аргументом методу.

НА ЗАМЕТКУ

Факториалом числа n называется число, равное произведению натуральных чисел от 1 до этого числа: $n! = n(n-1)(n-2)\ldots 2 \cdot 1$.

При описании метода `calc()` в обоих классах использована рекурсия (в соответствии с которой метод вызывает сам себя).

ПОДРОБНОСТИ

В классе `Alpha` метод `calc()` определяется так. Если значение аргумента `n` метода равно `1` или `2`, то это и есть результат. В противном случае результат вычисляется как произведение значения аргумента `n` на результат метода, вызванного с аргументом `n-2`. При вычислении значения данного выражения метод снова вызывает сам себя, но аргумент, который передается методу, будет меньше еще на `2`. Эта цепочка вызовов продолжается до тех пор, пока метод не будет вызван с аргументом `1` или `2`.

Примерно так же организован код метода в классе `Bravo`, а именно: если аргумент метода меньше `1`, то результатом является `1` (здесь мы учли, что по общепринятой договоренности факториал для числа `0` равен `1`). Если аргумент не меньше `1`, то результат вычисляется как произведение значения аргумента `n` и результата вызова метода с аргументом `n-1`.

В главном методе программы объявляется интерфейсная переменная `R` (типом переменной указан интерфейс `Calculator`). Команда `R=new Alpha()` создает объект класса `Alpha`, и ссылка на него записывается в переменную `R`. Так можно делать, поскольку в классе `Alpha` реализован интерфейс `Calculator`. При вызове метода `calc()` через переменную `R` (команда `R.calc(5)`) вызывается версия метода из класса `Alpha` (вычисляется двойной факториал $5!! = 1 \cdot 3 \cdot 5 = 15$). Но класс `Bravo` тоже реализует интерфейс `Calculator`, поэтому законна и команда `R=new Bravo()`, которая создает объект класса `Bravo`, а ссылка на этот объект записывается в переменную `R`. Теперь при выполнении команды `R.calc(5)` вызывается версия метода `calc()` из класса `Bravo` (вычисляется факториал $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$).

В листинге 7.3 приведена программа, в которой один класс реализует сразу несколько интерфейсов и наследует суперкласс.

Листинг 7.3. Реализация интерфейсов и наследование суперкласса

```
// Первый интерфейс:
interface Alpha{
   void apply(int n);
}
// Второй интерфейс:
interface Bravo{
   void set(int n);
}
// Суперкласс:
class Base{
   int number;
   void show(){
      System.out.println("Поле number: "+number);
   }
```

```
}
// Подкласс наследует суперкласс и реализует интерфейсы:
class MyClass extends Base implements Alpha,Bravo{
   int value;
   // Метод из первого интерфейса:
   public void apply(int n){
      number=n;
   }
   // Метод из второго интерфейса:
   public void set(int n){
      value=n;
   }
   // Переопределение метода из суперкласса:
   void show(){
      super.show();
      System.out.println("Поле value: "+value);
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Интерфейсные переменные:
      Alpha A;
      Bravo B;
      // Создание объекта:
      MyClass obj=new MyClass();
      // Интерфейсные ссылки:
      A=obj;
      B=obj;
      // Вызов методов:
      A.apply(100);
      B.set(200);
      // Проверка результата:
      obj.show();
   }
}
```

Результат выполнения программы имеет следующий вид:

Результат выполнения программы (из листинга 7.3)

```
Поле number: 100
Поле value: 200
```

Кратко поясним основные этапы реализации программы. Итак, имеются два интерфейса, `Alpha` и `Bravo`, которые реализуются классом `MyClass`. Кроме того, класс `MyClass` наследует суперкласс `Base`. В каждом из интерфейсов объявлено по одному методу: в интерфейсе `Alpha` метод `apply()`, а в интерфейсе `Bravo` метод `set()`. Оба метода не возвращают результат и имеют один целочисленный аргумент.

У суперкласса `Base` объявлено поле `int number` и определен метод `show()`, который отображает значение поля. При наследовании в подклассе `MyClass` этот метод

переопределяется так, что он отображает значения двух полей: наследуемого из суперкласса поля `number` и поля `value`, описанного непосредственно в подклассе.

В классе `MyClass` методы `apply()` и `set()` реализованы следующим образом: первый метод присваивает значение полю `number`, второй — полю `value`.

В главном методе программы создаются две интерфейсные переменные: переменная `A` типа `Alpha` и переменная `B` типа `Bravo`. Кроме того, создается объект `obj` класса `MyClass`. В качестве значений интерфейсным переменным присваиваются ссылки на объект `obj`. Это возможно, поскольку класс `MyClass` реализует интерфейсы `Alpha` и `Bravo`. Однако в силу того же обстоятельства через переменную `A` можно получить доступ только к методу `apply()`, а через переменную `B` — только к методу `set()`. Команды `A.apply(100)` и `B.set(200)` присваивают значения полям объекта `obj`, а команда `obj.show()` отображает в области вывода значения полей.

Расширение интерфейсов

Ну, а это довесок к кошмару.

из к/ф «Старики-разбойники»

Подобно классам, один интерфейс может наследовать другой интерфейс. В этом случае говорят о *расширении* интерфейса. Как и при наследовании классов, при расширении интерфейсов указывается ключевое слово `extends`. Синтаксис для расширения интерфейса фактически такой же, как и синтаксис для реализации наследования классов:

```
interface имя extends интерфейс{
   // Объявление методов
}
```

В листинге 7.4 приведен пример, в котором использовано расширение интерфейса.

Листинг 7.4. Расширение интерфейса

```
// Интерфейс:
interface Alpha{
   int alpha(int n);
}
// Расширение интерфейса:
interface Bravo extends Alpha{
   int bravo(int n);
}
// Реализация интерфейса:
class MyClass implements Bravo{
   // Метод для вычисления суммы чисел:
```

```
   public int alpha(int n){
      if(n==0) return 0;
      else return n+alpha(n-1);
   }
   // Метод для вычисления числа Фибоначчи:
   public int bravo(int n){
      if(n==1||n==2) return 1;
      else return bravo(n-1)+bravo(n-2);
   }
}
class Demo{
   public static void main(String[] args){
      MyClass obj=new MyClass();
      System.out.println("Сумма: "+obj.alpha(20));
      System.out.println("Число: "+obj.bravo(20));
   }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 7.4)

```
Сумма: 210
Число: 6765
```

Код достаточно прост. Интерфейс `Alpha` содержит объявление метода `alpha()`. У метода целочисленный аргумент, и результат метода — тоже целое число. Интерфейс `Bravo` расширяет (наследует) интерфейс `Alpha`. Непосредственно в интерфейсе `Bravo` объявлен метод `bravo()` с целочисленным результатом и целочисленным аргументом. Учитывая наследуемый из интерфейса `Alpha` метод `alpha()`, интерфейс `Bravo` содержит методы `alpha()` и `bravo()`. Поэтому в классе `MyClass`, который реализует интерфейс `Bravo`, необходимо описать оба этих метода. Метод `alpha()` описывается как возвращающий результатом сумму натуральных чисел. Метод `bravo()` определен так, что возвращается число из последовательности Фибоначчи.

ПОДРОБНОСТИ

Методы `alpha()` и `bravo()` описаны в классе `MyClass` с использованием рекурсии. Метод `alpha()`, если он вызван с нулевым аргументом, возвращает нулевое значение. В противном случае результат вычисляется как сумма значения аргумента `n` и результата вызова метода с аргументом `n-1`. Таким образом, метод вызывает сам себя, пока не будет вызван с нулевым аргументом.

В последовательности Фибоначчи первые два числа равны `1`, а каждое последующее — это сумма двух предыдущих. Поэтому метод `bravo()` определен так, что если аргумент равен `1` или `2`, то возвращается `1`. Иначе результат вычисляется как сумма значений, получаемых при вызове метода с аргументами `n-1` и `n-2`. В результате по порядковому номеру числа в последовательности Фибоначчи (аргумент метода) вычисляется само это число.

В главном методе программы создается объект `obj` класса `MyClass` и последовательно вызываются методы `alpha()` и `bravo()` (в обоих случаях с аргументом `20`). Хочется верить, что результат особых комментариев не требует.

Анонимный класс на основе интерфейса

> Вообще-то, я не специалист по этим гравицапам.
>
> *из к/ф «Кин-дза-дза»*

Как мы уже знаем, анонимный класс можно создавать путем наследования класса (обычно абстрактного). Но еще можно создавать анонимный класс реализацией интерфейса. Фактически интерфейс играет роль абстрактного класса. Ссылку на созданный на основе анонимного класса объект можно записать в интерфейсную переменную. Синтаксис создания объекта на основе анонимного класса посредством реализации интерфейса выглядит так:

```
интерфейс переменная=new интерфейс(){
   // Описание методов
};
```

Для создания объекта на основе анонимного класса указывается оператор `new`, имя интерфейса и пустые круглые скобки. Далее в блоке из фигурных скобок описываются методы из интерфейса. Объект создается на основе анонимного класса, который реализует данный интерфейс. Методы в классе описаны так, как это сделано в блоке из фигурных скобок. Ссылка на созданный объект записывается в интерфейсную переменную. Пример создания и использования анонимного класса на основе интерфейса приведен в листинге 7.5.

Листинг 7.5. Анонимный класс на основе интерфейса

```
// Интерфейс:
interface Alpha{
   void set(int n);
   int get();
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объекта на основе анонимного класса:
      Alpha A=new Alpha(){
         int number;
         // Описание методов из интерфейса:
         public void set(int n){
            number=n;
         }
```

```
        public int get(){
           return number;
        }
     };
     // Вызов методов из объекта:
     A.set(123);
     System.out.println("Поле: "+A.get());
  }
}
```

Результат выполнения программы показан ниже:

Результат выполнения программы (из листинга 7.5)

```
Поле: 123
```

В данном случае мы описали интерфейс `Alpha` с двумя методами. Методу `set()` передается целочисленный аргумент, и он не возвращает результат. У метода `get()` нет аргументов, а результатом он возвращает целое число. Интерфейс `Alpha` используется в главном методе программы для создания объекта на основе анонимного класса, в частности, при помощи такой команды:

```
Alpha A=new Alpha(){
   int number;
   public void set(int n){
      number=n;
   }
   public int get(){
      return number;
   }
};
```

Этой командой создается объект, и ссылка на этот объект записывается в интерфейсную переменную `A`. Объект создается на основе класса, который реализует интерфейс `Alpha` (поэтому ссылку на объект можно записать в интерфейсную переменную типа `Alpha`). У объекта есть целочисленное поле `number`, но прямого доступа к этому полю через переменную `A` нет, поскольку через эту переменную можно получить доступ только к тем методам, которые объявлены в интерфейсе `Alpha`. Доступ к полю реализуется с помощью методов `set()` и `get()`. Они описаны так, что метод `set()` позволяет присвоить значение полю `number`, а метод `get()` позволяет получить значение поля `number`. Так, командой `A.set(123)` мы присваиваем полю значение `123`, и поэтому такое значение получаем с помощью инструкции `A.get()`.

Еще один пример использования анонимных классов, созданных на основе интерфейсов, имеет определенную практическую направленность. В рассматриваемой далее программе отображается окно с картинкой, причем картинка активная: если навести курсор на область изображения, то оно изменится.

НА ЗАМЕТКУ

Для корректной работы программы в папке `D:\Pictures` размещаются два файла, `smile.png` и `sad.png`, размером 150 пикселов в ширину и высоту каждый.

При реализации данной программы использована обработка событий, для чего создается объект-обработчик. Обработчик создается на основе анонимного класса, реализующего интерфейс `MouseListener` (листинг 7.6).

Листинг 7.6. Интерфейсы и обработка событий

```
import static javax.swing.JOptionPane.*;
import javax.swing.*;
import java.awt.event.*;
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Заголовок для окна:
      String title="Улыбнитесь!";
      // Папка с файлами изображений:
      String path="d:/pictures/";
      // Объекты изображений:
      ImageIcon smile=new ImageIcon(path+"smile.png");
      ImageIcon sad=new ImageIcon(path+"sad.png");
      // Создание объекта метки:
      JLabel label=new JLabel(sad);
      // Создание обработчика для метки:
      MouseListener handler=new MouseListener(){
         public void mouseEntered(MouseEvent e){
            label.setIcon(smile);
         }
         public void mouseExited(MouseEvent e){
            label.setIcon(sad);
         }
         // Методы с пустым телом:
         public void mouseClicked(MouseEvent e){}
         public void mouseReleased(MouseEvent e){}
         public void mousePressed(MouseEvent e){}
      };
      // Регистрация обработчика в метке:
      label.addMouseListener(handler);
      // Отображение окна с меткой:
      showMessageDialog(null,label,title,PLAIN_MESSAGE);
   }
}
```

При запуске программы появляется диалоговое окно с картинкой (рис. 7.1).

Если навести курсор мыши на область изображения, то картинка изменится (рис. 7.2).

Рис. 7.1. Окно с картинкой отображается при запуске программы

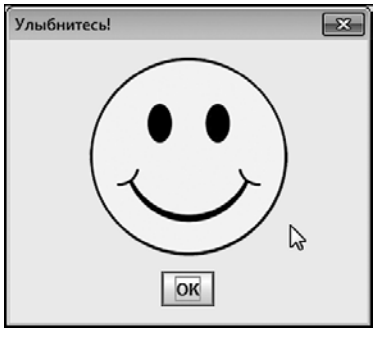


Рис. 7.2. При наведении курсора мыши на область изображения картинка меняется

Проанализируем основные моменты кода программы. Мы создаем два объекта, `smile` и `sad`, класса `ImageIcon`. Это те изображения, которые появляются в диалоговом окне. Но мы их планируем отображать в окне не напрямую, а поместив в метку. Объект метки создается командой `JLabel label=new JLabel(sad)`. Аргументом конструктору класса `JLabel` передается ссылка на объект изображения.

НА ЗАМЕТКУ

Хотя формально метка, которая реализуется на основе класса `JLabel`, текстовая, она может содержать и изображение. Изображение, которое содержит метка, может передаваться аргументом конструктору при создании метки. Если впоследствии возникнет необходимость изменить изображение в метке, это можно сделать с помощью метода `setIcon()`.

Чтобы оживить метку, создадим объект-обработчик. Для событий разного типа создаются разные обработчики. Нас интересует обработка событий класса `MouseEvent`, связанных с действиями мыши. Этот обработчик должен создаваться на основе класса, который реализует интерфейс `MouseListener`.

ПОДРОБНОСТИ

Ранее мы с подобной ситуацией уже сталкивались. Тогда мы создавали обработчик путем наследования класса-адаптера `MouseAdapter`. Класс `MouseAdapter` реализует интерфейс `MouseListener`, но только методы из интерфейса в классе описаны с пустым телом. Поэтому при создании объекта-обработчика мы их переопределяли. В данном случае мы для создания обработчика воспользуемся анонимным классом, который создается не наследованием класса-адаптера `MouseAdapter`, а реализацией интерфейса `MouseListener`. У этого подхода есть слабое место: в интерфейсе объявлено пять методов, а для работы нам нужны только два метода.

Поэтому три лишних метода придется описывать с пустым телом. В случае, когда класс для обработчика создается наследованием класса-адаптера, достаточно было переопределить только те методы, которые нужны для работы. С другой стороны, не всегда можно воспользоваться наследованием класса-адаптера. Проблема в том, что наследоваться может только один класс, а интерфейсов может быть реализовано несколько. Это важно, если мы хотим создать один обработчик для событий нескольких классов.

Объект-обработчик создается следующей командой:

```
MouseListener handler=new MouseListener(){
   public void mouseEntered(MouseEvent e){
      label.setIcon(smile);
   }
   public void mouseExited(MouseEvent e){
      label.setIcon(sad);
   }
   public void mouseClicked(MouseEvent e){}
   public void mouseReleased(MouseEvent e){}
   public void mousePressed(MouseEvent e){}
};
```

В интерфейсе `MouseListener` объявлено пять методов:

- `mouseEntered()` (вызывается при наведении курсора на область компонента);
- `mouseExited()` (вызывается, когда курсор покидает область компонента);
- `mouseClicked()` (вызывается при щелчке на компоненте);
- `mouseReleased()` (вызывается при отпускании кнопки мыши);
- `mousePressed()` (вызывается при нажатии кнопки мыши).

Мы должны описать все методы, но нужны нам только два: `mouseEntered()` и `mouseExited()`. В этих методах всего по одной команде (`label.setIcon(smile)` и `label.setIcon(sad)` соответственно), которыми задается изображение для метки. Еще три метода описываются с пустым телом. После создания объекта-обработчика он регистрируется в метке командой `label.addMouseListener(handler)`. Для отображения окна с меткой использована команда `showMessageDialog(null,label, title,PLAIN_MESSAGE)`.

НА ЗАМЕТКУ

Мы вторым аргументом методу `showMessageDialog()` передаем ссылку `label` на объект метки. Создавая и регистрируя для метки обработчик, мы получаем возможность менять изображение в метке. Другими словами, в окне отображается одна и та же метка, но картинка в метке может меняться.

Лямбда-выражения и функциональные интерфейсы

Я всегда знал, чем это все кончится.

из к/ф «Айболит-66»

Далее нам предстоит познакомиться с *лямбда-выражениями*. Этот важный механизм появился в Java относительно недавно (начиная с версии Java 8) и во многих случаях позволяет упростить структуру кода.

Лямбда-выражение можно интерпретировать как некую конструкцию, которая соответствует методу (или определяет его). Иногда удобно представлять лямбда-выражение как метод без имени (анонимный метод). Но все же, чтобы понять, что такое лямбда-выражение, имеет смысл сконцентрироваться на практической стороне вопроса. Точнее, двух вопросов. Первый состоит в том, как описывать лямбда-выражения. Второй связан с использованием лямбда-выражений.

Синтаксис описания лямбда-выражения (в базовом варианте) напоминает описание метода, но с некоторыми особенностями:

- Тип результата не указывается.
- Имя метода не указывается.
- Между круглыми скобками с аргументами и телом лямбда-выражения с командами указывается оператор «стрелка» `->`.

Таким образом, синтаксис описания лямбда-выражения следующий:

```
(аргументы)->{
   // Команды
}
```

Но как используется лямбда-выражение? Лямбда-выражение можно присвоить в качестве значения интерфейсной переменной. Но интерфейс при этом должен быть *функциональным*. Функциональный интерфейс — это интерфейс, у которого ровно один абстрактный метод (метод, который объявлен, но не описан).

НА ЗАМЕТКУ

Функциональный интерфейс может содержать несколько методов, но все они, за исключением одного, должны иметь код по умолчанию.

Если лямбда-выражение присваивается значением интерфейсной переменной (и интерфейс функциональный), то в результате такой операции автоматически создается объект, ссылка на который записывается в интерфейсную переменную. Объ-

ект создается на основе анонимного класса, реализующего данный функциональный интерфейс. При этом единственный абстрактный метод из интерфейса реализуется в соответствии с лямбда-выражением, которое присваивается интерфейсной переменной. Понятно, что структура лямбда-выражения (количество и тип аргументов, а также тип результата) должна соответствовать сигнатуре абстрактного метода из функционального интерфейса. Рассмотрим небольшой пример (листинг 7.7).

Листинг 7.7. Знакомство с лямбда-выражениями

```
// Интерфейс:
interface Alpha{
   int get(String t);
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Использование лямбда-выражений:
      Alpha A=(String t)->{
         return t.length();
      };
      Alpha B=(String t)->{
         return (int)t.charAt(0);
      };
      // Текстовая переменная:
      String t="Java";
      // Вызов метода get() из объектов A и B:
      System.out.println("Длина текста: "+A.get(t));
      System.out.println("Код символа: "+B.get(t));
   }
}
```

Результат выполнения программы таков:

Результат выполнения программы (из листинга 7.7)

```
Длина текста: 4
Код символа: 74
```

Мы описали интерфейс `Alpha` с методом `get()`. У метода текстовый аргумент и целочисленный результат. Поскольку это единственный абстрактный метод в интерфейсе, то интерфейс `Alpha` является функциональным. Поэтому интерфейсной переменной типа `Alpha` значением можно присвоить лямбда-выражение. У лямбда-выражения должен быть один аргумент текстового типа и целочисленный результат. Мы используем две переменные, `A` и `B`. Переменной `A` значение присваивается следующей командой:

```
Alpha A=(String t)->{
   return t.length();
};
```

В данном случае переменной A присваивается лямбда-выражение (String t)->{return t.length();}, которое соответствует методу с текстовым аргументом, а результатом является число, равное количеству символов в тексте. В итоге создается объект анонимного класса, ссылка на этот объект записывается в переменную A. Метод get() этого объекта определен так, что результатом возвращает длину текста, переданного аргументом.

Ниже представлена команда, которой присваивается значение переменной B:

```
Alpha B=(String t)->{
   return (int)t.charAt(0);
};
```

Лямбда-выражение (String t)->{return (int)t.charAt(0);} определяет метод, который результатом возвращает код первого символа в текстовой строке, переданной аргументом методу.

После того как объекты созданы, проверяется работа метода get(). Несложно заметить, что метод выполняется именно так, как он был определен лямбда-выражением при создании соответствующего объекта.

Если проанализировать код, связанный с использованием лямбда-выражений, то можно заметить, что лямбда-выражения в командах присваивания содержат избыточную информацию. Например, в объявлении метода get() в интерфейсе Alpha указано, что аргумент относится к типу String. Эта же информация содержится в лямбда-выражении. Ситуацию можно упростить. Точнее, можно использовать упрощенный синтаксис для лямбда-выражений. Вот основные правила:

- Если по команде присваивания возможно определить тип аргументов, в лямбда-выражении тип аргументов можно не указывать.
- Если аргумент один и его тип не указывается, то круглые скобки можно не использовать. Если аргументов нет, то используются пустые круглые скобки.
- Если тело лямбда-выражения состоит из одной команды, то фигурные скобки можно не использовать.
- Если тело метода состоит из одной return-инструкции, то инструкцию return можно не указывать (указывается только выражение, значение которого возвращается методом в качестве результата).

Например, в рассмотренном выше примере переменной A значение можно присвоить такой командой:

```
Alpha A=t->t.length();
```

Значение переменной B можно присвоить так:

```
Alpha B=t->t.charAt(0);
```

Причем в данном случае мы не используем инструкцию явного приведения к целочисленному типу: поскольку результат метода `get()` относится к типу `int`, то такое приведение будет выполнено автоматически. Следовательно, рассмотренный выше код мог бы выглядеть немного проще (листинг 7.8, комментарии удалены).

Листинг 7.8. Упрощенный синтаксис лямбда-выражений

```
interface Alpha{
   int get(String t);
}
class Demo{
   public static void main(String[] args){
      Alpha A=t->t.length();
      Alpha B=t->t.charAt(0);
      String t="Java";
      System.out.println("Длина текста: "+A.get(t));
      System.out.println("Код символа: "+B.get(t));
   }
}
```

Результат выполнения программы — точно такой же, как и в предыдущем случае.

Еще один пример с использованием лямбда-выражений касается обработки событий в приложениях с графическим интерфейсом. Сначала рассмотрим результаты выполнения программы, а затем проанализируем код. При запуске программы появляется окно с изображением льва (и снизу под изображением — имя животного, рис. 7.3).

Рис. 7.3. При запуске программы появляется окно с изображением льва

Если навести курсор на изображение и выполнить щелчок, то лев заменится на тигра (рис. 7.4).

Если еще раз щелкнуть в области изображения, то картинка с тигром заменяется на картинку с медведем (рис. 7.5).

Рис. 7.4. После щелчка по изображению льва появляется картинка с тигром

Рис. 7.5. После щелчка по изображению тигра появляется картинка с медведем

Следующий щелчок на изображении приводит к появлению самой первой картинки со львом (см. рис. 7.3), и так далее.

Рассмотрим принцип организации программы. Диалоговое окно отображается с помощью метода `showMessageDialog()` из класса `JOptionPane`. Отображаемым компонентом (второй аргумент метода) в данном случае является специально настроенная *кнопка*. Кнопка создается на основе класса `JButton` (пакет `javax.swing`). Объект класса `JButton` может содержать текст и изображение. По сути, программа работает так: щелчок на изображении на самом деле означает нажатие кнопки. При нажатии кнопки меняется изображение и текст на кнопке. Это главная идея, реализованная в программе. Теперь обратимся к коду в листинге 7.9.

Листинг 7.9. Лямбда-выражения и обработка событий

```
import static javax.swing.JOptionPane.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
// Главный класс:
class Demo{
   // Статическое поле:
   static int index=0;
   // Главный метод:
   public static void main(String[] args){
      // Путь к файлам с изображениями:
      String path="D:/Pictures/Animals/";
      // Названия животных:
      String[] names={"Лев","Тигр","Медведь"};
      // Названия файлов:
      String[] files={"lion.png","tiger.png","bear.png"};
      // Массив изображений:
      ImageIcon[] imgs=new ImageIcon[files.length];
```

```
      // Создание объектов изображений:
      for(int k=0;k<imgs.length;k++){
         imgs[k]=new ImageIcon(path+files[k]);
      }
      // Создание объекта кнопки:
      JButton button=new JButton(names[index],imgs[index]);
      // Способ выравнивания текста по вертикали:
      button.setVerticalTextPosition(JLabel.BOTTOM);
      // Способ выравнивания текста по горизонтали:
      button.setHorizontalTextPosition(JLabel.CENTER);
      // Шрифт для кнопки:
      Font F=new Font("Courier New",Font.BOLD,25);
      button.setFont(F);
      // Цвет шрифта для кнопки:
      button.setForeground(Color.BLUE);
      // Цвет фона для кнопки:
      button.setBackground(Color.WHITE);
      // Отмена отображения границ кнопки:
      button.setBorderPainted(false);
      // Отмена режима отображения рамки фокуса:
      button.setFocusPainted(false);
      // Создание и регистрация обработчика:
      button.addActionListener(e->{
         index=(index+1)%files.length;
         button.setIcon(imgs[index]);
         button.setText(names[index]);
      });
      // Альтернативный способ создания и регистрации
      // обработчика для кнопки:
      /*
      button.addActionListener(new ActionListener(){
         public void actionPerformed(ActionEvent e){
            index=(index+1)%files.length;
            button.setIcon(imgs[index]);
            button.setText(names[index]);
         }
      });
      */
      // Отображение окна с кнопкой:
      showMessageDialog(
         null,button,"Хищники",PLAIN_MESSAGE
      );
   }
}
```

Названия животных заносятся в текстовый массив `names`. Соответствующие им названия файлов с изображениями собраны в массиве `files`. Массив `imgs` состоит из объектных переменных класса `ImageIcon`. Его размер — такой же, как у массива `files`. Каждый элемент массива `imgs` ссылается на объект изображения, сформированный на основе соответствующего файла (для создания объектов изображений использован оператор цикла).

Кнопка создается командой `JButton button=new JButton(names[index],imgs[index])`. Аргументами конструктору класса `JButton` передается текст `names[index]` для отображения в кнопке и ссылка на объект изображения `imgs[index]`, которое также будет отображаться в кнопке. Мы хотим, чтобы текст отображался внизу под изображением. Поэтому задаем способ выравнивания текста по вертикали с помощью метода `setVerticalTextPosition()` (константа `BOTTOM` означает, что текст размещается в нижней части области кнопки), а также определяем способ выравнивания текста по горизонтали с помощью метода `setHorizontalTextPosition()` (константа `CENTER` означает, что текст выравнивается по центру).

Командой `Font F=new Font("Courier New",Font.BOLD,25)` создается объект для шрифта, а с помощью инструкции `button.setFont(F)` этот шрифт применяется к кнопке (в результате текст в кнопке будет отображаться назначенным шрифтом). Синий цвет шрифта для кнопки устанавливается командой `button.setForeground(Color.BLUE)`. Белый цвет фона для кнопки задаем командой `button.setBackground(Color.WHITE)`. Кроме этого, командой `button.setBorderPainted(false)` отменяется режим отображения границ кнопки, а командой `button.setFocusPainted(false)` — режим отображения рамки фокуса.

НА ЗАМЕТКУ

Если не отменить режим отображения рамки для кнопки, то вокруг кнопки будет отображаться рамка, а при наведении курсора на область кнопки рамка будет утолщаться (создавая эффект выделения кнопки). Рамка фокуса отображается в области кнопки в случае, если кнопка активна (ей передан фокус). Мы пытаемся сделать кнопку максимально похожей на метку, поэтому все такие режимы отменяем.

Кнопка может реагировать на событие класса `ActionEvent` (пакет `java.awt.event`). Событие состоит в том, что на кнопке выполняется щелчок мышью. Для обработки этого события создается объект класса, реализующего интерфейс `ActionListener` (пакет `java.awt`). В этом интерфейсе всего один абстрактный метод `actionPerformed()`, поэтому интерфейс `ActionListener` является функциональным; для создания обработчика мы можем воспользоваться лямбда-выражением. Передадим лямбда-выражение аргументом методу `addActionListiner()`:

```
button.addActionListener(e->{
   index=(index+1)%files.length;
   button.setIcon(imgs[index]);
   button.setText(names[index]);
});
```

Обработка события выполняется следующим образом. Значение поля `index` увеличивается на единицу (с учетом циклической перестановки). Поле `index` определяет индекс элементов из массивов `names` и `files`, которые используются для отображения в кнопке: командой `button.setIcon(imgs[index])` задается изображение для кнопки, а командой `button.setText(names[index])` определяется отображаемый в кнопке текст.

ПОДРОБНОСТИ

У метода `actionPerformed()`, определяемого с помощью лямбда-выражения, один аргумент класса `ActionEvent`. Метод не возвращает результат. В лямбда-выражении аргумент метода формально обозначен как `e`, но в теле лямбда-выражения он не используется.

Методу `addActionListiner()` передается объект класса, реализующего интерфейс `ActionListiner`. Под этот аргумент выделяется интерфейсная переменная. Если мы аргументом передаем лямбда-выражение, то оно присваивается в качестве значения интерфейсной переменной. В результате автоматически создается объект класса, реализующего интерфейс `ActionListiner`, и ссылка на объект записывается в интерфейсную переменную, то есть фактически передается аргументом методу `addActionListiner()`.

После того как объект для кнопки создан, а обработчик зарегистрирован, с помощью метода `showMessageDialog()` отображается диалоговое окно. При этом ссылка на объект кнопки передается вторым аргументом методу.

Также для сравнения приведем альтернативный блок кода, которым создается и регистрируется обработчик для кнопки, но только вместо лямбда-выражения задействован анонимный объект анонимного класса (в коде программы соответствующий фрагмент выделен комментарием):

```
button.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e){
      index=(index+1)%files.length;
      button.setIcon(imgs[index]);
      button.setText(names[index]);
   }
});
```

В данном случае аргументом методу `addActionListener()` передается ссылка на анонимный объект, который создается на основе анонимного класса, реализующего интерфейс `ActionListener`. Желающие могут сравнить преимущества и недостатки такого подхода.

Ссылки на методы

> Если мы допустим беспорядок в документации, потомки нам этого не простят.
>
> *из к/ф «Гостья из будущего»*

Читатели уже знают, как с помощью лямбда-выражения определить метод. Но есть и обратная процедура, позволяющая на основе существующего метода фактически определить лямбда-выражение (или, если точнее, то мы получаем некий эквивалент

лямбда-выражения). Речь идет о *ссылке на метод*. Есть несколько типов ссылок на методы:

- Ссылка на нестатический метод объекта.
- Ссылка на нестатический метод класса.
- Ссылка на статический метод класса.
- Ссылка на конструктор.

Если имеется некоторый `объект`, то у этого объекта есть `метод`, вызываемый с определенными `аргументами`, а ссылка на данный метод выполняется в формате `объект::метод`. Другими словами, указывается объект, оператор `::` (два двоеточия) и затем название метода. Ссылка `объект::метод` эквивалентна лямбда-выражению вида `(аргументы)->объект.метод(аргументы)`. Для определенности пусть речь идет об объекте `obj`, у которого есть метод `method()` с аргументами `A` и `B` (тип аргументов в данном случае неважен). Тогда ссылка `obj::method` является эквивалентом лямбда-выражения `(A,B)->obj.method(A,B)`. Рассмотрим еще один небольшой пример (листинг 7.10).

Листинг 7.10. Ссылка на метод объекта

```
// Интерфейсы:
interface Alpha{
   void apply(String t);
}
interface Bravo{
   void display();
}
// Класс:
class MyClass{
   // Поле:
   String name;
   // Методы:
   void set(String t){
      name=t;
   }
   void show(){
      System.out.println("Имя: "+name);
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объекта:
      MyClass obj=new MyClass();
      // Интерфейсные переменные и ссылки на методы:
      Alpha A=obj::set;
      Bravo B=obj::show;
      // Вызов методов:
      A.apply("Красный");
```

```
        obj.show();
        obj.set("Желтый");
        B.display();
        // Создание нового объекта:
        obj=new MyClass();
        // Вызов методов:
        obj.set("Зеленый");
        B.display();
        A.apply("Синий");
        obj.show();
        B.display();
    }
}
```

Результат выполнения программы показан ниже:

Результат выполнения программы (из листинга 7.10)

```
Имя: Красный
Имя: Желтый
Имя: Желтый
Имя: Зеленый
Имя: Синий
```

В программе описаны два функциональных интерфейса (`Alpha` и `Bravo`). В интерфейсе `Alpha` есть метод `apply()` с текстовым аргументом. Метод не возвращает результат. В интерфейсе `Bravo` объявлен метод `display()`, у которого нет аргументов и который не возвращает результат. Также мы описываем класс `MyClass`. У класса есть текстовое поле `name` и методы `set()` и `show()`. Методом `set()` на основании переданного текстового аргумента определяется значение поля `name`. Метод `show()` отображает значение поля `name`.

В главном методе программы создаем объект `obj` класса `MyClass`, объявляем интерфейсные переменные `A` и `B` и сразу присваиваем им значениями ссылки на методы. Интерфейсная переменная `A` типа `Alpha` в качестве значения получает ссылку `obj::set` на метод `set()` объекта `obj`. Что происходит в этом случае? Ссылка `obj::set` является эквивалентом лямбда-выражения вида `(String t)->obj.set(t)`. Это лямбда-выражение по своей структуре соответствует сигнатуре метода `apply()` из интерфейса `Alpha` — совпадает тип и количество аргументов, а также тип результата (он не возвращается). Когда ссылка `obj::set` присваивается переменной `A` типа `Alpha`, то автоматически создается объект анонимного класса, реализующего интерфейс `Alpha` с методом `apply()` таким, что при вызове метода с некоторым текстовым аргументом `t` на самом деле с таким же аргументом вызывается метод `set()` из объекта `obj`. Поэтому при выполнении команды `A.apply("Красный")` полю `name` объекта `obj` присваивается значение `"Красный"` (в чем мы и убеждаемся с помощью команды `obj.show()`).

Нечто похожее происходит при выполнении команды, в которой интерфейсной переменной `B` в качестве значения присваивается ссылка `obj::show` на метод `show()`

объекта obj, то есть команды, посредством которой автоматически создается объект анонимного класса, реализующего интерфейс Bravo, и в этом классе метод display() определен так, что при его вызове вызывается метод show() из объекта obj (ссылка obj::show является эквивалентом лямбда-выражения вида ()->obj.show()). Ссылка на созданный объект записывается в переменную B. Как следствие, после того как командой obj.set("Желтый") меняется значение поля name объекта obj, при выполнении команды B.display() отображается новое значение этого поля.

Еще один важный этап в работе программы связан с созданием нового объекта командой obj=new MyClass(). В этом случае в переменную obj записывается ссылка на новый созданный объект. Но переменные A и B продолжают ссылаться на объекты, связанные, в свою очередь, с методами объекта, на который ранее ссылалась переменная obj. Командой obj.set("Зеленый") полю name созданного объекта присваивается значение "Зеленый". Но на значении поля name предыдущего объекта это никак не скажется. Поэтому при выполнении команды B.display() отображается значение поля name предыдущего объекта. Аналогично, при выполнении команды A.apply("Синий") значение присваивается полю предыдущего объекта (значение поля проверяется командой B.display()). Если командой obj.show() проверить значение поля name объекта, на который в данный момент ссылается переменная obj, то станет понятно, что в этом объекте поле не изменило значения.

Ссылку можно выполнять не только на метод какого-то конкретного объекта, но и на метод класса. Ссылка на метод класса выполняется в формате класс::метод, то есть указывается имя класса, оператор :: и название метода. Такая ссылка является эквивалентом некоторого лямбда-выражения. Какого именно — зависит от того, статический метод или нет. Ели метод статический и при вызове ему передаются определенные аргументы, то ссылка класс::метод эквивалентна лямбда-выражению вида (аргументы)->класс.метод(аргументы). Допустим, в классе MyClass описан статический метод method(), у которого два аргумента (назовем их A и B). Тогда ссылка MyClass::method эквивалентна лямбда-выражению (A,B)->MyClass.method(A,B). Если ссылка выполняется на нестатический метод класса, то лямбда-выражение обретает более замысловатый вид. Ссылка вида класс::метод эквивалентна лямбда-выражению (объект,аргументы)->объект.метод(аргументы), причем здесь объект относится к классу, на метод которого выполняется ссылка. Так, если выполняется ссылка MyClass::method на нестатический метод класса MyClass с аргументами A и B, то ссылка определяет лямбда-выражение (obj,A,B)->obj.method(A,B), и здесь первый аргумент obj в лямбда-выражении является объектом класса MyClass (листинг 7.11).

Листинг 7.11. Ссылки на методы класса

```
// Класс:
class MyClass{
   // Поле:
   int code;
   // Методы:
   void set(int n){
```

```
      code=n;
   }
   void display(){
      System.out.println("Поле: "+code);
   }
   // Статический метод:
   static void show(String t){
      System.out.println("MyClass: "+t);
   }
}
// Интерфейсы:
interface Alpha{
   void alpha(MyClass obj,int n);
}
interface Bravo{
   void bravo(MyClass obj);
}
interface Charlie{
   void charlie(String t);
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объекта:
      MyClass obj=new MyClass();
      // Интерфейсные переменные:
      Alpha A=MyClass::set;
      Bravo B=MyClass::display;
      Charlie C=MyClass::show;
      // Вызов методов:
      obj.set(123);
      B.bravo(obj);
      A.alpha(obj,321);
      obj.display();
      C.charlie("Hello");
   }
}
```

Ниже приведен результат выполнения программы:

Результат выполнения программы (из листинга 7.11)

```
Поле: 123
Поле: 321
MyClass: Hello
```

В классе `MyClass` есть целочисленное поле `code`, обычные (не статические) методы `set()` и `display()`, а также статический метод `show()`. Еще в программе описаны три интерфейса. В интерфейсе `Alpha` объявлен метод `alpha()` с двумя аргументами (объект класса `MyClass` и значение типа `int`). В интерфейсе `Bravo` объявлен метод `bravo()` с аргументом, который является объектом класса `MyClass`. В интерфейсе `Charlie` объявлен метод `charlie()` с тестовым аргументом. Все эти интерфейсы

функциональные, а сигнатуры объявленных в них методов такие, что соответствуют ссылкам на методы из класса `MyClass`.

В главном методе создается объект `obj` класса `MyClass`. Переменной `A` типа `Alpha` значением присваивается ссылка `MyClass::set` на нестатический метод `set()` класса `MyClass`. Метод `set()` описан с целочисленным аргументом и как не возвращающий результат. Поэтому ссылка `MyClass::set` является эквивалентом лямбда-выражения `(MyClass obj,int n)->obj.set(n)`. Структура этого лямбда-выражения соответствует сигнатуре метода `alpha()` из интерфейса `Alpha`. В результате выполнения команды в переменную `A` записывается ссылка на объект анонимного класса, в котором реализован интерфейс `Alpha`. Интерфейс реализован так, что при вызове метода `alpha()` с аргументами `obj` типа `MyClass` и `n` типа `int` из объекта `obj` вызывается метод `set()` с аргументом `n`.

Интерфейсной переменной `B` типа `Bravo` значением присваивается ссылка `MyClass::display` на нестатический метод `display()` класса `MyClass`. Метод `display()` в классе `MyClass` описан без аргументов и как не возвращающий результат. Поэтому ссылке `MyClass::display` соответствует лямбда-выражение `(MyClass obj)->obj.display()`. Это лямбда-выражение соответствует методу `bravo()` из интерфейса `Bravo`. В итоге создается объект анонимного класса, в котором метод `bravo()` из интерфейса `Bravo` реализован так, что при вызове метода `bravo()` с аргументом `obj` типа `MyClass` из объекта `obj` вызывается метод `display()`.

Интерфейсной переменной `C` типа `Charlie` значением присваивается ссылка `MyClass::show` на статический метод `show()` класса `MyClass`. Метод `show()` описан с текстовым аргументом, поэтому ссылке `MyClass::show` соответствует лямбда-выражение `(String t)->MyClass.show(t)`. В результате присваивания переменной `C` ссылки на метод `MyClass::show` создается объект, в котором метод `charlie()` из интерфейса `Charlie` реализован так, что при его вызове с текстовым аргументом вызывается статический метод `show()` из класса `MyClass` с тем же аргументом.

После выполнения присваиваний вызываются методы. При выполнении команды `obj.set(123)` полю `code` объекта `obj` присваивается значение `123`. Когда выполняется команда `B.bravo(obj)`, из объекта `obj` вызывается метод `display()`. Команда `A.alpha(obj,321)` означает вызов из объекта `obj` метода `set()` с аргументом `321`. В результате поле `code` объекта `obj` получает значение `321` (в чем убеждаемся с помощью команды `obj.display()`). Наконец, командой `C.charlie("Hello")` из класса `MyClass` вызывается статический метод `show()` с аргументом `"Hello"`.

Ссылки можно создавать и на конструкторы. Ссылка на конструктор выполняется в формате `класс::new`. Что означает такая ссылка? Если у `класса` есть конструктор с некоторыми аргументами, то ссылка вида `класс::new` соответствует лямбда-выражению вида `(аргументы)->new класс(аргументы)`. Скажем, если в классе `MyClass` есть конструктор с аргументами `A` и `B`, то ссылка `MyClass::new` соответствует лямбда-выражению `(A,B)->new MyClass(A,B)`. Несложно догадаться, что инструкцией `new MyClass(A,B)` создается объект класса `MyClass`, а результатом является ссылка

на созданный объект. Поэтому в соответствующем функциональном интерфейсе метод, определяемый ссылкой на конструктор, аргументами должен принимать значения аргументов конструктора, а метод должен возвращать ссылку на объект класса. Пример использования ссылки на конструктор приведен в листинге 7.12.

Листинг 7.12. Ссылка на конструктор

```
// Класс:
class MyClass{
   // Поле:
   int code;
   // Метод:
   void show(){
      System.out.println("Поле: "+code);
   }
   // Конструктор:
   MyClass(int n){
      code=n;
   }
}
// Интерфейс:
interface Alpha{
   MyClass create(int n);
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Интерфейсная переменная:
      Alpha A=MyClass::new;
      // Создание объекта:
      MyClass obj=A.create(123);
      // Проверка значения поля:
      obj.show();
   }
}
```

Как выглядит результат выполнения программы, показано ниже:

Результат выполнения программы (из листинга 7.12)

```
Поле: 123
```

Мы используем простой класс `MyClass`, в котором есть целочисленное поле `code`, метод `show()` без аргументов (методом отображается значение поля `code`), а также в классе описан конструктор с одним целочисленным аргументом (он задает значение поля `code`). В интерфейсе `Alpha` описан метод `create()` с целочисленным аргументом. Метод возвращает объектную ссылку класса `MyClass`. Сигнатура этого метода соответствует параметрам конструктора класса `MyClass`. В главном методе программы интерфейсной переменной `A` типа `Alpha` значением присваивается ссылка `MyClass::new` на конструктор класса `MyClass`. В результате в переменную `A`

записывается ссылка на автоматически созданный объект. В этом объекте метод `create()` реализован так, что при вызове метода с некоторым целочисленным аргументом создается объект класса `MyClass` и конструктору класса передается аргумент метода `create()`. Результатом метод `create()` возвращает ссылку на созданный объект. Поэтому когда выполняется инструкция `A.create(123)`, создается новый объект класса `MyClass`. Конструктору при создании объекта передается значение `123`. Ссылка на созданный объект записывается в объектную переменную `obj` класса `MyClass`. Для проверки значения поля созданного объекта используем команду `obj.show()`.

Резюме

...Ну, так получилось... Отошли на секундочку — и затерялись в песках.

из к/ф «Кин-дза-дза»

- Интерфейс напоминает абстрактный класс. Он содержит поля-константы (значения полей не могут изменяться), объявления методов, а также описание методов (в этом случае используется ключевое слово `default`). Интерфейс описывается так же, как класс, но только с использованием ключевого слова `interface`.
- Интерфейсы реализуются в классах. Для реализации интерфейса в описании класса используется инструкция `implements`. Соответствующий класс должен содержать описание всех методов из интерфейса. Методы описываются как открытые со спецификатором `public`. Если метод в интерфейсе имеет код по умолчанию и в классе этот метод не описан, то в классе наследуется код по умолчанию. Один класс может реализовать несколько интерфейсов.
- Интерфейс может указываться в качестве типа переменной. Такая переменная называется интерфейсной. Интерфейсная переменная может ссылаться на объект класса, реализующего соответствующий интерфейс. Через интерфейсную переменную доступны только те методы, которые объявлены в интерфейсе.
- К интерфейсам применимо наследование. В таком случае говорят о расширении интерфейсов. По аналогии с наследованием классов в этом случае используется ключевое слово `extends`.
- Анонимные классы можно создавать на основе интерфейса. Интерфейс фактически играет роль абстрактного класса. Для создания объекта на основе анонимного класса используется оператор `new`, указывается имя интерфейса, пустые круглые скобки и в фигурных скобках описываются методы из интерфейса. Ссылка на объект, который создается на основе такого анонимного класса, может быть записана в интерфейсную переменную.

- Функциональным называется интерфейс, у которого один и только один абстрактный метод.
- Лямбда-выражение является синтаксической конструкцией, определяющей метод. Описывается лямбда-выражение подобно методу, но название и тип результата не указываются, а между круглыми и фигурными скобками размещается «стрелка» `->`. Лямбда-выражение можно присвоить значением интерфейсной переменной (при условии, что интерфейс функциональный). В таком случае в переменную записывается ссылка на автоматически созданный объект. Объект создается на основе анонимного класса, реализующего данный функциональный интерфейс. При этом лямбда-выражение определяет код абстрактного метода из функционального интерфейса.
- Ссылка на метод фактически позволяет получить эквивалент лямбда-выражения на основе уже существующего метода. Можно выполнить ссылку на метод объекта, на нестатический и статический методы класса, а также на конструктор. Ссылка на метод объекта выполняется в формате `объект::метод`, ссылка на метод класса имеет вид `класс::метод`, а ссылка на конструктор выглядит как `класс::new`. Используются ссылки на методы так же, как и лямбда-выражения.

8 Работа с текстом

Ты, значит, здесь вместо работы латынь изучаешь?

из к/ф «Формула любви»

Мы уже сталкивались с текстом и использовали текстовые объекты в программах. Теперь пришло время систематизировать знания и узнать кое-что новое.

Читатели помнят, что текст реализуется с помощью объектов. Основным классом для работы с текстом является класс `String`. Кроме класса `String` можно также использовать классы `StringBuffer` и `StringBuilder`.

НА ЗАМЕТКУ

Текстовые литералы (текст в двойных кавычках) реализуются как объекты класса `String`. Классы `StringBuffer` и `StringBuilder` позволяют создавать более функциональные объекты по сравнению с объектами, созданными на основе класса `String`. Классы `StringBuffer` и `StringBuilder` примерно эквивалентны по своим возможностям. Если в программе используется однопоточная модель, то обычно используют класс `StringBuilder`. Если программа многопоточная, то рекомендуется использовать класс `StringBuffer`.

Потоки — это блоки кода, которые выполняются одновременно. К ним мы еще вернемся.

С формальной точки зрения создание текста сводится к созданию объекта одного из этих классов. В этой главе рассмотрены все три класса. Каждый из них имеет свои особенности, хотя у них много сходств. Главное принципиальное различие состоит в том, что объекты класса `String` изменять нельзя, а объекты классов `StringBuffer` и `StringBuilder` — можно. Содержание этой главы ограничивается описанием свойств и возможностей классов `String`, `StringBuffer` и `StringBuilder`.

Классы `String`, `StringBuffer` и `StringBuilder` определены в базовом пакете `java.lang`, который доступен по умолчанию, поэтому для создания объекта класса `String`,

`StringBuffer` и `StringBuilder` импорт пакетов выполнять не нужно. Классы определены с ключевым словом `final`, и поэтому они не могут быть суперклассами для создания подклассов.

Объекты класса String

> Так, значит, русский язык знаем. Зачем потребовалось скрывать?
>
> *из к/ф «Кин-дза-дза»*

Общая схема реализации текстовых объектов — такая же, как и для реализации объектов всех прочих классов: объектная переменная, ссылающаяся на объект. Текст содержится в объекте, а доступ к этому объекту (и тексту) реализуется через объектную переменную. Ранее при работе с текстом мы в основном объявляли переменную типа `String` и присваивали ей в качестве значения текстовый литерал. Текстовые литералы реализуются как объекты класса `String`. Когда объектной переменной класса `String` присваивается текстовый литерал, то в действительности в объектную переменную просто записывается ссылка на объект, в который спрятан текстовый литерал.

Один из способов создания текстового объекта подразумевает вызов конструктора класса `String`. Заметим, что конструкторов у класса `String` довольно много.

- *Конструктор создания пустой строки.* В этом случае конструктору аргументы не передаются. Пример команды создания объекта класса `String` со значением в виде пустой строки имеет такой вид:

```
String str=new String();
```

- *Конструктор создания текстовой строки на основе символьного массива.* В этом случае аргументом конструктору передается имя массива символов. Результатом является текст, составленный из всех символов массива в порядке их размещения в массиве. Пример создания текстовой строки на основе символьного массива:

```
char symbols[]={'a','b','c'};
String str=new String(symbols);
```

  В этом конструкторе, помимо имени массива, можно указать индекс элемента массива, начиная с которого будет формироваться строка, а также длину строки в символах. Например, так:

```
char symbols={‘a','b','c','d','e','f'};
String str=new String(symbols,2,3);
```

В результате текст создается на основе не всего массива `symbols`, а лишь трех элементов, начиная с элемента с индексом `2` (получается текст `"cde"`).

- *Конструктор копирования объекта.* Аргументом конструктора указывается переменная текстового типа, ссылающаяся на уже существующий текстовый объект или текстовый литерал. В результате создается новый объект с таким же текстовым значением, как и исходный. Например:

```
String obj=new String("Текстовая строка");
String s=new String(obj);
```

Это далеко не весь список доступных конструкторов. В частности, существует конструктор, принимающий в качестве аргумента массив типа `byte` с кодами символов, которые автоматически преобразуются в буквы, а буквы — в текст. В этом конструкторе также можно указывать второй и третий аргументы — соответственно начальный индекс элемента массива, с которого начинается формирование текстовой строки, и длину строки в символах.

В листинге 8.1 представлена программа, в которой иллюстрируются различные способы создания текстовых объектов.

Листинг 8.1. Создание текстовых объектов

```
class Demo{
   public static void main(String[] args){
      // Символьный массив:
      char[] symbs={'Я','з','ы','к',' ','J','a','v','a'};
      // Создание текстового объекта на основе
      // символьного массива:
      String A=new String(symbs);
      System.out.println(A);
      String B=new String(symbs,5,4);
      System.out.println(B);
      // Копия текстового объекта:
      String C=new String(A);
      System.out.println(C);
      // Числовой массив:
      byte[] nums={65,66,67,68,69,70};
      // Создание текстового объекта на основе
      // числового массива:
      String D=new String(nums);
      System.out.println(D);
      String E=new String(nums,2,3);
      System.out.println(E);
   }
}
```

Результат выполнения программы таков:

Результат выполнения программы (из листинга 8.1)

```
Язык Java
Java
Язык Java
ABCDEF
CDE
```

В программе создается символьный массив `symbs`, а на его основе — текстовые объекты `A` и `B`. При создании первого объекта используется содержимое всего символьного массива, а при создании второго объекта в текстовую строку включается 4 символа из массива, начиная с символа с индексом `5`. В результате в объект `A` записан текст `"Язык Java"`, а в объект `B` записывается текст `"Java"`.

НА ЗАМЕТКУ

Здесь и далее, если это не будет приводить к недоразумениям, мы будем отождествлять объектную переменную с объектом или текстом, записанным в этот объект.

Текстовый объект `C` создается как копия текстового объекта `A`. В результате переменные `A` и `C` ссылаются на объекты, содержащие одинаковые текстовые значения.

ПОДРОБНОСТИ

Если двум текстовым переменным в качестве значения присвоить один и тот же текстовый литерал, то из-за работы системы оптимизации использования ресурсов литерал не станет дублироваться, а будет реализован одним и тем же объектом. В результате обе переменные будут ссылаться на один и тот же объект. То же происходит, если одной текстовой переменной в качестве значения присваивается другая текстовая переменная. В принципе, это не проблема, поскольку текстовые объекты изменять все равно нельзя (но можно сменить объект, на который ссылается объектная переменная). Если же используется конструктор класса `String` и аргументом конструктору передается текстовый объект, то в результате создается новый объект, который содержит такое же значение, что и строка, переданная аргументом.

Как отмечалось выше, текстовую строку можно создать на основе числового массива. В программе мы создаем числовой массив `nums` с кодами символов (это коды букв от `'A'` до `'F'` включительно). При создании текстового объекта `D` аргументом конструктору передается массив `nums`. В результате создается строка из символов, коды которых представлены в массиве `nums`. При создании объекта `E` конструктору передается имя массива и два целых числа (`2` и `3`). Второй аргумент `2` определяет индекс, начиная с которого элементы включаются в текстовую строку. Третий аргумент `3` определяет количество элементов, на основе которых формируется

текстовая строка. В итоге текстовый объект содержит строку "ABCDEF", а объект E содержит текстовую строку "CDE".

Обычно создание текстового объекта не является самоцелью. Важно знать, какие операции можно выполнять с такими объектами. Так, текст, записанный в объект класса String, после создания объекта изменить нельзя. Чтобы изменить уже существующий текст, необходимо создавать новый объект.

НА ЗАМЕТКУ

Некоторые методы, с помощью которых выполняется обработка текстовых строк, рассматриваются немного позже.

Узнать длину текстовой строки (в символах) можно, вызвав метод length() из текстового объекта, в котором содержится текст. Например, если str является объектом класса String, то определить длину текстовой строки, на которую ссылается объектная переменная str, можно командой str.length(). Причем поскольку текстовые литералы реализуются как объекты класса String, метод length() можно вызывать из текстовых литералов. В этом смысле вполне корректной является, например, команда "Язык Java".length() (значение выражения равно 9).

Мы уже знаем, что из базовых операций по отношению к текстовым строкам можно применять операцию сложения (с оператором +). При сложении текстовых объектов выполняется объединение (конкатенация) соответствующих текстовых значений.

ПОДРОБНОСТИ

Текстовый объект изменить нельзя. Поэтому когда складываются два текстовых объекта, в действительности создается новый объект и текст в этом новом объекте формируется конкатенацией исходных текстовых строк.

Если в команде сложения один операнд текстовый, а другой относится к нетекстовому типу (например, число), то выполняется автоматическое приведение к текстовому формату (преобразование в объект типа String). При этом могут возникать довольно неожиданные ситуации. Например, значением выражения "Число три: "+1+2 является текст "Число три: 12". Причина кроется в способе вычисления выражения. Поскольку в нем, кроме двух числовых операндов 1 и 2, присутствует еще и текстовый операнд "Число три: ", причем в выражении он идет первым (а выражение вычисляется слева направо), то к тексту "Число три: " добавляется (путем объединения строк) текстовое значение "1", после чего к полученному тексту добавляется текстовое значение "2". Чтобы предотвратить такое экзотическое сложение, вместо приведенной выше команды следует использовать инструкцию "Число три: "+(1+2). Здесь с помощью скобок изменен порядок вычисления выражения: сначала вычисляется сумма чисел, а уже после этого полученное число преобразуется в текстовый формат. В результате получается текст "Число три: 3".

НА ЗАМЕТКУ

К тексту можно прибавлять не только текст или число, но и объекты, в том числе классов, определенных пользователем. Правила преобразования объекта в текстовый формат определяются методом `toString()`. В силу особой важности этого метода рассмотрим его отдельно.

Метод toString()

> А этот пацак все время говорит на языках, продолжения которых не знает.
>
> *из к/ф «Кин-дза-дза»*

Метод `toString()` определен в классе `Object`, находящемся на вершине иерархии классов Java (это общий суперкласс Java). Метод вызывается по умолчанию при преобразовании объекта к текстовому формату следующим образом. Если ссылка на объект указана в выражении, в котором в соответствующем месте должен быть текст (например, объект передан аргументом методу `println()` или к объекту прибавляется текстовое значение), то из объекта автоматически вызывается метод `toString()`. Метод возвращает текстовое значение, которое используется вместо ссылки на объект. В классе `Object` метод `toString()` определен так, что результатом является текстовая строка с названием класса объекта и дополнительным техническим кодом (то есть строка неинформативна и в большинстве случаев для использования не годится). Однако метод `toString()` можно переопределять, и в большинстве библиотечных классов это сделано так, что метод возвращает полезную информацию об объекте. В классах, которые создаются в программе, также можно переопределить метод `toString()`.

Для переопределения `toString()` метод просто описывается в классе. Метод результатом возвращает ссылку класса `String` и не имеет аргументов. В описании метода используется спецификатор доступа `public` (листинг 8.2).

Листинг 8.2. Переопределение метода toString()

```
// Класс:
class MyClass{
   // Поля класса:
   String name;
   int code;
   char symb;
   // Конструктор:
   MyClass(String name,int code,char symb){
      this.name=name;
      this.code=code;
      this.symb=symb;
   }
   // Переопределение метода toString():
```

```
   public String toString(){
      String res="Имя: "+name+"\n";
         res+="Число: "+code+"\n";
         res+="Символ: "+symb;
      return res;
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объекта:
      MyClass A=new MyClass("Первый",100,'A');
      // Автоматический вызов метода toString():
      System.out.println(A);
      // Создание объекта:
      MyClass B=new MyClass("Второй",200,'B');
      // Автоматический вызов метода toString():
      String str="Создан новый объект\n"+B;
      // Проверка результата:
      System.out.println(str);
   }
}
```

В программе описывается класс `MyClass` с тремя полями: текстовым `name`, целочисленным `code` и символьным `symb`. У конструктора класса три аргумента, которые определяют значения полей. Также в классе переопределен метод `toString()`. При вызове метода формируется текстовая строка, которая содержит информацию о значениях полей объекта (из которого вызывается метод). После того как строка сформирована, она возвращается в качестве результата метода.

В главном методе программы создается объект `A` класса `MyClass`. Затем ссылка на объект передается аргументом методу `println()` (речь о команде `System.out.println(A)`). В этом случае автоматически из объекта `A` вызывается метод `toString()`, а текстовый результат, который возвращает метод, передается вместо ссылки `A` аргументом методу `println()`.

Еще один объект класса `MyClass` (речь об объекте `B`) используется как операнд в выражении `"Создан новый объект\n"+B`, которое в качестве значения присваивается текстовой переменной `str`. Это также пример ситуации, когда из объекта автоматически вызывается метод `toString()`. В итоге вместо ссылки на объект `B` используется текстовое значение, вычисленное при вызове из объекта `B` метода `toString()`. Результат выполнения программы имеет такой вид:

Результат выполнения программы (из листинга 8.2)

```
Имя: Первый
Число: 100
Символ: A
Создан новый объект
Имя: Второй
Число: 200
Символ: B
```

Переопределение метода `toString()` — очень удобный прием, который не только позволяет экономить усилия при отображении информации об объектах, но и создавать компактные и продуктивные программы.

Базовые операции с текстом

— Да, как эксперимент это интересно. Но какое практическое применение?

— Господи, именно практическое!

из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Конкатенация (объединение) текстовых строк — далеко не единственная операция, которую можно выполнять с текстом. Мы, например, уже знаем, что длину текста (количество символов в тексте) можно узнать с помощью метода `length()`, который вызывается из текстового объекта. Получить значение символа в тексте позволяет метод `charAt()`. Аргументом методу передается индекс символа в тексте (индексация, как и в массиве, начинается с нуля). Для преобразования текста в символьный массив используется метод `toCharArray()`.

НА ЗАМЕТКУ

Здесь еще раз уместно напомнить, что объекты класса `String` относятся к неизменяемым. Такой объект после создания изменить нельзя. Поэтому все методы из класса `String` не изменяют объект, из которого вызываются. Например, метод `toCharArray()` не делает из текста символьный массив: этот метод при вызове из текстового объекта формирует массив, составленный из букв текста, и возвращает ссылку на этот массив в качестве результата.

Если из текстового объекта вызвать метод `getBytes()` без аргументов, то мы получим массив из `byte`-значений с кодами символов из текстовой строки (определяется объектом, из которого вызывается метод).

Метод `getChars()` не возвращает результат и используется для заполнения символьного массива буквами из текста. У метода четыре аргумента. Первый аргумент — это индекс, начиная с которого выполняется считывание символов. Второй аргумент — это индекс первого символа, который не включается в массив. Третий аргумент метода — ссылка на массив, который заполняется символами. Четвертый аргумент определяет индекс элемента в массиве, начиная с которого символы из текста записываются в массив.

Небольшой пример, в котором используются данные методы, представлен в листинге 8.3.

Листинг 8.3. Базовые операции с текстом

```
import java.util.Arrays;
class Demo{
   public static void main(String[] args){
      // Текстовая строка:
      String str="Программируем на Java";
      // Символьный массив:
      char[] symbs=new char[12];
      // Заполнение массива:
      "Изучаем Pascal".getChars(0,8,symbs,0);
      str.getChars(str.length()-4,str.length(),symbs,8);
      // Проверка содержимого массива:
      System.out.println(symbs);
      // Отображение строки в обратном порядке:
      for(int k=str.length()-1;k>=0;k--){
         System.out.print("|"+str.charAt(k));
      }
      System.out.println("|");
      // Переменная массива:
      byte[] nums;
      // Массив с кодами символов:
      nums="Java".getBytes();
      System.out.println(Arrays.toString(nums));
      // Создание массива на основе текста:
      symbs="Java".toCharArray();
      // Отображение содержимого массива:
      System.out.println(Arrays.toString(symbs));
   }
}
```

Результат выполнения программы показан ниже:

Результат выполнения программы (из листинга 8.3)

```
Изучаем Java
|a|v|a|J| |a|н| |м|е|у|р|и|м|м|а|р|г|о|р|П|
[74, 97, 118, 97]
[J, a, v, a]
```

В программе создается текстовая строка `str` со значением `"Программируем на Java"`. Также создается символьный массив `symbs` из `12` элементов. Содержимое массива формируется двумя командами. Команда `"Изучаем Pascal".getChars(0,8,symbs,0)` из текста `"Изучаем Pascal"` считывает символы с индексами от `0` до `7` включительно и записывает в массив `symbs`, начиная с позиции с индексом `0`. Команда `str.getChars(str.length()-4,str.length(),symbs,8)` из текстовой строки `str` считывает символы, начиная с символа с индексом `str.length()-4` и до конца текстовой строки, и эти символы записываются в массив `symbs`, начиная с позиции с индексом `8`. Содержимое массива `symbs` отображается командой `System.out.println(symbs)`.

НА ЗАМЕТКУ

Напомним, что если методу `println()` передать аргументом символьный массив, то содержимое массива отображается в виде текста, составленного из элементов массива.

С помощью оператора цикла содержимое строки `str` отображается в обратном порядке. При этом для считывания символов из строки использован метод `charAt()`.

Командой `nums="Java".getBytes()` в переменную массива `nums` записывается ссылка на созданный массив, который содержит коды символов из текста `"Java"`. Командой `symbs="Java".toCharArray()` на основе текста `"Java"` формируется символьный массив, и ссылка на этот массив записывается в переменную `symbs`. Для проверки содержимого массивов использован метод `toString()` из класса `Arrays`. Методом формируется текстовая строка с содержимым массива (элементы заключены в квадратные скобки и разделены запятыми).

Сравнение текстовых строк

> Какое глубокое проникновение в суть вещей! Впрочем, принц всегда очень тонко анализировал самую сложную ситуацию.
>
> *из к/ф «Приключения принца Флоризеля»*

При работе с текстом нередко возникает необходимость сравнить разные текстовые строки на предмет совпадения. Используют в этом случае методы `equals()` и `equalsIgnoreCase()`. Разница между методами состоит в том, что первый метод сравнивает строки с учетом состояния регистра, а второй состояние регистра игнорирует. Метод `equals()` возвращает в качестве значения `true`, если строки состоят из одинаковых символов, размещенных на одинаковых позициях в тексте. При этом строчная и прописная буквы интерпретируются как разные символы. Если полного совпадения нет, то в качестве результата методом возвращается значение `false`. Метод `equalsIgnoreCase()` при сравнении текстов интерпретирует строчную и прописную буквы как один и тот же символ. Каждый из упомянутых методов вызывается из текстового объекта, аргументом методу передается другой текстовый объект. Это именно те объекты, текст из которых сравнивается методами.

ПОДРОБНОСТИ

Операторы «равно» == и «не равно» != для сравнения текстовых значений не используются. Точнее, эти операторы формально задействовать можно, однако результат будет несколько неожиданным.

Предположим, необходимо сравнить текстовые строки A и B (объектные переменные объявлены как относящиеся к типу String и им присвоены значения). Если для сравнения использовать команду A==B или A!=B, то сравниваться будут значения объектных переменных A и B, а не «текстовое содержимое» объектов, на которые эти переменные ссылаются. Значениями переменных A и B являются адреса соответствующих объектов. Поэтому результатом выражения A==B является значение true, если переменные A и B ссылаются на один и тот же объект. Если же переменные ссылаются на разные объекты, значение выражения равно false. При этом разные объекты могут содержать одинаковые текстовые значения.

В листинге 8.4 приведен пример программы, в которой выполняется сравнение текстовых объектов.

Листинг 8.4. Сравнение текстовых строк

```
class Demo{
   public static void main(String[] args){
      String A="Java";
      String B=new String("Java");
      String C="Java";
      String D="JAVA";
      System.out.println("A.equals(B): "+A.equals(B));
      System.out.println("A==B: "+(A==B));
      System.out.println("A.equals(C): "+A.equals(C));
      System.out.println("A==C: "+(A==C));
      System.out.println("A.equals(D): "+A.equals(D));
      System.out.println(
         "A.equalsIgnoreCase(D): "+A.equalsIgnoreCase(D)
      );
   }
}
```

Результат выполнения программы имеет следующий вид:

Результат выполнения программы (из листинга 8.4)

```
A.equals(B): true
A==B: false
A.equals(C): true
A==C: true
A.equals(D): false
A.equalsIgnoreCase(D): true
```

В программе объявляются четыре текстовые переменные A, B, C и D. Переменным A и C в качестве значения присваивается текстовый литерал "Java". Переменной D присваивается литерал "JAVA". А переменная B ссылается на объект, который создается вызовом конструктора класса String с передачей аргументом литерала "Java". Таким образом, переменные A, B и C ссылаются на объекты с текстом "Java". Переменная D ссылается на объект с текстом "JAVA".

Далее для сравнения значений переменных A, B, C и D используются методы equals(), equalsIgnoreCase() и оператор «равно» ==. Результаты сравнений, думается, понятны. Хочется обратить внимание лишь на два обстоятельства. Результатом выражения A==C является значение true, поскольку переменным A и C значением присваивался один и тот же литерал "Java", реализованный одним и тем же объектом (из-за работы системы оптимизации ресурсов). Поэтому переменные A и C ссылаются на один и тот же текстовый объект. А вот результатом выражения A==B является значение false. Причина в том, что объект, на который ссылается переменная B, создавался не присваиванием литерала, а передачей литерала аргументом конструктору класса String. В таком случае создается новый объект.

Поиск символов и подстрок в тексте

Здесь были люди, и я их найду!

из к/ф «Чародеи»

Еще одна распространенная задача связана с поиском подстроки или символа в тексте. Если мы просто хотим выяснить, содержится ли подстрока в каком-либо тексте, то из соответствующего объекта вызывается метод contains(), а искомая подстрока передается ему аргументом. Метод возвращает значение true, если подстрока есть в тексте. В противном случае возвращается значение false.

Если нас интересует позиция, на которой подстрока находится в тексте, то полезными будут методы indexOf() и lastIndexOf(). Первым аргументом каждому из методов передается символ (значение типа char) или подстрока (объект класса String). Может указываться и второй целочисленный аргумент. Он определяет индекс в строке, с которого начинается поиск (точка поиска). Методом indexOf() выполняется поиск от точки поиска до конца строки, а методом lastIndexOf() выполняется поиск от точки поиска до начала строки. Результатом метода indexOf() является индекс первого (от точки поиска) появления символа/подстроки в тексте. Результатом метода lastIndexOf() является индекс последнего (от точки поиска) появления символа/подстроки в тексте. Если совпадений не найдено (символа/подстроки в тексте нет), возвращается значение -1. В листинге 8.5 представлена программа, в которой используются данные методы.

Листинг 8.5. Поиск символов и подстрок в тексте

```
class Demo{
   public static void main(String[] args){
      String str="Всегда слова обдумывая чьи-то\n"+
         "Ты видеть должен, что за ними скрыто.\n"+
         "И помни, что уменье что-то скрыть\n"+
         "Порой ценней уменья говорить!";
      System.out.println(str);
```

```
      System.out.println("str.contains(\"a\"): "+
         str.contains("a"));
      System.out.println("str.contains(\"ы\"): "+
         str.contains("ы"));
      System.out.println("str.contains(\"должен\"): "+
         str.contains("должен"));
      System.out.println("str.contains(\"ну\"): "+
         str.contains("ну"));
      System.out.println("str.indexOf(\'a\'): "+
         str.indexOf('a'));
      System.out.println("str.lastIndexOf(\'a\'): "+
         str.lastIndexOf('a'));
      System.out.println("str.indexOf(\"то\"): "+
         str.indexOf("то"));
      System.out.println("str.lastIndexOf(\"то\"): "+
         str.lastIndexOf("то"));
      System.out.println("str.indexOf(\'a\',10): "+
         str.indexOf('a',10));
      System.out.println("str.lastIndexOf(\'a\',10): "+
         str.lastIndexOf('a',10));
      System.out.println("str.indexOf(\"то\",30): "+
         str.indexOf("то",30));
      System.out.println("str.lastIndexOf(\"то\",30): "+
         str.lastIndexOf("то",30));
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 8.5)

```
Всегда слова обдумывая чьи-то
Ты видеть должен, что за ними скрыто.
И помни, что уменье что-то скрыть
Порой ценней уменья говорить!
str.contains("a"): true
str.contains("ы"): false
str.contains("должен"): true
str.contains("ну"): false
str.indexOf('a'): 5
str.lastIndexOf('a'): 53
str.indexOf("то"): 27
str.lastIndexOf("тоv): 92
str.indexOf('a',10): 11
str.lastIndexOf('a',10): 5
str.indexOf("то",30): 49
str.lastIndexOf("то",30): 27
```

В программе объявляется переменная `str` класса `String` и в качестве значения этой переменной присваивается текстовый литерал. Соответствующее значение отображается в области вывода.

НА ЗАМЕТКУ

Поскольку литерал достаточно большой, он разбит на отдельные текстовые фрагменты. Итоговое текстовое значение получается объединением этих фрагментов.

Далее с помощью методов `contains()`, `indexOf()` и `lastIndexOf()` вычисляются некоторые выражения.

НА ЗАМЕТКУ

В исходном тексте использовалась инструкция `\n` для перехода на новую строку. При подсчете позиции символов наличие этой инструкции также принимается в расчет — она обрабатывается как символ окончания строки.

В случаях, когда нам в тексте нужно было использовать кавычки, мы их предваряли обратным слешем `\`.

Методы для работы с текстом

Пацак пацака не обманывает. Это некрасиво, родной.

из к/ф «Кин-дза-дза»

Несмотря на то что объекты класса `String` не могут изменяться, есть целая группа методов, которые позволяют на основе одного текстового объекта, путем обработки текста, создать другой текстовый объект. При вызове метода создается новый текстовый объект, и ссылка на него возвращается как результат метода. В табл. 8.1 представлены некоторые методы, полезные при работе с текстом.

Таблица 8.1. Методы для работы с текстом

Метод	Описание
`concat()`	Методом `concat()` выполняется объединение текстовых строк: тест, указанный аргументом метода, добавляется в конец текстовой строки, из которой вызывается метод. Получаемый в результате объединения текстовый объект класса `String` возвращается в качестве результата метода
`join()`	Статический метод, используемый для объединения текстовых строк. Результатом метод возвращает текстовую строку, которая получается объединением текстовых аргументов метода (кроме первого), и разделителем при этом является текст, указанный первым аргументом метода

Таблица 8.1 (*окончание*)

Метод	Описание
`replace()`	Метод заменяет символы или подстроки в тексте. Текст, полученный в результате замены, возвращается результатом метода. Первым аргументом передается символ (подстрока), который (которую) нужно заменить. Вторым аргументом передается символ (подстрока), на который (которую) выполняется замена
`split()`	Методом возвращается текстовый массив. Элементы массива получаются в результате разбивки на текстовые фрагменты исходного текста (из которого вызывается метод). Аргументом методу передается текст, определяющий разделитель (служит индикатором для разбивки текста на блоки). С помощью второго целочисленного аргумента можно ограничить количество блоков, на которые разбивается текст
`substring()`	Методом `substring()` в качестве результата возвращается текстовая подстрока (объект класса `String`) строки, из которой вызывается метод. Аргументами метода указывают индекс начала подстроки в строке и индекс первого не входящего в подстроку символа строки. Можно указывать только первый аргумент (в таком случае подстрока извлекается до конца текста)
`toLowerCase()`	Методом `toLowerCase()` в качестве результата возвращается текстовая строка (объект класса `String`), которая получается из исходной строки (из которой вызывается метод) переводом всех букв в нижний регистр (все буквы маленькие). Метод аргументов не имеет
`toUpperCase()`	Методом `toUpperCase()` в качестве результата возвращается текстовая строка (объект класса `String`), которая получается из исходной строки (из которой вызывается метод) переводом всех букв в верхний регистр (все буквы большие). Метод аргументов не имеет
`trim()`	Результатом метода `trim()` возвращается ссылка на текстовый объект, который получается из исходного текста (из которого вызывается метод) удалением начальных и конечных пробелов. Метод не имеет аргументов

Примеры использования перечисленных методов для обработки текста представлены в программе (листинг 8.6).

Листинг 8.6. Обработка текста встроенными методами

```
class Demo{
   public static void main(String[] args){
      // Текстовая строка:
      String str="  Программируем на C++  ";
      System.out.println("str: "+str);
```

```
      // Удаление начальных и конечных пробелов:
      str=str.trim();
      System.out.println("str: "+str);
      String A,B,C,D,E,F;
      // Извлечение подстроки:
      A=str.substring(3,8);
      System.out.println("A: "+A);
      // Объединение строк:
      B=str.concat(" и Java");
      System.out.println("B: "+B);
      // Замена символов:
      C=B.replace(‘ ‘,'_');
      System.out.println("C: "+C);
      // Перевод в нижний регистр:
      D=B.toLowerCase();
      System.out.println("D: "+D);
      // Перевод в верхний регистр:
      E=B.toUpperCase();
      System.out.println("E: "+E);
      // Объединение текстовых строк:
      F=String.join("+","один","два","три");
      System.out.println("F: "+F);
      // Разбивка текста на блоки:
      String[] txt=B.split(" ");
      for(int k=0;k<txt.length;k++){
         System.out.print("|"+txt[k]);
      }
      System.out.println("|");
   }
}
```

Ниже показано, как выглядит результат выполнения программы:

Результат выполнения программы (из листинга 8.6)

```
str: Программируем на C++
str: Программируем на C++
A: грамм
B: Программируем на C++ и Java
C: Программируем_на_C++_и_Java
D: программируем на c++ и java
E: ПРОГРАММИРУЕМ НА C++ И JAVA
F: один+два+три
|Программируем|на|C++|и|Java|
```

В программе объявляется текстовая переменная `str` со значением `" Программируем на C++ "`. В этом значении — двойной начальный и двойной конечный пробелы. Командой `str=str.trim()` эти пробелы в начале и конце строки удаляются. Мы используем еще несколько текстовых строк. Так, командой `A=str.substring(3,8)` из текстовой строки `str` извлекается подстрока, состоящая из символов с индексами от `3` до `7` включительно. Командой `B=str.concat(" и Java")` создается новая

текстовая строка, которая получается объединением текста из переменной `str` и литерала, переданного аргументом методу `concat()`. При выполнении команды `C=B.replace(' ','_')` новая строка создается так: в тексте из строки `B` пробелы заменяются символами подчеркивания. При вызове метода `toLowerCase()` из объекта `B` (команда `D=B.toLowerCase()`) получаем такую же строку, но состоящую из маленьких букв. Чтобы получить строку, состоящую из больших букв, используем инструкцию `E=B.toUpperCase()`.

Команда `F=String.join("+","один","два","три")` создает текстовую строку объединением аргументов `"один"`, `"два"` и `"три"` статического метода `join()`, причем в качестве разделителя используется первый аргумент `"+"`.

При выполнении команды `String[] txt=B.split(" ")` текст из строки `B` разбивается на блоки. Разделителем является пробел (аргумент метода `split()`). Из текстовых блоков формируется текстовый массив, и ссылка на него возвращается результатом метода `split()` (и записывается в переменную `txt`). Затем содержимое массива отображается в области вывода, для чего используется оператор цикла.

Форматированный текст

— А чем они друг от друга отличаются?

— Ты что, дальтоник, Скрипач? Зеленый цвет от оранжевого отличить не можешь?

из к/ф «Кин-дза-дза»

В классе `String` есть статический метод `format()`, с помощью которого можно создавать форматированные текстовые строки. Речь идет о текстовых строках, в которых используются определенные правила для отображения данных разного типа.

Первым аргументом методу `format()` передается *строка форматирования*, которая содержит специальные *инструкции форматирования*. Прочие аргументы метода `format()` — это значения, которые подставляются вместо инструкций форматирования при формировании текстовой строки. Фактически инструкции форматирования определяют, в каком виде (или формате) данные будут инкапсулированы в текстовую строку. Например, инструкцией `String.format("Число: %7.3f",1.23456)` создается текстовая строка `"Число: 1,235"`. Почему так? Строка формируется следующим образом. В текст `"Число: %7.3f"` вместо инструкции форматирования `%7.3f` вставляется значение `1.23456`. В инструкции `%7.3f` символ `%` является признаком начала инструкции форматирования. Символ `f` означает, что в соответствующее место вставляется действительное число. Цифра `7` означает, что под это число выделяется не менее семи позиций, а цифра `3` после точки означает, что в дробной части отображается три цифры.

ПОДРОБНОСТИ

Хотя действительные литералы вводятся с точкой в качестве десятичного разделителя, некоторые методы, в соответствии с локальными настройками, вместо десятичной точки отображают запятую. Это замечание относится и к методу `format()`. Если это является проблемой, то ситуацию можно исправить. В частности, самым первым аргументом методу при вызове достаточно указать ссылку на объект класса `Locale`, определяющий локальные настройки. Ссылку на такой объект можно получить с помощью одного из статических полей класса `Locale`. Например, инструкцией `String.format(Locale.ENGLISH, "Число: %7.3f",1.23456)` возвращается текстовая строка `"Число: 1.235"`, в которой используется десятичная точка.

Принципы создания строки форматирования (точнее, инструкций форматирования, которые входят в нее) довольно разнообразны. Мы остановимся лишь на ключевых моментах.

Во-первых, строка форматирования может содержать несколько инструкций форматирования, вместо которых подставляются разные аргументы метода `format()`. В таком случае после символа `%` указывается номер аргумента, который вставляется вместо соответствующей инструкции, и символ `$`. Цифра `1` означает, что вставляется первый аргумент после строки форматирования, цифра `2` соответствует второму аргументу, и так далее. Например, результатом инструкции `String.format("%1$3d +%2$3d/%1$1d =%3$3d",5,30,11)` является текстовая строка `" 5 + 30/5 = 11"`. В текстовую строку `"%1$3d +%2$3d/%1$1d =%3$3d"` вместо инструкции `%1$3d` вставляется первый (после строки форматирования) аргумент `5`. Под него выделяется не менее трех позиций (число `3` перед символом `d`). Символ `d` означает, что вставляется целое число. Вместо инструкции `%2$3d` вставляется второй аргумент `30`, и под него выделяется не менее трех позиций (число `3` перед символом `d`). Вместо инструкции `%1$1d` вставляется первый аргумент `5`, и под него выделяется не менее одной позиции (число `1` перед символом `d`). Наконец, вместо инструкции `%3$3d` вставляется третий аргумент `11` и под него выделяется не менее трех позиций (число `3` перед символом `d`).

В общем случае инструкция форматирования (для данных базовых типов) имеет такой вид:

```
%[номер$][параметры][ширина][.точность]тип
```

То, что в квадратных скобках, не является обязательным. Вообще же первым блоком, как отмечалось, указывается номер аргумента для вставки, и после номера указывается символ `$` (то есть аргументы нумеруются как `1$`, `2$` и так далее).

После номера аргумента может указываться параметр, определяющий режим отображения данных. Например, если указать символ «минус» `-`, то данные в области отображения будут выравниваться по левому краю (имеется в виду выравнивание в блоке, выделенном для отображения значения аргумента). Если указать «плюс» `+`, то для числовых значений всегда будет отображаться знак. Пробел в качестве параметра означает, что положительные числа отображаются с начальным пробелом.

Если указать `0`, то все свободные позиции в блоке, выделенном для представления числа, заполняются нулями (например, для отображения числа выделено `6` позиций, а число реально занимает `2` позиции, тогда оставшиеся `4` начальные позиции будут заполнены нулями).

ПОДРОБНОСТИ

Некоторые параметры могут указываться одновременно — например - и +.

После параметров указывается ширина блока (в символах), выделяемого для отображения значения аргумента. Через точку можно указать точность — количество цифр, отображаемых в дробной части числа. Последним указывается (обязательно) идентификатор, определяющий тип значения, которое вставляется в строку форматирования вместо соответствующей инструкции форматирования. Символы `X` или `x` означают, что число будет отображаться в шестнадцатеричной системе счисления. Для отображения числа в восьмеричной системе используют символ `o`. Текстовые значения идентифицируются символами `s` или `S`, а символьные — символами `c` или `C`. Символ `f` соответствует числу в формате с плавающей точкой, символ `d` обозначает целое десятичное число. Для отображения действительного числа в научной нотации используют символы `e` или `E`. Если использовать символы `g` или `G`, то число будет отображаться в формате с плавающей точкой или в научной нотации в зависимости от фактического значения числа. Есть и другие варианты, которые для нас интереса не представляют.

Небольшой пример, в котором используются строки форматирования и метод `format()`, представлен в листинге 8.7.

Листинг 8.7. Форматированный текст

```
class Demo{
   public static void main(String[] args){
      // Переменные:
      int num=54321;
      double val=12.34567;
      char symb='R';
      String txt="Java";
      // Форматированный текст:
      String A=String.format(
         "Целое число %1$+010d и символ %2$4c",num,symb);
      String B=String.format(
         "Текст \"%1$-7s\" и число %2$e",txt,val);
      String C=String.format(
         "Число %1$07x — это то же, что и %1$07o",num);
      String D=String.format(
         "Число: %1$ d\nЧисло: %2$ d",num,-num);
      String E=String.format(
         "Действительное число: %012.3f",val);
      String F=String.format(
```

```
            "Научная нотация: %12.3e",val);
        // Отображение результата:
        System.out.println(A);
        System.out.println(B);
        System.out.println(C);
        System.out.println(D);
        System.out.println(E);
        System.out.println(F);
    }
}
```

Результат выполнения программы показан ниже:

Результат выполнения программы (из листинга 8.7)

```
Целое число +000054321 и символ    R
Текст "Java   " и число 1,234567e+01
Число 000d431 — это то же, что и 0152061
Число:  54321
Число: -54321
Действительное число: 00000012,346
Научная нотация:    1,235e+01
```

Код простой, поэтому прокомментируем только инструкции форматирования, использованные при создании форматированного текста (табл. 8.2).

Таблица 8.2. Инструкции форматирования

Инструкция	Описание
`%1$+010d`	Значение первого аргумента отображается как целое число. Под это число выделяется не менее `10` позиций. Всегда отображается знак числа (`+` для положительных и `–` для отрицательных). Свободные позиции заполняются нулями
`%2$4c`	Значение второго аргумента отображается как символ, и под это значение выделяется `4` позиции
`%1$-7s`	Значение первого аргумента отображается как текст, текст выравнивается по левому краю и под значение отводится не менее `7` позиций
`%2$e`	Значение второго аргумента отображается как действительное число в научной нотации
`%1$07x`	Значение первого аргумента отображается в шестнадцатеричной системе счисления. Под значение отводится не менее `7` позиций. Все свободные позиции заполняются нулями
`%1$07o`	Значение первого аргумента отображается в восьмеричной системе счисления. Под значение отводится не менее `7` позиций. Все свободные позиции заполняются нулями

Таблица 8.2 (*окончание*)

Инструкция	Описание
`%1$ d`	Значение первого аргумента отображается как целое число. Для положительных чисел перед числом добавляется пробел (для отрицательных чисел перед числом отображается знак «минус»)
`%2$ d`	Значение второго аргумента отображается как целое число. Для положительных чисел перед числом добавляется пробел (для отрицательных чисел перед числом отображается знак «минус»)
`%012.3f`	Значение единственного (кроме текстовой строки форматирования) аргумента отображается в формате числа с плавающей точкой. Под число выделяется не менее `12` позиций. Все свободные позиции заполняются нулями. В дробной части числа отображается `3` цифры
`%12.3e`	Значение единственного (кроме текстовой строки форматирования) аргумента отображается как действительное число в научной нотации. Под значение выделяется не менее `12` позиций. В дробной части числа отображается `3` цифры

Вообще стоит отметить, что использование метода `format()` часто позволяет отображать информацию в наглядном, читабельном виде. Поэтому не стоит пренебрегать этим методом.

НА ЗАМЕТКУ

С помощью метода `printf()` можно осуществлять форматированное отображение данных. Разница между методами `format()` и `printf()` в том, что метод `printf()` отображает форматированное значение, а метод `format()` создает форматированную текстовую строку, которую затем можно отображать, например, методом `println()` или использовать иным образом.

Класс StringBuffer

А здесь из луца воду делают.

из к/ф «Кин-дза-дза»

Текстовые строки могут быть реализованы не только как объекты класса `String`, но и как объекты класса `StringBuffer`. Принципиальное отличие классов `String` и `StringBuffer`, в том, что объекты класса `StringBuffer` можно изменять. Другими словами, если текст реализован в виде объекта класса `StringBuffer`, в этот текст можно вносить изменения, причем без создания нового объекта. Например, при работе с объектами класса `StringBuffer` можно изменить какую-то букву или до-

бавить подстроку в конец или даже середину строки. Реализуется все это благодаря тому, что при создании объектов класса `StringBuffer` выделяется дополнительный объем памяти, за счет которого впоследствии можно выполнять упомянутые и многие другие операции.

НА ЗАМЕТКУ

Вообще, следует понимать, что класса `String` для работы с текстом обычно более чем достаточно. Возможность выполнять с объектами класса `StringBuffer` дополнительные операции скорее влияет на скорость выполнения операций, но не на конечный результат.

У класса `StringBuffer` есть несколько конструкторов: конструктор без аргументов, конструктор с целочисленным аргументом и конструктор с текстовым аргументом (например, типа `String` или `StringBuffer`).

Если используется конструктор без аргумента, то создается объект класса `StringBuffer` со значением в виде пустой текстовой строки, а также автоматически резервируется память еще для 16 символов (буфер памяти). Чтобы в явном виде указать размер буфера памяти при создании объекта класса `StringBuffer`, используют конструктор с целочисленным аргументом. Для создания копии уже существующего текстового объекта применяют конструктор с текстовым аргументом. В последнем случае созданный объект содержит соответствующий текст и дополнительный объем памяти для еще 16 символов.

Кроме конструкторов, в классе `StringBuffer` имеются и иные методы (табл. 8.3).

Таблица 8.3. Методы для работы с классом StringBuffer

Метод	Описание
`append()`	Методом в конец строки (из объекта которой вызывается метод) добавляется текст, переданный аргументом методу. Аргументом методу можно передавать символ, символьный массив, текст, число, объект (в этом случае в строку дописывается текстовое представление для объекта). Если аргументом методу передается символьный массив, то также можно указать индексы символов, которые добавляются в строку, а именно: указывается индекс, начиная с которого символы добавляются к строке, и индекс первого символа, который не входит в добавляемую последовательность. Результатом метод возвращает ссылку на объект, из которого вызывается метод
`capacity()`	Методом возвращается значение для размера выделенной памяти (в символах) для текстового объекта, из которого вызывается метод
`charAt()`	Методом возвращается символ в строке с индексом, указанным в качестве аргумента метода

Таблица 8.3 (*продолжение*)

Метод	Описание
`delete()`	Методом удаляется подстрока из строки, из которой вызывается метод. Первым аргументом методу передается индекс первого удаляемого символа, а вторым аргументом указывается индекс первого символа после удаляемой подстроки. Результатом метод возвращает ссылку на объект, из которого он вызывается
`deleteCharAt()`	Методом из строки, из которой вызывается метод, удаляется символ с индексом, указанным аргументом метода. Результатом метод возвращает ссылку на объект, из которого он вызывается
`ensureCapacity()`	Метод используется для выделения памяти для уже созданного объекта. Дополнительная память для объекта выделяется, если текущее значение для объема памяти меньше значения, указанного аргументом метода. Метод не возвращает результат
`getChars()`	Методом выполняется копирование символов из текстовой строки в символьный массив. Аргументы метода: индекс первого из копируемых символов, индекс первого некопируемого символа, ссылка на символьный массив и индекс элемента в массиве, начиная с которого символы копируются в массив. Метод не возвращает результат
`indexOf()`	Методом в качестве результата возвращается индекс первого вхождения подстроки, указанной аргументом метода, в строку, из которой вызывается метод. Если подстрока в строке не встречается, результатом возвращается значение `-1`. Вторым аргументом можно указать индекс символа, с которого начинается поиск подстроки
`insert()`	Методом выполняется вставка в строку, из которой вызывается метод, текста, определяемого аргументом метода. Первым аргументом метода указывается индекс, определяющий место вставки текста. Вторым аргументом может быть символ, символьный массив, число, текст, объект (в этом случае используется текстовое представление для объекта). Если вторым аргументом передан текст, то можно также указать индексы символов, формирующих подстроку для вставки (указывается индекс первого символа в подстроке и индекс первого символа, который в подстроку не входит). Результатом метод возвращает ссылку на объект, из которого вызывается
`lastIndexOf()`	Методом в качестве результата возвращается индекс последнего вхождения подстроки, указанной аргументом метода, в строку, из которой вызывается метод (поиск выполняется с конца строки в начало). Если подстрока в строке не встречается, результатом возвращается значение `-1`. Вторым аргументом можно указать индекс символа, с которого начинается поиск подстроки

Метод	Описание
`length()`	Метод возвращает текущую длину текстовой строки, записанной в объект, из которого вызывается метод
`replace()`	Метод используется для замещения фрагмента текста подстрокой. Замещается фрагмент текста в объекте, из которого вызывается метод. Замещаемый фрагмент определяется двумя индексами: индексом первого символа замещаемого фрагмента и индексом первого символа, который не входит в замещаемый фрагмент. Это первые два аргумента метода. Третий аргумент метода — подстрока, вставляемая вместо замещаемого фрагмента. Результатом метод возвращает ссылку на объект, из которого он вызывается
`reverse()`	Метод меняет порядок следования символов в тексте, из объекта которого вызывается метод. Аргументов у метода нет. Результатом метод возвращает ссылку на объект, из которого он вызывается
`setCharAt()`	Метод предназначен для изменения значения символа в тексте, из объекта которого он вызывается. Индекс изменяемого символа и новое значение для символа передаются аргументами методу. Метод не возвращает результат
`setLength()`	Методом устанавливается длина текстовой строки (аргумент метода)
`substring()`	Методом в качестве результата возвращается объект класса `String`. Объект содержит подстроку из текста (объект, из которого вызывается метод). Первым аргументом методу передается индекс символа, начиная с которого считывается подстрока. Вторым аргументом методу передается индекс первого символа, который не включается в подстроку. Если второй аргумент не указать, то подстрока считывается до конца текста
`toString()`	Методом возвращается объект класса `String` с текстом из объекта, из которого вызывается метод
`trimToSize()`	При вызове метода выполняется попытка оптимизировать объем памяти, выделенной под текстовый объект, так, чтобы он соответствовал размеру записанного в объект текста. Метод не возвращает результат

Примеры использования некоторых методов класса `StringBuffer` представлены в листинге 8.8.

Листинг 8.8. Использование класса StringBuffer

```
class Demo{
   // Статический метод для отображения информации
   // об объекте:
   static void show(StringBuffer txt){
```

```
      System.out.println("Длина текста: "+txt.length());
      System.out.println("Объем памяти: "+txt.capacity());
   }
   // Главный метод:
   public static void main(String[] args){
      // Объекты класса StringBuffer:
      StringBuffer A=new StringBuffer("Изучаем Java");
      StringBuffer B=new StringBuffer();
      StringBuffer C=new StringBuffer(30);
      // Параметры объектов:
      System.out.println("Объект A:");
      show(A);
      System.out.println("Объект B:");
      show(B);
      System.out.println("Объект C:");
      show(C);
      // Вставка подстроки:
      A.insert(8,"C++ и ");
      System.out.println(A);
      // Замена подстроки:
      A.replace(8,11,"Python");
      System.out.println(A);
      System.out.println("Объект A:");
      show(A);
      // Изменение объема памяти:
      A.trimToSize();
      show(A);
      // Изменение длины текста:
      A.setLength(14);
      System.out.println(A);
      show(A);
      // Увеличение объема памяти:
      A.ensureCapacity(50);
      show(A);
      // Инверсия текста:
      A.reverse();
      System.out.println(A);
      System.out.println("Объект B:");
      // Добавление подстроки:
      B.append("Python и Basic");
      System.out.println(B);
      // Удаление подстроки:
      B.delete(1,10);
      System.out.println(B);
      // Удаление символа:
      B.deleteCharAt(3);
      System.out.println(B);
      // Добавление символов:
      B.append(‘a');
      B.append('l');
      System.out.println(B);
   }
}
```

Результат выполнения программы показан ниже:

Результат выполнения программы (из листинга 8.8)

```
Объект A:
Длина текста: 12
Объем памяти: 28
Объект B:
Длина текста: 0
Объем памяти: 16
Объект C:
Длина текста: 0
Объем памяти: 30
Изучаем C++ и Java
Изучаем Python и Java
Объект A:
Длина текста: 21
Объем памяти: 28
Длина текста: 21
Объем памяти: 21
Изучаем Python
Длина текста: 14
Объем памяти: 21
Длина текста: 14
Объем памяти: 50
nohtyP меачузИ
Объект B:
Python и Basic
Pasic
Pasc
Pascal
```

В программе мы описываем вспомогательный статический метод `show()`, предназначенный для отображения такой информации об объекте класса `StringBuffer` (аргумент метода), как длина текста (вычисляется методом `length()`) и объем памяти, выделенный для соответствующего объекта (вычисляется методом `capacity()`). Этот метод используется в главном методе программы.

В первую очередь в программе создаются три объекта класса `StringBuffer`. Здесь мы иллюстрируем способы создания объектов. Объект `A` создается с передачей текстового аргумента `"Изучаем Java"` конструктору класса `StringBuffer`. Длина текста составляет `12` символов. Память под объект выделяется еще под `16` дополнительных символов. Поэтому объем выделяемой под объект памяти рассчитан на запись `28` символов.

При создании объекта `B` используется конструктор без аргументов. В этом случае длина текста равна `0` (объект не содержит текст), и под объект выделена память, достаточная для записи `16` символов.

Объект `C` создается путем вызова конструктора класса `StringBuffer` с аргументом `30`. Поэтому созданный объект не содержит текста, а для объекта выделяется объем памяти, достаточный для записи `30` символов.

После создания объектов выполняются некоторые операции с использованием методов класса `StringBuffer`. Сначала командой `A.insert(8, "C++ и ")` в строку `A` на позицию с индексом `8` добавляется текст `"C++ и "`. Командой `A.replace(8,11, "Python")` выполняется замена подстроки: символы с индексами от `8` до `10` включительно удаляются, а вместо этой подстроки вставляется текст `"Pythonм"`. На этом этапе объем памяти рассчитан на `28` символов, а длина текста составляет `21` символ. Вызвав метод `trimToSize()` из объекта `A`, мы уменьшаем объем памяти, выделенной под объект, до размеров, соответствующих длине записанного в объект текста. Объем памяти после выполнения команды равен `21`. А вот командой `A.setLength(14)` задается новая длина текста, и она меньше текущей длины текста. Лишние символы отбрасываются. Текст теперь имеет длину в `14` символов (значение `"Изучаем Python"`), а объем памяти рассчитан на запись `21` символа.

Инструкцией `A.ensureCapacity(50)` объем памяти, выделенной под объект `A`, увеличивается. Теперь в объект можно записать `50` символов. После выполнения команды `A.reverse()` текст в объекте `A` инвертируется.

Объект `B` после создания не содержит текста, но в нем достаточно места для записи текста из `16` символов. Командой `B.append("Python и Basic")` мы формально в конец текущего текстового значения добавляем литерал `"Python и Basic"`. Но поскольку ранее там текста не было, то фактически литерал определяет новое значение объекта `B`.

Командой `B.delete(1,10)` из объекта `B` удаляется подстрока с символами, индексы которых лежат в диапазоне от `1` до `9` включительно. В итоге в объекте `B` будет содержаться текст `"Pasic"`. После этого командой `B.deleteCharAt(3)` удаляется символ с индексом `3`, и в итоге получаем текст `"Pasc"`. Затем командами `B.append('a')` и `B.append('l')` в конец текста последовательно добавляются символы `'a'` и `'l'`, что дает слово `"Pascal"`.

Класс StringBuilder

> Прекратите панику, Фукс. И слушайте меня внимательно!
>
> *из м/ф «Приключения капитана Врунгеля»*

По своим функциональным возможностям объекты класса `StringBuilder` аналогичны объектам класса `StringBuffer`. У класса `StringBuilder` фактически те же методы, что и у класса `StringBuffer`. Конструкторов у класса `StringBuilder`, как и у класса `StringBuffer`, тоже три: без аргументов, с целочисленным аргументом и с текстовым аргументом.

НА ЗАМЕТКУ

Наличие конструктора с текстовым аргументом означает, что аргументом такому конструктору можно передавать ссылку на объект класса `String`. А еще аргументом конструктору можно передавать ссылку на объект класса, реализующего интерфейс `CharSequence` (это относится к классам `StringBuffer` и `StringBuilder`). Интерфейс `CharSequence` реализуется, кроме прочего, в классах `String`, `StringBuffer` и `StringBuilder`.

Главное отличие между классами `StringBuffer` и `StringBuilder` состоит в том, что при многопоточном программировании использование класса `StringBuilder` может привести к проблемам. Поэтому его обычно используют в однопоточной модели. Если же в программе используется многопоточная модель, то рекомендуется использовать класс `StringBuffer`. В прикладном плане работа с классом `StringBuilder` аналогична работе с классом `StringBuffer`. Небольшой пример, в котором задействован класс `StringBuilder`, представлен в листинге 8.9.

Листинг 8.9. Использование класса StringBuilder

```
class Demo{
   public static void main(String[] args){
      // Текстовая строка:
      StringBuilder A=new StringBuilder(30);
      System.out.println("Длина текста: "+A.length());
      System.out.println("Объем памяти: "+A.capacity());
      // В объект добавляется текст:
      A.append("Изучаем Java");
      System.out.println(A);
      // Вставка подстроки в текст:
      A.insert(8,"язык ");
      System.out.println(A);
      // Замена подстрок:
      A.replace(0,3,"У").replace(2,4,"и");
      System.out.println(A);
      // Создание объекта класса String:
      String B=A.toString();
      System.out.println(B);
   }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 8.9)

```
Длина текста: 0
Объем памяти: 30
Изучаем Java
Изучаем язык Java
Учим язык Java
Учим язык Java
```

Объект A класса StringBuilder создается вызовом конструктора с аргументом 30. Поэтому объект не содержит текста. Памяти, выделенной под объект, достаточно для записи текста из 30 символов. Командой A.append("Изучаем Java") в объект добавляется текст "Изучаем Java". Затем с помощью инструкции A.insert(8, "язык ") в текущее текстовое значение на позицию с индексом 8 вставляется подстрока "язык ", в результате чего получаем текстовое значение "Изучаем язык Java". Команда A.replace(0,3,"У").replace(2,4,"и") выполняет двойную замену подстрок. Здесь мы воспользовались тем обстоятельством, что метод replace() (как и некоторые другие методы) возвращает ссылку на объект, из которого он вызывается. Поэтому при выполнении инструкции A.replace(0,3,"У") в тексте из объекта A последовательность символов с индексами от 0 до 2 включительно заменяется на текст "У" (получается текстовое значение "Учаем язык Java"). А значением выражения A.replace(0,3,"У") является ссылка на объект A. Поэтому через данную ссылку можем снова вызвать метод replace(), что мы и делаем (команда A.replace(0,3, "У").replace(2,4, "и")). В итоге в тексте "Учаем язык Java" символы с индексами 2 и 3 заменяются на текст "и", после чего в объекте A содержится текст "Учим язык Java". Наконец, мы объявляем переменную B типа String и в качестве значения присваиваем ей выражение A.toString(). Хотя во многих случаях метод toString() вызывается автоматически, в данном случае мы его вызываем сами, в явном виде. В результате на основе объекта A создается объект класса String и ссылка на этот объект записывается в переменную B.

Обработка текста

Ну, полетели, полетели, родной.

из к/ф «Кин-дза-дза»

Рассмотренных ранее методов обычно вполне хватает для выполнения всевозможных операций с текстом. Но кроме этих встроенных методов мы можем описать собственные. Мы могли бы описать собственные методы для выполнения многих из описанных ранее операций с текстом — но в этом нет необходимости. Однако иногда бывает удобно создавать специальные методы для обработки текста (листинг 8.10). В программе описываются статические методы для выполнения таких операций с текстом, как определение количества слов в тексте, вычисление количества появлений символа в тексте, вычисление позиций, на которых искомый символ появляется в тексте, а также определение набора букв (разных), которые есть в тексте. Созданные методы используются для обработки текста.

Листинг 8.10. Обработка текста

```
import java.util.Arrays;
class Demo{
   // Метод для подсчета количества появлений
   // символа в тексте:
```

```
static int count(CharSequence txt,char smb){
   int res=0;
   for(int k=0;k<txt.length();k++){
      if(txt.charAt(k)==smb) res++;
   }
   return res;
}
// Метод для подсчета количества слов:
static int words(CharSequence txt){
   return count(txt,' ')+1;
}
// Метод для поиска символа в тексте:
static int[] find(String txt,char smb){
   int[] res={-1};
   int[] t;
   int k=0;
   // Поиск первого появления символа в тексте:
   while(k<txt.length()){
      if(txt.charAt(k)==smb){
         res[0]=k;
         break;
      }
      k++;
   }
   // Поиск символа в тексте:
   while(k<txt.length()-1){
      k++;
      if(txt.charAt(k)==smb){
         t=new int[res.length+1];
         for(int m=0;m<res.length;m++){
            t[m]=res[m];
         }
         t[res.length]=k;
         res=t;
      }
   }
   // Результатом является ссылка на массив:
   return res;
}
// Метод для поиска разных букв в тексте:
static char[] getSymbs(String txt){
   // Перевод строки в нижний регистр:
   String str=txt.toLowerCase();
   String res="";
   char s;
   // Перебор символов в тексте:
   for(int k=0;k<str.length();k++){
      s=str.charAt(k);
      // Символы, которые игнорируются:
      switch(s){
         case ' ':
         case '.':
         case ',':
```

```
            case '?':
            case '!':
            case ':':
            case ';':
            case '-':
               continue;
         }
         // Если буква еще не встречалась в тексте:
         if(!res.contains(""+s)){
            res+=s;
         }
      }
      // Результатом является массив из букв:
      return res.toCharArray();
   }
   // Главный метод:
   public static void main(String[] args){
      String txt="Я помню чудное мгновенье";
      System.out.println(txt);
      char smb='н';
      System.out.printf(
         "Поиск символа \'%1$c\' в тексте: %2$s\n",smb,
         Arrays.toString(find(txt,smb))
      );
      smb='ю';
      System.out.printf(
         "Поиск символа \'%1$c\' в тексте: %2$s\n",smb,
         Arrays.toString(find(txt,smb))
      );
      smb='а';
      System.out.printf(
         "Поиск символа \'%1$c\' в тексте: %2$s\n",smb,
         Arrays.toString(find(txt,smb))
      );
      // Текстовые объекты:
      String A=new String(txt);
      StringBuffer B=new StringBuffer(txt);
      StringBuilder C=new StringBuilder(txt);
      smb='о';
      System.out.printf(
         "Букв \'%1$c\' в тексте: %2$s\n",smb,
         count(A,smb));
      smb='м';
      System.out.printf(
         "Букв \'%1$c\' в тексте: %2$s\n",smb,
         count(B,smb));
      smb='ы';
      System.out.printf(
         "Букв \'%1$c\' в тексте: %2$s\n",smb,
         count(C,smb));
      System.out.println("Слов в тексте: "+words(txt));
      System.out.println("Буквы в тексте: "+
         Arrays.toString(getSymbs(txt))
```

```
      );
      System.out.println("Буквы в пустом тексте: "+
         Arrays.toString(getSymbs(""))
      );
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 8.10)

```
Я помню чудное мгновенье
Поиск символа 'н' в тексте: [5, 11, 17, 21]
Поиск символа 'ю' в тексте: [6]
Поиск символа 'а' в тексте: [-1]
Букв 'о' в тексте: 3
Букв 'м' в тексте: 2
Букв 'ы' в тексте: 0
Слов в тексте: 4
Буквы в тексте: [я, п, о, м, н, ю, ч, у, д, е, г, в, ь]
Буквы в пустом тексте: []
```

Метод для подсчета количества появлений символа в тексте `count()` имеет два аргумента. Первый аргумент является интерфейсной переменной типа `CharSequence`. Поскольку этот интерфейс реализуется в классах `String`, `StringBuffer`, `StringBuilder`, то первым аргументом можно передавать объекты данных классов. Первый аргумент определяет текст, в котором выполняется поиск символа. Вторым аргументом методу передается символьное значение. В теле метода выполняется оператор цикла, в котором подсчитывается количество вхождений символа в текст.

Метод `words()` предназначен для вычисления количества слов в тексте. Текстовый объект передается аргументом методу. Под словами мы подразумеваем блоки текста, разделенные пробелами. Поэтому в нашей (достаточно наивной) интерпретации количество слов в тексте на единицу больше, чем количество пробелов. А количество пробелов подсчитываем с помощью созданного нами метода `count()`, передав ему вторым аргументом символ пробела.

Метод `find()`возвращает ссылку на целочисленный массив, элементы которого соответствуют индексам позиций в тексте (первый аргумент метода), на которых встречается заданный символ (второй аргумент метода). Если символа в тексте нет, то результатом возвращается ссылка на массив из одного элемента со значением `-1`.

ПОДРОБНОСТИ

В теле метода создается массив `res` с одним элементом `-1`. Затем запускается оператор цикла, в котором поиск выполняется до первого появления символа в тексте. Как только символ найден, единственный элемент в массиве `res` меняет значение (теперь это индекс первого появления символа в тексте), и инструкцией `break` завершается выполнение оператора цикла. Затем снова запускается оператор цикла, в котором перебираются оставшиеся символы в тексте. Если символ

найден, то создается новый массив (его размер на единицу больше размера массива `res`), он заполняется (значения начальных элементов такие же, как у массива `res`, а в последний лишний элемент записывается значение индекса позиции, на которой найден символ), и ссылка на новый массив записывается в переменную `res`.

Метод `getSymbs()` возвращает символьный массив с буквами (разными), которые встречаются в тексте (аргумент метода).

ПОДРОБНОСТИ

Мы исходим из того, что большая и маленькая буква — это один символ. Поэтому в исходном тексте все символы переводятся в нижний регистр с помощью метода `toLowerCase()`. Начальным значением текстовой переменной `res` является пустой текст. Затем перебираются все символы в тексте. Если прочитанный символ совпадает с одним из знаков пунктуации или пробелом, то инструкцией `continue` в операторе выбора соответствующая итерация завершается досрочно. В противном случае с помощью метода `contains()` мы проверяем, содержится ли прочитанная буква в строке `res`. Если буквы в строке нет, то прочитанный символ дописывается в конец строки. После завершения оператора цикла текст из переменной `res` преобразуется в символьный массив с помощью метода `toCharArray()`. Ссылка на этот массив возвращается методом в качестве результата.

Стоит заметить, что поскольку методу `contains()` должен передаваться текст, то мы к символьной переменной прибавляем пустой текстовый литерал, чтобы выполнилось автоматические преобразование к текстовому значению.

Если массив создается на основе пустого текста, то результатом является массив нулевого размера: массив как бы есть, но элементов в нем нет.

В главном методе программы проверяется работа созданных методов.

НА ЗАМЕТКУ

Стоит напомнить, что классы `String`, `StringBuffer` и `StringBuilder` описаны с ключевым словом `final`, поэтому суперклассами они быть не могут. Мы не можем создать собственный класс наследованием одного из перечисленных классов.

Резюме

Братцы, кончайте философствовать.

из к/ф «Кин-дза-дза»

- Для работы с текстом в Java предусмотрены классы `String`, `StringBuffer` и `StringBuilder`.

- Принципиальная разница между текстом, реализованным в виде объектов класса `String`, с одной стороны, и `StringBuffer` или `StringBuilder`, с другой стороны, состоит в том, что в первом случае созданный текст не может быть изменен, во втором — может. Под изменением текста в данном случае подразумевается изменение содержимого объекта, через который реализован текст. При реализации текста с помощью объекта класса `String` для изменения текста создается новый объект, и ссылка на него присваивается соответствующей объектной переменной. При реализации текста с помощью объектов класса `StringBuffer` или `StringBuilder` автоматически выделяется дополнительная память для записи символов текста. В объекты классов `StringBuffer` и `StringBuilder` можно вносить изменения.
- В классах `String`, `StringBuffer` и `StringBuilder` имеются специальные методы, которые позволяют выполнять все базовые операции с текстом.
- При преобразовании объектов в тип `String` (например, если объект передается аргументом методу `println()`) автоматически вызывается метод `toString()`. Путем переопределения этого метода для класса можно создать простой и эффективный механизм отображения информации об объектах класса.

9 Обработка исключений

Это безобразие так оставлять нельзя.

из к/ф «Карнавальная ночь»

Программы пишут для того, чтобы они работали. Работали быстро и, самое главное, правильно. К сожалению, так происходит не всегда. И проблема не всегда сводится к уровню подготовки программиста: бывают ситуации, когда невозможно гарантировать безошибочную работу программы. Желательно хотя бы свести к минимуму негативные последствия от возможных ошибок.

Самая большая неприятность, связанная с ошибками, возникающими в процессе выполнении кода, состоит в том, что программа экстренно завершает работу. Но во многих языках программирования (в том числе и в Java) предусмотрены защитные механизмы. О них и пойдет речь далее.

Исключительные ситуации

Ма тант, не будем устраивать эль скандаль при посторонних.

из к/ф «Формула любви»

Ошибки, которые возникают в результате выполнения программы, принято называть *исключительными ситуациями*. Что же происходит, если в процессе выполнения программы случается ошибка? Дальнейшее выполнение программы приостанавливается, и автоматически создается объект, описывающий возникшую исключительную ситуацию. Такой объект называют объектом исключения, или *исключением*. Созданный объект исключения передается для обработки методу, в котором возникла ошибка. Метод может обработать исключение или передать его для обработки дальше, то есть в тот метод, в котором вызывался данный метод (при выполнении которого произошла ошибка). Этот новый метод может обработать исключение или передать его дальше по цепочке вызовов

следующему методу. В итоге исключение или будет обработано, или не будет обработано. В последнем случае в игру вступает так называемый *обработчик по умолчанию*. Его действие сводится к прекращению работы программы, что, конечно, нежелательно.

НА ЗАМЕТКУ

Исключительные ситуации можно генерировать искусственно (или вручную) — то есть специальными программными методами. На первый взгляд, такая возможность кажется лишней и ненужной, но это не так. Механизм обработки исключительных ситуаций, в том числе искусственное генерирование исключений, нередко позволяет делать код более компактным и даже элегантным, значительно упрощая решение сложных задач.

Что же нужно сделать, чтобы в программном коде реализовать обработку ошибок? Во-первых, следует выделить фрагмент кода, который должен контролироваться на предмет генерирования исключительной ситуации (это так называемый *контролируемый код*). Во-вторых, необходимо создать код, непосредственно обрабатывающий исключительную ситуацию, то есть код, который выполняется в случае возникновения ошибки.

В Java для обработки исключительных ситуаций используется конструкция `try-catch`. В блок, помеченный ключевым словом `try`, помещается контролируемый код. После `try`-блока размещается блок, помеченный ключевым словом `catch` (причем таких блоков может быть несколько). Если при выполнении контролируемого кода в `try`-блоке ошибка не возникает, то все `catch`-блоки игнорируются. Если же ошибка возникла, то выполнение команд из `try`-блока прекращается и начинают выполняться команды в том `catch`-блоке, который предназначен для обработки ошибок данного типа.

ПОДРОБНОСТИ

Объект исключения содержит информацию о возникшей ошибке. Объекты создаются на основе классов. Фактически класс, на основе которого создается объект ошибки, определяется типом возникшей ошибки. Когда описывается `catch`-блок, то в круглых скобках после ключевого слова `catch` указывается класс ошибок, которые обрабатываются этим блоком, и формальное название для объекта ошибки — описание — должно быть таким же, как и описание аргументов метода. Если ошибка возникла, то в соответствии с типом ошибки на основе определенного класса создается объект. Затем просматриваются все `catch`-блоки, и если находится блок, соответствующий данному типу ошибки, то этот блок и выполняется.

После блоков `try` и `catch` можно указать блок `finally` с кодом, который выполняется в любом случае вне зависимости от того, возникла исключительная ситуация или нет. Общая схема описания конструкции `try-catch-finally` выглядит так:

```
try{
   // Контролируемый код
}
catch(класс объект){
   // Код для обработки ошибки
}
catch(класс объект){
   // Код для обработки ошибки
}
// Другие catch-блоки
finally{
   // Код выполняется всегда
}
```

Вся конструкция, если она содержит несколько `catch`-блоков и блок `finally`, работает так: если при исполнении кода в блоке `try` ошибки нет, то после выполнения этого блока выполняется блок `finally`, затем управление передается следующей после конструкции `try-catch-finally` команде. При возникновении ошибки в процессе выполнения кода в блоке `try` выполнение кода в этом блоке останавливается и начинается поиск подходящего блока `catch`. Если подходящий блок найден, то выполняется его код, после чего выполняется код блока `finally`. Далее выполняется код, следующий после конструкции `try-catch-finally`.

ПОДРОБНОСТИ

Блок `finally` проявляет себя в случае, если используются вложенные конструкции `try-catch`. Если, например, во внутреннем `try`-блоке возникла ошибка и для ее обработки подходящего внутреннего `catch`-блока нет, то исключение для обработки передается во внешние `catch`-блоки. Но перед этим выполняется блок `finally` (если он есть).

Мы рассмотрим процесс обработки исключений на примерах. Но сначала познакомимся с классами, которые соответствуют различным исключительным ситуациям.

Классы исключений

> Когда у общества нет цветовой дифференциации штанов, то нет цели. А когда нет цели — нет будущего.
>
> *из к/ф «Кин-дза-дза»*

В Java существует целая иерархия классов, предназначенных для обработки исключительных ситуаций. В вершине этой иерархии находится суперкласс

`Throwable`. У этого суперкласса есть два подкласса: `Exception` и `Error`. К классу `Error` относятся катастрофические ошибки, которые невозможно обработать в программе, например переполнение стека памяти. У класса `Exception` есть подкласс `RuntimeException`. К классу `RuntimeException` относятся ошибки времени выполнения программы, которые могут перехватываться и обрабатываться в программе.

НА ЗАМЕТКУ

Каждому типу ошибки соответствует определенный класс. Точнее, класс и является типом ошибки. При этом у ошибок некоторых типов есть подтипы. На уровне классов это соответствует ситуации, когда у класса есть подклассы. В итоге получается целая иерархия классов, связанных наследованием.

Исключения делятся на *контролируемые* и *неконтролируемые*. Неконтролируемые исключения соответствуют подклассам классов `RuntimeException` и `Error`. Прочие исключения относятся к контролируемым.

Деление (основанное на иерархии наследования) исключений на контролируемые и неконтролируемые очень важно. Если некоторый метод может сгенерировать контролируемое исключение и оно в методе не обрабатывается, то в сигнатуре метода должен быть отображен соответствующий факт: после круглых скобок с аргументами метода (но перед открывающей фигурной скобкой) указывается ключевое слово `throws` и название класса исключения.

НА ЗАМЕТКУ

Если метод генерирует и не обрабатывает неконтролируемое исключение, отражать этот факт в сигнатуре метода не нужно.

Иерархия наследования классов важна в силу еще одного обстоятельства. Дело в том, что при перехвате и обработке исключений в `catch`-блоках перехватывается не только исключения класса, указанного в `catch`-блоке, но и исключения всех подклассов данного класса.

НА ЗАМЕТКУ

Если создать `catch`-блок для обработки исключений класса `Exception`, то такой блок будет перехватывать практически все исключения.

Некоторые наиболее часто используемые неконтролируемые исключения перечислены в табл. 9.1.

Таблица 9.1. Классы неконтролируемых исключений

Класс исключения	Описание ошибки
`ArithmeticException`	Арифметическая ошибка (например, деление на ноль)
`ArrayIndexOutOfBoundsException`	Индекс массива выходит за пределы допустимых границ
`ArrayStoreException`	Присваивание элементу массива недопустимого значения
`ClassCastException`	Недопустимое приведение типов
`IllegalArgumentException`	Методу передан недопустимый аргумент
`IndexOutOfBoundsException`	Индекс находится за пределами допустимых для него границ
`InputMismatchException`	Ошибка, связанная с некорректно введенным значением
`NegativeArraySizeException`	Создание массива отрицательного размера
`NullPointerException`	Недопустимое использование ссылки
`NumberFormatException`	Недопустимое преобразование текстовой строки в числовой формат
`StringIndexOutOfBoundsException`	Попытка индексирования вне пределов строки
`UnsupportedOperationException`	Недопустимая операция

Некоторые контролируемые исключения представлены в табл. 9.2.

Таблица 9.2. Классы контролируемых исключений

Класс исключения	Описание ошибки
`ClassNotFoundException`	Класс не найден
`IllegalAccessException`	В доступе к классу отказано
`InstantiationException`	Попытка создать объект абстрактного класса или интерфейса
`InterruptedException`	Один поток прерван другим потоком
`NoSuchFieldException`	Поле не существует
`NoSuchMethodException`	Метод не существует

Кроме библиотечных классов, для работы с исключениями можно создавать собственные классы. Обычно их создают наследованием класса `Exception` или `RuntimeException`. В первом случае получаем класс для исключения контролируемого типа, а во втором — неконтролируемого.

Далее мы рассмотрим разные способы использования механизма обработки исключительных ситуаций.

Пример обработки исключений

Меня терзают смутные сомнения.

из к/ф «Иван Васильевич меняет профессию»

Начнем со следующего примера: пользователя просят ввести текст, который записывается в текстовую переменную `text`, в поле ввода диалогового окна, а в следующем окне отображается информация о том, сколько во введенном тексте символов (листинг 9.1).

Листинг 9.1. Пример обработки исключений

```
import static javax.swing.JOptionPane.*;
class Demo{
   public static void main(String[] args){
      String text=showInputDialog(null,
         "Введите текст:", // Надпись над полем
         "Нужен текст", // Заголовок окна
         QUESTION_MESSAGE // Пиктограмма
      );
      try{ // Контролируемый код
         showMessageDialog(null,
            // Сообщение:
            "В тексте "+text.length()+" символов",
            "Символы", // Заголовок окна
            INFORMATION_MESSAGE // Пиктограмма
         );
      }catch(Exception e){ // Обработка ошибки
         showMessageDialog(null,
            "Что-то пошло не так...", // Сообщение
            "Ошибка", // Заголовок окна
            ERROR_MESSAGE // Пиктограмма
         );
      }
   }
}
```

Окно с текстом в поле показано на рис. 9.1.

Если пользователь вводит текст и нажимает кнопку OK, то появляется следующее окно с информацией о количестве символов во введенном тексте (рис. 9.2).

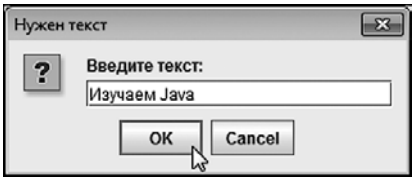


Рис. 9.1. Окно с полем и введенным в него текстом

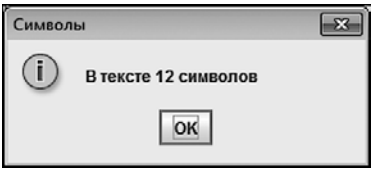


Рис. 9.2. Окно с информацией о количестве символов в тексте

Проблемы могли бы возникнуть, если бы пользователь в окне с полем ввода вместо кнопки OK нажал кнопку Cancel или системную пиктограмму. В таком случае текстовый объект не создается, а переменная `text` содержит пустую ссылку `null`. При попытке сформировать текст сообщения для второго окна из переменной `text` вызывается метод `length()`. Если переменная на объект не ссылается, то возникает ошибка. Поэтому команда с вызовом метода `length()` из переменной `text` помещается в `try`-блок, а для перехвата и обработки ошибки предусмотрен `catch`-блок. В этом блоке есть команда для отображения другого диалогового окна. Поэтому если пользователь передумает вводить текст, то появится окно с сообщением об ошибке (рис. 9.3).

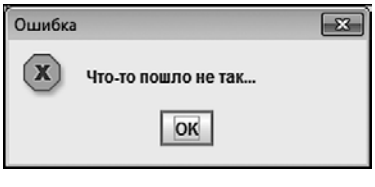


Рис. 9.3. Окно появляется, если пользователь передумал вводить текст

Но самое главное — программа продолжает работу в штатном режиме.

ПОДРОБНОСТИ

Описанная выше ошибка относится к классу `NullPointerException`. Мы в `catch`-блоке указали класс `Exception`. Поскольку класс `Exception` является суперклассом (прямым или опосредованным) для классов перехватываемых исключений, то соответствующий `catch`-блок обрабатывает все ошибки (которые можно обработать), в том числе и ошибки класса `NullPointerException`. Если не угадать тип ошибки в `catch`-блоке, то ошибка таким блоком не перехватывается.

Также стоит отметить, что идентификатор `e` после названия класса `Exception` в `catch`-блоке является ссылкой на объект исключения, который передается в блок для обработки. Просто в нашем случае мы этот объект не используем.

Использование объекта исключения

Ваше Высочество, здесь вам никто ничего не скажет.

из к/ф «Приключения принца Флоризеля»

При обработке ошибки в соответствующий `catch`-блок передается ссылка на объект исключения. Объект создается автоматически и содержит информацию о произошедшей ошибке. Этой информацией можно воспользоваться. Здесь уместно вспомнить, что в вершине иерархии классов, связанных с обработкой исключений, находится класс `Throwable`. Поэтому существует набор методов (из класса `Throwable`), которые есть у любого объекта исключения. Некоторые методы класса `Throwable` перечислены в табл. 9.3.

Таблица 9.3. Методы класса Throwable

Метод	Описание
`fillInStackTrace()`	Метод в качестве результата возвращает объект `Throwable`, который содержит полную трассу стека. Метод не имеет аргументов
`getLocalizedMessage()`	В качестве результата метод возвращает строку (объект класса `String`) с локализованным описанием исключения. Метод не имеет аргументов
`getMessage()`	Методом возвращается строка (объект класса `String`) с описанием исключения. Метод не имеет аргументов
`printStackTrace()`	Методом отображается трасса стека. Метод не имеет аргументов
`toString()`	Метод в качестве значения возвращает объект класса `String`, содержащий описание исключения. Метод не имеет аргументов

В частности, для получения информации об ошибке обычно используют методы `getLocalizedMessage()` и `getMessage()`. Также (явно или неявно) задействуется метод `toString()`. Для объектов исключений он переопределен так, что возвращает строку с описанием ошибки.

НА ЗАМЕТКУ

Напомним, метод `toString()` вызывается автоматически, например, при передаче объекта аргументом методу `println()`.

В листинге 9.2 представлен пример использования объекта исключения.

Листинг 9.2. Использование объекта исключения

```
import java.util.Scanner;
class Demo{
   public static void main(String[] args){
      Scanner p=new Scanner(System.in);
      int a,b;
      System.out.print("Введите первое число: ");
      try{ // Контролируемый код
         // Считывание первого числа:
         a=p.nextInt();
         System.out.print("Введите второе число: ");
         // Считывание второго числа:
         b=p.nextInt();
         System.out.println("Деление нацело:");
         // Одно число делится на другое:
         System.out.println(a+" / "+b+" = "+a/b);
      }catch(Exception e){ // Обработка ошибок
         System.out.println("Произошла ошибка!");
         System.out.println(e);
         System.out.println("Описание: "+e.getMessage());
      }
      System.out.println("Выполнение программы завершено");
   }
}
```

Пользователю предлагается ввести два целых числа, после чего вычисляется результат целочисленного деления одного числа на другое, при этом могут возникнуть две проблемы. Первая связана с тем, что пользователь может ввести не целое число. Вторая проблема связана с возможным делением на ноль (в случае, если второе введенное пользователем число является нулем). Для перехвата и обработки указанных ошибок мы используем конструкцию `try-catch`, причем в `catch`-блоке для отображения информации об ошибке используем объект исключения (который в случае возникновения ошибки передается в `catch`-блок для обработки). Результат выполнения программы зависит от того, какие значения вводит пользователь. Если все происходит в штатном режиме, то результат может быть таким, как показано ниже (здесь и далее жирным шрифтом выделены значения, которые вводит пользователь):

Результат выполнения программы (из листинга 9.2)

```
Введите первое число: 17
Введите второе число: 3
Деление нацело:
17 / 3 = 5
Выполнение программы завершено
```

Если пользователь вводит второе нулевое число, то результат будет таким:

Результат выполнения программы (из листинга 9.2)

```
Введите первое число: 17
Введите второе число: 0
Деление нацело:
Произошла ошибка!
java.lang.ArithmeticException: / by zero
Описание: / by zero
Выполнение программы завершено
```

В этом случае в дело вступает `catch`-блок. По результату приведения объекта исключения к текстовому формату (с помощью неявного вызова метода `toString()`) видим, что произошла ошибка класса `ArithmeticException`. Описание ошибки (текст `"/ by zero"`) является лаконичной реализацией фразы *деление на ноль*.

Если пользователь вводит не целочисленное значение, то возможно следующее:

Результат выполнения программы (из листинга 9.2)

```
Введите первое число: 17
Введите второе число: x
Произошла ошибка!
java.util.InputMismatchException
Описание: null
Выполнение программы завершено
```

Как видим, происходит ошибка класса `InputMismatchException` из пакета `java.util` (ошибка, связанная с некорректно введенным значением), а ключевое слово `null` в описании исключения означает, что особых комментариев по поводу ошибки нет.

Как бы там ни было, но использование объекта позволяет в некоторой степени классифицировать ошибки разных типов. Правда, для решения этой задачи есть более простой и эффективный механизм, основанный на использовании нескольких `catch`-блоков, предназначенных для обработки исключений разных классов.

Использование нескольких catch-блоков

> Все глупости в мире бывают только от умных разговоров.
>
> *из к/ф «Айболит-66»*

Для каждого типа исключений можно предусмотреть свой блок `catch` для обработки. Блоки размещаются один за другим, и для них указываются разные классы исключений (соответственно объекты исключений разных классов).

НА ЗАМЕТКУ

Не следует забывать, что каждый `catch`-блок обрабатывает не только ошибки класса, указанного непосредственно в этом блоке, но и ошибки всех подклассов данного класса. Поэтому, например, если среди `catch`-блоков есть такой, что предназначен для обработки исключений класса `Exception`, то он должен размещаться последним среди `catch`-блоков. В противном случае он бы перехватывал все исключения и до проверки следующих после него `catch`-блоков дело просто бы не дошло.

В листинге 9.3 приведен пример программы, в которой, помимо ошибки деления на ноль, обрабатывается ошибка неверной индексации элементов массива, а также ошибка, связанная с неверным форматом введенных данных. В программе создается массив; пользователя просят ввести два индекса, определяющих последовательность элементов из этого массива. Также в процессе выполнения программы вычисляется число, равное результату целочисленного деления суммы из последовательности элементов на разность индексов. Потенциальные проблемы — следующие:

- Пользователь вводит нечисловое значение для одного из индексов.
- Пользователь вводит числовое значение для индексов, но один или оба индекса выходят за допустимый диапазон значений.
- Пользователь вводит два одинаковых значения для индексов, и в результате при делении суммы элементов на разность индексов возникает ошибка деления на ноль.

Листинг 9.3. Использование нескольких catch-блоков

```
import java.util.*;
class Demo{
   public static void main(String[] args){
      int a,b,s=0;
      Scanner p=new Scanner(System.in);
      // Массив:
      int[] nums={0,1,2,3,4,5,6,7};
      System.out.println("Укажите два индекса (от 0 до "+
      (nums.length-1)+")");
      try{ // Контролируемый код
         System.out.print("Первый индекс: ");
         // Считывается первый индекс:
         a=p.nextInt();
         System.out.print("Второй индекс: ");
         // Считывается второй индекс:
         b=p.nextInt();
         // Отображение элементов массива и вычисление
         // суммы элементов:
         for(int k=a;k<=b;k++){
            s+=nums[k];
            System.out.print(nums[k]+" ");
         }
```

```
            System.out.println();
            System.out.println("Еще одно число: "+s/(b-a));
        }
        // Перехват и обработка ошибок:
        catch(InputMismatchException e){
            System.out.println("Ошибка формата данных");
        }
        catch(ArrayIndexOutOfBoundsException e){
            System.out.println("\nВыход за пределы массива");
        }
        catch(ArithmeticException e){
            System.out.println("Ошибка деления на ноль");
        }
        System.out.println("Выполнение программы завершено");
    }
}
```

Если пользователь вводит два корректных и разных значения для индексов, то получим результат наподобие того, который представлен ниже (здесь и далее жирным шрифтом выделены значения, которые вводит пользователь):

Результат выполнения программы (из листинга 9.3)

```
Укажите два индекса (от 0 до 7)
Первый индекс: 2
Второй индекс: 5
2 3 4 5
Еще одно число: 4
Выполнение программы завершено
```

Если при вводе индекса указать нечисловое значение, в дело вступит обработчик исключения класса `InputMismatchException` и результат будет, например, таким:

Результат выполнения программы (из листинга 9.3)

```
Укажите два индекса (от 0 до 7)
Первый индекс: 2
Второй индекс: пять
Ошибка формата данных
Выполнение программы завершено
```

Если указать два одинаковых значения, то генерируется (и обрабатывается) исключение класса `ArithmeticException`.

Результат выполнения программы (из листинга 9.3)

```
Укажите два индекса (от 0 до 7)
Первый индекс: 5
Второй индекс: 5
5
Ошибка деления на ноль
Выполнение программы завершено
```

Если же для индекса указано целочисленное значение за пределами допустимого диапазона, то возникает ошибка класса `ArrayIndexOutOfBoundsException`. Таким будет результат выполнения программы, если неверно указано значение для второго индекса.

Результат выполнения программы (из листинга 9.3)

```
Укажите два индекса (от 0 до 7)
Первый индекс: 2
Второй индекс: 10
2 3 4 5 6 7
Выход за пределы массива
Выполнение программы завершено
```

Если неверно указан первый индекс, то результат будет таким:

Результат выполнения программы (из листинга 9.3)

```
Укажите два индекса (от 0 до 7)
Первый индекс: -1
Второй индекс: 3
Выход за пределы массива
Выполнение программы завершено
```

Если первый индекс больше второго, то формально ошибок не возникает (даже если оба индекса выходят за допустимые границы), поскольку в этом случае условие в операторе цикла сразу же окажется ложным.

Результат выполнения программы (из листинга 9.3)

```
Укажите два индекса (от 0 до 7)
Первый индекс: 10
Второй индекс: -5
Еще одно число: 0
Выполнение программы завершено
```

Соответственно, команды, в которых могут возникнуть ошибки, не будут выполняться совсем.

Вложенные конструкции try-catch

> Будьте внимательны, помните о строжайшей секретности этой операции.
>
> *из м/ф «Приключения капитана Врунгеля»*

Один блок `try` может размещаться внутри другого блока `try`. Если во внутреннем блоке `try` возникает ошибка и для этого блока `try` нет `catch`-блока, предназначен-

ного для обработки исключений данного класса, то исключение передается для обработки во внешнюю конструкцию `try-catch`. Начинается последовательный просмотр `catch`-блоков, связанных с внешним `try`-блоком, на предмет обработки возникшей ошибки.

НА ЗАМЕТКУ

Если внутренняя конструкция `try-catch` содержит еще и блок `finally`, то перед тем, как передать необработанное исключение для обработки во внешнюю конструкцию `try-catch`, будут выполнены команды из блока `finally`.

Может сложиться и более нетривиальная ситуация, например, когда метод, который вызывается в блоке `try`, сам содержит конструкцию `try-catch`. Если в блоке `try` метода возникает ошибка, не обрабатываемая методом, то ее попытается перехватить внешняя конструкция `try-catch`, в которой вызывается метод.

В листинге 9.4 представлена программа, в которой используются вложенные конструкции `try-catch`.

Листинг 9.4. Вложенные конструкции try-catch

```
import java.util.Random;
class Demo{
   public static void main(String[] args){
      // Объект для генерирования случайных чисел:
      Random r=new Random();
      // Переменные:
      int a,b;
      // Массив:
      int nums[]={-1,1};
      for(int k=1;k<=5;k++){
         System.out.println("Попытка №"+k);
         // Внешний try-блок:
         try{
            a=r.nextInt(3); // Значение 0,1 или 2
            b=100/a; // Возможно деление на ноль
            System.out.println("b="+b);
            // Внутренний try-блок:
            try{
               if(a==1) a=a/(a-1); // Деление на ноль
               else nums[a]=200; // Неправильный индекс
            // Внутренний catch-блок:
            }catch(ArrayIndexOutOfBoundsException e){
               System.out.println(
                  "Неправильный индекс: "+e
               );
            }
            // Блок finally:
            finally{
               System.out.println(
```

```
                "Ошибка происходит всегда!"
            );
        }
      // Внешний catch-блок:
      }catch(ArithmeticException e){
         System.out.println("Деление на ноль: "+e);
      }
   }
}
```

Результат выполнения программы (с учетом того, что используется генератор случайных чисел) может быть таким, как показано ниже:

Результат выполнения программы (из листинга 9.4)

```
Попытка №1
b=50
Неправильный индекс: java.lang.ArrayIndexOutOfBoundsException: 2
Ошибка происходит всегда!
Попытка №2
b=100
Ошибка происходит всегда!
Деление на ноль: java.lang.ArithmeticException: / by zero
Попытка №3
b=100
Ошибка происходит всегда!
Деление на ноль: java.lang.ArithmeticException: / by zero
Попытка №4
b=50
Неправильный индекс: java.lang.ArrayIndexOutOfBoundsException: 2
Ошибка происходит всегда!
Попытка №5
Деление на ноль: java.lang.ArithmeticException: / by zero
```

В программе генерируются случайные числа. Для этого создается объект `r` класса `Random`. В этом объекте мы используем метод `nextInt()`, который результатом возвращает случайное целое число в диапазоне значений от `0` (включительно), строго меньшее числа, указанного аргументом метода. Объявляются две целочисленные переменные `a` и `b`, а также целочисленный массив `nums`, состоящий всего из двух элементов (со значениями `-1` и `1`).

В операторе цикла внешнего блока `try` последовательное выполнение команд `a=r.nextInt(3)` и `b=100/a` может закончиться генерированием исключения, поскольку среди потенциальных значений переменной `a` есть и нулевое, что, в свою очередь, означает ошибку деления на ноль. На этот случай предусмотрен внешний блок `catch` (связанный с внешним блоком `try`). В случае ошибки выполняется команда: `System.out.println("Деление на ноль: "+e)`. Здесь через `e` обозначен объект класса `ArithmeticException`, который передается для обработки в блок `catch`.

Если в указанном месте ошибка деления на ноль не возникает, то значение переменной b выводится на экран (эта переменная может принимать всего два значения: 100 при значении переменной a равном 1 и 50 при значении переменной a равном 2), после чего выполняется серия команд, заключенных во внутренний блок try. В этом блоке обязательно генерируется одно из двух возможных исключений (деление на ноль или выход за пределы массива). Но обрабатывается внутренним блоком catch только исключение, связанное с выходом за пределы массива (исключение класса ArrayIndexOutOfBoundsException). В случае возникновения соответствующей ошибки выполняется команда System.out.println("Неправильный индекс: "+e). Причем после catch-блока расположен блок finally. Код в блоке finally (команда System.out.println("Ошибка происходит всегда!")) выполняется всегда, вне зависимости от того, каким именно catch-блоком перехватывается ошибка.

ПОДРОБНОСТИ

Схема такая. Если при выполнении try-блока ошибок нет (хотя это не наш случай), то catch-блоки игнорируются и выполняется блок finally. Если при выполнении try-блока возникло исключение и оно обрабатывается в catch-блоке, то после выполнения этого блока выполняется блок finally. Если при выполнении try-блока возникает исключение и оно не обрабатывается в catch-блоке, то исключение передается для обработки во внешнюю try-catch-конструкцию, но перед этим выполняются команды из блока finally.

Код во внутреннем блоке try может вызывать исключения двух типов. В частности, с помощью условного оператора проверяется, равно ли значение переменной a единице (условие a==1). Если условие истинное, то при выполнении команды a=a/(a-1) происходит ошибка деления на ноль. В противном случае (то есть если значение переменной a отлично от единицы) выполняется команда nums[a]=200. До выполнения внутреннего блока try дело доходит, если значение переменной a равно 1 или 2. Если же значение переменной a равно 0, то еще раньше возникнет ошибка деления на ноль, которая перехватывается внешним блоком catch (связанным с внешним блоком try). Отсюда следует, что если во внутреннем блоке try значение переменной a отлично от единицы, то это автоматически означает, что значение переменной a равно 2. В результате при выполнении команды nums[a]=200 возникает ошибка выхода за пределы массива, поскольку в массиве nums всего два элемента и элемента с индексом 2 среди них нет.

Таким образом, во внутреннем блоке catch обрабатывается ошибка выхода за пределы массива. Если во внутреннем блоке try возникает ошибка деления на ноль, то она обрабатывается внешним блоком catch. Блок finally в первом случае выполняется после внутреннего catch-блока, а во втором случае блок finally выполняется перед внешним catch-блоком.

НА ЗАМЕТКУ

Если ошибка деления на ноль происходит во внешнем `try`-блоке, то для обработки ошибки выполняется внешний `catch`-блок, а до выполнения блока `finally` из внутренней конструкции `try-catch-finally` дело не доходит.

Еще один пример вложенных конструкций `try-catch` представлен в листинге 9.5. На этот раз в блоке `try` вызывается метод, у которого есть собственная конструкция `try-catch`.

Листинг 9.5. Метод с конструкцией try-catch

```
import java.util.Random;
class Demo{
   // Статический метод:
   static void generator(int a){
      int nums[]={-1,1};
      try{
         if(a==1) a=a/(a-1);
         else nums[a]=200;
      }catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Неправильный индекс: "+e);
      }
      finally{
         System.out.println("Ошибка происходит всегда!");
      }
   }
   // Главный метод:
   public static void main(String[] args){
      Random r=new Random();
      int a,b;
      for(int k=1;k<=5;k++){
         System.out.println("Попытка №"+k);
         try{
            a=r.nextInt(3);
            b=100/a;
            System.out.println("b="+b);
            // Вызов статического метода:
            generator(a);
         }catch(ArithmeticException e){
            System.out.println("Деление на ноль: "+e);
         }
      }
   }
}
```

Результат выполнения программы такой же, как и в предыдущем случае (с поправкой на генерирование случайных чисел). Фактически это предыдущая программа, только код внутреннего блока `try` реализован в виде статического метода `generator()`, содержащего конструкцию `try-catch-finally` (а также удалена основная часть комментариев). Там, где раньше был внутренний блок `try`, теперь

вызывается метод `generator()`. Если при вызове метода возникает ошибка выхода за пределы массива, то она обрабатывается в самом методе. Ошибка деления на ноль обрабатывается во внешнем блоке `catch`.

НА ЗАМЕТКУ

Хотя метод `generator()` может генерировать необрабатываемое исключение класса `ArithmeticException`, в сигнатуре метода через инструкцию `throws` это исключение указывать не нужно. Причина в том, что исключение `ArithmeticException` относится к неконтролируемым.

Генерирование исключений

Бредит, бедняга. Похоже на тропическую лихорадку.

из м/ф «Приключения капитана Врунгеля»

Исключения можно генерировать вручную, то есть создавать видимость ошибки там, где ее нет. Для генерирования исключения используется инструкция `throw`. После инструкции `throw` необходимо указать объект исключения, то есть объект, описывающий создаваемую исключительную ситуацию. Причем предварительно этот объект нужно создать. Команда генерирования исключения имеет следующий синтаксис:

```
throw объект_исключения;
```

Объект исключения — это объект класса `Throwable` или его подкласса. Существуют два способа создания объекта исключения. Во-первых, можно воспользоваться объектом исключения, переданным в блок `catch` для обработки. Во-вторых, можно создать с помощью оператора `new` новый объект исключения на основе библиотечного или собственного класса. Процедура стандартная, подразумевающая вызов конструктора класса соответствующего исключения. Все исключения времени выполнения программы (подклассы класса `RuntimeException`) имеют конструкторы без аргументов и с текстовым аргументом. В последнем случае текст, переданный конструктору, является описанием ошибки и содержится в описании объекта исключения, если последний приводится к текстовому формату (например, при передаче объекта исключения методам `print()` и `println()`).

После выполнения `throw`-инструкции поток выполнения останавливается и начинается поиск блока `catch`, подходящего для обработки сгенерированного исключения. Если такой блок не обнаруживается, то используется обработчик по умолчанию. Пример программы с искусственным генерированием исключения приведен в листинге 9.6.

Листинг 9.6. Генерирование исключения

```
class Demo{
   // Статический метод:
   static void generator(){
      // Контролируемый код:
      try{
         // Создание объекта исключения:
         NullPointerException obj=new NullPointerException(
            "Наша Ошибка"
         );
         // Генерирование исключения:
         throw obj;
      // Обработка исключения:
      }catch(NullPointerException e){
         System.out.println("Первый перехват исключения:");
         System.out.println("<"+e+">");
         // Повторное генерирование исключения:
         throw e;
      }
   }
   // Главный метод:
   public static void main(String[] args){
      // Контролируемый код:
      try{
         generator();
      // Обработка исключения:
      }catch(NullPointerException e){
         System.out.println("Повторный перехват:");
         System.out.println("{"+e+"}");
      }
      System.out.println("Выполнение программы завершено");
   }
}
```

Результат выполнения программы таков:

Результат выполнения программы (из листинга 9.6)

```
Первый перехват исключения:
<java.lang.NullPointerException: Наша Ошибка>
Повторный перехват:
{java.lang.NullPointerException: Наша Ошибка}
Выполнение программы завершено
```

В классе `Demo`, помимо главного метода программы `main()`, описывается метод `generator()`, в котором явно генерируется исключение. Сначала создается объект исключения командой `NullPointerException obj=new NullPointerException("Наша Ошибка")`. Это объект `obj` класса `NullPointerException` (ошибка операции со ссылкой). Конструктору класса `NullPointerException` передается текстовый аргумент

"Наша Ошибка". Этот текст впоследствии используется в описании сгенерированной ошибки. Генерирование ошибки осуществляется командой `throw obj`. Поскольку это происходит в блоке `try`, то начинается поиск подходящего блока `catch` для обработки исключения. В данном случае имеется всего один блок `catch`, и это именно тот блок, который нужен. В этом блоке выводится сообщение об исключении в методе `generator()` посредством команд `System.out.println("Первый перехват исключения:")` и `System.out.println("<"+e+">")` (здесь `e` является ссылкой на объект исключения). После этого командой `throw e` снова генерируется исключение. Чтобы его обработать, нужен внешний `catch`-блок. Поскольку в главном методе программы метод `generator()` вызывается в блоке `try` и для исключения класса `NullPointerException` описан обработчик, то сгенерированное и не обработанное методом `generator()` исключение перехватывается и обрабатывается во внешнем `catch`-блоке.

НА ЗАМЕТКУ

Вместо явного создания в методе `generator()` объекта исключения `obj` можно было ограничиться созданием анонимного объекта, объединив создание объекта исключения и его генерирование в одну команду вида `throw new NullPointerException("Наша Ошибка")`.

ПОДРОБНОСТИ

Сообщение программы о повторном перехвате исключения появляется в результате обработки повторно сгенерированного исключения. Исключение генерируется в методе `generator()`, а обрабатывается — вне этого метода. При генерировании исключения в методе `generator()` объект исключения `obj` передается в блок `catch`. В этом блоке доступ к объекту реализуется через переменную `e`. Но на самом деле это объект, который был создан ранее. Далее в блоке `catch` командой `throw e` исключение генерируется повторно, причем объект исключения — все тот же объект, который ранее был создан и передан в блок `catch`. Речь об объекте `obj`. Второе исключение в методе `generator()` не обрабатывается, а передается во внешний блок `catch` для обработки. Поэтому внешний блок `catch` получает объект исключения (точнее, ссылку на объект), который является, как несложно догадаться, многострадальным объектом `obj`. Как следствие, мы в итоге получаем для исключения описание, которое использовали при создании объекта исключения в методе `generator()`.

НА ЗАМЕТКУ

Метод `generator()` генерирует необрабатываемое исключение класса `NullPointerException`. Поскольку класс относится к неконтролируемым исключениям, то в сигнатуре метода `throws`-инструкцию использовать не нужно.

Методы и контролируемые исключения

Аврал! Всех крокодилов — за борт!

из м/ф «Приключения капитана Врунгеля»

Если метод может сгенерировать необрабатываемое (в методе) контролируемое исключение, то это нужно отразить при описании метода. В сигнатуре метода после ключевого слова `throws` перечисляются классы исключений, которые может генерировать метод, — это позволяет сообщить внешним методам, к каким неприятностям следует быть готовыми при вызове метода.

НА ЗАМЕТКУ

Ранее мы не использовали ключевое слово `throws`, хотя некоторые методы и выбрасывали исключения. Но то были неконтролируемые исключения. Их указывать в сигнатуре метода не нужно. Напомним, что исключения подклассов класса `RuntimeException` являются неконтролируемыми.

Описание метода, генерирующего контролируемые исключения (которые не обрабатываются в методе), выглядит так:

```
тип имя (аргументы) throws исключения{
   // Команды
}
```

Если исключений несколько, то они (названия классов исключений) перечисляются через запятую. Пример использования метода, генерирующего необрабатываемое контролируемое исключение, представлен в листинге 9.7.

Листинг 9.7. Методы и контролируемые исключения

```
class Demo{
   // Метод генерирует контролируемое исключение:
   static void generator() throws IllegalAccessException{
      System.out.println("Выполняется метод");
      // Генерирование исключения:
      throw new IllegalAccessException("Большая Ошибка");
   }
   // Главный метод:
   public static void main(String[] args){
      // Контролируемый код:
      try{
         // Вызов метода, генерирующего исключение:
         generator();
      // Обработка ошибки:
      }catch(IllegalAccessException e){
```

```
            System.out.println("Есть проблема:");
            System.out.println("<"+e+">");
        }
    }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 9.7)

```
Выполняется метод
Есть проблема:
<java.lang.IllegalAccessException: Большая Ошибка>
```

Главная особенность программы в том, что в описании метода `generator()` явно указано, что метод может генерировать (а в нашем случае он всегда генерирует) исключение класса `IllegalAccessException` (ошибка доступа). Методом `generator()` отображается сообщение, а затем командой `throw new IllegalAccessException("Большая Ошибка")` генерируется исключение. Мы используем анонимный объект класса `IllegalAccessException`, при создании которого вызывается конструктор с текстовым аргументом. В методе это исключение не обрабатывается, о чем и свидетельствует наличие в сигнатуре метода ключевого слова `throws` и названия класса исключения `IllegalAccessException`. Отметим, что если бы это исключение в методе обрабатывалось, то не было бы и необходимости указывать в сигнатуре метода ключевое слово `throws` (и класс исключения).

В главном методе программы в `try`-блоке вызывается метод `generator()`, и генерируемое методом исключение обрабатывается (объект сгенерированного методом исключения передается в блок `catch`).

Создание классов исключений

> Мы в некотором роде не совсем гавайцы. Скорее даже совсем не гавайцы.
>
> *из м/ф «Приключения капитана Врунгеля»*

Кроме библиотечных классов исключений можно создавать собственные классы. Чтобы эти классы были встроены в общую схему обработки исключений, они должны наследовать один из библиотечных классов исключений. Например, можно собственный класс создать как подкласс класса `Exception` или как подкласс класса `RuntimeException`. В первом случае мы получим пользовательский класс, который соответствует контролируемому исключению. Во втором случае класс будет относиться к исключениям неконтролируемого типа. Процесс создания пользовательских исключений представлен в листинге 9.8.

Листинг 9.8. Создание класса исключения

```
// Класс исключения:
class MyException extends RuntimeException{
   // Закрытое поле:
   private int code;
   // Конструктор:
   MyException(int code){
      super("Ошибка класса MyException");
      this.code=code;
   }
   // Переопределение методов:
   public String getMessage(){
      return super.getMessage()+"\nКод ошибки "+code;
   }
   public String toString(){
      return "<MyException: "+code+">";
   }
}
// Главный класс:
class Demo{
   // Статический метод:
   static void generator(int n){
      // Генерирование исключения:
      throw new MyException(n);
   }
   // Главный метод:
   public static void main(String[] args){
      // Контролируемый код (внешний блок):
      try{
         // Контролируемый код (внутренний блок):
         try{
            // При вызове метода генерируется исключение:
            generator(123);
         // Обработка исключения (внутренний блок):
         }catch(MyException e){
            System.out.println(e.getMessage());
            // Повторное генерирование исключения:
            throw e;
         }
      // Обработка исключения (внешний блок):
      }catch(Exception e){
         System.out.println(e);
      }
   }
}
```

Результат выполнения программы таков:

Результат выполнения программы (из листинга 9.8)

```
Ошибка класса MyException
Код ошибки 123
<MyException: 123>
```

Класс исключения `MyException` создается наследованием класса `RuntimeException` (поэтому исключения класса `MyException` относятся к неконтролируемым). У класса есть закрытое целочисленное поле `code`. Конструктору передается целое число, которое становится значением поля `code`. Конструктору суперкласса передается текст `"Ошибка класса MyException"`. Этот текст становится описанием ошибки. Такой текст мы получили бы при вызове метода `getMessage()` из объекта исключения. Но мы переопределяем унаследованный из суперкласса метод `getMessage()` так, что новая версия метода возвращает текст `super.getMessage()+"\nКод ошибки "+code`. В данном случае получен текст с описанием ошибки из суперкласса (инструкция `super.getMessage()` означает вызов версии метода из суперкласса), к которому добавлен дополнительный текст.

Кроме того, переопределен метод `toString()` — так, что результатом метод возвращает текст `"<MyException: "+code+">"`.

В главном классе мы описываем статический метод `generator()` с целочисленным аргументом. Метод генерирует исключение класса `MyException`, причем аргумент метода передается конструктору класса `MyException`. Исключение в классе не обрабатывается, но поскольку оно относится к неконтролируемым, то `throws`-инструкцию в сигнатуре метода указывать не нужно.

В главном методе программы, во внешнем `try`-блоке, размещен внутренний `try`-блок, в котором вызывается метод `generator()` с аргументом `123`. Метод генерирует ошибку, и она перехватывается во внутреннем `catch`-блоке. В процессе обработки информацию об ошибке получаем с помощью метода `getMessage()`, после чего снова генерируем исключение, обрабатываемое во внешнем `catch`-блоке. Информация об объекте исключения отображается посредством неявного вызова метода `toString()` из объекта исключения.

НА ЗАМЕТКУ

Во внутреннем `catch`-блоке перехватываются исключения класса `MyException`. Во внешнем `catch`-блоке перехватываются исключения класса `Exception`. Но поскольку класс `MyException` является подклассом класса `RuntimeException`, а он является подклассом `Exception`, то внешний `catch`-блок обрабатывает в том числе и исключения класса `MyException`.

Резюме

Действуйте только по моим инструкциям.

из м/ф «Приключения капитана Врунгеля»

- В Java есть механизм, позволяющий обрабатывать ошибки, которые возникают в процессе выполнения программы.

- Для обработки исключений используют конструкцию `try-catch`. В блоке `try` помещается контролируемый код, блоки `catch` (их может быть несколько) содержат код для обработки ошибок. Если при выполнении команд в `try`-блоке ошибок не было, то `catch`-блоки игнорируются. Если при выполнении `try`-блока возникла ошибка, то выполнение команд в `try`-блоке прекращается и выполняются команды в `catch`-блоке, предназначенном для обработки ошибок данного типа. Также можно использовать блок `finally`, команды в котором выполняются всегда (и если есть ошибки, и если ошибок нет).
- При возникновении ошибки автоматически создается объект, который содержит информацию об ошибке. Он передается в `catch`-блок для обработки.
- Каждому типу ошибки соответствует определенный класс (класс исключения). Классы исключений образуют иерархию наследования, в вершине которой находится класс `Throwable`.
- Исключения делятся на контролируемые и неконтролируемые. Если метод может генерировать исключение контролируемого типа и не обрабатывает его, то в сигнатуре метода через ключевое слово `throws` необходимо указать класс генерируемого исключения.
- Исключения также можно генерировать самостоятельно. В таком случае используется инструкция `throw`, после которой указывается объект генерируемого исключения.
- Помимо использования встроенных исключений, можно создавать собственные классы исключений. Для этого необходимо создать подкласс класса `Exception` или класса `RuntimeException`.

10 Многопоточное программирование

Ты посмотри, как интересно. Похоже на вулкан.

из м/ф «Приключения капитана Врунгеля»

В Java поддерживается *многопоточное программирование*. Многопоточная программа содержит несколько блоков кода, которые могут выполняться одновременно. Каждый такой блок кода называется *потоком*.

НА ЗАМЕТКУ

С понятием *многопоточности* тесно связано понятие *многозадачности*. Многозадачность реализуется, как правило, либо на потоках, либо на *процессах*. Различие между потоками и процессами достаточно зыбкое. Обычно для процессов выделяется отдельная область памяти, которая доступна только для них. Это повышает безопасность, но снижает скорость выполнения программы. На процессах основана работа операционных систем.

При многопоточности обычно память по потокам не разбивается. Хотя такая ситуация может сказаться на стабильности программы, системные ресурсы используются экономнее и программа работает быстрее.

Обычно многопоточное программирование применяют для сведения к минимуму времени простоя системы, поскольку сразу несколько задач могут решаться одновременно.

Создание дочернего потока

Нужно, чтобы она тебя брала. Нужно, чтобы
она тебя вела. Но в то же время и не уводила!

из к/ф «Карнавальная ночь»

При запуске программы автоматически запускается единственный и самый важный поток, который называется *главным*. Из главного потока можно запускать другие потоки, которые называются *дочерними*.

Чтобы разобраться с тем, как в Java реализуются потоки, сначала следует пояснить, для чего они нужны. Реализация потоков необходима для программ с несколькими блоками кода, выполняемыми одновременно.

Мы решаем две задачи. Во-первых, нужно где-то описать код, предназначенный для выполнения в потоке. Во-вторых, необходимо запустить этот код в специальном режиме (чтобы он выполнялся как поток, а не как обычный блок кода). Для решения второй задачи нужен объект класса `Thread`. Если объект класса `Thread` создан, то для запуска потока необходимо из этого объекта вызвать метод `start()`. В результате начнет выполняться поток. Но какой? И здесь мы подходим к решению первой задачи, связанной с организацией кода потока.

Итак, необходимо создать объект на основе класса, реализующего интерфейс `Runnable`, который является функциональным и в нем объявлен один метод `run()` — без аргументов и не возвращающий результат. В классе, который реализует интерфейс, описывается метод `run()`. И это именно тот код, который будет выполняться в потоке. Остается только связать объект класса `Thread`, из которого вызывается метод `start()` для запуска потока, с объектом с методом `run()`, код которого будет выполняться в потоке. Связывание объектов выполняется на этапе создания объекта класса `Thread`.

НА ЗАМЕТКУ

Пикантность ситуации в том, что класс `Thread` реализует интерфейс `Runnable`, но только метод `run()` описан там с пустым кодом. Поэтому вместо того, чтобы создавать два объекта, можно описать подкласс класса `Thread`, переопределить там метод `run()` и использовать этот объект и для запуска потока, и для хранения кода потока.

Далее рассмотрим создание потока путем реализации интерфейса `Runnable`. Для этого опишем класс, реализующий интерфейс `Runnable`, а в этом классе опишем метод `run()`. Код метода `run()` — это и есть тот код, который будет выполняться в рамках создаваемого потока.

ПОДРОБНОСТИ

Говорят, что метод `run()` определяет точку входа в поток. Как отмечалось выше, метод `run()` не имеет аргументов и не возвращает результат. В классе метод должен описываться со спецификатором доступа `public`.

После описания класса, реализующего интерфейс `Runnable`, создадим на его основе объект. Кроме этого, создадим объект класса `Thread`. Конструктору класса `Thread` передается ссылка на объект, созданный на основе класса, реализующего интерфейс `Runnable`. Таким образом, связываются объект, с помощью которого будем запускать поток (объект класса `Thread`), и объект, который содержит метод `run()` с кодом потока. Теперь для запуска потока достаточно из объекта класса `Thread` вызвать метод `start()`.

ПОДРОБНОСТИ

У класса `Thread` есть несколько конструкторов. Среди них конструктор без аргументов, с одним аргументом (объект класса, реализующего интерфейс `Runnable`), и конструктор с двумя аргументами (объект класса, реализующего интерфейс `Runnable` и текст). Если вызывается конструктор без аргументов, то создается объект класса `Thread`, который при запуске потока (вызовом метода `start()`) использует код собственного метода `run()`. Если аргументом конструктора класса `Thread` указан объект класса, реализующего интерфейс `Runnable`, то при запуске потока будет выполняться метод `run()` объекта, переданного аргументом конструктору. Второй текстовый аргумент (если он есть) определяет название создаваемого потока.

Вообще, поток отождествляется с объектом класса `Thread` (или подкласса, созданного на основе класса `Thread`). Поэтому если мы создали объект класса `Thread` и с его помощью запустили поток, то данный объект соответствует потоку и операции с потоком выполняются как операции с объектом. Создание дочернего потока представлено в листинге 10.1.

Листинг 10.1. Создание дочернего потока

```
// Класс реализует интерфейс Runnable:
class MyClass implements Runnable{
   // Закрытое поле – ссылка на текстовый массив:
   private String[] txt;
   // Конструктор:
   MyClass(String[] txt){
      this.txt=txt;
   }
   // Код потока:
   public void run(){
      for(int k=0;k<txt.length;k++){
         try{
            // Задержка в выполнении потока:
            Thread.sleep(1000);
         }
         // Обработка исключения (нет команд):
         catch(InterruptedException e){}
         // Отображение значения элемента массива:
         System.out.println("["+(k+1)+"] "+txt[k]);
      }
```

```
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Массивы:
      String[] cls={"Красный","Желтый","Зеленый","Синий"};
      char[] symbs={'J','A','V','A'};
      // Объект с методом run():
      MyClass obj=new MyClass(cls);
      // Создание объекта потока:
      Thread t=new Thread(obj);
      // Запуск дочернего потока:
      t.start();
      for(int k=0;k<symbs.length;k++){
         try{
            // Задержка в выполнении потока:
            Thread.sleep(1500);
         }
         // Обработка исключения (нет команд):
         catch(InterruptedException e){}
         // Отображение значения элемента массива:
         System.out.println("Главный поток: "+symbs[k]);
      }
   }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 10.1)

```
[1] Красный
Главный поток: J
[2] Желтый
Главный поток: A
[3] Зеленый
[4] Синий
Главный поток: V
Главный поток: A
```

Здесь реализована достаточно простая идея: есть текстовый массив, значения элементов которого отображаются в дочернем потоке. Одновременно в главном потоке отображаются значения элементов другого, символьного массива. Поскольку оба потока завершаются очень быстро, мы для большей наглядности делаем паузы между выполнением команд потоками. Теперь перейдем к анализу кода.

В классе `MyClass` реализуется интерфейс `Runnable`. У класса есть закрытое поле, являющееся ссылкой на текстовый массив. Значение полю присваивается на основе аргумента конструктора. Код, предназначенный для выполнения в дочернем потоке, содержится в методе `run()`. Там перебираются элементы текстового мас-

сива, и значения элементов отображаются в области вывода. Незнакомой для нас является лишь команда `Thread.sleep(1000)`. Благодаря этой инструкции поток приостанавливает выполнение на одну секунду. В данном случае из класса `Thread` вызывается статический метод `sleep()`. В результате вызова метода поток, в котором вызывается метод, приостанавливает работу.

НА ЗАМЕТКУ

Поскольку код метода `run()` предназначен для выполнения в дочернем потоке, то вызываться метод `sleep()` будет в дочернем потоке, и, соответственно, в данном случае речь идет о приостановке выполнения дочернего потока.

Время, на которое поток приостанавливает работу, определяется аргументом метода `sleep()`. Время задается в миллисекундах (`1000` миллисекунд — это `1` секунда).

Метод `sleep()` может генерировать необрабатываемое исключение класса `InterruptedException` (связано с тем, что один поток прерывает работу другого потока). Оно относится к контролируемым, поэтому мы должны его обработать. Для обработки исключения инструкцию с вызовом метода `sleep()` размещаем в `try`-блоке. Блок `catch`, предназначенный для обработки исключения, пустой, поскольку мы не предполагаем выполнения каких-либо действий на случай, если один поток вмешается в работу другого потока.

В главном методе создается текстовый массив `cls` и символьный массив `symbs`. Мы создаем объект `obj` класса `MyClass`; конструктору класса передается ссылка на массив `cls`. Поэтому при выполнении дочернего потока будут отображаться значения элементов массива `cls`. Объект потока `t` создается как объект класса `Thread`, а аргументом конструктору передается объектная переменная `obj`. Поэтому когда командой `t.start()` из объекта `t` вызывается метод `start()`, то в режиме дочернего потока начинает выполняться метод `run()` из объекта `obj`.

При этом в главном методе (в основном потоке) с помощью оператора цикла отображаются значения элементов массива `symbs`. Между отображением значений элементов делается пауза в полторы секунды (аргументом методу `sleep()` передано значение `1500`).

НА ЗАМЕТКУ

Как и в случае с дочерним потоком, поскольку в главном потоке вызывается метод `sleep()`, то нам следует организовать обработку исключения класса `InterruptedException`. Поэтому инструкция вызова метода `sleep()` помещается в `try`-блок. В `catch`-блоке никакие команды не выполняются.

В программе одновременно выполняются два оператора цикла. Между командами делаются заметные паузы, так что процесс выполнения потоков можно наблюдать воочию.

НА ЗАМЕТКУ

Командой `t.start()` запускается дочерний поток. В потоке выполняется метод `run()` из объекта `obj`. Оператор цикла (после команды запуска потока) выполняется сразу, не дожидаясь завершения метода `run()`. Если бы мы запустили метод `run()` из объекта `obj` не в режиме дочернего потока, а как обычно вызываются методы, то сначала завершилось бы выполнение метода `run()` и только после этого начал бы выполняться оператор цикла.

В рассмотренном примере явно использовались два объекта: объект с кодом, выполняемым в потоке, и объект, с помощью которого запускался поток. Но возможны и более замысловатые комбинации. Например, процесс создания дочернего потока можно реализовать по следующей схеме:

- В классе, реализующем интерфейс `Runnable` (пусть для определенности это будет класс `MyClass`), не только описываем метод `run()`, но и создаем поле, являющееся ссылкой на объект класса `Thread`.
- Объект потока (объект класса `Thread`) создается в конструкторе класса `MyClass`, причем первым аргументом конструктору класса `Thread` передается ссылка `this` на создаваемый объект класса `MyClass`. Таким образом, одновременно с созданием объекта класса `MyClass` создается и объект потока.
- В конструкторе класса `MyClass` можно также вызвать из объекта потока (объекта класса `Thread`) метод `start()`. Это приводит к запуску потока. В таком случае создание объекта класса `MyClass` будет означать автоматическое создание и запуск дочернего потока.

Пример использования описанного выше подхода представлен в листинге 10.2.

Листинг 10.2. Еще один способ создать дочерний поток

```
// Класс реализует интерфейс Runnable:
class MyClass implements Runnable{
   // Поле – ссылка на массив:
   private String[] txt;
   // Поле – ссылка на объект потока:
   private Thread t;
   // Конструктор класса:
   MyClass(String[] txt,String name){
      // Ссылка на массив:
      this.txt=txt;
      // Создание объекта потока:
      t=new Thread(this,name);
      // Отображение названия потока:
      System.out.println("Создан поток: "+t.getName());
      // Запуск потока на выполнение:
      t.start();
   }
```

```
   // Код для выполнения в потоке:
   public void run(){
      for(int k=0;k<txt.length;k++){
         // Задержка в выполнении потока:
         try{
            Thread.sleep(1000);
         }
         // Обработка исключения (нет команд):
         catch(InterruptedException e){}
         System.out.println("["+(k+1)+"] "+txt[k]);
      }
      System.out.println("Завершен поток "+t.getName());
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Массивы:
      String[] cls={"Красный","Желтый","Зеленый","Синий"};
      char[] symbs={'J','A','V','A'};
      // Создание анонимного объекта:
      new MyClass(cls,"ДОЧЕРНИЙ");
      for(int k=0;k<symbs.length;k++){
         // Задержка в выполнении потока:
         try{
            Thread.sleep(1500);
         }
         // Обработка исключения (нет команд):
         catch(InterruptedException e){}
         System.out.println("Главный поток: "+symbs[k]);
      }
      System.out.println("Главный поток завершен");
   }
}
```

Результат выполнения программы — практически такой же, как и в предыдущем случае:

Результат выполнения программы (из листинга 10.2)

```
Создан поток: ДОЧЕРНИЙ
[1] Красный
Главный поток: J
[2] Желтый
Главный поток: A
[3] Зеленый
[4] Синий
Завершен поток ДОЧЕРНИЙ
Главный поток: V
Главный поток: A
Главный поток завершен
```

Мы создаем класс `MyClass`, реализующий интерфейс `Runnable`. У класса есть два поля: одно для записи ссылки на текстовый массив, другое является ссылкой на объект класса `Thread`. В конструкторе класса `MyClass` командой `t=new Thread(this,name)` полю `t` в качестве значения присваивается ссылка на создаваемый объект класса `Thread`. Поскольку первым аргументом конструктору класса `Thread` передается ссылка `this` на объект класса `MyClass`, то при запуске потока будет выполняться код метода `run()` из этого объекта. Вторым аргументом конструктору класса `Thread` передается текстовый аргумент `name` конструктора класса `MyClass`. Этот текстовый аргумент определяет *название потока*. Его можно получить с помощью метода `getName()`, который вызывается из объекта потока. Также в конструкторе класса `MyClass` командой `t.start()` дочерний поток запускается на выполнение.

Поскольку класс `MyClass` реализует интерфейс `Runnable`, то в классе описывается метод `run()`. В этом методе с интервалом в одну секунду отображается сообщение со значением элемента текстового массива, на который ссылается поле `txt` объекта с методом `run()`.

В главном методе командой `new MyClass(cls, "ДОЧЕРНИЙ")` создается анонимный объект класса `MyClass`, чем автоматически запускается дочерний поток (с названием `"ДОЧЕРНИЙ"`). В дочернем потоке отображаются значения из массива `cls`. Затем в главном потоке с интервалом в полторы секунды отображаются сообщения со значениями массива `symbs` (массивы `cls` и `symbs` создаются в главном методе).

НА ЗАМЕТКУ

Интервалы между сообщениями в главном и дочернем потоках подобраны так, что дочерний поток завершает выполнение раньше главного. В принципе, может случиться, что главный поток завершает выполнение раньше, чем запущенные из него дочерние потоки.

Как уже отмечалось ранее, класс `Thread` реализует интерфейс `Runnable`. Это означает, что у класса `Thread`, кроме метода `start()`, есть и метод `run()`. Правда, метод `run()` описан с пустым телом, поэтому при вызове метода ничего не происходит. Но если мы путем наследования создадим подкласс на основе класса `Thread` и переопределим в нем метод `run()`, то у этого подкласса будет и метод `start()`, и метод `run()` (с нужным нам кодом). Следовательно, с помощью объекта данного подкласса можно создать поток и запустить его. Таков еще один способ создания потоков.

НА ЗАМЕТКУ

До этого у нас (явно или неявно) было два объекта. Один содержал код для выполнения в потоке, а другой использовался для запуска потока. Теперь ситуация меняется: один и тот же объект и содержит код для выполнения в потоке, и используется для запуска потока.

В листинге 10.3 представлен пример создания дочернего потока на основе объекта подкласса, созданного наследованием класса `Thread`.

Листинг 10.3. Поток на основе объекта подкласса

```
// Класс наследует Thread:
class MyClass extends Thread{
   // Поле – ссылка на массив:
   private String[] txt;
   // Конструктор класса:
   MyClass(String[] txt,String name){
      // Вызов конструктора суперкласса:
      super();
      // Задается название потока:
      setName(name);
      // Ссылка на массив:
      this.txt=txt;
      // Отображение названия потока:
      System.out.println("Создан поток: "+getName());
      // Запуск потока на выполнение:
      start();
   }
   // Код для выполнения в потоке:
   public void run(){
      for(int k=0;k<txt.length;k++){
         // Задержка в выполнении потока:
         try{
            Thread.sleep(1000);
         }
         // Обработка исключения (нет команд):
         catch(InterruptedException e){}
         System.out.println("["+(k+1)+"] "+txt[k]);
      }
      System.out.println("Завершен поток "+getName());
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Массивы:
      String[] cls={"Красный","Желтый","Зеленый","Синий"};
      char[] symbs={'J','A','V','A'};
      // Создание анонимного объекта:
      new MyClass(cls,"ДОЧЕРНИЙ");
      for(int k=0;k<symbs.length;k++){
         // Задержка в выполнении потока:
         try{
            Thread.sleep(1500);
         }
         // Обработка исключения (нет команд):
         catch(InterruptedException e){}
         System.out.println("Главный поток: "+symbs[k]);
      }
      System.out.println("Главный поток завершен");
   }
}
```

Результат выполнения программы — точно такой, как и в предыдущем примере. Фактически мы создали очень похожую программу, но немного иначе реализовали класс `MyClass`, который теперь наследует класс `Thread`. Необходимость в поле, которое ссылается на объект класса `Thread`, отпала.

В конструкторе класса `MyClass` первой командой вызывается конструктор суперкласса `Thread` без аргументов. В таком случае объект потока использует собственный метод `run()`, который мы в классе `MyClass` переопределяем. Чтобы присвоить название потоку, используем метод `setName()`. Для считывания названия потока используется уже знакомый нам метод `getName()`. Запускается поток методом `start()`, который вызывается в конструкторе класса `MyClass`.

Еще один концептуальный подход в создании дочернего потока основывается на использовании лямбда-выражений. Дело в том, что интерфейс `Runnable` является функциональным: у него всего один абстрактный метод `run()`. Поэтому если интерфейсной переменной типа `Runnable` значением присвоить лямбда-выражение, то автоматически на основе анонимного класса, реализующего интерфейс, будет создан объект, и ссылка на него записывается в интерфейсную переменную. Метод `run()` в этом объекте определяется лямбда-выражением, которое присваивается интерфейсной переменной. Конструктору класса `Thread` первым аргументом передается значение типа `Runnable`, то есть это может быть ссылка на любой объект, созданный на основе класса, реализующего интерфейс `Runnable`. Если передать первым аргументом лямбда-выражение, то такой объект будет автоматически создан, и ссылка на него передается первым аргументом конструктору. На этом принципе базируется применение лямбда-выражений для создания дочерних потоков (листинг 10.4).

Листинг 10.4. Поток на основе лямбда-выражения

```
class Demo{
   public static void main(String[] args){
      // Массивы:
      String[] cls={"Красный","Желтый","Зеленый","Синий"};
      char[] symbs={'J','A','V','A'};
      // Создание объекта класса Thread:
      Thread t=new Thread(()->{
         for(int k=0;k<cls.length;k++){
            // Задержка в выполнении потока:
            try{
               Thread.sleep(1000);
            }
            // Обработка исключения (нет команд):
            catch(InterruptedException e){}
            System.out.println("["+(k+1)+"] "+cls[k]);
         }
      });
      // Запуск дочернего потока на выполнение:
      t.start();
      for(int k=0;k<symbs.length;k++){
```

```
         // Задержка в выполнении главного потока:
         try{
            Thread.sleep(1500);
         }
         // Обработка исключения (нет команд):
         catch(InterruptedException e){}
         System.out.println("Главный поток: "+symbs[k]);
      }
   }
}
```

Результат выполнения программы аналогичен результату выполнения программы из листинга 10.1.

Управление потоками

> Бабу-ягу со стороны приглашать не будем.
> Воспитаем в своем коллективе!
>
> *из к/ф «Карнавальная ночь»*

В предыдущих примерах главный и дочерний потоки выполнялись фактически независимо один от другого. Но во многих ситуациях необходимо из одного потока в той или иной мере управлять другим потоком. Рассмотрим некоторые из них, например, когда один поток (например, главный) должен дождаться окончания работы другого потока (дочернего, листинг 10.5).

Листинг 10.5. Ожидание завершения потока

```
// Класс для создания дочернего потока:
class MyThread extends Thread{
   public void run(){
      System.out.println("Выполняется дочерний поток");
      for(int k=1;k<5;k++){
         try{
            Thread.sleep(1500);
         }catch(InterruptedException e){}
         System.out.println("Дочерний поток: "+k);
      }
      System.out.println("Дочерний поток завершен");
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      System.out.println("Выполняется главный поток");
      // Создание и запуск дочернего потока:
      MyThread mt=new MyThread();
```

```
      mt.start();
      // Ожидание завершения дочернего потока:
      try{
         if(mt.isAlive()) mt.join();
      }catch(InterruptedException e){}
      System.out.println("Главный поток завершен");
   }
}
```

Результат выполнения программы таков:

Результат выполнения программы (из листинга 10.5)

```
Выполняется главный поток
Выполняется дочерний поток
Дочерний поток: 1
Дочерний поток: 2
Дочерний поток: 3
Дочерний поток: 4
Дочерний поток завершен
Главный поток завершен
```

В главном потоке отображается начальное сообщение, затем запускается дочерний поток (он с интервалом в полторы секунды отображает пять сообщений, не считая начального и последнего), после чего главным потоком отображается последнее сообщение. Пикантность ситуации в том, что это последнее сообщение отображается только после того, как завершит работу дочерний поток (хотя исходя из времени выполнения потоков главный поток должен был бы завершить работу раньше). Чтобы добиться желаемого эффекта, мы из объекта дочернего потока `mt` вызываем метод `join()` (команда `mt.join()`). Последствия вызова метода `join()` таковы: поток, в котором вызывается метод, приостанавливает свое выполнение до того момента, пока не завершится поток, из объекта которого вызывался метод. В данном случае главный поток будет дожидаться завершения выполнения дочернего потока, реализованного через объект `mt`.

ПОДРОБНОСТИ

Метод `join()` может генерировать контролируемое исключение класса `InterruptedException`. Поэтому инструкция с вызовом метода помещается в блок `try` (блок `catch` не содержит команд). Метод `join()` вызывается в условном операторе. Условием является выражение `mt.isAlive()`. Метод `isAlive()` возвращает значение `true`, если поток, из объекта которого вызывается метод, продолжает выполнение. В противном случае метод возвращает значение `false`. Мы, таким образом, действуем по схеме «если поток еще выполняется, дождаться его завершения».

Ситуация может быть немного иной: например, мы можем захотеть, чтобы один поток остановил выполнение другого потока (листинг 10.6).

Листинг 10.6. Принудительное завершение потока

```
// Класс для создания дочернего потока:
class MyThread extends Thread{
   public void run(){
      // Контролируемый код:
      try{
         int k=1;
         System.out.println("Выполняется дочерний поток");
         // Бесконечный цикл:
         while(true){
            // Задержка в выполнении потока:
            Thread.sleep(1500);
            System.out.println("Дочерний поток: "+k++);
         }
      // Обработка прерывания потока:
      }catch(InterruptedException e){
         System.out.println("Дочерний поток завершен");
      }
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      System.out.println("Выполняется главный поток");
      // Создание и запуск дочернего потока:
      MyThread mt=new MyThread();
      mt.start();
      // Контролируемый код:
      try{
         // Задержка в выполнении главного потока:
         Thread.sleep(8000);
         // Прерывание дочернего потока:
         mt.interrupt();
         // Ожидание завершения дочернего потока:
         if(mt.isAlive()) mt.join();
      // Обработка исключения (нет команд):
      }catch(InterruptedException e){}
      System.out.println("Главный поток завершен");
   }
}
```

Ниже показано, как может выглядеть результат выполнения программы:

Результат выполнения программы (из листинга 10.6)

```
Выполняется главный поток
Выполняется дочерний поток
Дочерний поток: 1
Дочерний поток: 2
Дочерний поток: 3
Дочерний поток: 4
Дочерний поток: 5
Дочерний поток завершен
Главный поток завершен
```

Как видим, в данном случае создается дочерний поток, в котором выполняется оператор цикла (формально бесконечный, поскольку условием указано значение `true`), позволяющий отображать сообщения (интервал между сообщениями составляет полторы секунды). Весь оператор цикла помещен в `try`-блок, а в блоке `catch` обрабатывается исключение класса `InterruptedException` (отображается сообщение о завершении дочернего потока).

В главном методе программы создается и запускается дочерний поток. Затем делается пауза в восемь секунд, после чего командой `mt.interrupt()` выполняется прерывание дочернего потока. В результате дочерний поток прерывается, и в дело вступает обработчик (`catch`-блок из метода `run()`). После прерывания дочернего потока главный поток дожидается его завершения (использована команда `mt.join()`), поскольку процесс завершения дочернего потока может занять некоторое время.

Дочерний поток выполняется около восьми секунд, и за это время он успевает отобразить порядка пяти сообщений (не считая начального и самого последнего).

Фоновые потоки

> Были демоны — мы этого не отрицаем. Но они самоликвидировались. Так что прошу эту глупую панику прекратить.
>
> *из к/ф «Иван Васильевич меняет профессию»*

Потоки бывают *приоритетными* и *фоновыми* (их иногда еще называют *демон-потоками*). По умолчанию поток, который создается, является приоритетным. Поэтому и все потоки, с которыми мы до этого имели дело, являлись приоритетными. Чтобы сделать поток фоновым, из объекта потока следует вызвать метод `setDaemon()` с аргументом `true`. Особенность фонового потока в том, что он автоматически завершается при завершении главного потока (листинг 10.7).

Листинг 10.7. Фоновый поток

```
class MyThread extends Thread{
   public void run(){
      try{
         int k=1;
         System.out.println("Выполняется дочерний поток");
         while(true){
            Thread.sleep(1500);
            System.out.println("Дочерний поток: "+k++);
         }
      }catch(InterruptedException e){}
   }
}
class Demo{
```

```
   public static void main(String[] args){
      System.out.println("Выполняется главный поток");
      MyThread mt=new MyThread();
      // Фоновый поток:
      mt.setDaemon(true);
      mt.start();
      try{
         Thread.sleep(8000);
      }catch(InterruptedException e){}
      System.out.println("Главный поток завершен");
   }
}
```

Как будет выглядеть результат выполнения программы, показано ниже:

Результат выполнения программы (из листинга 10.7)

```
Выполняется главный поток
Выполняется дочерний поток
Дочерний поток: 1
Дочерний поток: 2
Дочерний поток: 3
Дочерний поток: 4
Дочерний поток: 5
Главный поток завершен
```

После создания объекта потока `mt` командой `mt.setDaemon(true)` превращаем этот поток в фоновый. Поэтому при завершении выполнения главного потока работа фонового потока завершается автоматически.

НА ЗАМЕТКУ

В дочернем потоке интервал между сообщениями достаточно большой (составляет полторы секунды). Поэтому при завершении главного потока после последней выполненной команды в главном потоке никаких сообщений от дочернего потока не отображается. Если бы дочерний поток отображал сообщения с меньшим интервалом, то после последнего сообщения главного потока могло бы появиться еще несколько сообщений от дочернего потока.

Создание нескольких потоков

Ой, что было, шеф, что было! Это было так интересно!

из м/ф «Приключения капитана Врунгеля»

Дочерних потоков может быть несколько. Так, в листинге 10.8 представлен пример программы, в которой создаются три дочерних потока.

Листинг 10.8. Несколько дочерних потоков

```
// Импорт класса Date:
import java.util.Date;
// Класс MyThread наследует класс Thread:
class MyThread extends Thread{
   // Время задержки и количество итераций:
   private int time;
   private int count;
   // Конструктор:
   MyThread(String name,int time,int count){
      super(name);
      this.time=time;
      this.count=count;
      System.out.print("Создан поток: "+getName()+". ");
      // Дата и время:
      System.out.println("Время: "+new Date());
      // Запуск потока:
      start();
   }
   // Код для выполнения в потоке:
   public void run(){
      try{
         for(int k=1;k<=count;k++){
            System.out.print(
               "["+k+"/"+count+"] "+getName()+". "
         );
         // Дата и время:
         System.out.println("Время: "+new Date());
         // Задержка в выполнении потока:
         Thread.sleep(time);
      }
      // Обработка исключения (нет команд):
      }catch(InterruptedException e){}
      System.out.print("Завершен поток "+getName()+". ");
      // Дата и время:
      System.out.println("Время: "+new Date());
   }
}
// Главный класс:
class Demo{
   // Исключение InterruptedException в методе main()
   // не обрабатывается:
   public static void main(String args[]) throws InterruptedException{
      System.out.print("Главный поток. ");
      // Дата и время:
      System.out.println("Время: "+new Date());
      // Создание трех дочерних потоков:
      MyThread A=new MyThread("Alpha",5000,5);
      MyThread B=new MyThread("Bravo",6000,4);
      MyThread C=new MyThread("Charlie",7000,3);
      // Ожидание завершения дочерних потоков:
      if(A.isAlive()) A.join();
```

```
      if(B.isAlive()) B.join();
      if(C.isAlive()) C.join();
      System.out.print("Главный поток. ");
      // Дата и время:
      System.out.println("Время: "+new Date());
   }
}
```

Результат выполнения программы таков:

Результат выполнения программы (из листинга 10.8)

```
Главный поток. Время: Sun Mar 17 21:32:33 EET 2019
Создан поток: Alpha. Время: Sun Mar 17 21:32:34 EET 2019
Создан поток: Bravo. Время: Sun Mar 17 21:32:34 EET 2019
Создан поток: Charlie. Время: Sun Mar 17 21:32:34 EET 2019
[1/5] Alpha. Время: Sun Mar 17 21:32:34 EET 2019
[1/3] Charlie. Время: Sun Mar 17 21:32:34 EET 2019
[1/4] Bravo. Время: Sun Mar 17 21:32:34 EET 2019
[2/5] Alpha. Время: Sun Mar 17 21:32:39 EET 2019
[2/4] Bravo. Время: Sun Mar 17 21:32:40 EET 2019
[2/3] Charlie. Время: Sun Mar 17 21:32:41 EET 2019
[3/5] Alpha. Время: Sun Mar 17 21:32:44 EET 2019
[3/4] Bravo. Время: Sun Mar 17 21:32:46 EET 2019
[3/3] Charlie. Время: Sun Mar 17 21:32:48 EET 2019
[4/5] Alpha. Время: Sun Mar 17 21:32:49 EET 2019
[4/4] Bravo. Время: Sun Mar 17 21:32:52 EET 2019
[5/5] Alpha. Время: Sun Mar 17 21:32:54 EET 2019
Завершен поток Charlie. Время: Sun Mar 17 21:32:55 EET 2019
Завершен поток Bravo. Время: Sun Mar 17 21:32:58 EET 2019
Завершен поток Alpha. Время: Sun Mar 17 21:32:59 EET 2019
Главный поток. Время: Sun Mar 17 21:32:59 EET 2019
```

У конструктора класса `MyThread`, наследующего класс `Thread`, три аргумента: имя потока (передается аргументом методу `setName()`), интервал задержки при отображении сообщений (поле `time`) и количество сообщений (поле `count`).

Основу кода метода `run()`, переопределяемого в классе `MyThread`, составляет оператор цикла. Количество итераций определяется полем `count`. За каждую итерацию отображается сообщение с названием потока (получаем вызовом метода `getName()`), номером сообщения, а также текущей датой и временем. Для получения даты и времени используется анонимный объект класса `Date`. Класс `Date` импортируется командой `import java.util.Date` в заголовке программы, а анонимный объект создается командой `new Date()`. При передаче такого объекта методу `println()` отображаются текущие (на момент создания объекта класса `Date`) системные дата и время. После отображения сообщения командой `Thread.sleep(time)` выполняется задержка в выполнении потока.

В главном методе программы на основе класса `MyThread` создается три объекта, что означает запуск трех дочерних потоков.

Еще одна особенность рассмотренного выше кода — в том, что в методе `main()` не производится обработка исключения `InterruptedException`. Поэтому в сигнатуру метода `main()` вынесена инструкция `throws InterruptedException`, означающая, что метод может генерировать необрабатываемое исключение соответствующего типа.

Главный поток

> А голова — предмет темный, исследованию не подлежит.
>
> *из к/ф «Формула любви»*

Напомним, при запуске программы автоматически начинает выполняться главный поток. Особенность главного потока — в том, что там порождаются все дочерние потоки. Фактически главный поток отождествляется с программой. Причем в отличие от дочерних потоков главный поток создается автоматически. Поэтому в предыдущих примерах никаких дополнительных действий для создания главного потока не предпринималось. Вместе с тем главным потоком можно управлять, предварительно получив к нему доступ с помощью статического метода `currentThread()` класса `Thread`. Методом в качестве результата возвращается ссылка на поток, в котором он вызывается. Если вызвать метод в главном потоке, то получим ссылку на объект этого потока (хотя мы его явно не создавали — он создается автоматически). В листинге 10.9 показано выполнение операций с объектом главного потока.

Листинг 10.9. Главный поток программы

```
class Demo{
   public static void main(String[] args){
      // Объектная переменная t класса Thread:
      Thread t;
      // Получение ссылки на главный поток:
      t=Thread.currentThread();
      // Отображение информации о потоке:
      System.out.println("Главный поток: "+t);
      // Потоку присваивается имя:
      t.setName("Demo");
      // Отображение информации о потоке:
      System.out.println("Поток поменял имя: "+t);
      // Изменение приоритета потока:
      t.setPriority(7);
      // Отображение информации о потоке:
      System.out.println("Поток поменял приоритет: "+t);
   }
}
```

Результат выполнения программы будет таким:

Результат выполнения программы (из листинга 10.9)

```
Главный поток: Thread[main,5,main]
Поток поменял имя: Thread[Demo,5,main]
Поток поменял приоритет: Thread[Demo,7,main]
```

В главном методе программы командой `Thread t` объявляется объектная переменная `t` класса `Thread`. Значением этой переменной присваивается ссылка на главный поток — посредством команды `t=Thread.currentThread()`. После этого для обращения к главному потоку можно воспользоваться переменной `t`.

НА ЗАМЕТКУ

Хочется обратить внимание читателя на уже упоминавшийся факт: главный поток создается автоматически. Он существует безотносительно того, объявляем мы переменную `t` или нет. Эта переменная нужна лишь для того, чтобы идентифицировать поток, так сказать, поймать его за руку.

Командой `System.out.println("Главный поток: "+t)` в области вывода отображается информация о главном потоке. Объект `t`, переданный аргументом методу `println()`, приводится к текстовому формату (благодаря переопределенному для класса `Thread` методу `toString()`). В результате появляется сообщение, которое содержит блок `Thread[main,5,main]`. Это и есть результат приведения объекта потока к текстовому формату. В квадратных скобках после ключевого слова `Thread` указывается имя потока, приоритет и группа потока.

НА ЗАМЕТКУ

Приоритет потока — это число в диапазоне от `1` до `10`. Само значение приоритета особой важности не имеет, важно только, у какого потока оно больше. Если несколько потоков конкурируют за ресурс, то преимущество будет у того потока, у которого приоритет больше. По умолчанию приоритет потока равен `5`. Изменить значение приоритета потока можно с помощью метода `setPriority()`. Чтобы узнать значение приоритета потока, используют метод `getPriority()`.

У каждого потока есть имя. Главный поток по умолчанию называется `main`. Еще потоки разбиваются на группы. Главный поток относится к группе `main`.

Командой `t.setName("Demo")` потоку присваивается новое имя. Новое значение для приоритета потока задаем командой `t.setPriority(7)`.

Синхронизация потоков

Только не это... Только не это, только не это, шеф!

из м/ф «Приключения капитана Врунгеля»

При работе с потоками приходится решать проблему *синхронизации потоков*. Обычно необходимость в синхронизации возникает, если разные потоки имеют доступ к одному и тому же ресурсу. Какие в таком случае могут появиться проблемы?

Допустим, имеется банковский счет. Доступ к счету имеют несколько человек, которые могут одновременно вносить деньги на счет и снимать деньги со счета. Процесс изменения состояния счета можно отождествить с потоком. Таким образом, может выполняться сразу несколько потоков.

Процесс изменения состояния счета состоит из двух этапов. Сначала сумма, находящаяся на счету, считывается. Затем с прочитанным значением выполняется нужная операция, после чего вычисленное значение регистрируется как новое состояние счета. Если в процесс изменения состояния счета между считыванием и записью нового значения вклинится другой поток, последствия могут быть катастрофическими. Например, пусть значение счета равно `1000`. Первым потоком значение счета увеличивается на `500`, а второй поток уменьшает значение счета на `300`. Несложно понять, что новое значение счета должно быть равно `1200`. А теперь представим следующую ситуацию. Первым потоком прочитано значение `1000`. Далее первый поток должен увеличить данное значение на `500`. Но до этого второй поток также прочитает значение `1000`. Второй поток должен уменьшить данное значение на `300`. Первый поток вычисляет значение `1500` и записывает его как новое значение счета. После чего второй поток вычисляет значение `700` и тоже записывает его как новое значение счета. То есть получилось не очень хорошо (для владельцев счета).

Другой пример — продажа железнодорожных билетов из разных касс. В этом случае из базы данных считывается информация о наличии свободных мест, и на одно из них выписывается билет. Место, на которое продан билет, помечается как занятое. Понятно, что если подобные операции выполняются сразу несколькими потоками, возможны неприятности, поскольку на одно и то же место могут продать несколько билетов. Это случится, если при выписке билета другой поток успеет прочитать из базы данных старую информацию, в которой не отражены вносимые при покупке билета изменения. Поэтому потоки синхронизируют, что делает невозможным ситуации, подобные описанным.

Теперь вернемся к программированию и рассмотрим два способа создания синхронизированного кода:

- создание синхронизированных методов;
- создание синхронизированных блоков.

В обоих случаях используется ключевое слово `synchronized`. Если создается синхронизированный метод, то ключевое слово `synchronized` указывается в его сигнатуре. При вызове синхронизированного метода потоком другие потоки на этом методе блокируются — они не смогут его вызвать, пока работу с методом не завершит вызвавший его поток.

Можно также синхронизировать объект в блоке команд. Для этого блок выделяется фигурными скобками, перед которыми указывается ключевое слово `synchronized`, а в скобках после этого ключевого слова — синхронизируемый объект. Если с таким объектом начинает работать поток, то для других потоков объект будет недоступен. Пример программы с синхронизированным методом представлен в листинге 10.10.

Листинг 10.10. Синхронизированный метод

```
// Класс для реализации потока:
class MyThread extends Thread{
   // Конструктор:
   MyThread(String name){
      super(name);
      start();
   }
   // Код для выполнения в потоке:
   public void run(){
      // Вызов статического метода:
      Demo.show();
   }
}
// Главный класс:
class Demo{
   // Статический метод:
   public synchronized static void show(){
      // Получение ссылки на поток, в котором
      // вызывается метод:
      Thread t=Thread.currentThread();
      // Отображение сообщений:
      for(int k=1;k<=3;k++){
         System.out.println(t.getName()+" ["+k+"]");
         // Задержка в выполнении потока:
         try{
            Thread.sleep(1000);
         }
         catch(InterruptedException e){}
      }
   }
   // Главный метод:
   public static void main(String[] args){
      // Создание дочерних потоков:
      new MyThread("Alpha");
      new MyThread("Bravo");
   }
}
```

Результат выполнения программы может быть таким:

Результат выполнения программы (из листинга 10.10)

```
Alpha [1]
Alpha [2]
Alpha [3]
Bravo [1]
Bravo [2]
Bravo [3]
```

В рассматриваемом примере класс для реализации потока `MyThread` создается наследованием класса `Thread`. Текстовый аргумент конструктора становится названием потока, а командой `start()` поток запускается (поэтому создание объекта класса `MyThread` означает создание и запуск потока). В потоке выполняется метод `run()`, в теле которого в свою очередь вызывается статический метод `show()` из главного класса `Demo`.

Статический метод `show()` описан в классе `Demo`, и в его сигнатуре указано ключевое слово `synchronized` (метод синхронизированный). В теле метода в объектную переменную `t` класса `Thread` записывается ссылка на поток, в котором вызывается метод. Такую ссылку получаем, вызвав из класса `Thread` статический метод `currentThread()`. После этого запускается оператор цикла, в котором с интервалом в одну секунду выполняются три итерации. За каждую итерацию отображается название потока (инструкция `t.getName()`) и номер итерации (в квадратных скобках).

В главном методе программы командами `new MyThread("Alpha")` и `new MyThread("Bravo")` создаются два анонимных объекта класса `MyThread`. Это означает, что создаются и запускаются два дочерних потока, в результате чего и появляются сообщения в области вывода.

Чтобы понять, в чем же эффект от синхронизации метода `show()`, имеет смысл внести минимальные изменения в код — убрать из сигнатуры метода ключевое слово `synchronized`. Код метода будет выглядеть так (комментарии удалены):

```
public static void show(){
   Thread t=Thread.currentThread();
   for(int k=1;k<=3;k++){
      System.out.println(t.getName()+" ["+k+"]");
      try{
         Thread.sleep(1000);
      }
      catch(InterruptedException e){}
   }
}
```

Результат выполнения программы теперь будет выглядеть иначе:

Результат выполнения программы (из листинга 10.10)

```
Alpha [1]
Bravo [1]
Alpha [2]
```

```
Bravo [2]
Alpha [3]
Bravo [3]
```

Если метод `show()` не синхронизирован, то он вызывается практически одновременно двумя дочерними потоками. Код метода выполняется в каждом из потоков. Сообщения от каждого из потоков отображаются практически одновременно (но обычно первый поток это делает немного быстрее). А вот если метод синхронизирован, то, как только его вызвал один поток, второй поток не сможет вызвать метод, пока он не завершит выполнение в другом (уже вызвавшем его) потоке. Поэтому сначала отображаются все сообщения от одного потока, а затем начнут отображаться сообщения от другого потока.

Синхронизировать можно и отдельный блок кода (листинг 10.11).

Листинг 10.11. Синхронизированный блок

```
// Класс для создания потока:
class MyThread extends Thread{
   // Конструктор:
   MyThread(String name){
      super(name);
      start();
   }
   // Код для выполнения в потоке:
   public void run(){
      // Синхронизированный блок:
      synchronized(Demo.nums){
         for(int k=0;k<Demo.nums.length;k++){
            // Задержка в выполнении потока:
            try{
               Thread.sleep(1000);
            }
            catch(InterruptedException e){}
            // Отображение сообщений:
            System.out.println(
               getName()+" ["+Demo.nums[k]+"]"
            );
         }
      }
   }
}
// Главный класс:
class Demo{
   // Статическое поле:
   public static int[] nums={1,2,3};
   // Главный метод:
   public static void main(String[] args){
      // Создание и запуск на выполнение потоков:
      new MyThread("Alpha");
      new MyThread("Bravo");
   }
}
```

Программа в плане результатов похожа на предыдущую, но организована иначе. В главном классе `Demo` есть статическое поле `nums`, являющееся ссылкой на целочисленный массив. В главном методе создается два дочерних потока, которые в процессе выполнения отображают сообщения с названием соответствующего потока и значением элемента из массива `nums`. То есть два потока используют один массив. Потоки запускаются один за другим (почти одновременно). В классе `MyThread`, на основе которого создаются потоки, в методе `run()` использован синхронизированный блок. Синхронизируемым объектом является массив `nums`. Поэтому как только один из потоков получает доступ к массиву, для другого потока этот ресурс блокируется. В результате сообщения в области вывода появляются сначала от одного потока, а затем от другого. Например, результат выполнения программы может быть таким:

Результат выполнения программы (из листинга 10.11)

```
Alpha [1]
Alpha [2]
Alpha [3]
Bravo [1]
Bravo [2]
Bravo [3]
```

Но если синхронизированный блок не использовать, то ситуация изменится. Скажем, применим следующий код для метода `run()` (комментарии удалены):

```
public void run(){
   for(int k=0;k<Demo.nums.length;k++){
      try{
         Thread.sleep(1000);
      }
      catch(InterruptedException e){}
      System.out.println(getName()+" ["+Demo.nums[k]+"]");
   }
}
```

В этом случае потоки одновременно получают доступ к массиву `nums` и сообщения отображаются вперемешку.

Результат выполнения программы (из листинга 10.11)

```
Alpha [1]
Bravo [1]
Alpha [2]
Bravo [2]
Alpha [3]
Bravo [3]
```

Понятно, что пример учебный и катастрофы в случае отсутствия синхронизации потоков не будет. Но на практике правильная организация работы потоков может стать сложной задачей. Более того, работа с потоками требует предельной аккуратности и продуманности в реализации алгоритмов и общих подходов.

Резюме

Какое-то загадочное явление природы.

из м/ф «Приключения капитана Врунгеля»

- В Java поддерживается многопоточное программирование, когда несколько частей программы выполняются одновременно.
- Потоки реализуются с использованием класса `Thread` и интерфейса `Runnable`. Для создания потока необходимо либо создать подкласс класса `Thread`, либо описать класс, реализующий интерфейс `Runnable`.
- Код, который должен выполняться в потоке, описывается в методе `run()` класса, реализующего интерфейс `Runnable`. На основе класса создается объект и связывается с объектом класса `Thread`. Для запуска потока необходимо из объекта класса `Thread` вызвать метод `start()`.
- Поскольку класс `Thread` реализует интерфейс `Runnable`, то объект потока можно создать на основе подкласса класса `Thread`, переопределив в этом подклассе нужным образом метод `run()`. Для запуска потока из объекта подкласса вызывается метод `start()`, а в режиме потока выполняется метод `run()` того же объекта.
- Запуск программы означает выполнение главного потока. В главном потоке порождаются все дочерние потоки. Главный поток отождествляется с программой.
- Для управления потоками используются специальные встроенные методы (в том числе и статические методы класса `Thread`).
- В некоторых случаях, когда разные потоки обращаются к общим ресурсам, необходимо выполнять синхронизацию потоков. Синхронизация позволяет разграничить и упорядочить доступ потоков к ресурсам. Можно создавать синхронизированные методы или синхронизированные блоки кода.
- Синхронизированный метод содержит в сигнатуре ключевое слово `synchronized`. Если такой метод вызван в потоке, то пока метод не завершит выполнение, он недоступен для вызова в других потоках.
- Синхронизированный блок также помечается ключевым словом `synchronized`. В круглых скобках после ключевого слова `synchronized` указывается объект. Если этот объект уже используется одним потоком, то для других потоков доступ к объекту блокируется.

11 Обобщенные типы

Видел чудеса техники, но такого!

из к/ф «Иван Васильевич меняет профессию»

В этой главе речь пойдет об *обобщенных типах*. Это достаточно мощный и эффективный инструмент, который позволяет просто и красиво решать сложные задачи. Мы обсудим способы использования обобщенных методов, классов и интерфейсов, а также познакомимся с некоторыми интерфейсами и классами для создания стандартных *коллекций* (группы объектов, которые можно обрабатывать с помощью специальных методов).

Обобщенные методы

Вы еще ответите за ваши антиобщественные опыты!

из к/ф «Иван Васильевич меняет профессию»

Представим себе следующую ситуацию: необходимо описать метод, который отображает содержимое массива, переданного аргументом методу. Что особенного в такой задаче? Главная ее особенность в том, что алгоритм, который предстоит реализовать в методе, не зависит от типа массива. Другими словами, для текстового, символьного или целочисленного массивов код будет отличаться только идентификатором типа в описании аргумента метода. Чтобы метод был достаточно универсальным и мог использоваться с массивами разных типов, следует описать несколько однотипных версий метода. Но есть и другой подход, в соответствии с которым в описании метода тип не конкретизируется, а указывается в виде параметра. Такие методы называются *обобщенными*. Параметр, который обозначает тип данных, называется *обобщенным параметром*. При вызове метода фактический тип, который нужно использовать вместо обобщенного параметра, определяется на основании типа аргумента метода.

Описывается обобщенный метод практически так же, как обычный метод, но перед идентификатором типа результата метода в угловых скобках указывается формальное обозначение для обобщенного параметра. Этот обобщенный параметр и используется в описании метода в качестве идентификатора типа. Ниже приведен шаблон описания обобщенного метода:

```
<параметры> тип название(аргументы){
   // Команды
}
```

Если в методе используется несколько обобщенных параметров, то в угловых скобках они перечисляются через запятую (листинг 11.1).

Листинг 11.1. Обобщенные статические методы

```
// Класс:
class MyClass{
   // Поле:
   int code;
   // Конструктор:
   MyClass(int code){
      this.code=code;
   }
   // Переопределение метода toString():
   public String toString(){
      return "MyClass "+code;
   }
}
// Главный класс:
class Demo{
   // Обобщенный метод с одним обобщенным параметром:
   static <T> void show(T[] a){
      for(int k=0;k<a.length;k++){
         System.out.print("|"+a[k]);
      }
      System.out.println("|");
   }
   // Обобщенный метод с одним обобщенным параметром:
   static <T> String getText(T a,int n){
      String res=n+": ";
      res+=a;
      return res;
   }
   // Обобщенный метод с двумя обобщенными параметрами:
   static <T,U> T getArg(T x,T y,U z){
      int val=z.toString().length();
      if(val%2==0) return x;
      else return y;
   }
   // Главный метод:
   public static void main(String[] args){
      // Массивы:
```

```
      Integer[] nums={1,3,5,7,9,11,13};
      Character[] symbs={'A','B','C','D'};
      String[] txt={"Alpha","Bravo","Charlie"};
      // Вызов обобщенных методов:
      show(nums);
      show(symbs);
      show(txt);
      System.out.println(getText('A',1));
      System.out.println(getText("Alpha",2));
      System.out.println(getText(100,3));
      // Создание объекта:
      MyClass obj=new MyClass(200);
      // Вызов обобщенных методов:
      System.out.println(getText(obj,4));
      String A=getArg("Alpha","Bravo",obj);
      MyClass B=getArg(
         new MyClass(300),new MyClass(400),1234
      );
      Integer C=getArg(123,321,"Hello");
      // Отображение результата:
      System.out.println("A: "+A);
      System.out.println("B: "+B);
      System.out.println("C: "+C);
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 11.1)

```
|1|3|5|7|9|11|13|
|A|B|C|D|
|Alpha|Bravo|Charlie|
1: A
2: Alpha
3: 100
4: MyClass 200
A: Bravo
B: MyClass 300
C: 321
```

В программе (в ее главном классе) описывается несколько статических методов. В описании метода `show()` использована инструкция `<T>`, означающая, что через `T` в коде метода обозначен некоторый тип данных. Описание аргумента метода аналогично описанию для типа `T[]`. Таким образом, аргумент метода — это массив с элементами некоторого типа (обозначенного как `T`). Метод предназначен для отображения значений элементов массива. Что происходит при вызове метода? Например, в главном методе объявляются три массива: целочисленный `nums`, символьный `symbs` и текстовый `txt`. Затем метод `show()` вызывается с разными аргументами. Если методу аргументом передается массив `nums`, то код метода выполняется и вместо параметра `T` используется класс `Integer`. Если метод вызывается с аргументом

symbs, то вместо параметра T используется класс Character, а при вызове метода с аргументом txt вместо параметра T используется класс String.

ПОДРОБНОСТИ

Мы не случайно вместо базовых типов использовали классы-оболочки Integer и Character, а также класс String. Дело в том, что в качестве значений для обобщенных параметров могут использоваться только ссылочные типы (то есть классы).

У метода getText() два аргумента: первый относится к обобщенному типу T, а второй аргумент — целочисленный (тип int). Результатом метод возвращает текстовое значение. В теле метода формируется текстовая строка, которая состоит из значения второго аргумента, двоеточия с пробелом и текстового представления для первого аргумента. Эта строка возвращается как результат метода.

ПОДРОБНОСТИ

В теле метода к текстовой строке прибавляется первый аргумент (тип которого определяется обобщенным параметром T). Мы не знаем, какой тип будет использован вместо обобщенного параметра, но в любом случае это будет ссылочный тип (класс). Для соответствующего объекта автоматически будет вызван метод toString(), и результат этого метода используется вместо объекта в операции формирования текстовой строки.

В главном методе есть несколько вариантов вызова метода getText(). При выполнении инструкции getText('A',1) вместо обобщенного параметра используется тип Character в соответствии с типом первого аргумента. Хотя формально первым аргументом указан литерал типа char, выполняется автоматическое преобразование к ссылочному типу Character. То же относится и к инструкции getText(100,3), при выполнении которой для обобщенного параметра используется значение Integer, образуемое в результате автоматического преобразования из базового типа int. При вызове метода инструкцией getText("Alpha",2) вместо обобщенного параметра используется класс String (тип первого аргумента). Также создается объект obj класса MyClass, используемый в инструкции getText(obj,4). Поэтому значением обобщенного параметра является MyClass. При формировании текстовой строки (результата метода) для объекта obj вызывается метод toString(), который возвращает текстовую строку с названием класса MyClass и значением 200 поля code этого объекта.

У статического обобщенного метода getArg() два обобщенных параметра (обозначены как T и U) и, соответственно, в описании этого метода использована инструкция <T,U>. У метода три аргумента: первые два (x и y) относятся к обобщенному типу T, а третий z — к обобщенному типу U. В качестве результата метод возвращает значение, которое относится к обобщенному типу T. Таким образом, тип результата совпадает с типом первых двух аргументов.

В теле метода третий аргумент z с помощью метода toString() явно приводится к текстовому формату, и из этого текстового значения вызывается метод length() (инструкция z.toString().length()). Результат записывается в целочисленную переменную val. Далее в условном операторе проверяется условие val%2==0, и если оно истинно, то результатом возвращается первый аргумент метода. При ложном условии результатом возвращается второй аргумент метода.

НА ЗАМЕТКУ

Резюме такое: если в текстовом представлении для третьего аргумента метода getArg() количество символов четное, то метод возвращает ссылку на свой первый аргумент. Если количество символов нечетное, то возвращается ссылка на второй аргумент.

При вызове метода мы можем передавать ему практически любые аргументы — главное, чтобы тип первого и второго аргументов совпадал. Например, результатом выражения getArg("Alpha","Bravo",obj) является значение "Bravo", поскольку в текстовом представлении для объекта obj (это текст "MyClass 200") нечетное количество символов. Первые два аргумента метода относятся к классу String, поэтому для обобщенного параметра T используется значение String. Соответственно, в качестве результата метод возвращает значение класса String. Третий аргумент относится к классу MyClass, поэтому значением параметра U является MyClass.

При вычислении выражения getArg(new MyClass(300),new MyClass(400),1234) возвращается ссылка на первый анонимный объект (со значением поля 300), поскольку в текстовом представлении "1234" четное количество символов. В качестве значения параметра T используется класс MyClass, а в качестве параметра U — класс Integer.

Значением выражения getArg(123,321, "Hello") является число 321, поскольку в тексте "Hello" нечетное количество символов. В качестве параметра T используется класс Integer, а в качестве параметра U — класс String.

Как отмечалось, значения для обобщенных параметров при вызове метода определяются по типу переданных методу аргументов. Вместе с тем существует возможность определять значение обобщенных параметров в явном виде. В таком случае в инструкции вызова обобщенного метода перед именем метода в угловых скобках указываются значения для обобщенных параметров (листинг 11.2).

Листинг 11.2. Явное определение типа

```
// Класс со статическим обобщенным методом:
class MyClass{
   // Обобщенный метод:
   static <T> void show(T t){
      System.out.println("Значение: "+t);
   }
}
```

```
// Главный класс:
class Demo{
   public static void main(String[] args){
      MyClass.show(123);
      MyClass.<Integer>show(123);
      MyClass.show(321.0);
      MyClass.<Double>show(321.0);
      MyClass.show('A');
      MyClass.<Character>show('A');
   }
}
```

Результат выполнения программы таков:

Результат выполнения программы (из листинга 11.2)

```
Значение: 123
Значение: 123
Значение: 321.0
Значение: 321.0
Значение: A
Значение: A
```

В программе объявляется класс `MyClass` со статическим обобщенным методом `show()`. При вызове метода отображается значение его аргумента. В главном методе программы есть примеры вызова метода с разными аргументами, причем в одних случаях значение для обобщенного параметра не указывается, а в других — указывается в явном виде. На результат это не влияет.

Обобщенный метод не обязательно должен быть статическим. Другими словами, можно описать обобщенный метод и в обычном классе (листинг 11.3).

Листинг 11.3. Нестатические обобщенные методы

```
// Обычный класс:
class Alpha{
   // Поле:
   int code;
   // Конструктор:
   Alpha(int code){
      this.code=code;
   }
   // Переопределение метода toString():
   public String toString(){
      return "<Alpha "+code+">";
   }
}
// Класс с обобщенными методами:
class MyClass{
   // Текстовое поле:
   String name;
   // Обобщенные методы:
```

```
   <T> void set(T obj){
      name=obj.toString();
   }
   <T> T get(T[] objs,char symb){
      // Индекс символа в тексте:
      int index=name.indexOf(symb);
      // Уточнение значения индекса:
      if(index<0) index=0;
      if(index>=objs.length) index=objs.length-1;
      // Результат метода:
      return objs[index];
   }
   // Переопределение метода toString():
   public String toString(){
      return "{MyClass "+name+"}";
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объектов:
      Alpha A=new Alpha(100);
      MyClass obj=new MyClass();
      // Вызов обобщенного метода:
      obj.set(A);
      System.out.println(obj);
      obj.set(obj);
      System.out.println(obj);
      obj.set(200);
      System.out.println(obj);
      obj.set("объект");
      System.out.println(obj);
      // Массивы:
      Integer[] nums={1,2,3,4,5};
      Alpha[] objs=new Alpha[5];
      for(int k=0;k<objs.length;k++){
         objs[k]=new Alpha((k+1)*10);
      }
      String[] txt={"один","два","три","четыре","пять"};
      Character[] symbs={'A','B','C','D','E'};
      // Вызов обобщенного метода:
      Integer n=obj.get(nums,'м');
      Alpha a=obj.get(objs,'6');
      String t=obj.get(txt,'e');
      Character s=obj.get(symbs,'т');
      // Проверка результата:
      System.out.println("Integer: "+n);
      System.out.println("Alpha: "+a);
      System.out.println("String: "+t);
      System.out.println("Character: "+s);
   }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 11.3)

```
{MyClass <Alpha 100>}
{MyClass {MyClass <Alpha 100>}}
{MyClass 200}
{MyClass объект}
Integer: 1
Alpha: <Alpha 20>
String: четыре
Character: E
```

В программе описывается обычный класс `Alpha` с целочисленным полем `code`, конструктором (с одним аргументом) и переопределенным методом `toString()`.

Класс `MyClass` содержит текстовое поле `name`. В классе описан обобщенный метод `set()` с одним аргументом обобщенного типа. В теле метода командой `name=obj.toString()` полю `name` значением присваивается текстовое представление для объекта `obj`, переданного аргументом методу.

Еще один обобщенный метод `get()`, описанный в классе `MyClass`, в качестве аргументов получает массив `objs` с элементами обобщенного типа и символьное значение. Результатом метод возвращает ссылку на элемент массива. Индекс элемента определяется индексом символа (второй аргумент) в текстовом значении, на которое ссылается поле `name`. Индекс вычисляется с помощью выражения `name.indexOf(symb)`. При этом следует учесть, что если символа в тексте нет, то метод `indexOf()` возвращает значение `-1`. Также индекс может выходить за пределы массива, переданного первым аргументом методу. Поэтому с помощью условного оператора мы уточняем значение индекса. Кроме этого, в классе `MyClass` переопределяется метод `toString()`.

В главном методе программы создается объект `A` класса `Alpha` и объект `obj` класса `MyClass`. После этого из объекта `obj` вызывается метод `set()` с разными аргументами (в том числе и с самим объектом `obj` в качестве аргумента). Каждый раз проверяется значение поля `name` объекта `obj` (путем неявного вызова метода `toString()` при попытке напечатать объект `obj`).

Кроме этого, мы создаем несколько массивов и используем их при вызове из объекта `obj` обобщенного метода `get()`.

Обобщенным может быть не только нестатический метод, но и конструктор класса. В таком конструкторе в описании перед именем конструктора указывается в угловых скобках обозначение для обобщенного параметра (или параметров, листинг 11.4).

Листинг 11.4. Обобщенный конструктор

```
// Обычный класс:
class Alpha{
```

```
   int code;
   Alpha(int n){
      code=n;
   }
   public String toString(){
      return "<Alpha "+code+">";
   }
}
// Класс с обобщенным конструктором:
class MyClass{
   String name;
   // Обобщенный конструктор:
   <T> MyClass(T t){
      name=t.toString();
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объектов:
      MyClass A=new MyClass(123);
      System.out.println(A.name);
      MyClass B=new MyClass("Java");
      System.out.println(B.name);
      MyClass C=new MyClass(new Alpha(300));
      System.out.println(C.name);
   }
}
```

Ниже показано, как выглядит результат выполнения программы:

Результат выполнения программы (из листинга 11.4)

```
123
Java
<Alpha 300>
```

Мы описываем обычный класс `Alpha` с целочисленным полем, конструктором и переопределенным методом `toString()`. Еще один класс `MyClass` имеет текстовое поле `name` и обобщенный конструктор (с одним обобщенным параметром, который обозначен как `T`). Обобщенный параметр объявляется в угловых скобках перед именем конструктора. Конструктору класса `MyClass` передается один аргумент обобщенного типа. В теле конструктора из этого аргумента вызывается метод `toString()`, а полученный в результате текст записывается в поле `name`.

Положительный эффект от использования обобщенного конструктора в данном случае в том, что аргументом такому конструктору можно передавать практически все что угодно. Подтверждением тому являются команды, которыми в главном методе создаются объекты класса `MyClass`.

Перегрузка обобщенных методов

— Бабушка, ты гений!

— Нет, у меня есть жизненный опыт.

из к/ф «Гостья из будущего»

Обобщенные методы, как и обычные, можно перегружать. Причем для обобщенных методов можно создавать как несколько обобщенных версий, так и обычные (не обобщенные) версии. В качестве примера мы рассмотрим процесс перегрузки статического метода, хотя с нестатическими методами можно проделать аналогичные операции (листинг 11.5).

Листинг 11.5. Перегрузка обобщенных методов

```
class Demo{
   // Обобщенный метод с одним параметром:
   static <T> void show(T t){
      System.out.println("[1] "+t);
   }
   // Метод с текстовым аргументом:
   static void show(String s){
      System.out.println("[2] Текст "+s);
   }
   // Обобщенный метод с двумя параметрами:
   static <T,U> void show(T t,U u){
      System.out.println("[3] "+t+" и "+u);
   }
   // Обобщенный метод с одним параметром
   // и двумя аргументами:
   static <T> void show(T t,int n){
      System.out.println("[4] "+t+" и число "+n);
   }
   // Метод с двумя аргументами (число и символ):
   static void show(int n,char s){
      System.out.println("[5] Число "+n+" и символ "+s);
   }
   // Главный метод:
   public static void main(String[] args){
      show(100);
      show('A');
      show("Alpha");
      show("Bravo","Charlie");
      show('B',200.0);
      show("Delta",300);
      show(400,'C');
      show(new Character('D'),new Integer(500));
      Demo.<Character,Integer>show('D',500);
      Demo.<Character>show(‘D',500);
   }
}
```

Результат выполнения программы таков:

Результат выполнения программы (из листинга 11.5)

```
[1] 100
[1] A
[2] Текст Alpha
[3] Bravo и Charlie
[3] B и 200.0
[4] Delta и число 300
[5] Число 400 и символ C
[3] D и 500
[3] D и 500
[4] D и число 500
```

Мы описываем пять версий статического метода `show()`. Некоторые из этих версий обобщенные, некоторые — нет. Есть версии с одним аргументом и с двумя аргументами. В любом случае, при вызове метода отображаются значения его аргументов. Для удобства в самом начале метод отображает свой номер. В нашем распоряжении несколько версий, в том числе:

- версия метода с одним аргументом обобщенного типа;
- версия метода с одним текстовым аргументом;
- версия метода с двумя аргументами обобщенных типов;
- версия метода с двумя аргументами (первый — обобщенного типа, второй — целочисленный);
- версия метода с двумя аргументами (первый — символьного типа, второй — целочисленного типа).

При вызове метода его версия определяется по количеству и типу аргументов, переданных методу. Если точного совпадения нет, то используется автоматическое приведение типов (там, где оно допустимо). Например, при выполнении команды `show(100)` методу `show()` передается один аргумент типа `int`. Такая версия метода в программе не описана. Но есть версия метода с одним аргументом обобщенного типа. Поэтому значение типа `int` автоматически преобразуется в объект класса `Integer` и вызывается версия метода с одним аргументом обобщенного типа, для которого используется значение `Integer`. Аналогично выполняется команда `show('A')`, только теперь тип `char` преобразуется в `Character`. А вот при выполнении команды `show("Alpha")` все происходит иначе. Формально здесь тоже подошла бы версия метода с одним аргументом обобщенного типа, но у нас есть и версия метода с текстовым аргументом. Именно она и используется.

При выполнении команды `show("Bravo","Charlie")` будет задействована версия метода с двумя аргументами обобщенных типов (каждый из них в данном случае будет соответствовать классу `String`). Эта же версия метода вызывается при вы-

полнении команды `show('B',200.0)`, но только в этом случае сначала выполняется преобразование значений типов `char` и `double` в объекты классов `Character` и `Double` соответственно.

Команда `show("Delta",300)` полностью, без всякого приведения типа, соответствует версии метода с первым аргументом обобщенного типа (со значением `String`) и вторым аргументом типа `int`.

В команде `show(400,'C')` методу передаются аргументы типов `int` и `char`, и такая версия метода в программе описана. Но вот если бы мы захотели вызвать метод `show()` с первым аргументом типа `char` и вторым аргументом типа `int`, то получили бы неоднозначную ситуацию (и, как следствие, ошибку при компиляции), поскольку именно такой версии метода (с аргументами типа `char` и `int`) в программе нет. И тут возможны два эквивалентных варианта: или преобразовать тип `char` в `Character` и использовать версию метода с первым аргументом обобщенного типа и вторым аргументом типа `int`, или преобразовать в тип `Integer` еще и тип `int` и вызвать версию метода с двумя аргументами обобщенного типа. Такие неоднозначные ситуации интерпретируются как ошибочные.

Решать проблему можно разными способами. В данном случае мы можем аргументами передать не значения базовых типов, а объекты, созданные на основе соответствующих значений. Примером является команда `show(new Character('D'),new Integer(500))`, при выполнении которой вызывается метод с двумя аргументами обобщенного типа (со значениями `Character` и `Integer`). Еще один вариант — явно указать версию обобщенного метода.

В команде `Demo.<Character,Integer>show('D',500)` мы указали, что следует использовать версию обобщенного статического метода `show()` из класса `Demo` со значениями обобщенных параметров `Character` и `Integer`. В соответствии с инструкцией `Demo.<Character>show('D',500)` используется версия обобщенного статического метода `show()` из класса `Demo` со значением обобщенного параметра `Character` (второй аргумент относится к типу `int`).

Обобщенные классы

Альфу Центавра знаешь? Тамошние мы.

из к/ф «Гостья из будущего»

Можно создавать не только обобщенные методы, но и *обобщенные классы*. В обобщенном классе тип данных является параметром. В описании обобщенного класса названия для обобщенных параметров объявляются в угловых скобках сразу после имени класса. В теле класса эти параметры используются как идентификаторы типа. Шаблон описания обобщенного класса выглядит следующим образом:

```
class имя<параметры>{
   // Описание класса
}
```

При создании объекта на основе обобщенного класса значения для обобщенных параметров указываются в угловых скобках после имени класса. Это делается в инструкции объявления объектной переменной. Допускается указать значения для параметров еще и непосредственно в инструкции создания объекта:

```
Класс<значения> переменная=new Класс<значения>(аргументы);
```

Но можно сэкономить на программном коде и в команде создания объекта значения для обобщенных параметров не указывать, при этом после имени класса все равно должны присутствовать пустые угловые скобки:

```
Класс<значения> переменная=new Класс<>(аргументы);
```

Пример, в котором используются обобщенные классы, представлен в листинге 11.6.

Листинг 11.6. Знакомство с обобщенными классами

```
// Обобщенный класс с одним параметром:
class Alpha<T>{
   // Поле обобщенного типа:
   T value;
   // Конструктор:
   Alpha(T val){
      value=val;
   }
   // Метод:
   void show(){
      System.out.println("<Alpha "+value+">");
   }
}
// Обобщенный класс с двумя параметрами:
class Bravo<T,U>{
   // Поля обобщенного типа:
   T first;
   U second;
   // Конструктор:
   Bravo(T a,U b){
      first=a;
      second=b;
   }
   // Метод:
   void show(){
      System.out.println("<Bravo "+first+"|"+second+">");
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
```

```
      // Создание объектов на основе
      // обобщенных классов:
      Alpha<Integer> A1=new Alpha<Integer>(123);
      Alpha<String> A2=new Alpha<>("Green");
      Bravo<Integer,String> B1;
      B1=new Bravo<Integer,String>(321,"Blue");
      Bravo<Character,Character> B2=new Bravo<>('B','C');
      // Проверка результата:
      A1.show();
      A2.show();
      B1.show();
      B2.show();
   }
}
```

В результате выполнения программы получаем следующее:

Результат выполнения программы (из листинга 11.6)

```
<Alpha 123>
<Alpha Green>
<Bravo 321|Blue>
<Bravo B|C>
```

Мы описываем два похожих обобщенных класса — `Alpha` и `Bravo`. В классе `Alpha` один обобщенный параметр (обозначен как `T`). У класса одно поле `value` обобщенного типа. В классе есть конструктор (с одним аргументом обобщенного типа, который определяет значение поля `value`), а также метод `show()` без аргументов, которым отображается значение поля `value`.

В классе `Bravo` имеются два обобщенных параметра (обозначены как `T` и `U`). У класса два поля обобщенного типа, конструктор с двумя аргументами обобщенного типа и метод `show()`, которым отображаются значения полей.

В главном методе программы создаются объекты данных классов: объявляется объектная переменная и создается соответствующий объект. Все это похоже на создание объектов на основе обычных классов. Правда, имеются и некоторые особенности (листинг 11.7).

Листинг 11.7. Особенности обобщенных классов

```
// Обобщенный класс:
class MyClass<T>{
   // Поле обобщенного типа:
   T value;
   // Метод:
   void show(){
      System.out.println("Выполняется проверка");
      System.out.println("Значение поля: "+value);
      System.out.println("Тип поля: "+
         value.getClass().getSimpleName());
   }
```

```
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объекта:
      MyClass<Integer> A=new MyClass<>();
      MyClass B=A;
      // Выполнение операций:
      A.value=100;
      A.show();
      B.show();
      A.value++;
      A.show();
      B.value="Это текст!";
      B.show();
      A.show();
   }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 11.7)

```
Выполняется проверка
Значение поля: 100
Тип поля: Integer
Выполняется проверка
Значение поля: 100
Тип поля: Integer
Выполняется проверка
Значение поля: 101
Тип поля: Integer
Выполняется проверка
Значение поля: Это текст!
Тип поля: String
Выполняется проверка
Значение поля: Это текст!
Тип поля: String
```

Здесь описан обобщенный класс `MyClass` с обобщенным параметром `T`. В классе есть поле `value` обобщенного типа. Также имеется метод `show()`, при вызове которого отображается значение поля и название класса объекта, на который ссылается поле `value`. В последнем случае мы вызываем через поле `value` метод `getClass()`, а затем из полученного выражения вызывается метод `getSimpleName()` (вся инструкция выглядит как `value.getClass().getSimpleName()`).

ПОДРОБНОСТИ

Метод `getClass()` в качестве результата возвращает ссылку на объект класса `Class`. Он содержит информацию о классе объекта, из которого вызывался метод. У объекта, который возвращается методом `getClass()`, есть метод

`getSimpleName()`, возвращающий текстовое значение с названием класса. Поэтому результатом выражения `value.getClass().getSimpleName()` является название класса объекта, на который ссылается переменная `value`.

В главном методе программы командой `MyClass<Integer> A=new MyClass<>()` на основе обобщенного класса `MyClass` создается объект `A`, причем вместо обобщенного параметра используется класс `Integer`. Поэтому полю `value` объекта `A` можно присвоить целочисленное значение, как, например, это делается командой `A.value=100`. Кроме того, выполняется команда `MyClass B=A`, которой переменной `B` в качестве значения присваивается ссылка на тот же объект, на который ссылается переменная `A`. Особенность команды в том, что класс `MyClass` (определяет тип переменной `B`) указан без спецификации значения для обобщенного параметра. И это будет иметь последствия.

Таким образом, переменные `A` и `B` ссылаются на один и тот же объект. Вызывая метод `show()` через каждую из этих переменных, получаем ожидаемый результат (команды `A.show()` и `B.show()`). Командой `A.value++` значение поля `value` увеличивается на единицу.

НА ЗАМЕТКУ

Стоит отметить, что через переменную `B` мы операцию, аналогичную `A.value++`, проделать не смогли бы.

Сюрпризом, скорее всего, будет команда `B.value="Это текст!"`, которая присваивает полю `value` текстовое значение. Проверка результата с помощью команд `B.show()` и `A.show()` подтверждает, что в поле записано текстовое значение (точнее, поле `value` содержит ссылку на текст).

НА ЗАМЕТКУ

Если мы теперь попытаемся выполнить команду `A.value++`, то на этапе выполнения программы произойдет ошибка.

Следует также учесть особенности создания обобщенных классов. В Java используется так называемый механизм *стирания типов*. Суть его в том, что информация о значении обобщенных параметров, используемых при создании объекта, в самом объекте неизвестна. Эта информация содержится в переменной, которая ссылается на объект. Переменная `A`, тип которой указан как `MyClass<Integer>`, знает, что объект, на который она ссылается, содержит целочисленное поле `value`. При доступе к объекту через переменную `A` значение, на которое ссылается поле `value`, обрабатывается как целое число. Переменная `B` объявлялась как относящаяся к классу `MyClass`, без указания значения обобщенного параметра. Переменная `B` ничего не знает о типе поля `value` (но знает, что такое поле существует). Переменная `B` интерпретирует поле `value` как относящееся к классу `Object` (поскольку переменная класса `Object`

может ссылаться на любое значение). Через переменную `B` удалось присвоить полю `value` ссылку на текстовое значение. А поскольку в объекте, в котором содержится поле `value`, информация о типе поля отсутствует, то такая манипуляция проходит. При этом переменная `A` продолжает интерпретировать поле `value` как целочисленное.

ПОДРОБНОСТИ

Стоит особо подчеркнуть, что поле `value` в любом случае не содержит значение (число или текст), а ссылается на него. Значением этого поля является адрес объекта. Сначала это объект класса `Integer`, а затем объект класса `String`. Метод `getClass()` возвращает объект, который содержит информацию о типе не поля `value`, а объекта, на который это поле ссылается.

В известном смысле все это выглядит экзотично, но имеет важные последствия, о которых необходимо знать. Например, мы не можем в обобщенном классе создать объект обобщенного типа или создать массив из элементов обобщенного типа. И откровенно говоря, это серьезно ограничивает возможности по использованию обобщенных типов.

Обобщенные интерфейсы

Он плод генетической ошибки.

из к/ф «Гостья из будущего»

Обобщенными могут быть не только классы и методы, но и интерфейсы. Причем на основе *обобщенного интерфейса* можно создавать как обычные классы, так и обобщенные (листинг 11.8).

Листинг 11.8. Обобщенный интерфейс

```
// Обобщенный интерфейс:
interface MyInterface<T>{
   void show(T t);
}
// Обобщенный класс:
class Alpha<T,U> implements MyInterface<T>{
   U value;
   public void show(T t){
      System.out.println(t+": "+value);
   }
}
// Обычный класс:
class Bravo implements MyInterface<String>{
```

```
   public void show(String t){
      System.out.println("Текст: "+t);
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      Alpha<Character,String> A=new Alpha<>();
      A.value="Alpha";
      A.show('A');
      Bravo B=new Bravo();
      B.show("Bravo");
   }
}
```

Результат выполнения программы таков:

Результат выполнения программы (из листинга 11.8)

```
A: Alpha
Текст: Bravo
```

Как видим, имеется обобщенный интерфейс `MyInterface` с обобщенным параметром `T`. В интерфейсе объявлен метод `show()` с аргументом обобщенного типа. Этот интерфейс реализуется в классах `Alpha` и `Bravo`. Класс `Alpha` является обобщенным с двумя параметрами — `T` и `U`. Интерфейс `MyInterface` реализуется с параметром `T`. В классе есть поле `value` обобщенного типа `U`. Метод `show()` получает в качестве аргумента значение обобщенного типа `T`. Метод при вызове отображает сообщение, которое содержит текстовое представление для аргумента метода и поля `value`.

Класс `Bravo` создается реализацией интерфейса `MyInterface` со значением `String` для обобщенного параметра. Метод `show()` с текстовым аргументом определен так, что отображается сообщение со значением аргумента.

В главном методе программы на основе классов `Alpha` и `Bravo` создаются объекты, из которых вызывается метод `show()`. Последствия этих операций должны быть понятны читателю.

Обобщенные классы и наследование

Приземленная ты субстанция.

из к/ф «Гостья из будущего»

Обобщенные классы могут использоваться в наследовании. Как и в случае с интерфейсами, на основе обобщенного класса можно создавать как обычный, так и обобщенный класс (листинг 11.9).

Листинг 11.9. Обобщенные классы и наследование

```
// Обобщенный суперкласс:
class MyClass<T,U>{
   // Поля:
   T value;
   U code;
   // Методы:
   void set(T a,U b){
      value=a;
      code=b;
   }
   void show(){
      System.out.println("[1] "+value);
      System.out.println("[2] "+code);
   }
}
// Обобщенный подкласс:
class Alpha<T> extends MyClass<T,String>{}
// Обычный подкласс:
class Bravo extends MyClass<Character,Integer>{}
// Главный класс:
class Demo{
   public static void main(String[] args){
      MyClass<String,Character> obj=new MyClass<>();
      obj.set("MyClass",'D');
      obj.show();
      Alpha<Double> A=new Alpha<>();
      A.set(123.0,"Alpha");
      A.show();
      Bravo B=new Bravo();
      B.set('B',321);
      B.show();
   }
}
```

Ниже показано, как выглядит результат выполнения программы:

Результат выполнения программы (из листинга 11.9)

```
[1] MyClass
[2] D
[1] 123.0
[2] Alpha
[1] B
[2] 321
```

Здесь описан обобщенный класс `MyClass` с двумя обобщенными параметрами — `T` и `U`. У класса есть два поля обобщенных типов, метод `set()`, предназначенный для присваивания значений полям, а также метод `show()` без аргументов, при вызове которого отображаются значения (текстовые представления) для полей объекта.

Два подкласса (`Alpha` и `Bravo`) создаются на основе обобщенного класса `MyClass`. Класс `Alpha` наследует класс `MyClass` со значением `String` для второго обобщенного параметра. Таким образом, класс `Alpha` является обобщенным с одним обобщенным параметром. Класс `Bravo` наследует класс `MyClass` со значениями `Character` и `Integer` для обобщенных параметров, то есть является обычным (не обобщенным). Никаких дополнительных полей или методов в классах `Alpha` и `Bravo` не объявляется, поэтому у обоих классов пустое тело.

В главном методе программы на основе каждого из классов создаются объекты и из этих объектов вызываются методы `set()` и `show()`.

Ограничения на обобщенные параметры

Что ей надо, я тебе потом скажу.

из к/ф «Бриллиантовая рука»

В некоторых случаях необходимо указать, что значением обобщенного параметра может быть не любой класс, а только подкласс (прямой или через цепочку наследования) определенного суперкласса. В таком случае при объявлении обобщенного параметра после его названия указывается ключевое слово `extends` и название суперкласса (листинг 11.10).

Листинг 11.10. Ограничение на обобщенные параметры

```
// Обычные классы:
class Alpha{
   String name;
   Alpha(String n){
      name=n;
   }
   void show(){
      System.out.println("[1] "+name);
   }
}
class Bravo extends Alpha{
   Bravo(String n){
      super(n);
   }
   void show(){
      System.out.println("[2] "+name);
   }
}
class Charlie extends Bravo{
   Charlie(String n){
      super(n);
   }
   void show(){
```

```
      System.out.println("[3] "+name);
   }
}
// Обобщенные классы:
class First<T extends U,U>{
   T code;
   U value;
   void set(T a,U b){
      code=a;
      value=b;
   }
   U get(boolean t){
      if(t) return code;
      else return value;
   }
}
class Second<T extends Alpha>{
   T obj;
   Second(T t){
      obj=t;
   }
   void display(){
      obj.show();
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      First<Bravo,Alpha> A=new First<>();
      A.set(new Bravo("Bravo"),new Alpha("Alpha"));
      A.get(false).show();
      A.get(true).show();
      Second<Charlie> B=new Second<>(
         new Charlie("Charlie")
      );
      B.display();
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 11.10)

```
[1] Alpha
[2] Bravo
[3] Charlie
```

Здесь, во-первых, мы создаем цепочку наследования, в которую входят обычные (не обобщенные) классы `Alpha`, `Bravo` и `Charlie`. У класса `Alpha` есть текстовое поле `name`, конструктор с одним текстовым аргументом и метод `show()`, который при вызове отображает текст `"[1] "` и название поля `name`. В классе `Bravo` метод `show()` пере-

определяется так, что он отображает текст "[2] " и значение поля name. В классе Charlie метод show() отображает текст "[3] " и значение поля name. Таким образом, по результату вызова метода show() легко определить класс, в котором описана соответствующая версия метода.

Во-вторых, мы создаем два обобщенных метода. В каждом из них используется ограничение для обобщенного параметра. В классе First обобщенные параметры объявляются инструкцией <T extends U,U>. Она означает, что у обобщенного класса два параметра (T и U), причем первый параметр T является подклассом (напрямую или через цепочку наследования) второго параметра U. У класса два поля обобщенного типа (code и value). Метод set() предназначен для присваивания значений полям. Еще у класса есть метод get() с логическим аргументом. Задекларировано, что результатом метод возвращает значение обобщенного типа U. Но по факту если аргумент равен false, то метод возвращает ссылку на поле value типа U. Если же аргумент равен true, то метод возвращает ссылку на поле code типа T. Однако поскольку T является подклассом U, то предлагаемый вариант действий допустим (переменная суперкласса может ссылаться на объект подкласса).

ПОДРОБНОСТИ

Ранее отмечалось, что в обобщенном классе мы не можем создать объект на основе обобщенного типа. Методом set() присваиваются значения полям code и value обобщенного типа. Но объекты обобщенного типа в данном случае не создаются. Они должны быть созданы вне обобщенного класса, а аргументами методу set() передаются только ссылки на эти объекты (и затем эти ссылки присваиваются в качестве значений полям code и value).

В классе Second — один обобщенный параметр T, объявленный с помощью инструкции <T extends Alpha>. Она означает, что вместо параметра может быть использован класс, который является подклассом класса Alpha (то есть это может быть класс Alpha, Bravo или Charlie). В классе есть поле obj обобщенного типа, конструктор с одним аргументом (определяет значение поля) и метод display(), при вызове которого вызывается метод show() из поля obj. Здесь учтено, что тип T является подклассом класса Alpha, а в классе Alpha есть метод show(). Поэтому у объекта, на который ссылается поле obj, тоже есть метод show().

В главном методе на основе обобщенных классов создаются объекты. Объект A создается на основе обобщенного класса First, а в качестве значений для обобщенных параметров использованы классы Bravo и Alpha (класс Bravo является подклассом класса Alpha). При вызове метода set() из объекта A аргументами передаются ссылки на анонимные объекты классов Bravo и Alpha (создаются инструкциями new Bravo("Bravo") и new Alpha("Alpha") соответственно). Эти ссылки записываются в поля code и value объекта A. При вызове метода get() с аргументом false из объекта A получаем ссылку на поле value, ссылающееся

на объект класса `Alpha` (анонимный объект, переданный вторым аргументом методу `set()`). Если из этого объекта (ссылки, полученной в результате вызова метода `get()`) вызвать метод `show()`, то это будет версия метода `show()` из класса `Alpha`. Если передать аргументом методу `get()` значение `true`, то метод `show()` будет вызываться из объекта класса `Bravo` (анонимный объект, переданный первым аргументом методу `set()`).

Объект `B` создается на основе обобщенного класса `Second`, и в качестве значения для обобщенного параметра используется класс `Charlie`. Это законно, поскольку класс `Charlie` является подклассом (через цепочку наследования) класса `Alpha`. Соответственно, аргументом конструктору класса `Second` передается анонимный объект класса `Charlie` (создается инструкцией `new Charlie("Charlie")`). Ссылка на этот объект записывается в поле `obj` объекта `B`. Поэтому при выполнении команды `B.display()` на самом деле из анонимного объекта класса `Charlie` вызывается метод `show()`.

Знакомство с коллекциями

> Ничего особенного. Обыкновенная контрабанда.
>
> *из к/ф «Бриллиантовая рука»*

Обобщенные типы широко используются при работе с *коллекциями*. Под коллекциями подразумевают организованные специальным образом группы объектов. Сюда же обычно относят обобщенные классы и интерфейсы, используемые для работы с группами объектов.

Когда речь идет о группе объектов, то на первый план выходит добавление объектов в группу, удаление объектов из группы и перебор объектов. Концептуально эти задачи решаются с помощью классов, которые реализуют определенные интерфейсы. Интерфейсы определяют стандарты работы с коллекциями. Классы фактически реализуют эти стандарты. Кроме классов и интерфейсов часто используются специальные методы, предназначенные для обработки групп объектов.

Важным в плане работы с коллекциями является обобщенный интерфейс `Collection`. Этот интерфейс определяет методы, которые должны быть у любой коллекции объектов. Другими словами, коллекция — это объект, созданный на основе класса, реализующего интерфейс `Collection`. Основные методы из интерфейса `Collection` перечислены в табл. 11.1.

Обобщенный интерфейс `List` расширяет интерфейс `Collection` и предназначен для реализации коллекций типа *списка*: это упорядоченные последовательности объектов, к которым можно получать доступ по индексу. Некоторые методы, дополнительно появляющиеся в интерфейсе `List`, перечислены в табл. 11.2.

Существуют и другие интерфейсы, предназначенные для работы с коллекциями. Все они расширяют интерфейс `Collection` и реализуются в классах, на основе которых создаются коллекции объектов.

Таблица 11.1. Методы интерфейса Collection

Метод	Описание
`add()`	Метод предназначен для добавления объекта, переданного аргументом, в коллекцию, из которой вызывается метод
`addAll()`	В коллекцию, из которой вызывается метод, добавляются все объекты из коллекции, переданной аргументом методу
`clear()`	Метод удаляет все объекты из коллекции (из которой вызывается метод)
`contains()`	Метод для проверки того, содержится ли объект, переданный аргументом, в коллекции, из которой вызывается метод
`containsAll()`	Метод позволяет проверить, содержится ли коллекция объектов, переданная аргументом методу, в коллекции, из которой вызывается метод
`equals()`	Метод для проверки (на предмет равенства) двух коллекций
`hashCode()`	Метод возвращает результатом хеш-код для коллекции
`isEmpty()`	Метод для проверки коллекции на предмет того, является ли она пустой
`iterator()`	Метод возвращает ссылку на объект *итератора* для коллекции. Итератор — специальный объект, с помощью которого реализуется доступ к объектам из коллекции
`remove()`	Метод для удаления из коллекции объекта, переданного аргументом методу
`removeAll()`	Аргументом методу передается коллекция объектов. Эти объекты удаляются из коллекции, из которой вызывается метод
`retainAll()`	При вызове метода из коллекции в ней удаляются все объекты, за исключением тех, которые входят в коллекцию, переданную аргументом методу
`size()`	Метод для определения размера коллекции (количество объектов, входящих в коллекцию)
`toArray()`	Метод возвращает ссылку на массив объектов, входящих в коллекцию, из которой вызывается метод

Таблица 11.2. Некоторые методы интерфейса List

Метод	Описание
`add()`	Метод для вставки в список объекта (второй аргумент метода) в место, определяемое индексом (первый аргумент метода). Метод вызывается из списка
`addAll()`	Метод для добавления в список коллекции объектов (второй аргумент метода), начиная с позиции, которая определяется индексом (первый аргумент метода). Метод вызывается из списка
`get()`	Метод возвращает ссылку на объект, индекс которого передается аргументом методу. Метод вызывается из списка
`indexOf()`	Метод возвращает индекс первого вхождения объекта (аргумент метода) в список, из которого вызывается метод. Если объекта в списке нет, возвращается значение `-1`
`lastIndexOf()`	Метод возвращает индекс последнего вхождения объекта (аргумент метода) в список, из которого вызывается метод. Если объекта в списке нет, возвращается значение `-1`
`listIterator()`	Метод возвращает ссылку на *итератор списка*. Итератор списка — специальный объект, используемый для получения доступа к элементам списка
`remove()`	Метод для удаления элемента с указанным индексом (аргумент метода) из списка, из которого вызывается метод
`set()`	В списке, из которого вызывается метод, объект с указанным индексом (первый аргумент метода) заменяется на объект, указанный вторым аргументом метода
`subList()`	Метод возвращает подсписок списка, из которого вызывается метод. Аргументами методу передается индекс первого объекта, который включается в подсписок, и индекс элемента после последнего, включаемого в список

Списки

Первый раз таких единоличников вижу.

из к/ф «Девчата»

Есть несколько обобщенных классов, которые реализуют интерфейс `List`. Среди них можно выделить классы `ArrayList`, `LinkedList`, `Stack` и `Vector` (все классы находятся в пакете `java.util`). Например, класс `ArrayList` предназначен для реализации *динамических списков*, которые могут изменять размер в процессе вы-

полнения программы. Такой список создается с некоторым начальным размером. Если в процессе работы возникает необходимость, то список автоматически расширяется. При удалении объектов из списка его размер может уменьшаться. У класса есть несколько конструкторов, в том числе конструктор без аргументов (создается пустой список), конструктор с целочисленным аргументом (определяет начальный размер списка) и конструктор, аргументом которому передается ссылка на коллекцию (объект класса, реализующего интерфейс `Collection`). Пример использования класса `ArrayList` для создания динамического списка представлен в листинге 11.11.

Листинг 11.11. Знакомство с динамическими списками

```
// Импорт класса:
import java.util.ArrayList;
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание пустого списка:
      ArrayList<Integer> nums=new ArrayList<>();
      // Отображение содержимого списка:
      System.out.println(nums);
      // Размер списка:
      System.out.println("Размер: "+nums.size());
      // Добавление элементов в список:
      nums.add(100);
      nums.add(200);
      nums.add(300);
      // Отображение содержимого списка:
      System.out.println(nums);
      // Размер списка:
      System.out.println("Размер: "+nums.size());
      // Добавление элементов в список:
      nums.add(0,-1);
      nums.add(2,0);
      // Отображение содержимого списка:
      System.out.println(nums);
      // Размер списка:
      System.out.println("Размер: "+nums.size());
      // Замена элемента в списке:
      nums.set(1,123);
      // Содержимое списка:
      System.out.println(nums);
      // Удаление элемента из списка:
      nums.remove(2);
      // Содержимое списка:
      System.out.println(nums);
      // Размер списка:
      System.out.println("Размер: "+nums.size());
      // Очистка списка:
      nums.clear();
      // Отображение содержимого списка:
      System.out.println(nums);
```

```
        // Размер списка:
        System.out.println("Размер: "+nums.size());
        // Если список пустой:
        if(nums.isEmpty()) nums.add(321);
        // Отображение содержимого списка:
        System.out.println(nums);
        // Размер списка:
        System.out.println("Размер: "+nums.size());
    }
}
```

Результат выполнения программы таков:

Результат выполнения программы (из листинга 11.11)

```
[]
Размер: 0
[100, 200, 300]
Размер: 3
[-1, 100, 0, 200, 300]
Размер: 5
[-1, 123, 0, 200, 300]
[-1, 123, 200, 300]
Размер: 4
[]
Размер: 0
[321]
Размер: 1
```

В программе на основе обобщенного класса `ArrayList` (импортируется из пакета `java.util`) создается список `nums` с целочисленными элементами. Проверка показывает, что вначале этот список пустой — в нем нет элементов.

ПОДРОБНОСТИ

Для отображения содержимого списка мы передаем его аргументом методу `println()` (в этом случае выполняется автоматическое приведение к текстовому формату), а узнать размер списка (количество элементов в нем) можно с помощью метода `size()`.

Затем последовательно командами `nums.add(100)`, `nums.add(200)` и `nums.add(300)` в список добавляются три элемента (`100`, `200` и `300`). Каждый очередной элемент добавляется в конец списка. Так что элементы в списке размещены в соответствии с тем, в какой последовательности они добавлялись в список. А вот командой `nums.add(0,-1)` новый элемент со значением `-1` добавляется в начало списка — на позицию с индексом `0`. Прочие элементы, включая и тот, что находился ранее в начале списка, сдвигаются вправо. После этого командой `nums.add(2,0)` в позицию с индексом `2` добавляется элемент со значением `0`.

На следующем этапе командой `nums.set(1,123)` заменяется элемент с индексом `1`. Теперь в списке на этой позиции будет значение `123`.

Командой `nums.remove(2)` из списка удаляется элемент, находящийся на позиции с индексом `2`. А командой `nums.clear()` выполняется очистка списка. После этого список не содержит элементов. В этот пустой список добавляется новый элемент, но при этом проверяется условие `nums.isEmpty()`, которое истинно, если список пустой. Новый элемент (со значением `321`) добавляется в список командой `nums.add(321)`.

Динамический список может создаваться на основе не только класса `ArrayList`, но и класса `Vector` из пакета `java.util` (листинг 11.12).

Листинг 11.12. Динамический список

```
// Импорт класса:
import java.util.Vector;
class Demo{
   public static void main(String[] args){
      // Создание пустого списка:
      Vector<String> str=new Vector<>();
      // Содержимое списка:
      System.out.println(str);
      System.out.println("Размер: "+str.size());
      // Добавление элементов в список:
      str.addElement("Первый");
      str.addElement("Второй");
      str.addElement("Третий");
      // Содержимое списка:
      System.out.println(str);
      System.out.println("Размер: "+str.size());
      // Добавление элементов в список:
      str.add(0,"Начальный");
      str.add(2,"Дополнительный");
      // Содержимое списка:
      System.out.println(str);
      System.out.println("Размер: "+str.size());
      // Изменение размера списка:
      str.setSize(str.size()-2);
      System.out.println(str);
      System.out.println("Размер: "+str.size());
   }
}
```

Результат выполнения программы показан ниже:

Результат выполнения программы (из листинга 11.12)

```
[]
Размер: 0
[Первый, Второй, Третий]
Размер: 3
[Начальный, Первый, Дополнительный, Второй, Третий]
```

```
Размер: 5
[Начальный, Первый, Дополнительный]
Размер: 3
```

В программе на основе обобщенного класса `Vector` создается список `str`, и в качестве значения обобщенного параметра используется класс `String` (то есть список предназначен для хранения текстовых значений). Это пустой список.

ПОДРОБНОСТИ

Если конструктору класса `Vector` аргументы не передаются, то создается пустой список. Аргументом можно передать коллекцию объектов, на основе которой создается список. При создании списка на основе класса `Vector` под него выделяется какой-то объем памяти. Если возникает необходимость, то объем памяти увеличивается (по умолчанию — в два раза). Один числовой аргумент у конструктора определяет объем выделяемой памяти, а второй аргумент (если он есть) определяет добавку к выделяемой под список памяти.

Элементы в список добавляются с помощью метода `addElement()`. Также можно использовать метод `add()`, указав первым аргументом индекс позиции, на которую вставляется элемент, а второй аргумент определяет значение элемента.

Размер списка можем узнать с помощью метода `size()`. А вот с помощью метода `setSize()` задаем размер списка. Если размер списка уменьшается, то лишние элементы отбрасываются. Если размер списка увеличивается, то дополнительными элементами будут пустые ссылки `null`.

ПОДРОБНОСТИ

Класс `Vector` аналогичен классу `ArrayList`, и он синхронизирован (то есть рассчитан на безопасную работу в многопоточной модели). Если многопоточный подход задействован не будет, то рекомендуется использовать класс `ArrayList`.

Класс `LinkedList` из пакета `java.util` реализует интерфейс `List` и позволяет создавать *связные списки* — цепочки объектов, в которые объекты можно добавлять и из которых объекты можно удалять.

ПОДРОБНОСТИ

Список, созданный на основе класса `LinkedList`, отличается от списка, созданного на основе `ArrayList`, способом реализации. Коллекция на основе класса `ArrayList` создается с определенным запасом. Размер коллекции в случае необходимости изменяется. Коллекция на основе `LinkedList` формируется как цепочка элементов и содержит ровно столько элементов, сколько их было добавлено в коллекцию (то есть такой список не содержит дополнительных позиций, не заполненных элементами).

Если конструктору класса `LinkedList` не передавать аргументы, то создается пустой список. Можно создать список на основе уже существующей коллекции. Для этого ссылка на коллекцию передается конструктору класса `LinkedList` (листинг 11.13).

Листинг 11.13. Связный список

```
// Импорт класса:
import java.util.LinkedList;
class Demo{
   public static void main(String[] args){
      // Создание пустого списка:
      LinkedList<Character> symbs=new LinkedList<>();
      // Добавление элементов в список:
      symbs.add('A');
      System.out.println(symbs);
      symbs.add(0,'B');
      System.out.println(symbs);
      symbs.addLast('C');
      System.out.println(symbs);
      // Добавление и удаление элементов:
      symbs.removeFirst();
      System.out.println(symbs);
      symbs.add(0,'D');
      System.out.println(symbs);
      symbs.removeLast();
      System.out.println(symbs);
      symbs.addLast('E');
      System.out.println(symbs);
   }
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 11.13)

```
[A]
[B, A]
[B, A, C]
[A, C]
[D, A, C]
[D, A]
[D, A, E]
```

Список `symbs` создается на основе обобщенного класса `LinkedList` со значением `Character` для обобщенного параметра. Помимо уже знакомых нам методов, для добавления нового элемента в конец списка используется метод `addLast()` (а в начало списка можно добавить элемент с помощью метода `addFirst()`). Первый элемент из списка удаляется с помощью метода `removeFirst()`, последний элемент из списка можно удалить посредством метода `removeLast()`.

Множества

Фигуры, может, и нет, а характер — налицо.

из к/ф «Девчата»

Обобщенный интерфейс `Set` расширяет интерфейс `Collection` и предназначен для реализации *множеств*. Множество представляет собой коллекцию, состоящую из уникальных элементов, порядок следования которых неважен.

НА ЗАМЕТКУ

При работе с множествами отсутствует такое понятие, как индекс объекта. Также отметим важную особенность множеств: при добавлении во множество элемента, который уже там есть, новый элемент не добавляется. То есть все элементы во множестве представлены только в одном экземпляре.

Одним из классов, которые реализуют интерфейс `Set` и с помощью которых можно создавать коллекции типа множества, является `HashSet`. Пример создания множества на основе класса `HashSet` представлен в листинге 11.14.

Листинг 11.14. Знакомство с множествами

```
// Импорт класса:
import java.util.HashSet;
class Demo{
   public static void main(String[] args){
      // Создание пустой коллекции:
      HashSet<String> objs=new HashSet<>();
      // Добавление элементов в коллекцию:
      objs.add("Первый");
      objs.add("Второй");
      objs.add("Третий");
      // Отображение содержимого коллекции:
      System.out.println(objs);
      // Попытка добавить существующий элемент:
      objs.add("Второй");
      // Отображение содержимого коллекции:
      System.out.println(objs);
      // Количество элементов в коллекции:
      System.out.println("Размер: "+objs.size());
      // Удаление элемента из коллекции:
      objs.remove("Первый");
      // Отображение содержимого коллекции:
      System.out.println(objs);
      // Добавление элемента в коллекцию:
      objs.add("Четвертый");
      // Отображение содержимого коллекции:
      System.out.println(objs);
```

```
      // Наличие/отсутствие элементов в коллекции:
      if(!objs.contains("Первый")){
         System.out.println("Элемента \"Первый\" нет");
      }
      if(objs.contains("Четвертый")){
         System.out.println("Элемент \"Четвертый\" есть");
      }
   }
}
```

Результат выполнения программы таков:

Результат выполнения программы (из листинга 11.14)

```
[Первый, Третий, Второй]
[Первый, Третий, Второй]
Размер: 3
[Третий, Второй]
[Четвертый, Третий, Второй]
Элемента "Первый" нет
Элемент "Четвертый" есть
```

Кроме этого, множество можно создавать на основе обобщенного класса `TreeSet` из пакета `java.util` (в классе `TreeSet` также реализуется интерфейс `Set`).

НА ЗАМЕТКУ

Принципиальная разница между классами `HashSet` и `TreeSet` состоит в способе реализации коллекций. Коллекция на основе класса `TreeSet` реализуется в виде древовидной структуры, а доступ к элементам из коллекции на основе класса `HashSet` базируется на использовании специальных хеш-кодов.

Пример использования этого класса представлен в листинге 11.15.

Листинг 11.15. Создание множества

```
// Импорт класса:
import java.util.TreeSet;
class Demo{
   public static void main(String[] args){
      // Создание пустой коллекции:
      TreeSet<Double> objs=new TreeSet<>();
      // Добавление элементов в коллекцию:
      objs.add(100.0);
      objs.add(200.0);
      objs.add(300.0);
      // Отображение содержимого коллекции:
      System.out.println(objs);
      // Попытка добавить существующий элемент:
      objs.add(200.0);
```

```
      // Отображение содержимого коллекции:
      System.out.println(objs);
      // Удаление элемента из коллекции:
      objs.remove(100.0);
      // Отображение содержимого коллекции:
      System.out.println(objs);
      // Добавление элемента в коллекцию:
      objs.add(123.0);
      // Отображение содержимого коллекции:
      System.out.println(objs);
   }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 11.15)

```
[100.0, 200.0, 300.0]
[100.0, 200.0, 300.0]
[200.0, 300.0]
[123.0, 200.0, 300.0]
```

Несмотря на кажущуюся экзотичность, множества обладают весьма полезными свойствами, например, позволяющими генерировать случайные числа (листинг 11.16).

Листинг 11.16. Генерирование случайных чисел

```
import java.util.*;
class Demo{
   public static void main(String[] args){
      // Объект для генерирования случайных чисел:
      Random rnd=new Random();
      // Количество разных случайных чисел:
      int n=12;
      // Множество чисел:
      HashSet<Integer> nums=new HashSet<>();
      // Генерирование случайных чисел:
      while(nums.size()<n){
         nums.add(rnd.nextInt(20)+1);
      }
      // Результат:
      System.out.println(nums);
   }
}
```

Результат выполнения программы (с учетом того, что используются случайные числа) может быть таким:

Результат выполнения программы (из листинга 11.16)

```
[1, 2, 19, 3, 20, 4, 5, 7, 8, 13, 14, 15]
```

Специфика задачи в том, что нас интересует не просто определенное количество случайных чисел (такая задача просто решается с помощью оператора цикла и массива), а определенное количество *разных* случайных чисел. Поэтому мы воспользовались множеством и тем его свойством, что при добавлении во множество уже существующего элемента этот элемент не дублируется. Решается задача просто: создается пустое множество и в него добавляются элементы. Оператор цикла выполняется до тех пор, пока размер множества меньше количества случайных чисел, которые необходимо сгенерировать.

Резюме

Прием окончен. Обеденный перерыв.

из к/ф «Иван Васильевич меняет профессию»

- В описании класса тип данных можно передавать в качестве параметра. В таком случае после имени класса в угловых скобках указывается обозначение для обобщенных параметров. В описании класса эти параметры используются как идентификаторы типа. При создании объекта на основе обобщенного класса в соответствующей команде после имени класса в угловых скобках указываются значения для обобщенных параметров. В качестве значений обобщенных параметров можно указывать только ссылочные типы (то есть названия классов).
- Обобщенными могут быть интерфейсы. На основе обобщенного интерфейса можно создать как обобщенный, так и обычный класс.
- Обобщенные классы могут участвовать в наследовании. На основе обобщенного класса можно создать и обобщенный, и обычный класс.
- Существует возможность создавать обобщенные методы (как статические, так и обычные, не статические). В описании обобщенного метода обобщенные параметры объявляются в угловых скобках перед идентификатором типа результата метода. Значения для обобщенных параметров при вызове метода определяются по типу аргументов, переданных методу. Также можно указать значения для обобщенных параметров в угловых скобках перед именем метода при его вызове.
- Обобщенные методы можно перегружать. Одновременно могут использоваться как обобщенные, так и обычные версии метода.
- На обобщенные параметры можно накладывать ограничение, указав, что класс, используемый в качестве обобщенного параметра, является подклассом некоторого суперкласса. В таком случае в объявлении обобщенного параметра после его имени указывается ключевое слово `extends` и название суперкласса.

- Существуют специальные обобщенные классы, на основе которых создаются коллекции — группы объектов, которые можно обрабатывать с помощью специальных методов (в том числе добавлять элементы в коллекцию и удалять элементы из коллекции). Классы, используемые для создания коллекций, реализуют обобщенный интерфейс `Collection` (этот интерфейс расширяется в интерфейсах `List` и `Set`).
- Список представляет собой упорядоченный набор элементов. Списки можно создавать на основе классов `ArrayList`, `Vector` и `LinkedList`.
- Множество представляет собой неупорядоченный набор уникальных элементов. Для создания множества можно использовать классы `HashSet` и `TreeSet`.

12 Программы с графическим интерфейсом

Что вы на это скажете, мой дорогой психолог?

из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

В этой главе обсудим создание приложений с графическим интерфейсом. Мы уже сталкивались с программами, в которых отображались диалоговые окна. Но там для отображения использовались специальные библиотечные методы. Теперь же нам предстоит научиться создавать окно самостоятельно с нуля.

В Java для создания графического интерфейса обычно используются библиотеки AWT и Swing. Исторически первой и базовой была библиотека AWT (*Abstract Window Toolkit*). Библиотека Swing появилась несколько позже, хотя она во многом базируется на библиотеке AWT. В этих библиотеках по-разному решается проблема универсальности кода, краеугольная для концепции, положенной в основу Java.

В библиотеке AWT для реализации графических компонентов привлекаются средства операционной системы. Такие компоненты называют *тяжелыми*. Главная проблема здесь связана с тем, что у разных операционных систем разный инструментарий и разные возможности.

Компоненты, созданные с помощью библиотеки Swing, реализуются средствами Java. Такие компоненты называются *легкими*.

НА ЗАМЕТКУ

Поскольку механизм отображения тяжелых компонентов базируется на привлечении средств операционной системы, то по виду такие компоненты сходны с соответствующими компонентами для используемой операционной системы. Легкие компоненты, созданные на основе библиотеки Swing, в этом отношении более независимые.

Компоненты, созданные на основе библиотеки Swing, более функциональны, поэтому для создания графических компонентов обычно используют их. При этом важно понимать, что библиотека Swing не заменяет библиотеку AWT, а дополняет ее. Без библиотеки AWT нам в любом случае не обойтись.

НА ЗАМЕТКУ

Поскольку механизмы реализации компонентов из библиотек AWT и Swing разные, крайне не рекомендуется использовать одновременно компоненты разных типов.

Что касается основных подходов к созданию графического интерфейса, то читателям с ними уже приходилось сталкиваться. Для каждого графического компонента (например, кнопки, текстового поля, метки) существует определенный класс, причем это может быть как класс из библиотеки AWT, так и класс из библиотеки Swing. На основе этого класса создается объект. С помощью методов объекта настраиваются свойства графического компонента. Если необходимо, компонент добавляется в контейнер (например, окно или панель) — для этого тоже есть специальные методы. Самая сложное — научить компонент реагировать на действия пользователя или другие события. На этом этапе будет задействована система *обработки событий*, о которой читатели тоже имеют определенное представление. Теперь нам предстоит познакомиться с системой обработки событий более основательно, на соответствующих примерах.

Создание простого окна

> — Друг, у вас какая система? Разрешите взглянуть?
>
> — Система обычная. Нажал на кнопку — и дома.
>
> *из к/ф «Кин-дза-дза»*

Знакомство с подходами по реализации графического интерфейса начнем с создания обычного, простого окна. Для этого можно создать объект или на основе класса `Frame` из библиотеки AWT, или на основе класса `JFrame` из библиотеки Swing.

НА ЗАМЕТКУ

Класс `JFrame` является подклассом класса `Frame`. Также следует отметить, что для многих классов из библиотеки Swing название отличается от названия соответствующего класса из библиотеки AWT наличием первой большой буквы `'J'`.

Сначала рассмотрим процесс создания окна средствами библиотеки AWT. Функциональность окна будет ограничена реагированием на нажатие системной пиктограммы (пиктограмма с крестиком в правом верхнем углу окна, предназначенная для его закрытия, листинг 12.1).

Листинг 12.1. Создание окна средствами AWT

```
// Подключение пакетов:
import java.awt.*;
import java.awt.event.*;
class Demo{
   public static void main(String[] args){
      // Создание объекта окна:
      Frame wnd=new Frame("Окно AWT");
      // Размеры окна:
      wnd.setSize(300,200);
      // Положение окна:
      wnd.setLocation(500,400);
      // Добавление обработчика в окно:
      wnd.addWindowListener(new WindowAdapter(){
         public void windowClosing(WindowEvent we){
            // Завершение выполнения программы:
            System.exit(0);
         }
      });
      // Отображение окна:
      wnd.setVisible(true);
   }
}
```

При запуске программы появляется окно (рис. 12.1).

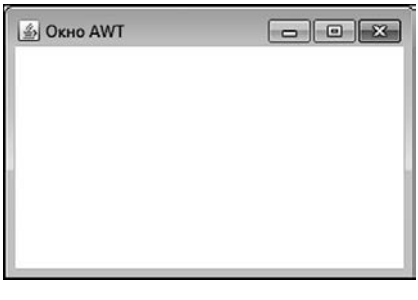


Рис. 12.1. Окно создано средствами библиотеки AWT

Окно можно перемещать по экрану, растягивать с помощью мышки, сворачивать и разворачивать нажатием системных пиктограмм в правом верхнем углу окна. Все это — встроенное поведение окна. Нажатие системной пиктограммы с крестиком приводит к закрытию окна и завершению выполнения программы. И это уже наша заслуга (так в программе организована обработка событий).

Проанализируем код примера. В заголовке программы используются две `import`-инструкции для импортирования классов из пакетов `java.awt` и `java.awt.event`. Они нужны для того, чтобы в программе стали доступными классы из библиотеки AWT. В главном методе на основе класса `Frame` создается объект окна, ссылка на который записывается в объектную переменную `wnd`. Текст `"Окно AWT"`, переданный аргументом конструктору класса `Frame`, определяет название окна (отображается в строке названия в верхней части окна). Размеры окна (ширина и высота в пикселах) задаются командой `wnd.setSize(300,200)`. Положение окна на экране определяется командой `wnd.setLocation(500,400)`. Методу `setLocation()` передаются координаты (в пикселах) левого верхнего угла окна по отношению к левому верхнему углу экрана.

ПОДРОБНОСТИ

При определении координат компонента определяются координаты точки в левом верхнем углу компонента. Координаты определяются по отношению к точке в левом верхнем углу контейнера (для окна контейнером является экран). Первая координата определяется вдоль горизонтали в направлении слева направо. Вторая координата определяется вдоль вертикали в направлении сверху вниз.

Команда `wnd.setVisible(true)` отображает на экране окно: метод `setVisible()` вызывается из объекта окна `wnd`, и аргументом методу передается значение `true`.

НА ЗАМЕТКУ

Если вызвать из объекта окна метод `setVisible()` с аргументом `false`, то окно пропадет с экрана (но не будет удалено из памяти).

Перед отображением окна в объекте окна регистрируется обработчик события класса `WindowEvent`. Для добавления обработчика в окно из объекта окна вызывается метод `addWindowListener()`. Аргументом методу `addWindowListener()` передается анонимный объект анонимного класса, созданного путем наследования класса-адаптера `WindowAdapter`. В этом анонимном классе переопределяется метод `windowClosing()`, который автоматически вызывается в случае, если пользователь пытается закрыть окно. В теле метода всего одна команда — `System.exit(0)`, которой завершается выполнение программы.

ПОДРОБНОСТИ

Любой графический компонент может реагировать на определенные события. Тип события соотносится с определенным классом. События, связанные с операциями на уровне окна, относятся к классу `WindowEvent`. Если мы хотим научить компонент реагировать на событие определенного типа, то нам необходимо создать специальный объект-обработчик и добавить его в компонент (зарегистрировать его в компоненте). Для регистрации обработчика для событий класса `WindowEvent` используют метод `addWindowListener()`. Объект-обработчик создается на основе класса, реализующего интерфейс `WindowListener`. В этом интерфейсе семь

абстрактных методов, каждый из которых вызывается в определенной ситуации. Нас интересует лишь метод `windowClosing()`, который вызывается при попытке закрыть окно. Чтобы не описывать все семь методов, мы используем класс-адаптер `WindowAdapter`, который реализует методы из интерфейса `WindowListener` с пустым телом. На основе класса `WindowAdapter` мы создаем (анонимный) подкласс и переопределяем в нем метод `windowClosing()`.

Для создания окна нами использован формально правильный подход, но обычно так не делают. Вместо того чтобы в главном методе поэтапно создавать окно и настраивать его параметры, разумнее на основе класса `Frame` создать подкласс и весь процесс формирования объекта окна вынести туда. Модификация предыдущего примера с учетом сделанного замечания представлена в листинге 12.2.

Листинг 12.2. Еще одно окно средствами AWT

```
// Подключение пакетов:
import java.awt.*;
import java.awt.event.*;
// Класс для создания окна:
class MyFrame extends Frame{
   // Конструктор:
   MyFrame(String name){
      // Вызов конструктора суперкласса:
      super(name);
      // Размеры окна:
      setSize(300,200);
      // Положение окна:
      setLocation(500,400);
      // Добавление обработчика в окно:
      addWindowListener(new WindowAdapter(){
         public void windowClosing(WindowEvent we){
            // Завершение выполнения программы:
            System.exit(0);
         }
      });
      // Отображение окна:
      setVisible(true);
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание и отображение окна:
      new MyFrame("Окно AWT");
   }
}
```

В программе мы описываем класс `MyFrame`, создаваемый наследованием класса `Frame`. Все команды по определению параметров окна вынесены в конструктор (в том числе и команда отображения окна). У конструктора есть текстовый ар-

гумент, который передается аргументом конструктору суперкласса, и это будет название окна. В главном методе мы создаем анонимный объект класса `MyFrame`. Результат выполнения программы — такой же, как и в предыдущем случае.

Теперь создадим аналогичное окно, но только средствами библиотеки Swing. Подход может быть сохранен, но вместо класса `Frame` будет использоваться класс `JFrame`, возможности которого немного шире, поэтому и часть задач решается проще (листинг 12.3).

Листинг 12.3. Создание окна средствами Swing

```
// Подключение пакетов:
import javax.swing.*;
class Demo{
   public static void main(String[] args){
      // Создание объекта окна:
      JFrame wnd=new JFrame("Окно Swing");
      // Размеры окна:
      wnd.setSize(300,200);
      // Положение окна:
      wnd.setLocation(500,400);
      // Реакция на нажатие системной пиктограммы:
      wnd.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      // Отображение окна:
      wnd.setVisible(true);
   }
}
```

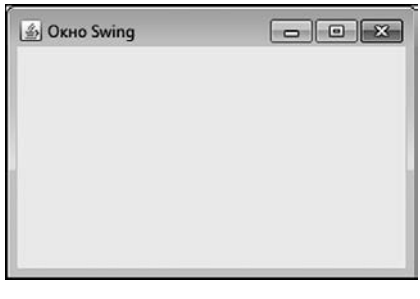


Рис. 12.2. Окно создано средствами библиотеки Swing

Обратим внимание на некоторые отличия. Во-первых, теперь нам в `import`-инструкции необходимо импортировать классы (на самом деле нам нужен класс `JFrame`) из пакета `javax.swing`. Объект окна создается на основе класса `JFrame`. Вместо регистрации обработчика для событий класса `WindowEvent` вызовем из объекта окна метод `setDefaultCloseOperation()` и аргументом ему передадим статическую константу `EXIT_ON_CLOSE` из класса `JFrame`. Такой аргумент означает, что при попытке закрыть окно программа завершает выполнение. При запуске программы появляется окно (рис. 12.2).

Окно по функциональным возможностям такое же, как и в рассмотренных ранее примерах.

При работе с классом `JFrame` можно использовать ту же стратегию, что и в программе из листинга 12.2. Модификация этого примера (с использованием класса `JFrame`) представлена в листинге 12.4.

Листинг 12.4. Еще одно окно средствами Swing

```
// Подключение пакетов:
import javax.swing.*;
// Класс для создания окна:
class MyFrame extends JFrame{
   // Конструктор:
   MyFrame(String name){
      // Вызов конструктора суперкласса:
      super(name);
      // Размеры окна:
      setSize(300,200);
      // Положение окна:
      setLocation(500,400);
      // Реакция на нажатие системной пиктограммы:
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      // Отображение окна:
      setVisible(true);
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание и отображение окна:
      new MyFrame("Окно Swing");
   }
}
```

При обращении к константе `EXIT_ON_CLOSE` в классе `MyFrame` ссылку на класс `JFrame` указывать не нужно, поскольку константа наследуется в классе `MyFrame`. Результат выполнения программы точно такой, как в предыдущем случае.

Окно с кнопками и меткой

> Ведь это же настоящая тайна! Ты потом никогда себе не простишь!
>
> *из к/ф «Гостья из будущего»*

Теперь немного усложним пример и создадим окно с *кнопками* и *меткой* — сначала средствами AWT, а потом с помощью библиотеки Swing. Но прежде чем приступить к анализу кода, рассмотрим результаты выполнения программы — так будет легче

понять принцип организации кода. Итак, при запуске программы появляется окно (рис. 12.3).

В центральной части окна — серая область (*панель*), на которой размещен текст (текстовая метка). Текст отображается синим цветом, содержит число, и в самом начале выполнения программы он такой: `Нажмите кнопку "Число": 0`. В нижней части окна есть две кнопки. Одна называется Закрыть, и при нажатии этой кнопки окно закрывается, а программа завершает выполнение. Еще одна кнопка называется Число. При нажатии этой кнопки в метке, которая находится в центральной части окна, на единицу увеличивается числовое значение (рис. 12.4).

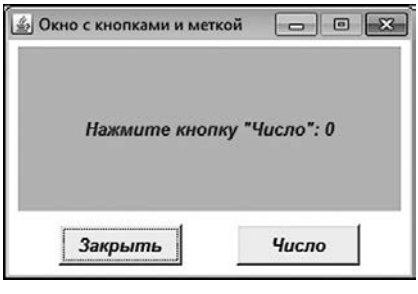


Рис. 12.3. Окно с меткой и двумя кнопками

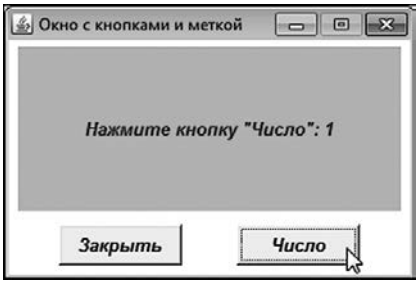


Рис. 12.4. При нажатии кнопки Число числовое значение в метке увеличивается на единицу

Но так может продолжаться не очень долго. Если последовательно нажать кнопку Число три раза, то в метке будет отображаться значение 3 (рис. 12.5).

Если после этого еще раз нажать кнопку Число, то текст в метке изменится, а кнопка будет заблокирована (рис. 12.6).

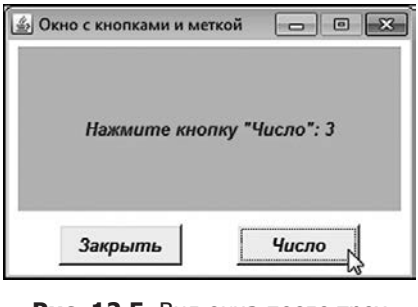


Рис. 12.5. Вид окна после трех последовательных нажатий кнопки Число

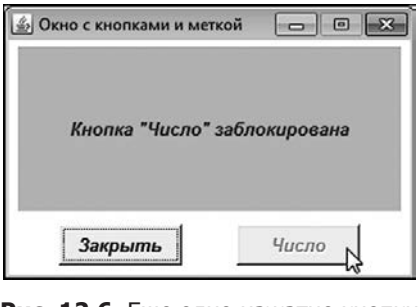


Рис. 12.6. Еще одно нажатие кнопки Число приводит к изменению текста в метке и блокированию кнопки

После этого остается только закрыть окно. Код представлен в листинге 12.5.

Листинг 12.5. Окно с кнопками и меткой на основе AWT

```
import java.awt.*;
import java.awt.event.*;
// Класс для реализации окна:
class MyFrame extends Frame{
   private int count=0;
   private String text="Нажмите кнопку \"Число\": ";
   // Конструктор:
   MyFrame(int x,int y){
      super();
      // Название окна:
      setTitle("Окно с кнопками и меткой");
      // Положение и размеры окна:
      setBounds(x,y,300,200);
      // Окно фиксированных размеров:
      setResizable(false);
      // Обработчик для окна:
      addWindowListener(new WindowAdapter(){
         public void windowClosing(WindowEvent we){
            System.exit(0);
         }
      });
      // Отключение менеджера компоновки для окна:
      setLayout(null);
      // Создание объекта шрифта:
      Font f=new Font(Font.DIALOG,
         Font.BOLD|Font.ITALIC,13);
      // Применение шрифта для окна:
      setFont(f);
      // Создание панели:
      Panel P=new Panel();
      // Положение и размеры панели:
      P.setBounds(10,30,280,120);
      // Цвет фона для панели:
      P.setBackground(Color.LIGHT_GRAY);
      // Определение менеджера компоновки для панели:
      P.setLayout(new BorderLayout());
      // Создание метки:
      Label L=new Label(text+count);
      // Выравнивание текста по центру метки:
      L.setAlignment(Label.CENTER);
      // Цвет для шрифта метки:
      L.setForeground(Color.BLUE);
      // Добавление метки в центр панели:
      P.add(L,BorderLayout.CENTER);
      // Добавление панели в окно:
      add(P);
      // Создание кнопки:
      Button A=new Button("Закрыть");
      // Положение и размеры кнопки:
      A.setBounds(40,160,90,30);
      // Обработчик для кнопки:
```

```
      A.addActionListener(ae->System.exit(0));
      // Добавление кнопки в окно:
      add(A);
      // Создание кнопки:
      Button B=new Button("Число");
      // Обработчик для кнопки:
      B.addActionListener(new ActionListener(){
         public void actionPerformed(ActionEvent ae){
            count++;
            if(count>3){
               B.setEnabled(false);
               L.setText("Кнопка \"Число\" заблокирована");
            }else{
               L.setText(text+count);
            }
         }
      });
      // Положение и размеры кнопки:
      B.setBounds(170,160,90,30);
      // Добавление кнопки в окно:
      add(B);
      // Отображение окна:
      setVisible(true);
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание и отображение окна:
      new MyFrame(400,300);
   }
}
```

Для реализации окна создаем специальный класс `MyFrame`, который наследует класс `Frame`. В классе `MyFrame` есть закрытое целочисленное поле `count` с начальным нулевым значением и закрытое текстовое поле `text` (оно содержит основную часть текста метки). Все самое интересное происходит в конструкторе.

Конструктору класса передаются два целочисленных значения, которые определяют положение на экране окна.

ПОДРОБНОСТИ

Многие задачи, связанные с созданием графического интерфейса, могут решаться разными способами. Например, размеры компонента можно задать с помощью метода `setSize()`, а положение в контейнере — с помощью метода `setLocation()`. Вместо этих двух методов можно использовать метод `setBounds()`. Аргументами методу передаются координаты (горизонтальная и вертикальная) компонента и его размеры (ширина и высота).

Название окна задаем с помощью метода `setTitle()`. Мы создаем окно фиксированных размеров (нельзя изменить размеры окна мышью), поэтому из объекта окна вызываем метод `setResizable()` с аргументом `false`. Также для окна регистрируется обработчик события класса `WindowEvent`, связанный с попыткой закрыть окно.

Мы планируем в окне размещать компоненты. За размещение компонентов в контейнере отвечает специальный объект, который называется *менеджером компоновки*. Они бывают разными. У каждого менеджера компоновки есть определенный алгоритм, в соответствии с которым он размещает компоненты в контейнере. Но можно отключить менеджер компоновки и размещать компоненты, указывая их координаты. Вызовем метод `setLayout()` и аргументом ему передадим пустую ссылку `null`. В общем случае аргументом методу передается ссылка на объект менеджера компоновки. Пустая ссылка означает, что менеджер компоновки не используется.

На основе класса `Font` создается объект шрифта, и затем этот объект передается аргументом методу `setFont()`. В результате данный шрифт применяется для окна.

ПОДРОБНОСТИ

Аргументами конструктору класса `Font` передаются приведенные ниже значения. Константа `DIALOG` является названием логического шрифта. Инструкция `Font.BOLD|Font.ITALIC` означает, что используется жирный курсивный стиль. Третий аргумент конструктора `13` определяет размер шрифта.

При создании окна используется *панель*, которая представляет собой прямоугольную область, служащую контейнером для размещения других компонентов. Крайней необходимости использовать панель в данном случае не было, скорее мы используем панель как декоративный элемент (поместив метку на панель). Панель создается на основе класса `Panel`, а ссылка на созданный объект записывается в переменную `P`. Положение и размеры панели задаются командой `P.setBounds(10,30,280,120)`.

ПОДРОБНОСТИ

Координаты компонента определяются по отношению к его контейнеру. Панель мы добавляем в окно (команда `add(P)`), поэтому в соответствии с командой `P.setBounds(10,30,280,120)` ширина панели равна `280` пикселам, высота панели составляет `120` пикселов. Левый верхний угол панели смещен по отношению к левому верхнему углу окна по горизонтали на `10` пикселов, а по вертикали — на `60` пикселов.

Командой `P.setBackground(Color.LIGHT_GRAY)` для панели задается светло-серый цвет фона, поэтому при отображении окна область панели выделена цветом. Также командой `P.setLayout(new BorderLayout())` задаем для панели менеджер компоновки. Объект менеджера компоновки создается на основе класса `BorderLayout`

и передается методу `setLayout()`, который вызывается из объекта панели. Благодаря использованию менеджера компоновки командой `P.add(L,BorderLayout.CENTER)` метка (ссылка на которую записывается в переменную `L`) добавляется в центральную часть панели.

ПОДРОБНОСТИ

У каждого контейнера (которыми являются окно и панель) есть свой менеджер компоновки. Мы отключили менеджер компоновки для окна, но не для панели. Менеджер компоновки, созданный на основе класса `BorderLayout`, позволяет добавлять компонент в центр контейнера и по его краям. Место, куда добавляется компонент, определяется вторым аргументом метода `add()`. Константа `CENTER` из класса `BorderLayout` означает, что компонент добавляется в центральную часть контейнера.

Сама метка создается как объект класса `Label`. Командой `L.setAlignment(Label.CENTER)` мы задаем для метки режим выравнивания текста по центру. Цвет для шрифта метки задается командой `L.setForeground(Color.BLUE)`.

Кнопки (их две) создаются на основе класса `Button`. Конструктору передается текст, который служит названием кнопки (отображается в области кнопки). Положение и размеры кнопок задаются с помощью метода `setBounds()`. Причем поскольку кнопки добавляются в окно (а не на панель), положение кнопок определяется по отношению к окну. Для добавления кнопок в окно используем метод `add()`.

Самая нетривиальная часть связана с созданием обработчиков для кнопок. Обе кнопки должны реагировать на нажатие. Это событие класса `ActionEvent`. Для обработки события данного класса необходимо создать объект-обработчик, и этот объект должен создаваться на основе класса, реализующего интерфейс `ActionListener`. Интерфейс является функциональным, и в нем объявлен всего один абстрактный метод `actionPerformed()`. Метод не возвращает результат, а аргументом метода является объект события класса `ActionEvent`. Данный метод будет автоматически вызываться в случае нажатия кнопки. Для регистрации обработчика в кнопке из объекта кнопки вызывается метод `addActionListener()`, а методу передается объект-обработчик.

НА ЗАМЕТКУ

Поскольку интерфейс `ActionListener` является функциональным, то методу `addActionListener()` можно передавать и лямбда-выражение.

Для первой кнопки (кнопка Закрыть) обработчик создается и регистрируется командой `A.addActionListener(ae->System.exit(0))`, в которой методу `addActionListener()` аргументом передано лямбда-выражение `ae->System.exit(0)`. Оно определяет код метода `actionPerformed()` в объекте, который будет создан и зарегистрирован обработчиком для кнопки.

ПОДРОБНОСТИ

У метода `actionPerformed()` есть аргумент, который мы в данном случае не используем. Поэтому у лямбда-выражения (в нашем случае это `ae->System.exit(0)`), которое определяет метод, тоже должен быть аргумент. Мы его назвали `ae`, но это непринципиально.

Обработчик для второй кнопки создается на основе анонимного класса, реализующего интерфейс `ActionListener`. Инструкция создания анонимного объекта на основе анонимного класса передана аргументом методу `addActionListener()`. В этой команде описывается метод `actionPerformed()`, вызываемый в случае нажатия кнопки Число. В теле метода командой `count++` на единицу увеличивается значение целочисленного поля `count`, а затем в дело вступает условный оператор. В нем проверяется условие `count>3`, и если оно истинно, то командой `B.setEnabled(false)` кнопка делается неактивной (ее нельзя нажать), а командой `L.setText("Кнопка \"Число\" заблокирована")` меняется текст метки. Если же условие ложно, то командой `L.setText(text+count)` для метки просто применяется новый текст (новое в нем — только числовое значение).

После того как все настройки выполнены, командой `setVisible(true)` окно отображается на экране. Поэтому когда в главном методе программы создается анонимный объект класса `MyFrame`, то на экране появляется окно.

Реализация приложения средствами Swing представлена в листинге 12.6 (для сокращения объема кода комментарии удалены).

Листинг 12.6. Окно с кнопками и меткой на основе Swing

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class MyFrame extends JFrame{
   private int count=0;
   private String text="Нажмите кнопку \"Число\": ";
   MyFrame(int x,int y){
      super();
      setTitle("Окно с кнопками и меткой");
      setBounds(x,y,300,200);
      setResizable(false);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setLayout(null);
      Font f=new Font(Font.DIALOG,
              Font.BOLD|Font.ITALIC,13);
      JPanel P=new JPanel();
      P.setBounds(10,10,275,120);
      P.setBackground(Color.LIGHT_GRAY);
      P.setLayout(new BorderLayout());
      JLabel L=new JLabel(text+count);
      L.setHorizontalAlignment(JLabel.CENTER);
      L.setForeground(Color.BLUE);
      L.setFont(f);
```

```
      P.add(L,BorderLayout.CENTER);
      add(P);
      JButton A=new JButton("Закрыть");
      A.setBounds(30,135,100,30);
      A.setFont(f);
      A.addActionListener(ae->System.exit(0));
      add(A);
      JButton B=new JButton("Число");
      B.setFont(f);
      B.addActionListener(new ActionListener(){
         public void actionPerformed(ActionEvent ae){
            count++;
            if(count>3){
               B.setEnabled(false);
               L.setText("Кнопка \"Число\" заблокирована");
            }else{
               L.setText(text+count);
            }
         }
      });
      B.setBounds(170,135,100,30);
      add(B);
      setVisible(true);
   }
}
class Demo{
   public static void main(String[] args){
      new MyFrame(400,300);
   }
}
```

По сравнению с предыдущей программой изменения минимальны. Концептуально важно, что вместо классов `Frame`, `Label`, `Button` и `Panel` из библиотеки AWT мы используем соответственно классы `JFrame`, `JLabel`, `JButton` и `JPanel` из библиотеки Swing. Для возможности работы с этими классами в заголовке программы импортируется пакет `javax.swing`. Вместо описания обработчика для объекта окна мы воспользовались методом `setDefaultCloseOperation()`. При определении способа выравнивания текста в метке используется метод `setHorizontalAlignment()`. Также в силу специфики отображения компонентов из библиотеки Swing уточнены некоторые параметры, отвечающие за размеры и размещение компонентов (кнопок, в частности). Шрифт для каждого компонента применяется индивидуально.

При запуске программы появляется окно (рис. 12.7).

Функциональность окна такая же, как и в предыдущем случае. Так, мы можем последовательно нажимать кнопку Число до трех раз. Как выглядит окно после трех нажатий, показано на рис. 12.8.

Еще одно нажатие кнопки Число приводит к тому, что кнопка блокируется (рис. 12.9).

Закрыть окно можно с помощью системной пиктограммы или нажав кнопку Закрыть.

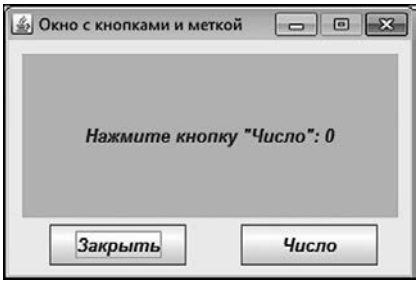


Рис. 12.7. Окно с двумя кнопками и текстовой меткой создано средствами Swing

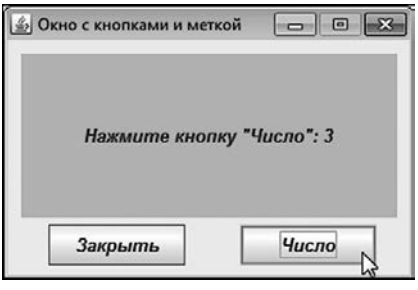


Рис. 12.8. Вид окна после трех последовательных нажатий кнопки Число

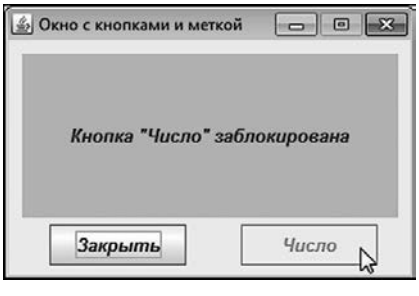


Рис. 12.9. Очередное нажатие кнопки Число приводит к блокировке кнопки и изменению текста метки

НА ЗАМЕТКУ

Хотя в рассмотренных примерах средствами AWT и Swing создаются, в общем-то, эквивалентные программы, это не означает, что компоненты, реализованные на основе классов из разных библиотек, имеют одинаковые возможности. Компоненты, созданные на основе классов из библиотеки Swing, более гибкие, функциональные, у них больше настроек и более высокий уровень совместимости с разными операционными системами. Поэтому в работе с графическим интерфейсом предпочтение отдается библиотеке Swing.

Компоненты и события

Огласите весь список, пожалуйста.

из к/ф «Операция Ы и другие приключения Шурика»

Здесь приводится краткий обзор классов компонентов, используемых при создании приложений с графическим интерфейсом. Кроме этого, мы проанализируем общие подходы, связанные с обработкой событий.

Мы уже знаем, что создание того или иного компонента означает создание объекта соответствующего класса. Поэтому необходимо знать, какие классы предназначены для создания того или иного компонента. Помимо таких компонентных классов, в библиотеках AWT и Swing есть много вспомогательных классов, важных при разработке интерфейса. Некоторые такие классы из библиотеки AWT перечислены в табл. 12.1.

Таблица 12.1. Некоторые классы из библиотеки AWT

Класс	Описание
`Button`	Класс для создания объекта кнопки
`Canvas`	Класс для реализации пустой прямоугольной области, которую можно использовать для рисования
`CardLayout`	Класс для создания менеджера компоновки
`Checkbox`	Класс для реализации переключателей и опций (компоненты, которые могут быть в одном из двух состояний — установленном и неустановленном)
`CheckboxGroup`	Класс для создания группы переключателей
`CheckboxMenuItem`	Класс для создания пункта меню типа переключателя
`Choice`	Класс для создания раскрывающегося списка
`Color`	Класс для реализации цветовых параметров
`Cursor`	Класс для реализации вида курсора мыши
`Dialog`	Класс для реализации диалогового окна
`Dimension`	Класс для реализации объекта, определяющего ширину и высоту компонента
`FileDialog`	Класс для реализации окна выбора файла
`FlowLayout`	Класс для создания менеджера компоновки
`Font`	Класс для создания объекта шрифта
`Frame`	Класс для создания окна
`GridBagLayout`	Класс для создания менеджера компоновки
`GridLayout`	Класс для создания менеджера компоновки
`Label`	Класс для создания метки
`List`	Класс для создания списка выбора
`Menu`	Класс для создания меню

Класс	Описание
`MenuBar`	Класс для создания панели меню
`MenuItem`	Класс для создания пункта меню
`Panel`	Класс для создания панели (прямоугольный компонент-контейнер)
`Point`	Класс для создания объекта, определяющего точку с двумя координатами
`Scrollbar`	Класс для реализации полосы прокрутки
`ScrollPane`	Класс для реализации контейнера с автоматической горизонтальной и вертикальной прокруткой
`TextArea`	Класс для создания текстовой области
`TextField`	Класс для создания текстового поля
`Window`	Класс для создания окна (без рамки и панели меню)

Полезные классы из библиотеки Swing представлены в табл. 12.2.

Таблица 12.2. Некоторые классы из библиотеки Swing

Класс	Описание
`BorderFactory`	Класс для создания объектов, определяющих рамки компонентов
`Box`	Контейнер, в котором используется менеджер компоновки класса `BoxLayout`
`BoxLayout`	Класс для создания менеджера компоновки
`ButtonGroup`	Класс для создания групп переключателей
`GroupLayout`	Класс для создания менеджера компоновки
`ImageIcon`	Класс для создания объекта изображения
`JButton`	Класс для создания кнопки
`JCheckBox`	Класс для создания опции (компонент, для которого можно установить или отменить флажок)
`JCheckBoxMenuItem`	Класс для создания пункта меню опционного типа
`JColorChooser`	Класс для реализации выбора цвета
`JComboBox`	Класс для создания раскрывающегося списка
`JDialog`	Класс для создания диалогового окна

Таблица 12.2 (*продолжение*)

Класс	Описание
`JEditorPane`	Класс для создания текстового компонента с возможностями редактирования
`JFileChooser`	Класс для реализации выбора файлов
`JFormattedTextField`	Класс для создания текстового поля с расширенными возможностями
`JFrame`	Класс для создания окна
`JLabel`	Класс для создания текстовой метки
`JList`	Класс для создания списка выбора
`JMenu`	Класс для реализации меню
`JMenuBar`	Класс для создания панели меню
`JMenuItem`	Класс для создания пункта меню
`JOptionPane`	Класс со статическими методами для отображения диалоговых окон
`JPanel`	Класс для создания панели
`JPasswordField`	Класс для создания поля ввода пароля
`JPopupMenu`	Класс для реализации контекстного меню
`JProgressBar`	Класс для реализации компонента, являющегося индикатором некоторого процесса
`JRadioButton`	Класс для создания переключателя
`JRadioButtonMenuItem`	Класс для создания пункта меню, который является переключателем
`JScrollBar`	Класс для создания полосы прокрутки
`JScrollPane`	Класс для создания контейнера с полосами прокрутки
`JSeparator`	Класс для реализации разделителя пунктов меню (декоративный элемент)
`JSlider`	Класс для создания слайдера (компонент со шкалой и ползунком)
`JSpinner`	Класс для создания спиннера (поле со значением и пиктограммы со стрелками, позволяющими изменять значение в поле)
`JSplitPane`	Класс для создания панели с разделителем (панель разделена на две части)
`JTabbedPane`	Класс для создания панели со вкладками

Класс	Описание
`JTable`	Класс для работы с таблицами
`JTextArea`	Класс для создания текстовой области
`JTextField`	Класс для создания текстового поля
`JTextPane`	Класс для создания текстовой панели
`JToggleButton`	Класс для создания кнопки опционного типа
`JToolBar`	Класс для создания панели инструментов
`JToolTip`	Класс используется для создания всплывающих подсказок для компонентов
`JTree`	Класс для работы с деревьями
`JWindow`	Класс для создания окна без меню, заголовка и прочих атрибутов
`OverlayLayout`	Класс для создания менеджера компоновки
`ScrollPaneLayout`	Класс для создания менеджера компоновки, используемого в панели с полосами прокрутки

Как видим, классов достаточно много. Некоторые из них использовались ранее или еще будут использоваться в рассматриваемых далее примерах.

Как неоднократно отмечалось, наиболее важная часть кода для приложений с графическим интерфейсом связана с обработкой событий. Более того, обработка событий является, пожалуй, краеугольным камнем любого графического приложения.

Взаимодействие пользователя с графическим интерфейсом реализуется через обработку *событий*. Под событием подразумевают некоторое действие, производимое с графическим компонентом: например, нажатие кнопки, изменение размеров окна, ввод символа в поле и так далее. Компонент, с которым произошло событие, называется *источником события*. Все возможные события разбиты на классы. Или, другими словами, есть набор специальных классов, каждый из которых соответствует событию определенного типа (в общем филологическом смысле этого слова). Классы, которые соответствуют событиям, имеют в своем названии слово `"Event"` (то есть *событие*). Например, `ActionEvent` (событие, связанное с нажатием кнопки или другим действием, специфичным для данного компонента), `KeyEvent` (событие, связанное, с действиями клавиатуры) или `MouseEvent` (событие, связанное с манипуляциями мышью). Основная часть классов событий собрана в пакете `java.awt.event`. Кроме этого, некоторые классы событий есть в пакете `javax.swing.event`.

ПОДРОБНОСТИ

Для каждого компонента есть набор событий, на которые компонент может реагировать.

Когда происходит определенное событие, автоматически создается *объект события*. Объект события создается на основе класса, соответствующего типу события, и содержит информацию о событии, включая данные о компоненте, который стал источником события. Событие может быть обработано компонентом, и в таком случае созданный объект события передается для обработки, а в компоненте следует зарегистрировать специальный объект-обработчик. Объект-обработчик должен содержать определенные методы, которые автоматически вызываются в процессе обработки события. Чтобы гарантировать наличие у объекта нужных методов, объект — обработчик создается на основе не любого класса, а такого, который реализует определенный интерфейс. Для каждого события интерфейс свой. Но если мы знаем класс события, то по названию этого класса можно определить название интерфейса, который должен быть реализован в классе, на основе которого создается объект-обработчик данного события. Для этого в названии класса события следует слово `"Event"` заменить на `"Listener"`. Например, обработчик события класса `ActionEvent` создается на основе класса, реализующего интерфейс `ActionListener`. Обработчик события класса `KeyEvent` создается на основе класса, реализующего интерфейс `KeyListener`. Для события класса `WindowEvent` соответствующий интерфейс называется `WindowListener`, и так далее.

Для регистрации объекта-обработчика в компоненте из объекта компонента вызывается специальный метод, аргументом которому передается регистрируемый объект-обработчик. Название данного метода можно определить по названию интерфейса, реализованного в классе, на основе которого создавался объект-обработчик. Для этого к названию интерфейса следует добавить слово `"add"`. Также для определения названия метода можно использовать название класса события, для которого регистрируется обработчик: в названии класса события добавляется слово `"add"`, а слово `"Event"` заменяется на `"Listener"`. Скажем, для регистрации обработчика события класса `WindowEvent` используется метод `addWindowListener()`. При этом сам обработчик создается на основе класса, реализующего интерфейс `WindowListener`. Еще пример: с помощью метода `addActionListener()` регистрируется обработчик для событий класса `ActionEvent`, а объект-обработчик создается на основе класса, реализующего интерфейс `ActionListener`.

ПОДРОБНОСТИ

Если класс реализует интерфейс, то в этом классе должны быть описаны все методы из интерфейса. В контексте обработки событий, на практике обычно из интерфейса используется лишь несколько методов. Поэтому иногда приходится лишние методы описывать с пустым телом. Вместе с тем для многих интерфейсов, связанных с обработкой событий, есть специальные классы-адаптеры, которые реализуют все методы из соответствующего интерфейса с пустым телом. В таком

случае объект-обработчик можно создать на основе подкласса для класса-адаптера, переопределив в этом подклассе только тот метод (методы), который (или которые) предполагается использовать для обработки событий. Название класса-адаптера можно узнать, заменив в названии интерфейса слово `"Listener"` на слово `"Adapter"`. Например, для интерфейса `KeyListener` класс-адаптер называется `KeyAdapter`, а для интерфейса `MouseListener` класс-адаптер называется `MouseAdapter`. Для интерфейса `ActionListener` класса-адаптера нет (поскольку в интерфейсе всего один метод).

НА ЗАМЕТКУ

При обработке события объект события передается в объект-обработчик. Достигается это за счет передачи аргумента методам из интерфейса, реализованного в классе объекта-обработчика. Если речь идет об обработке события определенного класса (например, `MouseEvent`), то методам из интерфейса (в данном случае `MouseListener`) аргументом передается объект класса `MouseEvent`. Это и есть объект события.

Интерфейсы, реализуемые в классах для объектов-обработчиков, и классы-адаптеры размещены в пакетах `java.awt.event` и `javax.swing.event`.

Классы некоторых событий из библиотеки AWT представлены в табл. 12.3.

Таблица 12.3. Некоторые классы событий из библиотеки AWT

Класс	Описание события
`ActionEvent`	Событие, специфичное для конкретного компонента (например, нажатие кнопки)
`AdjustmentEvent`	Событие, связанное с изменением состояния регулируемого элемента (перемещение ползунка для полосы прокрутки)
`ComponentEvent`	Событие, связанное с перемещением компонента, изменением его размеров, отображением и скрытием компонента
`ContainerEvent`	Событие, связанное с изменением содержимого компонента-контейнера
`FocusEvent`	Событие, связанное с получением или потерей компонентом фокуса
`ItemEvent`	Событие, связанное установкой (выбором) или отменой установки (отменой выбора) таких элементов, как списки, опции и переключатели
`KeyEvent`	Событие, связанное с вводом символа, нажатием и отпусканием клавиши

Таблица 12.3 (*окончание*)

Класс	Описание события
`MouseEvent`	Событие, связанное с манипуляциями мышью (например, нажатие кнопки мыши или перемещение курсора)
`MouseWheelEvent`	Событие, связанное с вращением колесика мыши
`PaintEvent`	Событие, связанное с перерисовкой компонента
`TextEvent`	Событие, связанное с изменением текста в текстовых компонентах
`WindowEvent`	Событие, связанное с операциями с окном (например, открытие, закрытие окна или сворачивание и разворачивание окна)

Некоторые классы событий из библиотеки Swing можно найти в табл. 12.4.

Таблица 12.4. Некоторые классы событий из библиотеки Swing

Класс	Описание события
`ChangeEvent`	Событие, связанное с изменением состояния компонента
`HyperlinkEvent`	Событие, связанное с гиперссылкой
`ListSelectionEvent`	Событие, связанное с изменением выбранного значения
`MenuEvent`	Событие, связанное с действиями с меню
`PopupMenuEvent`	Событие, связанное с контекстным меню
`UndoableEditEvent`	Событие, связанное с неотменяемой операцией

Далее рассмотрим несколько примеров, в которых используются классы компонентов и классы событий. Поскольку эти примеры учебные, некоторые задействованные в них механизмы имеют сугубо обучающий и поясняющий характер.

Создание графика функции

— По-вашему, это неинтересно?

— Интересно. Для любителей древности.

из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

В представленном далее примере используются средства библиотеки AWT. Процесс создания приложения с не очень сложным графическим интерфейсом рассмотрим

на примере программы, предназначенной для отображения графика функции (при положительных значениях аргумента x функции):

$$y(x) = (1 + \sin(x)) / (1 + |x|).$$

Левая граница диапазона отображения графика — нулевая; правая определяется в окне программы. Сама функция возвращает значения в диапазоне от `0` до `1`, что с прикладной точки зрения достаточно удобно. При построении графика предусматривается возможность выполнять некоторые дополнительные настройки. Код примера представлен в листинге 12.7.

Листинг 12.7. График функции

```
// Подключение пакетов:
import java.awt.*;
import java.awt.event.*;
// Класс окна:
class PlotFrame extends Frame{
   // Конструктор (аргументы – высота и ширина окна):
   PlotFrame(int H,int W){
      // Название окна:
      setTitle("График функции");
      // Положение и размеры окна:
      setBounds(100,50,W,H);
      // Цвет фона окна:
      setBackground(Color.GRAY);
      // Отключение менеджера компоновки:
      setLayout(null);
      // Объект шрифта:
      Font f=new Font("Arial",Font.BOLD,11);
      // Применение шрифта:
      setFont(f);
      // Окно фиксированных размеров:
      setResizable(false);
      // Пиктограмма для окна:
      setIconImage(
         getToolkit().getImage(
            "D:/Pictures/Icons/icon.png"
         )
      );
      // Создание панели с кнопками:
      ButtonPanel BtnPnl=new ButtonPanel(6,25,W/4,H-30);
      // Добавление панели в окно:
      add(BtnPnl);
      // Создание панели для отображения графика:
      PlotPanel PltPnl=new PlotPanel(
         W/4+10,25,3*W/4-15,H-120,BtnPnl
      );
      // Добавление панели в окно:
      add(PltPnl);
      // Панель для отображения справки:
```

```
      HelpPanel HlpPnl=new HelpPanel(
         W/4+10,H-90,3*W/4-15,85
      );
      // Добавление панели в окно:
      add(HlpPnl);
      // Регистрация обработчика для окна:
      addWindowListener(new WindowAdapter(){
         public void windowClosing(WindowEvent we){
            // Завершение выполнения программы:
            System.exit(0);
         }
      });
      // Регистрация обработчика для первой кнопки:
      BtnPnl.B1.addActionListener(
         new ButtonOneHandler(BtnPnl,PltPnl)
      );
      // Регистрация обработчика для второй кнопки:
      BtnPnl.B2.addActionListener(new ButtonTwoHandler());
      // Регистрация обработчика для опции
      // отображения сетки:
      BtnPnl.Cbx[3].addItemListener(
         new CheckboxHandler(BtnPnl)
      );
      // Отображение окна:
      setVisible(true);
   }
}
// Класс панели с кнопками:
class ButtonPanel extends Panel{
   // Массив меток:
   public Label[] L;
   // Группа переключателей:
   public CheckboxGroup CbxGrp;
   // Массив переключателей:
   public Checkbox[] Cbx;
   // Раскрывающийся список:
   public Choice Chc;
   // Текстовое поле:
   public TextField TxtFld;
   // Кнопки:
   public Button B1,B2;
   // Конструктор (аргументы – координаты
   // и размеры панели):
   ButtonPanel(int x,int y,int W,int H){
      // Отключение менеджера компоновки:
      setLayout(null);
      // Положение и размер панели:
      setBounds(x,y,W,H);
      // Цвет фона панели:
      setBackground(Color.LIGHT_GRAY);
      // Создание массива меток:
      L=new Label[3];
      // Первая текстовая метка:
```

```
L[0]=new Label("Выбор цвета:",Label.CENTER);
// Шрифт для текстовой метки:
L[0].setFont(new Font("Arial",Font.BOLD,12));
// Размеры метки:
L[0].setBounds(5,5,getWidth()-10,30);
// Добавление метки на панель:
add(L[0]);
// Создание группы переключателей:
CbxGrp=new CheckboxGroup();
// Создание массива переключателей:
Cbx=new Checkbox[4];
// Создание переключателей:
Cbx[0]=new Checkbox(" красный ",CbxGrp,true);
Cbx[1]=new Checkbox(" синий ",CbxGrp,false);
Cbx[2]=new Checkbox(" черный ",CbxGrp,false);
// Создание опции отображения сетки:
Cbx[3]=new Checkbox(" Сетка ",true);
// Задаем размеры и добавляем переключатели
// и опцию на панель:
for(int i=0;i<4;i++){
   Cbx[i].setBounds(5,30+i*25,getWidth()-10,30);
   add(Cbx[i]);
}
// Раскрывающийся список для определения
// цвета линий сетки:
Chc=new Choice();
// Добавление элементов в список:
Chc.add("Зеленый");
Chc.add("Желтый");
Chc.add("Серый");
// Размеры и положение раскрывающегося списка:
Chc.setBounds(20,140,getWidth()-25,30);
// Добавление списка на панель:
add(Chc);
// Вторая текстовая метка:
L[1]=new Label("Интервал по x:",Label.CENTER);
// Шрифт для метки:
L[1].setFont(new Font("Arial",Font.BOLD,12));
// Размеры и положение метки:
L[1].setBounds(5,220,getWidth()-10,30);
// Добавление метки на панель:
add(L[1]);
// Третья текстовая метка:
L[2]=new Label("От x=0 до x=",Label.LEFT);
// Размеры и положение метки:
L[2].setBounds(5,250,70,20);
// Добавление метки на панель:
add(L[2]);
// Текстовое поле для ввода верхней границы
// диапазона изменения аргумента функции:
TxtFld=new TextField("10");
// Размеры и положение поля:
TxtFld.setBounds(75,250,45,20);
```

```
      // Добавление поля на панель:
      add(TxtFld);
      // Первая кнопка ("Нарисовать"):
      B1=new Button("Нарисовать");
      // Вторая кнопка ("Закрыть"):
      B2=new Button("Закрыть");
      // Размеры и положение первой кнопки:
      B1.setBounds(5,getHeight()-75,getWidth()-10,30);
      // Размеры и положение второй кнопки:
      B2.setBounds(5,getHeight()-35,getWidth()-10,30);
      // Добавление первой кнопки на панель:
      add(B1);
      // Добавление второй кнопки на панель:
      add(B2);
   }
}
// Класс панели для отображения графика:
class PlotPanel extends Panel{
   // Ссылка на объект внутреннего класса:
   public Plotter G;
   // Внутренний класс для реализации графика функции:
   class Plotter{
      // Границы диапазона изменения координат:
      private double Xmin=0,Xmax,Ymin=0,Ymax=1.0;
      // Состояние опции отображения сетки:
      private boolean status;
      // Цвет для отображения графика:
      private Color clr;
      // Цвет для отображения линий сетки:
      private Color gClr;
      // Конструктор (аргумент – ссылка на панель
      // с кнопками):
      Plotter(ButtonPanel P){
         // Считывание значения текстового поля
         // и преобразование в число:
         try{
            Xmax=Double.parseDouble(P.TxtFld.getText());
         }
         catch(NumberFormatException e){
            P.TxtFld.setText("10");
            Xmax=10;
         }
         // Определение состояния опции:
         status=P.Cbx[3].getState();
         // Определение цвета линий сетки:
         switch(P.Chc.getSelectedIndex()){
            case 0:
               gClr=Color.GREEN;
               break;
            case 1:
               gClr=Color.YELLOW;
               break;
            default:
```

```
            gClr=Color.GRAY;
      }
      // Цвет линии графика:
      String name=
         P.CbxGrp.getSelectedCheckbox().getLabel();
      if(name.equalsIgnoreCase(" красный ")){
         clr=Color.RED;
      }else{
         if(name.equalsIgnoreCase(" синий ")){
            clr=Color.BLUE;
         }else{
            clr=Color.BLACK;
         }
      }
   }
   // Метод определяет отображаемую
   // на графике функцию:
   private double f(double x){
      return (1+Math.sin(x))/(1+Math.abs(x));
   }
   // Метод для считывания и запоминания настроек:
   public Plotter remember(ButtonPanel P){
      return new Plotter(P);
   }
   // Метод для отображения графика и сетки:
   public void plot(Graphics Fig){
      // Параметры области отображения графика:
      int H,W,h,w,s=20;
      H=getHeight();
      W=getWidth();
      h=H-2*s;
      w=W-2*s;
      // Очистка области графика:
      Fig.clearRect(0,0,W,H);
      // Индексная переменная и количество линий сетки:
      int k,nums=10;
      // Цвет координатных осей – черный:
      Fig.setColor(Color.BLACK);
      // Отображение координатных осей:
      Fig.drawLine(s,s,s,h+s);
      Fig.drawLine(s,s+h,s+w,s+h);
      // Отображение засечек и числовых значений
     // на координатных осях:
     for(k=0;k<=nums;k++){
        Fig.drawLine(s+k*w/nums,s+h,s+k*w/nums,s+h+5);
        Fig.drawLine(s-5,s+k*h/nums,s,s+k*h/nums);
        Fig.drawString(
           Double.toString(Xmin+k*(Xmax-Xmin)/nums),
           s+k*w/nums-5,s+h+15
        );
        Fig.drawString(
           Double.toString(Ymin+k*(Ymax-Ymin)/nums),
           s-17,s+h-1-k*h/nums
```

```
            );
         }
         // Отображение сетки (если установлена опция):
         if(status){
            Fig.setColor(gClr);
            // Отображение линий сетки:
            for(k=1;k<=nums;k++){
               Fig.drawLine(s+k*w/nums,s,s+k*w/nums,h+s);
               Fig.drawLine(
                  s,s+(k-1)*h/nums,s+w,s+(k-1)*h/nums
               );
            }
         }
         // Отображение графика:
         Fig.setColor(clr); // Установка цвета линии
         // Масштаб на один пиксел по каждой из координат:
         double dx=(Xmax-Xmin)/w,dy=(Ymax-Ymin)/h;
         // Переменные для записи декартовых координат:
         double x1,x2,y1,y2;
         // Переменные для записи координат
         // в окне отображения графика:
         int h1,h2,w1,w2;
         // Начальные значения:
         x1=Xmin;
         y1=f(x1);
         w1=s;
         h1=h+s-(int)Math.round(y1/dy);
         // Шаг в пикселах для базовых точек:
         int step=5;
         // Отображение базовых точек
         // и соединение их линиями:
         for(int i=step;i<=w;i+=step){
            x2=i*dx;
            y2=f(x2);
            w2=s+(int)Math.round(x2/dx);
            h2=h+s-(int)Math.round(y2/dy);
            // Линия:
            Fig.drawLine(w1,h1,w2,h2);
            // Базовая точка (квадрат):
            Fig.drawRect(w1-2,h1-2,4,4);
            // Новые значения для координат:
            x1=x2;
            y1=y2;
            w1=w2;
            h1=h2;
         }
      }
   }
   // Конструктор (аргументы – координаты и размеры
   // панели, а также ссылка на панель с кнопками):
   PlotPanel(int x,int y,int W,int H,ButtonPanel P){
      // Создание объекта внутреннего класса:
```

```
      G=new Plotter(P);
      // Цвет фона панели:
      setBackground(Color.WHITE);
      // Размеры и положение панели:
      setBounds(x,y,W,H);
   }
   // Переопределение метода для перерисовки панели:
   public void paint(Graphics g){
      // При перерисовке панели вызывается метод
      // для отображения графика:
      G.plot(g);
   }
}
// Класс для панели справки:
class HelpPanel extends Panel{
   // Метка:
   public Label L;
   // Текстовая область:
   public TextArea TxtA;
   // Конструктор (аргументы – координаты
   // и размеры панели):
   HelpPanel(int x,int y,int W,int H){
      // Цвет фона панели:
      setBackground(Color.LIGHT_GRAY);
      // Размеры и положение панели:
      setBounds(x,y,W,H);
      // Отключения менеджера компоновки:
      setLayout(null);
      // Метка для панели:
      L=new Label("СПРАВКА",Label.CENTER);
      // Размеры и положение метки:
      L.setBounds(0,0,W,20);
      // Добавление метки на панель:
      add(L);
      // Текстовая область для панели:
      TxtA=new TextArea(
         " График функции y(x)=(1+sin(x))/(1+|x|)"
      );
      // Шрифт для текстовой области:
      TxtA.setFont(new Font("Serif",Font.PLAIN,15));
      // Размер и положение текстовой области:
      TxtA.setBounds(5,20,W-10,60);
      // Область недоступна для редактирования:
      TxtA.setEditable(false);
      // Добавление текстовой области на панель:
      add(TxtA);
   }
}
// Класс обработчика для первой кнопки:
class ButtonOneHandler implements ActionListener{
   // Панель с кнопками:
   private ButtonPanel P1;
```

```
   // Панель для отображения графика:
   private PlotPanel P2;
   // Конструктор класса (аргументы – ссылки на панели):
   ButtonOneHandler(ButtonPanel P1,PlotPanel P2){
      this.P1=P1;
      this.P2=P2;
   }
   // Метод для обработки нажатия кнопки:
   public void actionPerformed(ActionEvent ae){
      // Обновление параметров (настроек)
      // для отображения графика:
      P2.G=P2.G.remember(P1);
      // Рисование графика:
      P2.G.plot(P2.getGraphics());
   }
}
// Класс обработчика для второй кнопки:
class ButtonTwoHandler implements ActionListener{
   // Метод для обработки нажатия кнопки:
   public void actionPerformed(ActionEvent ae){
      // Завершение выполнения программы:
      System.exit(0);
   }
}
// Класс обработчика для опции отображения сетки:
class CheckboxHandler implements ItemListener{
   // Список выбора цвета для сетки:
   private Choice ch;
   // Конструктор (аргумент – ссылка на панель
   // с кнопками):
   CheckboxHandler(ButtonPanel P){
      this.ch=P.Chc;
   }
   // Метод для обработки изменения состояния опции:
   public void itemStateChanged(ItemEvent ie){
      ch.setEnabled(
         ie.getStateChange()==ItemEvent.SELECTED
      );
   }
}
// Главный класс:
class Demo{
   public static void main(String args[]){
      // Создание (и отображение) окна:
      new PlotFrame(400,500);
   }
}
```

При запуске программы открывается окно (рис. 12.10), состоящее из трех панелей.

Первая панель с кнопками и другими элементами управления расположена в левой части главного окна и занимает примерно четверть его ширины. Правую часть

главного окна занимают две панели: большая белая вверху предназначена для отображения графика, а панель поменьше внизу содержит краткую справку (фактически выражение для отображаемой на графике функции). Для каждой из этих трех панелей создается собственный класс (путем наследования класса `Panel`).

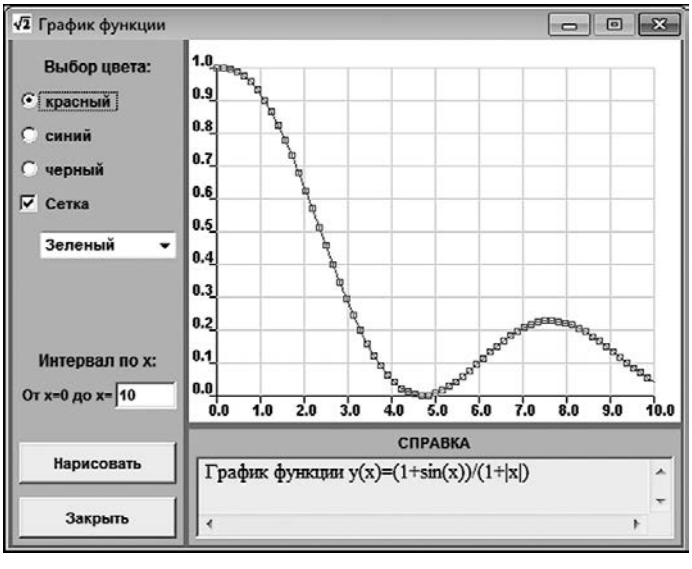


Рис. 12.10. Окно приложения с графиком функции

Вся функциональность приложения спрятана в компоненты, размещенные на первой панели (в левой части главного окна). Эта панель, в частности, содержит *группу переключателей* Выбор цвета с тремя *переключателями* (красный, синий и зеленый), предназначенными для выбора цвета, которым отображается график функции. Также там есть *опция* Сетка, установив которую можно включить режим отображения координатной сетки. При установленном флажке опции снизу доступен раскрывающийся список с пунктами Зеленый, Желтый и Серый для выбора цвета линий сетки. Если флажок опции Сетка не установлен, то список недоступен.

В поле в нижней части панели вводится значение для верхней границы диапазона изменения аргумента функции. По умолчанию значение в поле равно `10`. Кнопка Нарисовать предназначена для отображения графика при установленных настройках, а кнопка Закрыть — для закрытия окна. На рис. 12.11 показано окно приложения в режиме, когда линий сетки нет, а график функции отображается синим цветом в диапазоне значений аргумента от `0` до `5`.

Панель справки содержит статическую информацию, которая не меняется при выполнении программы.

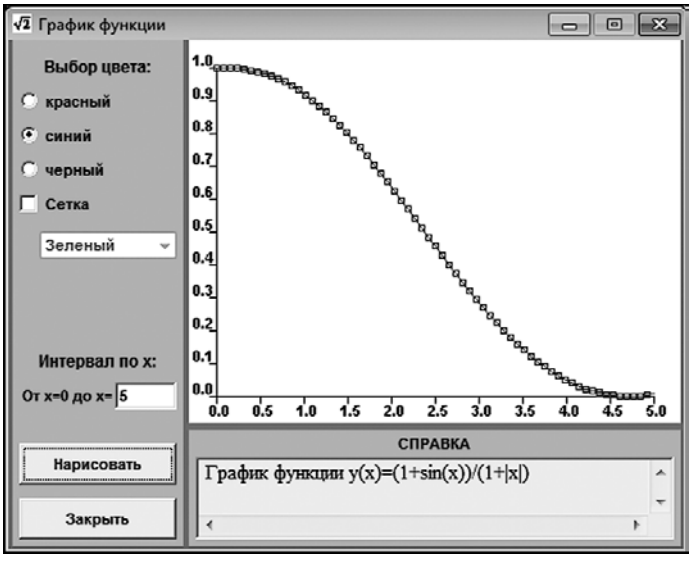


Рис. 12.11. График отображается синим цветом без линий сетки

Далее рассмотрим, как описанные возможности реализованы в программном коде (листинг 12.7). В первую очередь обращаем внимание на то, что программа состоит из нескольких классов, включая класс `PlotFrame`, предназначенный для реализации главного окна программы. Есть специальные классы для каждой из трех панелей: класс `ButtonPanel` для панели с кнопками, класс `PlotPanel` для панели, на которой отображается график, и класс `HelpPanel` для панели справки. Кроме того, используется и несколько вспомогательных классов, на основе которых создаются обработчики событий. В частности, классы `ButtonOneHandler` и `ButtonTwoHandler` предназначены для создания обработчика нажатия первой и второй кнопок соответственно. Класс `CheckboxHandler` нужен для создания обработчика события, связанного с изменением состояния опции. В классе `PlotPanel` описан внутренний класс `Plotter`, который мы используем в процессе отображения графика функции и запоминания настроек. В классе `Demo` описан главный метод программы.

Класс панели `ButtonPanel` создается путем наследования класса `Panel`. Полями класса являются:

- ссылка на массив меток `L`;
- ссылка на группу переключателей `CbxGrp`;
- ссылка на массив `Cbx` компонентов типа `Checkbox` (три из них являются переключателями и принадлежат группе `CbxGrp`, а один — это опция);
- ссылка на раскрывающийся список `Chc`;

- ссылка на текстовое поле `TxtFld`;
- а также ссылка на две кнопки `B1` и `B2`.

Полями являются ссылки на объекты — сами объекты создаются при вызове конструктора. Аргументами конструктору передаются координаты левого верхнего угла панели (по отношению к контейнеру — в данном случае к главному окну, в которое будет добавляться панель), а также ширина и высота панели.

НА ЗАМЕТКУ

Точка начала отсчета координат в окне находится в левом верхнем углу. По горизонтали координата отсчитывается в пикселах слева направо, по вертикали — сверху вниз.

В конструкторе класса `ButtonPanel` командой `setLayout(null)` отключается менеджер компоновки. Размер и положение панели задаются командой `setBounds(x,y,W,H)`. При этом методу `setBounds()` передаются аргументы конструктора. Цвет панели задается командой `setBackground(Color.LIGHT_GRAY)`, то есть в данном случае использовано статическое поле `LIGHT_GRAY` класса `Color` для определения цвета (светло-серый).

Командой `L=new Label[3]` создается массив из трех элементов, а ссылка на этот массив записывается в поле `L`. Элементами массива `L` являются ссылки на метки. Сами метки нужно создавать отдельно. Например, первая метка создается командой `L[0]=new Label("Выбор цвета:",Label.CENTER)`. В метке отображается текст `"Выбор цвета:"`, размещаемый по центру в области метки (использовано статическое поле `CENTER` из класса `Label`). Шрифт для метки задается командой `L[0].setFont(new Font("Arial",Font.BOLD,12))`. В этой команде методу `setFont()` передается анонимный объект класса `Font`, который создается командой `new Font("Arial",Font.BOLD,12)`. Размеры метки задаются командой `L[0].setBounds(5,5,getWidth()-10,30)` с использованием метода `getWidth()` для получения значения ширины панели. Командой `add(L[0])` метка добавляется на панель и предназначена для группы переключателей, которые размещаются под меткой.

Группа переключателей создается командой `CbxGrp=new CheckboxGroup()`. Командой `Cbx=new Checkbox[4]` создается массив из четырех элементов, которые являются ссылками на объекты класса `Checkbox`. Первые три объекта — это переключатели для выбора цвета графика, четвертый — опция для выбора режима отображения сетки. Переключатели группы создаются командами `Cbx[0]=new Checkbox(" красный ",CbxGrp,true)`, `Cbx[1]=new Checkbox(" синий ",CbxGrp,false)` и `Cbx[2]=new Checkbox(" черный ",CbxGrp,false)`. Конструктору класса `Checkbox()` передаются соответственно отображаемый возле переключателя текст, группа, к которой принадлежит переключатель, и состояние переключателя (`true`, если переключатель установлен, и `false`, если не установлен). Таким образом, при создании окна установлен переключатель, отвечающий

за отображение графика красным цветом. Опция создается командой `Cbx[3]=new Checkbox(" Сетка ",true)`. Поскольку опция к группе переключателей не принадлежит, второй аргумент пропускается — указываются только текст возле опции и ее состояние (установлена или не установлена). По умолчанию используется режим отображения координатной сетки. Поскольку три переключателя и опция располагаются упорядоченно, один под другим, процедура определения размера и положения переключателей и опции, а также добавление их на панель реализуются с помощью оператора цикла.

Объект для раскрывающегося списка (изначально незаполненного) создается командой `Chc=new Choice()`. Добавление пунктов в список осуществляется командами `Chc.add("Зеленый")`, `Chc.add("Желтый")` и `Chc.add("Серый")`. Пункты добавляются в список один за другим в соответствии с порядком следования команд. Размер и положение списка на панели задается командой `Chc.setBounds(20,140,getWidth()-25,30)`. Левый верхний угол элемента списка находится на расстоянии `20` пикселов вправо от левого верхнего угла панели и на `140` пикселов вниз. Высота поля составляет `30` пикселов, а ширина на `25` пикселов меньше ширины панели. Добавляется список на панель командой `add(Chc)`.

Далее добавляются еще две текстовые метки `L[1]` и `L[2]` (надписи `"Интервал по x"` и `"От x=0 до x="`). Метки добавляются практически так же, как и первая метка на панели, поэтому хочется верить, что особых комментариев эта часть кода не требует.

Текстовое поле создается командой `TxtFld=new TextField("10")`. При вызове конструктора класса `TextField` аргументом указано текстовое значение `"10"`. Именно оно отображается в поле по умолчанию. Размеры и положение поля на панели задаются командой `TxtFld.setBounds(75,250,45,20)` (поле шириной `45` пикселов и высотой `20` пикселов расположено на `75` пикселов вправо и `250` пикселов вниз от верхнего левого угла панели), а добавляется поле на панель с помощью команды `add(TxtFld)`.

Наконец, две кнопки (объекты класса `Button`) создаются командами `B1=new Button("Нарисовать")` и `B2=new Button("Закрыть")`. Текст, предназначенный для отображения на кнопках, передается аргументом конструктору класса `Button`. Размеры и положение кнопок определяются вызовом метода `setBounds()`, а добавляются кнопки на панель с помощью метода `add()`.

Достаточно прост класс `HelpPanel` для панели справки. Он создается на основе класса `Panel` и имеет метку (ссылка `L` класса `Label`) и *текстовую область* (ссылка `TxtA` класса `TextArea`). Соответствующие объекты создаются в конструкторе посредством методов и приемов, описывавшихся ранее. Обращаем внимание, что для текстовой области командой `TxtA.setEditable(false)` устанавливается режим, не позволяющий редактировать ее содержимое. Таким образом, компоненты справочной панели (метка и текстовая область) являются статическими.

Класс `PlotPanel` для панели вывода графика создается путем наследования класса `Panel` и имеет только одно поле `G` — объект класса `Plotter`. Это — внутренний

класс. Через него реализована процедура рисования графика функции, поэтому остановимся на нем подробнее.

У класса четыре поля типа `double`: `Xmin` (значение `0`), `Xmax`, `Ymin` (значение `0`) и `Ymax` (значение `1.0`). Эти поля определяют границы диапазонов значений по каждой из координатных осей. Три поля имеют начальные значения и в процессе выполнения программы не меняются. Значение поля `Xmax` вычисляется на основе состояния компонентов окна приложения. Поле логического типа `status` предназначено для записи состояния опции, определяющей режим отображения сетки. Поля `clr` и `gClr` являются объектами класса `Color` и предназначены для записи значений цвета графика функции и цвета линий сетки соответственно.

Конструктору класса `Plotter` передается ссылка на объект класса `ButtonPanel` — то есть на объект панели, содержащей элементы управления. В конструкторе считывается состояние элементов управления на панели (ссылка на которую передана аргументом конструктору), в частности значение текстового поля со значением верхней границы для диапазона изменения аргумента функции. При этом учитываем, что в поле может быть введено (по ошибке или специально) не число. Поэтому соответствующий фрагмент кода заключен в блок `try`. Командой `Xmax=Double.parseDouble(P.TxtFld.getText())` присваивается значение полю `Xmax`: сначала методом `getText()` объекта поля `TxtFld` панели `P` считывается содержимое поля, а затем методом `parseDouble()` класса-оболочки `Double` текстовое представление числа преобразуется в формат `double`. При некорректном значении в поле возникает ошибка (исключение класса `NumberFormatException`). Это исключение обрабатывается в блоке `catch` — командой `P.TxtFld.setText("10")` в поле заносится значение `10`. Также командой `Xmax=10` соответствующее значение присваивается полю `Xmax`.

Полю `status` значение присваивается командой `status=P.Cbx[3].getState()`. В этом случае использован метод `getState()` объекта опции `Cbx[3]` (который, в свою очередь, является полем объекта панели `P`). В результате поле `status` получает значение `true`, если флажок опции установлен, и `false`, если не установлен.

Инструкция `P.Chc.getSelectedIndex()` вычисляет индекс выбранного пункта в раскрывающемся списке `Chc`. Это значение используется в операторе `switch` для определения цвета линий координатной сетки. Индексация элементов раскрывающегося списка начинается с нуля. Индекс `0` соответствует зеленому цвету (значение `Color.GREEN`), индекс `1` — желтому цвету (значение `Color.YELLOW`), а индекс `2` — серому цвету (значение `Color.GRAY`). Соответствующее значение записывается в переменную `gClr`. Несколько по-иному определяется цвет для графика функции. Для этого командой `String name=P.CbxGrp.getSelectedCheckbox().getLabel()` объявляется текстовая переменная `name` и в качестве значения ей присваивается текст установленного пользователем переключателя. Объект этого переключателя в группе `CbxGrp` возвращается методом `getSelectedCheckbox()`. Из этого объекта вызывается метод `getLabel()`, которым в качестве значения возвращается текст для переключателя. Затем с помощью вложенных условных операторов проверяется

считанное значение и полю `clr` присваивается соответствующее значение (`Color.RED`, `Color.BLUE` или `Color.BLACK`).

Также в классе `Plotter` описан метод `f()` с одним аргументом типа `double`. Этот метод определяет функциональную зависимость, отображаемую на графике.

Аргументом методу `remember()` передается объект класса `ButtonPanel`. Метод возвращает ссылку на объект класса `Plotter`. Этот объект содержит информацию о настройках элементов управления на панели, переданной в качестве аргумента методу. Мы используем метод в тех случаях, когда необходимо запомнить состояние элементов управления панели, чтобы на его основе можно было нарисовать картинку с графиком функции.

Отображение графика функции и сопутствующих ему атрибутов реализуется с помощью метода `plot()` из класса `Plotter`. Аргументом метода является объект `Fig` класса `Graphics`. Это объект *графического контекста* (объект, с помощью которого реализуется графическое представление компонента). Для получения доступа к объекту графического контекста компонента (в данном случае панели) используется метод `getGraphics()` этого компонента.

Командами `H=getHeight()` и `W=getWidth()` определяются размеры панели, в которой будет отображаться график. Командами `h=H-2*s` и `w=W-2*s` определяются фактические размеры области отображения графика (при этом переменная `s` определяет ширину поля вокруг графика функции).

Командой `Fig.clearRect(0,0,W,H)` выполняется очистка области панели. Это необходимо делать для того, чтобы при выводе графика старое изображение убиралось. Метод вызывается из объекта графического контекста компонента, а аргументами ему передаются координаты левой верхней точки области очистки, ее ширина и высота.

В процессе отображения графика сначала рисуются координатные оси, но предварительно командой `Fig.setColor(Color.BLACK)` устанавливается черный цвет линий. Сами оси отображаются командами `Fig.drawLine(s,s,s,h+s)` и `Fig.drawLine(s,s+h,s+w,s+h)`. Линии (точнее, отрезки прямых) рисуются с помощью метода `drawLine()`, аргументами которому передаются координаты начальной и конечной точек отрезка. Засечки и текстовые подписи отображаются в операторе цикла. Вывод текста в графическом режиме осуществляется методом `drawString()`. Аргументом методу передается отображаемый текст и координаты для вывода этого текста. Для преобразования действительных чисел в формат текстовой строки используется метод `toString()` класса `Double`. Переменная `nums` определяет количество линий сетки.

Если установлен флажок для опции отображения сетки (значение переменной `status` равно `true`), то рисуется еще и сетка. Для этого командой `Fig.setColor(gClr)` задается цвет линий сетки, а затем в операторе цикла прорисовываются линии сетки.

Цвет для отображения графика задается командой `Fig.setColor(clr)`. График строится по базовым точкам. Расстояние по горизонтальной оси (в пикселах) между базовыми точками определяется переменной `step`. Эти точки соединяются линиями, а также выделяются квадратами. В последнем случае вызывается метод `drawRect()`, аргументами которому придаются координаты левой верхней точки отображаемого прямоугольника и его размеры (ширина и высота). На этом описание внутреннего класса `Plotter` завершается.

Конструктору класса-контейнера `PlotPanel` в качестве аргументов передаются координаты верхней левой точки панели в окне, ее размеры (ширина и высота), а также ссылка `P` на объект панели с элементами управления. В конструкторе командой `G=new Plotter(P)` создается новый объект класса `Plotter`, и ссылка на него записывается в поле `G`. Белый цвет фона устанавливается командой `setBackground(Color.WHITE)`. Положение и размеры панели определяются с помощью метода `setBounds()`, аргументом которому передаются первые четыре аргумента конструктора.

Также в классе `PlotPanel` переопределяется метод `paint()`. Этот метод автоматически вызывается при перерисовке компонентов, например при разворачивании свернутого окна. Если метод не переопределить, то в конечном варианте программы сворачивание или разворачивание окна будет приводить к исчезновению графика функции. Аргументом методу передается ссылка на объект графического контекста компонента. В данном случае метод переопределен так, что при его вызове выполнятся команда `G.plot(g)` (здесь `g` — аргумент метода `paint()`), в результате чего отображается график функции.

Класс окна `PlotFrame` создается на основе класса `Frame`. Конструктору класса передаются два числа, `W` и `H`, — размеры окна. Собственно, весь класс состоит из конструктора. В частности, командой `setTitle("График функции")` в конструкторе задается название окна (отображается в строке названия окна). Положение окна на экране и его размеры задаются командой `setBounds(100,50,W,H)`. Серый цвет фона устанавливается с помощью команды `setBackground(Color.GRAY)`, а менеджер компоновки отключается командой `setLayout(null)`. Командой `Font f=new Font("Arial",Font.BOLD,11)` создается объект шрифта, и командой `setFont(f)` этот шрифт применяется для окна. Также в окно добавляются (а сначала создаются) три панели. Панель с кнопками создается командой `ButtonPanel BtnPnl=new ButtonPanel(6,25,W/4,H-30)`. Добавляется в окно панель командой `add(BtnPnl)`. Панель для отображения графика создается командой `PlotPanel PltPnl=new PlotPanel(W/4+10,25,3*W/4-15,H-120,BtnPnl)`, а добавляется в окно командой `add(PltPnl)`. Панель для вывода справки создается командой `HelpPanel HlpPnl=new HelpPanel(W/4+10,H-90,3*W/4-15,85)`, а добавляется в окно командой `add(HlpPnl)`.

Мы создаем окно фиксированных размеров, поэтому соответствующий режим задаем командой `setResizable(false)`. Также мы применяем собственную пиктограмму для окна (отображается в левом верхнем углу окна в строке названия) посредством команды `setIconImage(getToolkit().getImage("D:/Pictures/Icons/icon.png"))`. В этой команде аргументом методу `setIconImage()` передается ссыл-

ка на объект изображения (ссылка класса `Image`). Доступ к объекту изображения получаем с помощью метода `getImage()`. Аргументом методу передается текстовая строка с полным путем к файлу с изображением. Вызывается метод из объекта класса `Toolkit`, а доступ к этому объекту (объект *инструментариев* окна) получаем с помощью метода `getToolkit()`.

НА ЗАМЕТКУ

Для корректной работы программы файл `icon.png` с изображением для пиктограммы должен размещаться в директории `D:\Pictures\Icons`. Если файл находится в другом месте, необходимо внести изменения в соответствующую команду, которой для окна определяется пиктограмма.

Отображается окно командой `setVisible(true)`.

Кроме описанных выше настроек, в конструкторе класса окна регистрируются обработчики для таких событий:

- нажатие системной пиктограммы для закрытия окна;
- нажатие кнопок (для каждой из двух кнопок — свой обработчик);
- установка или отмена флажка опции отображения сетки.

Обработчик для окна регистрируется методом `addWindowListener()`. Регистрация обработчика для первой кнопки выполняется командой `BtnPnl.B1.addActionListener(new ButtonOneHandler(BtnPnl,PltPnl))`, регистрация обработчика для второй кнопки — командой `BtnPnl.B2.addActionListener(new ButtonTwoHandler())`, регистрация обработчика для опции — командой `BtnPnl.Cbx[3].addItemListener(new CheckboxHandler(BtnPnl))`. В трех последних случаях обработчики реализуются как анонимные объекты специальных, описанных в программе классов. Код этих классов анализируется далее.

Класс обработчика для первой кнопки `ButtonOneHandler` создается реализацией интерфейса `ActionListener`. У класса есть поля `P1` и `P2`, являющиеся ссылками на объекты панелей (объекты класса `ButtonPanel` и `PlotPanel` соответственно). Такие же аргументы имеет и конструктор класса. Аргументы определяют значения упомянутых полей. Метод для обработки нажатия кнопки `actionPerformed()` содержит команду `P2.G=P2.G.remember(P1)`, которой обновляются параметры, используемые при отображении графика. Фактически командой присваивается новое значение полю `G` панели `P2`. Считывание параметров выполняется с помощью метода `remember()`, который вызывается из объекта `G`, а аргументом методу передается ссылка на объект панели с элементами управления. После этого командой `P2.G.plot(P2.getGraphics())` непосредственно строится график.

Класс обработчика для второй кнопки `ButtonTwoHandler` также создается на основе интерфейса `ActionListener`. Метод `actionPerformed()` определен так, что

выполняется всего одна команда `System.exit(0)`, которой завершается работа программы.

Поскольку от опции качественно зависит, как отображается график (с сеткой или без нее), необходимо предусмотреть реакцию на изменение состояния опции. Для этого реализацией интерфейса `ItemListener` создаем класс `CheckboxHandler`. Он нужен для создания объекта-обработчика событий, связанных с изменением состояния опции. Класс имеет поле `ch`, являющееся ссылкой на объект класса `Choice` (раскрывающийся список), и конструктор, в котором этому полю присваивается значение.

ПОДРОБНОСТИ

Аргументом конструктору передается ссылка на панель с элементами управления. Среди этих элементов есть раскрывающийся список для выбора цвета линий сетки. Ссылка на объект списка присваивается в качестве значения полю `ch`.

В классе `CheckboxHandler` метод `itemStateChanged()`, вызываемый для обработки события, связанного с изменением состояния опции, содержит команду `ch.setEnabled(ie.getStateChange()==ItemEvent.SELECTED)`. В ней для проверки состояния опции использована константа `SELECTED` из класса события `ItemEvent`. Метод `getStateChange()` возвращает в качестве результата значение `SELECTED`, если флажок опции установлен, и `DESELECTED`, если не установлен. Метод вызывается из объекта `ie` события класса `ItemEvent`. Таким образом, значение выражения `ie.getStateChange()==ItemEvent.SELECTED` равно `true`, если флажок опции установлен, и `false` в противном случае. Поэтому при вызове метода `setEnabled()` из объекта `ch` с аргументом `ie.getStateChange()==ItemEvent.SELECTED` доступность раскрывающегося списка определяется тем, установлен флажок опции или нет. Поскольку метод `itemStateChanged()` (после регистрации соответствующего обработчика) вызывается автоматически при изменении состояния опции, то каждый раз при установке или отмене флажка опции соответствующим образом изменяется доступность раскрывающегося списка для выбора цвета линий сетки.

В главном методе программы в классе `Demo` командой `new PlotFrame(400,500)` создается анонимный объект для главного окна программы. Аргументами конструктору класса `PlotFrame()` передаются размеры создаваемого окна.

НА ЗАМЕТКУ

В рассмотренном выше примере задача состоит не только в том, чтобы нарисовать график, но и в том, чтобы запомнить картинку. Последнее нужно на случай перерисовки компонента. В рамках использованного подхода запоминается не сама картинка, а параметры, на основе которых она создавалась. Для этого описывается специальный внутренний класс. При перерисовке компонента картинка отображается заново.

Калькулятор

— Чем желают заняться состоятельные кроты?

— Мы пока посчитаем.

из м/ф «Дюймовочка»

Следующий пример иллюстрирует возможности использования библиотеки Swing для создания приложения с графическим интерфейсом. В данном случае это классика жанра — программа-калькулятор. В учебных целях мы ограничимся одним из самых простых ее вариантов (листинг 12.8).

Листинг 12.8. Калькулятор

```
// Подключение пакетов:
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
// Класс для окна:
class CalculatorFrame extends JFrame{
   // Конструктор:
   CalculatorFrame(){
      // Размеры окна:
      int w=270,h=240;
      // Название окна:
      setTitle("Калькулятор");
      // Определение положения и размеров окна:
      setBounds(100,100,w,h);
      // Реакция на нажатие системной пиктограммы:
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      // Создание панели с кнопками и полем:
      CalcPanel panel=new CalcPanel(w,h);
      // Добавление панели в окно:
      add(panel);
      // Окно фиксированных размеров:
      setResizable(false);
      // Отображение окна:
      setVisible(true);
   }
}
// Класс панели:
class CalcPanel extends JPanel{
   // Текстовое поле:
   public JTextField TxtFld;
   // Обработчик нажатия кнопки:
   private BtnAction BtnPressed;
   // Конструктор (аргументы – размеры панели):
   CalcPanel(int W,int H){
      // Размеры кнопок и отступы:
```

```
    int w=W/5,h=H/8,sx=w/5,sy=h/3;
    // Отключение менеджера компоновки:
    setLayout(null);
    // Рамка вокруг панели:
    setBorder(BorderFactory.createEtchedBorder());
    // Создание текстового поля:
    JTextField TxtFld=new JTextField();
    // Выравнивание текста в поле по правому краю:
    TxtFld.setHorizontalAlignment(JTextField.RIGHT);
    // Положение и размеры поля:
    TxtFld.setBounds(sx,sy,2*sx+3*w,h);
    // Отмена возможности редактирования поля:
    TxtFld.setEditable(false);
    // Добавление поля на панель:
    add(TxtFld);
    // Создание обработчика нажатия кнопки:
    BtnPressed=new BtnAction(TxtFld);
    // Массив с названиями кнопок:
    String[] BtnTxt={
       "1","2","3","+","4","5","6","-",
       "7","8","9","/","0",".","=","*"};
    // Создание кнопок и добавление их на панель:
    for(int i=0;i<BtnTxt.length;i++){
       addBtn(sx+(w+sx)*(i%4),(2*sy+h)+(sy+h)*(i/4),
          w,h,BtnTxt[i],BtnPressed);
    }
    // Создание кнопки сброса параметров:
    JButton BtnC=new JButton("C");
    // Положение и размеры кнопки:
    BtnC.setBounds(4*sx+3*w,sy,w,h);
    // Добавление обработчика для кнопки:
    BtnC.addActionListener(BtnPressed);
    // Режим отмены отображения рамки фокуса:
    BtnC.setFocusPainted(false);
    // Красный цвет для названия кнопки:
    BtnC.setForeground(Color.RED);
    // Добавление кнопки на панель:
    add(BtnC);
}
// Метод для создания и добавления кнопок на панель
// (аргументы – положение и размеры кнопки,
// название и обработчик нажатия кнопки):
void addBtn(int i,int j,int w,int h,
    String txt,
    ActionListener AcList){
    // Создание кнопки:
    JButton b=new JButton(txt);
    // Положение и размеры кнопки:
    b.setBounds(i,j,w,h);
    // Режим отмены отображения рамки фокуса:
    b.setFocusPainted(false);
    // Добавление обработчика для кнопки:
```

```
      b.addActionListener(AcList);
      // Добавление кнопки на панель:
      add(b);
   }
}
// Класс обработчика нажатия кнопки:
class BtnAction implements ActionListener{
   // Текстовое поле для отображения информации:
   public JTextField TxtFld;
   // Индикатор состояния ввода числа:
   private boolean start;
   // Индикатор состояния ввода десятичной точки:
   private boolean point;
   // Текстовое представление последнего
   // введенного оператора:
   private String cmd;
   // Поле для записи промежуточного результата:
   private double result;
   // Метод для сброса параметров:
   private void onStart(){
      start=true;
      point=true;
      cmd="C";
      result=0;
      TxtFld.setText("0.0");
   }
   // Метод для вычисления результата последней операции:
   private void calc(){
      // Введенное в поле число:
      double x;
      x=Double.parseDouble(TxtFld.getText());
      // Вычисление результата:
      if(cmd.equals("*")) result*=x;
      else if(cmd.equals("/")) result/=x;
      else if(cmd.equals("-")) result-=x;
      else if(cmd.equals("+")) result+=x;
      else result=x;
      // Заполнение текстового поля:
      TxtFld.setText(Double.toString(result));
   }
   // Конструктор (аргумент – ссылка на текстовое поле):
   BtnAction(JTextField TxtFld){
      this.TxtFld=TxtFld;
      onStart();
   }
   // Реакция на нажатие кнопки:
   public void actionPerformed(ActionEvent ae){
      // Считывание текста на кнопке:
      String str=ae.getActionCommand();
      // Проверка вариантов:
      if(str.equals("C")){ // Кнопка сброса
         onStart();
```

```
            return;
         }
         // Вычисление результата:
         if(str.equals("+")|str.equals("-")|str.equals("*")|
            str.equals("/")|str.equals("=")){
            calc();
            cmd=str;
            start=true;
            point=true;
            return;
         }
         // Ввод числа:
         if(start){ // Начало ввода числа
            // Ввод точки в начале ввода числа:
            if(str.equals(".")){
               TxtFld.setText("0.");
               point=false;
               start=false;
               return;
            }
            else{ // Ввод цифры в начале ввода числа
               TxtFld.setText(str);
               start=false;
               return;
            }
         }
         else{ // Продолжение ввода числа
         if(str.equals(".")){ // Попытка ввести точку
            str=point?str:"";
            point=false;}
            // Добавление цифры к числу:
            if(TxtFld.getText().equals("0")&!str.equals(".")){
               TxtFld.setText(str);
            }
            else{
               TxtFld.setText(TxtFld.getText()+str);
            }
         }
      }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание (и отображение) окна:
      new CalculatorFrame();
   }
}
```

На рис. 12.12 показано окно, которое отображается при запуске программы.

Калькулятор предназначен для выполнения базовых арифметических операций: сложение, вычитание, умножение и деление. Окно калькулятора содержит упоря-

доченный набор одинаковых по размеру кнопок и поле, используемое для отображения результатов вычислений. Такое окно создается достаточно просто. Трудности связаны с обработкой действий пользователя в окне.

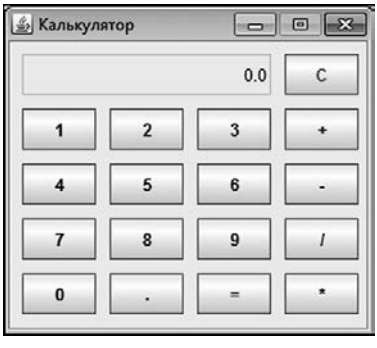


Рис. 12.12. Окно программы-калькулятора

Окно калькулятора реализуется как объект класса `CalculatorFrame`, который создается наследованием класса `JFrame` из библиотеки Swing. Класс состоит фактически из конструктора. В конструкторе командой `setTitle("Калькулятор")` задается название окна. Положение окна и его размеры устанавливаются с помощью метода `setBounds()`. Способ обработки нажатия системной пиктограммы для закрытия окна определяется командой `setDefaultCloseOperation(EXIT_ON_CLOSE)`. Также в конструкторе создается панель — объект класса `CalcPanel` (который, в свою очередь, наследует класс `JPanel`). Методом `add()` панель добавляется в окно. С помощью команды `setResizable(false)` устанавливается режим запрета изменения размеров окна, а само окно отображается посредством команды `setVisible(true)`.

Теперь рассмотрим класс `CalcPanel`, поскольку именно через объект этого класса реализуется содержимое окна приложения. Как уже отмечалось, класс создается наследованием класса `JPanel` из библиотеки Swing. У класса `CalcPanel` определены два поля: открытая ссылка `TxtFld` на объект класса `JTextField` (текстовое поле) и закрытая ссылка `BtnPressed` на объект класса `BtnAction` (класс обработчика события, связанного с нажатием кнопки). Сразу отметим, что хотя в данном случае имеется в общей сложности 17 кнопок, все они используют один и тот же обработчик, и ссылка на этот объект записывается в поле `BtnPressed`. Класс `BtnAction` рассматривается далее.

Конструктору класса панели в качестве аргументов передаются размеры окна (в которое, как предполагается, будет добавлена панель). Командой `setLayout(null)` отключается менеджер компоновки для панели (кнопки размещаются на панели в ручном режиме с явным указанием их положения). Также командой `setBorder(BorderFactory.createEtchedBorder())` для панели создается рамка.

ПОДРОБНОСТИ

Аргументом методу `setBorder()` передается объект класса `Border`, определяющий рамку вокруг компонента (панели в нашем случае). Для получения такого объекта можно использовать статический метод `createEtchedBorder()` из класса `BorderFactory`.

Стоит отметить еще одно важное обстоятельство. Дело в том, что для объекта окна мы не отключаем менеджер компоновки, поэтому панель, при добавлении ее в окно, будет занимать всю рабочую область окна. Как следствие, в конструкторе класса панели мы не задаем положение и размеры панели. Вместе с тем для упрощения расчетов при создании объекта панели мы передаем конструктору класса панели размеры окна, в которое добавляется панель.

В указанных обстоятельствах эффект от отображения рамки вокруг панели минимальный, но все же он есть.

Командой `JTextField TxtFld=new JTextField()` создается объект для текстового поля `TxtFld`. Данное текстовое поле предназначено для отображения результатов вычислений. Для выравнивания текста в поле по правому краю используется инструкция `TxtFld.setHorizontalAlignment(JTextField.RIGHT)`. Положение и размер поля задаются методом `setBounds()`, который вызывается из объекта `TxtFld`. Командой `TxtFld.setEditable(false)` поле переводится в режим, когда его содержимое невозможно редактировать (иначе пришлось бы писать обработчики событий нажатия клавиш на клавиатуре). На панель поле добавляется командой `add(TxtFld)`.

Объект для обработчика события, связанного с нажатием кнопки, создается командой `BtnPressed=new BtnAction(TxtFld)`. Этой же командой ссылка на объект обработчика записывается в поле `BtnPressed`. Аргументом конструктору класса `BtnAction` передается созданное на предыдущем этапе текстовое поле `TxtFld`. В обработчике это поле используется для вывода результатов вычислений и считывания введенных в поле значений (этот процесс описывается далее).

Командой `String[] BtnTxt={"1","2","3","+","4","5","6","-","7","8","9","/","0",".","=","*"}` создается текстовый массив с названиями для кнопок (всех, кроме кнопки сброса). Сами кнопки создаются и размещаются на панели в операторе цикла. Там индексная переменная `i` пробегает значения от `0` до `BtnTxt.length-1` включительно. Основу оператора цикла составляет команда вызова метода `addBtn()` (описание метода приводится далее), аргументами которому передаются координаты новой создаваемой кнопки, ее размеры, название и обработчик события, регистрируемого для кнопки.

НА ЗАМЕТКУ

Обращаем внимание, что ссылки на создаваемые кнопки никуда не записываются. Другими словами, если бы при таком подходе впоследствии понадобилось изменить атрибуты какой-то кнопки, то отловить ее было бы проблематично.

Отдельно создается кнопка сброса параметров, для чего используется команда `JButton BtnC=new JButton("C")`. В данном случае `BtnC` — это локальная переменная, которая создается в конструкторе. После вызова конструктора она удаляется, но кнопка при этом остается. Размеры и положение кнопки задаются вызовом из объекта `BtnC` метода `setBounds()`. Обработчик для кнопки добавляется командой `BtnC.addActionListener(BtnPressed)`. Для того чтобы при активации кнопки для нее не отображалась рамка фокуса, командой `BtnC.setFocusPainted(false)` отключаем этот режим. Кроме того, для кнопки сброса командой `BtnC.setForeground(Color.RED)` устанавливаем красный цвет для отображения названия кнопки. Наконец, добавляется кнопка на панель командой `add(BtnC)`.

Метод `addBtn()` класса `CalcPanel` предназначен для создания и размещения кнопок на панели. Кнопка создается командой `JButton b=new JButton(txt)`, при этом аргументом конструктору класса `JButton` передается текстовое название кнопки — пятый аргумент метода `addBtn()`. Положение и размеры кнопки определяются командой `b.setBounds(i,j,w,h)`. В данном случае мы использовали первые четыре аргумента метода `addBtn()`. Командой `b.setFocusPainted(false)` для кнопки отменяется режим отображения рамки фокуса. Командой `b.addActionListener(AcList)` в кнопке регистрируется обработчик (шестой аргумент метода `addBtn()`). Добавляется кнопка на панель командой `add(b)`.

На этом, собственно, все, что касается организации графического интерфейса как такового, заканчивается. Прочий код относится в основном к реализации взаимодействия между разными элементами интерфейса. Основная нагрузка лежит на методах класса обработчика `BtnAction`, который создается реализацией интерфейса `ActionListener`. В этом классе объявлено текстовое поле `JTextField TxtFld` и ряд закрытых технических полей. В частности, это логическое поле `start`, играющее роль индикатора начала ввода числа в текстовое поле `TxtFld`, логическое поле `point`, используемое в качестве индикатора при вводе десятичной точки, текстовое поле `cmd`, в которое записывается текстовый символ выполняемой операции, и еще поле `result` типа `double`, предназначенное для записи промежуточных результатов вычислений. Также класс имеет несколько закрытых методов, включая метод `onStart()`, используемый для сброса параметров: при вызове метода значение полей `start` и `point` устанавливается равным `true`, поле `cmd` получает значение `"C"`, обнуляется значение поля `result`, а значение в текстовом поле устанавливается равным `"0.0"` (используется команда `TxtFld.setText("0.0")`).

Метод `calc()` предназначен для вычисления результата последней операции. В этом методе командой `x=Double.parseDouble(TxtFld.getText())` значение из текстового поля считывается, преобразуется в формат `double` и записывается в локальную переменную `x`. Далее, в зависимости от того, какой оператор записан в переменную `cmd`, выполняется соответствующее арифметическое действие. Поскольку все реализуемые в данном проекте арифметические операции являются бинарными, нужен, кроме переменной `x`, еще один операнд. Его роль играет поле `result`. Предполагается, что на момент вызова метода `calc()` это поле содержит результат предыдущих вычис-

лений. Оператор, символ которого записан в поле `cmd`, вводится после вычисления этого результата и перед вводом числа, записанного в переменную `x`. В соответствии с этой схемой использована группа вложенных условных операторов, с помощью которых вычисляется результат арифметической операции и в качестве нового значения присваивается полю `result`. После этого командой `TxtFld.setText(Double.toString(result))` полученное значение отображается в текстовом поле. Для преобразования числового значения поля `result` в текстовое представление служит метод `toString()`, который вызывается из класса `Double`.

Конструктор класса `BtnAction` получает в качестве аргумента ссылку на текстовое поле, которая записывается в поле `TxtFld`. После этого выполняется метод `onStart()`, переводящий все поля и индикаторы в начальное состояние.

В классе `BtnAction` описывается метод `actionPerformed()`, которым и определяется реакция каждой из кнопок на нажатие. Первой командой `String str=ae.getActionCommand()` в методе производится считывание текста кнопки, на которую выполнено нажатие.

ПОДРОБНОСТИ

Методом `getActionCommand()` возвращается *команда действия* — по умолчанию это текст, отображаемый на кнопке. Такое поведение по умолчанию можно изменить, задав команду действия в явном виде с помощью метода `setActionCommand()`, который вызывается из объекта кнопки. В данном случае важно то, что объект события (из которого вызывается метод `getActionCommand()`) позволяет определить кнопку, на которой произошло событие. Другими словами, по объекту события мы определяем кнопку, которую нажал пользователь.

Результат (текст кнопки) записывается в текстовую переменную `str`. Далее проверяются разные варианты для конкретных кнопок. Так, если щелчок был произведен на кнопке сброса (условие `str.equals("C")`), то выполняется метод `onStart()` и инструкцией `return` завершается работа метода `actionPerformed()`. Если же щелчок был выполнен на одной из кнопок с символом арифметической операции или знаком равенства (условие `str.equals("+")|str.equals("-")|str.equals("*")|str.equals("/")|str.equals("=")`), то вызывается метод `calc()` (вычисление результата), символ нажатой клавиши (символ оператора) командой `cmd=str` записывается в поле `cmd`, полям `start` и `point` присваивается значение `true`, и работа метода завершается. Значение `true` для поля `start` означает, что далее будет вводиться число, а для поля `point` — что десятичная точка еще не вводилась.

При вводе числа важно разделять начальный этап ввода числа (когда вводится первая цифра) и продолжение ввода. В этом случае используется поле `start`. Так, если число только начинает вводиться (поле `start` имеет значение `true`), предусмотрен случай, когда нажата кнопка с десятичной точкой (условие `str.equals(".")`), и в поле `TxtFld` вводится текст `"0."` (ноль с точкой). Полям `point` и `start` присваивается значение `false`, и работа метода завершается (инструкцией

return). Если же первой вводится цифра (условие str.equals(".") ложно), то она заносится в поле TxtFld командой TxtFld.setText(str), после чего выполняются команды start=false и return.

Ложность условия start (значение переменной start при вызове метода actionPerformed() равно false) означает, что продолжается ввод числа в поле TxtFld. В этом случае при попытке ввести точку значение переменной str переопределяется командой str=point?str:"" и выполняется команда point=false. В результате первой из этих двух команд переменная str не изменит своего значения (то есть "."), если точка еще не вводилась. В противном случае она станет пустой строкой. Таким способом предотвращается попытка ввести больше десятичных точек, чем положено. Кроме того, отслеживается ситуация, когда вводится первый незначащий ноль, то есть когда сначала вводится ноль, а затем цифра. В этом случае первый ноль нужно игнорировать и не отображать его в поле TxtFld. Однако если после нуля вводится точка, то ноль нужно оставить. Поэтому проверяется условие TxtFld.getText().equals("0")&!str.equals("."), которое истинно, если в поле TxtFld записан ноль и следующей вводится не точка. Если это так, то выполняется команда TxtFld.setText(str). В результате первый ноль пропадает с экрана — вместо него вводится цифра (которая, кстати, тоже может быть нулем). Наконец, если это не так, то выполняется команда TxtFld.setText(TxtFld.getText()+str), которая добавляет в конец текстового представления вводимого числа введенную цифру.

В главном методе программы в классе Demo всего одна команда new CalculatorFrame(), которая создает анонимный объект для окна приложения.

Резюме

— Алле, алле, шеф. Вы нас слышите?

— По-моему, он нас не слышит.

из м/ф «Приключения капитана Врунгеля»

- Принцип создания приложений с графическим интерфейсом базируется на том, что для каждого графического компонента (например, кнопки, текстового поля или переключателя) на основе определенного класса создается объект. Есть две библиотеки — AWT и Swing, которые содержат классы для реализации графических компонентов.
- Для каждого компонента есть набор специальных методов, с помощью которых задаются параметры компонентов, а сами компоненты группируются и размещаются в контейнерах вроде окна или панели.
- Обработка событий при реализации графического интерфейса является самой сложной частью.

- Каждому событию соответствует определенный класс. При возникновении события автоматически создается объект этого события. Объект события содержит полезную информацию о специфике события и источнике, вызвавшем его.
- События могут обрабатываться компонентами. Для этого в компоненте регистрируется специальный объект-обработчик. Для регистрации обработчиков в компонентах используют специальные методы. Обработчики для событий разных типов регистрируются с помощью разных методов.
- Объект-обработчик создается на основе класса, который реализует определенный интерфейс. Реализуемый в классе обработчика интерфейс определяется классом обрабатываемого события.
- В классе обработчика описываются методы из соответствующего интерфейса. Каждый из таких методов автоматически вызывается в определенной ситуации (при возникновении определенного события). Методам передается объект события.
- Для многих событий существуют специальные классы-адаптеры, которые реализуют соответствующие интерфейсы так, чтобы методы из интерфейсов определялись с пустым телом. Один из путей создания обработчиков базируется на создании подкласса для класса-адаптера, и в этом подклассе определенным образом описывается нужный метод или методы.

13 Немного о разном

— Какая гадость.

— Это не гадость. Это последние достижения современной науки.

из к/ф «31 июня»

В этой главе мы обсудим вопросы, которые по разным причинам не обсуждались в предыдущих главах, но которые важны для эффективной работы с Java. Тематически эти вопросы весьма разнородны и рассматриваются на соответствующих примерах.

Работа с файлами

Очень убедительно. Мы подумаем, к кому это применить.

из к/ф «31 июня»

Мы уже сталкивались с ситуацией, когда в программу вводятся данные и когда они выводятся программой. Но речь шла о вводе данных через окно с полем ввода или о консольном вводе, а также об отображении сообщений в диалоговом окне или в области вывода. Однако часто возникает необходимость записывать данные в файл и считывать данные из файла. В таких случаях говорят о файловом вводе/выводе.

Ввод и вывод данных в Java реализуется через *потоки ввода/вывода*. В зависимости от того, какими блоками считывается и записывается информация, потоки делятся на *байтовые* и *символьные*.

НА ЗАМЕТКУ

В обоих случаях речь идет о байтах, но поскольку символ — это два байта, то при работе с символьными потоками байты обрабатываются, условно говоря, парами.

Для работы с символьными и байтовыми потоками используются специальные классы из пакета `java.io`. Запись данных в файл и считывание информации из файлов осуществляется достаточно просто: для этого создается объект потока ввода или вывода, связанный с соответствующим файлом. Для создания байтовых файловых потоков используются классы `FileInputStream` (файловый поток ввода) и `FileOutputStream` (файловый поток вывода). Конструкторы этих классов могут сгенерировать контролируемое исключение класса `FileNotFoundException` (ошибка, связанная с тем, что файл не найден). Объект файлового потока создается на основе файла, полное имя которого (в виде текстовой строки) передается аргументом конструктору.

Считывание информации из файла и запись информации в файл реализуются через объект соответствующего потока. Информация считывается и записывается байт за байтом. Для считывания информации используется метод `read()`, а для записи информации в файл — метод `write()`. После завершения работы с файлом поток необходимо закрыть, для чего используют метод `close()`. Методы `read()`, `write()` и `close()` могут генерировать контролируемое исключение класса `IOException` (который является суперклассом для класса `FileNotFoundException`).

ПОДРОБНОСТИ

Признаком окончания файла является значение `-1`, которое в этом случае возвращается методом `read()`.

В первой программе, которую мы рассмотрим (листинг 13.1), выполняется обычное побайтовое копирование файла. При этом мы будем использовать байтовые файловые потоки ввода и вывода.

Листинг 13.1. Байтовые файловые потоки

```
// Импорт классов из пакета:
import java.io.*;
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Целочисленная переменная:
      int a;
      // Отображение сообщения:
      System.out.println("Копирование графического файла");
      // Контролируемый код:
      try{
         // Объект для байтового файлового потока вывода:
         FileOutputStream fout=new FileOutputStream(
            "D:/Files/animal.png"
         );
         // Объект для байтового файлового потока ввода:
         FileInputStream fin=new FileInputStream(
            "D:/Pictures/Animals/panda.png"
```

```
            );
            // Считывание байта из файла:
            a=fin.read();
            // Пока не достигнут конец файла:
            while(a!=-1){
               // Запись байта в файл:
               fout.write(a);
               // Считывание байта из файла:
               a=fin.read();
            }
            // Закрывается поток вывода:
            fout.close();
            // Закрывается поток ввода:
            fin.close();
            // Отображение сообщения:
            System.out.println("Файл скопирован");
            // Обработка исключения:
         }catch(IOException e){
            System.out.println("Ошибка ввода/вывода: "+e);
         }
      }
}
```

Если все проходит штатно, то результат выполнения программы будет таков:

Результат выполнения программы (из листинга 13.1)

```
Копирование графического файла
Файл скопирован
```

Но главный результат выполнения программы связан не с отображением в области вывода, а с копированием графического файла `pand.png` из директории `D:\Pictures\Animals`. В процессе выполнения программы файл копируется в директорию `D:\Files` и имеет название `animal.png`. Для реализации этой задачи в главном методе программы создаются объекты для файлового потока ввода и вывода. Объект `fin` для байтового файлового потока ввода создается на основе класса `FileInputStream`, а аргументом конструктору передается текст `"D:/Pictures/Animals/panda.png"` с названием копируемого файла. Объект `fout` для байтового файлового потока вывода создается на основе класса `FileOutputStream`. Текст `"D:/Files/animal.png"`, переданный конструктору, определяет название файла после копирования (и место его хранения в компьютере).

НА ЗАМЕТКУ

Для использования классов `FileInputStream` и `FileOutputStream` в программе импортируется содержимое пакета `java.io`.

Для записи считываемых байтов мы объявляем целочисленную переменную `a`. Байт из файла, связанного с потоком ввода, считывается инструкцией `fin.read()`.

Результатом этого выражения является значение байта, прочитанного из файла. Командой `a=fin.read()` это прочитанное значение записывается в переменную `a`. Критерием того, что достигнут конец файла, является прочитанное значение `-1`. Поэтому в операторе цикла, с помощью которого выполняется побайтовое копирование файла, проверяется условие `a!=-1`. Пока оно истинно, конец файла не достигнут. Копирование выполняется просто: командой `fout.write(a)` прочитанный байт записывается в файл-копию, а командой `a=fin.read()` считывается новый байт из копируемого файла.

После того как копирование завершено, командами `fout.close()` и `fin.close()` потоки закрываются.

НА ЗАМЕТКУ

Если потоки не закрыть, то это может привести к потере данных, предназначенных для копирования.

Символьный поток отличается от байтового тем, что данные считываются посимвольно (это означает, что данные считываются блоками по два байта). Реализуются байтовые потоки на основе классов `FileReader` и `FileWriter`.

НА ЗАМЕТКУ

Конструкторы классов `FileReader` и `FileWriter` могут генерировать исключение класса `FileNotFoundException`.

Пример использования символьных потоков для копирования (с заменой пробелов на подчеркивания) текстового файла представлен в листинге 13.2.

Листинг 13.2. Символьные файловые потоки

```
import java.io.*;
class Demo{
   public static void main(String[] args){
      int a;
      System.out.println("Начинается копирование файла");
      try{
         // Объект для символьного файлового потока вывода:
         FileWriter fout=new FileWriter(
            "D:/Files/mydata.txt"
         );
         // Объект для символьного файлового потока ввода:
         FileReader fin=new FileReader(
            "D:/Files/base.txt"
         );
         a=fin.read();
         while(a!=-1){
            // Замена пробела на символ подчеркивания:
```

```
            if(a==(int)' ') a=(int)'_';
            fout.write((char)a);
            a=fin.read();
         }
         fout.close();
         fin.close();
         System.out.println("Файл скопирован");
      }catch(IOException e){
         System.out.println("Ошибка ввода/вывода: "+e);
      }
   }
}
```

Если при выполнении ошибок не возникает, то результат будет таким:

Результат выполнения программы (из листинга 13.2)

```
Начинается копирование файла
Файл скопирован
```

Программа достаточно простая и похожа на предыдущую: символ за символом считывается содержимое исходного текстового файла `base.txt` и записывается в файл `mydata.txt` (если такого файла нет, то он создается). При этом все пробелы заменяются символами подчеркивания.

ПОДРОБНОСТИ

Файл `base.txt` должен находиться в директории `D:\Files`. Файл `mydata.txt` создается при выполнении программы и будет находиться в той же директории.

Допустим, содержимое копируемого файла `base.txt` такое:

```
Для изучения Java нужны:
[1] Учебник.
[2] Компьютер.
[3] Программное обеспечение.
[4] Желание учиться.
```

В таком случае содержимое файла `mydata.txt` будет следующим:

```
Для_изучения_Java_нужны:
[1]_Учебник.
[2]_Компьютер.
[3]_Программное_обеспечение.
[4]_Желание_учиться.
```

НА ЗАМЕТКУ

Для корректной работы программы кодировка, в которой сохранен исходный файл `base.txt`, должна соответствовать кодировке, используемой в среде разработки.

Теперь проанализируем особенности выполнения программы. Командой `FileWriter fout=new FileWriter("D:/Files/mydata.txt")` создается объект символьного файлового потока вывода. Объект для символьного файлового потока ввода создается командой `FileReader fin=new FileReader("D:/Files/base.txt")`. При вызове метода `read()` из объекта потока ввода считывается очередной символ из соответствующего файла и возвращается его числовой код. Код записывается в переменную `a` (команда `a=fin.read()`). В операторе цикла проверяется условие `a!=-1` (при достижении конца файла метод `read()` возвращает значение `-1`). В теле оператора цикла проверяется условие `a==(int)' '`, истинное в случае, если считанный символ является пробелом. Если условие истинно, то выполняется команда `a=(int)'_'`, которой в переменную `a` записывается код, соответствующий символу подчеркивания. Командой `fout.write((char)a)` в файл, связанный с потоком вывода, записывается символ, соответствующий коду из переменной `a`. После этого командой `a=fin.read()` считывается следующий символ из исходного файла.

НА ЗАМЕТКУ

Если из файла, связанного с потоком ввода, прочитан не символ пробела, то он без изменений записывается в файл, связанный с потоком вывода. Если же считывается символ пробела, то он заменяется на символ подчеркивания.

После завершения копирования потоки закрываются (команды `fout.close()` и `fin.close()`).

Еще один пример, который рассматривается далее, связан с использованием *буферизированных потоков*. Эти потоки в процессе работы накапливают данные в специальном буфере, что повышает производительность работы (листинг 13.3).

Листинг 13.3. Буферизированный поток

```
import java.io.*;
import static javax.swing.JOptionPane.*;
class Demo{
   public static void main(String[] args){
      // Имя файла:
      String file="personal.txt";
      // Фамилия сотрудника:
      String name;
      // Переменная для считывания текста:
      String s;
      try{
         // Буферизированный поток:
         BufferedReader br=new BufferedReader(
            new FileReader("D:/Files/"+file)
         );
         // Фамилия сотрудника:
         name=showInputDialog(
            "Укажите фамилию сотрудника:");
```

```
        if(name==null||name.equals("")) System.exit(0);
        while(true){
           // Считывание строки из файла:
           s=br.readLine();
           try{
              if(s.equalsIgnoreCase(name)){
                 System.out.println("Фамилия: "+s);
                 System.out.println(
                    "Имя: "+br.readLine());
                 System.out.println(
                    "Отчество: "+br.readLine());
                 System.out.println(
                    "Возраст: "+br.readLine());
                 System.out.println(
                    "Тел.: "+br.readLine());
                 break;
              }
           }catch(NullPointerException e){
              System.out.println(
                 "Такого сотрудника нет!");
              break;
           }
        }
        // Поток закрывается:
        br.close();
     }catch(IOException e){
        System.out.println("Ошибка ввода/вывода: "+e);
     }
  }
}
```

Предположим, в директории `D:\Files` имеется файл `personal.txt` со следующим содержимым:

```
Петров
Иван
Сергеевич
52
526-44-12

Сидоров
Игорь
Степанович
46
526-00-13

Иванов
Семен
Николаевич
61
522-16-48
```

Это — условная база данных с фамилиями, именами, отчествами, возрастом и телефонами людей (например, сотрудников предприятия). При запуске программы появляется диалоговое окно с полем ввода, в которое предлагается ввести фамилию сотрудника (рис. 13.1).

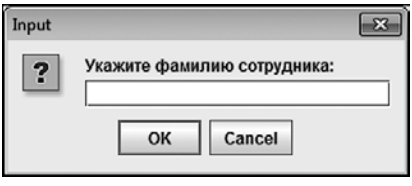


Рис. 13.1. Диалоговое окно для ввода фамилии пользователя

Дальше возможны варианты. Если в поле ввода указана фамилия, представленная в файле (например, `Сидоров`), то в окне вывода отображается информация по данному сотруднику:

Результат выполнения программы (из листинга 13.3)

```
Фамилия: Сидоров
Имя: Игорь
Отчество: Степанович
Возраст: 46
Тел.: 526-00-13
```

Если в поле ввода указать фамилию, которой нет в файле `personal.txt`, то результат выполнения программы будет таким:

Результат выполнения программы (из листинга 13.3)

```
Такого сотрудника нет!
```

Буферизированный поток создается как объект `br` класса `BufferedReader`. Конструктору передается анонимный объект символьного потока, который создается инструкцией `new FileReader("D:/Files/"+file)` (в переменную `file` записывается название файла, в котором хранится информация о сотрудниках). В переменную `name` записывается текст, который пользователь вводит в текстовом поле (фамилия сотрудника). После этого в игру вступает условный оператор, в котором проверяется значение переменной `name` (условие `name==null||name.equals("")`). Если пользователь не ввел фамилию или если это пустая текстовая строка, то командой `System.exit(0)` завершается выполнение программы.

Поиск в файле выполняется с помощью оператора цикла (формально бесконечного, поскольку в качестве условия указано значение `true`). В теле оператора цикла командой `s=br.readLine()` из файла, связанного с потока `br`, считывается строка и записывается в переменную `s`. В условном операторе проверяется условие `s.equalsIgnoreCase(name)`. Оно истинно, если прочитанная строка совпадает

(без учета регистра символов) с текстом, который ввел пользователь. В таком случае отображается несколько следующих строк, каждая из которых считывается командой `br.readLine()`. После этого инструкцией `break` завершается выполнение оператора цикла.

НА ЗАМЕТКУ

Мы исходим из того, что пользователь вводит именно фамилию. Если он введет, например, имя, то результаты могут быть несколько неожиданными (хотя и вполне объяснимыми).

При достижении конца файла попытка прочитать очередную строку методом `readLine()` приведет к тому, что результатом возвращается пустая ссылка `null`. Если так, то выполнение операций со строкой приведет к генерированию исключения класса `NullPointerException`. На этот случай предусмотрена обработка исключения соответствующего класса. Выполняется она следующим образом: отображается сообщение `"Такого сотрудника нет!"`, а затем инструкцией `break` прекращается выполнение оператора цикла.

НА ЗАМЕТКУ

Конец файла будет достигнут, если при просмотре содержимого файла текст, введенный пользователем, не найден.

Закрывается поток командой `br.close()`. На случай проблем с доступом к файлу или иными подобными неприятностями предусмотрена обработка исключения класса `IOException`.

Аргументы командной строки

Дикари, аж плакать хочется.

из к/ф «Кин-дза-дза»

Аргументы командной строки — это параметры, которые передаются программе при ее выполнении. При запуске консольных программ аргументы командной строки указываются через пробел после имени запускаемого файла. В случае использования среды разработки параметры командной строки задаются в настройках проекта.

Аргументы командной строки автоматически преобразуются в текстовый формат и передаются в главный метод программы в виде текстового массива. Элементы такого массива — это текстовые представления для аргументов командной строки. В листинге 13.4 представлена программа, в которой аргументы командной строки построчно отображаются в области вывода.

Листинг 13.4. Аргументы командной строки

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

Текстовый массив `args`, указанный аргументом главного метода `main()`, как раз и является массивом, сформированным на основе аргументов командной строки. В главном методе запускается оператор цикла с индексной переменной `i`, которая пробегает значения от `0` до `args.length-1` включительно. Каждый элемент массива `args` — это текстовое представление для соответствующего аргумента командной строки. Эти значения отображаются в области вывода. Например, если в качестве аргументов командной строки указать значения `Красный`, `Желтый` и `Зеленый`, то результат выполнения программы будет таким:

Результат выполнения программы (из листинга 13.4)

```
[0] Красный
[1] Желтый
[2] Зеленый
```

Если бы нам потребовалось передать в программу нетекстовое значение (например, число), то пришлось бы применить процедуру приведения типов.

ПОДРОБНОСТИ

При работе со средой IntelliJ IDEA задать аргументы командной строки можно следующим образом. В меню Run выбирается команда Edit Configurations. В результате откроется диалоговое окно Run/Debug Configurations, в котором есть поле Program arguments. В этом поле через пробел указываются аргументы, которые передаются в программу при ее запуске.

Методы с произвольным количеством аргументов

Это великая победа дедуктивного метода.

из к/ф «Гостья из будущего»

В Java можно описывать методы с *произвольным количеством аргументов*. Речь идет о том, что на момент описания метода количество аргументов, которые будут переданы методу при вызове, неизвестно. Например, требуется описать метод, который в качестве результата возвращает сумму чисел, переданных аргументами методу,

причем с варьируемым количеством аргументов: один аргумент, два аргумента, три аргумента и так далее. Еще один пример — метод, который возвращает текст, сформированный из символьных аргументов метода, и количество аргументов может быть произвольным. Во всех подобных случаях, когда мы хотим указать, что метод может принимать произвольное количество аргументов определенного типа, указывается идентификатор типа аргументов, затем следует многоточие и формальное название для аргумента. То есть весь набор значений описывается как один аргумент, обрабатываемый как массив.

ПОДРОБНОСТИ

Мы могли бы передавать набор значений в метод в виде массива. Но тогда перед передачей этих значений их сначала пришлось бы оформлять как массив. Это как минимум неудобно. Компромисс состоит в том, что в метод передаются отдельные значения, а обрабатываются они как массив. Массив создается автоматически, и мы в этом участия не принимаем.

Небольшой пример, в котором используются методы с произвольным количеством аргументов, представлен в листинге 13.5.

Листинг 13.5. Произвольное количество аргументов

```
class Demo{
   // Метод для вычисления суммы:
   static int sum(int... a){
      int s=0;
      for(int k=0;k<a.length;k++){
         s+=a[k];
      }
      return s;
   }
   // Метод для формирования текста:
   static String text(char... smb){
      String txt="";
      for(int k=0;k<smb.length;k++){
         txt+=smb[k];
      }
      return txt;
   }
   // Главный метод:
   public static void main(String[] args){
      System.out.println("1+3+5="+sum(1,3,5));
      System.out.println("4+5+6+7+8="+sum(4,5,6,7,8));
      System.out.println(
         text('A','l','p','h','a')
      );
      System.out.println(
         text('C','h','a','r','l','i','e')
      );
   }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 13.5)

```
1+3+5=9
4+5+6+7+8=30
Alpha
Charlie
```

В программе, кроме главного метода, описывается еще два статических метода, у которых произвольное количество аргументов. Метод `sum()` предназначен для вычисления суммы целочисленных значений, переданных в качестве аргументов методу. Формально метод описан с одним аргументом (инструкция `int... a`). Многоточие после идентификатора типа `int` означает, что под `a` скрывается набор целочисленных аргументов. Обрабатывается аргумент `a` как массив. При этом `a[0]` соответствует первому переданному методу аргументу, `a[1]` соответствует второму аргументу и так далее. Выражение `a.length` позволяет узнать количество аргументов, фактически переданных методу при вызове.

ПОДРОБНОСТИ

Алгоритм вычисления результата методом `sum()` простой. Объявляется локальная переменная `s` с начальным нулевым значением, а затем в операторе цикла перебираются все переданные методу аргументы и значения этих аргументов прибавляются к текущему значению переменной `s`. После завершения вычислений значение переменной `s` возвращается в качестве результата метода.

Аргумент метода `text()` описан инструкцией `char... smb`. Она означает, что аргумент `smb` на самом деле является набором символьных аргументов. Обрабатывается аргумент `smb` как массив символьных значений. В теле метода запускается оператор цикла, в котором перебираются аргументы, переданные методу при вызове. Символы дописываются в конец текстовой строки `txt`, начальным значением которой является пустой текстовый литерал. Сформированный текст возвращается результатом метода.

В главном методе программы есть примеры вызова методов `sum()` и `text()` с разным количеством аргументов.

У метода, кроме аргумента, соответствующего набору однотипных значений, могут быть и обычные аргументы. Но есть два важных правила. Во-первых, аргумент, соответствующий набору значений, всегда указывается последним. Во-вторых, такой аргумент может быть только один. Поэтому, например, нельзя описать метод с произвольным количеством целочисленных аргументов и с произвольным количеством символьных аргументов. Зато можно описать метод с фиксированным количеством целочисленных аргументов и произвольным количеством целочисленных аргументов. Также методы с произвольным количеством аргументов могут использоваться при перегрузке методов.

Еще один пример использования методов с произвольным количеством аргументов показан в листинге 13.6.

Листинг 13.6. Перегрузка методов с произвольным количеством аргументов

```
class Demo{
   // Метод с текстовым аргументом и произвольным
   // количеством целочисленных аргументов:
   static void show(String txt,int... nums){
      System.out.print(txt+":");
      for(int k=0;k<nums.length;k++){
         System.out.print(" "+nums[k]);
      }
      System.out.println();
   }
   // Метод с произвольным количеством
   // целочисленных аргументов:
   static void show(int... nums){
      System.out.print("Аргументы {");
      for(int k=0;k<nums.length-1;k++){
         System.out.print(nums[k]+",");
      }
      System.out.println(nums[nums.length-1]+"}");
   }
   // Метод без аргументов:
   static void show(){
      System.out.println("Нет аргументов");
   }
   // Метод с одним целочисленным аргументом:
   static void show(int a){
      System.out.println("Единственный аргумент "+a);
   }
   // Метод с двумя целочисленными аргументами:
   static void show(int a,int b){
      System.out.println("Аргументы "+a+" и "+b);
   }
   // Главный метод программы:
   public static void main(String[] args){
      show();
      show(100);
      show(10,20);
      show(1,2,3);
      show("Числа",1,2,3,4);
   }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 13.6)

```
Нет аргументов
Единственный аргумент 100
```

```
Аргументы 10 и 20
Аргументы {1,2,3}
Числа: 1 2 3 4
```

В этой программе создается несколько версий метода `show()` (то есть имеет место перегрузка метода). Причем некоторые из версий метода имеют произвольное количество аргументов. В частности, мы описываем версию метода с текстовым аргументом и произвольным количеством целочисленных аргументов, версию метода с произвольным количеством целочисленных аргументов, версию без аргументов, с одним целочисленным аргументом и двумя целочисленными аргументами. Интересно здесь то, что, например, версия с двумя целочисленными аргументами — это частный случай версии метода с произвольным количеством целочисленных аргументов. Но поскольку такая версия описана, то при вызове метода с двумя аргументами используется именно она. Это же замечание относится к версии с одним целочисленным аргументом и к версии без аргументов.

Цикл по коллекции

> История, леденящая кровь. Под маской овцы скрывался лев!
>
> *из к/ф «Покровские ворота»*

Если мы перебираем элементы массива, то обычно это делаем с помощью индекса — скажем, в операторе цикла используется индексная переменная, с помощью которой мы получаем доступ к элементам. В этом случае мы перебираем значения для индекса элементов массива и уже с помощью индекса получаем доступ к элементам. Но есть и другой подход, в рамках которого перебираются непосредственно элементы массива (или другого набора элементов). В таком случае можно использовать специальную форму для оператора цикла `for`, который обычно называют *циклом по коллекции*. Синтаксис использования такого оператора следующий:

```
for(тип переменная: массив){
   // Команды
}
```

После ключевого слова `for` в круглых скобках указывается выражение, которое состоит из объявления локальной переменной (указывается тип и название переменной), двоеточия и названия массива, в котором будут перебираться элементы (тип элементов массива должен совпадать с типом переменной). После всей `for`-инструкции в фигурных скобках размещается блок команд основного тела оператора цикла (если команда одна, то фигурные скобки можно не использовать).

В процессе работы оператора цикла переменная, объявленная в `for`-инструкции, будет последовательно принимать значения из массива (указанного после двоеточия

в той же for-инструкции). При каждом таком значении переменной выполняются команды в теле оператора цикла. Небольшой пример, в котором иллюстрируется работа цикла по коллекции, представлен в листинге 13.7.

Листинг 13.7. Цикл по коллекции

```
class Demo{
   public static void main(String[] args){
      // Одномерный массив:
      char[] symbs={'A','B','C','D'};
      // Двумерный массив:
      int[][] nums={{1,2},{3,4,5},{6,7,8,9}};
      // Цикл по коллекции:
      for(char s: symbs){
         System.out.print(s+" ");
      }
      System.out.println();
      // Цикл по коллекции:
      for(int[] p: nums){
         for(int n: p){
            System.out.print(n+" ");
         }
         System.out.println();
      }
   }
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 13.7)

```
A B C D
1 2
3 4 5
6 7 8 9
```

В программе создается два массива: одномерный символьный массив symbs и двумерный целочисленный массив nums. Причем в двумерном массиве имеется три строки с разным количеством элементов.

Операторы цикла по коллекции используются для отображения содержимого массивов. Для отображения одномерного символьного массива мы в операторе цикла объявляем символьную переменную s, а после двоеточия указываем имя массива symbs. За каждый цикл переменная s принимает значение очередного элемента из массива symbs. Поэтому при выполнении команды System.out.print(s+" ") в теле оператора цикла отображается значение соответствующего элемента.

Для отображения содержимого двумерного массива мы используем вложенные операторы цикла по коллекции. Во внешнем операторе цикла объявляется переменная p, тип которой определяется инструкцией int[], а перебирается содержимое массива nums. Здесь уместно вспомнить, что двумерный массив — это массив,

элементами которого являются переменные массива. Каждая такая переменная ссылается на одномерный массив, который, по сути, является строкой двумерного массива. Переменная p пробегает последовательность значений элементов из массива nums. Элементами этого массива являются ссылки на одномерные целочисленные массивы. Такие ссылки относятся к типу int[]. Получается, что переменная p за каждый цикл ссылается на одномерный целочисленный массив. Другими словами, переменную p можно отождествлять с одномерным целочисленным массивом. Поэтому во внутреннем операторе цикла по коллекции объявляется целочисленная переменная n, а перебираются с помощью этой переменной элементы из массива, на который в данный момент ссылается переменная p (то есть элементы в строке двумерного массива).

НА ЗАМЕТКУ

Оператор цикла по коллекции во многих случаях упрощает код, но имеет и недостатки. Например, иногда нужно перебирать не весь массив, а только его часть. Но еще важнее то, что с помощью оператора цикла по коллекции можно только прочитать значения элементов массива, но нельзя их изменить. Причина проста: переменная, которая объявляется в for-инструкции, является копией элемента. Непосредственного доступа к элементу через эту переменную нет.

Рекурсия

Именно так выражается ее потребность в мировой гармонии.

из к/ф «Покровские ворота»

С *рекурсией* мы уже сталкивались ранее. Под рекурсией подразумевают ситуацию, когда метод в процессе выполнения вызывает сам себя. Причем это может быть как прямой, непосредственный вызов метода, так и косвенный: один метод вызывает другой метод, а он, в свою очередь, вызывает метод исходный. Схемы могут быть и более замысловатыми. Но в любом случае в цепочке вызовов должна быть предусмотрена ситуация, когда такие вызовы прекращаются (иначе получится бесконечная рекурсия). Обычно соответствующий механизм базируется на том, что метод вызывает сам себя только при определенных обстоятельствах (при определенном значении аргумента или аргументов). Рассмотрим некоторые примеры. В листинге 13.8 представлена программа, в которой рекурсия использована в статическом методе.

Листинг 13.8. Рекурсия в статическом методе

```
class Demo{
   // Инверсия текстовой строки:
   static String inverse(String txt){
```

```
      if(txt.length()>0){
         return inverse(txt.substring(1))+txt.charAt(0);
      }else{
         return "";
      }
   }
   // Главный метод:
   public static void main(String[] args){
      String txt="Изучаем Java";
      // Вызов метода с рекурсией:
      String str=inverse(txt);
      System.out.println(txt);
      System.out.println(str);
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 13.8)

```
Изучаем Java
avaJ меачузИ
```

Мы описываем статический метод `inverse()`, у которого имеется текстовый аргумент и который в качестве результата возвращает текст, переданный аргументом, но буквы следуют в обратном порядке. Такая задача достаточно просто решается с применением оператора цикла. Но мы используем рекурсию. В теле метода в условном операторе проверяется условие `txt.length()>0`. Если условие истинно, то возвращается значение выражения `inverse(txt.substring(1))+txt.charAt(0)`, в котором метод `inverse()` вызывает сам себя.

ПОДРОБНОСТИ

Значением выражения `txt.charAt(0)` является первый (начальный) символ в строке `txt`. Значением выражения `txt.substring(1)` является подстрока, которая получается извлечением символов из текста `txt`, начиная с символа с индексом `1` и до конца текста. Фактически речь идет о строке, которая получается из текста `txt` отбрасыванием первого символа.

Если условие ложно, то результатом метода возвращается пустой текстовый литерал `""`. Таким образом, получается, что при вызове метода с некоторым текстовым аргументом метод формирует строку, в конце которой размещается первый символ из переданного аргументом текста, а начальная часть строки формируется вызовом этого же метода, но в аргументе отбрасывается первый символ. Метод вызывает по цепочке сам себя, пока аргументом не окажется пустая текстовая строка.

В главном методе программы есть пример вызова метода `inverse()`.

Рекурсия может быть и непрямой, когда метод вызывает не сам себя, а другой метод, который, в свою очередь, вызывает исходный метод. Вот соответствующий пример.

Попытаемся решить уравнение вида $x = f(x)$ методом последовательных итераций, в соответствии с которым для корня уравнения берется некоторое начальное приближение x_0, и на его основе вычисляется первое приближение $x_1 = f(x_0)$. Второе приближение вычисляется как $x_2 = f(x_1)$, и так далее. С помощью данного алгоритма решим уравнение $x = (x^2 + 12)/8$. Будем искать корень $x = 2$, а в качестве начального приближения возьмем значение $x_0 = 0$. Код примера представлен в листинге 13.9.

Листинг 13.9. Непрямая рекурсия

```
class Demo{
   // Метод для вычисления нового приближения:
   static void next(double x,int n){
      System.out.println("Приближение: "+x);
      solve((x*x+12)/8,n-1);
   }
   // Метод для вычисления корня уравнения:
   static void solve(double x,int n){
      if(n==0) System.out.println("Корень: "+x);
      else next(x,n);
   }
   // Главный метод:
   public static void main(String[] args){
      solve(0,7);
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 13.9)

```
Приближение: 0.0
Приближение: 1.5
Приближение: 1.78125
Приближение: 1.8966064453125
Приближение: 1.9496395010501146
Приближение: 1.9751367730068674
Приближение: 1.9876456590104978
Корень: 1.9938419082229095
```

Мы описали статический метод `next()` с аргументами `x` и `n`. В теле метода командой `System.out.println("Приближение: "+x)` отображается сообщение, а затем командой `solve((x*x+12)/8,n-1)` вызывается метод `solve()`. В методе `solve()`, в свою очередь, в условном операторе проверяется условие `n==0`, и если оно истинно, то командой `System.out.println("Корень: "+x)` отображается сообщение со значением корня уравнения. Если условие ложно, то командой `next(x,n)` вызывается метод `next()`. Получается, что метод `next()` вызывает метод `solve()`, а метод `solve()` вызывает метод `next()`.

Для вычисления корня необходимо вызвать метод `solve()`, передав ему аргументами начальное приближение для корня и количество выполняемых итераций. Например,

в главном методе мы используем команду `solve(0,7)`, что соответствует нулевому начальному значению для корня и семи итерациям для поиска корня. Работает вся схема так. При вызове метода `solve()` проверяется значение второго аргумента (равно `7`). Оно отлично от нуля, поэтому вызывается метод `next()`, но вторым аргументом ему передается значение `6`. В методе `next()` отображается текущее приближение для корня и вычисляется новое значение, которое передается в метод `solve()`. В методе `solve()` проверяется значение второго аргумента (которое равно `6`), и вызывается метод `next()`, но второй аргумент уже равен `5`. И так далее. Процесс завершится, когда метод `solve()` будет вызван со вторым нулевым аргументом.

Для использования рекурсии метод не обязательно должен быть статическим. В листинге 13.10 представлена программа, в которой описывается класс `MyClass` с целочисленным полем `num` и методом `factorial()`, возвращающим факториал числа, записанного в числовое поле.

Листинг 13.10. Рекурсия в нестатическом методе

```
// Класс с методом для вычисления факториала:
class MyClass{
   // Целочисленное поле:
   int num;
   // Конструктор:
   MyClass(int n){
      num=n;
   }
   // Метод для вычисления факториала:
   int factorial(){
      if(num<1) return 1;
      else return new MyClass(num-1).factorial()*num;
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Создание объекта:
      MyClass obj=new MyClass(5);
      // Вычисление факториала:
      System.out.println(obj.num+"! = "+obj.factorial());
      obj.num=10;
      System.out.println(obj.num+"! = "+obj.factorial());
   }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 13.10)

```
5! = 120
10! = 3628800
```

В данном случае интерес представляет метод `factorial()`. Его основу составляет условный оператор, в котором проверяется условие `num<1`. Если условие истинно,

то метод возвращает значение 1. В противном случае в качестве результата возвращается значение выражения new MyClass(num-1).factorial()*num. Означает оно следующее. Инструкцией new MyClass(num-1) создается анонимный объект класса MyClass, причем поле num этого объекта на единицу меньше поля num объекта, из которого вызывается метод. Из этого анонимного объекта вызывается метод factorial(), а результат вызова умножается на значение поля num. Поэтому если мы, например, вызываем метод factorial() из объекта obj, поле num которого имеет значение 5, то для вычисления результата создается анонимный объект со значением поля, равным 4, из этого объекта вызывается метод factorial(), и то, что получится, будет умножено на 5. При вызове метода factorial() из анонимного объекта создается еще один анонимный объект со значением поля, равным 3. Из этого объекта вызывается метод factorial() (и вычисленное значение будет умножено на 4). Вызов метода factorial() приводит к созданию очередного анонимного объекта со значением поля равным 2, и так далее. Процесс продолжается, пока не будет создан анонимный объект с полем, значение которого равно 0 (в этом случае вызов метода factorial() даст значение 1).

Перечисления

> Многие меня уважают. Некоторые даже боятся.
>
> *из к/ф «Служебный роман»*

Есть такой тип данных, как *перечисление*. Переменная, которая относится к перечислению, может принимать только ограниченный набор значений. Пример такой ситуации — переменная, определяющая день недели: возможны всего семь значений для такой переменной (дни с *понедельника* по *воскресенье*). Еще один пример — переменная, определяющая стороны света. Для такой переменной есть всего четыре значения (*север*, *юг*, *восток*, *запад*). То есть это вполне жизненная ситуация.

Перечисление объявляется так: после ключевого слова enum указывается название перечисления и в фигурных скобках — набор идентификаторов, которые определяют константы, являющиеся возможными значениями переменной, относящейся к типу перечисления. Вот пример объявления перечисления:

```
enum MyColor {RED, GREEN, BLUE}
```

В данном случае объявляется перечисление MyColor. Это — тип данных. Поэтому мы можем объявить переменную данного типа. Например, такую:

```
MyColor clr;
```

В качестве значения переменной clr можно присвоить одну из констант — RED, GREEN или BLUE, указанных в объявлении перечисления MyColor. Причем указывается не

просто название константы, а еще и название перечисления: `MyColor.RED`, `MyColor.GREEN` или `MyColor.BLUE`. В этом смысле законной будет команда `clr=MyColor.RED` или `clr=MyColor.BLUE`. Небольшой пример с использованием перечисления приведен в листинге 13.11.

Листинг 13.11. Знакомство с перечислениями

```
import java.util.Scanner;
// Перечисление:
enum Coin {ONE,THREE,FIVE,TEN,TWENTY,FIFTY}
// Главный класс:
class Demo{
   // Статический метод возвращает
   // значение типа перечисления:
   static Coin getCoin(){
      Scanner input=new Scanner(System.in);
      System.out.print("Какая монета? ");
      int n;
      n=input.nextInt();
      switch(n){
         case 1:
            return Coin.ONE;
         case 3:
            return Coin.THREE;
         case 5:
            return Coin.FIVE;
         case 10:
            return Coin.TEN;
         case 20:
            return Coin.TWENTY;
         default:
            return Coin.FIFTY;
      }
   }
   // Главный метод:
   public static void main(String[] args){
      // Переменные типа перечисления:
      Coin A=Coin.TEN;
      Coin B=getCoin();
      System.out.println("Первая монета: "+A);
      System.out.println("Вторая монета: "+B);
      // Сравнение значений:
      if(A==B){
         System.out.println("Это одно и то же");
      }else{
         if(A.ordinal()>B.ordinal()){
            System.out.println(A+" — больше, чем "+B);
         }else{
            System.out.println(B+" — больше, чем "+A);
         }
      }
   }
}
```

Результат выполнения программы может быть, например, таким (здесь и далее жирным шрифтом выделены значения, введенные пользователем):

Результат выполнения программы (из листинга 13.11)

```
Какая монета? 20
Первая монета: TEN
Вторая монета: TWENTY
TWENTY — больше, чем TEN
```

Таким:

Результат выполнения программы (из листинга 13.11)

```
Какая монета? 5
Первая монета: TEN
Вторая монета: FIVE
TEN — больше, чем FIVE
```

Или, например, таким:

Результат выполнения программы (из листинга 13.11)

```
Какая монета? 10
Первая монета: TEN
Вторая монета: TEN
Это одно и то же
```

Мы создаем перечисление `Coin` для реализации переменных, которые должны определять достоинство монеты.

НА ЗАМЕТКУ

Мы исходим из того, что монеты могут иметь достоинство 1, 3, 5, 10, 20 или 50 денежных единиц (копеек или центов, например).

Перечисление объявляется следующей инструкцией:

```
enum Coin {ONE,THREE,FIVE,TEN,TWENTY,FIFTY}
```

В главном методе объявляются две переменные, `A` и `B`, типа `Coin`. Переменной `A` значением присваивается константа `Coin.TEN`, а переменной `B` значением присваивается результат вызова метода `getCoin()`. При вызове метода пользователя просят ввести достоинство монеты и, в зависимости от введенного значения, метод возвращает результатом одну из констант перечисления `Coin`. Далее отображаются значения этих переменных, выполняется их сравнение. Здесь нужно учесть несколько обстоятельств.

Во-первых, перечисление (тип, объявленный с использованием идентификатора `enum`) на самом деле является классом. Причем этот класс автоматически

является подклассом библиотечного класса Enum. То есть фактически, когда мы объявляем перечисление, мы на самом деле создаем класс. Во-вторых, константы, указанные в объявлении перечисления, являются статическими константными объектами типа перечисления (то есть это константные ссылки на объекты типа перечисления). Переменная типа перечисления — это объектная переменная, которая ссылается на один из статических объектов. Следовательно, у этой переменной есть методы. И у констант, указанных в объявлении перечисления, тоже есть методы (поскольку константы являются ссылками на объекты). Среди этих методов представлен и метод toString(), который вызывается автоматически при выполнении команд System.out.println("Первая монета: "+A) и System.out.println("Вторая монета: "+B). В результате вместо переменных типа перечисления используется название константы, соответствующей значению переменной. Переменные типа перечисления можно сравнивать с помощью операторов == (равно) и != (не равно). В этом случае сравнение выполняется так, как сравниваются две объектные переменные: они равны, если ссылаются на один и тот же объект. Кроме этого, все константы, указанные в объявлении перечисления, имеют порядковый номер. Нумерация констант начинается с нуля и соответствует порядку, в котором константы объявлены в перечислении. То есть у константы ONE порядковый номер 0, у константы THREE порядковый номер 1, у константы FIVE порядковый номер 2, у константы TEN порядковый номер 3, у константы TWENTY порядковый номер 4, а у константы FIFTY порядковый номер 5. Узнать порядковый номер константы, которая является значением переменной, можно с помощью метода ordinal(). Именно этот метод используется для сравнения переменных A и B на предмет больше/меньше.

Есть еще несколько полезных методов. Метод name() позволяет получить название константы, которая является значением переменной типа перечисления. Если статическому методу valueOf() (вызывается из перечисления) передать текст с названием константы, то соответствующая константа будет возвращена результатом метода. Статический метод values(), который также вызывается непосредственно из перечисления, возвращает результатом массив с константами из этого перечисления. Небольшая вариация на тему предыдущего примера представлена в листинге 3.12.

Листинг 13.12. Операции с перечислениями

```
import java.util.*;
// Перечисление:
enum Coin {ONE,THREE,FIVE,TEN,TWENTY,FIFTY}
// Главный класс:
class Demo{
   public static void main(String[] args){
      Scanner input=new Scanner(System.in);
      // Переменные типа перечисления:
      Coin A,B;
      try{
         System.out.print("Первая монета: ");
```

```
            A=Coin.valueOf(input.nextLine());
            System.out.print("Вторая монета: ");
            B=Coin.valueOf(input.nextLine());
            // Отображение значений:
            System.out.println(A.name()+" и "+B.name());
         }catch(Exception e){
            System.out.println("Некорректное значение");
            System.exit(0);
         }
         // Константы перечисления:
         System.out.println(
            "Константы "+Arrays.toString(Coin.values())
         );
         for(Coin s: Coin.values()){
            System.out.println(
               "["+s.ordinal()+"] "+s.name()
            );
         }
      }
}
```

Результат выполнения программы может быть таким:

Результат выполнения программы (из листинга 13.12)

```
Первая монета: FIFTY
Вторая монета: THREE
FIFTY и THREE
Константы [ONE, THREE, FIVE, TEN, TWENTY, FIFTY]
[0] ONE
[1] THREE
[2] FIVE
[3] TEN
[4] TWENTY
[5] FIFTY
```

Мы объявляем две переменные, `A` и `B`, типа `Coin`. Значения для переменных вводит пользователь: текст, который вводится, передается аргументом методу `valueOf()`, который вызывается из перечисления `Coin`.

ПОДРОБНОСТИ

Если методу `valueOf()` передан аргумент, не соответствующий названию константы, метод генерирует исключение класса `IllegalArgumentException`. Если аргументом метода является пустая ссылка, то генерируется исключение класса `NullPointerException`. В программе используется обработка исключений на случай, если пользователь вводит некорректное значение.

Кроме этого, мы отображаем список с константами из перечисления `Coins` (ссылку на массив с константами получаем с помощью инструкции `Coin.values()`), а также

с помощью оператора цикла по коллекции отображаем названия констант и их порядковый номер.

В перечислениях можно описывать методы (листинг 13.13).

Листинг 13.13. Методы в перечислении

```
// Перечисление:
enum MyColor {
   // Константы:
   RED,GREEN,BLUE;
   // Метод:
   MyColor next(){
      if(this==RED) return GREEN;
      else if(this==GREEN) return BLUE;
      else return RED;
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Переменная типа перечисления:
      MyColor clr=MyColor.RED;
      for(int k=0;k<=MyColor.values().length;k++){
         System.out.println((k+1)+" – "+clr);
         // Вызов метода из переменной:
         clr=clr.next();
      }
   }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 13.13)

```
1 – RED
2 – GREEN
3 – BLUE
4 – RED
```

В этой программе мы описываем перечисление `MyColor`, в котором имеется три константы — `RED`, `GREEN` и `BLUE`. После указания констант (в конце ставится точка с запятой) описывается метод, который называется `next()`. Метод возвращает значение типа `MyColor`. В теле метода с помощью вложенных условных операторов определяется текущее значение переменной, из которой вызывается метод (ссылку на эту переменную получаем с помощью ключевого слова `this`). Метод `next()` возвращает константу, следующую (с учетом циклической перестановки) в списке констант перечисления за значением переменной, из которой вызван метод. Работа этого метода проверяется в главном методе программы.

Схема с описанием методов в перечислении может быть и более замысловатой. Так, метод может быть описан в перечислении, но, кроме этого, метод может переопределяться для константных объектов (всех или только некоторых). В листинге 13.14 представлена версия предыдущей программы, в которой реализован указанный подход.

Листинг 13.14. Переопределение метода в перечислении

```
// Перечисление:
enum MyColor {
   RED{
      MyColor next(){
         return GREEN;
      }
   },
   GREEN{
      MyColor next(){
         return BLUE;
      }
   },
   BLUE{
      MyColor next(){
         return RED;
      }
   };
   // Абстрактный метод:
   abstract MyColor next();
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      MyColor clr=MyColor.RED;
      for(int k=0;k<=MyColor.values().length;k++){
         System.out.println((k+1)+" – "+clr);
         clr=clr.next();
      }
   }
}
```

Результат выполнения программы — точно такой же, как и в предыдущем случае. Что касается самой программы, то перечисление `MyColor` мы описываем так, что метод `next()` описывается для каждого константного объекта. Когда метод вызывается из переменной типа перечисления, то используется версия метода, которая соответствует значению переменной.

ПОДРОБНОСТИ

Поскольку мы описали метод `next()` для каждой из констант, то в самом перечислении метод объявлен как абстрактный. Это не обязательно, но даже если бы мы описали метод, он бы фактически все равно не использовался. Что касается

> константа, то те из них, для которых описана персональная версия метода `next()`, будут ссылаться на объекты классов, которые являются подклассами данного перечисления. Если для константы персональную версию метода не описать, то такая константа будет ссылаться на объект класса перечисления. Но на практике данное обстоятельство особо большого значения не имеет.

Кроме методов, в перечислениях могут описываться поля и конструкторы. Несложный пример с иллюстрацией того, как это может выглядеть, представлен в листинге 13.15.

Листинг 13.15. Конструктор и поле в перечислении

```
// Перечисление:
enum MyColor {
   // Константы:
   RED(100),GREEN(200),BLUE(300);
   // Закрытое поле:
   private int code;
   // Конструктор:
   MyColor(int n){
      code=n;
   }
   // Метод:
   int getCode(){
      return code;
   }
}
// Главный класс:
class Demo{
   public static void main(String[] args){
      // Переменная типа перечисления:
      MyColor clr;
      // Перебор значений:
      for(int k=0;k<MyColor.values().length;k++){
         clr=MyColor.values()[k];
         System.out.println(
            "["+clr.ordinal()+"] "+clr.name()+": "+
            clr.getCode()
         );
      }
   }
}
```

Результат выполнения программы таков:

Результат выполнения программы (из листинга 13.15)

```
[0] RED: 100
[1] GREEN: 200
[2] BLUE: 300
```

Теперь в описании перечисления мы объявляем закрытое целочисленное поле `code` и метод `getCode()`, который в качестве результата возвращает значение поля `code`. Еще в перечислении описан конструктор с целочисленным аргументом. Аргумент конструктора определяет значение поля `code`. Все константы указаны с целочисленным значением в круглых скобках. Это значение, которое передается конструктору при создании объекта, на который ссылается соответствующая константа. Поэтому, например, у объекта, на который ссылается константа `RED`, значение поля `code` равно `100`. Для объекта, на который ссылается константа `GREEN`, значение поля `code` равно `200`. А для объекта, на который ссылается константа `BLUE`, значение поля `code` равно `300`. Узнать значение поля можно с помощью метода `getCode()`, что, собственно, и делается в главном методе программы.

Резюме

> Ну понимаете, я за кефиром пошел,
> а тут такие приключения.
>
> *из к/ф «Гостья из будущего»*

- Для записи данных в файл и чтения данных из файла используют байтовые и символьные потоки ввода/вывода.
- Байтовые потоки ввода/вывода можно создавать на основе классов `FileInputStream` (файловый поток ввода) и `FileOutputStream` (файловый поток вывода). Символьные потоки создаются на основе классов `FileReader` (файловый поток ввода) и `FileWriter` (файловый поток вывода). На основе соответствующего класса создается объект, который отождествляется с потоком ввода или вывода. При создании объекта потока аргументом конструктора класса передается полное имя файла, с которым связан поток.
- У объектов потоков есть методы `read()` и `write()`, с помощью которых данные читаются из файла и записываются в файл. Для закрытия потоков используют метод `close()`.
- Буферизированные потоки позволяют выполнять операции с использованием специального буфера обмена. Буферизированный поток создается на основе символьного.
- При запуске программы ей можно передавать параметры (аргументы командной строки). В программу такие параметры передаются в текстовом виде как элементы массива, указанного аргументом главного метода программы.
- В Java существует возможность создавать методы с произвольным количеством однотипных аргументов. Набор таких аргументов описывается как один

аргумент, а после идентификатора типа указывается многоточие. Обрабатывается данный аргумент как массив. Если у метода есть и обычные аргументы, то они следуют до аргумента, обозначающего набор значений. Методы с произвольным количеством аргументов можно перегружать.

- Оператор цикла по коллекции позволяет перебирать непосредственно элементы массива, не обращаясь к индексам элементов. В описании метода по коллекции в круглых скобках после ключевого слова `for` объявляется переменная и через двоеточие указывается массив (далее в фигурных скобках указываются команды, выполняемые оператором цикла). В процессе работы оператора цикла переменная поочередно принимает значение элементов указанного массива.
- Рекурсия подразумевает ситуацию, когда метод прямо или косвенно вызывает сам себя. Рекурсия может использоваться как со статическими, так и с нестатическими методами.
- Существует специальный тип данных, который называется перечислением. Переменная, относящаяся к перечислению, может принимать в качестве значения только одну из констант, указанных в описании перечисления.
- В наиболее простом случае перечисление описывается так: указывается ключевое слово `enum`, название перечисления и в фигурных скобках список констант, которые входят в перечисление. Кроме этого, в перечислении могут описываться методы, поля и конструкторы.

Заключение
Итоги и перспективы

— Интурист хорошо говорит.
— А что он говорит, конкретно что?

из к/ф «Иван Васильевич меняет профессию»

Итак, мы рассмотрели все основные механизмы и приемы, используемые при программировании на Java. Этого должно быть достаточно для написания собственных программ. Вместе с тем каждая рассмотренная нами тема могла бы стать предметом обсуждения для отдельной книги. При подборе материала всегда находишься между Сциллой и Харибдой: с одной стороны, необходимо ограничиваться разумными объемами текста, с другой — этот текст должен быть информативным. Более того, жизнь вокруг нас меняется, а вместе с ней меняются и технологии. Последнее относится и к Java. Процесс модернизации языка программирования (и Java-технологии в целом) непрерывен, и с ним следует идти в ногу. Отсюда — необходимость в универсализации подходов, используемых при изучении Java. Согласитесь, эти задачи сложно решать одновременно. Концепция, использованная в книге, базируется на том, что лучший учитель — это практика. Представленные примеры должны были проиллюстрировать концепцию и принципы применения Java-технологий. Главная цель, которую мы пытались достигнуть, — заложить основы в понимании принципов функционирования языка. Создать задел, который позволил бы читателю самостоятельно осваивать и разбирать подходы к созданию сложных программных продуктов. Другими словами, приоритетом является выработка определенных алгоритмов, которые могли бы эффективно применяться на практике. Благо наполнить эти алгоритмы фактической информативной базой особого труда не составляет. Насколько это удалось — судить, конечно, читателю. В любом случае хочется пожелать ему успехов и выразить благодарность за интерес к книге.

Приложение
Программное обеспечение

Нет денег. И деньги, и документы, и валюта — все осталось у экскурсовода.

из к/ф «Кин-дза-дза»

Здесь приводится краткая справка по необходимому для программирования в Java ПО. Мы остановимся на двух вопросах:

- Какое ПО необходимо установить на компьютер?
- Как его использовать?

Необходимо установить систему JDK (сокращение от *Java Development Kit*), а также интегрированную среду разработки. В качестве последней предлагается использовать среду разработки IntelliJ IDEA.

НА ЗАМЕТКУ

Хотя далее описываются (очень кратко) методы работы со средой IntelliJ IDEA, в случае использования другой среды разработки все операции обычно выполняются схожим образом. Различия сводятся к внешнему виду окна среды разработки и к иному способу размещения команд в меню.

Загрузка программного обеспечения

— Астронавты! Которая тут цапа?
— Там ржавая гайка, родной.
— У вас все тут ржавое.
— А эта самая ржавая.

из к/ф «Кин-дза-дза»

Для установки системы JDK перейдите на сайт www.oracle.com и найдите там страницу загрузки ПО. Есть несколько платформ Java. Нас интересует стандартная

платформа Java SE (сокращение от *Java Standard Edition*). На рис. П.1 окно браузера открыто на странице www.oracle.com.

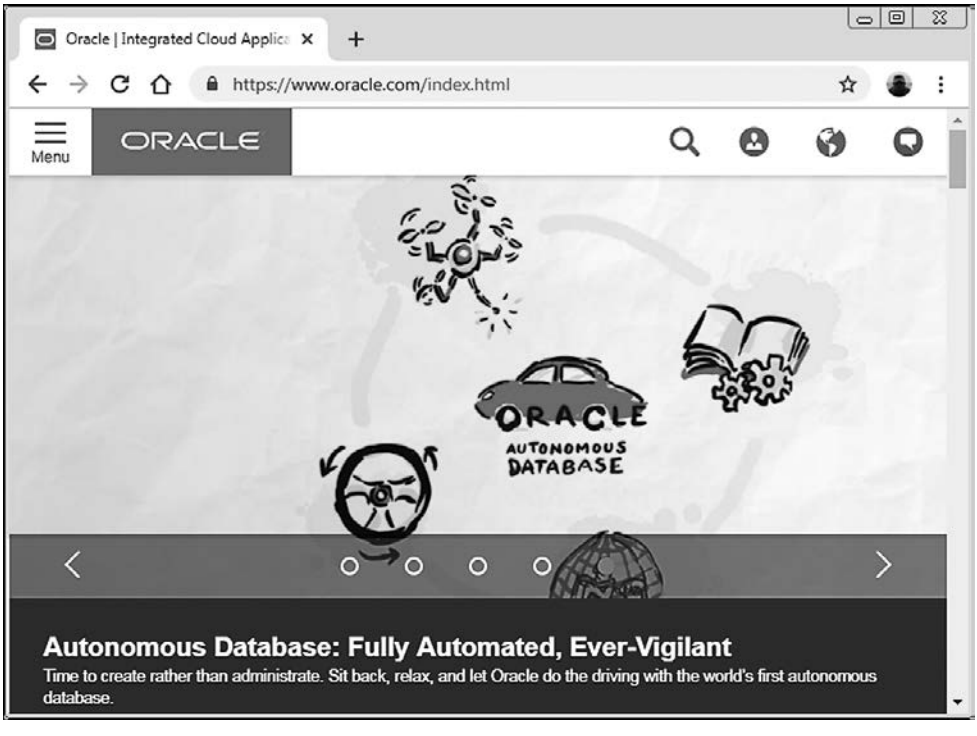


Рис. П.1. Окно браузера открыто на странице www.oracle.com

Как может выглядеть станица для загрузки системы JDK, показано на рис. П.2.

Выберите установочные файлы в соответствии с типом и разрядностью операционной системы.

НА ЗАМЕТКУ

Начиная с версии Java 9, выполняется поддержка только 64-разрядных систем. Примеры, представленные в книге, рассчитаны на использование версии не ниже Java 8.

Имейте в виду, что время от времени путь, с которого выполняется загрузка установочных файлов, меняется.

Обновления для исполнительной среды можно загружать на сайте www.java.com. Окно браузера, открытое на указанной странице, показано на рис. П.3.

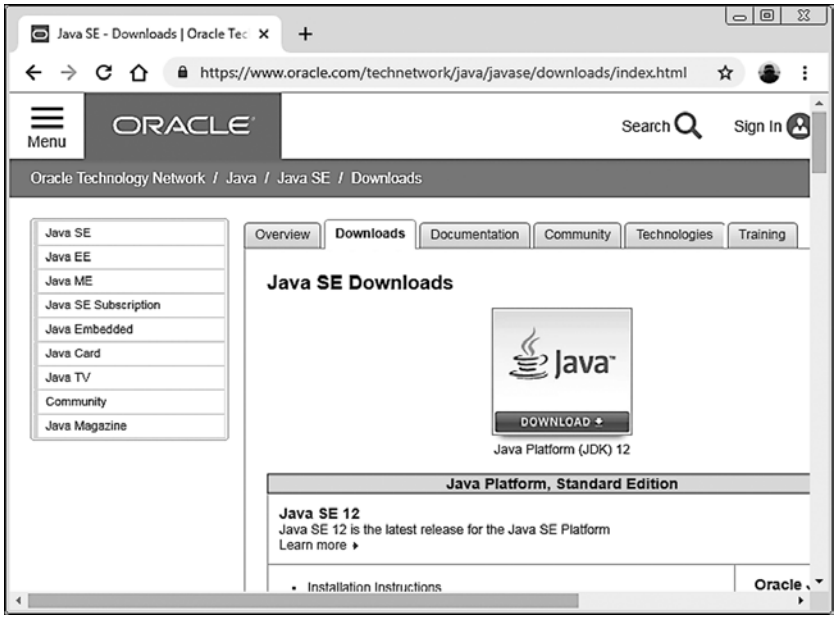


Рис. П.2. Окно для загрузки системы JDK

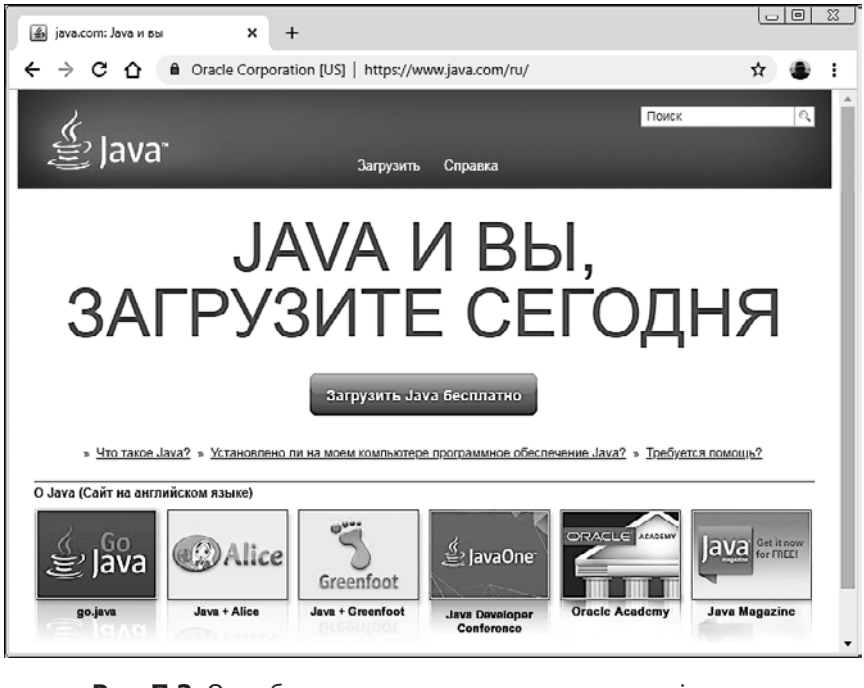


Рис. П.3. Окно браузера открыто на странице www.java.com

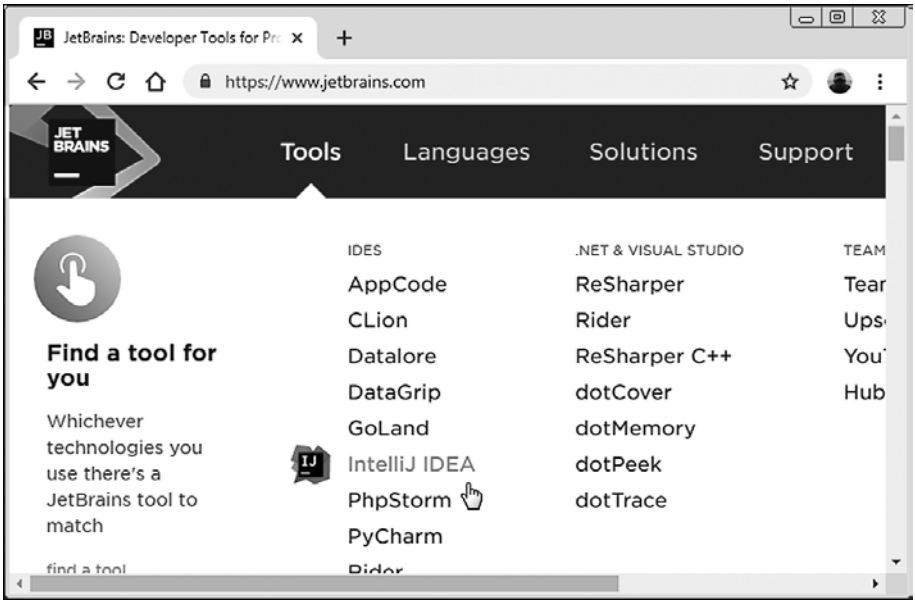


Рис. П.4. Окно браузера открыто на странице www.jetbrains.com

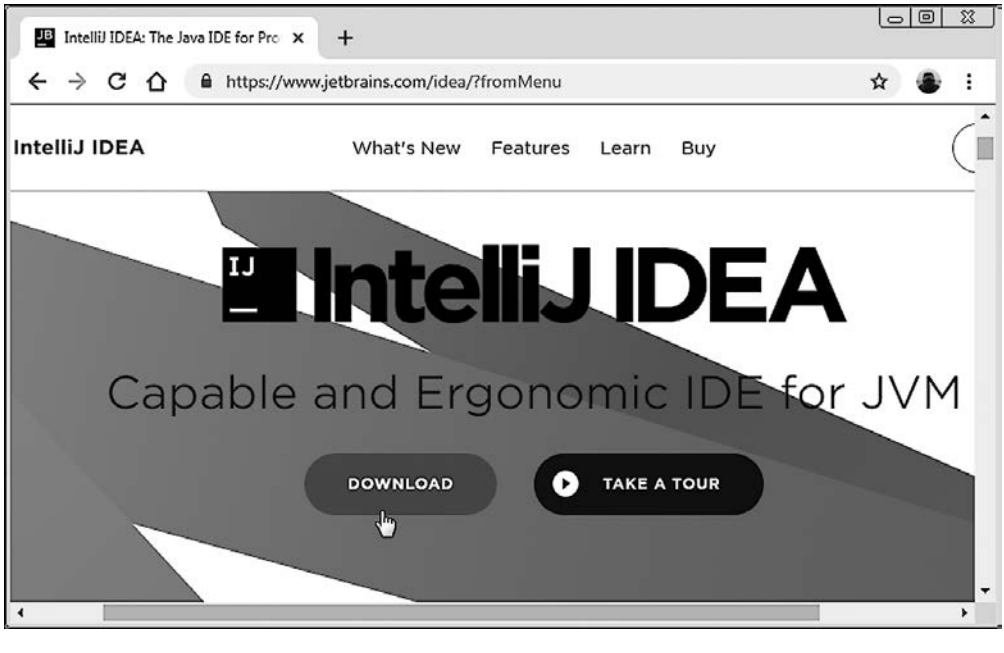


Рис. П.5. Процесс выбора загрузочных файлов среды IntelliJ IDEA

После загрузки установочных файлов выполните установку системы JDK.

На следующем этапе загрузите установочные файлы и установите интегрированную среду разработки. На момент написания книги самой популярной является среда IntelliJ IDEA. Именно в этой среде компилировались коды из книги. Загрузить файлы для установки среды IntelliJ IDEA можно на сайте www.jetbrains.com. Окно браузера, открытое на этой странице, показано на рис. П.4.

Здесь содержатся ссылки на установочные файлы разных сред разработки. Процесс выбора загрузочных файлов для среды IntelliJ IDEA показан на рис. П.5.

Установку среды разработки разумно выполнять после установки системы JDK.

Использование среды IntelliJ IDEA

Я не бездействовал. Я сразу на каппу нажал.
Скрипач свидетель.

из к/ф «Кин-дза-дза»

Окно среды разработки IntelliJ IDEA (с открытым в нем проектом) показано на рис. П.6.

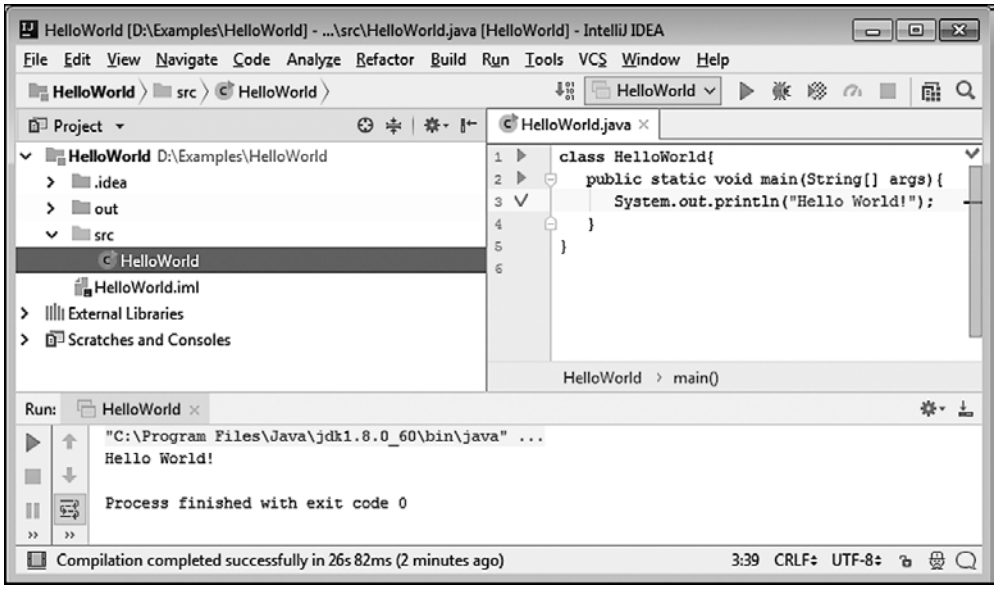


Рис. П.6. Окно среды разработки IntelliJ IDEA

Для эффективной работы со средой разработки необходимо как минимум уметь выполнять несколько базовых действий, среди которых:

- создание нового проекта;
- компилирование и запуск проекта;
- закрытие проекта;
- открытие уже существующего проекта;
- выполнение минимальных настроек и операций, связанных с проектом.

Начнем с создания нового проекта. Здесь возможны два варианта. Во-первых, если в окне среды разработки уже открыт какой-то проект, то можно воспользоваться командой Project из подменю New, размещенного в меню File (рис. П.7).

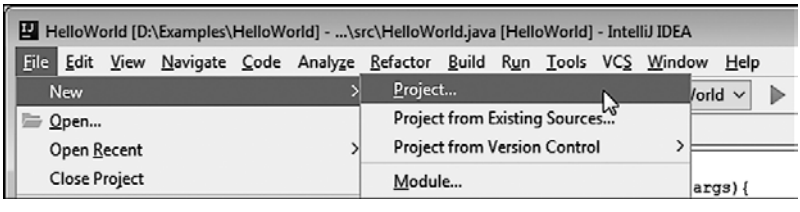


Рис. П.7. Создание нового проекта с помощью команды Project из подменю New меню File

В результате открывается окно создания нового проекта New Project (рис. П.8).

В этом окне выбирается тип проекта и при необходимости тип используемой системы JDK. Ее можно выбрать с помощью кнопки New. Откроется диалоговое окно (рис. П.9).

В процессе создания нового проекта в одном из окон введите название проекта и место его сохранения (рис. П.10).

Если на момент создания нового проекта в окне среды разработки никакой проект не открыт, то окно среды разработки будет выглядеть так, как показано на рис. П.11.

В стартовом окне среды разработки выберите команду Create New Project (рис. П.11). Далее действуйте так, как описано выше. Если вы создали пустой проект, то в этот проект следует добавить файл, в который будет вноситься код. Для этого в структуре проекта в окне среды разработки найдите папку src и добавьте в нее новый файл. Для этого выделите папку, нажмите правую кнопку мыши и в контекстном меню выберите подменю New, а в этом подменю — команду Java Class (рис. П.12).

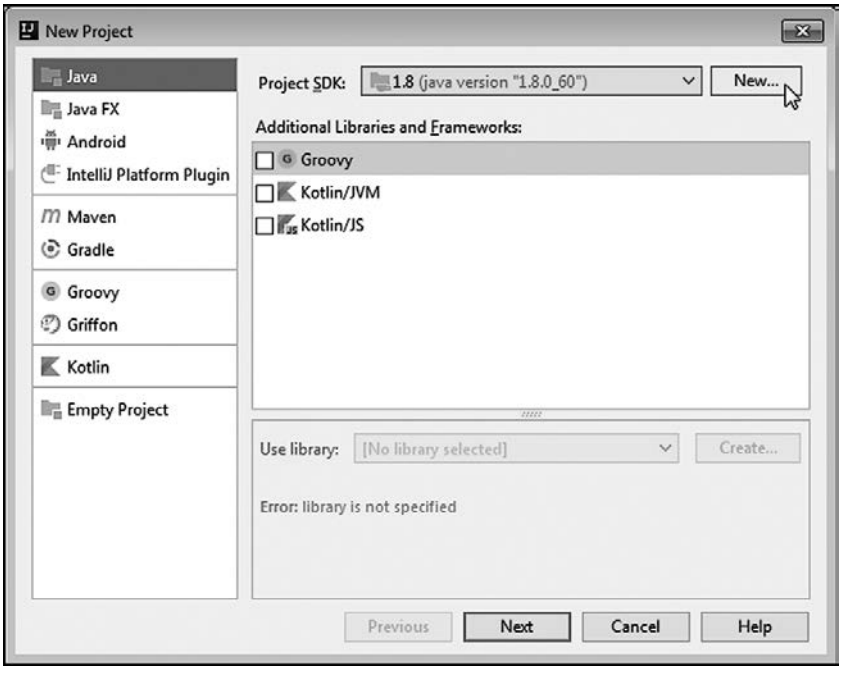


Рис. П.8. Окно создания нового проекта New Project

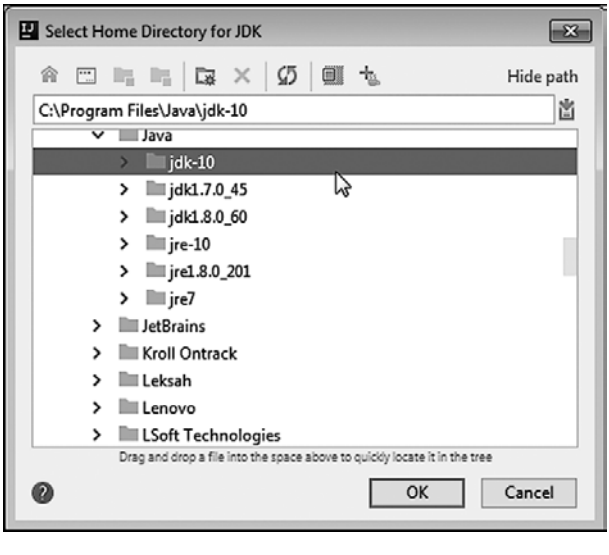


Рис. П.9. Выбор используемой системы JDK

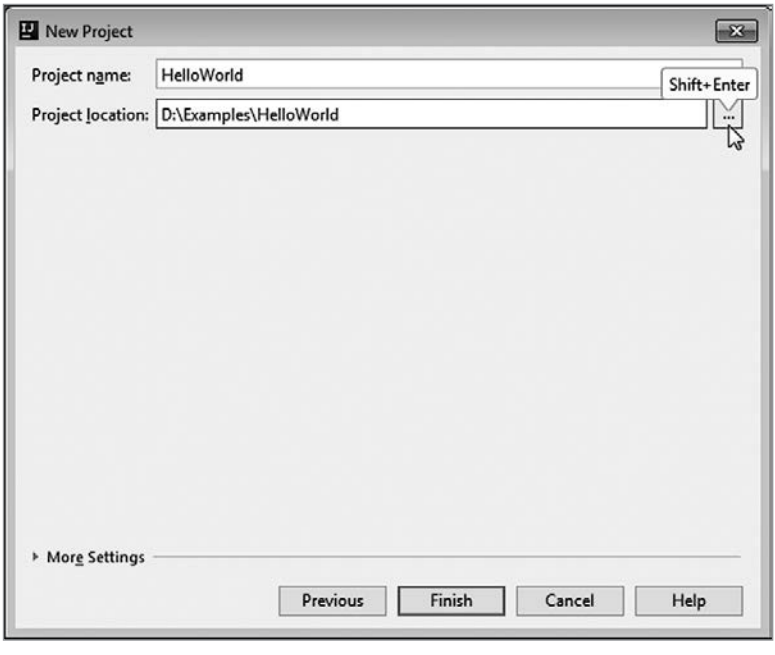


Рис. П.10. Определение названия проекта

Рис. П.11. Стартовое окно среды разработки без открытого текущего проекта

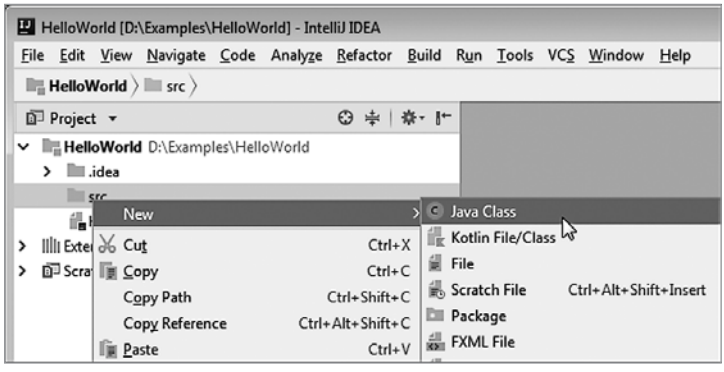


Рис. П.12. Добавление файла в проект

Откроется окно, где следует указать название файла (рис. П.13).

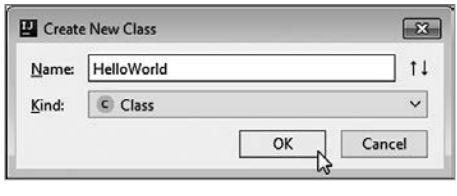


Рис. П.13. Определение названия добавляемого в проект файла

После того как файл создан, в него можно вводить код, который затем компилируется и выполняется. Для первой компиляции и запуска проекта можно воспользоваться контекстным меню для главного метода программы, как показано на рис. П.14.

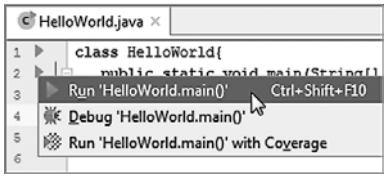


Рис. П.14. Запуск проекта с помощью контекстного меню главного метода программы

Также можно воспользоваться командой Run из меню Run (рис. П.15).

Можно воспользоваться командой Run из контекстного меню файла с кодом (рис. П.16).

На панели инструментов есть специальная кнопка с зеленой стрелкой (рис. П.17), нажатие которой (если она доступна) приводит к запуску кода.

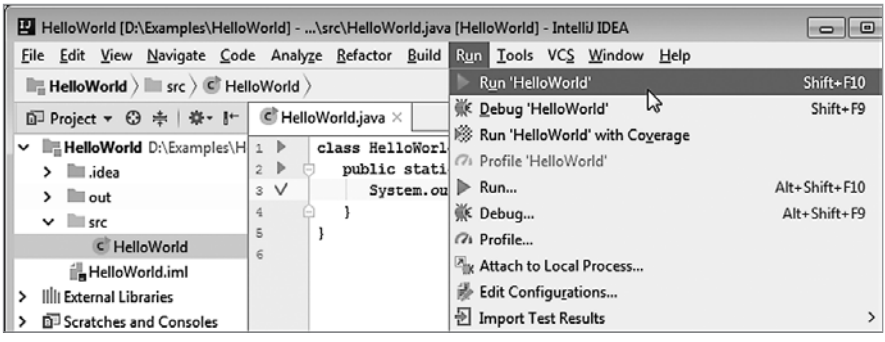


Рис. П.15. Запуск проекта с помощью команды Run из одноименного меню

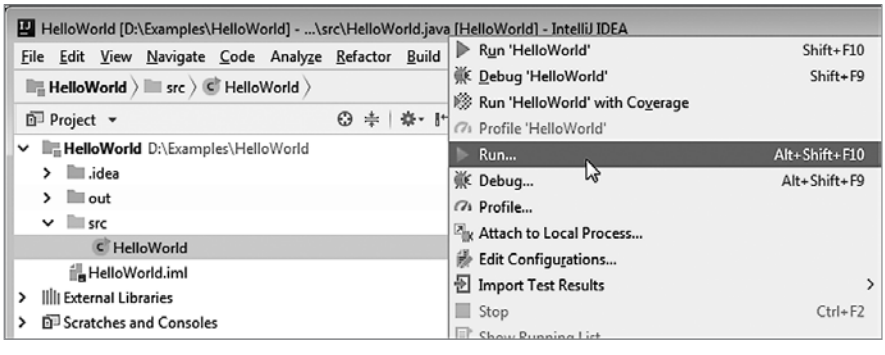


Рис. П.16. Запуск приложения с помощью команды Run из контекстного меню файла с кодом

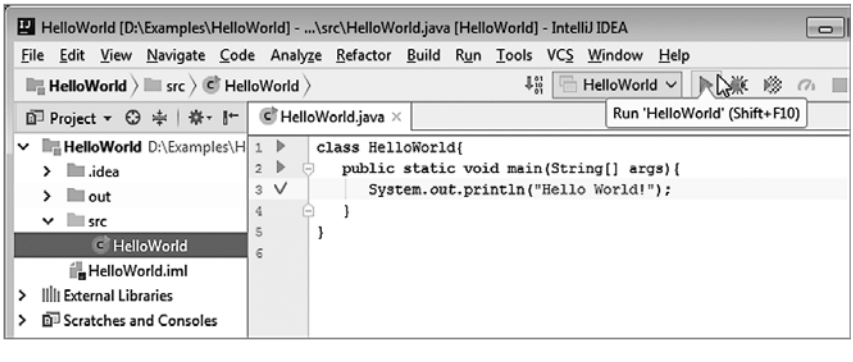


Рис. П.17. Запуск приложения нажатием кнопки на панели инструментов

Результат выполнения программы отображается в области вывода в нижней части рабочего окна среды разработки (рис. П.18).

Рис. П.18. Результат выполнения программы отображается в области вывода

НА ЗАМЕТКУ

Слева от результата выполнения программы в области вывода есть кнопка с зеленой стрелкой, нажатие которой также приводит к запуску программы.

Если при запуске программы нужно указать параметры (аргументы) командной строки, то необходимо сделать дополнительные настройки. В меню Run выберите команду Edit Configurations (рис. П.19).

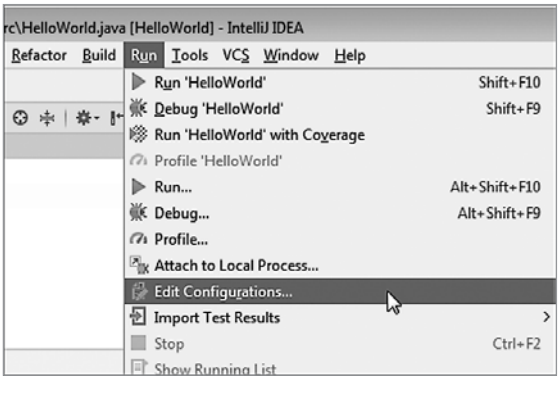


Рис. П.19. Выбор команды Edit Configurations из меню Run

Откроется окно Run/Debug Configurations (рис. П.20).

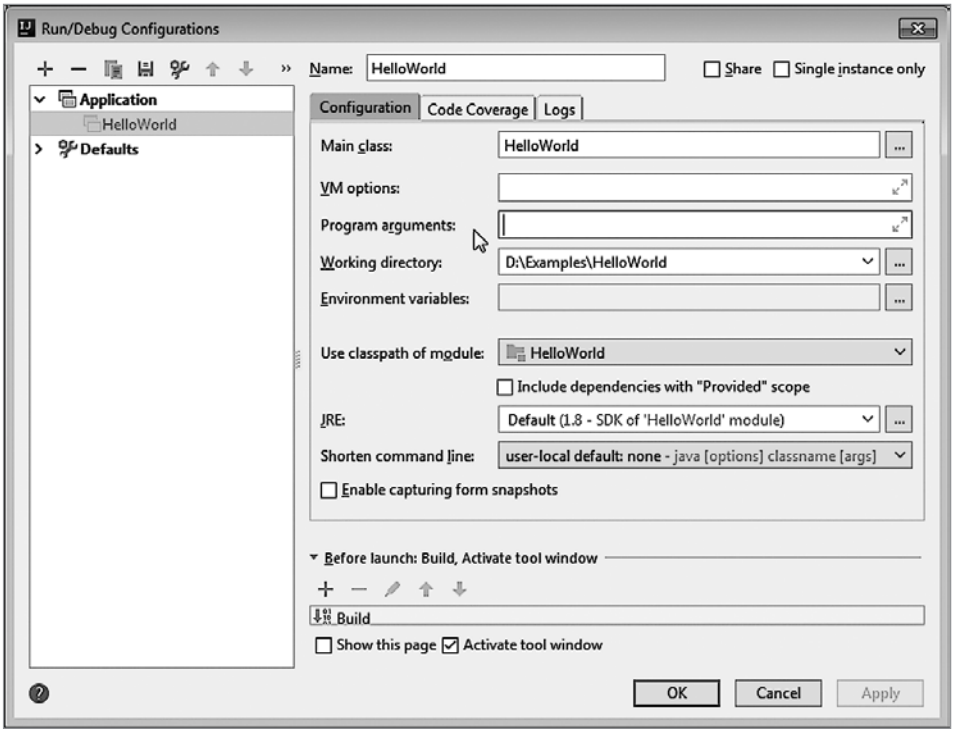


Рис. П.20. Окно Run/Debug Configurations с полем Program arguments для ввода параметров командной строки

В поле Program arguments вводятся параметры командной строки.

ПОДРОБНОСТИ

Параметры командной строки передаются в главный метод программы в виде текстового массива, который описывается как аргумент главного метода.

В некоторых случаях необходимо в явном виде задать кодировку символов, которая используется в среде разработки. В этом случае, например, можно воспользоваться специальной пиктограммой в правом нижнем углу окна (рис. П.21).

Для более основательных настроек можно воспользоваться командой Settings из меню File (рис. П.22).

Полезным также может быть подменю Other Settings из меню File (рис. П.23).

Чтобы сохранить изменения, внесенные в проект, выберите Save All в меню File (рис. П.24).

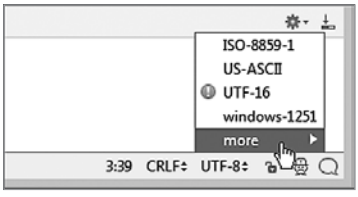


Рис. П.21. Меню для изменения кодировки

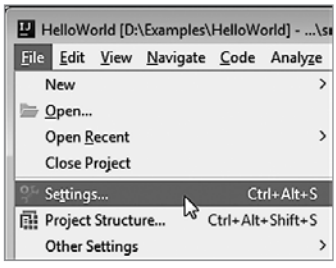


Рис. П.22. Выбор команды Settings из меню File

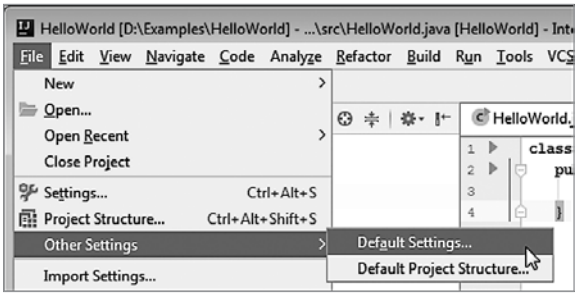


Рис. П.23. Содержимое подменю Other Settings из меню File

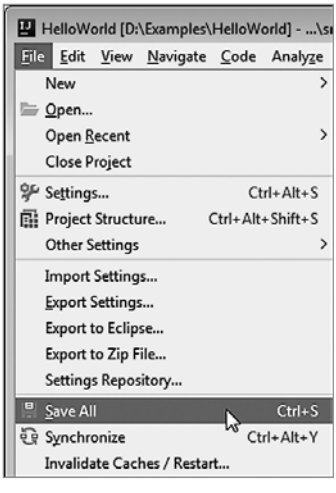


Рис. П.24. Выбор команды Save All из меню File

НА ЗАМЕТКУ

При компиляции программы изменения сохраняются автоматически.

Чтобы закрыть проект, выберите Close Project в меню File (рис. П.25).

Чтобы открыть уже существующий проект, выберите Open в меню File (рис. П.26).

Если на данный момент в среде разработки нет открытого проекта и отображается стартовое окно среды, то для открытия уже существующего проекта выберите команду Open (рис. П.27).

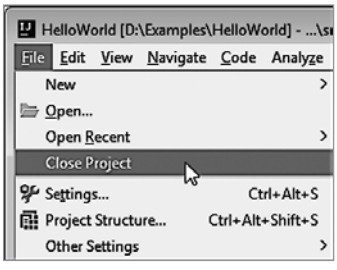


Рис. П.25. Для закрытия проекта выбираем команду Close Project из меню File

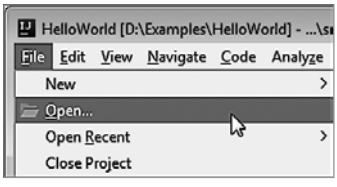


Рис. П.26. Выбор команды Open из меню File

Рис. П.27. Выбор команды Open в стартовом окне среды разработки

НА ЗАМЕТКУ

Если нужный проект представлен в списке ранее использовавшихся проектов в левой части стартового окна, можно открыть проект, выбрав его из списка.

Аналогичным образом выполняются прочие операции со средой разработки.

Алексей Васильев

Java для всех

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*Н. Римицан*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*М. Рогожин*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 08.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 09.08.19. Формат 70×100/16. Бумага офсетная. Усл. п. л. 41,280. Тираж 1500. Заказ 0000.