В книге рассматриваются все нововведения в Java 9 и объясняется, как ими пользоваться. Начав с подробного описания разработки приложений с использованием появившейся в Java 9 системы модулей, автор затем переходит к оболочке JShell, предназначенной для быстрого создания прототипа. Далее описываются все остальные новшества: измененная модульная структура образа среды выполнения JDK/JRE, новые фабричные методы для создания коллекций, уведомления процессора об активном ожидании с целью оптимизации потребления ресурсов, новый API платформенно-зависимого рабочего стола, API реактивных потоков и многое другое. Также уделено внимание несовместимым изменениям в Java 9.

Книга рассчитана на опытных Java-разработчиков, которым интересно, как перейти от Java 7 или 8 к Java 9.

Краткое содержание книги:

- новые возможности API процессов;
- инспекция стека потока с помощью API навигации по стеку;
- использование программы jlink для создания пользовательского образа среды выполнения;
- документация Java в формате HTML5 и новая функция поиска;
- новые методы и коллекторы в API потоков;
- создание пользовательского диспетчера протоколирования сообщений от платформенных классов и работа с журналами JVM;
- новые методы класса Optional;
- сравнение массивов и срезок массивов;
- усовершенствованные блоки try с ресурсами;
- повышение безопасности десериализации с помощью фильтров десериализации объектов.

Интернет-магазин: www.dmkpress.com
Книга-почтой: orders@alians-kniga.ru
Оптовая продажа: «Альянс-книга»
(499)782-3889, books@alians-kniga.ru

www.дмк.рф

apress
www.apress.com

ISBN 978-5-97060-575-2

Java 9. Полный обзор нововведений

Java 9. Полный обзор нововведений

Для быстрого ознакомления и миграции

Кишори Шаран

издательство

Кишори Шаран

Java 9
Полный обзор нововведений

Для быстрого ознакомления и миграции

Java 9 Revealed

For Early Adoption and Migration

Kishori Sharan

Apress®

Java 9
Полный обзор нововведений

Для быстрого ознакомления и миграции

Кишори Шаран

ДМК Пресс
издательство
Москва, 2018

УДК 004.438Java
ББК 32.973.26-018.1
Ш25

Ш25 Кишори Шаран
Java 9. Полный обзор нововведений. Для быстрого ознакомления и миграции. / пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2018. – 544 с.: ил.

ISBN 978-5-97060-575-2

В книге рассмотрены все нововведения в Java 9. Приведена разработка приложений с использованием системы модулей. Описаны измененная модульная структура образа среды выполнения JDK/JRE, новые фабричные методы для создания коллекций, уведомления процессора об активном ожидании с целью оптимизации потребления ресурсов, новый API платформенно-зависимого рабочего стола, API реактивных потоков и др. Уделено внимание несовместимым изменениям в Java 9. Также рассказано об оболочке JShell, предназначенной для быстрого создания прототипа.

Издание предназначено опытным Java-разработчикам, которым необходима миграция от Java 7 и 8 к Java 9.

УДК 004.438Java
ББК 32.973.26-018.1

ISBN 978-1-4842-2591-2 (англ.)
ISBN 978-5-97060-575-2 (рус.)

Моему другу Ричарду Кастилло, оказывавшему неоценимую помощь в моих писательских странствиях. Без него мне никогда бы не написать трехтомную эпопею «Harnessing Java 7». Спасибо, друг мой, за помощь.

Оглавление

Об авторе

Кишори Шаран работает старшим программистом и руководителем группы в компании IndraSoft, Inc. Ученую степень магистра по компьютерным информационным системам получил в Тройском университете, штат Алабама. Имеет сертификат программиста на Java 2 от компании Sun. Свыше 20 лет занимается разработкой корпоративных приложений и обучением профессиональных разработчиков на платформе Java.

О техническом рецензенте

Мануэль Джордан Элера – программист и исследователь-самоучка, обожающий изучать новые технологии ради собственного удовольствия и создания новых способов интеграции.

Мануэль получил звание Springy Award-Community Champion в 2010 году и Spring Champion в 2013. В немногие свободные минуты он читает Библию и сочиняет гитарные композиции. Выступал в качестве технического рецензента ряда книг издательства Apress: «Pro Spring», издание 4 (2014), «Practical Spring LDAP» (2013), «Pro JPA 2», издание 2 (2013) и «Pro Spring Security (2013).

Написанные им 13 подробных пособий по различным технологиям Spring можно прочитать в блоге по адресу `http://www.manueljordanelera.blogspot.com` и там же задать ему вопросы. Его адрес в Твиттере `@dr_pompeii`.

Благодарности

Моя жена Эллен терпеливо сносила долгие часы, которые я проводил за компьютером, работая над этой книгой. Я благодарен ей за поддержку.

Хочу выразить признательность за ободрение и поддержку своим родственникам: маме, Пратима Деви, старшим братьям, Джанки Шаран и д-ру Сита Шарану, племянникам, Гаураву и Саураву, сестре Ратне, а также друзьям Картхикейя Венкатесану, Притхи Васудеву, Рахулу Нагпалу, Рави Датла и многим-многим другим.

Я благодарен своим коллегам: Форресту «Корки» Баттсу, Антонио Мэтьюсу и Иоланде Робинсон за поддержку.

Я искренне признателен замечательной команде в издательстве Apress за поддержку в течение всего периода подготовки книги к печати. Спасибо Марку Пауэрсу, начальнику отдела редактирования, который всегда был рад помочь и проявлял чудеса терпения, когда я задерживал рукопись. Отдельное спасибо Мануэлю Джордану Элера, моему техническому рецензенту, скрупулезность которого помогла устранить немало технических ошибок.

И наконец, я благодарю Стива Энглина, главного редактора Apress, предложившего мне написать эту книгу.

Предисловие

Как появилась эта книга

Сообщество Java и я вместе с ним с радостью восприняли известие о том, что в JDK 9 наконец-то добавлена система модулей и оболочка Java. Больше десяти лет мы мечтали посмотреть на живую систему модулей. В нескольких предыдущих выпусках JDK были ее прототипы, но впоследствии от них отказывались. Включение модулей в JDK 9 также было усыпано терниями. Было несколько раундов предложений и прототипов. Эту книгу я начал писать в начале 2016 года и должен признать, что предприятие оказалось непростым. Я вынужден был, с одной стороны, опередить выпуск JDK, а, с другой, учитывать все изменения, которые разработчики Java вносили в систему модулей. Я боялся, что книга выйдет спустя несколько месяцев после выпуска окончательной версии JDK 9, а весь материал при этом устареет. Сейчас на дворе последний день февраля 2017 года и, похоже, пыль наконец улеглась – разработчики и сообщество довольны системой модулей, и книга выйдет в ее нынешнем виде. Выпуск JDK 9 намечен на конец июля 2017 года. На данный момент функциональность JDK 9 полностью определена, и маловероятно, что случаев, когда что-то описанное в книге не работает, будет много. Однако на шкале выпуска версий ПО пять месяцев – долгий срок, так что не удивляйтесь, если придется внести в код кое-какие мелкие изменения по сравнению с тем, что написано в книге.

Первоначально предполагалось, что в книге будет 140 страниц. Но по ходу дела я понял, что было бы нечестно по отношению к читателям написать такую коротенькую книжку об одном из самых крупных изменений на платформе Java. Я благодарен издательству, не возражавшему против увеличения объема на несколько сотен страниц. Девять глав (со второй по десятую) я посвятил исключительно описанию новой системы модулей. В главе 11 очень подробно описывается оболочка Java (JShell).

Я потратил бесчисленные часы на изучение этой темы. Я писал о частях, которые только еще разрабатывались. Ни в Интернете, ни в книгах не было ничего по этому вопросу. Самым сложным было поспевать за быстро изменяющейся реализацией. Основными моими источниками были исходный код Java, документы, публикуемые как часть процесса совершенствования Java (Java Enhancement Process, JEP), и запросы об изменении спецификации Java (Java Specification Request, JSR). Много времени я затратил на чтение исходного кода Java, чтобы больше узнать о некоторых нововведениях в JDK 9. Всегда было любопытно сначала поэкспериментировать с программами на Java, иногда в течение нескольких часов, а потом добавить их код в книгу. Случались и горькие разочарования, когда код, работавший еще неделю назад, вдруг переставал работать. Подписка на списки рассылки по всем проектам, относящимся к JDK 9, помогала мне идти вровень с разработчиками JDK. Несколько раз я вынужден был просматривать списки всех ошибок в различных проектах JDK, чтобы убедиться, что нашел еще не исправленную ошибку.

Но все хорошо, что хорошо кончается. Наконец-то я с удовлетворением могу сказать, что включил все, что может понадобиться читателям, желающим изучить Java SE 9. Надеюсь, что книга вам понравится и окажется полезной.

Об организации книги

Книга состоит из 20 глав. Глава 1 содержит введение в JDK 9 и рекомендации о том, как работать с книгой и включенным в нее исходным кодом. В главах 2–10 описывается система модулей. Их лучше читать последовательно.

Глава 11 содержит подробное описание оболочки Java (JShell). Для чтения большей ее части знакомство с первыми 10 главами не обязательно. Однако для формирования полной картины я рекомендую читать их подряд.

Главы с 12 по 20 можно читать в любом порядке, т. к. рассматриваемые в них темы независимы. И в последнюю главу вошли вопросы, для которых не нашлось места в других главах и не стоило заводить отдельную главу.

Предполагаемая аудитория

Книга рассчитана на опытных Java-разработчиков, знакомых с Java SE 8. В отдельных местах используются Streams API, Time API и лямбда-выражения, появившиеся в Java SE 8. Но и знакомые только с Java SE 7 или еще более ранними версиями, безусловно, получат пользу от чтения глав 1–11. Из них вы узнаете о системе модулей и оболочке Java.

При чтении книги рекомендуется иметь под рукой документацию по API языка Java. Там вы найдете полный перечень всех библиотечных классов. Документацию можно скачать (или просмотреть) на официальном сайте Oracle по адресу `www.oracle.com`. В процессе чтения рекомендуется попрактиковаться в самостоятельном написании программ на Java, например, путем модификации приведенного в книге кода. От чтения без практики толку мало. Помните, что «навык мастера ставит», это в полной мере относится и к изучению программирования с применением JDK 9.

Исходный код и ошибки

Исходный код и перечень найденных ошибок и опечаток можно найти по адресу `http://www.apress.com/us/book/9781484225912`.

Вопросы и замечания

Все вопросы и замечания отправляйте автору по адресу `ksharan@jdojo.com`.

Глава 1

Введение

Краткое содержание главы:

- что включено в JDK 9;
- как читать эту книгу;
- требования к системе;
- как установить NetBeans;
- как скачать и использовать исходный код примеров.

Введение в JDK 9

JDK – девятая основная версия Java Development Kit, выпуск которой намечен на конец июля 2017 года. В нее входит несколько важных новых возможностей. В этой книге описаны те возможности, которые полезны разработчикам Java-приложений в их повседневной работе. Полный перечень всех изменений опубликован по адресу `http://openjdk.java.net/projects/jdk9/`.

Одна из самых важных и долгожданных возможностей JDK 9 – *система модулей*, разрабатывавшаяся в рамках проекта с кодовым названием *Jigsaw*. Проектировщики JDK пытались включить систему модулей в платформу Java более 10 лет. Планировалось, что она станет частью предыдущих версий JDK, но каждый раз что-то срывалось. Из-за включения Jigsaw даже несколько раз переносились сроки выпуска JDK 9. Но наконец-то Jigsaw вышел, и скоро вы сможете увидеть его в действии.

Что же такого трудного в Jigsaw, что на доведение до ума понадобилось много лет? Основная цель Jigsaw – предоставить программистам методику разработки приложений в терминах компонентов, именуемых *модулями*. Модуль может экспортировать свой открытый API и инкапсулировать внутренние детали. Модуль также может объявить зависимости от других модулей, и они будут проверяться на этапе инициализации, что позволит впоследствии избежать сюрпризов из-за отсутствия типов. Сам JDK представлен в виде набора взаимодействующих модулей, так что среда выполнения оказывается масштабируемой. Теперь, если в приложении используется только подмножество JDK, то можно создать образ среды выполнения, который будет содержать только необходимые модули JDK и модули приложения. Во всем этом нет ничего удивительного. Главная же пробле-

ма, преследовавшая проектировщиков JDK, заключалась в обеспечении обратной совместимости и порядка перехода на систему модулей. Язык Java существует уже больше 20 лет. Для любого крупного нововведения, в т. ч. системы модулей, необходимо обеспечить преемственность. В результате нескольких итераций и уточнений, основанных на замечаниях сообщества, цель наконец-то достигнута! Я посвятил 9 глав (со 2 по 10), т. е. свыше 40% книги описанию системы модулей.

Еще одно важное новшество в JDK 9 – оболочка JShell, реализующая цикл чтения-выполнения-печати (Read-Eval-Print Loop, REPL) для Java. JShell – одновременно командная утилита и API, который позволяет выполнить фрагмент кода и сразу получить результат. До появления JShell для получения результата необходимо было написать полную программу, откомпилировать и выполнить ее. JShell – инструмент, которым вы будете пользоваться постоянно, удивляясь, как столько лет обходились без него. JShell – бесценное подспорье для начинающих, поскольку поможет им быстро выучить язык, не вдаваясь в такие детали структуры программы, как модули и пакеты. Обсуждению JShell посвящена глава 11.

В JDK 9 есть еще несколько нововведений, облегчающих жизнь разработчика, в том числе: API реактивных потоков, методы фабрики коллекций, инкубаторный клиентский API протокола HTTP/2, API навигации по стеку, API платформенного протоколирования и унифицированного протоколирования JVM. Все эти темы подробно обсуждаются в книге.

Как читать эту книгу

Книга состоит из 20 глав. Главы со второй по десятую посвящены одной теме – системе модулей. Их рекомендуется читать последовательно. Конечно, можно перенести существующее приложение на JDK 9, вообще ничего не зная о модулях. Но если вам хочется получить преимущества, которые дает JDK 9, то необходимо изучить, что такое модули, и начать пользоваться ими при разработке новых и переносе старых приложений. Главы с 11 по 20 можно читать в любом порядке.

Глава 2 «Система модулей» содержит краткую историю проблем, с которыми сталкивались разработчики в JDK версии 8 и более ранних. Здесь же дается введение в систему модулей и объясняется, как она решает проблемы инкапсуляции API, конфигурирования и масштабируемости. Вводится много новых терминов, относящихся к системе модулей. Даже если вы их не понимаете, продолжайте читать. Более полные пояснения будут даны в последующих главах.

В *главе 3 «Создаем свой первый модуль»* вы увидите модули в действии. Будет показано, как объявить модуль, написать его код, откомпилировать и выполнить класс, являющийся частью модуля. Но в детали объявления модулей мы пока не вдаемся.

Глава 4 «Зависимости модулей» представляет собой интенсивный практический курс по системе модулей. Она опирается на две предыдущие главы и содержит более полное изложение вопроса о зависимостях модулей и экспорте пакетов из модуля. Показано, как объявляются модули разных типов, как смешивать модульный и немодульный код в одном приложении и еще несколько возможностей системы модулей. Материал, изложенный в этой главе, должен понимать любой серьезный Java-разработчик, планирующий использовать JDK 9.

В *главе 5 «Реализация служб»* показано, как с помощью простых директив системы модулей реализовать службы в JDK 9. Здесь же объясняется, как службы реализовывались раньше.

В *главе 6 «Упаковка модулей»* описываются различные форматы упаковки модулей и объясняется, когда какой формат лучше использовать. Вплоть до версии JDK 8 существовал только один способ упаковки Java-приложений – JAR-файлы. В этой главе показано, как упаковывать модули в файл формата JMOD.

В *главе 7 «Создание пользовательских образов среды выполнения»* описано, как создавать образы среды выполнения во внутреннем формате JIMAGE с помощью программы `jlink`.

В *главе 8 «Несовместимые изменения в JDK 9»* описываются изменения, которые могут привести к тому, что приложения, перенесенные на JDK 9, перестанут работать. Здесь же даются рекомендации, как исправить код, и сообщается о предлагаемых альтернативах, если таковые существуют.

В *главе 9 «Нарушение инкапсуляции модуля»* объясняется, как можно обойти главный постулат системы модулей – инкапсуляцию. JDK 9 предлагает несколько нестандартных параметров командной строки, которые позволяют получить доступ к инкапсулированному и вообще-то недоступному извне коду модуля. Это может понадобиться при переносе унаследованных приложений, ранее имевших доступ к внутренним API JDK или сторонних библиотек, которые после перехода на JDK 9 оказались инкапсулированы в модулях.

В *главе 10 «API модулей»* речь пойдет о том, как обращаться к системе модулей из программы с помощью API модулей (Module API). Все, что можно сделать посредством объявлений модулей и параметров командной строки, доступно и через API модулей.

Глава 11 «Оболочка Java» посвящена программе JShell и JShell API. Сюда входит печать сообщений, объявление классов и использование существующих модулей внутри `jshell`. Описывается, как конфигурировать `jshell` и как использовать JShell API для выполнения фрагментов кода.

В *главе 12 «Изменения API процессов»* описываются новые возможности API процессов (Process API) в JDK 9. Полного описания API процессов в том виде, в каком он существует со времен JDK 1.0, вы здесь не найдете.

В *главе 13 «Изменения API коллекций»* описываются новые возможности API коллекций (Collection API) в JDK 9, в т. ч. статические фабричные методы создания списков, множеств и отображений.

Глава 14 «Клиентский API HTTP/2» посвящена клиентскому API новой версии протокола, HTTP/2. Это часть инкубаторного, т. е. еще не окончательно стандартизованного, модуля. В версии JDK 10 он либо будет стандартизован и станет полноправной частью платформы Java SE, либо будет исключен.

В *главе 15 «Модифицированный тип Deprecated»* описываются изменения в типе аннотаций `Deprecated`, порядок использования аннотации `@SuppressWarnings` в JDK 9 и сканирование кода для поиска нерекомендованных API с помощью программы `jdeprscan`.

В *главе 16 «Навигация по стеку»* рассматривается API навигации по стеку (Stack-Walking API), появившийся в версии JDK 9. Его можно использовать для обхода стека и нахождения ссылок на вызывающие классы.

В *главе 7 «Реактивные потоки»* речь идет об API реактивных потоков (Reactive Streams API) – реализации спецификации реактивных объектов в Java.

В *главе 18 «Изменения API потоков»* описываются новые методы, добавленные в API потоков (Streams API). Рассматриваются также новые коллекторы в API потоков.

В *главе 19 «Протоколирование на уровне платформы и JVM»* описывается новый API платформенного протоколирования, позволяющий отправлять сообщения из платформенных классов пользовательскому диспетчеру протоколирования. Здесь показано, как использовать для этой цели библиотеку Log4j. В этой же главе показано, как обращаться к журналам JVM с помощью параметра командной строки и как настроить сообщения JVM перед записью в журнал.

В *главе 20 «Прочие изменения в JDK 9»* собраны все остальные изменения, представляющие интерес для разработчиков приложений и не нашедшие отражения в предыдущих главах. В каждом разделе рассматривается одна тема. Можете прочитать только то, что вам интересно, а остальное пропустить. К вопросам, обсуждаемым в этой главе, относятся использование знака подчеркивания в качестве ключевого слова, улучшенные блоки `try` с ресурсами, закрытые методы в интерфейсах, потоки объектов типа `Optional`, использование фильтров в процессе десериализации объектов, новые методы в API ввода-вывода и др.

Требования к системе

Чтобы выполнять описанные в книге программы и писать свои собственные, необходимо установить на свой компьютер JDK 9.

На момент написания книги JDK 9 еще разрабатывался. Выпуск производственной версии запланирован на конец июля 2017 года. Текущая сборка называется *Early Access* Build 157. Скачать самую свежую версию можно со страницы по адресу `https://jdk9.java.net/download/`. Там же приведены инструкции по установке JDK 9 для разных операционных систем.

Чтобы проверить версию установленного JDK, выполните команду:

```
c:\> java -version
```

```
java version "9-ea"
Java(TM) SE Runtime Environment (build 9-ea+157)
Java HotSpot(TM) 64-Bit Server VM (build 9-ea+157, mixed mode)
```

Обратите внимание на строку `9-ea`. Она означает, что установлена версия JDK 9 Early Access. В окончательной версии будет указан только номер 9.

Возможно, что при выполнении команды будет выдана ошибка вида:

```
c:\> java -version
```

```
'java' is not recognized as an internal or external command,
operable program or batch file.
```

Она означает, что каталог, содержащий команду `java`, не включен в переменную среды `PATH`. Команда `java` находится в каталоге `JDK_HOME\bin`, где `JDK_HOME` – каталог, в который установлен JDK 9. На моей системе Windows это каталог `C:\java9`. Для добавления каталога `JDK_HOME\bin` в переменную среды `PATH` в каждой операционной системе есть свои методы. Можно также указывать полный путь к команде, например: `C:\java9\bin\java` в Windows. Ниже для печати сведений о версии указан полный путь к команде `java`:

```
c:\> C:\java9\bin\java -version
```

```
java version "9-ea"
Java(TM) SE Runtime Environment (build 9-ea+157)
Java HotSpot(TM) 64-Bit Server VM (build 9-ea+157, mixed mode)
```

Установка NetBeans IDE

Для компиляции и выполнения программы из книги понадобится интегрированная среда разработки (IDE), например, NetBeans или Eclipse. На момент написания книги ни одна IDE не поддерживала систему модулей в полном объеме. В настоящее время NetBeans поддерживает создание одного модуля на проект. Следовательно, если вы хотите создать в NetBeans три Java-модуля, то нужно будет создать три проекта. При этом один модуль может ссылаться на другие, пользуясь зависимостями проекта. Актуальную информацию о поддержке модулей в NetBeans можно найти на вики-странице `http://wiki.netbeans.org/JDK9Support`.

Скачать самую свежую сборку NetBeans с поддержкой JDK 9 можно по ссылке `http://bits.netbeans.org/download/trunk/nightly/latest/`.

На этой странице можно скачать файлы разных типов. Самый маленький «весит» меньше 100 МБ и содержит Java SE. Он годится во всей книге, кроме главы 14, где рассматривается веб-приложение. Для выполнения примеров из главы 14 понадобится версия NetBeans для Java EE.

NetBeans работает поверх JDK версий 8 и 9. Я рекомендую устанавливать JDK 9 раньше NetBeans. Если на машине установлено несколько версий JDK, то NetBeans по умолчанию выбирает JDK 8. Чтобы использовался JDK 9, его установочный каталог нужно указать явно. Для случая, когда NetBeans устанавливался поверх JDK 8, в главе 3 имеются пошаговые инструкции, как использовать JDK 9 в качестве платформы Java в NetBeans. Там, где нужно, я буду приводить снимки окон NetBeans, чтобы помочь вам в создании программ и настройке IDE для их компиляции и выполнения.

Скачивание исходного кода

Исходный код примеров можно скачать по адресу `http://www.apress.com/us/book/9781484225912`. Это ZIP-файл, в котором корневой каталог называется `Java9Revealed`. Пользователям Windows я рекомендую распаковать архив в каталог `C:\`. В результате будет создан каталог `C:\Java9Revealed`, и в нем будет находиться весь исходный код.

Во всех примерах предполагается, что исходный код находится в каталоге `C:\Java9Revealed`. Если вы работаете не в Windows или скопировали исходный код не в каталог `C:\`, то в коде нужно будет заменить путь к каталогу с учетом платформенных соглашений о записи пути в файловой системе. Так, в Linux вместо `C:\Java9Revealed` нужно будет указать путь вида `/usr/ks/Java9Revealed`.

Каждый модуль в исходном коде представляет собой отдельный проект NetBeans и размещается в отдельном каталоге. Иногда в примерах используется несколько модулей, тогда будет и несколько каталогов. Так, в главе 5 создается пять модулей: `com.jdojo.prime`, `com.jdojo.prime.generic`, `com.jdojo.prime.faster`, `com.jdojo.prime.probable` и `com.jdojo.prime.client`. Соответственно в исходном коде имеется пять каталогов:

- `Java9Revealed/com.jdojo.prime`
- `Java9Revealed/com.jdojo.prime.generic`
- `Java9Revealed/com.jdojo.prime.faster`
- `Java9Revealed/com.jdojo.prime.probable`
- `Java9Revealed/com.jdojo.prime.client`

Каждый каталог с проектом NetBeans содержит подкаталоги `build`, `dist` и `src`. В подкаталоге `src` находится исходный код модуля, в `build` – его откомпилированный код, в `dist` – JAR-файл модуля. Примеры в этой книге ссылаются на подкаталоги `build` и `dist`. Если примеры работают неправильно, то, возможно, все наладится, если открыть проект в NetBeans, выполнить команду **Clean and Build** (Очистить и собрать), которая очищает подкаталоги `build` и `dist`, перекомпилирует проект в каталоге `build` и заново создает JAR-файл в каталоге `dist`. Рассмотрим следующую команду:

```
C:\Java9Revealed> java --module-path com.jdojo.intro\build\classes
 --module com.jdojo.intro/com.jdojo.intro.Welcome
```

Здесь предполагается, что `C:\Java9Revealed` – текущий каталог, т. е. относительному пути `com.jdojo.intro\build\classes` соответствует полный путь `C:\Java9Revealed\com.jdojo.intro\build\classes`. Встретив подобные команды в книге, заменяйте относительный путь в соответствии с тем, какой каталог является текущим и где находится исходный код.

Есть также возможность выполнять классы прямо в NetBeans, что гораздо проще, чем из командной строки. Некоторые примеры строятся в книге пошагово, и я показываю, что получается на каждом шаге. В таких случаях в исходном коде приводится окончательный вариант программы, а чтобы воспроизвести результаты на каждом шаге, вам придется отредактировать код.

Очень может статься, что формат файла откомпилированного класса немного изменится к тому моменту, когда вы будете их использовать. Я настоятельно рекомендую открывать каждый проект в NetBeans IDE и выполнять команду **Clean and Build**, прежде чем пытаться их запустить.

Глава 2

Система модулей

Краткое содержание главы:

- как производилось написание, упаковка и развертывание кода на Java до выхода JDK 9 и какие при этом возникали проблемы;
- что такое модуль в JDK 9;
- как объявляются модули и их зависимости;
- как упаковываются модули;
- что такое путь к модулю;
- что такое видимые модули;
- как напечатать список видимых модулей;
- как напечатать описание модуля.

Жизнь до Java 9

До выхода JDK 9 разработка Java-приложения состояла из следующих шагов.

- Написать код на Java, состоящий из различных типов: классов, интерфейсов, перечислений, аннотаций и т. п.
- Распределить типы по пакетам. Каждый тип принадлежит некоторому пакету – явно или неявно. Пакет – это логический контейнер типов, по существу он является пространством имен для находящихся в нем типов. В пакете могут быть открытые и закрытые типы, а также типы, которые, хотя и объявлены открытыми, на самом деле содержат внутренние реализации.
- Упаковать откомпилированный код типов в один или несколько JAR-файлов. Код, принадлежащий одном пакету, может размещаться в нескольких JAR-файлах.
- В приложении могут использоваться библиотеки, которые также поставляются в виде одного или нескольких JAR-файлов.
- Произвести развертывание приложения, т. е. копирование всех JAR-файлов, прикладных и библиотечных, в каталоги, перечисленные в пути к классам.

На рис. 2.1 показана типичная организация кода, упакованного в один JAR-файл. Представлены только пакеты и типы, а прочие артефакты, в частности файл `manifest.mf` и файлы ресурсов, опущены.

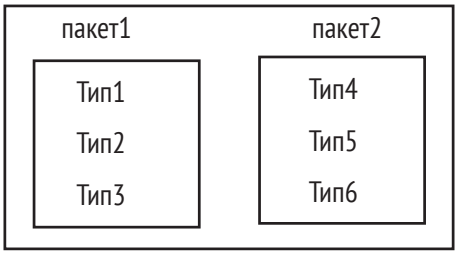

Рис. 2.1. Организация кода в JAR-файле

Свыше 20 лет сообщество Java разрабатывало код именно так. Но эта жизнь не была такой безоблачной, как хотелось бы! Перечислим проблемы, которые возникали при таком способе организации и выполнения кода.

- Пакет – это всего лишь контейнер типов, он не определяет границы доступности. Открытый тип, находящийся в некотором пакете, доступен всем остальным пакетам, не существует никакого способа ограничить глобальную видимость открытых типов.
- Все пакеты, кроме тех, чьи имена начинаются с java и javax, открыты для расширения. Если в вашем JAR-файле имеются типы с пакетным уровнем доступа, то они будут видны в других JAR-файлах, где определены типы, принадлежащие пакету с таким же именем, как ваш.
- Среда выполнения Java видит плоский набор пакетов, загруженных из списка JAR-файлов. Нет никакого способа узнать, существуют ли разные экземпляры одного и того же типа в разных JAR-файлах. Среда выполнения загрузит тип из того файла, который указан в пути к классам первым.
- Если какой-нибудь JAR-файл не включен в путь к классам, то среда выполнения выдаст сообщение об отсутствующих типах, причем это произойдет не на этапе компиляции, а на этапе выполнения, когда программа попытается обратиться к такому типу.
- На этапе инициализации приложение никак не может узнать, что некоторые используемые типы отсутствуют. Кроме того, можно включить неправильную версию JAR-файла, что закончится ошибкой на этапе выполнения.

Эти проблемы встречаются так часто и пользуются такой печальной известностью, что получили в сообществе Java специальное название – *ад JAR*. Я попадал в этот ад несколько раз на протяжении своей карьеры! Термину «ад JAR» даже посвящен специальный раздел статьи в википедии – см. https://en.wikipedia.org/wiki/Java_Classloader#JAR_hell.

Упаковка JDK и JRE тоже составляет проблему. Они поставлялись в виде гигантских монолитов, что увеличивает время загрузки, время инициализации и объем потребляемой памяти. Из-за монолитного JRE Java-приложения невозможно использовать на устройствах с небольшим объемом памяти. Если Java-приложение развертывается в облаке, то приходится платить за потребляемую память. Чаще всего монолитный JRE занимает больше памяти, чем необходимо, а это значит, что вы платите лишнее за предоставление облачных служб. Компактные профили, появившиеся в Java 8, – шаг в направлении уменьшения размера JRE, а, значит, и памяти, потребляемой средой выполнения. Это достигается путем упаковки

подмножества JRE в специальный образ среды выполнения, называемый *компактным профилем*.

> **Совет.** В ознакомительных версиях JDK 9 было три модуля: `java.compact1`, `java.compact2` и `java.compact3`, соответствующих трем компактным профилям JDK 8. Впоследствии от них отказались, потому что модули в JDK дают полный контроль над списком модулей, включаемых в пользовательский JRE.

Проблемы, мучившие JDK/JRE, до появления JDK 9 можно отнести к трем категориям:

- ненадежная конфигурация;
- слабая инкапсуляция;
- монолитная структура JDK/JRE.

На рис. 2.2 показано, как среда исполнения Java видит JAR-файлы на пути к классам и как к коду, находящемуся в одном JAR-файле, можно обратиться из других JAR-файлов без каких-либо ограничений, кроме заданных в объявлениях типов в терминах контроля доступа.

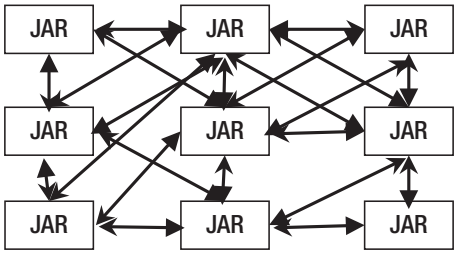

Рис. 2.2. JAR-файлы на пути к классам: загрузка средой выполнения и доступ

В Java 9 эти проблемы решаются посредством нового способа разработки, упаковки и развертывания приложений. Приложение в Java 9 состоит из небольших взаимодействующих компонентов, называемых *модулями*. JDK/JRE в Java 9 тоже организованы в виде набора модулей. В этой главе мы познакомимся с модулями в общих чертах, а в последующих рассмотрим их подробно.

Новая система модулей

В Java 9 введен новый компонент программы – *модуль*. Теперь приложение можно рассматривать как набор взаимодействующих модулей с четко определенными границами и зависимостями. При разработке системы модулей ставились следующие цели:

- надежная конфигурация;
- строгая инкапсуляция;
- модульная организация JDK/JRE.

Надежная конфигурация решает проблему провоцирующего ошибки механизма пути к классам, который используется для поиска типов. Модуль обязан явно объявить зависимости от других модулей. Система модулей проверяет зависимос-

ти на всех этапах разработки приложения: компиляции, компоновки и выполнения. Если один модуль объявил зависимость от другого, а этот другой модуль отсутствует на этапе инициализации приложения, то JVM обнаружит отсутствие зависимости и откажется запускать приложение. До Java 9 ошибка возникла бы на этапе выполнения (а не инициализации) – при попытке воспользоваться отсутствующими типами.

Строгая инкапсуляция решает проблему беспрепятственного доступа к открытым типам через границы JAR-файлов на пути к классам. Модуль должен явно объявить, какие из находящихся в нем открытых типов доступны другим модулям, ко всем остальным открытым типам обратиться из других модулей невозможно. В Java 9 слово `public` еще не означает, что тип доступен из любой части программы. Система модулей уточняет контроль доступности.

> **Совет.** В Java 9 надежная конфигурация достигается за счет того, что модуль явно объявляет зависимости, а система проверяет их на всех этапах разработки. Для обеспечения строгой инкапсуляции модуль объявляет, в каких пакетах могут находиться открытые типы, доступные другим модулям.

JDK 9 переписан и вместо монолитной структуры представляет собой набор так называемых *платформенных модулей*. Введен факультативный *этап компоновки*, который может находиться между этапами компиляции и выполнения. На этом этапе используется компоновщик `jlink`, входящий в состав JDK 9, который создает пользовательский образ среды исполнения, включающий только модули, используемые в приложении. В результате оптимизируется размер среды исполнения.

Что такое модуль?

Модуль – это именованная коллекция кода и данных. Он может содержать Java-код и платформенный код. Java-код организован в виде набора пакетов, содержащих типы: классы, интерфейсы, перечисления и аннотации. В состав данных могут входить ресурсы, например, графические и конфигурационные файлы.

С точки зрения Java-кода, модуль предстает в виде набора, содержащего нуль или более пакетов. На рис. 2.3 показано три модуля: `policy`, `claim` и `utility`, причем `policy` содержит два пакета, `claim` – один пакет, а `utility` – ни одного.

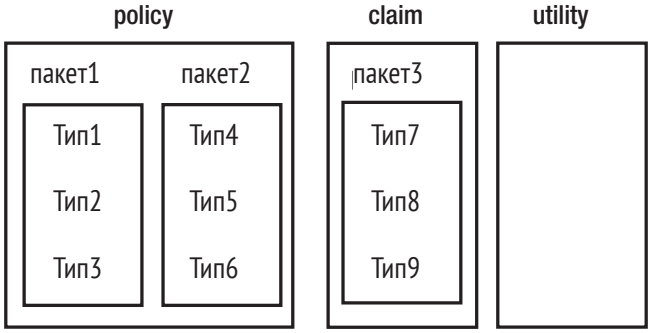

Рис. 2.3. Типы, пакеты и модули

Модуль – не просто контейнер пакетов. Помимо имени, определение модуля содержит:

- список *требуемых* модулей (от которых зависит данный);
- список *экспортируемых* пакетов (открытый API), которые могут использоваться другими модулями;
- список *раскрываемых* (с полным API, открытым и закрытым) пакетов, доступных другим модулям для рефлексии;
- список *используемых* (обнаруженных и загруженных с помощью класса `java.util.ServiceLoader`) служб;
- список предоставляемых реализаций служб.

Работа с модулем сводится к работе с одним или несколькими вышеперечисленными аспектами.

В спецификации Java SE 9 платформа представлена в виде набора *платформенных модулей*. Конкретная реализация Java SE 9 может содержать все или некоторые платформенные модули, что обеспечивает масштабируемость среды исполнения Java. Имена стандартных модулей начинаются словом `java`, например: `java.base`, `java.sql`, `java.xml`, `java.logging`. API стандартных платформенных модулей официально поддерживаются и предназначены для разработчиков.

Нестандартные платформенные модули являются частью JDK, но не включены в спецификацию платформы Java SE 9. Имена таких модулей начинаются словом `jdk`, например: `jdk.charsets`, `jdk.compiler`, `jdk.jlink`, `jdk.policytool`, `jdk.zipfs`. API, определенные в JDK-зависимых модулях, не предназначены для обычных разработчиков, а используются самим JDK и разработчиками библиотек, которые не могут добраться до нужной функциональности с помощью Java SE API. API, определенные в этих модулях, могут не поддерживаться или изменяться без предупреждения.

JavaFX не является частью спецификации платформы Java SE 9, но соответствующие модули устанавливаются вместе с JDK/JRE. Имена таких модулей начинаются словом `javafx`, например: `javafx.base`, `javafx.controls`, `javafx.fxml`, `javafx.graphics`, `javafx.web`.

Модуль `java.base`, являющийся частью платформы Java SE 9, можно назвать «первородным». Он не зависит ни от каких модулей. Система модулей знает только о `java.base`, а все остальные модули обнаруживает по определенным в модулях зависимостям. Модуль `java.base` экспортирует пакеты Java SE: `java.lang`, `java.io`, `java.math`, `java.text`, `java.time`, `java.util` и т. д.

Зависимости модулей

Вплоть до версии JDK 8 открытые типы, определенные в одном пакете, были доступны другим пакетам без ограничений, т. е. пакеты не контролировали доступность находящихся в них типов. Система модулей в JDK 9 обеспечивает более точный контроль доступности типов.

Доступность – это двустороннее соглашение между используемым и использующим модулем. Модуль может явно сделать свои открытые типы доступными другим модулям, а модуль, использующий типы, определенные в другом модуле, явно объявляет о зависимости от него. Все неэкспортируемые пакеты являются частной собственностью модуля и не могут использоваться вне него.

Объявление определенных в пакете API доступными другим модулям называется *экспортированием* пакета. Если модуль `policy` делает открытые типы в пакете `pkg1` открытыми другим модулям, то говорят, что `policy` экспортирует пакет `pkg1`. Если модуль `claim` объявляет зависимость от модуля `policy`, то говорят, что `claim` *читает* модуль `policy`. Если модуль `claim` читает модуль `policy`, значит, все открытые типы в пакетах, экспортируемых модулем `policy`, будут доступны в `claim`. Модуль может экспортировать пакет избирательно одному или нескольким именованным модулям. Такой экспорт называется *квалифицированным*, или *дружественным* (module-friendly). Открытые типы в квалифицированно экспортируемом пакете доступны только указанным модулям.

В контексте системы модулей три термина – *требует*, *читает* и *зависит от* – являются синонимами. В документации по Java предпочтение отдается термину *читает*.

На рис. 2.4 показана зависимость между модулями `policy` и `claim`. Модуль `policy` содержит два пакета, `pkg1` и `pkg2`, и экспортирует пакет `pkg1`, который обведен штриховой линией, чтобы отличить его от неэкспортируемого пакета `pkg2`. Модуль `claim` содержит два пакета, `pkg3` и `pkg4` и ни один из них не экспортирует. Он объявляет зависимость от модуля `policy`.

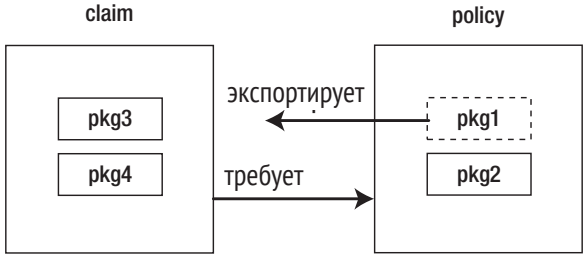

Рис. 2.4. Объявление зависимостей между модулями

В JDK 9 эти модули можно объявить так:

```
module policy {
    exports pkg1;
}

module claim {
    requires policy;
}
```

Совет. Синтаксис объявления зависимостей несимметричен: мы экспортируем *пакет*, а требуем *модуль*.

Предполагается, что имя модуля, от которого зависит ваш модуль, известно. Некоторые каркасы и инструменты Java опираются на рефлексию, чтобы получить доступ к коду неэкспортированных модулей на этапе выполнения. Они предоставляют такие важные средства, как внедрение зависимостей, сериализация, реализация Java Persistence API, автоматическое завершение кода и отладка. Примерами могут служить Spring, Hibernate и XStream. Такие каркасы и библиотеки

не знают о прикладных модулях, но тем не менее нуждаются в доступе к типам, определенным внутри модулей. Более того, им нужен доступ к закрытым членам модулей, что нарушает предположение о строгой инкапсуляции в JDK 9. Если модуль экспортирует пакет, то зависящие от него модули могут обращаться только к открытому API экспортированного пакета. Чтобы дать глубокий рефлексивный доступ (к открытому и закрытому API) ко всем пакетам внутри модуля на этапе выполнения, можно объявить *раскрытый* (open) модуль.

На рис. 2.5 показан раскрытый модуль `policy.model`. И сам модуль, и пакеты в нем обведены штриховыми линиями, чтобы показать, что модуль раскрытый и его пакеты доступны любому другому модулю. На этапе выполнения модуль `jdojo.jpa` обращается к типам в `policy.model` с помощью рефлексии. Модуль `jdojo.jpa` предоставляет реализацию JPA, а `policy.model` явно зависит от `jdojo.jpa`.

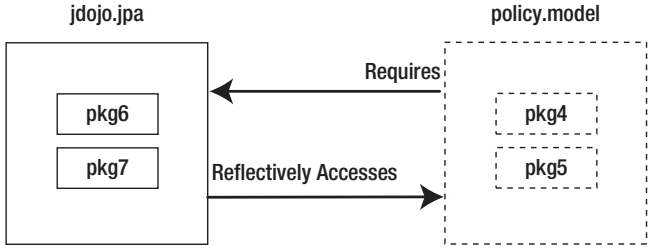

Рис. 2.5. Раскрытый модуль, дающий рефлексивный доступ ко всем своим членам

В JDK 9 эти модули объявляются так:

```
open module policy.model {
    requires jdojo.jpa;
}

module jdojo.jpa {
    // Здесь модуль экспортирует свои пакеты
}
```

Граф модулей

Система модулей знает только об одном модуле: `java.base`, который не зависит от других модулей. Все остальные неявно зависят от `java.base`.

Модульную структуру приложения можно визуализировать с помощью *графа модулей*, в котором каждый модуль представлен вершиной. Два модуля соединены ориентированным ребром, если начальный зависит от конечного. Граф модулей строится путем разрешения зависимостей набора *корневых модулей* от известных системе модулей, называемых *видимыми* (observable).

> **Совет.** Говорят, что модуль *разрешим*, если доступны все модули, от которых он зависит. Предположим, что модуль P зависит от модулей Q и R. Модуль P разрешим, если мы можем найти модули Q и R и рекурсивно разрешить их.

Граф модулей строится путем разрешения зависимостей на этапах компиляции, компоновки и выполнения. Процесс разрешения начинается с корневых

модулей и следует по зависимостям, пока не будет достигнут модуль `java.base`. Иногда модуль находится на пути к модулям, но система все равно сообщает, что модуль не найден. Так может случиться, если модуль не удалось разрешить и включить в граф модулей. Чтобы модуль был разрешим, он должен принадлежать цепочке зависимостей, начинающейся с корневого модуля. Набор корневых модулей по умолчанию выбирается исходя из того, как вызван компилятор или программа запуска приложений Java. Можно также добавлять модули в этот набор. Важно понимать, как в разных ситуациях выбираются корневые модули по умолчанию.

- Если код приложения скомпилирован с использованием пути к классам или программа запуска нашла главный класс приложения на пути к классам, то набор корневых модулей по умолчанию состоит из модуля `java.se` и всех системных модулей, имена которых не начинаются словом `java`, например: `jdk.*` и `javafx.*`. Если модуль `java.se` отсутствует, то этот набор состоит из всех системных модулей: как с именами вида `java.*`, так и не `java.*`.
- В противном случае набор корневых модулей по умолчанию зависит от этапа:
  - на этапе компиляции он состоит из всех компилируемых модулей;
  - на этапе компоновки он пуст;
  - на этапе выполнения он содержит модуль, в котором находится главный класс. Для задания модуля и его главного класса служит параметр `--module` (или `-m`) команды `java`.

Продолжая предыдущий пример, предположим, что в модуле `claim` главным классом является `pkg3.Main` и что оба модуля упакованы в модульные JAR-файлы в каталоге `C:\Java9Revealed\lib`. На рис. 2.6 показан граф модулей, который будет построен на этапе выполнения, когда приложение запускается командой:

```
C:\Java9Revealed>java -p lib -m claim/pkg3.Main
```

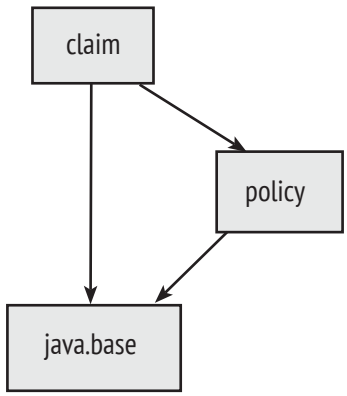

Рис. 2.6. Пример графа модулей

Модуль `claim` содержит главный класс приложения. Следовательно, это единственный корневой модуль в графе. Модуль `policy` разрешим, потому что от него зависит `claim`. Модуль `java.base` разрешим, потому что от него зависят все вообще модули.

Сложность графа модулей зависит от количества корневых модулей и зависимостей между ними. Предположим, что модуль `claim` зависит не только от `policy`, но и от платформенного модуля `java.sql`, т. е. объявление `claim` выглядит так:

```
module policy {
    requires policy;
    requires java.sql;
}
```

На рис. 2.7 показан граф модулей, который будет построен при запуске приложения с указанием главного класса `pkg3.Main` в модуле `claim`. Отметим, что в граф вошли также модули `java.xml` и `java.logging`, потом что от них зависит `java.sql`. Единственным корневым модулем в графе по-прежнему является `claim`.

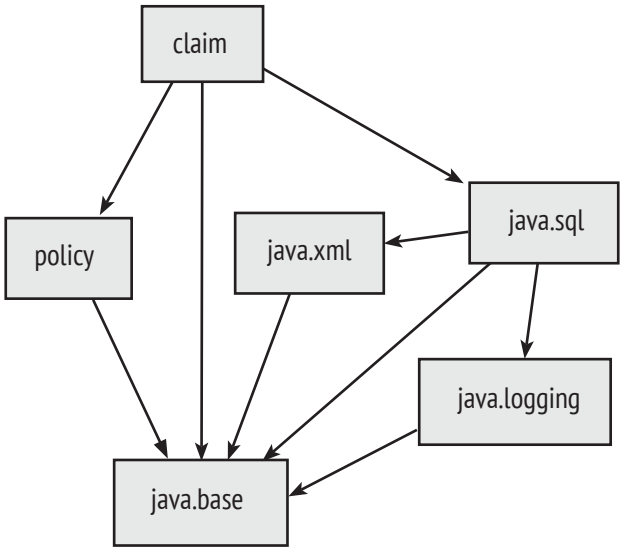

Рис. 2.7. Граф модулей с зависимостью от модуля `java.sql`

На рис. 2.8 показан гораздо более сложный граф для платформенного модуля `java.se`, объявление которого выглядит следующим образом:

```
module java.se {
    requires transitive java.sql;
    requires transitive java.rmi;
    requires transitive java.desktop;
    requires transitive java.security.jgss;
    requires transitive java.security.sasl;
    requires transitive java.management;
    requires transitive java.logging;
    requires transitive java.xml;
    requires transitive java.scripting;
    requires transitive java.compiler;
    requires transitive java.naming;
    requires transitive java.instrument;
```

```
    requires transitive java.xml.crypto;
    requires transitive java.prefs;
    requires transitive java.sql.rowset;
    requires java.base;
    requires transitive java.datatransfer;
}
```

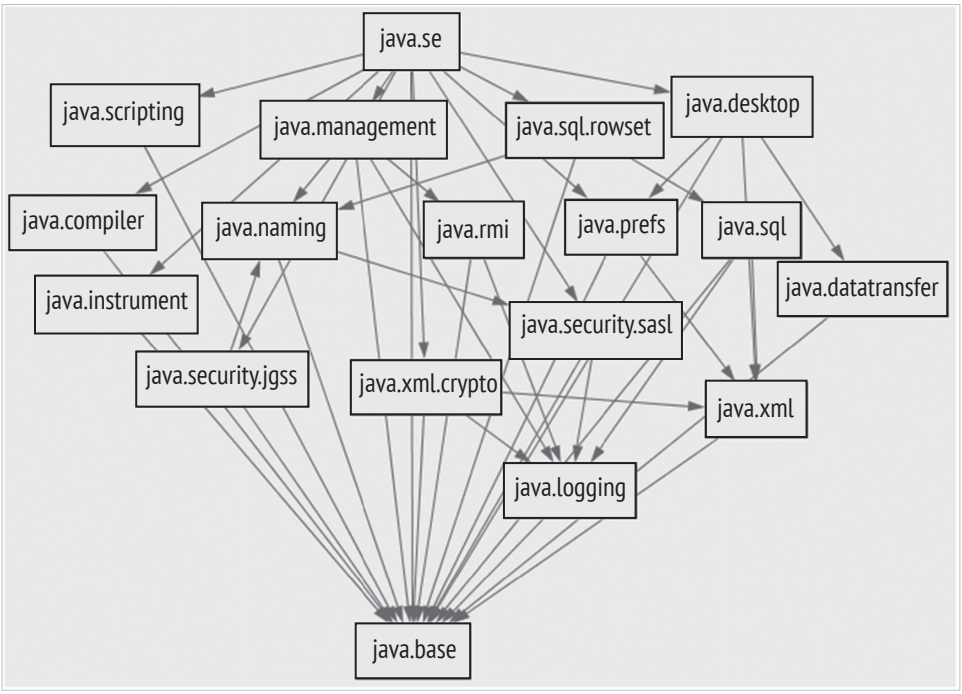

Рис. 2.8. Граф модулей с корневым модулем `java.se`

Иногда необходимо добавить модули в набор корневых модулей по умолчанию. Для этого служит параметр `--add-modules`, указываемый в командной строке компиляции, компоновки и выполнения:

```
--add-modules <module-list>
```

где `<module-list>` – список имен модулей через запятую.

Список имен модулей может включать специальные значения:

- `ALL-DEFAULT`
- `ALL-SYSTEM`
- `ALL-MODULE-PATH`

На этапе выполнения допустимо любое из этих значений, а на этапе компиляции только `ALL-MODULE-PATH`.

Если в списке модулей присутствует `ALL-DEFAULT`, то в набор корневых модулей включаются все определенные выше корневые модули, выбираемые по умолчанию. Это полезно, когда приложение является контейнером, содержащим другие приложения, зависящие от модулей, не нужных самому контейнерному приложе-

нию. Таким образом можно сделать доступными контейнеру все модули Java SE, чтобы любое внутреннее приложение могло ими воспользоваться.

Если в списке модулей присутствует `ALL-SYSTEM`, то в список корневых добавляются все системные модули. Это полезно для организации работы тестовых стендов.

Если в списке модулей присутствует `ALL-MODULE-PATH`, то в набор корневых добавляются все модули, найденные на путях к модулям. Это полезно для таких программ, как Maven, когда заведомо известно, что все модули на путях к модулям действительно необходимы приложению.

> **Совет.** Сообщение о том, что модуль не найден, может быть выдано, даже когда модуль встречается на пути к модулям. В таком случае нужно добавить не найденный модуль в набор корневых модулей по умолчанию с помощью параметра командной строки `--add-modules`.

JDK 9 поддерживает полезный нестандартный параметр командной строки `-Xdiag:resolver`, который включает диагностическую печать шагов разрешения модулей при построении графа. Ниже показана часть вывода команды, запускающей выполнение класса `pkg3.Main` в модуле `claim`. В конце печатается раздел `Result:`, содержащий список всех разрешенных модулей.

```
C:\Java9Revealed>java -Xdiag:resolver -p lib -m claim/pkg3.Main
```

```
[Resolver] Root module claim located
[Resolver]   (file:///C:/Java9Revealed/lib/claim.jar)
[Resolver] Module java.base located, required by claim
[Resolver]   (jrt:/java.base)
[Resolver] Module policy located, required by claim
[Resolver]   (file:///C:/Java9Revealed/lib/policy.jar)
...
[Resolver] Result:
[Resolver]   claim
[Resolver]   java.base
...
[Resolver]   policy
```

Модули-агрегаторы

Мы можем создать модуль, который сам не содержит никакого кода, а просто реэкспортирует содержимое других модулей. Такой модуль называется *агрегатором*. Предположим, что каждый из нескольких модулей зависит от пяти конкретных модулей. Мы можем создать агрегатор этих пяти модулей, и сделать остальные модули зависящими только от него.

Модули-агрегаторы служат только для удобства. В Java 9 есть несколько таких модулей, в т. ч. `java.se` и `java.se.ee`. Модуль `java.se` объединяет части Java SE, не пересекающиеся с Java EE, а модуль `java.se.ee` – все модули, входящие в состав Java SE, включая и те, что являются частью Java EE.

Объявление модулей

В этом разделе кратко описан синтаксис объявления модулей, подробности будут приведены в следующих главах.

Объявление модуля – новая синтаксическая конструкция в языке Java:

```
[open] module <module> {
    <module-statement>;
    <module-statement>;
    ...
}
```

Если присутствует необязательный модификатор `open`, то объявляется *раскрытый модуль* (open module). Раскрытый модуль экспортирует все свои пакеты для рефлексивного доступа. `<module>` – имя объявляемого модуля, `<module-statement>` – предложение модуля. В объявлении может быть нуль или более предложений модуля одного из пяти типов:

- предложение `exports`;
- предложение `opens`;
- предложение `requires`;
- предложение `uses`;
- предложение `provides`.

Предложения `exports` и `opens` служат для управления доступом к коду модуля. Предложение `requires` объявляет зависимость от другого модуля. Предложения `uses` и `provides` объявляют о потребляемых и предоставляемых службах соответственно. В примере ниже объявляется модуль `myModule`:

```
module myModule {
  // Экспортирует пакеты com.jdojo.util и com.jdojo.util.parser
  exports com.jdojo.util;
  exports com.jdojo.util.parser;

  // Читает модуль the java.sql
  requires java.sql;

  // Раскрывает пакет com.jdojo.legacy для рефлексивного доступа
  opens com.jdojo.legacy;

  // Пользуется интерфейсом службы java.sql.Driver
  uses java.sql.Driver;

  // Предоставляет класс com.jdojo.util.parser.FasterCsvParser,
  // реализуюший интерфейс службы com.jdojo.util.CsvParser
  provides com.jdojo.util.CsvParser
      with com.jdojo.util.parser.FasterCsvParser;
}
```

Раскрытый модуль (с модификатором `open`) предоставляет другим модулям рефлексивный доступ ко всем своим пакетам. Запрещается использовать предло-

жения opens в объявлении раскрытого модуля, поскольку все его пакеты и так уже раскрыты. Ниже объявлен раскрытый модуль myLegacyModule:

```
open module myLegacyModule {
    exports com.jdojo.legacy;
    requires java.sql;
}
```

Имена модулей

Имя модуля должно быть *квалифицированным* идентификатором Java, т. е. содержать один или несколько идентификаторов, разделенных точками, например: policy, com.jdojo.common, com.jdojo.util. Если какая-то часть имени модуля не является допустимым идентификатором Java, возникает ошибка на этапе компиляции. Так, com.jdojo.common.1.0 – недопустимое имя модуля, поскольку части 1 и 0 не являются допустимыми идентификаторами.

Для назначения модулям уникальных имен применяется то же соглашение об инвертированных доменных именах, что и для именования пакетов. При таком соглашении объявление простейшего модуля com.jdojo.common выглядит так:

```
module com.jdojo.common {
    // Ни одного предложения модуля
}
```

Имя модуля не маскирует одноименные переменные, типы и пакеты, т. е. допустимы, например, модуль и переменная с одинаковыми именами. Какая именно сущность именуется, определяется по контексту.

В JDK 9 слова open, module, requires, transitive, exports, opens, to, uses, provides и with зарезервированы. Вкратце они объясняются в этой главе, а более подробно в последующих. Но специальное значение они имеют только в определенных местах внутри объявления модуля, а в остальных частях программы могут употребляться в качестве идентификаторов. Например, следующее объявление модуля допустимо, хотя называть модуль таким образом вряд ли имеет смысл:

```
// Объявить модуль с именем module
module module {
    // Здесь могут быть предложения модуля
}
```

Здесь первое вхождение module – ключевое слово, а второе – имя модуля. В другом месте программы может быть объявлена переменная module:

```
String module = "myModule";
```

Контроль доступа к модулям

Предложение exports служит для экспорта указанного пакета всем модулям или только поименованным модулям из списка на этапах компиляции и выполнения. У него есть две формы:

```
exports <package>;
exports <package> to <module1>, <module2>...;
```

Например:

```
module M {
exports com.jdojo.util;
exports com.jdojo.policy
     to com.jdojo.claim, com.jdojo.billing;
}
```

Предложение `opens` предоставляет рефлексивный доступ к указанному пакету всем модулям или только поименованным модулям из списка на этапе выполнения. Эти модули могут воспользоваться рефлексией для доступа ко всем типам в указанном пакете и всем (открытым и закрытым) членам типов. У предложения `opens` есть две формы:

- `opens <package>;`
- `opens <package> to <module1>, <module2>...;`

Например:

```
module M {
    opens com.jdojo.claim.model;
    opens com.jdojo.policy.model to core.hibernate;
    opens com.jdojo.services to core.spring;
}
```

> **Совет.** Предложение `exports` дает доступ только к открытому API пакета на этапах компиляции и выполнения, тогда как предложение `opens` открывает доступ к открытым и закрытым членам всех типов в указанном пакете с помощью рефлексии, но только на этапе выполнения.

Чтобы предоставить другим модулям доступ к открытым типам на этапе компиляции и к закрытым членам типов на этапе выполнения с помощью рефлексии, модуль-владелец может одновременно экспортировать и раскрыть один и тот же пакет:

```
module N {
    exports com.jdojo.claim.model;
    opens com.jdojo.claim.model;
}
```

В литературе, посвященной модулям, можно встретить такие фразы:

- модуль `M` экспортирует пакет `P`;
- модуль `M` раскрывает пакет `Q`;
- модуль `M` содержит пакет `R`.

Первые две соответствуют предложениям `exports` и `opens`. А третья означает, что модуль содержит пакет `R`, который не экспортируется и не раскрывается. На ранних стадиях проектирования системы модулей третья ситуация описывалась фразой «модуль `M` *скрывает* пакет `R`».

Объявление зависимости

Предложение `requires` объявляет зависимость текущего модуля от другого и имеет следующие формы:

- `requires <module>;`
- `requires transitive <module>;`
- `requires static <module>;`
- `requires transitive static <module>.`

Модификатор `static` говорит, что зависимость обязательна на этапе компиляции, но факультативна на этапе выполнения, т. е. предложение `requires static N` в модуле с именем `M` означает, что `M` зависит от `N`, и `N` должен присутствовать во время компиляции, но может отсутствовать на этапе выполнения. При наличии модификатора `transitive` любой модуль, зависящий от данного, будет неявно зависеть и от модуля, указанного в предложении `requires`. Пусть имеется три модуля: `P`, `Q`, `R` и пусть в объявлении `Q` присутствует предложение `requires transitive R`. Если модуль `P` содержит предложение `requires Q`, то `P` неявно зависит от `R`.

Конфигурирование служб

Java располагает механизмом поставщика служб, позволяющим разорвать связь между поставщиком и потребителями службы. JDK 9 позволяет реализовывать службы с помощью предложений модуля `uses` и `provides`.

Предложение `uses` определяет имя интерфейса службы, которую данный модуль может обнаружить и загрузить с помощью класса `java.util.ServiceLoader`. Оно имеет вид:

```
uses <service-interface>;
```

Например:

```
module M {
    uses com.jdojo.prime.PrimeChecker;
}
```

Здесь `com.jdojo.PrimeChecker` – интерфейс службы, реализация которой будет предоставлена другими модулями.

Предложение `provides` определяет один или несколько классов реализации поставщика службы:

```
provides <service-interface>
    with <service-impl-class1>, <service-impl-class2>...;
```

Например:

```
module N {
    provides com.jdojo.prime.PrimeChecker
        with com.jdojo.prime.generic.GenericPrimeChecker;
}
```

Один и тот же модуль может как предоставлять реализации службы, так и обнаруживать и загружать службу. Разрешается также обнаруживать и загружать одну службу и предоставлять реализацию другой. Например:

```
module P {
  uses com.jdojo.CsvParser;

  provides com.jdojo.CsvParser
      with com.jdojo.CsvParserImpl;

  provides com.jdojo.prime.PrimeChecker
      with com.jdojo.prime.generic.FasterPrimeChecker;
}
```

Дескрипторы модулей

После того как мы узнали об объявлении модулей, возникает ряд вопросов.

- Где хранится исходный код объявления модуля? Если в файле, то как он называется?
- В каком месте должен находиться файл с исходным кодом объявления модуля?
- Как компилируется объявление модуля?

Я отвечу на эти вопросы, прежде чем демонстрировать первую модульную программу в действии.

Компиляция объявлений модулей

Объявление модуля хранится в файле с именем `module-info.java`, который должен находиться в корне иерархии файлов, относящихся к данному модулю. Результатом компиляции объявления является файл с именем `module-info.class`, который называется *дескриптором модуля* и помещается в корень иерархии откомпилированных файлов модуля. Если откомпилированный код модуля записывается в JAR-файл, то файл `module-info.class` будет находиться в корне этого JAR-файла.

В объявлении модуля нет выполняемого кода. По существу это просто конфигурация модуля. Тогда почему оно хранится в файле класса, а не в текстовом файле формата XML или JSON? Только потому, что формат класса расширяемый и имеет точное определение. Дескриптор модуля содержит откомпилированную форму исходного объявления. Другие инструменты, например `jar`, могут включать в атрибуты файла класса дополнительную информацию уже после первоначальной компиляции. Кроме того, благодаря формату класса разработчик может включать в объявление модуля директивы импорта и аннотации.

Версия модуля

В первоначальном прототипе системы модулей в объявление модуля включался еще номер версии. Но поскольку это усложняло реализацию системы, впоследствии от версии отказались.

Для добавления версии модуля используется расширяемость формата дескриптора модуля (файла класса). При упаковке откомпилированного кода модуля в JAR-файл программой `jar` можно задать номер версии, который включается в файл `module-info.class`. Как это сделать, я объясню в главе 3.

Структура исходных файлов модуля

Рассмотрим организацию исходного и откомпилированного кода на примере модуля `com.jdojo.contact`. Он содержит два пакета для работы с контактной информацией, в частности, адресами и телефонами:

- `com.jdojo.contact.info`
- `com.jdojo.contact.validator`

Пакет `com.jdojo.contact.info` содержит два класса: `Address` и `Phone`, а пакет `com.jdojo.contact.validator` – интерфейс `Validator` и два класса: `AddressValidator` и `PhoneValidator`. На рис. 2.9 показано содержимое модуля `com.jdojo.contact module`.

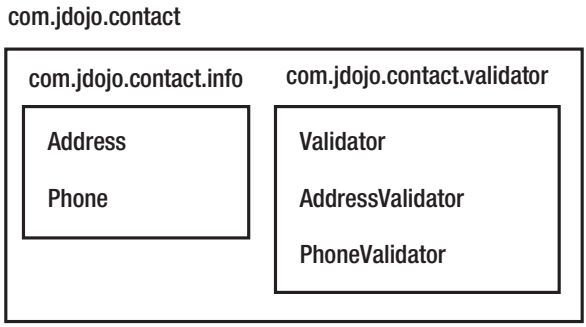

Рис. 2.9. Содержимое модуля `com.jdojo.contact module`

В Java 9 у компилятора `javac` появилось несколько новых параметров. Разрешается компилировать как по одному модулю за раз, так и сразу несколько модулей. Если вы хотите откомпилировать несколько модулей, то *обязаны* поместить исходный код каждого модуля в отдельный каталог с таким же именем, как у модуля. Такого соглашения об именовании каталогов можно придерживаться и при компиляции единственного модуля.

Пусть требуется откомпилировать модуль `com.jdojo.contact`. Мы можем поместить его исходный код в каталог `C:\j9r\src`, который будет содержать следующие файлы:

```
module-info.java
com\jdojo\contact\info\Address.java
com\jdojo\contact\info\Phone.java
com\jdojo\contact\validator\Validator.java
com\jdojo\contact\validator\AddressValidator.java
com\jdojo\contact\validator\PhoneValidator.java
```

Отметим, что для хранения исходных файлов интерфейсов и классов необходимо придерживаться иерархии пакетов, как повелось со времен Java 1.0.

Чтобы откомпилировать сразу несколько модулей, мы должны назвать каталог с исходным кодом `com.jdojo.contact`, т. е. так же, как сам модуль. В таком случае в каталоге `C:\j9r\src` будет создана такая иерархия:

```
com.jdojo.contact\module-info.java
com.jdojo.contact\com\jdojo\contact\info\Address.java
com.jdojo.contact\com\jdojo\contact\info\Phone.java
```

```
com.jdojo.contact\com\jdojo\contact\validator\Validator.java
com.jdojo.contact\com\jdojo\contact\validator\AddressValidator.java
com.jdojo.contact\com\jdojo\contact\validator\PhoneValidator.java
```

Каталог с откомпилированным кодом будет устроен точно так же.

Упаковка модулей

Результат компиляции модуля можно хранить в:

- каталоге;
- модульном JAR-файле;
- JMOD-файле, это новый формат упаковки модулей, появившийся в JDK 9.

Хранение модуля в каталоге

Если откомпилированный код модуля хранится в каталоге, то в корне этого каталога находится дескриптор модуля (файл `module-info.class`), а подкаталоги отражают иерархию пакетов. Продолжая пример из предыдущего раздела, предположим, что откомпилированный код модуля `com.jdojo.contact` хранится в каталоге `C:\j9r\mods\com.jdojo.contact`. Тогда его содержимое выглядит так:

```
module-info.class
com\jdojo\contact\info\Address.class
com\jdojo\contact\info\Phone.class
com\jdojo\contact\validator\Validator.class
com\jdojo\contact\validator\AddressValidator.class
com\jdojo\contact\validator\PhoneValidator.class
```

Хранение модуля в модульном JAR-файле

В состав JDK входит программа `jar` для упаковки Java-кода в файл формата JAR (**J**ava **A**rchive). Формат JAR основан на формате ZIP. В JDK 9 программа `jar` дополнена и теперь может упаковывать код модуля в JAR-файл. JAR-файл, содержащий откомпилированный код модуля, называется *модульным JAR-файлом*. В корне такого файла находится файл `module-info.class`.

Всюду, где раньше использовались JAR-файлы, теперь можно использовать модульные JAR-файлы. В частности, модульный JAR-файл можно поместить на путь к классам, и в таком случае файл `module-info.class` в нем игнорируется, потому что `module-info` – недопустимое имя класса в Java.

Для упаковки в модульный JAR-файл в программу `jar` были добавлены параметры, позволяющие включать в дескриптор модуля различную информацию, например, версию модуля и имя главного класса.

> **Совет.** Модульный JAR-файл – это самый обычный JAR-файл, только в его корне находится дескриптор модуля.

Типичное нетривиальное Java-приложение состоит из нескольких модулей. Но модульный JAR-файл может содержать откомпилированный код только *одного*

модуля. Чтобы распространять приложение в виде одного артефакта, должна быть возможность упаковать все модули в один JAR-файл. На момент написания книги этот вопрос был открыт, и его обсуждение можно было найти по адресу http://openjdk.java.net/projects/jigsaw/spec/issues/#MultiModuleExecutableJARs.

Продолжая предыдущий пример, приведем содержимое модульного JAR-файла для модуля `com.jdojo.contact`. Отметим, что JAR-файл всегда содержит файл `MANIFEST.MF` в каталоге `META-INF`.

```
module-info.class
com/jdojo/contact/info/Address.class
com/jdojo/contact/info/Phone.class
com/jdojo/contact/validator/Validator.class
com/jdojo/contact/validator/AddressValidator.class
com/jdojo/contact/validator/PhoneValidator.class
META-INF/MANIFEST.MF
```

Хранение модуля в JMOD-файле

В JDK 9 определен новый формат, *JMOD*, для упаковки модулей. У JMOD-файла расширение `.jmod`. Модули самого JDK упакованы в формате JMOD и находятся в каталоге `JDK_HOME\jmods`; например, файл `java.base.jmod` содержит весь модуль `java.base`. JMOD-файлы поддерживаются только на этапах компиляции и компоновки, но не на этапе выполнения. Рассмотрение деталей формата JMOD мы отложим до главы 6.

Путь к модулям

Механизм пути к классам для поиска типов существует с самой первой версии JDK. Путь к классам включает каталоги, JAR-файлы и ZIP-файлы. В процессе поиска типов на различных этапах (компиляции, выполнения, в инструментах) Java просматривает элементы пути к классам.

В Java 9 типы являются частью модулей. Java должен искать на различных этапах модули, а не типы, как раньше. Поэтому был разработан новый механизм поиска – *путь к модулям*.

Путь к модулям – это последовательность имен путей, содержащих модули, где именем пути может быть модульный JAR-файл, JMOD-файл или каталог. Имена путей разделяются платформенно-зависимым разделителем: двоеточием (:) в UNIX или точкой с запятой (;) в Windows.

Когда имя пути – модульный JAR-файл или JMOD-файл, все просто. В этом случае, если дескриптор модуля содержит определение искомого модуля, то модуль считается найденным. Если же имя пути – каталог, то возможно два случая.

- Если в корне каталога существует файл класса, то каталог считается определением модуля, а этот файл класса интерпретируется как дескриптор модуля. Все остальные файлы и подкаталоги считаются частями одного модуля. Если в корне есть несколько файлов класса, то дескриптором модуля считается первый найденный. После нескольких экспериментов в сборке 126 JDK 9, похоже, принято решение выбирать первый файл класса в алфавит-

ном порядке. Такой способ хранения откомпилированного модуля гарантированно создаст вам головную боль. Поэтому старайтесь не добавлять в путь к модулям каталог, если в корне этого каталога есть несколько файлов классов.

- Если в корне каталога не существует файла класса, то содержимое каталога интерпретируется иначе. Каждый модульный JAR-файл или JMOD-файл в каталоге считается определением модуля. Подкаталоги, содержащие файл `module-info.class` в корне, считаются определением модуля в формате каталога. Если в корне подкаталога нет файла `module-info.class`, то считается, что подкаталог не содержит определения модуля. Отметим, что если подкаталог содержит определение модуля, то имена подкаталога и модуля не обязаны совпадать. Имя модуля читается из файла `module-info.class`.

Ниже приведены три допустимых пути к модулям в Windows:

- `C:\mods`
- `C:\mods\com.jdojo.contact.jar;C:\mods\com.jdojo.person.jar`
- `C:\lib;C:\mods\com.jdojo.contact.jar;C:\mods\com.jdojo.person.jar`

Первый путь к модулям содержит только путь к каталогу `C:\mods`, второй – пути к двум модульным JAR-файлам – `com.jdojo.contact.jar` и `com.jdojo.person.jar`, а третий – путь к каталогу `C:\lib` и пути к двум модульным JAR-файлам – `com.jdojo.contact.jar` и `com.jdojo.person.jar`. Эквиваленты этих путей на платформе UNIX таковы:

- `/usr/ksharan/mods`
- `/usr/ksharan/mods/com.jdojo.contact.jar:/usr/ksharan/com.jdojo.person.jar`
- `/usr/ksharan/lib:/usr/ksharan/mods/com.jdojo.contact.jar:`
  `/usr/ksharan/mods/com.jdojo.person.jar`

Чтобы избежать проблем с путем к модулям, лучше всего не использовать в качестве определения модуля развернутое дерево. Включите в путь к модулям два каталога: один пусть содержит все модульные JAR-файлы приложения, а другой – модульные JAR-файлы всех внешних библиотек. Например, в Windows можно в качестве такого пути к модулям задать `C:\applib;C:\extlib`.

В JDK 9 все инструменты обновлены с целью поиска модулей на пути к модулям. Для задания пути к модулям появились новые параметры. Раньше параметры записывались, как принято в UNIX – с одним дефисом (`-`), например: `-cp` или `-classpath`. Но после того как в JDK 9 появилось много новых параметров, у разработчиков кончились осмысленные короткие имена, поэтому пришлось перейти на принятый в продуктах GNU стиль: параметру предшествуют два дефиса, а слова разделяются дефисами. Ниже приведено несколько примеров:

- `--class-path`
- `--module-path`
- `--module-version`
- `--main-class`
- `--print-module-descriptor`

Совет. Для получения списка всех стандартных параметров, поддерживаемых программой, запустите ее с параметром `--help` или `-h`, для получения нестандартных параметров – с параметром `-X`.

Большинство инструментов в JDK 9, включая `javac`, `java` и `jar`, поддерживают два параметра для задания пути к модулям: `-p` и `--module-path`. Все старые параметры в стиле UNIX по-прежнему поддерживаются ради обратной совместимости. Ниже показано, как эти параметры используются в случае программы `java`:

```
// Задание параметра в стиле GNU
C:\>java --module-path C:\applib;C:\lib other-args-go-here

// Задание параметра в стиле UNIX
C:\>java -p C:\applib;C:\extlib other-args-go-here
```

В этой книге я всюду буду использовать параметр `--module-path`. Значение параметра в стиле GNU можно задать двумя способами:

- `--<name> <value>`
- `--<name>=<value>`

Поэтому предыдущую команду можно записать и так:

```
// Задание параметра в стиле GNU
C:\>java --module-path=C:\applib;C:\lib other-args-go-here
```

Если имя и значение разделяются пробелами, то нужен хотя бы один пробел. Если же используется разделитель `=`, то вокруг него не должно быть пробелов. Запись `--module-path=C:\applib` правильна, а запись `--module-path =C:\applib` неправильна, потому что `=C:\applib` будет интерпретироваться как путь к модулям, а такой путь недопустим.

Видимые модули

В процессе поиска модулей система просматривает различные типы путей к модулям. Модули, найденные на пути к модулям, а также системные модули называются *видимыми*. Можно считать, что видимые модули – это все модули, доступные инструменту или системе модулей на определенном этапе: компиляции, компоновки или выполнения. В JDK 9 команда `java` получила новый параметр `--list-modules`, задаваемый в одной из двух форм:

- `--list-modules`
- `--list-modules <module1>,<module2>...`

В первом случае печатается список всех видимых модулей, а во втором – описания указанных модулей. Показанная ниже команда печатает список видимых модулей, в который входят только системные модули:

```
c:\Java9Revealed> java --list-modules
```

```
java.base@9-ea
java.se.ee@9-ea
java.sql@9-ea
javafx.base@9-ea
javafx.controls@9-ea
jdk.jshell@9-ea
```

```
jdk.unsupported@9-ea
...
```

Список приведен не полностью. Каждый элемент списка состоит из двух частей: имя и версия модуля, разделенные знаком `@`. В строке версии число 9 соответствует JDK 9, а `ea` означает *early access* (ознакомительная версия). Выполнив эту команду на своей машине, вы можете получить другие версии.

В моем каталоге `C:\Java9Revealed\lib` есть три модульных JAR-файла. Если я укажу `C:\Java9Revealed\lib` в качестве пути к модулям в команде `java`, то эти модули будут включены в список видимых. Ниже показано, как изменился список видимых модулей при задании пути к модулям. Здесь `lib` – относительный путь, а `C:\Java9Revealed` – текущий каталог.

```
C:\Java9Revealed>java --module-path lib --list-modules
```

```
claim (file:///C:/Java9Revealed/lib/claim.jar)
policy (file:///C:/Java9Revealed/lib/policy.jar)
java.base@9-ea
java.xml@9-ea
javafx.base@9-ea
jdk.unsupported@9-ea
jdk.zipfs@9-ea
...
```

Отметим, что в случае модулей приложения печатается местонахождение модуля. Это полезно для отладки, когда получается неожиданный результат и вы не знаете, какие модули используются и где они находятся.

В следующей команде мы задали модуль `claim` в качестве значения параметра `--list-modules`, чтобы напечатать описание этого модуля:

```
C:\Java9Revealed>java --module-path lib --list-modules claim
```

```
module claim (file:///C:/Java9Revealed/lib/claim.jar)
  exports com.jdojo.claim
  requires java.sql (@9-ea)
  requires mandated java.base (@9-ea)
  contains pkg3
```

Первая строка распечатки содержит имя модуля и местоположение модульного JAR-файла. Из второй строки мы видим, что модуль экспортирует пакет `com.jdojo.claim`. Третья строка показывает, что модулю требуется модуль `java.sql`, а четвертая – что модуль зависит от обязательного модуля `java.base`. Напомним, что от `java.base` зависит любой модуль, кроме него самого. Поэтому строка `requires mandated java.base` присутствует в описании любого модуля, кроме `java.base`. Пятая строка говорит, что модуль содержит пакет `pkg3`, который не экспортируется и не раскрывается.

Можно напечатать и описание системных модулей, например `java.base` и `java.sql`, Ниже показано описание модуля `java.sql`:

```
C:\Java9Revealed>java --list-modules java.sql
```

```
module java.sql@9-ea
exports java.sql
exports javax.sql
exports javax.transaction.xa
requires transitive java.xml
requires mandated java.base
requires transitive java.logging
uses java.sql.Driver
```

Резюме

В Java пакеты всегда использовались как контейнеры для типов. Приложение состояло из нескольких JAR-файлов, находящихся на пути к классам. Но пакеты не определяли границы доступности. Доступность типа определялась в нем самом с помощью модификаторов видимости. Если пакет содержал внутренние реализации, то не было никакого способа запретить доступ к ним со стороны других частей программы. Механизм путей к классам осуществлял линейный поиск типа в момент его использования. Это могло приводить к ошибкам на этапе выполнения, если тип отсутствовал в развернутых JAR-файлах, а такое иногда случалось спустя много времени после первоначального развертывания приложения. Такого рода проблемы можно отнести к двум категориям: инкапсуляция и конфигурация.

В JDK 9 введена система модулей, позволяющая лучше организовать Java-программы. Она преследует две цели: *строгую инкапсуляцию* и *надежную конфигурацию*. Приложение состоит из модулей – именованных совокупностей кода и данных. С помощью своего объявления модуль контролирует, какие его части доступны другим модулям. Модуль, желающий получить доступ к частям другого модуля, должен объявить зависимость от последнего. Оба аспекта – контроль доступа и объявление зависимостей – лежат в основе строгой инкапсуляции. Зависимости модулей разрешаются на этапе инициализации приложения. В JDK 9, если один модуль зависит от другого, а этот другой модуль в момент запуска приложения отсутствует, то ошибка будет выдана сразу, а не в какой-то момент после начала работы приложения. Это основа надежной конфигурации.

Для описания модуля служит его объявление, которое обычно хранится в файле `module-info.java`. Результатом компиляции модуля является файл `module-info.class`. Откомпилированное объявление модуля называется *дескриптором модуля*. В объявлении модуля версия не задается, но инструменты типа `jar` могут добавить сведения о версии в дескриптор модуля на этапе построения JAR-файла.

Для объявления модуля служит ключевое слово `module`, за которым следует имя модуля. В объявлении могут встречаться предложения модуля пяти видов: `exports`, `opens`, `requires`, `uses` и `provides`. Предложение `exports` служит для экспорта указанного пакета на этапах компиляции и выполнения всем модулям или только модулям с

указанными именами. Предложение `opens` предоставляет рефлексивный доступ к указанному пакету на этапе выполнения всем модулям или только модулям с указанными именами. Получившие доступ модули могут обращаться ко всем типам в указанном пакете и всем членам (открытым и закрытым) этих типов с помощью рефлексии. Предложения `uses` и `provides` используются, чтобы сообщить, реализации каких служб модуль обнаруживает и предоставляет соответственно.

В JDK 9 `open`, `module`, `requires`, `transitive`, `exports`, `opens`, `to`, `uses`, `provides` и `with` являются контекстно-зависимыми ключевыми словами. Они приобретают специальное значение, только если встречаются в определенных местах объявления модуля.

Исходный и откомпилированный код модуля могут быть организованы в виде каталога, JAR-файла или JMOD-файла. В каталоге и JAR-файле файл `module-info.class` находится в корне.

По аналогии с путем к классам в JDK 9 введено понятие пути к модулям. Однако используются они по-разному. Путь к классу используется для поиска определений типов, а путь к модулям – для поиска модулей, а не конкретных типов внутри модуля. Инструменты Java, в частности `java` и `javac`, модифицированы и теперь знают как о пути к классам, так и о пути к модулям. Для задания пути к модулям служат параметры командной строки `--module-path` или `-p`.

В JDK 9 для задания параметров командной строки используется подход GNU. Параметр начинается двумя дефисами и слова также разделяются дефисами, например: `--module-path`, `--class-path`, `--list-modules`. Если с параметром связано значение, то оно указывается после имени через пробел или знак `=`. Следующие два варианта записи параметра эквивалентны:

- `--module-path C:\lib`
- `--module-path=C:\lib`

Модули, доступные системе модулей на каком-то этапе (компиляции, выполнения, в инструментах и т. д.), называются видимыми. Если задать параметр `--list-modules`, то команда `java` напечатает список модулей, видимых на этапе выполнения. Этот же параметр позволяет напечатать описание модуля.

Глава 3

Создаем свой первый модуль

Краткое содержание главы:

- написание модульных программ на Java;
- компиляция модульных программ;
- упаковка артефактов модуля в модульный JAR-файл;
- запуска модульной программы.

В этой главе я объясню, как работать с модулем – от написания исходного кода до компиляции, упаковки и выполнения. Глава состоит из двух частей. В первой показаны все шаги написания и запуска модульной программы из командной строки, а во второй – как то же самое делается в NetBeans IDE.

На момент написания книги NetBeans IDE еще находилась в процессе разработки и не поддерживала все возможности JDK 9. В частности, для каждого модуля в NetBeans нужно было создавать отдельный проект. В окончательной версии будет разрешено иметь несколько модулей в одном проекте. При рассмотрении инструментов командной строки будет охвачено больше функций JDK 9, чем при описании работы с NetBeans IDE.

В качестве примера в этой главе взята очень простая программа, которая печатает сообщение и имя модуля, которому принадлежит главный класс.

Работа с инструментами командной строки

Ниже описаны шаги создания и запуска модуля из командной строки.

Подготовка каталогов

Для написания, компиляции, упаковки и выполнения исходного кода нам понадобятся такие каталоги:

- `C:\Java9Revealed`
- `C:\Java9Revealed\lib`
- `C:\Java9Revealed\mods`
- `C:\Java9Revealed\src`
- `C:\Java9Revealed\src\com.jdojo.intro`

Так структура каталогов выглядит в Windows. В других ОС можно построить аналогичную иерархию. Главное, чтобы наверху был каталог `Java9Revealed`, а в нем три подкаталога: `lib`, `mods` и `src`.

В каталоге `src` хранится исходный код, для которого создан подкаталог `com.jdojo.intro`. Это сделано, потому что я намереваюсь создать в этом подкаталоге модуль `com.jdojo.intro`. Действительно ли в этом случае необходимо называть каталог `com.jdojo.intro`? Нет. Его можно было бы назвать как угодно или вообще поместить исходный код прямо в каталог `src`. Но принято размещать код модуля в подкаталоге и называть этот подкаталог так же, как сам модуль. У компилятора Java есть параметры, позволяющие компилировать сразу несколько модулей, но только если соблюдается это соглашение.

Каталог `mods` будет использоваться для хранения неупакованного откомпилированного кода. При желании программу можно выполнить прямо из этого каталога.

Откомпилированный код упаковывается в модульный JAR-файл, который помещается в каталог `lib`. Этот модульный JAR-файл можно использовать для запуска программы или передать его другим разработчикам.

Далее в этом разделе указывается относительный путь к каталогу, например `src` или `src\com.jdojo.intro`. Пути откладываются от каталога `C:\Java9Revealed`, т. е. `src` означает `C:\Java9Revealed\src`. Если вы работаете не в Windows или организовали иерархию каталогов по-другому, внесите соответствующие коррективы.

Написание исходного кода

Для написания кода можно использовать любой редактор, например Блокнот в Windows. Для начала создадим модуль с именем `com.jdojo.intro`. В листинге 3.1 показано его объявление.

Листинг 3.1. Объявление модуля com.jdojo.intro

```
// module-info.java
module com.jdojo.intro {
    // Предложений модуля нет
}
```

В объявлении этого модуля нет ни одного предложения. Сохраните его в файле `module-info.java` в каталоге `src\com.jdojo.intro`.

Далее мы создадим класс `Welcome` в пакете `com.jdojo.intro`. Обратите внимание, что пакет назван так же, как модуль. Обязательно ли это? Нет. Имя пакета может быть любым. В классе будет метод с сигнатурой `public status void main(String[])`, который станет точкой входа в приложение. В этом методе мы напечатаем сообщение. Мы хотим вывести имя модуля, которому принадлежит класс `Welcome`. В JDK 9 в пакет `java.lang` добавлен класс `Module`. Экземпляр этого класса представляет модуль. Любой тип Java в JDK 9 является частью модуля, это относится даже к примитивным типам `int`, `long`, `char` и т. д. Все примитивные типы входят в модуль `java.base`. В классе `Class` в JDK 9 появился новый метод `getModule()`, возвращающий ссылку на модуль, которому принадлежит класс. В следующем фрагменте печатается имя модуля, содержащего класс `Welcome`.

```
Class<Welcome> cls = Welcome.class;
Module mod = cls.getModule();
String moduleName = mod.getName();
System.out.format("Имя модуля: %s%n", moduleName);
```

Эти четыре предложения можно заменить одним:

```
System.out.format("Имя модуля: %s%n",
                  Welcome.class.getModule().getName());
```

Совет. Все примитивные типы данных принадлежат модулю `java.base`. Чтобы получить ссылку на модуль примитивного типа, можно написать `int.class.getModule()`.

В листинге 3.2 приведен полный код класса `Welcome`. Сохраните его в файле `Welcome.java` в каталоге `src\com\jdojo\intro`.

Листинг 3.2. Исходный код класса Welcome

```
// Welcome.java
package com.jdojo.intro;

public class Welcome {
    public static void main(String[] args) {
        System.out.println("Добро пожаловать в систему модулей.");

        // Напечатать имя модуля, содержащего класс Welcome
        Class<Welcome> cls = Welcome.class;
        Module mod = cls.getModule();
        String moduleName = mod.getName();
        System.out.format("Имя модуля: %s%n", moduleName);
    }
}
```

Компиляция исходного кода

Чтобы откомпилировать исходный код и сохранить результат в каталоге `C:\java9Revealed\mods`, мы воспользуемся командой `javac`, которая находится в каталоге `JDK_HOME\bin`. Для компиляции выполните следующую команду (ее надо вводить на одной строке, а не на трех):

```
C:\Java9Revealed>javac -d mods --module-source-path src
 src\com.jdojo.intro\module-info.java
 src\com.jdojo.intro\com\jdojo\intro\Welcome.java
```

Отметим, что в момент выполнения команды текущим каталогом является `C:\Java9Revealed`. Параметр `-d mods` означает, что откомпилированные файлы классов должны записываться в каталог `mods`. Поскольку команда запускается из каталога `C:\java9revealed directory`, то `mods` соответствует каталогу `C:\Java9Revealed\mods`. Если хотите, можете задать полное имя: `-d C:\Java9Revealed\mods`.

Параметр `--module-source-path src` говорит, что в подкаталогах каталога `src` может находиться код нескольких модулей, причем имя подкаталога совпадает с именем хранящегося в нем модуля. Отсюда вытекает несколько следствий.

- В каталоге `src` должно быть организовано несколько подкаталогов, по одному для каждого модуля, и имя подкаталога должно быть таким же, как имя модуля в нем.
- Компилятор Java создает параллельную структуру в каталоге `mods`, где будет хранить откомпилированные классы.
- Если не задавать этот параметр, то сгенерированные файлы классов записываются непосредственно в каталог `mods`.

Последние два аргумента команды `javac` – исходные файлы: объявление модуля и код класса `Welcome`. Если `javac` отработает успешно, то в каталоге `C:\Java9Revealed\mods\com.jdojo.intro` окажутся два файла:

- `module-info.class`
- `com\jdojo\intro\Welcome.class`

Следующая команда компилирует модуль `com.jdojo.intro` так, как это было принято до JDK 9. В ней задан только параметр `-d`, определяющий, куда поместить откомпилированные файлы классов.

```
C:\ Java9Revealed>javac -d mods\com.jdojo.intro
 src\com.jdojo.intro\module-info.java
 src\com.jdojo.intro\com\jdojo\intro\Welcome.java
```

Результат будет таким же, как после выполнения предыдущей команды. Но такой подход не годится, если вы хотите одной командой откомпилировать сразу несколько модулей и поместить результаты в разные каталоги.

Параметр `--module-version` компилятора `javac` позволяет задать версию компилируемого модуля. Номер версии хранится в файле `module-info.class`. Следующая команда создает такой же набор откомпилированных файлов, как предыдущие, но еще записывает в `module-info.class` версию 1.0:

```
C:\Java9Revealed>javac -d mods\com.jdojo.intro --module-version 1.0
 src\com.jdojo.intro\module-info.java
 src\com.jdojo.intro\com\jdojo\intro\Welcome.java
```

Как убедиться, что `javac` сохранила версию модуля в файле `module-info.class`? Можно воспользоваться командой `javap` для дизассемблирования файла класса. Если указать путь к файлу `module-info.class`, то `javap` напечатает определение модуля, содержащее и версию, если она задана. Имя модуля в таком случае имеет вид `moduleName@moduleVersion`. Выполните следующую команду:

```
C:\Java9Revealed>javap mods\com.jdojo.intro\module-info.class
```

```
Compiled from "module-info.java"
module com.jdojo.intro@1.0 {
  requires java.base;
}
```

Команда `jar` в JDK 9 тоже модифицирована. Она позволяет задать версию модуля при создании модульного JAR-файла. В следующем разделе я покажу, как это делается.

Чтобы откомпилировать несколько модулей, нужно указать в команде `javac` исходные файлы каждого. Это слишком долго, поэтому я покажу, как можно просто откомпилировать все модули за один раз. В Windows для этого достаточно такой однострочной команды:

```
C:\Java9Revealed>FOR /F "tokens=1 delims=" %A in ('dir src\*.java /S /B') do javac
 -d mods --module-source-path src %A
```

Эта команда перебирает все файлы с расширением `.java` в каталоге `src` и компилирует их по одному. Я не смог найти в Windows команду, которая передает компилятору `javac` все исходные файлы сразу. Если ваша структура каталогов отличается от моей, измените имена каталогов в этой команде. Если вы хотите сохранить эту команду в пакетном файле, то замените `%A` на `%%A`.

В UNIX эквивалентная команда выглядит так:

```
$ javac -d mods --module-source-path src $(find src -name "*.java")
```

Упаковка кода модуля

Теперь упакуем откомпилированный код в модульный JAR-файл. Для этого понадобится команда `jar`, находящаяся в каталоге `JDK_HOME\bin`. Следующую команду нужно вводить на одной строке, а не на нескольких. Я разбил ее на несколько строк для большей понятности. Обратите внимание на последнюю точку, которая обозначает текущий каталог.

```
C:\Java9Revealed>jar --create
 --file lib/com.jdojo.intro-1.0.jar
 --main-class com.jdojo.intro.Welcome
 --module-version 1.0
 -C mods/com.jdojo.intro .
```

Опишем параметры этой команды.

- Параметр `--create` говорит, что мы создаем новый модульный JAR-файл.
- Параметр `-file` задает имя и местоположение модульного JAR-файла. Мы сохраняем JAR-файл в каталоге `lib` и называем его `com.jdojo.intro-1.0.jar`. Я включил в имя файла номер версии 1.0.
- Параметр `--main-class` задает полное имя класса, содержащего точку входа в приложение – метод `public static void main(String[])`. Если этот параметр задан, то `jar` добавит в файл `module-info.class` атрибут, значением которого будет указанное имя класса, и одновременно добавит атрибут `Main-Class` в файл `MANIFEST.MF`.
- Параметр `--module-version` задает версию модуля, 1.0. Команда `jar` записывает эту информацию в атрибут в файле `module-info.class`. Заметим, что задание версии 1.0 никак не отражается на имени модульного JAR-файла. Чтобы было понятно, что именно содержит файл, мы явно добавили в его имя строку 1.0, а сам номер версии задается с помощью описываемого параметра.

- Параметр `-C` определяет, какой каталог считать текущим при выполнении `jar` – в данном случае `mods\com.jdojo.intro`. В результате `jar` будет читать все файлы, подлежащие включению в модульный JAR-файл, из этого каталога.
- Последняя часть команды – точка (.), означающая, что `jar` должна включить все файлы и каталоги, находящиеся в текущем каталоге `mods\com.jdojo.intro`. Отметим, что этот параметр работает в тандеме с параметром `-C`. Если параметр `-C` не задан, то точка интерпретируется как каталог `C:\Java9Revealed`, из которого запускалась команда.

Если команда отработает успешно, то будет создан файл

```
C:\Java9Revealed\lib\com.jdojo.intro-1.0.jar
```

Чтобы проверить, содержит ли модульный JAR-файл определение модуля `com.jdojo.intro`, выполните такую команду:

```
C:\Java9Revealed>java --module-path lib --list-modules com.jdojo.intro
```

Здесь мы задали в качестве пути к модулям каталог `lib`, т. е. именно в нем будут искаться модули приложения. В качестве значения параметра `-listmodules` указано имя модуля `com.jdojo.intro`, в результате будет напечатано его описание и местоположение. Если будет напечатан результат, похожий на показанный ниже, значит модульный JAR-файл создан правильно:

```
module com.jdojo.intro@1.0 (file:///C:/Java9Revealed/lib/com.jdojo.intro-1.0.jar)
  requires mandated java.base (@9-ea)
  contains com.jdojo.intro
```

Если вы не понимаете, что здесь напечатано, обратитесь к главе 2, где подробно описано, как получить список видимых модулей и описания отдельных модулей с помощью параметра `--list-modules`.

Выполнение программы

Для выполнения Java-программы служит команда `java`:

```
java --module-path <module-path> --module <module>/<main-class>
```

Здесь `<module-path>` – путь к модулям, используемый для нахождения модулей. Параметр `--module` определяет, какой модуль выполнять, а также главный класс этого модуля. Если модульный JAR-файл содержит атрибут `main-class`, то можно задать только часть `<module>`, а `<main-class>` будет прочитан из JAR-файла.

> **Совет.** Вместо параметров `--module-path` и `-module` можно использовать короткие варианты `-p` и `-m`.

Следующая команда выполняет класс `com.jdojo.intro.Welcome` в модуле `com.jdojo.intro`. Предполагается, что текущим каталогом является `C:\Java9Revealed`, а модульный JAR-файл – `C:\java9Revealed\lib\com.jdojo.intro-1.0.jar`.

```
C:\Java9Revealed>java --module-path lib
 --module com.jdojo.intro/com.jdojo.intro.Welcome
```

```
Добро пожаловать в систему модулей.
Имя модуля: com.jdojo.intro
```

Как видим, программа отработала правильно. Поскольку при упаковке мы указали, что главным классом модуля является `com.jdojo.intro.Welcome`, то следующая команда дает такой же результат, как предыдущая:

```
C:\Java9Revealed>java --module-path lib --module com.jdojo.intro
```

Можно также указать в качестве пути к модулям каталог, содержащий весь код модуля. Напомним, что откомпилированный код находится в каталоге `mods`. Следующая команда дает точно такой же результат:

```
C:\Java9Revealed>java --module-path mods
 --module com.jdojo.intro/com.jdojo.intro.Welcome
```

Попробуем выполнить модуль из каталога `mods`, указав только имя модуля:

```
C:\Java9Revealed>java --module-path mods --module com.jdojo.intro
```

```
module com.jdojo.intro does not have a MainClass attribute, use -m <module>/<main-class>
```

Вот те на! Ошибка. Сообщение говорит, что в файле `module-info.class`, найденном в каталоге `mods\com.jdojo.intro`, нет имени главного класса. Это правда. В объявлении модуля мы не можем указать ни имя главного класса, ни версию. При компиляции можно задать только версию. А имя главного класса (и версию тоже) можно задать лишь при упаковке модуля командой `jar`. Следовательно, файл `module-info.class` в `lib\com.jdojo.intro-1.0.jar` содержит имя главного класса, а одноименный файл в каталоге `mods\com.jdojo.intro` – нет. Так что если вы хотите выполнять откомпилированный код из развернутого каталога, то должны указывать имя главного класса вместе с именем модуля.

У команды `java` есть также параметр `-jar`, позволяющий выполнить главный класс из JAR-файла. Выполните такую команду:

```
C:\Java9Revealed>java -jar lib\com.jdojo.intro-1.0.jar
```

```
Добро пожаловать в систему модулей.
Имя модуля: null
```

Опять неожиданность! Правильно напечатана только первая строка. Программа нашла главный класс и выполнила код его метод `main()`. Но вместо имени модуля напечатала `null`. В чем же дело? Ведь раньше все было правильно.

Вы должны понимать, как ведет себя команда `java` в JDK 9. Параметр `-jar` существовал и раньше. Но в JDK 9 тип можно найти в модуле, пользуясь путем к модулям, или на пути к классам. Если тип найден на пути к классам, то он становится частью так называемого *безымянного* модуля. Он теряет свою связь с исход-

ным модулем, хотя и загружен из модульного JAR-файла. Вообще, если модульный JAR-файл находится на пути к классам, то он рассматривается как обычный (а не модульный) JAR, и файл `module-info.class` игнорируется. С любым загрузчиком классов приложения ассоциирован безымянный модуль. Все типы, загруженные из пути к классам, становятся членами безымянного модуля загрузчика. Безымянному модулю соответствует экземпляр класса `Module`, для которого метод `getName()` возвращает `null`.

В предыдущей команде модульный JAR-файл `com.jdojo.intro-1.0.jar` трактовался как обычный JAR, и все определенные в нем типы (в данном случае только класс `Welcome`) загружались как часть безымянного модуля. Потому-то мы и получили `null` в качестве имени модуля. А как команда `java` нашла имя главного класса? Напомним, что когда мы задаем имя главного класса в программе `jar`, это имя сохраняется в двух местах:

- в файле `module-info.class`;
- в файле `META-INF/MANIFEST.MF`.

Так вот команда `java` читает имя главного класса из `META-INF/MANIFEST.MF`.

Мы можем также воспользоваться параметром `--class-path` команды `java` для выполнения класса `Welcome`. Если поместить файл `lib\com.jdojo.intro-1.0.jar` на путь к классам, то он будет трактоваться как обычный JAR, и класс `Welcome` будет загружен в безымянный модуль загрузчика классов приложения. Таким способом мы привыкли выполнять класс до выхода JDK 9:

```
C:\Java9Revealed>java --class-path lib\com.jdojo.intro-1.0.jar
 com.jdojo.intro.Welcome
```

```
Добро пожаловать в систему модулей.
Имя модуля: null
```

Работа с NetBeans IDE

Если вы прочитали предыдущие разделы этой главы, то сейчас будет гораздо проще. Мы рассмотрим, как создать модуль в NetBeans IDE. О том, как установить версию NetBeans с поддержкой JDK 9, было сказано в главе 1.

Настройка IDE

Запустите NetBeans. При первом запуске появится начальная страница, показанная на рис. 3.1. Если вы больше не хотите ее видеть, сбросьте флажок **Show On Startup** (Показывать при запуске) в правом верхнем углу. Чтобы закрыть начальную страницу, щелкните по крестику в заголовке ее вкладки.

Выберите из меню пункт **Tools** → **Java Platforms** (Сервис → Платформы Java) – появится диалоговое окно **Java Platform Manager** (Диспетчер платформ Java), показанное на рис 3.2. Я работаю с NetBeans на JDK 1.8, показанном в списке платформ. Если вы работаете с JDK 9, то в списке будет присутствовать JDK 9 и дополнительная настройка не требуется, так что можете переходить к следующему разделу.

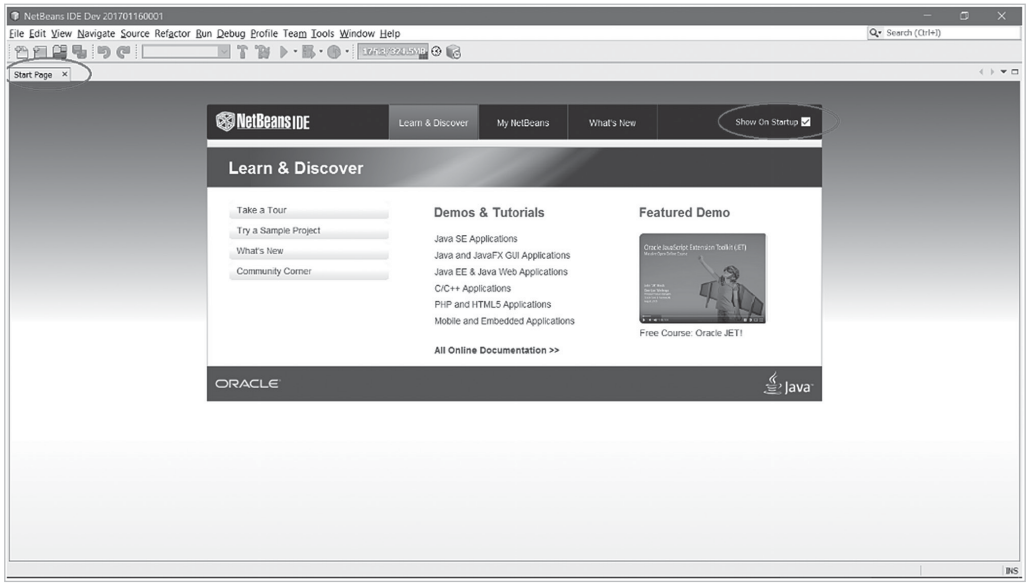

Рис. 3.1. Начальный экран NetBeans

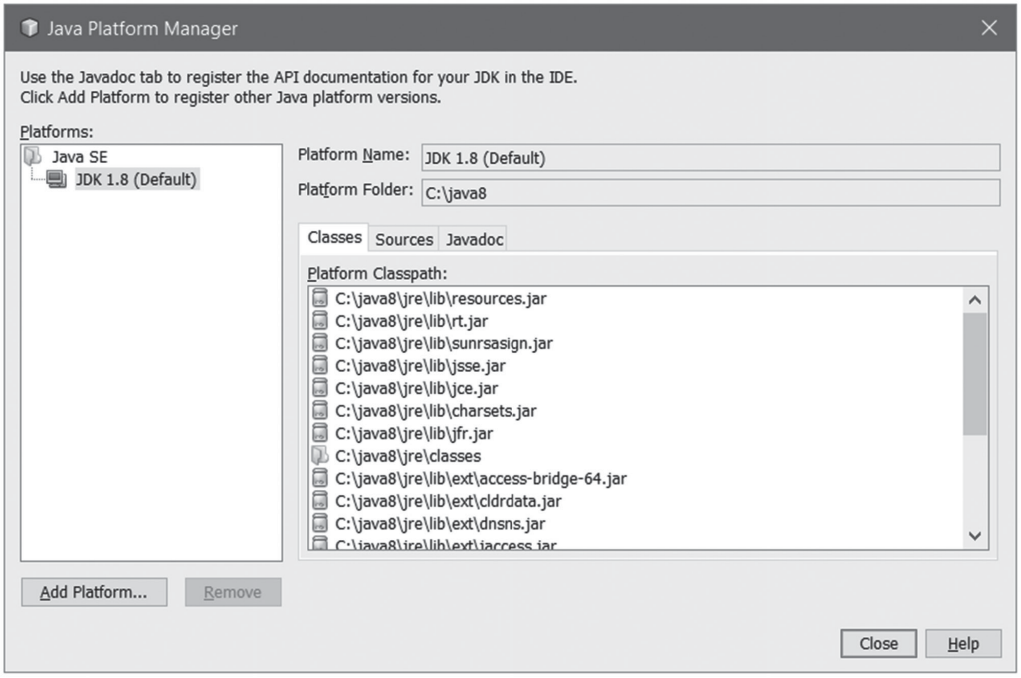

Рис. 3.2. Диалоговое окно Java Platform Manager

Если в списке **Platforms** (Платформы) присутствует JDK 9, то IDE уже настроена на работу с JDK 9, и это диалоговое окно можно закрыть нажатием на кнопку **Close**. В противном случае нажмите кнопку **Add Platform** (Добавить платформу) – откроется диалоговое окно **Add Java Platform**, показанное на рис. 3.3. Выберите

переключатель **Java Standard Edition**. Нажмите кнопку **Next** (Далее) для отображения папок платформ (рис. 3.4).

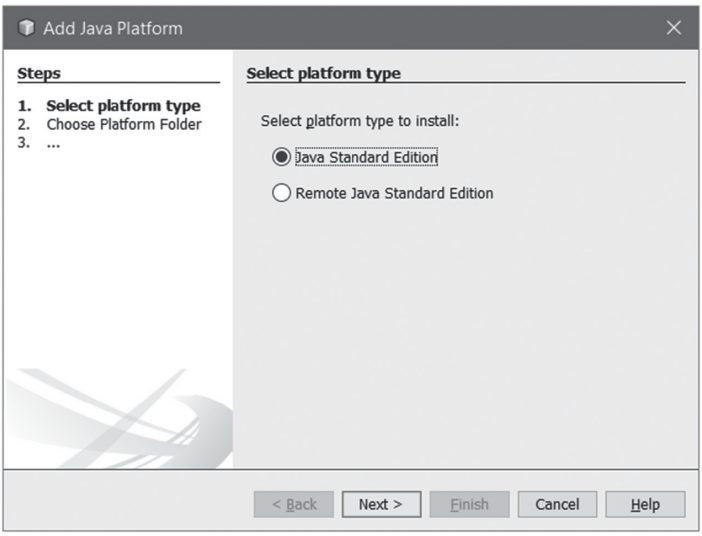

Рис. 3.3. Выбор типа платформы

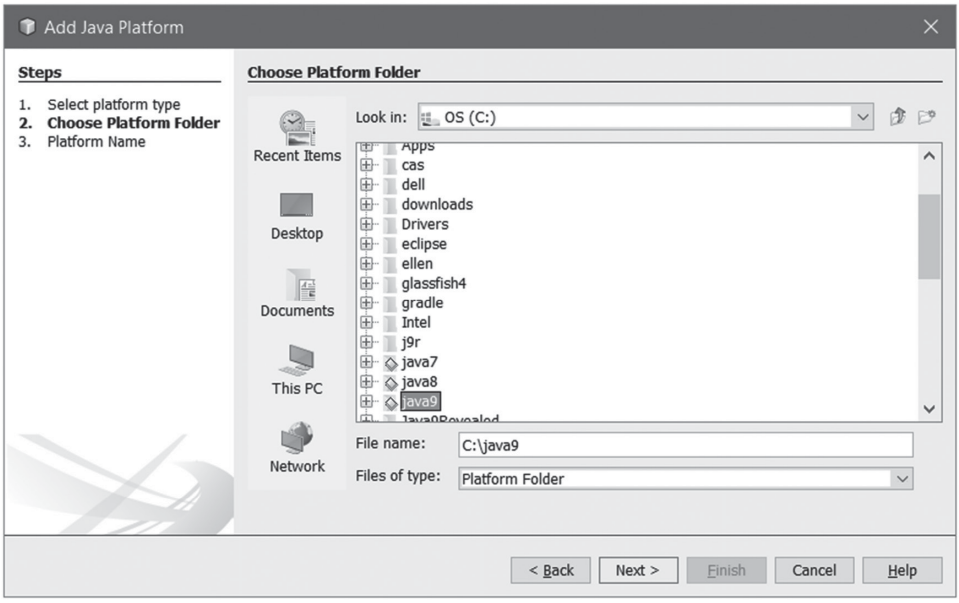

Рис. 3.4. Выбор папки платформы

В диалоговом окне **Add Java Platform** выберите каталог, куда будет установлен JDK 9. Я выбрал `C:\java9`. Нажмите кнопку **Next**. Появляется окно, показанное на рис. 3.5. Поля **Platform Name** (Имя платформы) и **Platform Sources** (Исходные файлы платформы) уже заполнены.

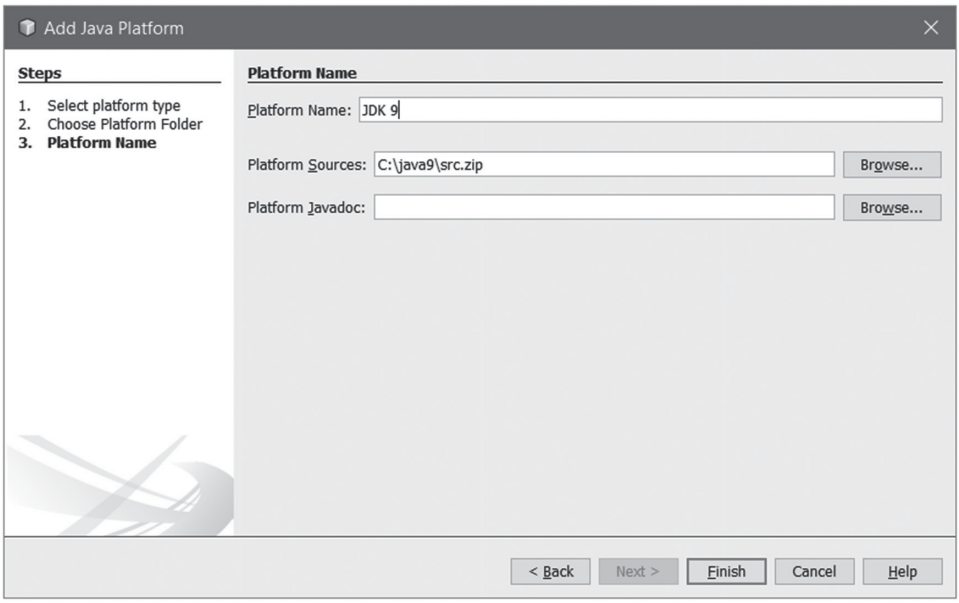

Рис. 3.5. Диалоговое окно Add Java Platform

После нажатия кнопки **Finish** (Готово) снова появится окно **Java Platform Manager**, но в списке платформ уже будет присутствовать JDK 9, как показано на рис. 3.6. Закройте диалоговое окно, нажав кнопку **Close**. Настройка NetBeans завершена.

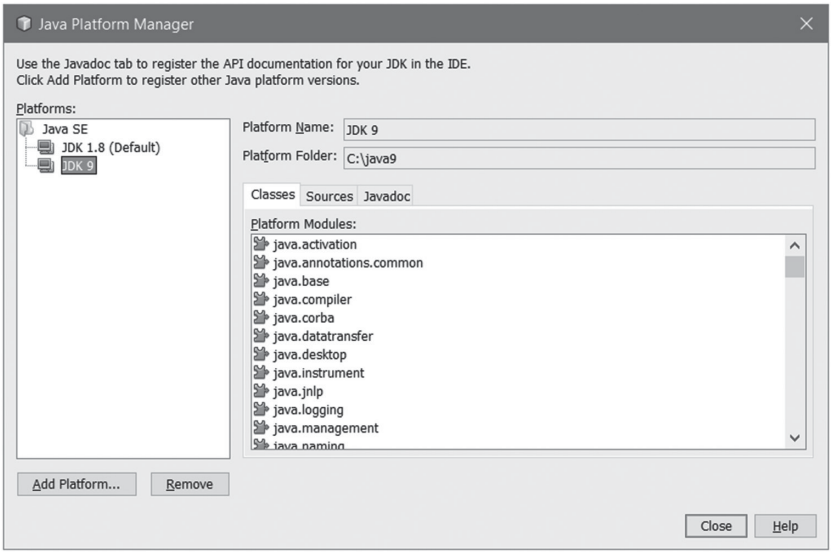

Рис. 3.6. Окно Java Platform Manager, в котором присутствуют платформы JDK 1.8 и JDK 9

Создание проекта Java

Выберите команду **File** → **New Project** (Файл → Создать проект) или нажмите клавиши **Ctrl+Shift+N** – откроется окно **New Project** (Создать проект), показанное на рис 3.7. Убедитесь, что в списке **Categories** (Категории) выбран элемент Java, а в списке **Projects** (Проекты) – элемент Java Application (Приложение Java). Нажмите кнопку **Next** – откроется окно **New Java Application** (Новое приложение Java), показанное на рис. 3.8.

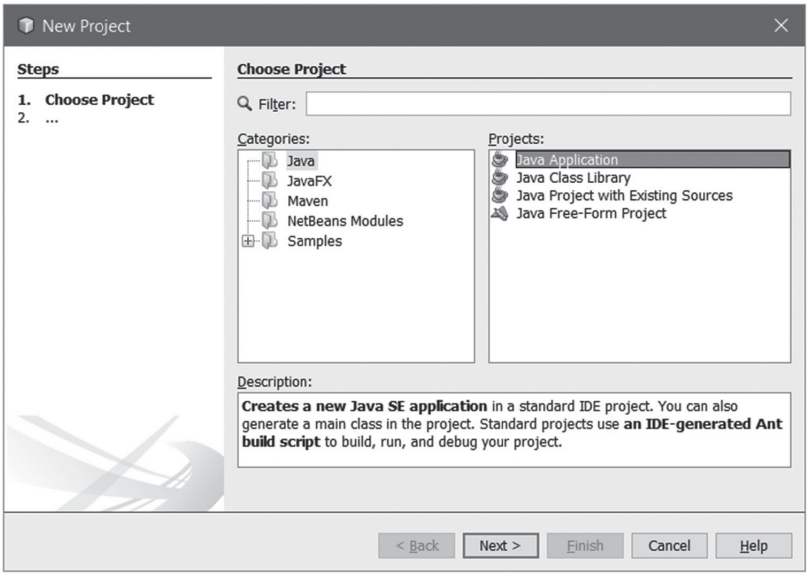

Рис. 3.7. Диалоговое окно New Project

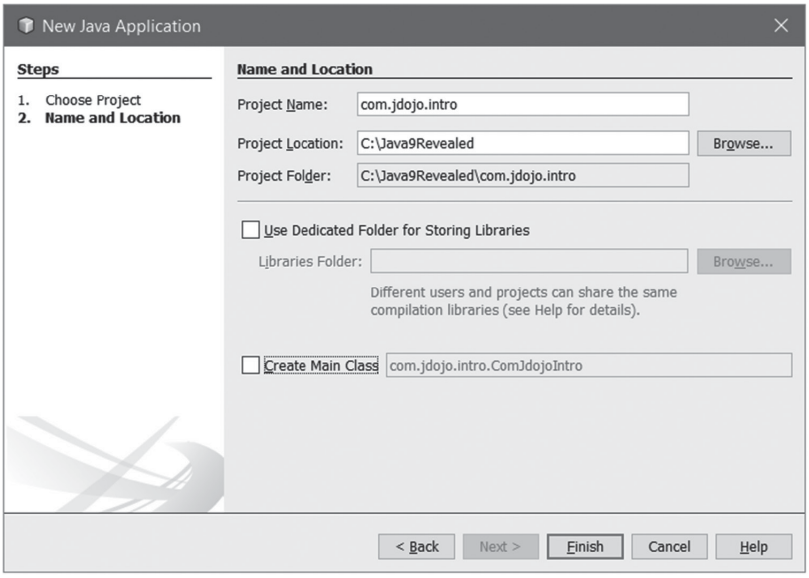

Рис. 3.8. Диалоговое окно New Java Application

Заполните следующие поля:

- введите имя проекта `com.jdojo.intro`;
- введите местоположение проекта `C:\Java9Revealed` или любое другое по своему выбору. В этом каталоге будет создан проект NetBeans;
- флажок **Use Dedicated Folder for Storing Libraries** (Использовать отдельную папку для хранения библиотек) оставьте неотмеченным;
- флажок **Create Main Class** (Создать главный класс) не отмечайте.

Нажмите кнопку **Finish**, чтобы завершить создание проекта. На рис. 3.9. показана NetBeans в этот момент.

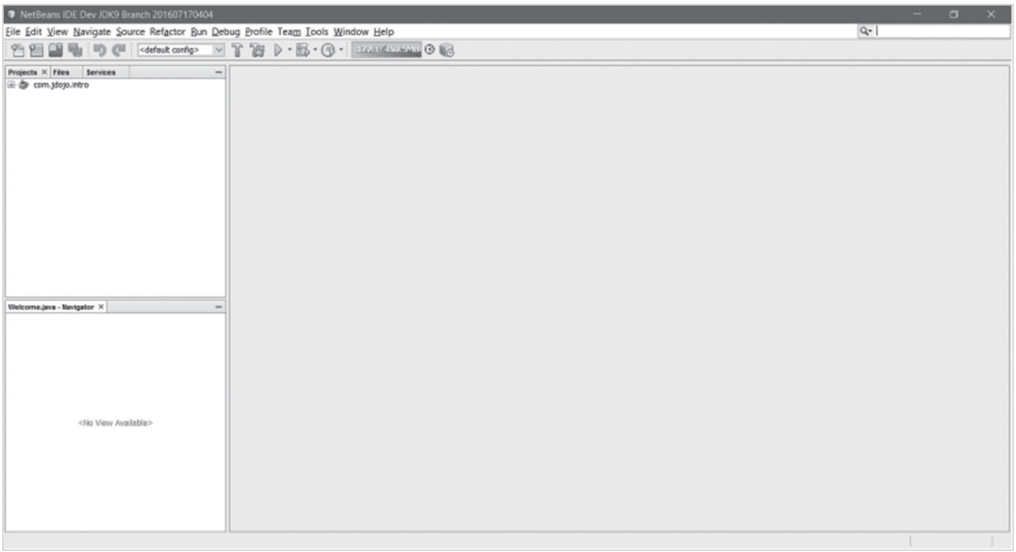

Рис. 3.9. NetBeans после создания нового проекта Java

Для нового проекта NetBeans создает стандартный набор каталогов. Мы создали проект с именем `com.jdojo.intro` в каталоге `C:\Java9Revealed`. Среда NetBeans создаст подкаталоги для хранения исходных файлов, откомпилированных файлов классов и модульного JAR-файла. Она также подготовит служебные каталоги и файлы для самого проекта. Будут созданы следующие каталоги:

```
C:\Java9Revealed
  com.jdojo.intro
    build
      classes
    dist
    src
```

Каталог `com.jdojo.intro` назван по имени проекта. В каталоге `src` хранится исходный код, а в каталоге `build` – весь сгенерированный и откомпилированный код. Весь откомпилированный код проекта находится в каталоге `build\classes`. В каталоге `dist` находится модульный JAR-файл. Отметим, что каталоги `build` и `dist` создаются при добавлении первого класса в проект, а не в момент создания проекта без главного класса.

Задание свойств проекта

Проект `com.jdojo.intro` по-прежнему настроен на версию JDK 1.8. Ее надо сменить на JDK 9. Выберите проект `com.jdojo.intro` на вкладке **Projects** (Проекты) и щелкните правой кнопкой мыши. Выберите из контекстного меню пункт **Properties** (Свойства), как показано на рис. 3.10. Открывается окно **Project Properties** (Свойства проекта), показанное на рис. 3.11.

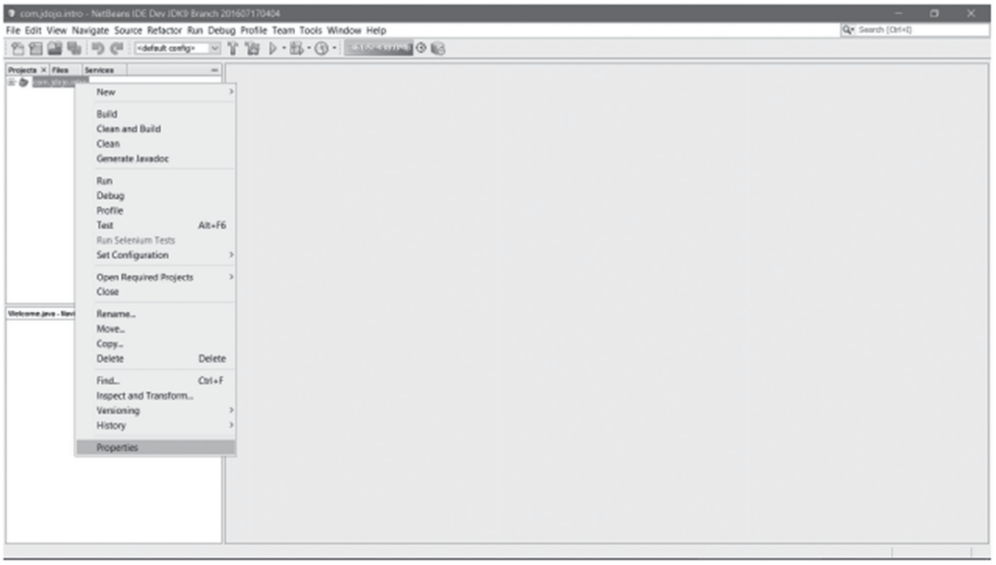

Рис. 3.10. Открытие диалогового окна Project Properties

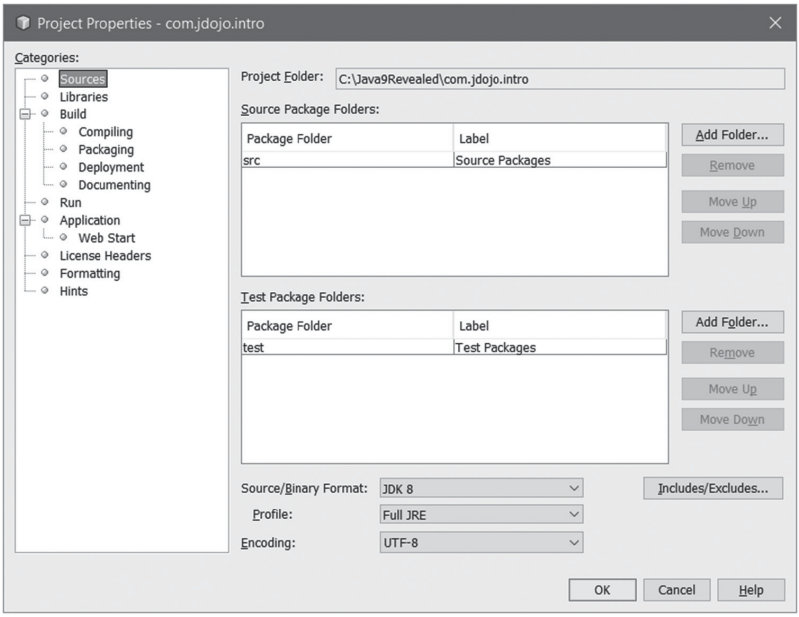

Рис. 3.11. Диалоговое окно Project Properties

Выберите в списке категорий пункт **Libraries** (Библиотеки). В выпадающем списке **Java Platform** (Платформа Java) выберите JDK 9, как показано на рис. 3.12.

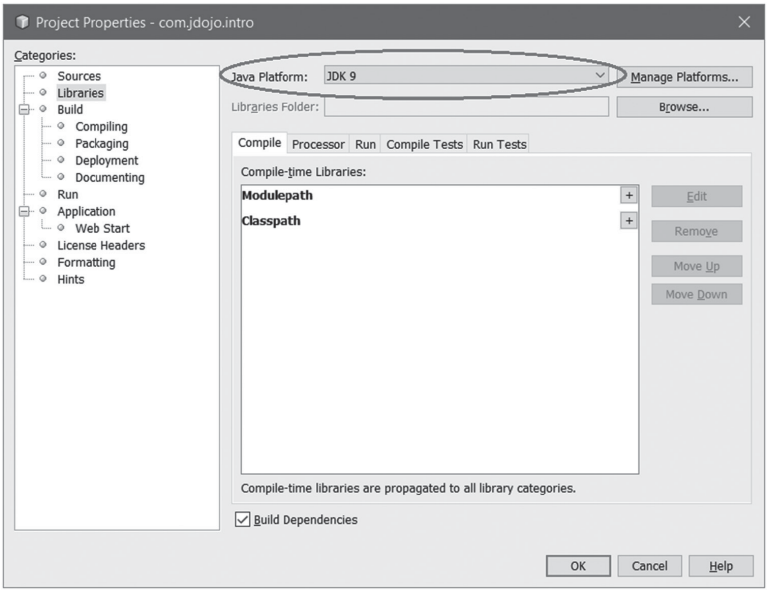

Рис. 3.12. Выбор JDK 9 в качестве платформы Java для проекта

Выберите в списке категорий пункт **Sources** (Исходные файлы). В выпадающем списке **Source/Binary Format** (Формат исходного/бинарного файла) выберите JDK 9, как показано на рис 3.13.

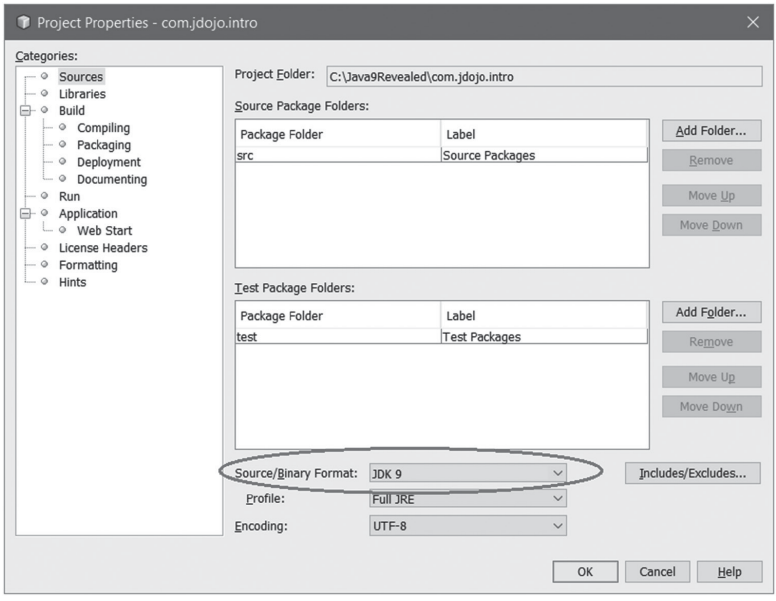

Рис. 3.13. Выбор JDK 9 в качестве формата исходного/двоичного кода

Нажмите кнопку **OK**. Проект `com.jdojo.intro` настроен на JDK 9.

Добавление объявления модуля

В этом разделе описано, как добавить определение модуля `com.jdojo.intro` в проект NetBeans. Для этого мы должны включить в проект файл `module-info.java`. Щелкните правой кнопкой мыши по узлу проекта и выберите из контекстного меню команду **New** (Создать), как показано на рис. 3.14. Если вы увидите в меню пункт **Java Module Info** (Модуль Java), выберите его. В противном случае выберите пункт **Other** (Другое) – тогда откроется диалоговое окно **New File** (Создать файл), как показано на рис. 3.15.

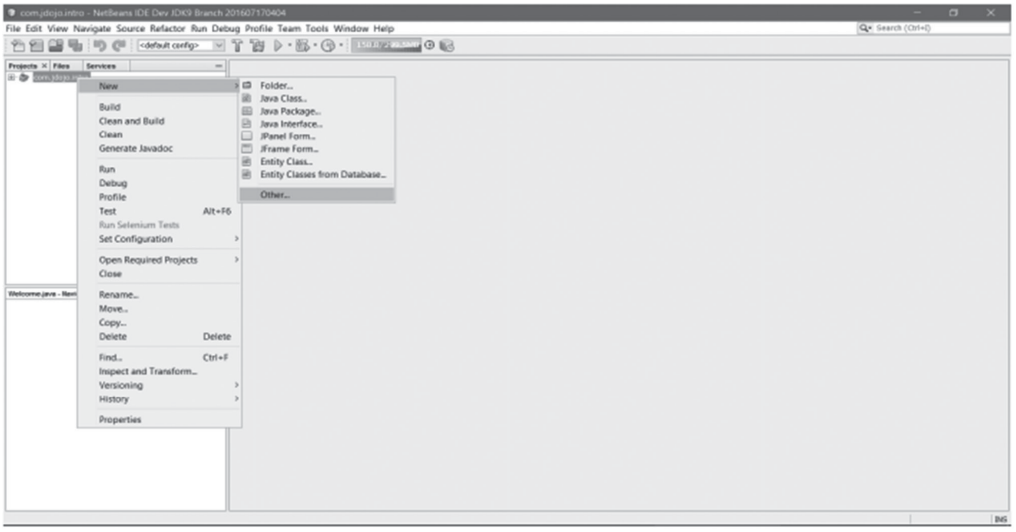

Рис. 3.14. Добавление в проект определения модуля

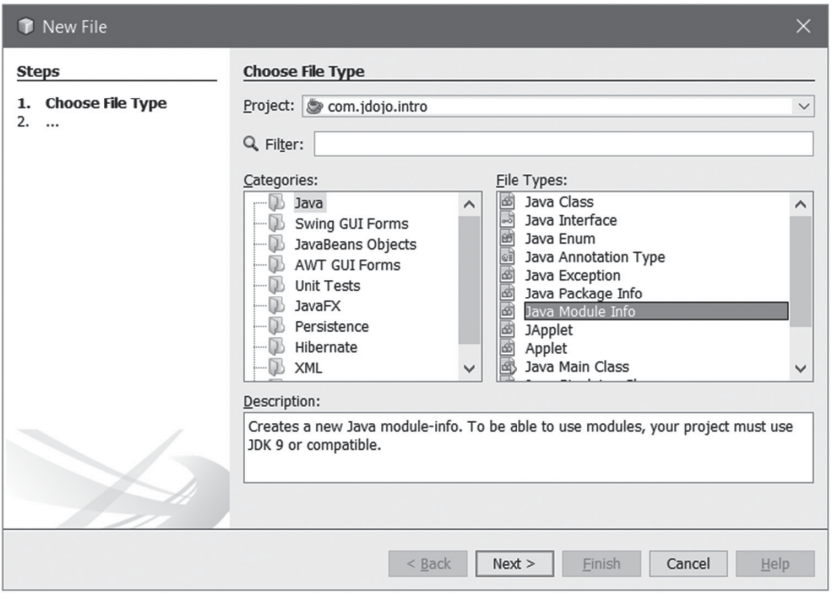

Рис. 3.15. Диалоговое окно New File для добавления информации о модуле Java

Выберите в списке категорий Java, а в списке типов файлов – Java Module Info. Нажмите кнопку **Next** – откроется окно **New Java Module Info**. Нажмите кнопку **Finish**, чтобы завершить создание определения модуля. В проект будет добавлен файл `module-info.java`, как показано на рис. 3.16. Он добавляется в корень каталога с исходным кодом и в дереве файлов проекта находится под элементом `<default-package>`.

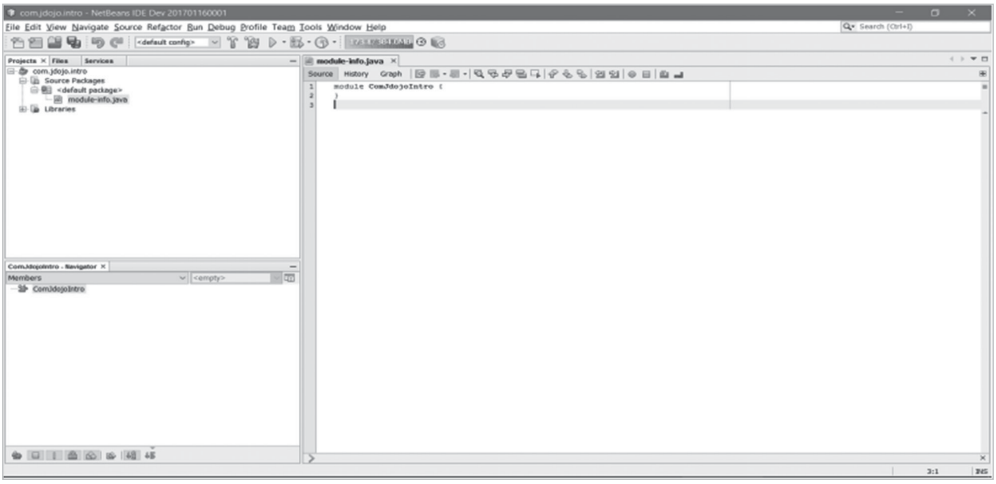

Рис. 3.16. Файл `module-info.java` в редакторе

По умолчанию NetBeans присваивает модулю имя, построенное по имени проекта. Точки удаляются, а начальная буква каждой части имени становится заглавной. Поскольку проект называется `com.jdojo.intro`, то модуль в файле `module-info.java` назван `ComJdojoIntro`. Измените это имя на `com.jdojo.intro`, как показано на рис. 3.17. Объявление модуля приведено в листинге 3.1 – оно такое же, как при работе с инструментами командной строки.

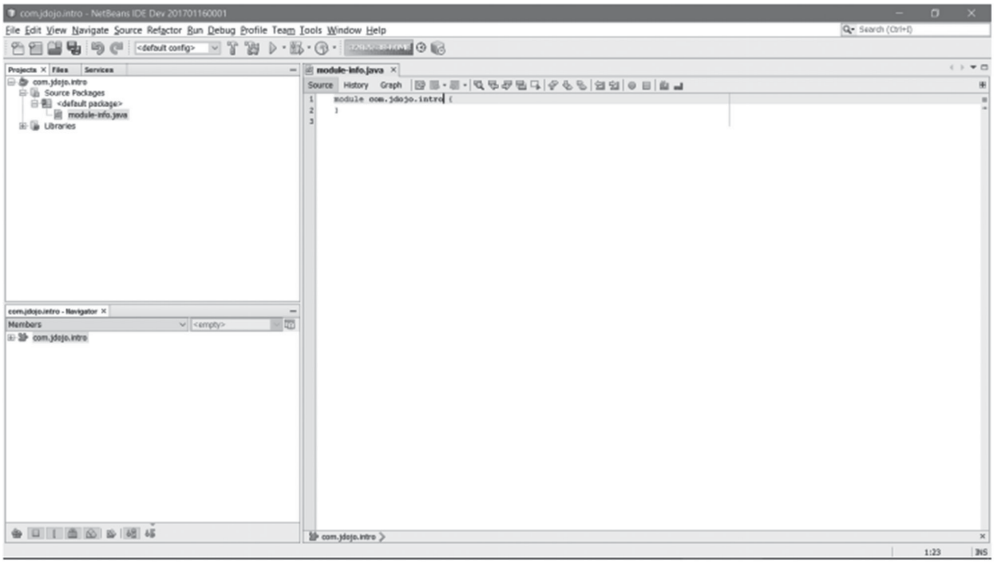

Рис. 3.17. Файл `module-info.java`, содержащий модуль с именем `com.jdojo.intro`

Просмотр графа модуля

NetBeans позволяет просмотреть граф модуля. Откройте файл `module-info.java` в редакторе и перейдите на вкладку **Graph** (Граф). Граф модуля `com.jdojo.intro` показан на рис. 3.18.

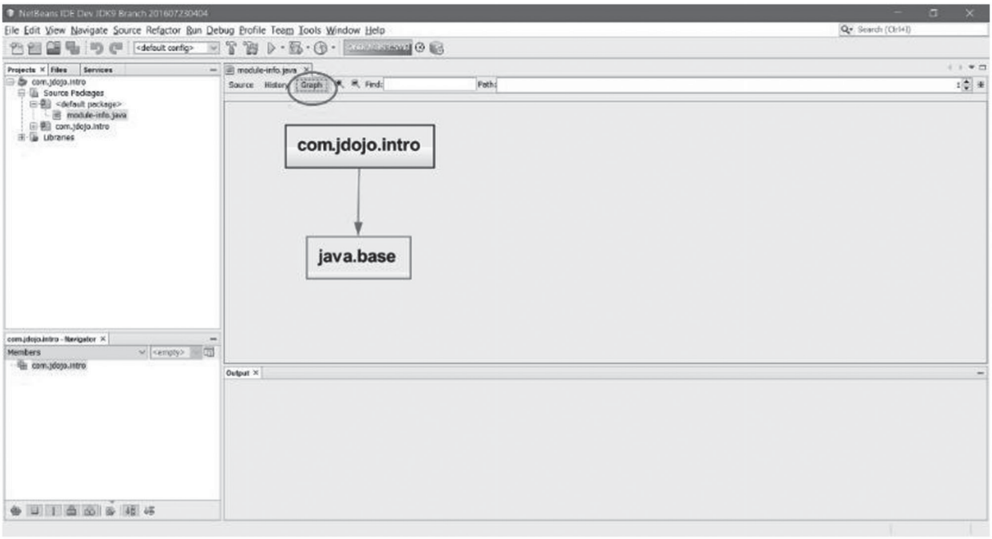

Рис. 3.18. Граф модуля `com.jdojo.intro`

Граф можно увеличивать и уменьшать, изменять расположение вершин и сохранять в виде изображения. Для получения меню относящихся к графу команд щелкните правой кнопкой мыши по занятой им области. Чтобы увидеть только ребра, входящие в некоторую вершину или исходящие из нее, выберите эту вершину. Для изменения положения верши просто буксируйте их мышью.

Пока файл `module-info.java` открыт в редакторе, вы можете перейти на вкладку **Source** (Источник) и отредактировать объявление модуля.

Написание исходного кода

В этом разделе мы добавим в проект класс `Welcome`, который будет частью пакета `com.jdojo.intro`. Из контекстного меню узла проекта выберите команду **New** → **Java Class** (Новый → Класс Java) – откроется диалоговое окно **New Java Class** (Новый класс Java), показанное на рис. 3.19. Введите строку `Welcome` в поле **Class Name** (Имя класса) и `com.jdojo.intro` в поле **Package** (Пакет). Закройте окно нажатием кнопки **Finish**.

Полный код класса `Welcome` приведен в листинге 3.2, он ничем не отличается от рассмотренного ранее. Замените им сгенерированный NetBeans код. На рис. 3.20 показано, как выглядит код класса `Welcome` в редакторе.

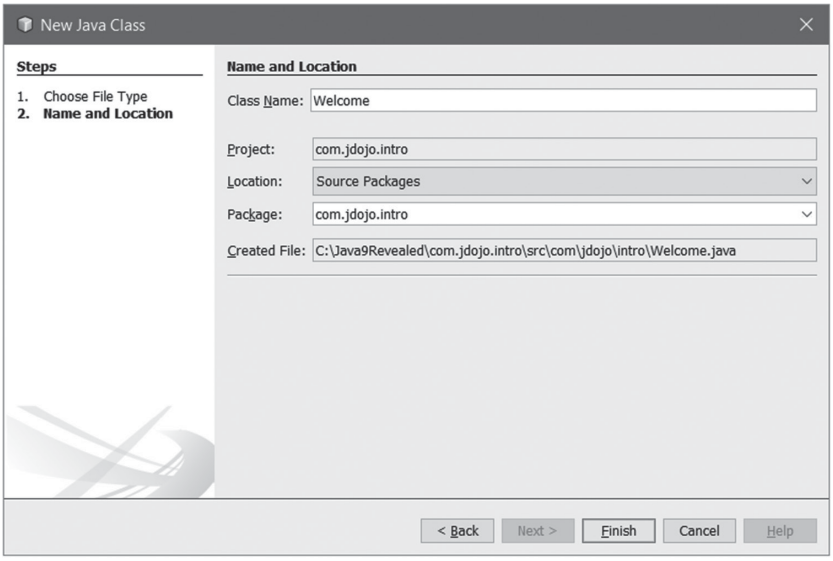

Рис. 3.19. Добавление в проект класса `Welcome`

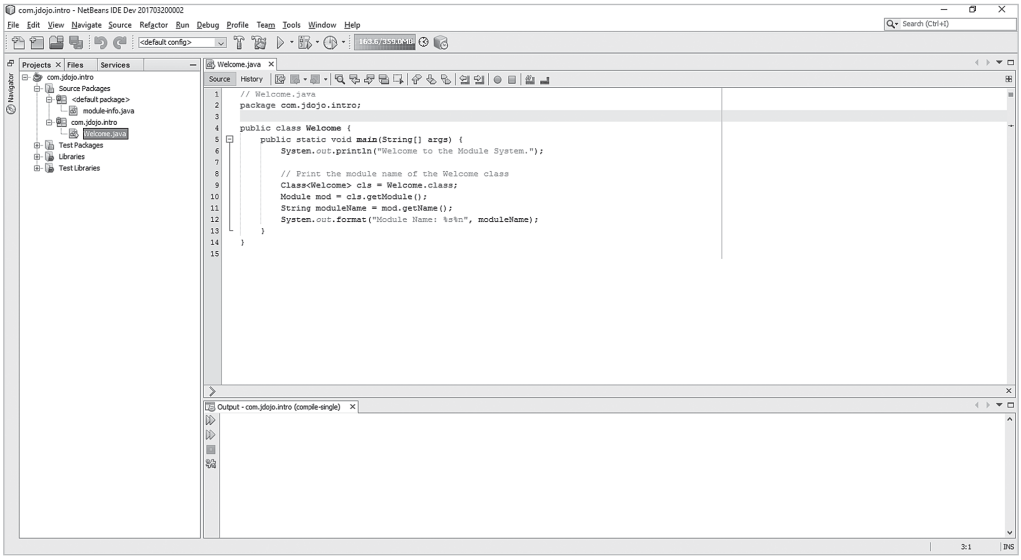

Рис. 3.20. Исходный код класса `Welcome` в редакторе

Компиляция исходного кода

NetBeans автоматически компилирует исходные файлы Java в момент сохранения. Для сохранения файла выполните команду **File** → **Save** (Файл → Сохранить) или нажмите **Ctrl+S**. Компиляцию при сохранении можно отключить, сбросив флажок **Compile on Save** (Компиляция при сохранении) на странице свойств проекта, как показано на рис. 3.21. По умолчанию этот флажок отмечен. Чтобы попасть на эту страницу, щелкните правой кнопкой мыши по узлу проекта и вы-

берите из меню пункт **Properties** (Свойства), а затем из списка **Categories** – категорию **Build** → **Compiling** (Собрать → Компиляция).

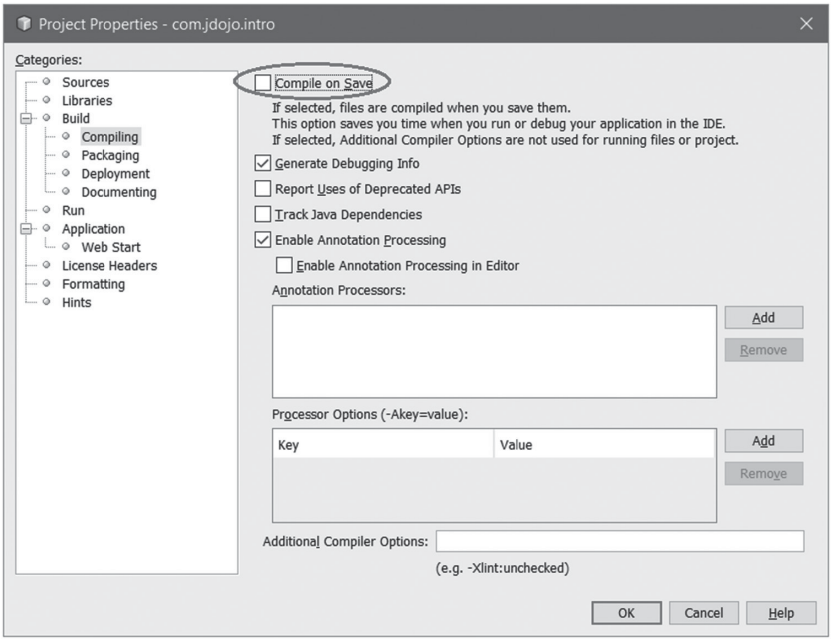

Рис. 3.21. Настройка режима компиляции при сохранении в NetBeans

Если компиляция при сохранении выключена, то компилировать исходный код нужно вручную посредством сборки проекта. Для этого выберите команду **Run** → **Build Project** (Выполнить → Собрать проект) или нажмите **F11**. Рекомендуется оставлять этот режим включенным.

Упаковка кода модуля

Для создания модульного JAR-файла необходимо предварительно собрать проект. Для этого выберите команду **Run** → **Build Project** или нажмите **F11**. Модульный JAR-файл создается в каталоге `<project-directory>\dist` и называется так же, как проект. Для модуля `com.jdojo.intro` модульный JAR-файл будет находиться в файле `C:\Java9Revealed\com.jdojo.intro\dist\com.jdojo.intro.jar`.

На момент написания книги NetBeans не поддерживала добавление имени главного класса и версии модуля в модульный JAR-файл. Это можно сделать вручную с помощью команды `jar`. Для этого воспользуйтесь параметром `-update`, как показано ниже (команду следует вводить на одной строке).

```
C:\Java9Revealed>jar --update
 --file com.jdojo.intro\dist\com.jdojo.intro.jar
 --module-version 1.0
 --main-class com.jdojo.intro.Welcome
```

Чтобы убедиться в правильности обновления модульного JAR-файла введите следующую команду. Она должна напечатать примерно то, что показано ниже:

```
C:\Java9Revealed>java --module-path com.jdojo.intro\dist
 --list-modules com.jdojo.intro
```

```
module com.jdojo.intro@1.0 (file:///C:/Java9Revealed/com.jdojo.intro/dist/com.jdojo.intro.jar)
 requires mandated java.base (@9-ea)
 contains com.jdojo.intro
```

Отметим, что при каждой сборке проекта NetBeans пересоздает файл `com.jdojo.intro.jar` в каталоге `C:\Java9Revealed\com.jdojo.intro\dist`, так что команду обновления модульного JAR-файла придется выполнять заново. В окончательной версии NetBeans, которая выйдет примерно в то же время, что и JDK 9, эти свойства, вероятно, можно будет задать прямо в IDE.

Выполнение программы

Чтобы выполнить программу, выберите команду **Run** → **Run Project** (Выполнить → Запустить проект) или нажмите **F6**. Если вы увидите диалоговое окно **Run Project** (Выполнить проект), показанное на рис. 3.22, значит, проект не настроен для выполнения.

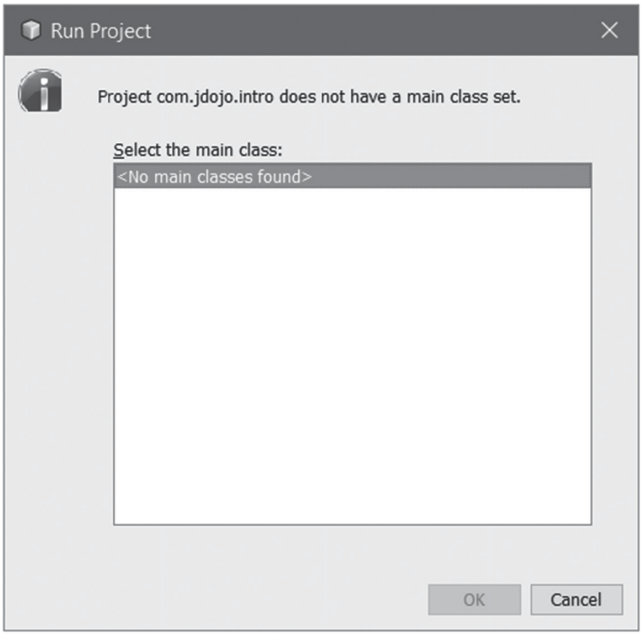

Рис. 3.22. Диалоговое окно Run Project

В таком случае есть два варианта.

- Можно указать класс, в котором есть метод `main`. Кстати, на стадии обучения в проекте может быть несколько классов, содержащих метод `main`, так что эта возможность будет не лишней.
- Можно настроить проект для выполнения. Этот вариант имеет смысл использовать, когда в проекте существует один и только один класс с методом `main`.

Чтобы выполнить класс, щелкните правой кнопкой мыши по исходному файлу (с расширением `.java`) класса, содержащего метод `main()`, на вкладке **Projects** и выберите из меню команду **Run File** (Выполнить файл) или выберите файл класса и нажмите **Shift+F6**. Для выполнения класса `Welcome` выберите файл `Welcome.java` и выполните его. Программа печатает результат на панели вывода, как показано на рис. 3.23.

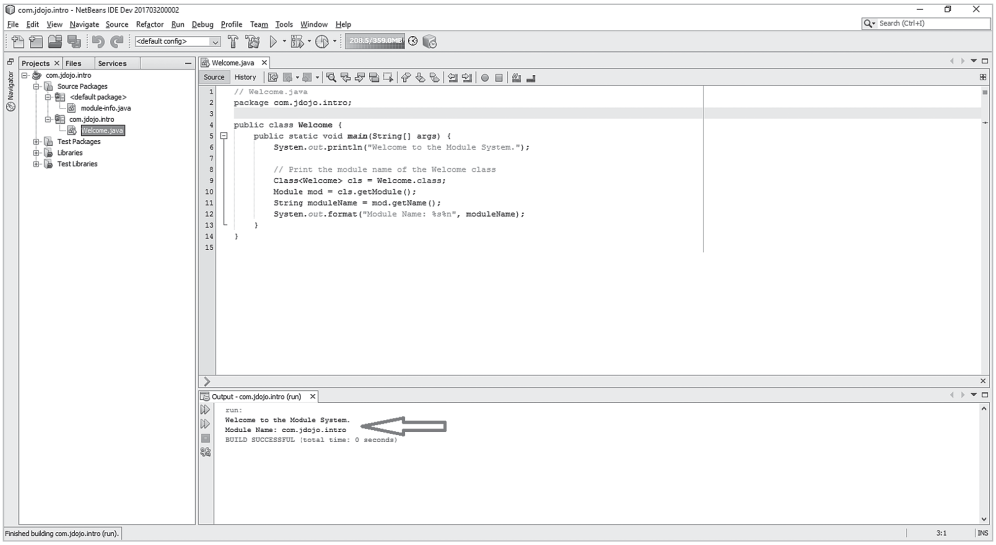

Рис. 3.23. Выполнение файла `Welcome.java`

Чтобы настроить проект для выполнения, щелкните по нему правой кнопкой мыши и выберите пункт **Properties**. В диалоговом окне **Project Properties** выберите узел **Run** (Выполнение) в списке **Categories**. Введите `com.jdojo.intro.Welcome` в поле **Main Class** (Главный класс) (см. рис. 3.24). Чтобы закрыть окно, нажмите **OK**.

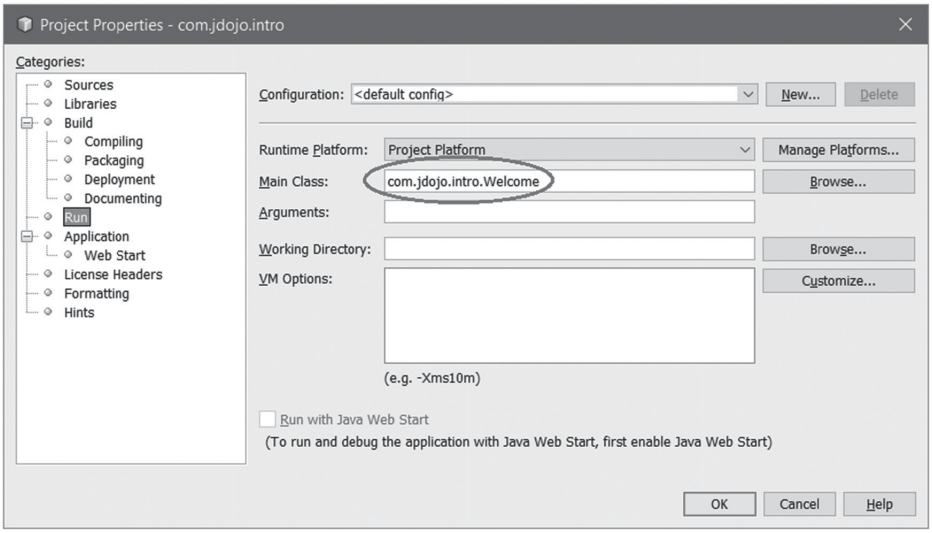

Рис. 3.24. Настройка проекта NetBeans для выполнения

Настроив проект, вы можете выполнить его, выбрав команду **Run** → **Run Project** или нажав клавишу **F6**. При этом будет выполняться главный класс, заданный в конфигурации проекта.

Резюме

Разработка Java-приложения с помощью модулей никак не изменяет способ организации типов в пакеты. Исходный код модуля включает файл `module-info.java` в корне иерархии пакетов. Это означает, что сам файл `module-info.java` принадлежит безымянному пакету. Этот файл содержит объявление модуля.

Компилятор `javac`, а также программы `jar` и `java` в JDK 9 модифицированы с учетом модулей. Компилятор понимает новые параметры: `--module-path` для поиска модулей приложения, `--module-source-path` для поиска исходного кода модуля и `--module-version` для задания версии компилируемого модуля. Команда `jar` позволяет задать главный класс и версию модуля, записываемые в модульный JAR-файл, для чего предназначены соответственно параметры `--main-class` и `--module-version`. Команду `java` можно запустить в режиме пути к классам, пути к модулям или в смешанном. Чтобы выполнить класс из модуля, необходимо задать путь к модулям с помощью параметра `--module-path` и имя главного класса с помощью параметра `-module`. Главный класс задается в виде `<module>/<main-class>`, где `<module>` – имя модуля, содержащего главный класс, а `<main-class>` – полное имя класса, в котором определен метод `main()`, являющийся точкой входа в приложение.

В JDK 9 каждый тип принадлежит какому-то модулю. Если тип найден на пути к классам, то он принадлежит безымянному модулю соответствующего загрузчика классов. В JDK 9 с каждым загрузчиком классов ассоциирован безымянный модуль, в который входят все типы, найденные этим загрузчиком на пути к классам. Тип, найденный и загруженный на пути к модулям, принадлежит модулю, в котором определен.

Экземпляр класса `Module` представляет модуль на этапе выполнения. Этот класс находится в пакете `java.lang` и позволяет получить любую информацию о модуле во время выполнения. Класс `Class` в JDK 9 также дополнен. Его метод `getModule()` возвращает объект `Module`, представляющий модуль, которому этот класс принадлежит. Класс `Module` содержит метод `getName()`, который возвращает имя модуля в виде строки, а для безымянного модуля – `null`.

Среда разработки NetBeans модифицируется для поддержки JDK 9 и создания модульных Java-приложений. На момент написания книги NetBeans позволяла создавать модули, компилировать их, упаковывать в модульные JAR-файлы и выполнять из IDE. Для каждого модуля необходимо было создавать отдельный проект Java. В окончательной версии будет разрешено включать в один проект несколько модулей. IDE поддерживает добавление файл `module-info.java` и располагает средствами для просмотра и сохранения графов модулей.

Глава 4

Зависимости модулей

Краткое содержание главы:

- как объявляются зависимости модулей;
- что такое неявное чтение модуля и как оно объявляется;
- различие между квалифицированным и неквалифицированным экспортом;
- объявление факультативной зависимости модуля на этапе выполнения;
- как раскрыть весь модуль или отдельные содержащиеся в нем пакеты для рефлексивного доступа;
- доступность типов в JDK 9;
- разделение пакета между несколькими модулями;
- ограничения на объявления модулей;
- различные типы модулей: именованные, безымянные, явные, автоматические, нормальные и раскрытые;
- как дизассемблировать определение модуля с помощью программы `javap`.

Пример программы в этой главе разрабатывается поэтапно. В прилагаемом к книге коде приведен окончательный результат. Чтобы посмотреть, как работают промежуточные версии, этот код придется немного модифицировать.

Объявление зависимостей модуля

Если модулю необходимы открытые типы, определенные в каком-то другом модуле, то второй модуль должен экспортировать пакет, содержащий эти типы, а первый модуль должен прочитать второй. Обратите внимание на асимметрию: модуль, содержащий типы, экспортирует *пакет*, а модуль, потребляющий типы, читает *модуль*.

Рассмотрим два модуля: `com.jdojo.address` и `com.jdojo.person`. Модуль `com.jdojo.address` содержит пакет `com.jdojo.address`, в котором определен класс `Address`. Модулю `com.jdojo.person` необходим класс `Address` из модуля `com.jdojo.address`. На рис. 4.1 показан граф модуля `com.jdojo.person`. Опишем все шаги разработки этих модулей.

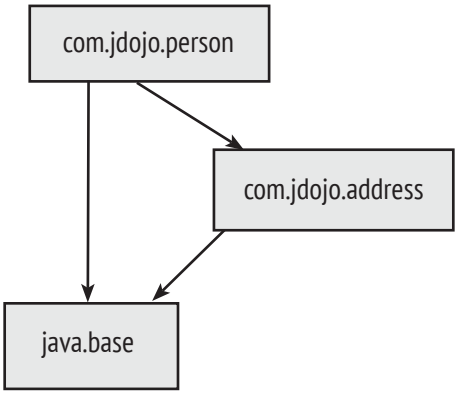

Рис. 4.1. Граф модуля `com.jdojo.person`

Создайте в NetBeans два проекта Java: `com.jdojo.address` и `com.jdojo.person`. В каждом проекте будет находиться код одноименного модуля. В листингах 4.1 и 4.2 показаны объявление модуля и код класса `Address` соответственно. `Address` – простой класс, содержащий четыре поля и соответствующие акцессоры чтения и установки. Я задал для всех полей значения по умолчанию, чтобы не набирать их каждый раз в примерах.

Листинг 4.1. Объявление модуля com.jdojo.address

```
// module-info.java
module com.jdojo.address {
    // Экспортировать пакет com.jdojo.address
    exports com.jdojo.address;
}
```

Листинг 4.2. Класс Address

```
// Address.java
package com.jdojo.address;

public class Address {
    private String line1 = "1111 Main Blvd.";
    private String city = "Jacksonville";
    private String state = "FL";
    private String zip = "32256";

    public Address() {
    }

    public Address(String line1, String line2, String city,
                   String state, String zip) {
        this.line1 = line1;
        this.city = city;
```

```
        this.state = state;
        this.zip = zip;
    }

public String getLine1() {
    return line1;
}

public void setLine1(String line1) {
    this.line1 = line1;
}

public String getCity() {
    return city;
}

public void setCity(String city) {
    this.city = city;
}

public String getState() {
    return state;
}

public void setState(String state) {
    this.state = state;
}

public String getZip() {
    return zip;
}

public void setZip(String zip) {
    this.zip = zip;
}

@Override
public String toString() {
    return "[Line1:" + line1 + ", State:"" + state +
           ", City:" + city + ", ZIP:" + zip + "]";
}
}
```

Предложение `exports` экспортирует пакет всем или некоторым *именованным* модулям. Все открытые типы, определенные в экспортированном пакете, доступны на этапах компиляции и выполнения. На этапе выполнения с помощью рефлексии можно получить доступ только к открытым членам открытых типов. Неот-

крытые члены открытых типов недоступны для рефлексии, даже если выполнить для них метод `setAccessible(true)`. Синтаксически предложение `exports` имеет вид:

```
exports <package>;
```

В таком виде оно экспортирует все открытые типы в пакете `<package>` всем модулям, т. е. любой модуль, читающий данный, сможет использовать все открытые типы, определенные в `<package>`.

Модуль `com.jdojo.address` экспортирует пакет `com.jdojo.address`, поэтому находящийся в нем открытый класс `Address` доступен любому модулю.

В листингах 4.3 и 4.4 показаны объявление модуля `com.jdojo.person` и код класса `Person` соответственно.

Листинг 4.3. Объявление модуля `com.jdojo.person`

```
// module-info.java
module com.jdojo.person {
    // Читать модуль com.jdojo.address
    requires com.jdojo.address;

    // Экспортировать пакет com.jdojo.person
    exports com.jdojo.person;
}
```

Листинг 4.4. Класс Person

```
// Person.java
package com.jdojo.person;

import com.jdojo.address.Address;
    public class Person {
private long personId;
private String firstName;
private String lastName;
private Address address = new Address();

public Person(long personId, String firstName, String lastName) {
        this.personId = personId;
        this.firstName = firstName;
        this.lastName = lastName;
}

public long getPersonId() {
    return personId;
}

public void setPersonId(long personId) {
    this.personId = personId;
```

```
}

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

public Address getAddress() {
    return address;
}

public void setAddress(Address address) {
    this.address = address;
}

@Override
public String toString() {
    return "[Person Id:" + personId + ", First Name:" + firstName +
           ", Last Name:" + lastName + ", Address:" + address + "]";
}
}
```

Класс `Person` находится в модуле `com.jdojo.person` и содержит поле типа `Address`, определенного в модуле `com.jdojo.address`. Это означает, что модуль `com.jdojo.person` читает модуль `com.jdojo.address`. Для объявления этой зависимости служит предложение `requires` в объявлении модуля `com.jdojo.person`:

```
// Читать модуль com.jdojo.address
requires com.jdojo.address;
```

Предложение `requires` используется, чтобы объявить зависимость одного модуля от другого. Синтаксически оно имеет вид:

```
requires [transitive] [static] <module>;
```

Здесь `<module>` – имя модуля, читаемого данным модулем. Модификаторы `transitive` и `static` факультативные. Если модификатор `static` присутствует, то модуль `<module>` должен присутствовать на этапе компиляции, но может отсутствовать на

этапе выполнения. Наличие модификатора `transitive` означает, что модуль, который читает данный, неявно читает и модуль `<module>`. Пример использования модификатора `transitive` будет рассмотрен чуть ниже. Компилятор добавляет предложение `requires java.base`, если оно не было задано явно. Следующие два объявления модуля `com.jdojo.common` эквивалентны:

```
// Объявление 1
module com.jdojo.common {
    // Компилятор сам добавит предложение чтения модуля java.base
}

// Объявление 2
module com.jdojo.common {
    // Явное чтение java.base
    requires java.base;
}
```

Объявление модуля `com.jdojo.person` включает предложение `requires`, означающее, что модуль `com.jdojo.address` необходим как на этапе компиляции, так и на этапе выполнения. При компиляции модуля `com.jdojo.person` модуль `com.jdojo.address` должен находиться на пути к модулям. В NetBeans можно включить проект в путь к модулям. Щелкните правой кнопкой мыши по проекту `com.jdojo.person` и выберите в меню пункт **Properties** (Свойства). В списке категорий выберите **Libraries**. Перейдите на вкладку **Compile** (Компилировать) и нажмите знак + в строке **Modulepath**, затем выберите в меню пункт **Add Project** (Добавить проект), как показано на рис. 4.2. Появится окно **Add Project**, показанное на рис. 4.3.

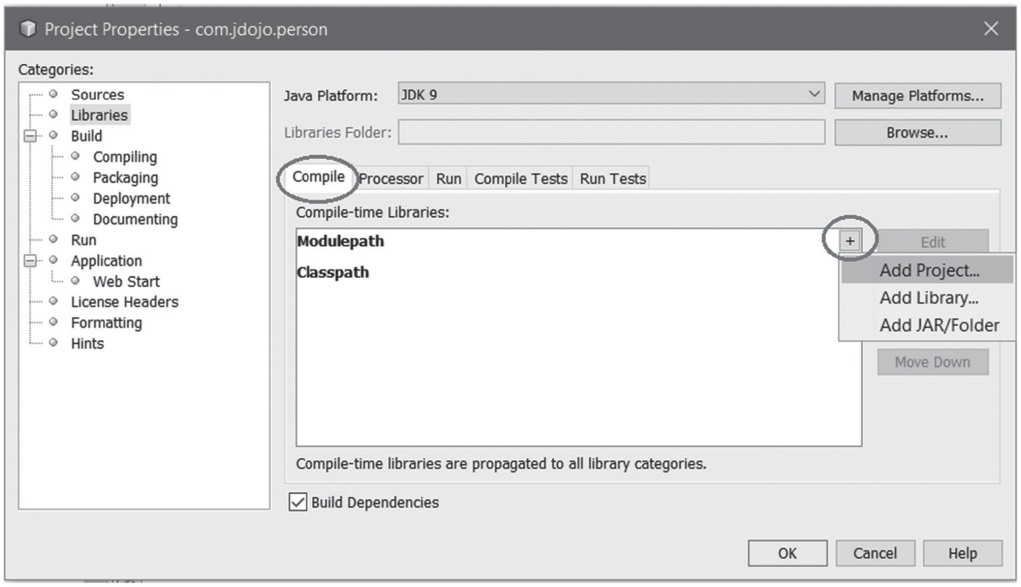

Рис. 4.2. Задание пути к модулям для проекта в NetBeans

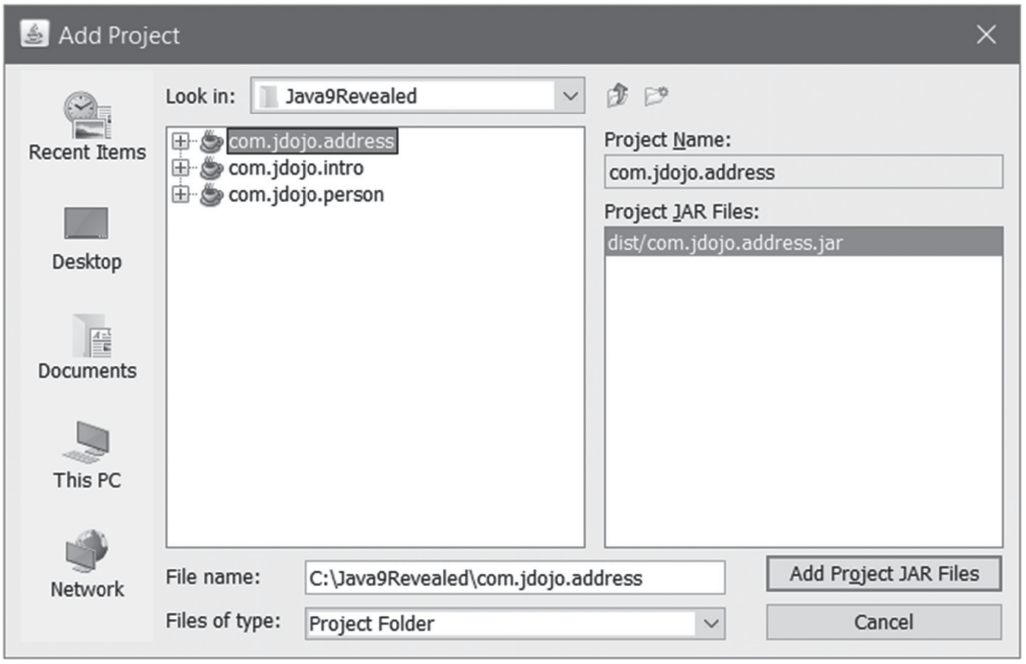

Рис. 4.3. Выбор проекта NetBeans, добавляемого в путь к модулям

В этом окне перейдите в каталог, содержащий модуль `com.jdojo.address`, выберите его и нажмите кнопку **Add Project JAR Files** (Добавить файлы JAR проекта). После этого вы вернетесь в окно **Properties**, где увидите, что проект добавлен в путь к модулям, как показано на рис. 4.4. Для завершения этого шага нажмите **OK**.

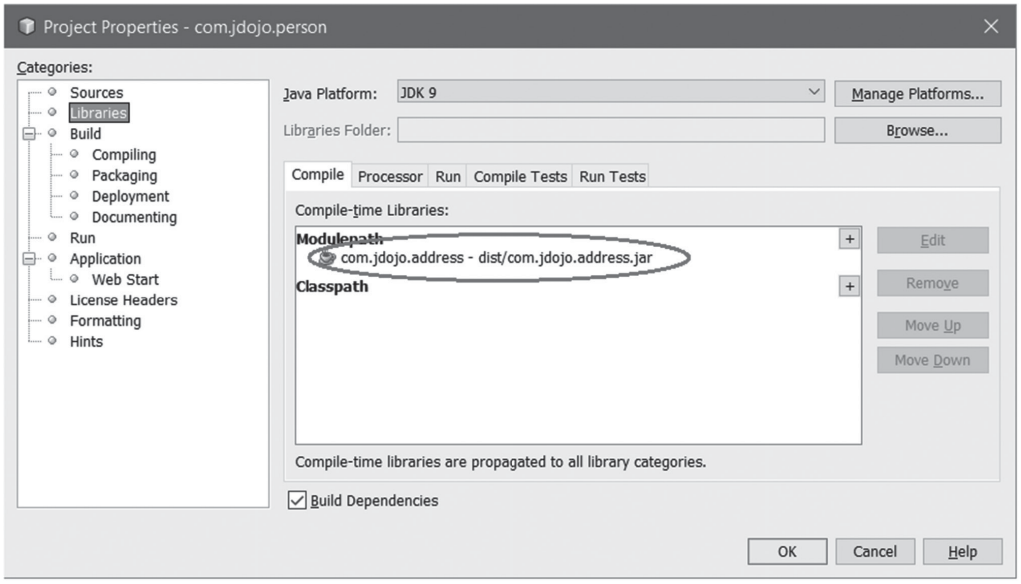

Рис. 4.4. Проект NetBeans добавлен в путь к модулям

Модуль com.jdojo.person также экспортирует пакет com.jdojo.person, чтобы находящиеся в нем открытые типы, например класс Person, можно было использовать в других модулях.

В листинге 4.5 приведен код класса Main, находящегося в модуле com.jdojo.person. Из результатов его выполнения явствует, что мы можем использовать класс Address из модуля com.jdojo.address. Если при выполнении примера вы столкнулись с какими-то проблемами, обратитесь к следующему разделу, где перечислены возможные ошибки и способы их исправления.

Листинг 4.5. Класс Main для тестирования модуля com.jdojo.person

```
// Main.java
package com.jdojo.person;

import com.jdojo.address.Address;

public class Main {
    public static void main(String[] args) {
Person john = new Person(1001, "John", "Jacobs");

String fName = john.getFirstName();
String lName = john.getLastName();
Address addr = john.getAddress();

System.out.printf("%s %s%n", fName, lName);
System.out.printf("%s%n", addr.getLine1());
System.out.printf("%s, %s %s%n", addr.getCity(),
addr.getState(), addr.getZip());
  }
}
```

```
John Jacobs
1111 Main Blvd.
Jacksonville, FL 32256
```

Сейчас мы можем запустить пример и из командной строки. Необходимо включить каталоги с откомпилированными классами или модульные JAR-файлы модулей com.jdojo.person и com.jdojo.address в путь к модулям. В следующей команде используются откомпилированные классы из обоих каталогов build\classes:

```
C:\Java9Revealed>java --module-path
 com.jdojo.person\build\classes;com.jdojo.address\build\classes
 --module com.jdojo.person/com.jdojo.person.Main
```

При сборке проекта NetBeans, содержащего модуль, модульный JAR-файл записывается в подкаталог dist каталога проекта. Так, в процессе сборки проекта com.jdojo.person создается файл com.jdojo.person.jar в каталоге C:\Java9Revealed\com.jdojo.

person\dist. Кроме того, пересобираются все проекты, от которых зависит данный. В данном случае сборка проекта com.jdojo.person влечет за собой также сборку проекта com.jdojo.address. После сборки модуля com.jdojo.person его можно выполнить такой командой:

```
C:\Java9Revealed>java --module-path
 com.jdojo.person\dist;com.jdojo.address\dist
 --module com.jdojo.person/com.jdojo.person.Main
```

Исправление возможных ошибок

В начале работы с JDK 9 могут произойти различные ошибки. В следующих разделах мы обсудим, что может случиться и как это исправить.

Пустой пакет

Сообщение об ошибке выглядит так:

```
error: package is empty or does not exist: com.jdojo.address
    exports com.jdojo.address;
                    ^
1 error
```

Ошибка возникает при попытке откомпилировать объявление модуля com.jdojo.address, не включив исходный код класса Address. Модуль экспортирует пакет com.jdojo.address, и в этом пакете должен быть определен хотя бы один тип.

Модуль не найден

Сообщение об ошибке выглядит так:

```
error: module not found: com.jdojo.address
    requires com.jdojo.address;
                     ^
1 error
```

Ошибка возникает при попытке откомпилировать объявление модуля com.jdojo.person, не включив com.jdojo.address в путь к модулям. Модуль com.jdojo.person читает модуль com.jdojo.address, поэтому должен находить его на пути к модулям как на этапе выполнения, так и на этапе компиляции. При запуске из командной строки нужно указать путь к модулю com.jdojo.address с помощью параметра --module-path. О том, как настраивать путь к модулям в NetBeans, написано в предыдущем разделе.

Пакет не существует

Сообщение об ошибке выглядит так:

```
error: package com.jdojo.address does not exist
import com.jdojo.address.Address;
                         ^
error: cannot find symbol
```

```
    private Address address = new Address();
            ^
symbol:   class Address
location: class Person
```

Ошибка возникает при попытке откомпилировать классы `Person` и `Main` в модуле `com.jdojo.person`, не добавив предложение `requires` в объявление модуля. Сообщение говорит, что компилятор не смог найти класс `com.jdojo.address.Address`. Чтобы исправить ошибку, нужно добавить предложение `requires com.jdojo.address` в объявление модуля `com.jdojo.person` и поместить `com.jdojo.address` в путь к модулям.

Исключение при разрешении модуля

Часть сообщения об ошибке выглядит так:

```
Error occurred during initialization of VM
java.lang.module.ResolutionException: Module com.jdojo.person not found
...
```

Ошибка возникает по следующим причинам при попытке запустить пример из командной строки:

- путь к модулю указан неправильно;
- путь к модулю правильный, но в указанных каталогах нет откомпилированного кода или не найдены модульные JAR-файлы.

Предположим, что пример был запущен такой командой:

```
C:\Java9Revealed>java --module-path
 com.jdojo.person\dist;com.jdojo.address\dist
 --module com.jdojo.person/com.jdojo.person.Main
```

В случае ошибки убедитесь, что следующие модульные JAR-файлы существуют:

- `C:\Java9Revealed\com.jdojo.person\dist\com.jdojo.person.jar`
- `C:\Java9Revealed\com.jdojo.address\dist\com.jdojo.address.jar`

Если это не так, то соберите проект `com.jdojo.person` в NetBeans.

Если при запуске примера были указаны каталоги с откомпилированным кодом:

```
C:\Java9Revealed>java --module-path
com.jdojo.person\build\classes;com.jdojo.address\build\classes
--module com.jdojo.person/com.jdojo.person.Main
```

то проверьте, были ли проекты откомпилированы в NetBeans.

Неявное чтение

Если модуль может читать другой модуль, хотя тот не был упомянут в предложении `requires`, то говорят, что первый модуль *неявно* читает второй. Любой модуль неявно читает модуль `java.base`, но им неявное чтение не ограничивается. Прежде чем показывать, как добавить неявное чтение модуля, я на примере продемонстрирую, зачем это может понадобиться. В предыдущем разделе мы создали два

модуля: com.jdojo.address и com.jdojo.person, причем второй читает первый с помощью такого объявления:

```
module com.jdojo.person {
    requires com.jdojo.address;
...
}
```

Класс Person в модуле com.jdojo.person ссылается на класс Address в модуле com.jdojo.address. Создадим еще один модуль, com.jdojo.person.test, который читает com.jdojo.person. Его объявление приведено в листинге 4.6.

Листинг 4.6. Объявление модуля com.jdojo.person.test

```
// module-info.java
module com.jdojo.person.test {
    requires com.jdojo.person;
}
```

Добавьте проект com.jdojo.person в путь к модулям в проекте com.jdojo.person.test, иначе при компиляции кода в листинге 4.6 возникнет ошибка:

```
C:\Java9Revealed\com.jdojo.person.test\src\module-info.java:3: error: module not found:
com.jdojo.person
    requires com.jdojo.person;
1 error
```

В листинге 4.7. приведен код класса Main в модуле com.jdojo.person.test.

Листинг 4.7. Класс Main для тестирования модуля com.jdojo.person.test

```
// Main.java
package com.jdojo.person.test;

import com.jdojo.person.Person;

public class Main {
    public static void main(String[] args) {
        Person john = new Person(1001, "John", "Jacobs");

        // Получить город, где живет Джон, и напечатать его
        String city = john.getAddress().getCity();
        System.out.printf("John lives in %s%n", city);
    }
}
```

Код в методе main() совсем простой – создается объект Person и читается значение города в адресе человека. Но при попытке откомпилировать этот модуль возникает ошибка:

```
com.jdojo.person.test\src\com\jdojo\person\test\Main.java:11: error: getCity() in
Address is defined in an inaccessible class or interface
```

```
        String city = john.getAddress().getCity();
                                        ^
1 error
```

В сообщении утверждается, что класс `Address` недоступен модулю `com.jdojo.person.test`. Напомним, что класс `Address` находится в модуле `com.jdojo.address`, который `com.jdojo.person.test` не читает. На первый взгляд кажется, что код должен компилироваться. У нас есть доступ к классу `Person`, который использует класс `Address`, поэтому и у нас должна быть возможность использовать `Address`. Но метод `john.getAddress()` возвращает объект типа `Address`, к которому у нас нет доступа. Система модулей сделала то, что от нее требуется, – обеспечила инкапсуляцию, определенную в модуле `com.jdojo.address`. Если некоторый модуль хочет использовать класс `Address`, явно или неявно, то он должен прочитать модуль `com.jdojo.address`. И как исправить ошибку? Да просто нужно изменить объявление модуля `com.jdojo.person.test`, добавив чтение `com.jdojo.address`, как показано в листинге 4.8.

Листинг 4.8. Модифицированное объявление модуля `com.jdojo.person.test`

```
// module-info.java
module com.jdojo.person.test {
    requires com.jdojo.person;
    requires com.jdojo.address;
}
```

После добавления предложения `com.jdojo.address` мы получаем другую ошибку – не найден модуль `com.jdojo.address`. Для ее исправления нужно добавить проект `com.jdojo.address` в путь к модулям для проекта `com.jdojo.person.test`, как показано на рис. 4.5.

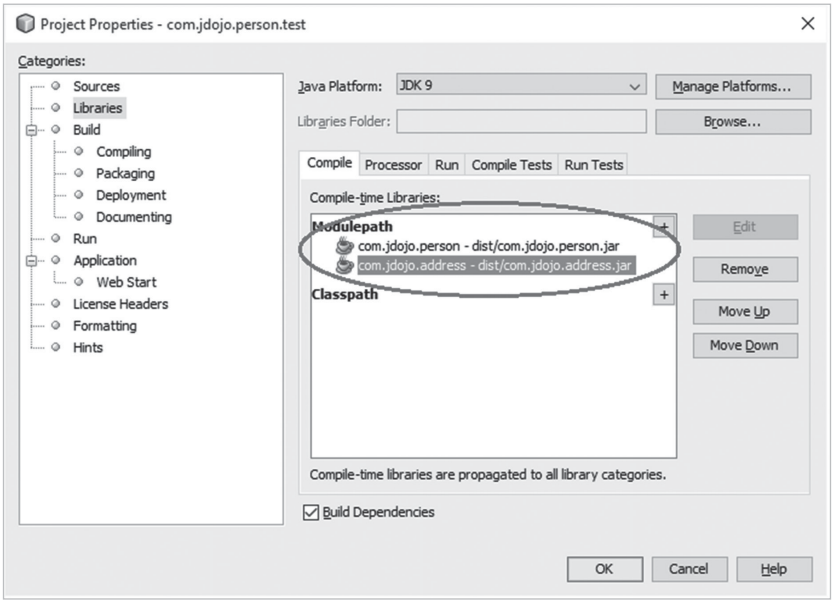

Рис. 4.5. Путь к модулям в проекте `com.jdojo.person.test`

На рис. 4.6 показан граф модуля `com.jdojo.person.test` в этот момент.

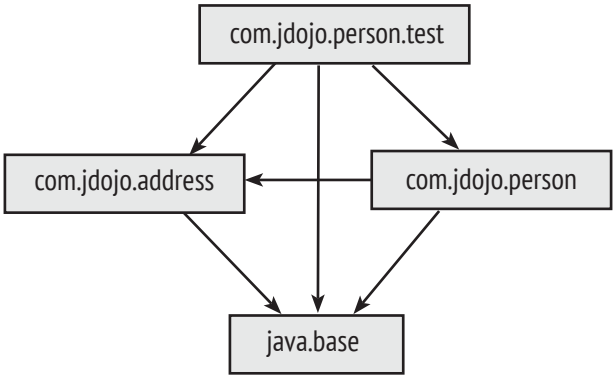

Рис. 4.6. Граф модуля `com.jdojo.person.test`

Откомпилируйте и выполните класс `Main` в модуле `com.jdojo.person.test module`.

```
John lives in Jacksonville
```

Мы решили проблему, добавив предложение `requires`. Но вполне может статься, что другие модули, читающие модуль `com.jdojo.person`, тоже хотят работать с адресом и, следовательно, должны будут добавить такое же предложение `requires`. Если модуль `com.jdojo.person` использует в своем открытом API типы из нескольких модулей, то каждый модуль, читающий `com.jdojo.person`, должен будет добавить предложения `requires` для всех них. Проектировщики JDK 9 знали об этой проблеме и предложили простой способ ее решения. Нам нужно будет только изменить объявление модуля `com.jdojo.person`, добавив модификатор `transitive` в предложение `requires com.jdojo.address`. Новое объявление показано в листинге 4.9.

Листинг 4.9. Модифицированное объявление модуля com.jdojo.person

```
// module-info.java
module com.jdojo.person {
    // Читать модуль com.jdojo.address
    requires transitive com.jdojo.address;

    // Экспортировать пакет com.jdojo.person
    exports com.jdojo.person;
}
```

Теперь можно удалить предложение `requires com.jdojo.address` из объявления модуля `com.jdojo.person.test`. Но проект `com.jdojo.address` все равно должен находиться на пути к модулям в проекте `com.jdojo.person.test`, потому что используется определенный в нем тип `Address`. Перекомпилируйте модуль `com.jdojo.person`, а затем перекомпилируйте и запустите модуль `com.jdojo.person.test`. Будет напечатан желаемый результат.

Если предложение `requires` содержит модификатор `transitive`, то модули, зависящие от данного, будут читать указанный в этом предложении модуль. Если говорить о листинге 4.9, то любой модуль, читающий `com.jdojo.person`, неявно читает и `com.jdojo.address`. Неявное чтение упрощает чтение объявления, но усложняет рассуждения о нем, потому что, глядя на объявление, мы не видим все зависимости модуля. На рис. 4.7 показан окончательный граф модуля `com.jdojo.person.test`.

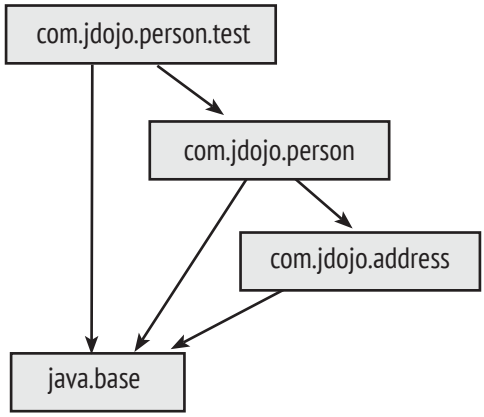

Рис. 4.7. Граф модуля `com.jdojo.person.test`

В процессе разрешения модулей граф пополняется ребром чтения для каждой транзитивной зависимости. В данном случае добавляется ребро чтения от модуля `com.jdojo.person.test` к модулю `com.jdojo.address` (см. рис. 4.8).

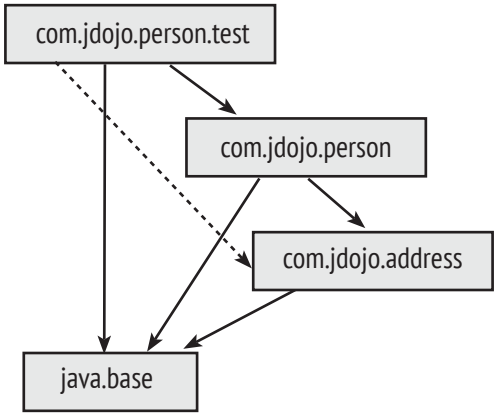

Рис. 4.8. Граф модуля `com.jdojo.person.test` после пополнения ребрами неявного чтения

Квалифицированный экспорт

Допустим, мы разрабатываем библиотеку или каркас, состоящий из нескольких модулей. В некотором модуле есть пакеты, содержащие API, предназначенные только для внутреннего использования в каких-то модулях. Это значит, что паке-

ты не должны экспортироваться всем модулям, их доступность должна быть ограничена. Эту задачу можно решить с помощью предложения квалифицированного экспорта:

```
exports <package> to <module1>, <module2>...;
```

Здесь `<package>` – имя пакета, экспортируемого из данного модуля, а `<module1>`, `<module2>` – имена модулей, которым разрешено читать данный. В следующем объявлении присутствуют предложения неквалифицированного и квалифицированного экспорта:

```
module com.jdojo.common {
    // Предложение неквалифицированного экспорта
    exports com.jdojo.zip;

    // Предложение квалифицированного экспорта
    exports com.jdojo.internal to com.jdojo.address;
}
```

Модуль `com.jdojo.common` экспортирует пакет `com.jdojo.zip` всем модулям, а пакет `com.jdojo.internal` – только модулю `com.jdojo.address`. Все открытые типы из пакета `com.jdojo.zip` будут доступны всем модулям, читающим `com.jdojo.common`, а открытые типы из пакета `com.jdojo.internal` – только модулю `com.jdojo.address`, если тот читает `com.jdojo.common`.

В JDK 9 можно найти много примеров квалифицированного экспорта. Модуль `java.base` содержит пакеты `sun.*` и `jdk.*`, экспортируемые нескольким именованным модулям. Следующая команда печатает объявление модуля `java.base` и, как видим, в ней есть несколько предложений квалифицированного экспорта:

```
c:\>javap jrt:/java.base/module-info.class
```

```
Compiled from "module-info.java"
module java.base {
  exports sun.net to jdk.plugin, jdk.incubator.httpclient;
  exports sun.nio.cs to java.desktop, jdk.charsets;
  exports sun.util.resources to jdk.localedata;
  exports jdk.internal.util.xml to jdk.jfr;
  exports jdk.internal to jdk.jfr;
  ...
}
```

Не все внутренние API в JDK 9 инкапсулированы. В пакетах `sun.*` есть несколько критически важных внутренних API, например класс `sun.misc.Unsafe`, которые использовались разработчиками раньше, и в JDK 9 они по-прежнему доступны. Эти пакеты помещены в модуль `jdk.unsupported`. Следующая команда печатает объявление модуля `jdk.unsupported`:

```
C:\Java9Revealed>javap jrt:/jdk.unsupported/module-info.class
```

```
Compiled from "module-info.java"
module jdk.unsupported@9-ea {
  requires java.base;
  exports sun.misc;
  exports com.sun.nio.file;
  exports sun.reflect;
  opens sun.misc;
  opens sun.reflect;
}
```

Факультативная зависимость

Система модулей проверяет зависимости модулей на этапах компиляции и выполнения. Но иногда нужно сделать зависимость обязательной на этапе компиляции и необязательной на этапе выполнения.

Возможно, вы разрабатываете библиотеку, которая работает лучше, если на этапе выполнения доступен некоторый модуль, а в противном случае соглашается на другой модуль, при котором тоже работает, но не оптимально. Такая библиотека должна гарантировать, что код, зависящий от факультативного модуля, не будет выполняться, если этого модуля нет.

Другой пример – модуль, экспортирующий аннотации. Среда выполнения Java игнорирует несуществующие типы аннотаций. Если используемая в программе аннотация отсутствует на этапе выполнения, то она игнорируется. Зависимости модулей проверяются на стадии инициализации и, если модуль отсутствует, приложение не запустится. Поэтому зависимость от модуля, содержащего типы аннотаций, должна быть объявлена факультативной.

Для объявления факультативной зависимости служит модификатор `static` в предложении `requires`:

```
requires static <optional-package>;
```

В следующем объявлении модуля присутствует факультативная зависимость от модуля `com.jdojo.annotation`:

```
module com.jdojo.claim {
    requires static com.jdojo.annotation;
}
```

В предложении `requires` могут быть одновременно модификаторы `transitive` и `static`:

```
module com.jdojo.claim {
    requires transitive static com.jdojo.annotation;
}
```

В таком случае они могут следовать в любом порядке. Следующее объявление эквивалентно предыдущему:

```
module com.jdojo.claim {
    requires static transitive com.jdojo.annotation;
}
```

Доступ к модулям с помощью рефлексии

Свыше 20 лет в Java был разрешен доступ ко всем членам типа – закрытым, открытым и защищенным – с помощью рефлексии. Можно было получить доступ к закрытому члену класса или объекта. Для этого достаточно всего лишь вызвать метод `setAccessible(true)` для члена объекта (поля, метода и т. д.). Далее в этой главе я буду называть доступ к неоткрытым членам типа с помощью рефлексии *глубокой рефлексией*.

Когда мы экспортируем пакет модуля, другим модулям доступны только открытые типы и открытые и защищенные члены открытых типов этого пакета – статически на этапе компиляции или рефлексивно на этапе выполнения. При проектировании системы модулей это оказалось большой проблемой. Существует несколько очень популярных каркасов, в т. ч. Spring и Hibernate, для работы которых совершенно необходим глубокий рефлексивный доступ к членам типов, определенных в прикладных библиотеках.

Серьезные трудности вызвало проектирование глубокого рефлексивного доступа к модульному коду. Разрешить глубокую рефлексию к типам экспортированного пакета значило нарушить *строгую инкапсуляцию*. При этом внешнему коду становится доступно все, даже если разработчик модуля не хотел раскрывать какие-то части. С другой стороны, если запретить глубокую рефлексию, то сообщество Java лишится некоторых широко распространенных каркасов и перестанут работать многие приложения, опирающиеся на рефлексию. Их просто нельзя будет перенести на платформу JDK 9.

После нескольких попыток проектировщикам системы модулей удалось найти компромисс – сесть на ежа и не уколоться! Модуль может поддерживать и строгую инкапсуляцию, и глубокий рефлексивный доступ, и даже то и другое одновременно. Действуют следующие правила.

- Экспортированный пакет разрешает доступ только к открытым типам и их открытым членам на этапах компиляции и выполнения. Тем самым обеспечивается строгая инкапсуляция.
- Можно раскрыть модуль, разрешив глубокую рефлексию для всех типов во всех пакетах этого модуля на этапе выполнения. Такой модуль называется *раскрытым*.
- Возможен нормальный модуль – не раскрытый для глубокой рефлексии, – в котором некоторые пакеты раскрыты на этапе выполнения, а все остальные строго инкапсулированы. Те пакеты, для которых глубокая рефлексия разрешена, называются *раскрытыми*.
- Иногда необходим доступ к типам пакета на этапе компиляции, чтобы можно было писать код, в котором эти типы используются, и одновременно рефлексивный доступ к тем же типам на этапе выполнения. Для этого пакет можно одновременно экспортировать и раскрыть.

Раскрытые модули

Далее я покажу, как объявить раскрытый модуль и раскрытый пакет. Сначала – синтаксис. Чтобы объявить раскрытый модуль, добавьте перед ключевым словом `module` модификатор `open`:

```
open module com.jdojo.model {
    // Здесь будут предложения модуля
}
```

В данном случае модуль `com.jdojo.model` раскрытый. В объявлении раскрытого модуля могут присутствовать предложения `exports`, `requires`, `uses` и `provides`, но не предложение `opens`, которое служит для раскрытия отдельного пакета для глубокой рефлексии. Поскольку раскрытый модуль раскрывает все свои пакеты, предложение `opens` было бы излишним и потому запрещено.

Раскрытие пакетов

Раскрытие пакета означает, что другие модули могут применять глубокую рефлексию к типам этого пакета. Пакет можно раскрыть всем или только избранным модулям. Синтаксически предложение `opens`, раскрывающее пакет всем модулям, выглядит так:

```
opens <package>;
```

Здесь пакет `<package>` доступен для глубокой рефлексии всем модулям. Чтобы раскрыть пакет конкретным модулям, используется квалифицированное предложение `opens`:

```
opens <package> to <module1>, <module2>...;
```

Здесь `<package>` раскрывается только модулям `<module1>`, `<module2>` и т. д. Ниже приведен пример предложения `opens` в объявлении модуля:

```
module com.jdojo.model {
    // Экспортировать пакет com.jdojo.util всем модулям
    exports com.jdojo.util;

    // Раскрыть пакет com.jdojo.util всем модулям
    opens com.jdojo.util;

    // Раскрыть пакет com.jdojo.model.policy только модулю hibernate.core
    opens com.jdojo.model.policy to hibernate.core;
}
```

Модуль `com.jdojo.model` экспортирует пакет `com.jdojo.util`, т. е. все его открытые типы и их открытые члены доступны на этапе компиляции и для обычной рефлексии на этапе выполнения. Во втором предложении тот же пакет раскрыт для глубокой рефлексии на этапе выполнения, а в третьем пакет `com.jdojo.model.policy` открыт для глубокой рефлексии только модулю `hibernate.core`, так что никакие другие модули не смогут получить доступ ни к одному из его типов, а `hibernate.core` вправе обращаться ко всем типам и всем их членам с помощью глубокой рефлексии на этапе выполнения.

Совет. Модулю, применяющему глубокую рефлексию к раскрытым пакетам другого модуля, необязательно читать модуль, содержащий эти пакеты. Однако добавлять зависимость от модуля, содержащего раскрытые пакеты, разрешается и даже рекомендуется (если вы знаете имя модуля), чтобы система модулей могла проверить зависимости на этапах компиляции и выполнения.

Если модуль `M` раскрывает свой пакет `P` модулю `N`, то `N` вправе передать имеющийся у него рефлексивный доступ к пакету `P` другому модулю `Q`. Для этого `N` должен вызвать метод `addOpens()` класса `Module`. Делегирование рефлексивного доступа позволяет не раскрывать весь модуль всем вообще модулям, но при этом возлагает дополнительную работу на модуль, получивший рефлексивный доступ.

Использование глубокой рефлексии

В этом разделе я объясню, как раскрывать модули и пакеты для глубокой рефлексии. Начнем с простого сценария и постепенно будем усложнять его. Вот наш план.

- Продемонстрировать код, который пытается что-то сделать с помощью глубокой рефлексии. Обычно такой код приводит к ошибкам.
- Объяснить, в чем причины ошибок.
- Показать, как эти ошибки исправить.

Я буду использовать модуль `com.jdojo.reflect`, который содержит класс `Item` в пакете `com.jdojo.reflect`. В листингах 4.10 и 4.11 показан исходный код модуля и класса.

Листинг 4.10. Объявление модуля com.jdojo.reflect

```
// module-info.java
module com.jdojo.reflect {
    // Предложений модуля нет
}
```

Листинг 4.11. Класс Item с четырьмя статическими переменными

```
// Item.java
package com.jdojo.reflect;

public class Item {
    static private int s = 10;
    static int t = 20;
    static protected int u = 30;
    static public int v = 40;
}
```

Отметим, что модуль не экспортирует и не раскрывает никаких пакетов. Класс `Item` очень простой – он содержит четыре статические переменные со всеми возможными модификаторами доступа: `private`, `package`, `protected` и `public`. Я специально упростил его до предела, чтобы вы могли сосредоточиться на правилах систе-

мы модулей, а не на уяснении кода. Доступ к переменным будет осуществляться с помощью глубокой рефлексии.

Мы также будем использовать модуль `com.jdojo.reflect.test`, объявление которого приведено в листинге 4.12. Это нормальный модуль без единого предложения, т. е. он зависит только от `java.base`.

Листинг 4.12. Объявление модуля com.jdojo.reflect.test

```
// module-info.java
module com.jdojo.reflect.test {
    // Предложений модуля нет
}
```

Модуль `com.jdojo.reflect.test` содержит класс `ReflectTest`, показанный в листинге 4.13.

Листинг 4.13. Класс ReflectTest для демонстрации рефлексивного доступа к типам и их членам

```
// ReflectTest.java
package com.jdojo.reflect.test;

import java.lang.reflect.Field;
import java.lang.reflect.InaccessibleObjectException;

public class ReflectTest {
    public static void main(String[] args) throws ClassNotFoundException {
        // Получить объект Class для класса com.jdojo.reflect.Item,
        // принадлежащего модулю com.jdojo.reflect
        Class<?> cls = Class.forName("com.jdojo.reflect.Item");

        Field[] fields = cls.getDeclaredFields();
        for (Field field : fields) {
            printFieldValue(field);
        }
    }

    public static void printFieldValue(Field field) {
        String fieldName = field.getName();

        try {
            // Сделать поле доступным, на случай если его доступность ограничена
            // в объявлении, например, модификатором private
            field.setAccessible(true);

            // Напечатать значение поля
            System.out.println(fieldName + " = " + field.get(null));
```

```
        } catch (IllegalAccessException | IllegalArgumentException |
                 InaccessibleObjectException e) {
            System.out.println("Accessing " + fieldName +
                               ". Error: " + e.getMessage());
        }
    }
}
```

В методе `main()` класса `ReflectTest` вызывается метод `Class.forName()`, который загружает класс `com.jdojo.reflect.Item` и пытается напечатать значения всех четырех полей.

Мы можем загрузить класс из любого модуля методом `Class.forName()`, если только класс доступен на этапе выполнения. Необязательно, чтобы класс был указан в числе зависимостей модуля – предложениях `exports` и `requires`. Тут может возникнуть вопрос: не нарушает ли это положение о строгой инкапсуляции? Разве можно загружать класс из модуля, если этот модуль не экспортировал класс? Можно, потому что мы загружаем всего лишь дескриптор класса (объект `Class`), а это еще не означает, что мы можем создавать объекты класса и получать доступ к его членам. Строгая инкапсуляция означает, что доступны только экспортированные или раскрытые типы, а о загрузке дескрипторов класса речь вообще не идет.

При попытке выполнить класс `ReflectTest` возникает ошибка:

```
Exception in thread "main" java.lang.ClassNotFoundException: com.jdojo.reflect.Item
      at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass
      (BuiltinClassLoader.java:532)
      at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass
      (ClassLoaders.java:186)
      at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:473)
      at java.base/java.lang.Class.forName0(Native Method)
      at java.base/java.lang.Class.forName(Class.java:292)
      at com.jdojo.reflect.test/com.jdojo.reflect.test.ReflectTest.main
      (ReflectTest.java:12)
```

В сообщении говорится, что исключение `ClassNotFoundException` возбуждено при попытке загрузить класс `com.jdojo.reflect.Item`. Но ведь я только что говорил, что загрузить этот класс можно! И был прав, а ошибка связана с другой проблемой.

Модуль, содержащий загружаемый класс, должен быть известен системе модулей. До выхода JDK 9 исключение `ClassNotFoundException` означало, что класс не находится на пути к классам. Для исправления ошибки нужно было добавить каталог или JAR-файл в путь к классам. Но в JDK 9 модули ищутся на пути к модулям. Поэтому добавим модуль `com.jdojo.reflect` в путь к модулям и снова выполним `ReflectTest`. В NetBeans необходимо добавить проект `com.jdojo.reflect` в путь к модулям проекта `com.jdojo.reflect.test`, как показано на рис. 4.9.

Можно также выполнить класс `ReflectTest` следующей командой – в предположении, что в NetBeans оба проекта собраны и их модульные JAR-файлы находятся в соответственных каталогах `dist`:

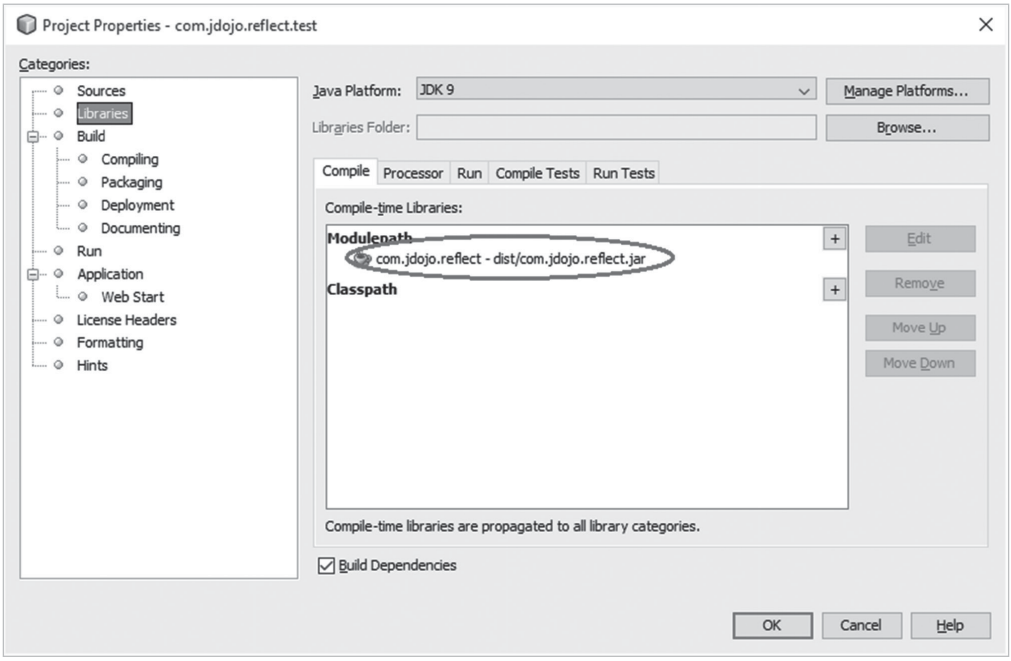

Рис. 4.9. Добавление проекта `com.jdojo.reflect` в путь к модулям проекта `com.jdojo.reflect.test` в NetBeans

```
C:\Java9Revealed>java
 --module-path com.jdojo.reflect\dist;com.jdojo.reflect.test\dist
 --module com.jdojo.reflect.test/com.jdojo.reflect.test.ReflectTest
```

При попытке выполнить `ReflectTest` в NetBeans и из командной строки снова возникает то же исключение `ClassNotFoundException`. Так что же, добавление модуля `com.jdojo.reflect` в путь к модулям не помогло? Не совсем так. Помогло, но мы решили только половину проблемы. Вторая половина связана с графом модулей.

Фраза «путь к модулям» в JDK 9 звучит очень похоже на «путь к классам», но принципы работы различаются. Путь к модулям нужен для того, чтобы находить модули в процессе разрешения модуля – когда строится и пополняется граф модулей. А путь к классам используется для нахождения класса, подлежащего загрузке. Для обеспечения надежности конфигурации система модулей проверяет наличие всех необходимых зависимостей на этапе инициализации. После того как приложение запущено, все необходимые модули разрешены и добавление модулей в путь к модулям уже ничего не может изменить. При выполнении класса `ReflectTest` при условии, что оба модуля `com.jdojo.reflect` и `com.jdojo.reflect.test` находятся на пути к модулям, граф модулей выглядит, как показано на рис. 4.10.

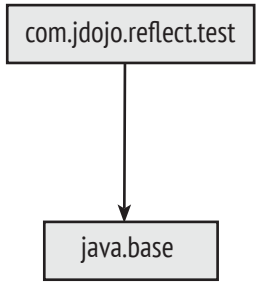

Рис. 4.10. Граф модулей при выполнении класса `ReflectTest`

Когда выполняется класс из модуля – как в случае класса `ReflectTest`, – единственным корневым модулем является модуль, содержащий главный класс. Граф модулей содер-

жит все модули, от которых зависит главный класс, а также их зависимости. В данном случае `com.jdojo.reflect.test` – единственный модуль в наборе корневых модулей по умолчанию, и система модулей ничего не знает о существовании модуля `com.jdojo.reflect`, пусть даже он находится в пути к модулям. Что нужно сделать для того, чтобы включить модуль `com.jdojo.reflect` в граф? Добавить его в набор корневых модулей с помощью параметра командной строки `--add-modules`. Значением этого параметра является список модулей через запятую:

```
--add-modules <module1>,<module2>...
```

На рис. 4.11 показано диалоговое окно свойств проекта `com.jdojo.reflect.test` с параметром VM, который необходим для добавления `com.jdojo.reflect` в набор корневых модулей.

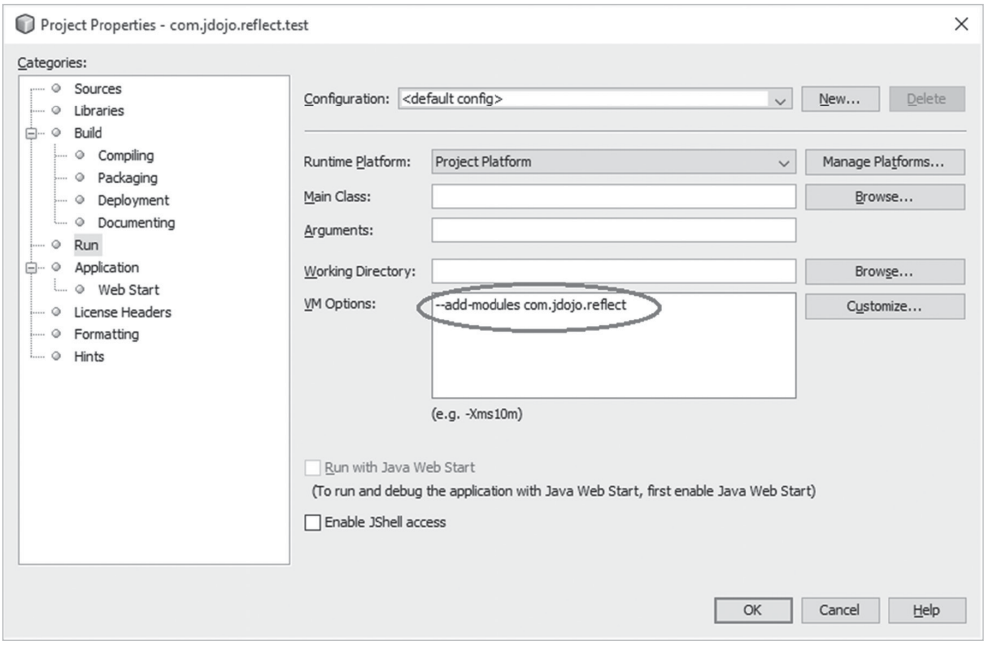

Рис. 4.11. Добавление модуля `com.jdojo.reflect` в набор корневых модулей по умолчанию

На рис. 4.12 показан граф модулей на этапе выполнения после добавления `com.jdojo.reflect` в набор корневых модулей.

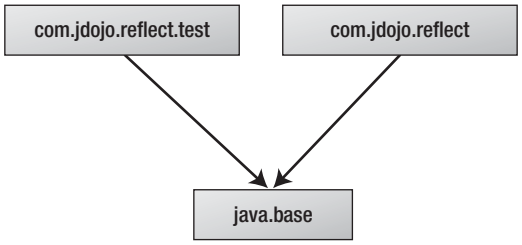

Рис. 4.12. Граф модулей после добавления `com.jdojo.reflect` в набор корневых модулей по умолчанию

Есть и другой способ разрешения модуля `com.jdojo.reflect` – добавить предложение `requires com.jdojo.reflect;` в объявление модуля `com.jdojo.reflect.test`. Тогда `com.jdojo.reflect` разрешается как зависимость модуля `com.jdojo.reflect.test`. В этом случае задавать параметр `--add-modules` не нужно.

Снова выполните класс `ReflectTest` в NetBeans или такой командой:

```
C:\Java9Revealed>java
 --module-path com.jdojo.reflect\dist;com.jdojo.reflect.test\dist
 --add-modules com.jdojo.reflect
 --module com.jdojo.reflect.test/com.jdojo.reflect.test.ReflectTest
```

```
Accessing s. Error: Unable to make field private static int com.jdojo.reflect.Item.s
accessible: module com.jdojo.reflect does not "opens com.jdojo.reflect" to module
com.jdojo.reflect.test

Accessing t. Error: Unable to make field static int com.jdojo.reflect.Item.t
accessible: module com.jdojo.reflect does not "opens com.jdojo.reflect" to module
com.jdojo.reflect.test

Accessing u. Error: Unable to make field protected static int com.jdojo.reflect.
Item.u accessible: module com.jdojo.reflect does not "opens com.jdojo.reflect" to
module com.jdojo.reflect.test

Accessing v. Error: Unable to make field public static int com.jdojo.reflect.Item.v
accessible: module com.jdojo.reflect does not "exports com.jdojo.reflect" to module
com.jdojo.reflect.test
```

Гораздо лучше. Класс `com.jdojo.reflect.Item` загрузился. Но когда программа пытается выполнить метод `setAccessible(true)` для полей, возбуждается исключение `InaccessibleObjectException`. Обратите внимание, что сообщения не одинаковы. Для полей `s`, `t` и `u` говорится, что доступ к ним запрещен, потому что модуль `com.jdojo.reflect` не раскрывает пакет `com.jdojo.reflect`. А для поля `v` в сообщении говорится, что модуль не экспортирует пакет `com.jdojo.reflect`. Дело в том, что поле `v` открытое, а остальные – неоткрытые. Для доступа к открытому полю пакет должен экспортироваться, это минимально возможная доступность. Для доступа к неоткрытым полям пакет необходимо раскрыть, это максимально возможная доступность.

В листинге 4.14 показано модифицированное объявление модуля `com.jdojo.reflect`. В нем пакет `com.jdojo.reflect` экспортируется, поэтому все открытые типы и их открытые члены доступны внешней программе.

Листинг 4.14. Модифицированное объявление модуля com.jdojo.reflect

```
// module-info.java
module com.jdojo.reflect {
    exports com.jdojo.reflect;
}
```

Перекомпилируйте оба модуля и выполните класс `ReflectTest` в NetBeans или следующей командой:

```
C:\Java9Revealed>java
 --module-path com.jdojo.reflect\dist;com.jdojo.reflect.test\dist
 --add-modules com.jdojo.reflect
 --module com.jdojo.reflect.test/com.jdojo.reflect.test.ReflectTest
```

```
Accessing s. Error: Unable to make field private static int com.jdojo.reflect.Item.s
accessible: module com.jdojo.reflect does not "opens com.jdojo.reflect" to module
com.jdojo.reflect.test

Accessing t. Error: Unable to make field static int com.jdojo.reflect.Item.t
accessible: module com.jdojo.reflect does not "opens com.jdojo.reflect" to module
com.jdojo.reflect.test

Accessing u. Error: Unable to make field protected static int com.jdojo.reflect.
Item.u accessible: module com.jdojo.reflect does not "opens com.jdojo.reflect" to
module com.jdojo.reflect.test

v = 40
```

Как и следовало ожидать, мы смогли получить доступ к значению открытого поля `v`. Но экспорт пакета дает доступ только к открытым типам и их открытым членам, а неоткрытые поля остались недоступными. Чтобы получить глубокий рефлексивный доступ к классу `Item`, мы должны раскрыть содержащий его модуль или только пакет. В листинге 4.15 показано модифицированное объявление модуля `com.jdojo.reflect`, в котором он объявлен как раскрытый.

Листинг 4.15. Модифицированное объявление модуля com.jdojo.reflect, в котором он объявлен раскрытым

```
// module-info.java
open module com.jdojo.reflect {
    // Предложений модуля нет
}
```

Перекомпилируйте оба модуля и выполните класс `ReflectTest` в NetBeans или следующей командой:

```
C:\Java9Revealed>java
 --module-path com.jdojo.reflect\dist;com.jdojo.reflect.test\dist
 --add-modules com.jdojo.reflect
 --module com.jdojo.reflect.test/com.jdojo.reflect.test.ReflectTest
```

```
s = 10
t = 20
```

```
u = 30
v = 40
```

Как видим, мы получили доступ ко всем полям класса `Item` из модуля `com.jdojo.reflect.test`. Результат будет таким же, если раскрыть не весь модуль, а только пакет `com.jdojo.reflect`, как показано в листинге 4.6. Перекомпилировав оба модуля и выполнив класс `ReflectTest`, мы получим тот же результат, что и выше.

Листинг 4.16. Модифицированное объявление модуля com.jdojo.reflect, в котором объявлен раскрытым только пакет com.jdojo.reflect

```
// module-info.java
module com.jdojo.reflect {
    opens com.jdojo.reflect;
}
```

Мы почти закончили с этим примером. Осталось сделать несколько замечаний.

- Раскрытый модуль или модуль с раскрытыми пакетами дает глубокий рефлексивный доступ ко всем типам и их членам другим модулям, которые для этого не обязаны объявлять зависимость от модуля-владельца. В нашем примере модуль `com.jdojo.reflect.test` смог получить доступ к классу `Item` и его членам, не объявляя зависимость от модуля `com.jdojo.reflect`. Это правило введено для того, чтобы не вынуждать такие каркасы, как Hibernate и Spring, в которых применяется глубокая рефлексия, объявлять зависимости от модулей приложения.
- Если вам нужен доступ к открытому API пакета на этапе компиляции и глубокий рефлексивный доступ к тому же пакету на этапе выполнения, то можете одновременно раскрыть его и экспортировать. В нашем примере экспортирован и раскрыт пакет `com.jdojo.reflect` в модуле `com.jdojo.reflect`.
- Если модуль или некоторые его пакеты раскрыты, то объявлять зависимость от них можно, но необязательно. Это правило помогает перейти на JDK 9. Если модуль применяет глубокую рефлексию для доступа к другим известным модулям, то он должен объявить зависимость от них, чтобы получить все выгоды от надежной конфигурации.

Теперь приведем окончательные варианты модулей.

Листинг 4.17. Модифицированное объявление модуля com.jdojo.reflect, в котором экспортируется и раскрывается пакет com.jdojo.reflect

```
// module-info.java
module com.jdojo.reflect {
    exports com.jdojo.reflect;
    opens com.jdojo.reflect;
}
```

Листинг 4.18. Модифицированное объявление модуля com.jdojo.reflect.test, в котором читается модуль com.jdojo.reflect

```
// module-info.java
module com.jdojo.reflect.test {
    requires com.jdojo.reflect;
}
```

Теперь для выполнения класса `ReflectTest` необязательно задавать параметр VM `--add-modules`. Модуль `com.jdojo.reflect` разрешается благодаря наличию предложения `requires com.jdojo.reflect;` в объявлении модуля `com.jdojo.reflect.test`. На рис. 4.13 показан граф модулей, создаваемый при выполнении класса `ReflectTest`.

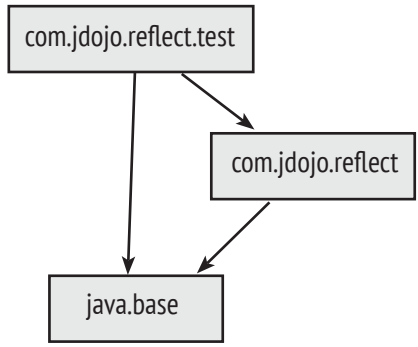

Рис. 4.13. Граф модулей для окончательной версии модулей `com.jdojo.reflect` и `com.jdojo.reflect.test`

Пересоберите оба проекта в NetBeans и выполните класс в NetBeans или из командной строки. Результат будет такой же, как и раньше.

```
C:\Java9Revealed>java
 --module-path com.jdojo.reflect\dist;com.jdojo.reflect.test\dist
 --module com.jdojo.reflect.test/com.jdojo.reflect.test.ReflectTest
```

```
s = 10
t = 20
u = 30
v = 40
```

Вы подумали о ситуациях, когда вашему модулю нужен глубокий рефлексивный доступ к другому модулю, который не раскрыл ни один из своих пакетов? Это возможно с помощью параметра `--add-opens`. Этот и многие другие параметры мы рассмотрим в главе 9 при описании различных способов нарушения инкапсуляции модулей, поддерживаемых JDK 9.

Доступность типов

До JDK 9 существовало четыре вида доступности:

- `public`
- `protected`
- `<package>`
- `private`

В JDK 8 слово `public` означало, что тип доступен из любой части программы. В JDK 9 ситуация изменилась. Открытый тип необязательно открыт для всех. Открытый тип, определенный внутри модуля, может относиться к одной из трех категорий:

- открыт только внутри модуля, в котором определен;
- открыт только для конкретных модулей;
- открыт для всех.

Если тип определен в модуле как `public`, но модуль не экспортирует содержащий его пакет, то тип будет открытым только внутри самого модуля, а другие модули не смогут получить к нему доступ.

Если тип определен в модуле как `public`, но модуль *квалифицированно* экспортирует содержащий его пакет, то тип будет доступен только модулям, указанным в предложении экспорта.

Если тип определен в модуле как `public`, и модуль экспортирует содержащий его пакет без квалификации, то тип будет открыт любому модулю, читающему модуль-владелец.

Расщепление пакетов между несколькими модулями

Расщеплять пакет между несколькими модулями *запрещено*. Это означает, что один и тот же пакет не может быть определен в нескольких модулях. Необходимо либо объединить модули в один, либо переименовать пакет в одном из модулей. Иногда подобные модули компилируются успешно, а ошибка возникает только во время выполнения, а иногда – уже на этапе компиляции. Расщепление пакетов не запрещено безусловно, как я сказал в начале раздела. Необходимо понимать простые правила, нарушение которых влечет ошибки.

Если в двух модулях `M` и `N` определен один и тот же пакет `P`, то не должно существовать модуля `Q` такого, что пакет `P` из `M` и из `N` доступен `Q`. Иначе говоря, один и тот же пакет в двух разных модулях не должен одновременно читаться третьим модулем. Если некий модуль пользуется пакетом, обнаруженным в двух разных модулях, то система модулей не принимает за вас решение, т. к. оно может оказаться неправильным. Она выдает ошибку и предоставляет вам решить проблему. Рассмотрим следующий фрагмент кода:

```
// Test.java
package java.util;

public class Test {
}
```

При попытке откомпилировать класс `Test` в JDK 9 как часть модуля или автономно возникнет следующая ошибка:

```
error: package exists in another module: java.base
package java.util;
^
1 error
```

Если этот класс находится в модуле `M`, то сообщение говорит, что пакет `java.util` встречается как в этом модуле, так и в модуле `java.base`. Вы должны изменить имя

пакета в этом модуле, так чтобы оно не совпадало ни с одним пакетом в других видимых модулях.

Ограничения в объявлениях модулей

На объявление модуля накладывается несколько ограничений. Их нарушение влечет ошибки на этапе компиляции или во время инициализации приложения.

- Граф модулей не может содержать циклических зависимостей, т. е. никакие два модуля не могут читать друг друга. Если такое происходит, то это должен быть один модуль, а не два. Отметим, что на этапе выполнения циклические зависимости возможны вследствие добавления ребер чтения – программно или с помощью параметров командной строки.
- В объявлениях модулей не поддерживаются номера версий. Версия добавляется в качестве атрибута файла класса программой `jar` или какой-то другой, например `javac`.
- В системе нет понятия подмодуля, т. е. из двух модулей `com.jdojo.person` и `com.jdojo.person.client` второй не является подмодулем первого.

Типы модулей

Язык Java существует уже больше 20 лет, и приложения – как старые, так и новые – по-прежнему будут использовать библиотеки, неважно, преобразованы они в модульную форму или нет. Если бы все разработчики были вынуждены преобразовать свои приложения в модульную форму, то JDK 9, скорее всего, не получил бы широкого распространения. Проектировщики JDK 9 всегда помнили об обратной совместимости. Вы можете переводить свое приложение на модули постепенно или не переводить вовсе, а просто запускать его в JDK 9, как будто ничего не произошло. Как правило, приложения, работавшие в JDK 8 или более ранних версиях, будут работать и в JDK 9 без каких-либо изменений. Чтобы упростить миграцию, в JDK 9 определено четыре типа модулей:

- нормальные;
- раскрытые;
- автоматические;
- безымянные.

На самом деле, при описании модулей встречается шесть терминов, но начинающего изучать JDK 9 это будет только смущать без надобности. Оставшиеся два типы служат для обозначения объемлющих категорий. На рис. 4.14 показаны все типы модулей.

Прежде чем переходить к описанию основных типов модулей, дам краткие определения.

- Модулем называется совокупность кода и данных.
- В зависимости от наличия имени модули подразделяются на *именованные* и *безымянные*.
- У безымянных модулей дальнейшего деления на типы нет.

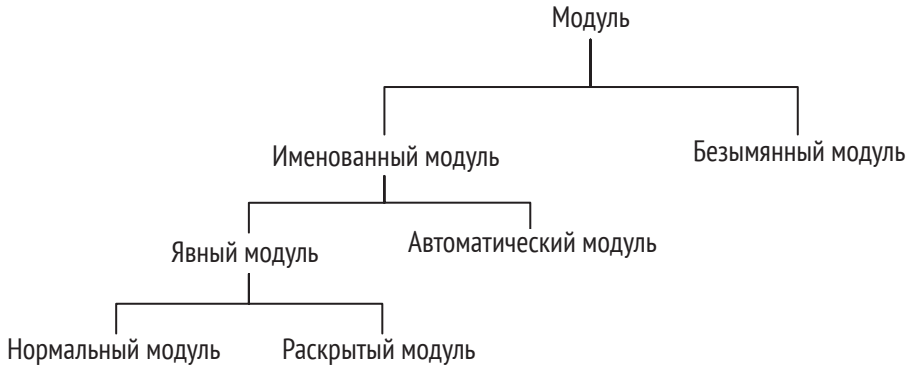

Рис. 4.14. Типы модулей

- Имя модуля может быть задано явно в его объявлении или сгенерировано автоматически (неявно). Если имя задано в объявлении, то модуль называется *явным*. Если же имя генерируется системой по имени JAR-файла на пути к модулям, то модуль называется *автоматическим*.
- Если в объявлении модуля отсутствует модификатор `open`, то модуль называется *нормальным*, а если присутствует, то *раскрытым*.

Таким образом, раскрытый модуль является также явным и именованным. Автоматический модуль является именованным, потому что имеет автоматически сгенерированное имя, но не является явным, потому что неявно объявляется системой модулей на этапах компиляции и выполнения. Далее типы модулей описываются более подробно.

> **Совет.** Если бы платформа Java изначально проектировалась с системой модулей, то тип модулей был бы всего один – нормальный! Все остальные типы введены только для обеспечения обратной совместимости и облегчения перехода на JDK 9.

Нормальные модули

Модуль, для которого существует явное объявление без модификатора `open`, всегда получает имя и называется *нормальным модулем*, или просто *модулем*. До сих пор мы имели дело преимущественно с нормальными модулями. Я называл их просто модулями и буду придерживаться этой привычки, если только не возникнет необходимость различать типы. По умолчанию все типы, определенные в нормальном модуле, инкапсулированы. Вот пример нормального модуля:

```
module a.normal.module {
    // Здесь могут находиться предложения модуля
}
```

Раскрытые модули

Если в объявлении модуля присутствует модификатор `open`, то модуль называется раскрытым, например:

```
open module a.open.module {
    // Здесь могут находиться предложения модуля
}
```

Автоматические модули

Для обеспечения обратной совместимости механизм пути к классам работает и в JDK 9. JAR-файлы можно помещать на путь к классам, на путь к модулям или туда и сюда. Отметим, что и модульные JAR-файлы можно помещать на путь к модулям и на путь к классам.

Если обычный JAR-файл помещен на путь к модулям, то он считается модулем и называется *автоматическим модулем*. Автоматический он потому, что модуль автоматически определяется на основе JAR-файла, в котором нет объявления в файле `module-info.class`. У автоматического модуля есть имя. Но какое? И что этот модуль читает, какие пакеты он экспортирует? Ниже приводятся ответы на эти вопросы.

Автоматический модуль является именованным. Его имя и версия выводятся из имени JAR-файла по следующим правилам.

- Расширение `.jar` удаляется. Так, из имени `com.jdojo.intro-1.0.jar` получается `com.jdojo.intro-1.0`, и по этой строке определяются имя и версия модуля.
- Если в конце строки находится дефис, за которым следует хотя бы одна цифра и, возможно, точка, то в качестве имени модуля берется часть строки, предшествующая последнему дефису. Часть, следующая за дефисом, становится версией модуля, если ее можно разобрать как номер версии. В нашем случае именем модуля будет `com.jdojo.intro`, а версией – `1.0`.
- Все символы имени, отличные от букв и цифр, заменяются точками и в получившейся строке несколько идущих подряд точек заменяются одной. Начальные и конечные точки удаляются. В нашем примере имя не содержит символов, отличных от букв и цифр, поэтому остается равным `com.jdojo.intro`.

Последовательное применение этих правил дает имя и версию модуля. В конце раздела я покажу, как определить имя автоматического модуля с помощью JAR-файла. В табл. 4.1 приведено несколько имен JAR-файлов (без расширения `.jar`), а также выведенные по ним имена и версии автоматических модулей.

Таблица 4.1. Примеры вывода имен и версий автоматических модулей

Имя JAR-файла	Имя модуля	Версия модуля
`com.jdojo.intro-1.0`	`com.jdojo.intro`	`1.0`
`junit-4.10`	`junit`	`4.10`
`jdojo-logging1.5.0`	ошибка	нет версии
`spring-core-4.0.1.RELEASE`	`spring.core`	`4.0.1.RELEASE`
`jdojo-trans-api_1.5_spec-1.0.0`	ошибка	`1.0.0`
–	ошибка	нет версии

Рассмотрим три случая, когда вместо имени модуля мы получаем ошибку. Первый из них – файл с именем `jdojo-logging1.5.0`. Применим к нему правила автоматического вывода.

- В имени JAR-файла нет дефиса, за которым следовала бы цифра, поэтому номер версии отсутствует. Вся строка целиком используется для вывода имени модуля.
- Все символы, отличные от букв и цифр, заменяются точками – получается строка `jdojo.logging1.5.0`. Напомним, что каждая часть имени модуля должна быть допустимым идентификатором Java. В данном случае части 5 и 0 таковыми не являются, поэтому выведенное имя модуля недопустимо, и мы получаем ошибку

Второе имя JAR-файла, для которого выдается ошибка, — `jdojo-trans-api_1.5_spec-1.0.0`. Применим к нему правила вывода.

- Система находит последний дефис, за которым следуют только цифры и точки, и разбивает имя JAR-файла на две части: `jdojo-trans-api_1.5_spec` и `1.0.0`. Первая часть используется для вывода имени модуля, а вторая является номером версии.
- Все символы, отличные от букв и цифр, заменяются точками – получается строка `jdojo.trans.api.1.5.spec`, которая не может быть именем модуля, потому что 1 и 5 – недопустимые идентификаторы Java.

И в последней строке таблицы в качестве имени JAR-файла фигурирует знак подчеркивания. Если применить к нему правила вывода, то подчерк будет заменен точкой, а точка удалена, потому что это единственный символ в строке. В результате получается пустая строка, не являющаяся допустимым именем модуля.

Если на пути к модулям встретится обычный JAR-файл, по которому нельзя вывести имя автоматического модуля, то система возбудит исключение. Например, наличие файла `_.jar` на пути к модулям приводит к такому исключению:

```
java.lang.module.ResolutionException: Unable to derive module descriptor for: _.jar
```

Команда `jar` с параметром `--describe-module` печатает дескриптор модуля для модульного JAR-файла и выведенное имя автоматического модуля для обычного JAR-файла. В последнем случае печатается также список содержащихся в файле пакетов. Синтаксис команды следующий:

```
jar --describe-module --file <path-to-JAR>
```

Показанная ниже команда печатает имя автоматического модуля для обычного JAR-файла `cglib-2.2.2.jar`:

```
C:\Java9Revealed>jar --describe-module --file lib\cglib-2.2.2.jar
```

```
No module descriptor found. Derived automatic module.
module cglib@2.2.2 (automatic)
  requires mandated java.base
  contains net.sf.cglib.beans
  contains net.sf.cglib.core
```

```
  contains net.sf.cglib.proxy
  contains net.sf.cglib.reflect
  contains net.sf.cglib.transform
  contains net.sf.cglib.transform.impl
  contains net.sf.cglib.util
```

Эта команда печатает сообщение, если не смогла найти дескриптор модуля в JAR-файле и вывела по нему автоматический модуль. Если имя JAR-файла не может быть преобразовано в имя автоматического модуля (например, `cglib.1-2.2.2.jar`), то команда печатает сообщение об ошибке, в котором говорится, почему именно не удалось вывести имя модуля, например:

```
C:\Java9Revealed>jar --describe-module --file lib\cglib.1-2.2.2.jar
```

```
Unable to derive module descriptor for: lib\cglib.1-2.2.2.jar
cglib.1: Invalid module name: '1' is not a Java identifier
```

Если имя автоматического модуля известно, то другие явные модули могут прочитать его с помощью предложения `requires`. В следующем объявлении читается автоматический модуль с именем `cglib`, образованным из имени файла `cglib-2.2.2.jar` на пути к модулям:

```
module com.jdojo.lib {
requires cglib;
    //...
}
```

Чтобы от автоматического модуля была польза, он должен экспортировать пакеты и читать другие модули. Перечислим соответствующие правила.

- Автоматический модуль читает все остальные модули. Важно отметить, что ребра чтения, исходящие из автоматического модуля к прочим, добавляются после разрешения графа модулей.
- Все пакеты, содержащиеся в автоматическом модуле, экспортированы и раскрыты.

Эти правила объясняются тем, что не существует никакого разумного способа определить, от каких модулей зависит автоматический модуль и к каким пакетам автоматического модуля другим модулям может понадобиться глубокий рефлексивный доступ.

Автоматический модуль, читающий все остальные модули, может создавать циклические зависимости, наличие которых допускается после разрешения графа модулей. Напомним, что в процессе разрешения графа модулей циклические зависимости запрещены, т. е. их не должно быть в объявлениях модулей.

У автоматических модулей нет объявления, так что они не могут объявлять зависимости от других модулей. Явные модули могут объявлять зависимости от автоматических. Предположим, что явный модуль `M` читает автоматический мо-

дуль P, который, в свою очередь, использует тип T из другого автоматического модуля Q. При выполнении главного класса, находящегося в модуле M, граф модулей состоит только из M и P — java.base мы для краткости исключаем из рассмотрения. Процесс разрешения начинается с модуля M и выясняется, что он читает модуль P. Но понять, что P читает модуль Q, процесс разрешения не в состоянии. Мы можем откомпилировать оба модуля P и Q и поместить их на путь к классам, но при запуске приложения получим исключение ClassNotFoundException. Оно возникает, когда модуль P пытается обратиться к типу из модуля Q. Для решения этой проблемы Q должен быть включен в граф модулей, для чего нужно добавить его в качестве корневого модуля с помощью параметра --add-modules.

Следующая команда описывает автоматический модуль cglib, объявление которого выведено из файла cglib-2.2.2.jar, помещенного на путь к классам. Из описания видно, что этот модуль экспортирует и раскрывает все свои пакеты.

```
C:\Java9Revealed>java --module-path lib\cglib-2.2.2.jar --list-modules cglib
```

```
automatic module cglib@2.2.2 (file:///C:/Java9Revealed/lib/cglib-2.2.2.jar)
  exports net.sf.cglib.beans
  exports net.sf.cglib.core
  exports net.sf.cglib.proxy
  exports net.sf.cglib.reflect
  exports net.sf.cglib.transform
  exports net.sf.cglib.transform.impl
  exports net.sf.cglib.util
  requires mandated java.base
  opens net.sf.cglib.transform
  opens net.sf.cglib.transform.impl
  opens net.sf.cglib.beans
  opens net.sf.cglib.util
  opens net.sf.cglib.reflect
  opens net.sf.cglib.core
  opens net.sf.cglib.proxy
```

Безымянные модули

Обычные и модульные JAR-файлы можно помещать на путь к классам. Если при попытке загрузить тип содержащий его пакет не найден ни в одном из известных модулей, то система пробует найти тип на пути к классам. Если это получается, то тип становится частью *безымянного модуля*, принадлежащего тому загрузчику классов, который его загрузил. В каждом загрузчике классов определен безымянный модуль, членами которого являются все типы, загруженные им из пути к классам. Поскольку у безымянного модуля нет имени, никакой явный модуль не может объявить зависимость от него в предложении requires. Если какому-то явному модулю необходимы типы, находящиеся в безымянном модуле, то придется использовать JAR-файл безымянного модуля как автоматического, т. е. поместить его на путь к модулям.

Типичная ошибка – попытка обращаться к типам в безымянном модуле из явных модулей на этапе компиляции. Это попросту невозможно, потому что для чтения другого модуля на этапе компиляции явный модуль должен сослаться на него по имени, а у безымянного модуля имени нет. Автоматические модули играют роль моста между явными и безымянными модулями, как показано на рис. 4.15. Явные модули могут обращаться к автоматическим с помощью предложений `requires`, а автоматические могут получить доступ к безымянным.

Рис. 4.15. Автоматический модуль в роли моста между явным и безымянным модулем

У безымянного модуля нет имени. И это не означает, что его имя – пустая строка, «unnamed» или `null`. Следующее объявление модуля недопустимо:

```
module some.module {
    requires "";         // ошибка на этапе компиляции
    requires "unnamed";  // ошибка на этапе компиляции
    requires unnamed;    // ошибка на этапе компиляции, если только не существует
                         // модуля с именем unnamed
    requires null;       // ошибка на этапе компиляции
}
```

Безымянный модуль читает все остальные модули и экспортирует и раскрывает все свои пакеты в соответствии со следующими правилами.

- Безымянный модуль читает все остальные модули. Поэтому он имеет доступ к открытым типам из любого пакета, экспортированного любым модулем, в т. ч. платформенным. Благодаря этому правилу приложение, пользовавшееся путем к классам и работавшее в Java SE 8, будет компилироваться и работать также в Java SE 9 при условии, что в нем используются только стандартные и не вышедшие из употребления API Java SE.
- Безымянный модуль раскрывает все свои пакеты всем модулям. Поэтому любой явный модуль может получить доступ к типам, находящимся в безымянном модуле, применяя рефлексию во время выполнения.
- Безымянный модуль экспортирует все свои пакеты. Явный модуль не может прочитать безымянный модуль на этапе компиляции. Но после построения графа модулей всем автоматическим модулям разрешено читать безымянные.

Совет. Безымянный модуль может содержать пакет, экспортированный также именованным модулем. В таком случае пакет из безымянного модуля игнорируется.

Рассмотрим два примера использования безымянных модулей. В первом примере нормальный модуль получает доступ к безымянному с помощью рефлексии. Напомним, что на этапе компиляции нормальные модули не могут обращаться к безымянному. Во втором примере безымянный модуль обращается к нормальному.

Обращение из нормального модуля к безымянному

Безымянный модуль не объявляется. Чтобы получить безымянный модуль, нужно поместить обычный или модульный JAR-файл на путь к классам. Мы поступим так с модулем `com.jdojo.reflect`, превратив его в безымянный.

В листингах 4.19 и 4.20 приведены объявление модуля `com.jdojo.unnamed.test` и код класса `Main` в нем соответственно. В методе `main()` этот класс пытается загрузить класс `com.jdojo.reflect.Item` и прочитать его поля.

Листинг 4.19. Объявление модуля com.jdojo.unnamed.test

```
// module-info.com
module com.jdojo.unnamed.test {
    // Предложений модуля нет
}
```

Листинг 4.20. Класс Main в модуле com.jdojo.unnamed.test

```
// Main.java
package com.jdojo.unnamed.test;

import java.lang.reflect.Field;

public class Main {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("com.jdojo.reflect.Item");

        Field[] fields = cls.getDeclaredFields();
        for (Field field : fields) {
            field.setAccessible(true);
            System.out.println(field.getName() + " = " + field.get(null));
        }
    }
}
```

В NetBeans добавьте проект `com.jdojo.reflect` в путь к классам в проекте `com.jdojo.unnamed.test`, как показано на рис. 4.16.

Чтобы выполнить класс `Main`, воспользуйтесь NetBeans или следующей командой (предварительно не забудьте собрать оба проекта – `com.jdojo.reflect` и `com.jdojo.unnamed.test`).

```
C:\Java9Revealed>java --module-path com.jdojo.unnamed.test\dist
 --class-path com.jdojo.reflect\dist\com.jdojo.reflect.jar
 --module com.jdojo.unnamed.test/com.jdojo.unnamed.test.Main
```

```
s = 10
t = 20
u = 30
v = 40
```

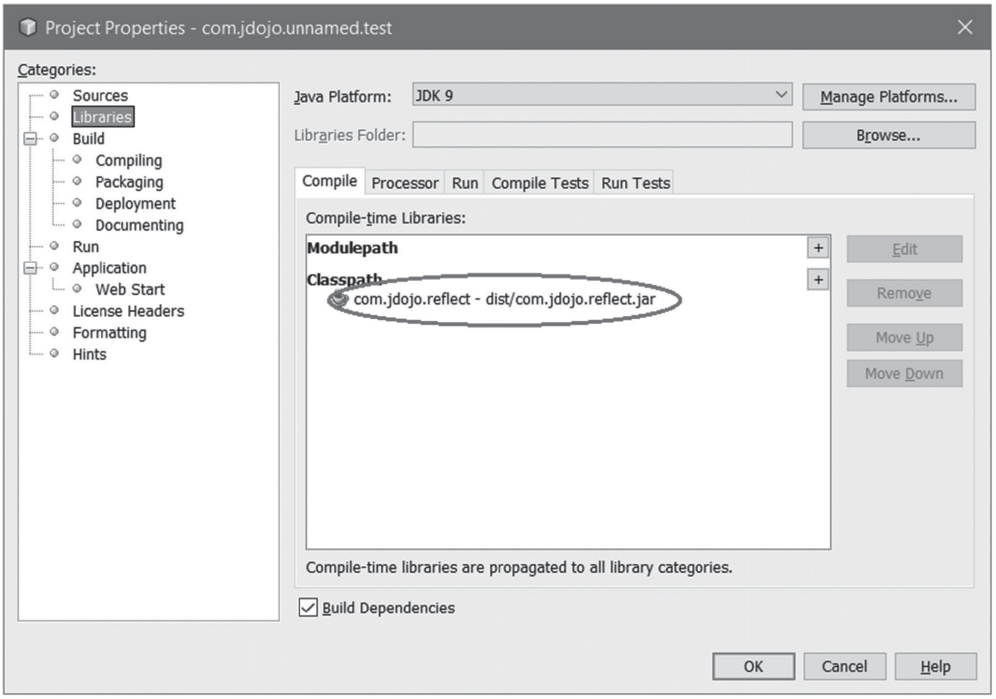

Рис. 4.16. Добавление проекта `com.jdojo.reflect` в путь к классам в проекте `com.jdojo.unnamed.test`

Поскольку файл `com.jdojo.reflect.jar` находится на пути к классам, содержащийся в нем класс `Item` будет загружен в безымянный модуль загрузчика классов. Из распечатки следует, что мы успешно получили доступ к классу `Item` в безымянном модуле, применив глубокую рефлексию к именованному модулю `com.jdojo.unnamed.test`. Но попытавшись обратиться к классу `Item` на этапе компиляции, мы получим ошибку, потому что в объявлении модуля `com.jdojo.unnamed.test` не может быть предложения `requires`, читающего безымянный модуль.

Обращение из безымянного модуля к нормальному

В этом разделе мы увидим, как обращаться к типам, находящимся в именованном модуле, из безымянного модуля. Создайте в NetBeans проект `com.jdojo.unnamed`. Это будет немодульный проект, т. к. в нем нет файла `module-info.java`, содержащего объявление модуля. Подобные проекты вы создавали в JDK 8. Добавьте в проект класс `Main`, показанный в листинге 4.21. В нем используется класс `Item` из пакета `com.jdojo.reflect`, входящего в существующий проект `com.jdojo.reflect`, где имеется модуль.

Листинг 4.21. Класс Main в проекте com.jdojo.unnamed

```
// Main.java
package com.jdojo.unnamed;
```

```
import com.jdojo.reflect.Item;

public class Main {
    public static void main(String[] args) {
        int v = Item.v;
        System.out.println("Item.v = " + v);
    }
}
```

Главный класс не компилируется, поскольку не знает, где находится класс `Item`. Добавьте проект `com.jdojo.reflect` на путь к модулям в проекте `com.jdojo.unnamed`, как показано на рис. 4.17.

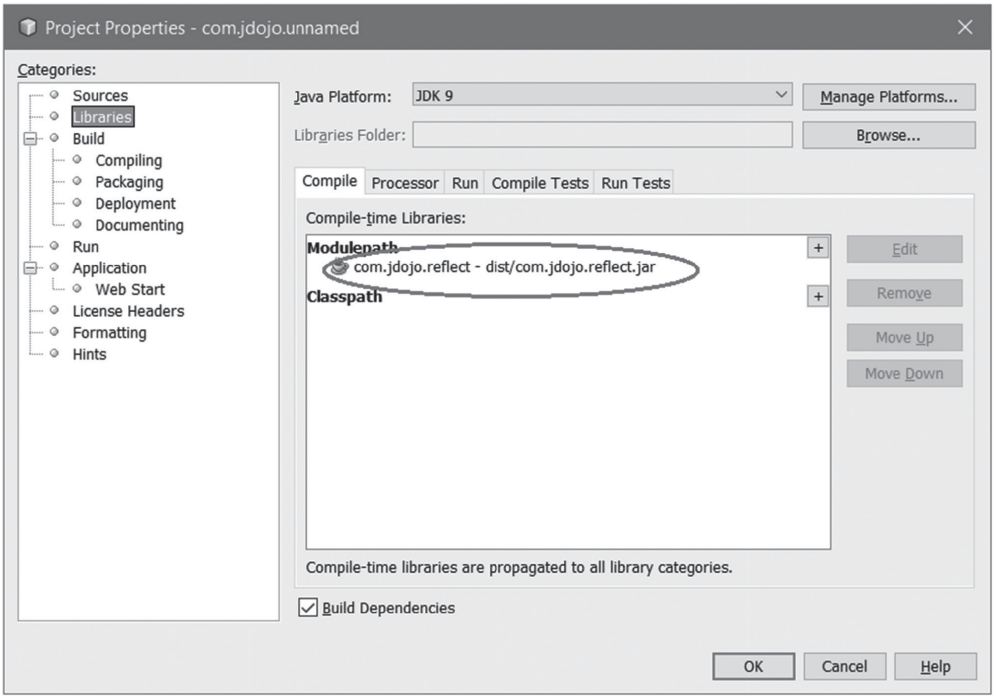

Рис. 4.17. Добавление проекта `com.jdojo.reflect` на путь к модулям в проекте `com.jdojo.unnamed`

При попытке откомпилировать класс `com.jdojo.unnamed.Main` возникает ошибка:

```
C:\Java9Revealed\com.jdojo.unnamed\src\com\jdojo\unnamed\Main.java:4: error: package
com.jdojo.reflect is not visible
import com.jdojo.reflect.Item;
  (package com.jdojo.reflect is declared in module com.jdojo.reflect,
  which is not in the module graph)
1 error
```

В сообщении говорится, что класс `Main` не может импортировать пакет `com.jdojo.reflect`, поскольку тот невидим. В скобках приведена точная причина и рекоменда-

ция по исправлению ошибки. Вы добавили модуль `com.jdojo.reflect` в путь к модулям. Однако этот модуль не добавился в граф модулей, потому что никакие другие модули от него не зависят. Чтобы исправить ошибку, добавьте модуль `com.jdojo.reflect` в набор корневых модулей с помощью параметра компилятора `--add-modules`, как показано на рис. 4.18. Теперь класс `com.jdojo.unnamed.Main` успешно компилируется.

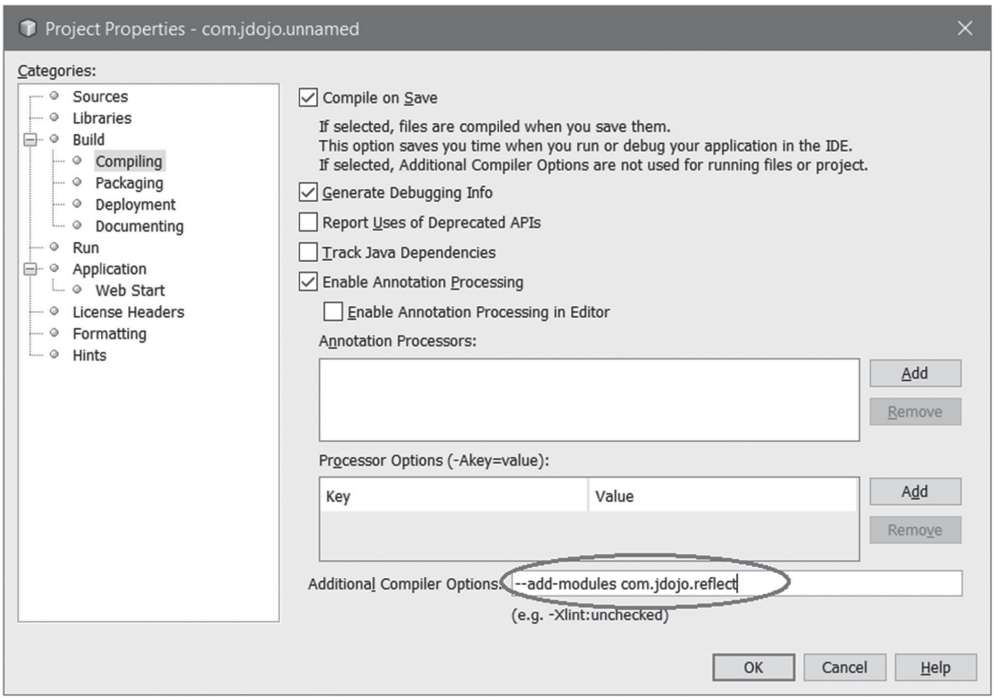

Рис. 4.18. Добавление модуля `com.jdojo.reflect` в набор корневых модулей в проекте `com.jdojo.unnamed` на этапе компиляции

Попробуйте выполнить класс `com.jdojo.unnamed.Main` в NetBeans или следующей командой:

```
C:\Java9Revealed>java --module-path com.jdojo.reflect\dist
 --class-path com.jdojo.unnamed\dist\com.jdojo.unnamed.jar
 com.jdojo.unnamed.Main
```

```
Exception in thread "main" java.lang.NoClassDefFoundError: com/jdojo/reflect/Item
        at com.jdojo.unnamed.Main.main(Main.java:8)
Caused by: java.lang.ClassNotFoundException: com.jdojo.reflect.Item
        at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass
        (BuiltinClassLoader.java:532)
        at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass
        (ClassLoaders.java:186)
        at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:473)
        ... 1 more
```

Сообщение говорит, что класс `com.jdojo.reflect.Item` не найден. На этот раз причина ошибки не так очевидна, как при первой попытке откомпилировать класс. Тем не менее она та же самая – модуль `com.jdojo.reflect` не включен в граф модулей на этапе выполнения. Для исправления нужно задать тот же параметр `--add-modules`, но уже для VM. На рис. 4.19 показано, как это делается в NetBeans.

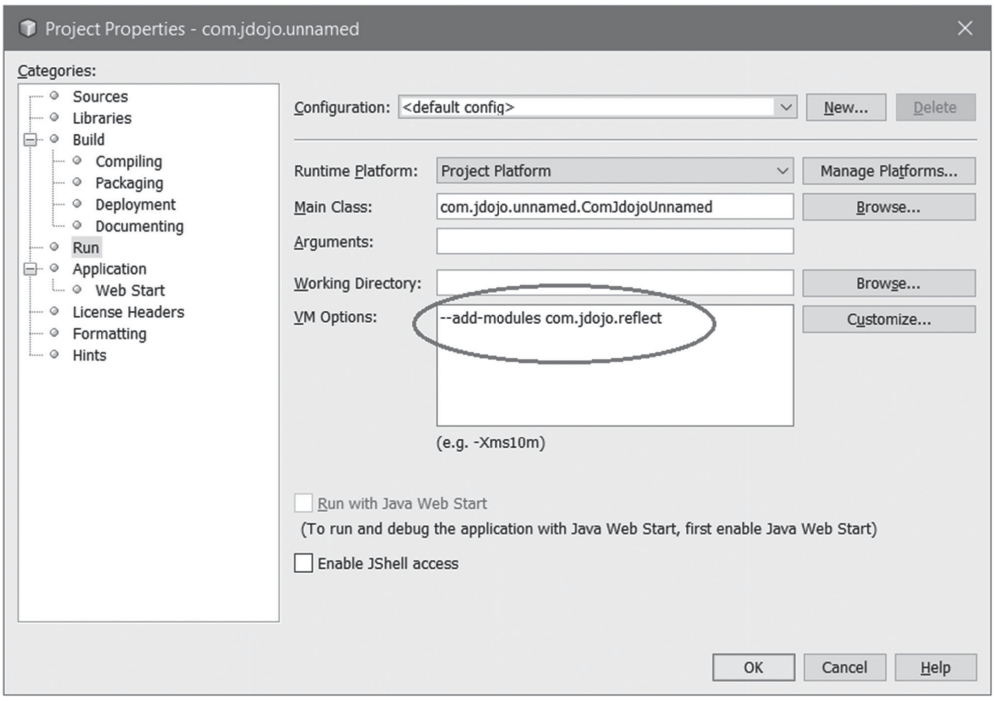

Рис. 4.19. Добавление модуля `com.jdojo.reflect` в набор корневых модулей в проекте `com.jdojo.unnamed` на этапе выполнения для VM

Снова выполните класс `com.jdojo.unnamed.Main` в NetBeans или следующей командой:

```
C:\Java9Revealed>java --module-path com.jdojo.reflect\dist
 --add-modules com.jdojo.reflect
 --class-path com.jdojo.unnamed\dist\com.jdojo.unnamed.jar
com.jdojo.unnamed.Main
```

```
Item.v = 40
```

Как видим, безымянный модуль смог получить доступ к открытым типам и их открытым членам, находящимся в экспортированном пакете именованного модуля. Отметим, что получить доступ к другим статическим полям (`s`, `t` и `u`) класса `Item` из класса `com.jdojo.unnamed.Main` не удастся, потому что они неоткрытые.

Порядок перехода на JDK 9

При переносе приложения на JDK 9 следует помнить о двух преимуществах системы модулей: строгой инкапсуляции и надежной конфигурации. Ваша цель – получить приложение, состоящее из нормальных модулей и, быть может, нескольких раскрытых модулей. Если вы надеетесь, что кто-нибудь снабдит вас четким списком шагов по переносу приложений на JDK 9, то вынужден вас разочаровать: это невозможно из-за огромного разнообразия приложений, их взаимозависимостей и различия требуемых конфигураций. Я могу лишь дать общие рекомендации, которые помогут спланировать процесс миграции.

Раньше нетривиальное Java-приложение состояло из нескольких JAR-файлов, принадлежащих трем уровням:

- уровень приложения – этот код создается разработчиками приложения;
- библиотечный уровень – код, написанный сторонними разработчиками;
- уровень JVM – среда выполнения Java.

В JDK 9 среда выполнения Java уже преобразована к модульной форме, т. е. состоит из модулей и только из них.

Библиотечный уровень состоит в основном из сторонних JAR-файлов, помещенных на путь к классам. И возможно, что их модульных версий пока не существует. Кроме того, вы контролируете, как разработчики сторонних библиотек преобразуют их в модули. Вы можете поместить библиотечные JAR-файлы на путь к модулям и работать с ними, как с автоматическими модулями.

Есть несколько подходов к преобразованию приложения в модульную форму. Ниже перечислены возможные типы модулей от наименее к наиболее желательному:

- безымянные;
- автоматические;
- раскрытые;
- нормальные.

Первый шаг миграции – проверить, работает ли приложение в среде JDK 9. Для этого нужно поместить все JAR-файлы – прикладные и библиотечные – на путь к классам, не внося никаких модификаций в код. Все типы, определенные в JAR-файлах на пути к модулям, станут частью безымянных модулей. В этом состоянии приложение пользуется JDK 9, но не получает ни строгой инкапсуляции, ни надежной конфигурации.

После того как доказано, что приложение вообще работает в JDK 9, можно начать преобразовывать код в автоматические модули. Все пакеты автоматического модуля раскрыты для рефлексивного доступа и экспортируются, так что на этапах компиляции и выполнения все открытые типы доступны как обычно. В этом отношении никакого выигрыша по сравнению с безымянными модулями нет, т. к. строгая инкапсуляция отсутствует. Однако автоматические модули обеспечивают надежную конфигурацию, поскольку другие модули могут объявлять зависимость от них.

Другой вариант – преобразовать код приложения в набор раскрытых модулей, что дает слабую форму строгой инкапсуляции: в раскрытых модулях все пакеты

раскрыты для глубокого рефлексивного доступа, но вы можете указать, какие пакеты экспортировать для доступа на этапах компиляции и выполнения. Кроме того, явные модули могут объявлять зависимость от раскрытых, так что налицо преимущества надежной конфигурации.

Нормальный модуль предлагает самую строгую инкапсуляцию, т. е. вы можете решать, какие пакеты раскрывать, экспортировать или то и другое вместе. Надежная конфигурация также присутствует, поскольку явные модули могут объявлять зависимость от нормальных.

В табл. 4.2 показано, какую степень строгой инкапсуляции и надежной конфигурации предоставляют модули разных типов.

Таблица 4.2. Типы модулей и предлагаемая ими степень строгой инкапсуляции и надежной конфигурации

Тип модуля	Строгая инкапсуляция	Надежная конфигурация
Безымянный	Нет	Нет
Автоматический	Нет	Слабая
Раскрытый	Слабая	Да
Нормальный	Сильная	Сильная

Дизассемблирование определения модуля

В этом разделе я расскажу о входящей в состав JDK команде `javap`, которая предназначена для дизассемблирования классов.

Этот инструмент очень полезен для изучения системы модулей, а особенно для декомпиляции дескрипторов модулей.

Сейчас у нас имеется два варианта файла `module-info.class` для модуля `com.jdojo.intro`: один в каталоге `mods\com.jdojo.intro`, другой – в модульном JAR-файле `lib\com.jdojo.intro-1.0.jar`. В момент упаковки кода модуля в JAR-файл мы задавали версию и главный класс модуля. И куда эта информация записалась? В файл `module-info.class` в виде атрибутов класса. Поэтому содержимое двух файлов `module-info.class` различается. Как убедиться в этом? Для начала распечатаем объявление модуля в обоих классах. Программу `javap`, находящуюся в каталоге `JDK_HOME\bin`, можно использовать для дизассемблирования любого файла класса. Ей нужно указать имя файла, URL-адрес или имя дизассемблируемого класса. Следующие команды печатают объявление модуля:

```
C:\Java9Revealed>javap mods\com.jdojo.intro\module-info.class
```

```
Compiled from "module-info.java"
module com.jdojo.intro {
  requires java.base;
}

C:\Java9Revealed>javap jar:file:lib/com.jdojo.intro-1.0.jar!/module-info.class

Compiled from "module-info.java"
```

```
module com.jdojo.intro {
  requires java.base;
}
```

В первой команде указано имя файла, а во второй – URL-адрес со схемой `jar`. В обоих случаях заданы относительные пути, но при желании можно задать и абсолютные.

Как видим, оба файла `module-info.class` содержат одно и то же объявление модуля. Чтобы увидеть атрибуты класса, нужно задать еще параметр `-verbose` (или `--v`). Следующая команда печатает содержимое файла `module-info.class` в каталоге `mods`. Как видно, ни имени главного класса, ни номера версии нет. Вывод показан не полностью.

```
C:\Java9Revealed>javap -verbose mods\com.jdojo.intro\module-info.class
```

```
Classfile /C:/Java9Revealed/mods/com.jdojo.intro/module-info.class
Last modified Jan 22, 2017; size 161 bytes
...
Constant pool:
   #1 = Class        #8              // "module-info"
   #2 = Utf8         SourceFile
   #3 = Utf8         module-info.java
   #4 = Utf8         Module
   #5 = Module       #9              // "com.jdojo.intro"
   #6 = Module       #10             // "java.base"
   #7 = Utf8         9-ea
   #8 = Utf8         module-info
   #9 = Utf8         com.jdojo.intro
  #10 = Utf8         java.base
{
}
SourceFile: "module-info.java"
Module:
  #5,0                               // "com.jdojo.intro"
  #0
  1                                  // requires
  #6,8000                            // "java.base" ACC_MANDATED
  #7                                 // 9-ea
  0                                  // exports
  0                                  // opens
  0                                  // uses
  0                                  // provides
```

Следующая команда печатает содержимое файла `module-info.class` в файле `lib\com.jdojo.intro-1.0.jar`. В этом случае имя главного класса и номер версии присутствуют. Соответствующие строки выделены полужирным шрифтом.

```
C:\Java9Revealed>javap -verbose jar:file:lib/com.jdojo.intro-1.0.jar!/module-info.class
```

```
Classfile jar:file:lib/com.jdojo.intro-1.0.jar!/module-info.class
...
Constant pool:
  ...
   #6 = Utf8               com/jdojo/intro
   #7 = Package            #6             // com/jdojo/intro
   #8 = Utf8               ModuleMainClass
   #9 = Utf8               com/jdojo/intro/Welcome
  #10 = Class      #9        // com/jdojo/intro/Welcome
  ...
  #14 = Utf8               1.0
  ...
{
}
SourceFile: "module-info.java"
ModulePackages:
#7                                      // com.jdojo.intro
ModuleMainClass: #10                    // com.jdojo.intro.Welcome
Module:
  #13,0                                 // "com.jdojo.intro"
  #14                                   // 1.0
  1                                     // requires
  #16,8000                              // "java.base" ACC_MANDATED
```

Можно также дизассемблировать код класса в модуле. Необходимо указать путь к модулю, имя модуля и полное имя класса. Следующая команда печатает код класса `com.jdojo.intro.Welcome`, находящегося в модульном JAR-файле:

```
C:\Java9Revealed>javap --module-path lib
 --module com.jdojo.intro com.jdojo.intro.Welcome
```

```
Compiled from "Welcome.java"
public class com.jdojo.intro.Welcome {
  public com.jdojo.intro.Welcome();
  public static void main(java.lang.String[]);
}
```

Подробные сведения можно напечатать и для системных классов. Следующая команда печатает информацию о классе `java.lang.Object` из модуля `java.base`. Отметим, что для системных классов задавать путь к модулю не нужно.

```
C:\Java9Revealed>javap --module java.base java.lang.Object
```

```
Compiled from "Object.java"
```

```
public class java.lang.Object {
  public java.lang.Object();
  public final native java.lang.Class<?> getClass();
  public native int hashCode();
  public boolean equals(java.lang.Object);
  ...
}
```

А как напечатать объявление системного модуля, например, `java.base` или `java.sql`? Напомним, что системные модули упакованы в файл специального формата, а не в модульный JAR-файл. В JDK 9 добавлена новая схема URL-адресов, `jrt` (аббревиатура «Java runtime»), для обозначения образов среды выполнения Java (или системных модулей):

```
jrt:/<module>/<path-to-a-file>
```

Следующая команда печатает объявление системного модуля `java.sql`:

```
C:\Java9Revealed>javap jrt:/java.sql/module-info.class
```

```
Compiled from "module-info.java"
module java.sql@9-ea {
  requires java.base;
  requires transitive java.logging;
  requires transitive java.xml;
  exports javax.transaction.xa;
  exports javax.sql;
  exports java.sql;
  uses java.sql.Driver;
}
```

Следующая команда печатает объявление модуля-агрегатора `java.se`:

```
C:\Java9Revealed>javap jrt:/java.se/module-info.class
```

```
Compiled from "module-info.java"
module java.se@9-ea {
  requires transitive java.sql;
  requires transitive java.rmi;
  requires transitive java.desktop;
  requires transitive java.security.jgss;
  requires transitive java.security.sasl;
  requires transitive java.management;
  requires transitive java.logging;
  requires transitive java.xml;
  requires transitive java.scripting;
```

```
  requires transitive java.compiler;
  requires transitive java.naming;
  requires transitive java.instrument;
  requires transitive java.xml.crypto;
  requires transitive java.prefs;
  requires transitive java.sql.rowset;
  requires java.base;
  requires transitive java.datatransfer;
}
```

Схему `jrt` можно также использовать для ссылки на системный класс. Следующая команда печатает информацию о классе `java.lang.Object` из модуля `java.base`:

```
C:\Java9Revealed>javap jrt:/java.base/java/lang/Object.class
```

```
Compiled from "Object.java"
  public class java.lang.Object {
  public java.lang.Object();
  public final native java.lang.Class<?> getClass();
  public native int hashCode();
  public boolean equals(java.lang.Object);
  ...
}
```

Резюме

Если модулю необходимы открытые типы, находящиеся в другом модуле, то второй модуль должен экспортировать пакет, содержащий типы, а первый модуль должен прочитать второй.

Для экспорта пакетов из модуля служит предложение `exports`. Модуль может экспортировать пакеты всем или только избранным модулям. Открытые типы, определенные в экспортированных пакетах, доступны другим модулям на этапах компиляции и выполнения. Экспортированный пакет не разрешает применять глубокую рефлексию к неоткрытым членам открытых типов.

Если модуль хочет дать другим модулям доступ ко всем членам – открытым и неоткрытым – с помощью рефлексии, то он должен быть объявлен раскрытым (`open`). Модуль также может избирательно раскрыть некоторые пакеты с помощью предложения `opens`. Для доступа к типам из раскрытых пакетов модуль не обязан читать модуль, содержащий эти пакеты.

Модуль объявляет зависимость от другого модуля в предложении `requires`. Зависимость может быть транзитивной, если указан модификатор `transitive`. Если модуль `M` объявляет транзитивную зависимость от модуля `N`, то любой модуль, объявивший зависимость от `M`, неявно объявляет зависимость и от `N`.

Зависимость можно объявить обязательной на этапе компиляции и факультативной на этапе выполнения, если включить в предложение `requires` модификатор `static`. Зависимость может быть одновременно факультативной и транзитивной.

В JDK 9 изменилась семантика открытых типов. Открытый тип, определенный в модуле, попадает в одну из трех категорий: открыт только внутри модуля-владельца, открыт только для указанных модулей и открыт для всех.

В зависимости от способа объявления и наличия имени модуль относится к одному из нескольких типов. Модули бывают *именованные* и *безымянные*. Если модуль имеет имя, то оно может быть задано явно в его объявлении или сгенерировано автоматически. Если имя задано явно в объявлении, то модуль называется *явным*, а если сгенерировано системой модулей по имени JAR-файла на пути к модулям, то *автоматическим*. Если в объявлении отсутствует модификатор `open`, то модуль называется *нормальным*, а если присутствует, то *раскрытым*. В соответствии с этими определениями раскрытый модуль является одновременно явным и именованным. Автоматический модуль является именованным, т. к. имеет имя, хотя и автоматически сгенерированное, но не является явным, поскольку имя неявно строится системой на этапах компиляции и выполнения.

Если обычный (не модульный) JAR-файл помещен на путь к модулям, то он представляет автоматический модуль, имя которого выведено из имени JAR-файла. Автоматический модуль читает все остальные модули, а все содержащиеся в нем пакеты экспортированы и раскрыты.

В JDK 9 загрузчик может загрузить класс из файла, находящегося на пути к модулям или к классам. С каждым загрузчиком классов ассоциирован безымянный модуль, в котором хранятся все типы, загруженные им из файлов на пути к классам. Безымянный модуль читает все остальные модули. Он экспортирует и раскрывает все свои пакеты всем остальным модулям. Поскольку у безымянного модуля нет имени, никакой явный модуль не может объявить зависимость от него на этапе компиляции. Если явному модулю необходим доступ к типам в безымянном модуле, то он может использовать автоматический модуль в качестве моста или прибегнуть к рефлексии.

Команда `javap` может напечатать объявление и атрибуты модуля. Для печати атрибутов дескриптора модуля нужно указать параметр `-verbose` (или `-v`). В JDK 9 образ среды выполнения хранится в специальном формате. В JDK 9 введена новая схема URL, `jrt`, позволяющая обратиться к содержимому образа среды выполнения. Ее синтаксическая структура – `jrt:/<module>/<path-to-a-file>`.

Глава 5

Реализация служб

Краткое содержание главы:

- что такое службы, интерфейсы служб и поставщики служб;
- как реализуются службы в JDK 9 и до JDK 9;
- использование интерфейса Java в качестве реализации службы;
- загрузка поставщика службы с помощью класса ServiceLoader;
- использование предложения uses в объявлении модуля для спецификации интерфейса службы, загружаемой модулем;
- использование предложения provides для спецификации поставщика службы, предоставляемой модулем;
- как обнаруживаются, фильтруются и выбираются поставщики служб на основе их класса без создания экземпляра;
- как поставщики служб упаковывались до JDK 9.

Что такое служба?

Функциональность, предоставляемая приложением (или библиотекой), называется *службой*. Так, могут существовать различные библиотеки, предоставляющие *службу простых чисел*, которая умеет проверять, является ли число простым, и находить простое число, следующее за указанным. Приложения и библиотеки, предоставляющие реализации служб, называются *поставщиками служб*. Приложения, использующие службу, называются *потребителями службы* или *клиентами*. Как клиент использует службу? Должен ли клиент знать всех поставщиков службы? Может ли клиент получить службу, не зная, кто ее предоставляет? Я отвечу на эти вопросы в этой главе.

В Java SE 6 имелся механизм, позволяющий разорвать связь между поставщиками и потребителями службы. То есть потребитель может использовать службу, ничего не зная о поставщике.

В Java *служба* определяется набором интерфейсов и классов. Служба содержит интерфейс или абстрактный класс, который определяет ее функциональность и называется *интерфейсом поставщика службы*, или просто *интерфейсом службы*. Отметим, что слово «интерфейс» в выражении «интерфейс службы» необязатель-

но подразумевает интерфейс в смысле языка Java. Это может быть как интерфейс, так и абстрактный класс. Возможно даже, хотя и не рекомендуется, использовать в качестве интерфейса службы конкретный класс. Иногда интерфейс службы называется также *типом службы*.

Конкретная реализация службы называется поставщиком *службы*. Может быть так, что несколько поставщиков реализуют один и тот же интерфейс службы.

В состав JDK входит класс `java.util.ServiceLoader<S>`, единственное назначение которого – обнаруживать и загружать поставщиков служб типа `S` во время выполнения. Класс `ServiceLoader` разрывает связь между поставщиками и потребителями служб. Потребитель службы знает только ее интерфейс, а класс `ServiceLoader` создает экземпляры поставщиков служб, которые реализуют известный клиентам интерфейс. На рис. 5.1 схематически изображена связь между службой, ее поставщиками и потребителем.

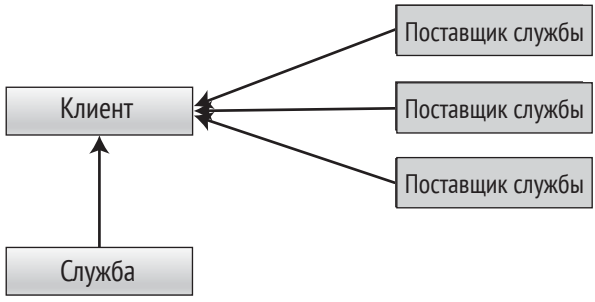

Рис. 5.1. Связь между службой, ее поставщиками и потребителем

Как правило, служба пользуется классом `ServiceLoader`, чтобы загрузить всех поставщиков службы и сделать их доступными клиентам. При такой архитектуре возможен механизм плагинов, когда поставщика можно добавить или удалить, не оказывая влияния на службу и ее потребителей. Потребители службы знают только о ее интерфейсе, но ничего не знают о его конкретных реализациях (поставщиках службы).

> **Совет.** Для получения полной информации о механизме загрузки служб в JDK 9 рекомендую прочитать документацию по классу `java.util.ServiceLoader`.

В этой главе мы будем работать с одним интерфейсом службы и тремя поставщиками. Их модули, имена классов или интерфейсов и краткие описания приведены в табл. 5.1. А на рис. 5.2 показаны связи между службой, ее поставщиками и потребителями.

Таблица 5.1. Список модулей, классов и интерфейсов, встречающихся в примерах из этой главы

Модуль	Класс/интерфейс	Описание
`com.jdojo.prime`	`PrimeChecker`	Играет роль интерфейса службы и самой службы
`com.jdojo.prime.generic`	`GenericPrimeChecker`	Поставщик службы

Модуль	Класс/интерфейс	Описание
com.jdojo.prime.faster	FasterPrimeChecker	Поставщик службы
com.jdojo.prime.probable	ProbablePrimeChecker	Поставщик службы
com.jdojo.prime.client	Main	Потребитель службы

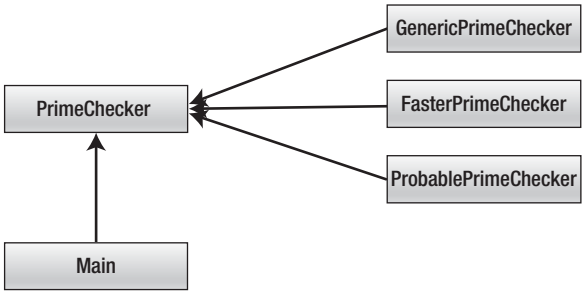

Рис. 5.2. Служба, три поставщика службы и потребитель службы, используемые в примерах из этой главы

Обнаружение служб

Чтобы воспользоваться службой, нужно найти и загрузить ее поставщиков. Этим занимается класс `java.util.ServiceLoader`. Модуль, который хочет обнаружить и загрузить поставщиков службы, должен включить в свое объявление предложение `uses` следующего вида:

```
uses <service-interface>;
```

Здесь `<service-interface>` – имя интерфейса службы, которое должно быть именем интерфейса Java, именем класса или именем типа аннотации. Если модуль пользуется классом `ServiceLoader<S>` для загрузки экземпляров поставщиков службы с интерфейсом `S`, то объявление должно содержать строку

```
uses S;
```

> **Совет.** На мой взгляд, слово `uses` в названии этого предложения неуместно. Создается впечатление, что указанную службу использует сам модуль, хотя на самом деле это не так. Служба используется клиентами, а не модулем, определяющим службу. Скорее подошло бы название `discovers` или `loads`. Однако нам предложено это название, и придется с ним жить. Правильно понять его смысл поможет такое прочтение определения: «Модуль, в объявлении которого встречается предложение `uses`, использует класс `ServiceLoader` для загрузки поставщиков службы с указанным интерфейсом». Нет необходимости использовать предложение `uses` в клиентских модулях, если эти модули не загружают поставщиков службы. Клиентские модули очень редко загружают службы.

Модуль может обнаруживать и загружать более одного интерфейса службы. В следующем объявлении модуля мы видим два предложения `uses`, т. е. модуль обнаруживает и загружает интерфейсы служб `com.jdojo.PrimeChecker` и `com.jdojo.CsvParser`:

```
module com.jdojo.loader {
    uses com.jdojo.PrimeChecker;
    uses com.jdojo.CsvParser:

    // Другие предложения модуля
}
```

В объявлении модуля могут быть предложения `import`. Предыдущее объявление будет понятнее, если переписать его в таком виде:

```
// Импортировать типы из других пакетов
import com.jdojo.PrimeChecker;
import com.jdojo.CsvParser:

module com.jdojo.loader {
    uses PrimeChecker;
    uses CsvParser:

// Другие предложения модуля
}
```

Интерфейс службы, указанной в предложении `uses`, может быть объявлен в данном или другом модуле. Если он объявлен в другом модуле, то должен быть доступен коду данного модуля, иначе при компиляции возникнет ошибка.

Обнаружение поставщика службы производится динамически во время выполнения. Модули, обнаруживающие поставщиков службы, обычно не объявляют зависимость от модулей поставщиков службы, потому что знать обо всех поставщиках заранее невозможно. Есть и другая причина не объявлять такую зависимость – чтобы не создавать тесную связь между поставщиком и потребителем.

Предоставление реализаций службы

В объявлении модуля, предоставляющего реализацию интерфейса службы, должно присутствовать предложение `provides`. Если модуль содержит поставщика службы, но предложение `provides` отсутствует, то класс `ServiceLoader` не загрузит такого поставщика. Таким образом, предложение `provides` – это сообщение классу `ServiceLoader`: «Послушай! У меня тут есть реализация службы. Если возникнет нужда, обращайся». Синтаксически предложение `provides` выглядит так:

```
provides <service-interface> with <service-implementation-name>;
```

Здесь `<service-interface>` – имя интерфейса службы, а `<service-implementation-name>` – имя класса, реализующего этот интерфейс. В JDK 9 поставщик службы может указать в качестве реализации интерфейса службы интерфейс Java. Именно так, как бы странно это ни звучало. Позже я приведу такой пример. В следующем объявлении есть два предложения `provides`:

```
module com.jdojo.provider {
    provides com.jdojo.PrimeChecker with com.jdojo.impl.PrimeCheckerFactory;
```

```
    provides com.jdojo.CsvParser with com.jdojo.impl.CsvFastParser;

    // Другие предложения модуля
}
```

В первом предложении объявлено, что `com.jdojo.impl.PrimeCheckerFactory` – одна из возможных реализаций интерфейса службы `com.jdojo.PrimeChecker`, а во втором – что `com.jdojo.impl.CsvFastParser` – одна из возможных реализаций интерфейса службы `com.jdojo.CsvParser`. До JDK 9 `PrimeCheckerFactory` и `CsvParser` должны были быть классами, а в JDK 9 могут быть также интерфейсами.

Объявление модуля может содержать предложения `uses` и `provides` в любой комбинации – один и тот же модуль может предоставлять реализацию службы и обнаруживать ее же; только предоставлять реализации одной или нескольких служб; предоставлять реализацию одной службы, а обнаруживать другую. Следующий модуль обнаруживает и предоставляет реализацию одной и той службы:

```
module com.jdojo.parser {
    uses com.jdojo.XmlParser;

    provides com.jdojo.XmlParser with com.jdojo.xml.impl.XmlParserFactory;

    // Другие предложения модуля
}
```

Чтобы было понятнее, можно переписать это объявление, воспользовавшись предложениями `import`:

```
import com.jdojo.XmlParser;
import com.jdojo.xml.impl.XmlParserFactory

module com.jdojo.parser {
    uses XmlParser;
    provides XmlParser with XmlParserFactory;

    // Другие предложения модуля
}
```

Совет. Класс или интерфейс реализации службы, указанный после слова `with`, должен быть объявлен в данном модуле, иначе возникнет ошибка компиляции.

Класс `ServiceLoader` создает экземпляры реализации службы. Если реализация службы – интерфейс, то `ServiceLoader` просто возвращает ссылку на интерфейс. Реализация службы (в виде класса или интерфейса) должна подчиняться следующим правилам.

- Если реализация службы явно или неявно объявляет открытый конструктор без параметров, то он называется *конструктором поставщика*.
- Если реализация службы содержит открытый статический метод `provider` без параметров, то он называется *методом поставщика*.

- Метод поставщика должен возвращать значение типа интерфейса службы или производного от него.
- Если реализация службы не содержит метода поставщика, то типом реализации должен быть класс с конструктором поставщика, и этот класс должен иметь тип интерфейса службы или производного от него.

Когда классу `ServiceLoader` поступает запрос обнаружить и загрузить поставщика службы, он проверяет, содержит ли реализация метод поставщика. Если такой метод существует, то возвращаемое им значение и есть служба, которую возвращает класс `ServiceLoader`. В противном случае вызывается конструктор поставщика, который создает экземпляр службы. Если в реализации службы нет ни метода, ни конструктора поставщика, то при компиляции возникнет ошибка.

В соответствии с этими правилами реализацией службы может быть интерфейс Java. В этом интерфейсе должен быть определен открытый статический метод `provider`, возвращающий экземпляр типа интерфейса службы.

В следующих разделах мы разберем шаги реализации службы в JDK 9, а в последнем разделе объясним, как заставить службу работать в немодульном окружении.

Определение интерфейса службы

В этом разделе мы разработаем службу проверки числа на простоту. Я сделаю ее совсем тривиальной, чтобы мы могли сосредоточиться на механизме поставщиков служб в JDK 9, а не на написании сложного кода, реализующего функциональность. Сформулируем требования к службе.

Служба должна предоставлять API для проверки простоты числа.

- Клиенты должны знать имя поставщика службы, т. е. каждый поставщик должен сообщать свое имя.
- У клиента должна быть возможность получить экземпляр службы, не указывая имя поставщика. В таком случае возвращается первый поставщик, найденный классом `ServiceLoader`. Если не найдено ни одного поставщика, то возбуждается исключение `RuntimeException`.
- У клиента должна быть возможность получить экземпляр службы, указав имя поставщика. Если поставщика с указанным именем не существует, то возбуждается исключение `RuntimeException`.

Теперь спроектируем службу. Ее функциональность будет представлена интерфейсом `PrimeChecker`, содержащим два метода:

```
public interface PrimeChecker {
    String getName();
    boolean isPrime(long n);
}
```

Метод `getName()` возвращает имя поставщика службы. Метод `isPrime()` возвращает `true`, если переданное число простое, и `false` в противном случае. Все поставщики службы будут реализовывать интерфейс `PrimeChecker`. Это и есть интерфейс (или тип) нашей службы.

Служба должна предоставлять клиентам API для получения экземпляра поставщика. Служба должна обнаружить и загрузить всех поставщиков, прежде чем клиенты смогут получить их. Поставщики службы загружаются классом `ServiceLoader`, не имеющим открытого конструктора. Мы можем вызвать один из его методов `load()` для загрузки службы определенного типа. Методу необходимо передать интерфейс поставщика службы. В классе `ServiceLoader` имеется метод `iterator()`, который возвращает объект типа `Iterator`, перебирающий всех загруженных поставщиков службы с определенным интерфейсом. В следующем фрагменте кода показано, как загрузить и перебрать всех поставщиков службы `PrimeChecker`:

```
// Загрузить поставщиков службы PrimeChecker
ServiceLoader<PrimeChecker> loader = ServiceLoader.load(PrimeChecker.class);

// Перебрать все загруженные экземпляры
Iterator<PrimeChecker> iterator = loader.iterator();

if(iterator.hasNext()) {
    PrimeChecker checker = iterator.next();
    // Использовать объект checker...
}
```

До JDK 8 необходимо было создать класс для обнаружения, загрузки и получения свойств службы. В JDK 8 появилась возможность включать статические методы в интерфейсы. Добавим два необходимых нам статических метода в интерфейс службы:

```
public interface PrimeChecker {
    String getName();
    boolean isPrime(long n);

    static PrimeChecker newInstance();
    static PrimeChecker newInstance(String providerName)
}
```

Метод `newInstance()` возвращает экземпляр первого найденного объекта типа `PrimeChecker`. Второй вариант этого метода возвращает экземпляр поставщика с указанным именем.

Создадим модуль `com.jdojo.prime`. В листинге 5.1 приведен полный код интерфейса `PrimeChecker`.

Листинг 5.1. Интерфейс поставщика службы PrimeChecker

```
// PrimeChecker.java
package com.jdojo.prime;

import java.util.ServiceLoader;

public interface PrimeChecker {
/**
```

```
* Возвращает имя поставщика службы.
*
* @return Имя поставщика службы
*/
String getName();

/**
* Возвращает true, если переданное число простое, иначе false.
*
* @param n Проверяемое число
* @return true, если число n простое, иначе false.
*/
boolean isPrime(long n);

/**
* Возвращает первого найденного поставщика службы PrimeChecker.
*
* @return Первый найденный поставщик службы PrimeChecker.
* @throws RuntimeException Если не найдено ни одного поставщика службы
*    PrimeChecker.
*/
static PrimeChecker newInstance() throws RuntimeException {
    return ServiceLoader.load(PrimeChecker.class)
                        .findFirst()
                        .orElseThrow(() -> new RuntimeException(
                          "No PrimeChecker service provider found."));
}

/**
* Возвращает экземпляр именованного поставщика службы PrimeChecker.
*
* @param providerName Имя поставщика службы PrimeChecker
* @return Экземпляр PrimeChecker
*/
static PrimeChecker newInstance(String providerName) throws RuntimeException {
    ServiceLoader<PrimeChecker> loader = ServiceLoader.load(PrimeChecker.class);
    for (PrimeChecker checker : loader) {
        if (checker.getName().equals(providerName)) {
            return checker;
        }
    }

    throw new RuntimeException("A PrimeChecker service provider with the name '"
        + providerName + "' was not found.");
}
}
```

В листинге 5.2 приведено объявление модуля `com.jdojo.prime`. Он экспортирует пакет `com.jdojo.prime`, поскольку интерфейс `PrimeChecker` будет нужен модулям поставщиков служб и клиентским модулям.

Листинг 5.2. Объявление модуля com.jdojo.prime

```
// module-info.java
module com.jdojo.prime {
    exports com.jdojo.prime;

    uses com.jdojo.prime.PrimeChecker;
}
```

Необходимо указывать полное имя интерфейса `PrimeChecker` в предложении `uses`, т. к. код модуля (методы `newInstance()`) пользуется классом `ServiceLoader` для загрузки поставщиков службы. Если хотите использовать в предложении `uses` простые имена, то добавьте соответствующие предложения `import`, как было показано выше.

Это все, что необходимо для определения службы проверки на простоту.

Определение поставщиков службы

В следующих двух разделах мы создадим два поставщика службы `PrimeChecker`. Первый реализует обычную проверку на простоту, второй – более быструю. Впоследствии мы напишем клиент для тестирования службы. Вы сможете воспользоваться любым поставщиком на выбор или сразу обоими.

Поставщики службы реализуют различные алгоритмы проверки числа на простоту. Напомним, что число называется простым, если оно делится только на 1 и на самого себя. 1 не считается простым числом. Вот первые простые числа: 2, 3, 5, 7, 11.

Определение обычного поставщика службы простых чисел

В этом разделе мы определим обычного поставщика службы `PrimeChecker`. Для этого нужно просто создать класс, реализующий интерфейс службы, или интерфейс, в котором определен метод поставщика. В данном случае мы создадим класс `GenericPrimeChecker`, который реализует интерфейс `PrimeChecker` и содержит конструктор поставщика.

Этот поставщик будет определен в отдельном модуле `com.jdojo.prime.generic`. В листинге 5.3 приведено объявления модуля. Но пока оно еще не компилируется.

Листинг 5.3. Объявление модуля com.jdojo.prime.generic

```
// module-info.java
module com.jdojo.prime.generic {
requires com.jdojo.prime;

provides com.jdojo.prime.PrimeChecker
    with com.jdojo.prime.generic.GenericPrimeChecker;
}
```

Предложение `requires` необходимо, потому что этот модуль использует интерфейс `PrimeChecker` из модуля `com.jdojo.prime`. В NetBeans поместите проект `com.jdojo.prime` на путь к модулям в проекте `com.jdojo.prime.generic`. Это устранит ошибку при компиляции объявления модуля, вызванную отсутствием модуля `com.jdojo.prime`, упомянутого в предложении `requires`.

В предложении `provides` говорится, что этот модуль предоставляет реализацию интерфейса `PrimeChecker`, а в части `with` указано имя класса реализации. Класс реализации должен удовлетворять следующим условиям.

- Это должен быть открытый конкретный класс, верхнего уровня или вложенный статический. Класс не может быть внутренним или абстрактным.
- В классе должен быть открытый конструктор без аргументов. Он используется классом `ServiceLoader` для создания экземпляра поставщика с помощью отражения. Отметим, что мы не включили в класс `GenericPrimeChecker` метод поставщика, являющийся альтернативой конструктору без аргументов.
- Экземпляр класса реализации должен быть совместим по присваиванию с интерфейсом поставщика службы.

Если какое-то условие не выполнено, то при компиляции возникнет ошибка. Отметим, что нет необходимости экспортировать пакет `com.jdojo.prime.generic`, содержащий класс реализации службы, потому что не предполагается, что клиенты будут зависеть от него напрямую. Клиентам нужно знать только интерфейс службы, а не его конкретную реализацию. Класс `ServiceLoader` может создать класс реализации и обращаться к нему, даже если модуль не экспортирует пакет, содержащий этот класс.

> **Совет.** Если в объявлении модуля есть предложение `provides`, то указанный в нем интерфейс службы может находиться в том же или в другом доступном модуле. Но реализация класса или интерфейса службы, указанного в части `with`, должна быть определена в данном модуле.

В листинге 5.4 показан код класса `GenericPrimeChecker`, реализующего интерфейс службы `PrimeChecker`. Метод `getName()` возвращает имя поставщика службы, я назвал его `jdojo.generic.primechecker`. Это имя можно выбрать произвольно. Метод `isPrime()` возвращает `true`, если переданное число простое, а иначе `false`.

Листинг 5.4. Класс `GenericPrimeChecker`, реализующий интерфейс службы `PrimeChecker`

```
// GenericPrimeChecker.java
package com.jdojo.prime.generic;

import com.jdojo.prime.PrimeChecker;

public class GenericPrimeChecker implements PrimeChecker {
private static final String PROVIDER_NAME = "jdojo.generic.primechecker";

@Override
public boolean isPrime(long n) {
```

```
    if(n <= 1) {
        return false;
    }
    if (n == 2) {
        return true;
    }
    if(n%2 == 0) {
        return false;
    }

    for(long i = 3; i < n; i += 2) {
        if(n%i == 0) {
            return false;
        }
    }
    return true;
}

@Override
public String getName() {
    return PROVIDER_NAME;
}
}
```

С этим модулем всё. Как уже было сказано, для его компиляции необходимо поместить модуль `com.jdojo.prime` на путь к модулям. Откомпилируйте код и упакуйте его в модульный JAR-файл. Тестировать пока нечего.

Определение быстрого поставщика службы простых чисел

В этом разделе мы определим еще одного поставщика службы `PrimeChecker`. Назовем его *быстрым* поставщиком, поскольку он реализует более быстрый алгоритм проверки. Этот поставщик будет находиться в отдельном модуле `com.jdojo.prime.faster`, а класс реализации будет называться `FasterPrimeChecker`.

В листинге 5.5 приведено объявление модуля, очень похожее на предыдущее. Изменилось только имя класса в части `with`.

Листинг 5.5. Объявление модуля com.jdojo.prime.faster

```
// module-info.java
module com.jdojo.prime.faster {
requires com.jdojo.prime;

provides com.jdojo.prime.PrimeChecker
    with com.jdojo.prime.faster.FasterPrimeChecker;
}
```

Из-за наличия предложения `requires` мы должны поместить проект `com.jdojo.prime` на путь к модулям в проекте `com.jdojo.prime.faster` в NetBeans.

В листинге 5.6 показан код класса `FasterPrimeChecker`, в котором метод `isPrime()` работает быстрее, чем в классе `GenericPrimeChecker`. На этот раз цикл начинается с 3 и заканчивается, когда дойдет до квадратного корня из проверяемого числа.

Листинг 5.6. Класс `FasterPrimeChecker`, реализующий интерфейс службы `PrimeChecker`

```
// FasterPrimeChecker.java
package com.jdojo.prime.faster;

import com.jdojo.prime.PrimeChecker;

public class FasterPrimeChecker implements PrimeChecker {
   private static final String PROVIDER_NAME = "jdojo.faster.primechecker";

   // Открытый конструктор поставщика отсутствует
   private FasterPrimeChecker() {
   }

   // Определить метод поставщика
   public static PrimeChecker provider() {
      return new FasterPrimeChecker();
   }

   @Override
   public boolean isPrime(long n) {
      if (n <= 1) {
          return false;
      }
      if (n == 2) {
          return true;
      }
      if (n % 2 == 0) {
          return false;
      }

      long limit = (long) Math.sqrt(n);
      for (long i = 3; i <= limit; i += 2) {
          if (n % i == 0) {
              return false;
          }
      }
      return true;
   }

   @Override
```

```
  public String getName() {
     return PROVIDER_NAME;
  }
}
```

Обратите внимание на различие между классами `GenericPrimeChecker` и `FasterPrimeChecker`. В классе `GenericPrimeChecker` имеется конструктор по умолчанию, играющий роль конструктора поставщика, но метода поставщика в нем нет. В классе `FasterPrimeChecker` конструктор без аргументов сделан закрытым, и, значит, конструктором поставщика он быть не может. Зато в этом классе есть метод поставщика:

```
public static PrimeChecker provider() {/*...*/}
```

Когда классу `ServiceLoader` понадобится создать экземпляр быстрой службы простых чисел, он вызовет этот метод, который просто создает и возвращает объект класса `FasterPrimeChecker`.

С этим модулем тоже всё. Для его компиляции поместите модуль `com.jdojo.prime` на путь к модулям. Откомпилируйте код и упакуйте его в модульный JAR-файл.

Определение самого быстрого поставщика службы простых чисел

В этом разделе я покажу, как использовать интерфейс Java для реализации службы. Мы определим еще одного поставщика службы PrimeChecker и назовем его *вероятностным*, потому что он говорит лишь, что число, возможно, простое. Поставщик будет определен в отдельном модуле `com.jdojo.prime.probable`, а интерфейс реализации будет называться `ProbablePrimeChecker`.

В классе `BigInteger` из пакета `java.math` имеется метод `isProbablePrime(int certainty)`, который возвращает `true`, если число может быть простым. Параметр `certainty` определяет степень уверенности в простоте числа. Чем этот параметр больше, тем дороже обходится вызов метода и тем выше вероятность, что число действительно простое, когда метод возвращает `true`.

В листинге 5.7 приведено объявление модуля, похожее на два предыдущих. Отличается только имя интерфейса в части `with`.

Листинг 5.7. Объявление модуля com.jdojo.prime.probable

```
// module-info.java
module com.jdojo.prime.probable {
requires com.jdojo.prime;
provides com.jdojo.prime.PrimeChecker
    with com.jdojo.prime.probable.ProbablePrimeChecker;
}
```

Из-за наличия предложения `requires` мы должны поместить проект `com.jdojo.prime` на путь к модулям в проекте `com.jdojo.prime.probable` в NetBeans. В листинге 5.8 показан код класса `ProbablePrimeChecker`.

Листинг 5.8. Класс `ProbablePrimeChecker`, реализующий интерфейс службы `PrimeChecker`

```
// ProbablePrimeChecker.java
package com.jdojo.prime.probable;

import com.jdojo.prime.PrimeChecker;
import java.math.BigInteger;

public interface ProbablePrimeChecker {
   // Метод поставщика
   public static PrimeChecker provider() {
      final String PROVIDER_NAME = "jdojo.probable.primechecker";

      return new PrimeChecker() {
          @Override
          public boolean isPrime(long n) {
              // Задаем высокую степень уверенности 1000, число выбрано произвольно
              int certainty = 1000;
              return BigInteger.valueOf(n).isProbablePrime(certainty);
          }

          @Override
          public String getName() {
              return PROVIDER_NAME;
          }
      };
   }
}
```

В интерфейсе `ProbablePrimeChecker` определен только метод поставщика:

```
public static PrimeChecker provider() {/*...*/}
```

Когда классу `ServiceLoader` понадобится создать экземпляр вероятностной службы простых чисел, он вызовет этот метод, который просто создает и возвращает экземпляр интерфейса `PrimeChecker`. Метод `isPrime()` использует класс `BigInteger` для проверки простоты числа. Поставщик называется `jdojo.probable.primechecker`. Обратите внимание, что этот интерфейс не расширяет интерфейс `PrimeChecker`. Чтобы считаться реализацией службы, метод поставщика должен возвращать экземпляр интерфейса службы (`PrimeChecker`) или его подтип. Объявив тип возвращаемого значения `PrimeChecker`, мы выполнили это требование.

С этим модулем тоже покончено. Для его компиляции поместите модуль `com.jdojo.prime` на путь к модулям. Откомпилируйте код и упакуйте его в модульный JAR-файл.

Тестирование службы простых чисел

В этом разделе мы протестируем службу, для чего создадим клиентское приложение в отдельном модуле `com.jdojo.prime.client`. В листинге 5.9 приведено объявление этого модуля.

Листинг 5.9. Объявление модуля com.jdojo.prime.client

```
// module-info.java
module com.jdojo.prime.client {
    requires com.jdojo.prime;
}
```

Клиентский модуль должен знать только об интерфейсе службы. В данном случае интерфейс определен в модуле `com.jdojo.prime`. Поэтому клиентский модуль читает этот модуль и больше ничего. В реальности клиентский модуль был бы гораздо сложнее и, наверное, читал бы и другие модули. Чтобы откомпилировать модуль в NetBeans, поместите проект `com.jdojo.prime` на путь к модулям. На рис. 5.3 показан граф модулей.

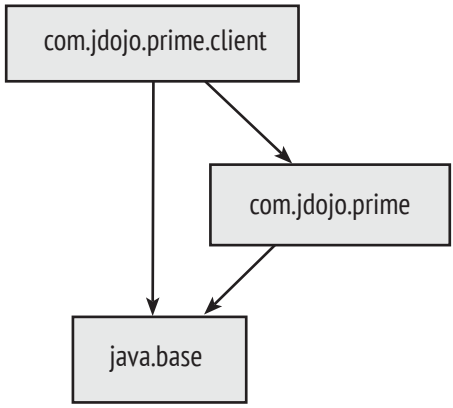

Рис. 5.3. Граф модуля `com.jdojo.prime.client`

Совет. Клиентский модуль не знает о модулях поставщиков служб и не обязан читать их напрямую. Обязанность обнаруживать всех поставщиков служб и делать их экземпляры доступными клиенту возлагается на службу. В данном случае модуль `com.jdojo.prime` определяет интерфейс `com.jdojo.prime.PrimeChecker`, выступающий в роли службы.

В листинге 5.10 приведен код клиента, пользующегося службой `PrimeChecker`.

Листинг 5.10. Класс Main для тестирования службы PrimeChecker

```
// Main.java
package com.jdojo.prime.client;

import com.jdojo.prime.PrimeChecker;

public class Main {
   public static void main(String[] args) {
      // Числа, проверяемые на простоту
      long[] numbers = {3, 4, 121, 977};

      // Попробовать поставщика службы по умолчанию
```

```
        try {
            PrimeChecker checker = PrimeChecker.newInstance();
            checkPrimes(checker, numbers);
        } catch (RuntimeException e) {
            System.out.println(e.getMessage());
        }

        // Попробовать быстрого поставщика службы
        try {
            PrimeChecker checker = PrimeChecker.newInstance("jdojo.faster.primechecker");
            checkPrimes(checker, numbers);
        } catch (RuntimeException e) {
            System.out.println(e.getMessage());
        }

        // Попробовать вероятностного поставщика службы
        try {
            PrimeChecker checker = PrimeChecker.newInstance("jdojo.probable.primechecker");
            checkPrimes(checker, numbers);
        } catch (RuntimeException e) {
            System.out.println(e.getMessage());
        }
    }

    public static void checkPrimes(PrimeChecker primeChecker, long... numbers) {
        System.out.format("Используется %s:%n", primeChecker.getName());

        for (long n : numbers) {
            if (primeChecker.isPrime(n)) {
                System.out.printf("%d простое.%n", n);
            } else {
                System.out. printf("%d не простое.%n", n);
            }
        }
    }
}
```

Метод `checkPrimes()` принимает экземпляр `PrimeChecker` и переменное число аргументов типа `long`. Он использует `PrimeChecker` для проверки числа на простоту и печатает соответствующие сообщения. Метод `main()` получает экземпляры поставщика службы по умолчанию, а также поставщиков `jdojo.faster.primechecker` и `jdojo.probable.primechecker`. Каждый из трех поставщиков используется для проверки одних и тех же чисел. Откомпилируйте и упакуйте код модуля. Если выполнить класс `Main`, когда на пути к модулям находятся только модули `com.jdojo.prime` и `com.jdojo.prime.client`, то будут выданы такие сообщения:

```
No PrimeChecker service provider found.
A PrimeChecker service provider with the name 'jdojo.faster.primechecker' was not found.
A PrimeChecker service provider with the name 'jdojo.probable.primechecker' was not found.
```

Поставщиков службы на пути к модулям нет, поэтому все три попытки найти поставщика заканчиваются неудачно.

Если выполнить класс `Main`, когда на пути к модулям находятся модули `com.jdojo.prime`, `com.jdojo.prime.generic` и `com.jdojo.prime.client`, то получим:

```
Используется jdojo.generic.primechecker:
3 простое.
4 не простое.
121 не простое.
977 простое.
A PrimeChecker service provider with the name 'jdojo.faster.primechecker' was not found.
A PrimeChecker service provider with the name 'jdojo.probable.primechecker' was not found.
```

На этот раз найден один поставщик, `jdojo.generic.primechecker`, а остальных двух на пути к модулям не оказалось.

Если выполнить класс `Main`, когда на пути к модулям находятся модули `com.jdojo.prime`, `com.jdojo.prime.generic`, `com.jdojo.prime.faster`, `com.jdojo.prime.probable` и `com.jdojo.prime.client`, то получим аналогичный результат. Первым итератор нашел обычного поставщика, так что, запросив поставщика по умолчанию, мы получили именно его. Однако не исключено, что при прогоне программы на вашей машине поставщиком по умолчанию окажется какой-то другой.

```
Используется jdojo.generic.primechecker:
3 простое.
4 не простое.
121 не простое.
977 простое.
Используется jdojo.faster.primechecker:
3 простое.
4 не простое.
121 не простое.
977 простое.
Используется jdojo.probable.primechecker:
3 простое.
4 не простое.
121 не простое.
977 простое.
```

Встретив предложение `uses` в объявлении модуля, система просматривает путь к модулям в поисках модулей, содержащих предложения `provides`, в которых упомянуты реализации интерфейса службы, указанного в предложении `uses`. В этом смысле предложение `uses` можно считать косвенной факультативной зависимостью от других модулей, разрешаемой на этапе выполнения.

Выборка и фильтрация поставщиков

Иногда требуется отобрать поставщиков по имени класса, например, выбрать только тех поставщиков службы простых чисел, для которых полное имя начинается с `com.jdojo`. Кажется естественным воспользоваться для этого итератором, возвращенным методом `iterator()` класса `ServiceLoader`. Однако такой способ обходится дорого, потому что итератор создает экземпляр поставщика, прежде чем вернуть его, пусть даже это происходит лениво, т. е. в тот момент, когда его действительно запрашивают.

В JDK 9 в класс `ServiceLoader` добавлен такой метод:

```
public Stream<ServiceLoader.Provider<S>> stream()
```

Он возвращает поток экземпляров интерфейса `ServiceProvider.Provider`, который объявлен как вложенный в классе `ServiceLoader`:

```
public static interface Provider<S> extends Supplier<S> {
    Class<? extends S> type();

    @Override
    S get();
}
```

Экземпляр интерфейса `ServiceLoader.Provider` представляет поставщика службы. Его метод `type()` возвращает объект `Class` реализации службы. Метод `get()` создает и возвращает экземпляр поставщика. Как этот интерфейс помогает? При использовании метода `stream()` каждый элемент потока имеет тип `ServiceLoader.Provider`, и мы можем профильтровать этот поток по имени класса или типу поставщика, при этом создавать экземпляр поставщика не нужно. В фильтре можно использовать метод `type()`, а когда требуемый поставщик будет найден, вызвать метод `get()`, чтобы создать его экземпляр. Таким образом, мы создаем поставщика, когда заведомо знаем, что он нужен, а не в процессе перебора всех поставщиков.

Ниже приведен пример использования метода `stream()` класса `ServiceLoader`. Он дает список всех поставщиков службы простых чисел, имена которых начинаются с указанной строки.

```
static List<PrimeChecker> startsWith(String prefix) {
    return ServiceLoader.load(PrimeChecker.class)
                        .stream()
                        .filter((Provider p) -> p.type().getName().startsWith(prefix))
                        .map(Provider::get)
                        .collect(Collectors.toList());
}
```

Этот метод можно добавить в интерфейс `PrimeChecker`, нужно только не забыть о нескольких предложениях импорта:

```
import java.util.List;
import java.util.ServiceLoader.Provider;
import java.util.stream.Collectors;
```

Вот пример вызова этого метода из клиентского класса:

```
// Получить список всех поставщиков службы простых чисел, имена которых начинаются с
// "com.jdojo"
List<PrimeChecker> jdojoService = PrimeChecker.startsWith("com.jdojo");
```

Тестирование службы простых чисел по-старому

Не все приложения будут переделаны под работу с модулями. Возможно, ваши модульные JAR-файлы службы простых чисел будут использоваться совместно с обычными JAR-файлами на пути к классам. Допустим, что вы поместили все пять модульных JAR-файлов службы в каталог `C:\Java9Revealed\lib`. Поместите четыре модульных JAR-файла на путь к классам и выполните класс `com.jdojo.prime.client.Main` командой

```
C:\Java9Revealed>java --class-path lib\com.jdojo.prime.jar; lib\com.jdojo.prime.
client.jar;lib\com.jdojo.prime.faster.jar; lib\com.jdojo.prime.generic.jar;lib\com.
jdojo.prime.probable.jar com.jdojo.prime.client.Main
```

```
No PrimeChecker service provider found.
A PrimeChecker service provider with the name 'jdojo.faster.primechecker' was not found.
A PrimeChecker service provider with the name 'jdojo.probable.primechecker' was not found.
```

Как видно, в унаследованном, предшествующем JDK 9 режиме, когда все модульные JAR-файлы помещаются на путь к классам, ни один поставщик службы не найден. В этом режиме механизм обнаружения поставщиков служб работает по-другому. Класс `ServiceLoader` перебирает все JAR-файлы на пути к классам и смотрит, какие файлы находятся в каталоге `META-INF/services`. В качестве имени файла указывается полное имя интерфейса службы, например:

```
META-INF/services/<service-interface>
```

В файле хранится список полных имен классов или интерфейсов реализации поставщиков службы. Каждое имя класса должно располагаться в отдельной строке. Допускаются однострочные комментарии, начинающиеся символом `#`.

Интерфейс службы называется `com.jdojo.prime.PrimeChecker`, поэтому в модульных JAR-файлах всех трех поставщиков этой службы будет присутствовать файл `com.jdojo.prime.PrimeChecker` с таким путем:

```
META-INF/services/com.jdojo.prime.PrimeChecker
```

Каталог `META-INF/services` необходимо поместить в корень каталога с исходным кодом. Интегрированные среды, в частности NetBeans, сами позаботятся о его создании. В листингах 5.11, 5.12 и 5.13 приведено содержимое этого файла в модульных JAR-файлах трех поставщиков службы простых чисел.

Листинг 5.11. Содержимое файла META-INF/services/com.jdojo.prime.PrimeChecker в модульном JAR-файле модуля com.jdojo.prime.generic

```
# Имя класса реализации обычного поставщика службы
com.jdojo.prime.generic.GenericPrimeChecker
```

Листинг 5.12. Содержимое файла META-INF/services/com.jdojo.prime.PrimeChecker в модульном JAR-файле модуля com.jdojo.prime.faster

```
# Имя класса реализации быстрого поставщика службы
com.jdojo.prime.faster.FasterPrimeChecker
```

Листинг 5.13. Содержимое файла META-INF/services/com.jdojo.prime.PrimeChecker в модульном JAR-файле модуля com.jdojo.probable

```
# Имя класса реализации вероятностного поставщика службы
com.jdojo.prime.probable.ProbablePrimeChecker
```

Заново откомпилируйте и упакуйте модульные JAR-файлы обычного и быстрого поставщиков службы простых чисел. Выполните следующую команду:

```
C:\Java9Revealed>java --class-path lib\com.jdojo.prime.jar; lib\com.jdojo.prime.
client.jar;lib\com.jdojo.prime.faster.jar; lib\com.jdojo.prime.generic.jar;lib\com.
jdojo.prime.probable.jar
com.jdojo.prime.client.Main
```

```
Exception in thread "main" java.util.ServiceConfigurationError: com.jdojo.prime.
PrimeChecker: com.jdojo.prime.faster.FasterPrimeChecker Unable to get public no-arg
constructor
...
at com.jdojo.prime.client.Main.main(Main.java:13)
Caused by: java.lang.NoSuchMethodException: com.jdojo.prime.faster.
FasterPrimeChecker.<init>()
...
```

Часть вывода опущена. Мы видим, что произошло исключение, когда класс `ServiceLoader` пытался создать экземпляр быстрого поставщика. Такая же ошибка возникнет при попытке создать экземпляр вероятностного поставщика. Добавление информации о службе в каталог `META-INF/services` – унаследованный способ реализации служб. Для обратной совместимости реализация службы

должна быть классом с открытым конструктором без аргументов. Однако конструктор поставщика мы реализовали только в классе `GenericPrimeChecker`. Поэтому два остальных поставщика в унаследованном режиме работать не будут. Чтобы класс `FasterPrimeChecker` заработал, можно добавить в него конструктор поставщика. Но добавить конструктор поставщика в интерфейс невозможно, так что `ProbablePrimeChecker` в режиме пути к классам не будет работать. Его необходимо загружать из явного модуля.

> **Совет.** Поставщики служб, развернутые на пути к классам или как автоматические модули, обязаны иметь открытый конструктор без аргументов.

Следующая команда добавляет модульный JAR-файл только для обычного поставщика службы, в котором есть открытый конструктор без аргументов. Как видим, поставщик найден, его экземпляр создан и успешно использован.

```
C:\Java9Revealed>java --class-path lib\com.jdojo.prime.jar; lib\com.jdojo.prime.
client.jar;lib\com.jdojo.prime.generic.jar com.jdojo.prime.client.Main
```

```
Используется jdojo.generic.primechecker:
3 простое.
4 не простое.
121 не простое.
977 простое.
A PrimeChecker service provider with the name 'jdojo.faster.primechecker' was not found.
A PrimeChecker service provider with the name 'jdojo.probable.primechecker' was not found.
```

Резюме

Функциональность, предоставляемая приложением (или библиотекой) называется *службой*, а приложения и библиотеки, предоставляющие службы, – поставщиками служб. Приложения, пользующиеся службами, называются *потребителями служб* или *клиентами*.

В Java служба определяется набором интерфейсов и классов. Служба содержит интерфейс или абстрактный класс, описывающий ее функциональность, он называется *интерфейсом поставщика службы*, *интерфейсом службы* или *типом службы*. Конкретная реализация интерфейса службы называется *поставщиком службы*. Для одного интерфейса службы может быть несколько поставщиков. В JDK 9 поставщик службы может быть классом или интерфейсом.

В JDK имеется класс `java.util.ServiceLoader<S>`, единственная задача которого – обнаруживать и загружать поставщиков службы типа `S` во время выполнения. Если JAR-файл (модульный или обычный), содержащий поставщика службы, помещен на путь к классам, то для нахождения поставщиков службы класс `ServiceLoader` пользуется каталогом `META-INF/services`. Имена файлов в этом каталоге должны быть полными именами интерфейса службы. Каждый файл содержит полные имена классов реализации поставщиков службы, по одному имени на строку.

В файле могут присутствовать однострочные комментарии, начинающиеся символом #. Для обнаружения поставщиков службы класс ServiceLoader просматривает все каталоги META-INF/services на пути к классам.

В JDK 9 механизм обнаружения поставщиков служб изменился. В объявлении модуля, который пользуется классом ServiceLoader для обнаружения и загрузки поставщиков, должно присутствовать предложение uses, описывающее интерфейс службы. Сам интерфейс службы, указанный в предложении uses, может быть объявлен как в данном модуле, так и в любом другом доступном ему. Для перебора всех поставщиков службы можно воспользоваться методом iterator() класса ServiceLoader. Метод stream() возвращает поток экземпляров интерфейса ServiceLoader.Provider. Его можно использовать для фильтрации и выборки поставщиков определенного типа по имени класса без создания экземпляров поставщиков.

Модуль, содержащий поставщика службы, должен специфицировать интерфейс этой службы и класс его реализации в предложении provides. Класс реализации должен находиться в данном модуле.

Глава 6

Упаковка модуля

Краткое содержание главы:

- различные форматы упаковки Java-модулей;
- новые возможности формата JAR;
- что такое многоверсионный JAR-файл;
- создание и использование многоверсионных JAR-файлов;
- что такое формат JMOD;
- использование команды `jmod` для работы с JMOD-файлами;
- создание JMOD-файлов, извлечение содержимого и описание;
- распечатка содержимого JMOD-файла;
- запись хэшей модулей в JMOD-файл для проверки зависимостей.

Модуль можно упаковать в файлы различных форматов для использования на трех этапах: компиляции, компоновки и выполнения. Не каждый формат поддерживается на каждом этапе. JDK 9 поддерживает следующие форматы упаковки модулей:

- развернутый каталог;
- формат JAR;
- формат JMOD;
- формат JIMAGE.

Развернутые каталоги и формат JAR поддерживались и до JDK 9. В JDK 9 формат JAR был доработан для поддержки модульных и многоверсионных файлов. Также появилось два новых формата для упаковки модулей: JMOD и JIMAGE. В этой главе мы обсудим новые возможности формата JAR и формат JMOD, а в главе 7 формат JIMAGE и программу `jlink`.

Формат JAR

В главе 3 рассказывалось о новых параметрах команды `jar` для создания модульных JAR-файлов. Команда `jar` позволяет также выводить список элементов JAR-файла, извлекать отдельные файлы и обновлять содержимое архива. Все эти

операции поддерживались и до JDK 9, и сейчас в них ничего не изменилось. В этой главе мы рассмотрим новую возможность формата JAR – многоверсионность.

Что такое многоверсионный JAR-файл?

Всякому опытному Java-разработчику доводилось использовать различные библиотеки и каркасы, например, Spring, Hibernate и т. д. Быть может, вы работаете с Java 8, а библиотеки все еще ориентированы на Java 6 или Java 7. Почему бы разработчикам библиотек не использовать последнюю версию JDK и появившиеся в ней новшества? Например, потому что не все пользователи работают с последним JDK. Переход библиотеки на последнюю версию JDK означал бы, что пользователей принудительно заставляют перейти на ту же версию, но на практике это не всегда возможно. Сопровождение и выпуск версий библиотеки, поддерживающих разные JDK, – дополнительная проблема, связанная с упаковкой кода. Обычно для каждой версии JDK существует свой JAR-файл. В JDK эта проблема решена благодаря добавлению нового способа упаковки библиотечного кода – записи в один JAR-файл одной и той же версии библиотеки для нескольких версий JDK. Такой JAR-называется *многоверсионным* (multi-release).

Таким образом, в виде многоверсионного JAR-файла (MRJAR) можно поставить библиотеку, которая будет работать и в JDK 8, и в JDK 9. Код, хранящийся в MRJAR-файле, будет содержать файлы классов, откомпилированных для JDK 8 и JDK 9. В классах, откомпилированных для JDK 9, могут использоваться новые API, а в классах, откомпилированных для JDK 8, та же функциональность библиотеки реализована с помощью старых API JDK 8.

MRJAR-файл расширяет существующую структуру каталогов. В JAR-файле имеется корневой каталог, в котором находится все его содержимое. Имеется также каталог `META-INF` для хранения метаданных о JAR-файле. Как правило, JAR-файл содержит файл `META-INF/MANIFEST.MF`, в котором хранятся атрибуты. Типичный JAR-файл выглядит следующим образом:

```
- jar-root
  - C1.class
  - C2.class
  - C3.class
  - C4.class
- META-INF
  - MANIFEST.MF
```

Этот JAR-файл содержит четыре файла классов и файл `MANIFEST.MF`. В MRJAR каталог `META-INF` расширен, в нем хранятся классы, относящиеся к конкретной версии JDK. В каталоге `META-INF` имеется подкаталог `versions`, в котором может находиться много подкаталогов с именами, соответствующими основной версии JDK. Например, классы, специфичные для JDK 9, могут быть помещены в каталог `META-INF/versions/9`, специфичные для JDK 10 — в каталог `META-INF/versions/10` и т. д. А типичном MRJAR-файле могут быть такие элементы:

```
- jar-root
  - C1.class
  - C2.class
```

```
  - C3.class
  - C4.class
- META-INF
  - MANIFEST.MF
  - versions
    - 9
      - C2.class
      - C5.class
    - 10
      - C1.class
      - C2.class
      - C6.class
```

Если этот MRJAR-файл используется в окружении, не поддерживающем формат MRJAR, то он будет рассматриваться как обычный JAR-файл – используется только содержимое корневого каталога, а файлы в каталогах `META-INF/versions/9` и `META-INF/versions/10` игнорируются. Так, если бы этот MRJAR-файл использовался в JDK 8, то были бы видны лишь классы `C1`, `C2`, `C3` и `C4`.

А в JDK 9 видны были бы пять классов: `C1`, `C2`, `C3`, `C4` и `C5`, причем вместо класса `C2` в корневом каталоге использовался бы класс `C2` из каталога `META-INF/versions/9`. Таким образом, MRJAR говорит, что для JDK 9 существует версия класса `C2`, которая находится в корневом каталоге и переопределяет версию для JDK 8 и младше. В версии для JDK 9 добавился также новый класс `C5`.

Аналогично для JDK 10 переопределены классы `C1` и `C2` и добавлен класс `C6`.

Поскольку формат MRJAR ориентирован на несколько версий JDK, процесс поиска в архиве устроен иначе, чем в обычном JAR-файле. При поиске ресурса или класса в MRJAR-файле действуют следующие правила.

- Определяется основная версия JDK в окружении, где используется MRJAR-файл. Предположим, что основная версия имеет номер `N`.
- Для поиска ресурса или класса с именем `R` просматривается платформенно-зависимый подкаталог каталога `META-INF/versions` для версии `N`.
- Если `R` найден в подкаталоге `N`, то он возвращается. В противном случае поиск продолжается в подкаталогах `META-INF/versions`, соответствующих версиям меньше `N`.
- Если `R` не найден ни в одном из подкаталогов вида `META-INF/versions/N`, то просматривается корневой каталог MRJAR-файла.

Для примера рассмотрим показанную выше структуру MRJAR-файла. Допустим, что программа ищет класс `C3.class` и текущая версия JDK имеет номер 10. Поиск начинается с каталога `META-INF/versions/10`, в котором класса `C3.class` нет. Затем поиск продолжается в каталоге `META-INF/versions/9`, где `C3.class` тоже нет. Далее просматривается корневой каталог, и в нем `C3.class` будет найден.

Теперь предположим, что программа ищет класс `C2.class` на машине, где установлен JDK 10. Поиск начинается с каталога `META-INF/versions/10`, и будет возвращен имеющийся там класс `C2.class`.

Теперь пусть ищется класс `C2.class`, а версия JDK равна 9. Поиск начнется с каталога `META-INF/versions/9`, в нем `C2.class` есть, он и возвращается.

Наконец, предположим, что ищется класс `C2.class`, а версия JDK равна 8. Платформенно-зависимого каталога `META-INF/versions/8` не существует, поэтому поиск начинается с корневого каталога, где `C2.class` присутствует.

Совет. В JDK 9 все команды, работающие с JAR-файлами, в т. ч. `java`, `javac` и `javap` модифицированы и умеют работать с многоверсионными JAR-файлами. API для работы с JAR-файлами также модифицированы.

Создание многоверсионных JAR-файлов

Существует ряд правил, касающихся содержания зависящих от версии JDK каталогов, которые я опишу в последующих разделах. А сейчас остановимся на создании MRJAR-файлов.

Для выполнения этого примера на машине должны быть установлены JDK 8 и JDK 9. Вместо JDK 8 подойдет любая другая версия JDK, только нужно будет внести изменения в код, чтобы он компилировался в этой версии.

Я буду хранить в MRJAR-файле версии приложения для JDK 8 и JDK 9. Приложение состоит из двух классов:

- `com.jdojo.mrjar.Main`
- `com.jdojo.mrjar.TimeUtil`

В классе `Main` создается объект класса `TimeUtil` и вызывается его метод. Класс `Main` может служить главным классом приложения. Класс `TimeUtil` содержит метод `getLocalDate(Instant now)`, который принимает аргумент типа `Instant` и возвращает значение типа `LocalDate`, соответствующее интерпретации этого аргумента в текущем часовом поясе. В JDK 9 в класс `LocalDate` добавлен метод `ofInstant(Instant instant, ZoneId zone)`. В версии приложения для JDK 9 будет использоваться этот метод, а в версии для JDK 8 – прежний Time API.

Исходный код содержит два проекта NetBeans, `com.jdojo.mrjar.jdk8` и `com.jdojo.mrjar.jdk9`, сконфигурированных соответственно под JDK 8 и JDK 9. В листингах 6.1 и 6.2 приведен код классов `TimeUtil` и `Main` для JDK 8. В NetBeans нужно будет задать свойства **Sources** и **Libraries** равными JDK 8 для проекта `com.jdojo.mrjar.jdk8` и JDK 9 – для проекта `com.jdojo.mrjar.jdk9`. Исходный код настолько прост, что не нуждается в пояснениях. Можно было бы сделать метод `getLocalDate()` в классе `TimeUtil` статическим, но я оставил его методом экземпляра, чтобы в выходных данных было видно, экземпляр какой версии класса создан. Класс `Main` печатает текущую локальную дату, которая на вашей машине будет другой.

Листинг 6.1. Класс TimeUtil для JDK 8

```
// TimeUtil.java
package com.jdojo.mrjar;

import java.time.Instant;
import java.time.LocalDate;
import java.time.ZoneId;

public class TimeUtil {
```

```
   public TimeUtil() {
      System.out.println("Создается версия TimeUtil для JDK 8...");
   }
   public LocalDate getLocalDate(Instant now) {
      return now.atZone(ZoneId.systemDefault())
                .toLocalDate();
   }
}
```

Листинг 6.2. Класс Main для JDK 8

```
// Main.java
package com.jdojo.mrjar;

import java.time.Instant;
import java.time.LocalDate;

public class Main {
   public static void main(String[] args) {
      System.out.println("В методе Main.main() для JDK 8...");

      TimeUtil t = new TimeUtil();
      LocalDate ld = t.getLocalDate(Instant.now());
      System.out.println("Местная дата: " + ld);
   }
}
```

```
В методе Main.main() для JDK 8...
Создается версия TimeUtil для JDK 8...
Местная дата: 2017-01-27
```

Все классы для JDK 9 мы поместим в модуль `com.jdojo.mrjar`, объявление которого показано в листинге 6.3. В листингах 6.4. и 6.5 приведен код классов `TimeUtil` и `Main` для JDK 9.

Листинг 6.3. Объявление модуля com.jdojo.mrjar

```
// module-info.java
module com.jdojo.mrjar {
    exports com.jdojo.mrjar;
}
```

Листинг 6.4. Класс TimeUtil для JDK 9

```
// TimeUtil.java
```

```
package com.jdojo.mrjar;

import java.time.Instant;
import java.time.LocalDate;
import java.time.ZoneId;

public class TimeUtil {
   public TimeUtil() {
      System.out.println("Создается версия TimeUtil для JDK 9...");
   }

   public LocalDate getLocalDate(Instant now) {
      return LocalDate.ofInstant(now, ZoneId.systemDefault());
   }
}
```

Листинг 6.5. Класс Main для JDK 9

```
// Main.java
package com.jdojo.mrjar;

import java.time.Instant;
import java.time.LocalDate;

public class Main {
   public static void main(String[] args) {
      System.out.println("В методе Main.main() для JDK 9...");

      TimeUtil t = new TimeUtil();
      LocalDate ld = t.getLocalDate(Instant.now());
      System.out.println("Местная дата: " + ld);
   }
}
```

```
В методе Main.main() для JDK 9...
Создается версия TimeUtil для JDK 9...
Местная дата: 2017-01-27
```

Я показал, что выводится при выполнении класса `Main` в JDK 8 и JDK 9. Но цель этого примера – не выполнить два класса по отдельности, а упаковать их в MRJAR-файл и затем выполнить из этого файла. К этому я скоро и перейду.

В JDK 9 команда `jar` модифицирована и теперь поддерживает создание MRJAR-файлов. Для этого введен новый параметр `--release`:

```
jar <options> --release N <other-options>
```

Здесь N – главная версия JDK, для JDK 9 она равна 9. Значение `N` должно быть больше или равно 9. Все файлы, перечисленные после `--release N`, помещаются в каталог `META-INF/versions/N` внутри MRJAR-файла.

Следующая команда создает MRJAR-файл `com.jdojo.mrjar.jar` в каталоге `C:\Java9Revealed\mrjars`:

```
C:\Java9Revealed>jar --create --file mrjars\com.jdojo.mrjar.jar
 -C com.jdojo.mrjar.jdk8\build\classes .
 --release 9 -C com.jdojo.mrjar.jdk9\build\classes .
```

Обратите внимание на использование параметра `--release 9`. Все файлы из каталога `com.jdojo.mrjar.jdk9\build\classes` будут помещены в каталог `META-INF/versions/9` внутри MRJAR-файла, а все файлы из каталога `com.jdojo.mrjar.jdk8\build\classes` – в корень MRJAR-файла. Содержимое MRJAR-файла будет таким:

```
- jar-root
  - com
    - jdojo
      - mrjar
        - Main.class
        - TimeUtil.class
- META-INF
  - MANIFEST.MF
  - versions
    - 9
      - module-info.class
      - com
        - jdojo
          - mrjar
            - Main.class
            - TimeUtil.class
```

При создании MRJAR-файлов очень полезно задавать параметр `--verbose`. При этом печатается много дополнительной информации, помогающей диагностировать ошибки. Вот что напечатает та же самая команда с параметром `--verbose`. Мы видим какие файлы были скопированы и куда:

```
C:\Java9Revealed>jar --create --verbose --file mrjars\com.jdojo.mrjar.jar
 -C com.jdojo.mrjar.jdk8\build\classes .
 --release 9 -C com.jdojo.mrjar.jdk9\build\classes .
```

```
added manifest
added module-info: META-INF/versions/9/module-info.class
adding: com/(in = 0) (out= 0)(stored 0%)
adding: com/jdojo/(in = 0) (out= 0)(stored 0%)
adding: com/jdojo/mrjar/(in = 0) (out= 0)(stored 0%)
adding: com/jdojo/mrjar/Main.class(in = 1100) (out= 592)(deflated 46%)
adding: com/jdojo/mrjar/TimeUtil.class(in = 884) (out= 503)(deflated 43%)
adding: META-INF/versions/9/(in = 0) (out= 0)(stored 0%)
```

```
adding: META-INF/versions/9/.netbeans_automatic_build(in = 0) (out= 0)(stored 0%)
adding: META-INF/versions/9/.netbeans_update_resources(in = 0) (out= 0)(stored 0%)
adding: META-INF/versions/9/com/(in = 0) (out= 0)(stored 0%)
adding: META-INF/versions/9/com/jdojo/(in = 0) (out= 0)(stored 0%)
adding: META-INF/versions/9/com/jdojo/mrjar/(in = 0) (out= 0)(stored 0%)
adding: META-INF/versions/9/com/jdojo/mrjar/Main.class(in = 1328) (out= 689)(deflated 48%)
adding: META-INF/versions/9/com/jdojo/mrjar/TimeUtil.class(in = 814) (out= 470)
(deflated 42%)
```

Допустим, что мы хотим создать MRJAR-файл для версий JDK 8, 9 и 10. В предположении, что каталог `com.jdojo.mrjar.jdk10\build\classes` содержит классы, специфичные для JDK 10, это сделает следующая команда:

```
C:\Java9Revealed>jar --create --file mrjars\com.jdojo.mrjar.jar
 -C com.jdojo.mrjar.jdk8\build\classes .
 --release 9 -C com.jdojo.mrjar.jdk9\build\classes .
 --release 10 -C com.jdojo.mrjar.jdk10\build\classes .
```

Для проверки содержимого MRJAR-файла можно задать параметр `--list`:

```
C:\Java9Revealed>jar --list --file mrjars\com.jdojo.mrjar.jar
```

```
META-INF/
META-INF/MANIFEST.MF
com/
com/jdojo/
com/jdojo/mrjar/
com/jdojo/mrjar/Main.class
com/jdojo/mrjar/TimeUtil.class
META-INF/versions/9/
META-INF/versions/9/com/
META-INF/versions/9/com/jdojo/
META-INF/versions/9/com/jdojo/mrjar/
META-INF/versions/9/com/jdojo/mrjar/Main.class
META-INF/versions/9/com/jdojo/mrjar/TimeUtil.class
META-INF/versions/9/module-info.class
META-INF/versions/10/
META-INF/versions/10/com/
META-INF/versions/10/com/jdojo/
META-INF/versions/10/com/jdojo/mrjar/
META-INF/versions/10/com/jdojo/mrjar/TimeUtil.class
```

Пусть имеется JAR-файл, содержащий файлы ресурсов и классов для JDK 8, и мы хотим превратить его в MRJAR-файл, добавив ресурсы и классы для JDK 9. Для этого нужно обновить JAR-файл с помощью параметра `--update`. Следующая команда создает JAR-файл, содержащий файлы только для JDK 8:

```
C:\Java9Revealed>jar --create --file mrjars\com.jdojo.mrjar.jar
 -C com.jdojo.mrjar.jdk8\build\classes .
```

А эта команда превращает JAR в MRJAR:

```
C:\Java9Revealed>jar --update --file mrjars\com.jdojo.mrjar.jar
 --release 9 -C com.jdojo.mrjar.jdk9\build\classes .
```

Посмотрим на этот MRJAR-файл в действии. Следующая команда выполняет класс `Main` в пакете `com.jdojo.mrjar`, поместив MRJAR на путь к классам. Класс выполняется в среде JDK 8.

```
C:\Java9Revealed> c:\java8\bin\java -classpath mrjars\com.jdojo.mrjar.jar
 com.jdojo.mrjar.Main
```

```
В методе Main.main() для JDK 8...
Создается версия TimeUtil для JDK 8...
Местная дата: 2017-01-27
```

Как видим, оба класса, `Main` и `TimeUtil`, взяты из корневого каталога MRJAR-файла, потому что JDK 8 не поддерживает формат MRJAR. В следующей команде тот же класс исполняется с помощью задания пути к модулям в среде JDK:

```
C:\Java9Revealed> c:\java9\bin\java --module-path mrjars\com.jdojo.mrjar.jar
 --module com.jdojo.mrjar/com.jdojo.mrjar.Main
```

```
В методе Main.main() для JDK 9...
Создается версия TimeUtil для JDK 9...
Местная дата: 2017-01-27
```

Теперь мы видим, что оба класса взяты из каталога `META-INF/versions/9` MRJAR-файла, потому что JDK 9 поддерживает формат MRJAR и в MRJAR-файле есть версии этих классов для JDK 9.

Давайте немного поэкспериментируем с этим MRJAR-файлом. Создадим MRJAR-файл с таким же содержимым, но без файла `Main.class` в каталоге `META-INF/versions/9`. В действительности в версии JDK 9 изменился только класс `TimeUtil`, поэтому отдельно упаковывать класс `Main` для JDK 9 нет смысла. В JDK 9 можно использовать тот же класс `Main`, что и в JDK 8. Следующая команда упаковывает все то же, что в прошлый раз, за исключением класса `Main` для JDK 9. Получившийся MRJAR-файл назван `com.jdojo.mrjar2.jar`.

```
C:\Java9Revealed>jar --create --verbose --file mrjars\com.jdojo.mrjar2.jar
 -C com.jdojo.mrjar.jdk8\build\classes .
 --release 9
 -C com.jdojo.mrjar.jdk9\build\classes module-info.class
 -C com.jdojo.mrjar.jdk9\build\classes com\jdojo\mrjar\TimeUtil.class
```

Проверить содержимое нового MRJAR-файл можно такой командой:

```
C:\Java9Revealed>jar --list --file mrjars\com.jdojo.mrjar2.jar
```

```
META-INF/
META-INF/MANIFEST.MF
META-INF/versions/9/module-info.class
com/
com/jdojo/
com/jdojo/mrjar/
com/jdojo/mrjar/Main.class
com/jdojo/mrjar/TimeUtil.class
META-INF/versions/9/com/jdojo/mrjar/TimeUtil.class
```

Выполнив класс `Main` в JDK 8, мы получим тот же результат, что и раньше. Но в JDK 9 результат будет иным:

```
C:\Java9Revealed>c:\java9\bin\java --module-path mrjars\com.jdojo.mrjar2.jar
 --module com.jdojo.mrjar/com.jdojo.mrjar.Main
```

```
В методе Main.main() для JDK 8...
Создается версия TimeUtil для JDK 9...
Местная дата: 2017-01-27
```

Как видим, класс `Main` взят из корневого каталога JAR-файла, тогда как `TimeUtil` – из каталога `META-INF/versions/9`.

Правила для многоверсионных JAR-файлов

При создании многоверсионных JAR-файлов нужно придерживаться нескольких правил. Если вы допустите ошибку, то `jar` ругнется. Иногда сообщения об ошибках выглядят непонятно. Лучше всего запускать `jar` с параметром `-verbose`, тогда будет выдана подробная информация об ошибке.

В основе большинства правил лежит один простой факт: формат MRJAR предлагает средства хранения одной версии API библиотеки или приложения для разных версий JDK. Например, MRJAR-файл `jdojo-lib-1.0.jar` может содержать версию 1.0 библиотеки `jdojo-lib`, пользующейся разными API из JDK 8 и JDK 9. Это означает, что MRJAR-файл должен предоставлять один и тот же API (в терминах открытых типов и их открытых членов) при использовании в JDK 8 с путем к классам, в JDK 9 с путем к классам или в JDK 9 с путем к модулям. MRJAR-файл, предоставляющий разные API в среде JDK 8 и JDK 9, нельзя считать допустимым. В следующих разделах описаны конкретные правила.

Модульные многоверсионные JAR-файлы

MRJAR может быть модульным JAR-файлом и в таком случае может содержать дескриптор модуля `module-info.class` в корневом каталоге, в одном или нескольких

версионных каталогах или в обоих местах. Версионные дескрипторы должны совпадать с корневым с несколькими исключениями:

- версионный дескриптор может иметь другие нетранзитивные предложения `requires`, описывающие зависимости от модулей `java.*` и `jdk.*`;
- у разных дескрипторов модулей не может быть различных нетранзитивных предложений `requires`, описывающие зависимости от модулей, не являющихся частью JDK;
- у версионных дескрипторов могут быть другие предложения `uses`.

Эти правила означают, что разрешено изменять детали реализации, но не сам API. Изменение в предложении `requires` для модулей, не входящих в состав JDK, считается изменением API – требуется наличие различных пользовательских модулей для разных версий JDK. Поэтому такое изменение запрещено.

В модульном MRJAR-файле может и не быть дескриптора модуля в корневом каталоге. Именно такую ситуацию мы наблюдали в примерах из предыдущего раздела: в корневом каталоге дескриптора модуля не было, а в каталоге `META-INF/versions/9` был. Это означает, что в одном MRJAR-файле может находиться немодульный код для JDK 8 и модульный для JDK 9.

Модульные многоверсионные JAR-файлы и инкапсуляция

При попытке добавить в версионный каталог новый открытый тип, отсутствующий в корневом каталоге, возникнет ошибка. Предположим, что мы хотим добавить открытый класс `Test` для JDK 9. Если класс `Test` находится в пакете `com.jdojo.mrjar`, то он будет экспортирован модулем и окажется доступен внешнему по отношению к MRJAR-файлу коду. Отметим, что корневой каталог не содержит класса `Test`, т. е. этот MRJAR-файл предлагает различные открытые API для JDK 8 и JDK 9. А это значит, что при добавлении класса `Test` в пакет `com.jdojo.mrjar` для JDK 9 возникнет ошибка.

Продолжая тот же пример, предположим, что мы хотим добавить класс `Test` в пакет `com.jdojo.test` для JDK 9. Отметим, что модуль не экспортирует этот пакет. Если MRJAR-файл находится на пути к модулям, то класс `Test` будет недоступен внешнему коду. И в этом смысле MRJAR-файл предлагает одинаковый открытый API для JDK 8 и JDK 9. Однако нас подстерегает ловушка! Этот MRJAR-файл может находиться и на пути к классам в JDK 9, и тогда класс `Test` оказывается доступен внешнему коду – нарушение инкапсуляции модуля и правила, согласно которому MRJAR-файл должен предоставлять один и тот же открытый API для всех версий JDK. Таким образом, добавление открытого типа к неэкспортированный пакет модуля в MRJAR-файле также запрещено. При попытке сделать это вы увидите сообщение вида:

```
entry: META-INF/versions/9/com/jdojo/test/Test.class, contains a new public class
not found in base entries
invalid multi-release jar file mrjars\com.jdojo.mrjar.jar deleted
```

Иногда для поддержки новой версии JDK в библиотеку необходимо включить дополнительные типы. Это возможно, но нужно сделать их закрытыми или пакетными в версионном каталоге MRJAR-файла. В примере выше класс `Test` для JDK 9 можно было бы добавить, сделав его неоткрытым.

Многоверсионные JAR-файлы и базовый загрузчик

Базовый загрузчик не поддерживает многоверсионные JAR-файлы, например, задание MRJAR-файла в параметре `-Xbootclasspath/a`. Такая поддержка усложнила бы реализацию базового загрузчика ради редко используемой возможности.

Одинаковые файлы для разных версий JDK

Предполагается, что MRJAR-файл содержит различные версии одного и того же файла в версионном каталоге. Файлы ресурсов или классов, которые не зависят от платформы, следует помещать в корневой каталог. В настоящее время команда `jar` выдает предупреждение, если встречает в точности один и тот же элемент в нескольких версионных каталогах.

Посмотрим, как это правило работает. Скопируйте содержимое каталога `com.jdojo.mrjar.jdk9\build\classes` в каталог `com.jdojo.mrjar.jdk10\build\classes`, так чтобы содержимое обоих совпадало. Выполните показанную ниже команду, которая должна создать MRJAR-файл с кодом для версий JDK 8, 9 и 10. Отметим, что файлы в каталогах для версий 9 и 10 будут одинаковы. Печатаемые предупреждения не допускают двояких толкований:

```
C:\Java9Revealed>jar --create --file mrjars\com.jdojo.mrjar.jar
 -C com.jdojo.mrjar.jdk8\build\classes .
 --release 9 -C com.jdojo.mrjar.jdk9\build\classes .
 --release 10 -C com.jdojo.mrjar.jdk10\build\classes .
```

```
Warning: entry META-INF/versions/9/com/jdojo/mrjar/Main.class contains a class that
is identical to an entry already in the jar
Warning: entry META-INF/versions/9/com/jdojo/mrjar/TimeUtil.class contains a class
that is identical to an entry already in the jar
```

Многоверсионные JAR-файлы и URL со схемой JAR

До появления формата MRJAR все ресурсы находились в корневом каталоге JAR-файла. Запрашивая ресурс у загрузчика класса (`ClassLoader.getResource("com/jdojo/mrjar/TimeUtil.class")`), мы получали в ответ URL-адрес вида:

```
jar:file:/C:/Java9Revealed/mrjars/com.jdojo.mrjar.jar! com/jdojo/mrjar/TimeUtil.class
```

В случае MRJAR-файлов может быть возвращен ресурс как из корневого, так и из версионного каталога. Если мы ищем файл `TimeUtil.class` в среде JDK 9, то URL будет таким:

```
jar:file:/C:/Java9Revealed/mrjars/com.jdojo.mrjar.jar!/META-INF/versions/9/com/jdojo/
mrjar/TimeUtil.class
```

Если программа ожидала получить URL ресурса в определенном формате или вы вручную кодировали URL со схемой jar, то в случае MRJAR-файлов результат может оказаться неожиданным. В таком случае следует внести изменения в код.

Атрибут Multi-Release в манифесте

Для MRJAR-файлов в файле MANIFEST.MF имеется специальный атрибут:

```
Multi-Release: true
```

Атрибут Multi-Release добавляется программой jar при построении MRJAR-файлов. Если он равен true, то JAR-файл многоверсионный, а если он равен false или отсутствует, то не многоверсионный. Атрибут находится в главной секции манифеста.

В класс Attributes.Name из пакета java.util.jar добавлена константа MULTI_RELEASE, представляющая атрибут Multi-Release. В программе к ней следует обращаться по имени Attributes.Name.MULTI_RELEASE.

Формат JMOD

В JDK 9 появился новый формат JMOD для упаковки модулей. JMOD-файлы могут содержать больше типов содержимого, чем JAR-файлы. В JMOD-файл можно упаковать платформенный код, конфигурационные файлы, платформенные команды и другие данные. На момент написания книги формат JMOD был основан на ZIP, но в будущем это изменится. Модули JDK 9 упакованы в формате JMOD, допускающем использование на этапах компиляции и сборки, но не на этапе выполнения. Они находятся в каталоге JDK_HOME\jmods, где JDK_HOME – каталог, в который установлен JDK 9. В формате JMOD вы можете упаковать и собственные модули. Файлы в формате JMOD имеют расширение .jmod. Так, платформенный модуль java.base упакован в файл java.base.jmod.

JMOD-файлы могут содержать платформенный код, но извлечь и скомпоновать его «на лету» в процессе выполнения затруднительно. Именно поэтому JMOD-файлы поддерживаются на этапах компиляции и сборки, но не на этапе выполнения.

Программа jmod

В состав JDK 9 вошла новая программа jmod, которая находится в каталоге JDK_HOME\bin. Она служит для создания JMOD-файла, печати его содержимого, печати описания модуля и сохранения хэшей используемых модулей. Порядок ее вызова следующий:

```
jmod <subcommand> <options> <jmod-file>
```

У команды jmod имеются следующие подкоманды:

- create
- extract
- list
- describe
- hash

Подкоманды `list` и `describe` не имеют параметров. `<jmod-file>` – имя создаваемого или существующего JMOD-файла. В табл. 6.1 перечислены параметры команды `jmod`.

Таблица 6.1. Параметры команды jmod

Параметр	Описание
`--class-path <path>`	Задает путь к классам, на котором следует искать подлежащие упаковке классы. `<path>` может быть списком путей к JAR-файлам или каталогам, содержащим классы приложения. Найденные файлы копируются в JMOD-файл.
`--cmds <path>`	Задает список каталогов, содержащих платформенные команды, которые нужно скопировать в JMOD-файл.
`--config <path>`	Задает список каталогов, содержащих редактируемые пользователем конфигурационные файлы, которые нужно скопировать в JMOD-файл.
`--dir <path>`	Задает целевой каталог, в который будет извлечено содержимое JMOD-файла.
`--do-not-resolve-by-default`	Если JMOD-файл создается в таком режиме, то содержащийся в нем модуль, будет исключен из набора корневых модулей по умолчанию. Чтобы разрешить такой модуль, его необходимо добавить в набор корневых, задав в командной строке параметр `--add-modules`.
`--dry-run`	Хэширование модулей без сохранения результата. В этом режиме хэши модулей вычисляются и печатаются, но не записываются в JMOD-файл.
`--exclude <pattern-list>`	Исключить файлы с именами, соответствующими перечисленным через запятую образцам. Каждый элемент списка образцов должен иметь вид `<glob-pattern>`, `glob:<glob-pattern>` или `regex:<regex-pattern>`.
`--hash-modules <regex-pattern>`	Вычисляет и записывает хэши, чтобы связать упакованный модуль с модулями, соответствующими заданной маске `<regex-pattern>` и зависящими от него прямо или косвенно. Хэши записываются в создаваемый JMOD-файл либо в JMOD-файл или модульный JAR-файл на пути к модулям, заданном в команде `jmod hash`.
`--help, -h`	Печатает справку о порядке использования с перечислением всех параметров команды `jmod`.
`--header-files <path>`	Задает список путей `<path>` к платформенным файлам-заголовкам, которые нужно скопировать в JMOD-файл.
`--help-extra`	Печатает справку по дополнительным параметрам команды `jmod`.
`--legal-notices <path>`	Задает местоположение правовых уведомлений, копируемых в JMOD-файл.
`--libs <path>`	Задает список каталогов, содержащих платформенные библиотеки, копируемые в JMOD-файл.

Параметр	Описание
`--main-class <class-name>`	Задает главный класс для запуска приложения.
`--man-pages <path>`	Задает местоположение страниц руководства.
`--module-version <version>`	Задает версию модуля, записываемую в файл `module-info.class`.
`--module-path <path>`, `-p <path>`	Задает путь к модулям, для которых нужно вычислить хэши.
`--os-arch <os-arch>`	Задает архитектуру операционной системы, записываемую в файл `module-info.class`.
`--os-name <os-name>`	Задает название операционной системы, записываемое в файл `module-info.class`.
`--version`	Печатает версию команды `jmod`.
`--warn-if-resolved <reason>`	Говорит `jmod`, что нужно печатать предупреждение, если разрешен модуль, объявленный нерекомендуемым, запланированный для удаления или помещенный в инкубатор. Параметр `<reason>` может принимать одно из трех значений: `deprecated`, `deprecated-for-removal`, `incubating`.
`@<filename>`	Читать параметры из указанного файла.

В следующих разделах подробно объясняется, как использовать команду `jmod`. Все команды, описываемые в этой главе, следует вводить на одной строке, хотя иногда они занимают несколько строк в тексте книги.

Создание JMOD-файлов

Для создания JMOD-файла служит подкоманда `create` команды `jmod`. Содержимым JMOD-файла является содержимое модуля. Предположим, что существуют следующие каталоги и файлы:

```
C:\Java9Revealed\jmods
C:\Java9Revealed\lib\com.jdojo.prime.jar
```

Тогда следующая команда создаст файл `com.jdojo.prime.jmod` в каталоге `C:\Java9Revealed\jmods`. Его содержимое копируется из файла `com.jdojo.prime.jar`.

```
C:\Java9Revealed>jmod create --class-path lib\com.jdojo.prime.jar
jmods\com.jdojo.prime.jmod
```

Обычно содержимое JMOD-файла берется из каталогов, содержащих откомпилированный код модуля. Следующая команда создает файл `com.jdojo.prime.jmod`, содержимое которого берется из каталога `mods\com.jdojo.prime`. Параметр `--module-version` задает версию модуля, записываемую в файл `module-info.class`, который помещается в каталог `com.jdojo.prime\build\classes`. Не забудьте удалить JMOD-файл, созданный на предыдущем шаге.

```
C:\Java9Revealed>jmod create --module-version 1.0
 --class-path com.jdojo.prime\build\classes jmods\com.jdojo.prime.jmod
```

Что можно сделать с этим JMOD-файлом? Можно поместить его на путь к модулям для использования на этапе компиляции. Можно указать в команде `jlink` для создания пользовательского образа среды выполнения, в которой будет запускаться приложение. Напомним, что использовать JMOD-файл на этапе выполнения нельзя. При попытке сделать это, поместив его на путь к модулям, будет выдано такое сообщение об ошибке:

```
Error occurred during initialization of VM
java.lang.module.ResolutionException: JMOD files not supported: jmods\com.jdojo.
prime.jmod
...
```

Извлечение содержимого JMOD-файла

Для извлечения содержимого JMOD-файла служит подкоманда `extract`. Следующая команда извлекает содержимое файла `jmods\com.jdojo.prime.jmod` в каталог `extracted`.

```
C:\Java9Revealed>jmod extract --dir extracted jmods\com.jdojo.prime.jmod
```

Если параметр `--dir` не задан, то содержимое извлекается в текущий каталог.

Печать содержимого JMOD-файла

Для печати имен элементов JMOD-файла служит подкоманда `list`. Следующая команда печатает содержимое созданного ранее файла `com.jdojo.prime.jmod`:

```
C:\Java9Revealed>jmod list jmods\com.jdojo.prime.jmod
classes/module-info.class
classes/com/jdojo/prime/PrimeChecker.class
```

Следующая команда печатает содержимое модуля `java.base`, поставляемого в виде JMOD-файла `java.base.jmod`. Предполагается, что JDK 9 установлен в каталог `C:\java9`. Вывод команды занимает свыше 120 страниц, поэтому показана только его часть. Отметим, что содержимое разных типов хранится в разных каталогах внутри JMOD-файла.

```
C:\Java9Revealed>jmod list C:\java9\jmods\java.base.jmod
```

```
classes/module-info.class
classes/java/nio/file/WatchEvent.class
classes/java/nio/file/WatchKey.class
bin/java.exe
bin/javaw.exe
native/amd64/jvm.cfg
native/java.dll
conf/net.properties
conf/security/java.policy
```

```
conf/security/java.security
...
```

Описание JMOD-файла

Для печати описания модуля, содержащегося в JMOD-файле, служит подкоманда `describe`. Следующая команда печатает описание модуля в файле `com.jdojo.prime.jmod`:

```
C:\Java9Revealed>jmod describe jmods\com.jdojo.prime.jmod
```

```
com.jdojo.prime@1.0
requires mandated java.base
uses com.jdojo.prime.PrimeChecker
exports com.jdojo.prime
```

Эта команда позволяет также получить описание платформенных модулей. Следующая команда печатает описание модуля в файле `java.sql.jmod` в предположении, что JDK 9 установлен в каталог `C:\java9`:

```
C:\Java9Revealed>jmod describe C:\java9\jmods\java.sql.jmod
```

```
java.sql@9-ea
requires mandated java.base
requires transitive java.logging
requires transitive java.xml
uses java.sql.Driver
exports java.sql
exports javax.sql
exports javax.transaction.xa
operating-system-name Windows
operating-system-architecture amd64
```

Запись хэшей модулей

Для записи в файл `module-info.class`, хранящийся в JMOD-файле данного модуля, хэшей других модулей служит подкоманда `hash`. Впоследствии хэши используются для проверки зависимостей. Предположим, что в четырех JMOD-файлах находятся четыре модуля:

- `com.jdojo.prime`
- `com.jdojo.prime.generic`
- `com.jdojo.prime.faster`
- `com.jdojo.prime.client`

И предположим, что вы хотите поставить эти модули клиентам, гарантировав, что код модулей не изменен. Для этой цели можно записать хэши модулей `com.jdojo.prime.generic`, `com.jdojo.prime.faster` и `com.jdojo.prime.client` в модуль `com.jdojo.prime`. Посмотрим, как это делается.

Чтобы вычислить хэши других модулей, команда `jmod` должна эти модули найти. Поэтому следует указать путь к модулям с помощью параметра `--module-path`. Кроме того, с помощью параметра `--hash-modules` нужно указать список масок, с которыми сравниваются имена модулей, чьи хэши мы собираемся записывать.

> **Совет.** Параметры `--hash-modules` и `--module-path` можно также задавать в команде `jar` для записи хэшей зависимостей при упаковке модуля в модульный JAR-файл.

Выполним следующие четыре команды для создания JMOD-файлов всех четырех модулей. Обратите внимание, что при создании файла `com.jdojo.prime.client.jmod` задан параметр `--main-class`. Я еще воспользуюсь этим файлом в главе 7 при обсуждении команды `jlink`. Если при попытке выполнить эти команды будет выдано сообщение «file already exists» (файл уже существует), то удалите существующий JMOD-файл из каталога `jmods` и выполните команду повторно.

```
C:\Java9Revealed>jmod create --module-version 1.0
 --class-path com.jdojo.prime\build\classes jmods\com.jdojo.prime.jmod

C:\Java9Revealed>jmod create --module-version 1.0
 --class-path com.jdojo.prime.generic\build\classes
 jmods\com.jdojo.prime.generic.jmod

C:\Java9Revealed>jmod create --module-version 1.0
 --class-path com.jdojo.prime.faster\build\classes
 jmods\com.jdojo.prime.faster.jmod

C:\Java9Revealed>jmod create --main-class com.jdojo.prime.client.Main
 --module-version 1.0
 --class-path com.jdojo.prime.client\build\classes
 jmods\com.jdojo.prime.client.jmod
```

Теперь можно записать в модуль `com.jdojo.prime` хэши всех модулей, имена которых начинаются строкой `com.jdojo.prime.`:

```
C:\Java9Revealed>jmod hash --module-path jmods
 --hash-modules com.jdojo.prime.? jmods\com.jdojo.prime.jmod

Hashes are recorded in module com.jdojo.prime
```

Посмотрим, какие хэши записаны в модуль `com.jdojo.prime`. Следующая команда печатает описание модуля `com.jdojo.prime` вместе с записанными хэшами:

```
C:\Java9Revealed>jmod describe jmods\com.jdojo.prime.jmod
```

```
com.jdojo.prime@1.0
  requires mandated java.base
  uses com.jdojo.prime.PrimeChecker
  exports com.jdojo.prime
  hashes com.jdojo.prime.client SHA-256
2ffb0d4413501e389d6712450bd138bbe82ca8abeb4e8b5d29b0c307d90a2e91
  hashes com.jdojo.prime.faster SHA-256
687e07c429080c48bed89a649dca20fa26dc28fab88a4905f1b5070560622a0c
  hashes com.jdojo.prime.generic SHA-256
f24556ef69c4345ad7a8e5e59d31ea2d52c8749714ede0c0dedf128255450708
```

Записать хэши других модулей можно и при создании нового JMOD-файла с помощью подкоманды `create`. В предположении, что все три модуля `com.jdojo.prime.generic`, `com.jdojo.prime.faster` и `com.jdojo.prime.client` находятся на пути к модулям, следующая команда создает файл `com.jdojo.prime.jmod`, в который записываются их хэши:

```
C:\Java9Revealed>jmod create --module-version 1.0
 --module-path jmods
 --hash-modules com.jdojo.prime.?
 --class-path com.jdojo.prime\build\classes jmods\com.jdojo.prime.jmod
```

Процесс хэширования можно запустить «вхолостую» – хэши печатаются, но не записываются в JMOD-файл. Этот режим полезен, когда нужно убедиться в правильности настроек без создания JMOD-файла. В описанной ниже последовательности команд продемонстрирован этот процесс. Сначала удалите файл `jmods\com.jdojo.prime.jmod`, созданный на предыдущем шаге.

Следующая команда создает файл `jmods\com.jdojo.prime.jmod`, не записывая в него хэши других модулей:

```
C:\Java9Revealed>jmod create --module-version 1.0
 --module-path jmods
 --class-path com.jdojo.prime\build\classes jmods\com.jdojo.prime.jmod
```

Следующая команда выполняет подкоманду `hash` в холостом режиме. Вычисляются и печатаются хэши модулей, имена которых соответствуют регулярному выражению, заданному в параметре `--hash-modules option`. Но в файл `jmods\com.jdojo.prime.jmod` хэши не записываются.

```
C:\Java9Revealed>jmod hash --dry-run --module-path jmods
--hash-modules com.jdojo.prime.? jmods\com.jdojo.prime.jmod
```

```
Dry run:
com.jdojo.prime
  hashes com.jdojo.prime.client SHA-256
2ffb0d4413501e389d6712450bd138bbe82ca8abeb4e8b5d29b0c307d90a2e91
  hashes com.jdojo.prime.faster SHA-256
```

```
687e07c429080c48bed89a649dca20fa26dc28fab88a4905f1b5070560622a0c
  hashes com.jdojo.prime.generic SHA-256
f24556ef69c4345ad7a8e5e59d31ea2d52c8749714ede0c0dedf128255450708
```

Следующая команда проверяет, что предыдущая действительно не записала хэши в JMOD-файл:

```
C:\Java9Revealed>jmod describe jmods\com.jdojo.prime.jmod
```

```
com.jdojo.prime@1.0
  requires mandated java.base
  uses com.jdojo.prime.PrimeChecker
  exports com.jdojo.prime
```

Резюме

В JDK 9 поддерживается четыре формата упаковки модулей: развернутые каталоги, JAR-файлы, JMOD-файлы и JIMAGE-файлы. Формат JAR модифицирован и теперь поддерживает модульные и многоверсионные JAR-файлы. В многоверсионный JAR-файл можно упаковать одну версию библиотеки или приложения для различных версий JDK. Например, MRJAR-файл может содержать версию 1.2 библиотеки в вариантах для JDK 8 и JDK 9. Если такой файл используется в JDK 8, то будет взять код библиотеки для JDK 8, а если в JDK 9 – то для JDK 9. Файлы, специфичные для JDK версии `N`, хранятся в каталоге `META-INF\versions\N` многоверсионного JAR-файла. Файлы, общие для всех версий JDK, хранятся в корневом каталоге. В тех средах, где многоверсионные JAR-файлы не поддерживаются, они трактуются как обычные JAR-файлы. Порядок поиска файла в обычных и многоверсионных JAR-файлах различен – сначала производится поиск во всех версионных каталогах, начиная с номера главной версии текущей платформы, а только потом в корневом каталоге.

JMOD-файлы могут содержать больше типов содержимого, чем JAR-файлы. В них можно упаковать платформенный код, конфигурационные файлы, платформенные команды и данные других типов. На момент написания книги формат JMOD был основан на ZIP, но в будущем это изменится. Модули JDK 9 упакованы в формате JMOD, допускающем использование на этапах компиляции и сборки, но не на этапе выполнения. Для работы с JMOD-файлами предназначена команда `jmod`.

Глава 7

Создание пользовательских образов среды выполнения

Краткое содержание главы:

- что такое пользовательский образ среды выполнения и формат JIMAGE;
- создание пользовательского образа среды выполнения с помощью программы `jlink`;
- задание в пользовательском образе имени команды, запускающей приложение;
- плагины для `jlink`.

Что такое пользовательский образ среды выполнения?

До JDK 9 образ среды выполнения Java существовал только в виде гигантского монолита, и это увеличивало время скачивания, время инициализации и объем потребляемой памяти. Из-за монолитности JRE было невозможно использовать Java на устройствах с небольшой памятью. При развертывании Java-приложений в облаке вы платите за объем используемой памяти, а, поскольку чаще всего монолитная JRE потребляет больше памяти, чем необходимо, то вы платите больше за облачную службу. Компактные профили, появившиеся в Java 8 стали шагом в направлении уменьшения размера JRE, а, значит, и потребляемой памяти, поскольку дали возможность упаковать в пользовательскую среду выполнения подмножество JRE – так называемый *компактный профиль*.

В Java 9 принят комплексный подход к упаковке образов среды выполнения. Весь платформенный код разбит на модули. Код приложения также упаковывается в виде модулей. Можно создать пользовательскую среду выполнения, содержащую модули вашего приложения и только те платформенные модули, которые ему нужны. В образ среды выполнения можно включить команды операционной системы. Еще одно преимущество состоит в том, что пользователям приложения нужно скачать только один файл, а не скачивать отдельно JRE, чтобы запустить приложение.

Образ среды выполнения хранится в специальном формате JIMAGE, оптимизированном с точки зрения памяти и быстродействия. Формат JIMAGE поддерживается только на этапе выполнения. Это контейнер, в котором хранятся индексированные модули, классы и ресурсы JDK. Поиск и загрузка классов из JIMAGE-файла производится гораздо быстрее, чем из файлов в формате JAR и JMOD. Формат JIMAGE является внутренним для JDK и разработчикам нечасто приходится иметь с ним дело напрямую.

Ожидается, что формат JIMAGE со временем будет эволюционировать, поэтому его внутреннее устройство не раскрывается. В состав JDK 9 входит команда `jimage`, позволяющая получать информацию о JIMAGE-файлах. Она будет рассмотрена в отдельном разделе этой главы.

Хочу предупредить тех, кто ожидает, что образ среды выполнения хранится в файле с именем `rt.jar`. До выхода JDK 9 так оно и было, но сейчас это не так. Если ваше приложение зависит от этого соглашения, то при переходе на JDK 9 оно может перестать работать.

Создание пользовательского образа среды выполнения

Для создания пользовательского образа среды выполнения служит команда `jlink`. Образ содержит указанные модули приложения и только необходимые системные модули, благодаря чему его размер уменьшается. Это полезно, если приложение должно работать на встраиваемых устройствах с небольшим объемом памяти. Команда `jlink` находится в каталоге `JDK_HOME\bin` и запускается следующим образом:

```
jlink <options> --module-path <modulepath> --add-modules <mods> --output <path>
```

Здесь `<options>` – необязательные параметры, перечисленные в табл. 7.1, а `<modulepath>` – путь к модулям, на котором находятся платформенные и прикладные модули, включаемые в образ. Модули могут быть представлены модульными JAR-файлами, развернутыми каталогами или JMOD-файлами. `<mods>` – список включаемых в образ модулей, в него могут быть добавлены дополнительные модули из-за транзитивных зависимостей. `<path>` – путь к каталогу, в который будет записан сгенерированный образ.

Таблица 7.1. Параметры команды jlink

Параметр	Описание
`--add-modules <mod>,<mod>...`	Задает список корневых модулей, подлежащих разрешению. Все разрешенные модули будут включены в образ среды выполнения.
`--bind-services`	Произвести полное связывание служб в процессе компоновки. Если включаемый модуль содержит предложения `uses`, то `jlink` просмотрит все JMOD-файлы на пути к модулям и включит в образ все модули, поставляющие службы, указанные в этих предложениях.

Параметр	Описание
`-c, --compress <0|1|2>[:filter=<patternlist>]`	Задает уровень сжатия ресурсов в выходном образе. 0 – разделение константных строк, 1 – сжатие методом ZIP, 2 – то и другое. Можно задать необязательный фильтр `<pattern-list>`, в котором перечислены маски сжимаемых файлов.
`--disable-plugin <plugin-name>`	Отключить указанный плагин.
`--endian <little|big>`	Задает порядок байтов в сгенерированном образе среды выполнения. По умолчанию подразумевается порядок на текущей платформе.
`-h,--help`	Напечатать информацию о порядке вызова и всех параметрах `jlink`.
`--ignore-signing-information`	Не считать фатальной ошибкой включение в образ подписанных модульных JAR-файлов. Сигнатуры файлов, связанных с таким JAR-файлом, не копируются в образ.
`--launcher <command>=<module>`	Задает команду запуска модуля. `<command>` – имя команды, которая должна запускать приложение, например, `runmyapp`. Будет создан скрипт или пакетный файл с именем `<command>` для выполнения главного класса в модуле `<module>`.
`--launcher <command>=<module>` `/<main-class>`	Задает команду запуска модуля и главный класс. `<command>` — имя команды, которая должна запускать приложение, например, `runmyapp`. Будет создан скрипт или пакетный файл с именем `<command>` для выполнения класса `<main-class>` в модуле `<module>`.
`--limit-modules <mod>[,<mod>...]`	Считать видимыми только модули, принадлежащие транзитивному замыканию поименованных модулей `mod`, плюс главный модуль, если он задан, а также дополнительные модули, заданные параметром `--add-modules`.
`--list-plugins`	Вывести список доступных плагинов.
`-p, --module-path <modulepath>`	Задает путь к модулям, на котором ищутся платформенные и прикладные модули, включаемые в образ.
`--no-header-files`	Не включать файлы-заголовки для платформенного кода.
`--no-man-pages`	Не включать страницы руководства.

Параметр	Описание
`--output <path>`	Задает каталог, в который будет записан образ среды выполнения.
`--save-opts <filename>`	Сохранить параметры `jlink` в указанном файле.
`-G, --strip-debug`	Удалить из выходного образа отладочную информацию.
`--suggest-providers [<service-name>,...]`	Если имя службы не задано, то предлагаются имена поставщиков всех служб, связываемых с включаемыми модулями. Если задано одно или несколько имен служб, то предлагаются поставщики именно этих служб. Этот параметр можно использовать до создания образа, чтобы узнать, какие службы будут включены при задании параметра `--bind-services`.
`-v, --verbose`	Печатать подробную информацию.
`--version`	Вывести версию `jlink`.
`@<filename>`	Читать параметры из указанного файла.

Создадим образ среды выполнения, содержащий четыре модуля приложения для проверки на простоту и необходимые ему платформенные модули, т. е. только `java.base`. Обратите внимание, что в показанной ниже команде перечислены только три модуля приложения, а четвертый будет добавлен, потому что остальные три зависят от него.

```
C:\Java9Revealed>jlink --module-path jmods;C:\java9\jmods
 --add-modules com.jdojo.prime.client,com.jdojo.prime.generic,com.jdojo.prime.faster
 --launcher runprimechecker=com.jdojo.prime.client
 --output primechecker
```

Прежде чем приступать к подробному объяснению параметров, проверим, что образ успешно задан. Предполагается, что команда скопирует образ среды выполнения в каталог `C:\Java9Revealed\primechecker`. Выполните следующую команду, чтобы проверить, что образ действительно содержит пять модулей:

```
C:\Java9Revealed>primechecker\bin\java --list-modules
```

```
com.jdojo.prime@1.0
com.jdojo.prime.client@1.0
com.jdojo.prime.faster@1.0
com.jdojo.prime.generic@1.0
java.base@9-ea
```

Если вы получили примерно такой результат, то образ создан правильно. Номера версий модулей после знака `@` могут отличаться.

В параметре `--module-path` задано два каталога, `jmods` и `C:\java9\jmods`. Я сохранил все четыре JMOD-файла приложения в каталоге `C:\Java9Revealed\jmods`. Первый элемент пути к модулям позволит `jlink` найти все модули приложения. А поскольку JDK 9 на моей машине установлен в каталог `C:\java9`, то второй элемент позволит `jlink` найти платформенные модули. Если опустить второй элемент, то будет выдано сообщение:

```
Module java.base not found.
```

Параметр `--add-modules` задает три модуля приложения для проверки на простоту. Возникает вопрос, почему не указан четвертый модуль, `com.jdojo.prime`. Дело в том, что этот список содержит только корневые модули, а не все включаемые в образ. Команда `jlink` транзитивно разрешает зависимости корневых модулей и включает найденные таким образом модули в образ. Все три модуля зависят от `com.jdojo.prime`, который будет разрешен, т. к. находится на пути к модулям и включен в образ. Образ будет также содержать модуль `java.base`, потому что все модули приложения неявно зависят от него.

Параметр `-output` задает каталог, куда будет записан образ, в данном случае `C:\Java9Revealed\primechecker`. Выходной каталог содержит следующие подкаталоги и файл `release`:

- `bin`
- `conf`
- `include`
- `legal`
- `lib`

В каталоге `bin` находятся исполняемые файлы. В Windows там же находятся динамически загружаемые библиотеки (dll-файлы).

В каталоге `conf` находятся редактируемые конфигурационные файлы, в частности, с расширениями `.properties` и `.policy`.

В каталоге `include` находятся файлы-заголовки для программ на C/C++.

В каталоге `legal` находятся правовые уведомления.

В каталоге `lib` находятся, в частности, модули, включенные в образ. На платформах Mac, Linux и Solaris здесь же находятся системные динамически загружаемые библиотеки, содержащие платформенный код.

В приведенной выше команде `jlink` был задан параметр `-launcher` с именем команды `runprimechecker` и указано имя модуля `com.jdojo.prime.client`. В результате `jlink` создаст в каталоге `bin` платформенно-зависимый исполняемый файл, например `runprimechecker.bat` в случае Windows. Этот файл позволяет запустить наше приложение. Он представляет собой простую обертку для выполнения главного класса в указанном модуле:

```
C:\Java9Revealed> primechecker\bin\runprimechecker

Используется jdojo.generic.primechecker:
3 простое.
4 не простое.
121 не простое.
```

```
977 простое.
Используется jdojo.faster.primechecker:
3 простое.
4 не простое.
121 не простое.
977 простое.
A PrimeChecker service provider with the name 'jdojo.probable.primechecker' was not found.
```

Для запуска приложения можно также использовать команду `java`, которая скопирована программой `jlink` в каталог `bin`:

```
C:\Java9Revealed>primechecker\bin\java --module com.jdojo.prime.client
```

Эта команда выводит то же самое, что предыдущая. Отметим, что путь к модулям задавать необязательно. Команда `jlink` позаботилась об этом на стадии создания образа среды выполнения. Также отметим, что мы не задавали имя главного класса приложения, а указали только имя модуля. Мы уже сконфигурировали атрибут `main-class` в модуле `com.jdojo.prime.client`. Когда модуль выполняется без указания главного класса, в качестве такового используется значение атрибута `main-class` в файле `module-info.class` внутри этого модуля.

Связывание служб

В предыдущем разделе мы создали образ среды выполнения для клиента службы простых чисел. В параметре `--add-modules` нам пришлось задать имена всех модулей-поставщиков служб, включаемых в образ. В этом разделе я покажу, как автоматически связывать службы в процессе создания образа с помощью параметра `--bind-services`. Для этого следует добавить модуль `com.jdojo.prime` в граф модулей, а `jlink` позаботится обо всем остальном. Модуль `com.jdojo.prime.client` читает `com.jdojo.prime`, поэтому добавление первого в граф модулей приведет к разрешению второго. Следующая команда печатает список поставщиков службы, предлагаемых для включения в образ. Показана только часть распечатки.

```
C:\Java9Revealed>jlink --module-path jmods;C:\java9\jmods
 --add-modules com.jdojo.prime.client
 --suggest-providers
```

```
module com.jdojo.prime located (file:///C:/Java9Revealed/jmods/com.jdojo.prime.jmod)
    uses com.jdojo.prime.PrimeChecker
module com.jdojo.prime.client located (file:///C:/Java9Revealed/jmods/com.jdojo.prime.
client.jmod)
module java.base located (file:///C:/java9/jmods/java.base.jmod)
    uses java.lang.System$LoggerFinder
    uses java.net.ContentHandlerFactory
...

Suggested providers:
```

```
  module com.jdojo.prime.faster provides com.jdojo.prime.PrimeChecker, used by com.
jdojo.prime
  module com.jdojo.prime.generic provides com.jdojo.prime.PrimeChecker, used by com.
jdojo.prime
  module com.jdojo.prime.probable provides com.jdojo.prime.PrimeChecker, used by com.
jdojo.prime
  module java.desktop provides java.net.ContentHandlerFactory, used by java.base
...
```

В параметре `--add-modules` задан только модуль `com.jdojo.prime.client`. А модули `com.jdojo.prime` и `java.base` разрешены, потому что их читает модуль `com.jdojo.prime.client`. Для каждого разрешенного модуля команда смотрит, есть ли в его объявлении предложение `uses`, затем все модули на пути к модулям анализируются на предмет предоставления служб, упомянутых в предложениях `uses`. Все найденные поставщики служб печатаются.

> **Совет.** У параметра `--suggest-providers` могут быть аргументы. Если параметр употребляется без аргументов, то он должен быть последним в команде, иначе параметр, следующий за `--suggest-providers`, будет интерпретирован как аргумент, и вы получите сообщение об ошибке.

В следующей команде `com.jdojo.prime.PrimeChecker` задано в качестве имени службы в параметре `--suggest-providers`, чтобы команда напечатала всех поставщиков этой службы:

```
C:\Java9Revealed>jlink --module-path jmods;C:\java9\jmods
 --add-modules com.jdojo.prime.client
 --suggest-providers com.jdojo.prime.PrimeChecker
```

```
Suggested providers:
  module com.jdojo.prime.faster provides com.jdojo.prime.PrimeChecker, used by com.
jdojo.prime
  module com.jdojo.prime.generic provides com.jdojo.prime.PrimeChecker, used by com.
jdojo.prime
  module com.jdojo.prime.probable provides com.jdojo.prime.PrimeChecker, used by com.
jdojo.prime
```

Следуя уже описанной логике, команда находит всех трех поставщиков службы. Создадим новый образ, включающий всех трех поставщиков. Это делает такая команда:

```
C:\Java9Revealed>jlink --module-path jmods;C:\java9\jmods
 --add-modules com.jdojo.prime.client
 --launcher runprimechecker=com.jdojo.prime.client
 --bind-services
 --output primecheckerservice
```

Сравните эту команду с командой из предыдущего раздела. На этот раз мы задали только один модуль в параметре `--add-modules`. То есть задавать имена поставщиков служб не понадобилось. Поскольку мы указали параметр `--bind-services`, все поставщики служб, на которых есть ссылки во включенных модулях, автоматически включаются в образ среды выполнения. Мы задали другой выходной каталог, `primecheckerservice`. Выполним вновь созданный образ:

```
C:\Java9Revealed>primecheckerservice\bin\runprimechecker
```

```
Используется jdojo.generic.primechecker:
3 простое.
4 не простое.
121 не простое.
977 простое.
Используется jdojo.faster.primechecker:
3 простое.
4 не простое.
121 не простое.
977 простое.
Используется jdojo.probable.primechecker:
3 простое.
4 не простое.
121 не простое.
977 простое.
```

Это доказывает, что все три поставщика службы простых чисел, находящиеся на пути к модулям, автоматически включены в образ среды выполнения.

Плагины команды jlink

В команде `jlink` применена архитектура плагинов. Она собирает все классы, платформенные библиотеки и конфигурационные файлы в множество ресурсов и строит конвейер преобразователей, являющихся плагинами, заданными в виде параметров командной строки. Ресурсы подаются на вход конвейера. Каждый преобразователь применяет к ресурсам некоторое преобразование, и преобразованные ресурсы поступают следующему преобразователю. В самом конце `jlink` передает преобразованные ресурсы построителю образа.

В комплект поставки JDK 9 входит несколько плагинов для `jlink`. Им соответствуют параметры командной строки. Чтобы воспользоваться плагином, нужно указать соответствующий ему параметр. При запуске с параметром `--list-plugins` команда `jlink` печатает список всех имеющихся плагинов с описаниями и соответствующими параметрами командной строки:

```
C:\Java9Revealed>jlink --list-plugins
```

```
List of available plugins:

Plugin Name: class-for-name
```

```
Option: --class-for-name
Description: Class optimization: convert Class.forName calls to constant loads.

Plugin Name: compress
Option: --compress=<0|1|2>[:filter=<pattern-list>]
Description: Compress all resources in the output image.
Level 0: constant string sharing
Level 1: ZIP
Level 2: both.
An optional <pattern-list> filter can be specified to list the pattern of
files to be included.

Plugin Name: dedup-legal-notices
Option: --dedup-legal-notices=[error-if-not-same-content]
Description: De-duplicate all legal notices. If error-if-not-same-content is
specified then it will be an error if two files of the same filename
are different.

Plugin Name: exclude-files
Option: --exclude-files=<pattern-list> of files to exclude
Description: Specify files to exclude. e.g.: **.java,glob:/java.base/native/client/**

Plugin Name: exclude-jmod-section
Option: --exclude-jmod-section=<section-name>
where <section-name> is "man" or "headers".
Description: Specify a JMOD section to exclude

Plugin Name: exclude-resources
Option: --exclude-resources=<pattern-list> resources to exclude
Description: Specify resources to exclude. e.g.: **.jcov,glob:**/META-INF/**

Plugin Name: generate-jli-classes
Option: --generate-jli-classes=@filename
Description: Takes a file hinting to jlink what java.lang.invoke classes to pre-generate.
If this flag is not specified a default set of classes will be generated.

Plugin Name: include-locales
Option: --include-locales=<langtag>[,<langtag>]*
Description: BCP 47 language tags separated by a comma, allowing locale matching
defined in RFC 4647. e.g.: en,ja,*-IN

Plugin Name: order-resources
Option: --order-resources=<pattern-list> of paths in priority order. If a @file
is specified, then each line should be an exact match for the path to be ordered
Description: Order resources. e.g.: **/module-info.class,@classlist,/java.base/java/lang/**

Plugin Name: release-info
```

```
Option: --release-info=<file>|add:<key1>=<value1>:<key2>=<value2>:...|del:<key list>
Description: <file> option is to load release properties from the supplied file.
add: is to add properties to the release file.
Any number of <key>=<value> pairs can be passed.
del: is to delete the list of keys in release file.

Plugin Name: strip-debug
Option: --strip-debug
Description: Strip debug information from the output image

Plugin Name: strip-native-commands
Option: --strip-native-commands
Description: Exclude native commands (such as java/java.exe) from the image

Plugin Name: system-modules
Option: --system-modules
Description: Fast loading of module descriptors (always enabled)

Plugin Name: vm
Option: --vm=<client|server|minimal|all>
Description: Select the HotSpot VM in the output image. Default is all
For options requiring a <pattern-list>, the value will be a comma separated
list of elements each using one the following forms:
  <glob-pattern>
  glob:<glob-pattern>
  regex:<regex-pattern>
  @<filename> where filename is the name of a file containing patterns to be
              used, one pattern per line
```

В следующей команде используются плагины `compress` и `strip-debug`. Плагин `compress` сжимает образ, уменьшая его размер. Я задаю степень сжатия 2, при которой сжатие максимально. Плагин `strip-debug` удаляет отладочную информацию из Java-программы, чем дополнительно уменьшает размер образа. Не забудьте удалить созданный ранее каталог `primechecker`, прежде чем выполнять эту команду:

```
C:\Java9Revealed>jlink --module-path jmods;C:\java9\jmods
 --compress 2
 --strip-debug
 --add-modules com.jdojo.prime.client,com.jdojo.prime.generic,com.jdojo.prime.faster
 --launcher runprimechecker=com.jdojo.prime.client
 --output primechecker
```

Совет. На момент написания книги API плагинов еще был экспериментальным и порядок выполнения плагинов не был определен. На ранней стадии реализации команда `jlink` поддерживала пользовательские плагины, но впоследствии от этого механизма отказались.

Команда jimage

В составе среды выполнения Java поставляется модульный образ в JIMAGE-файле, который называется `modules` и находится в каталоге `JAVA_HOME\lib`, где в роли `JAVA_HOME` может выступать переменная среды `JDK_HOME` или `JRE_HOME`. Для ознакомления с содержимым JIMAGE-файлов служит команда `jimage`, поддерживающая следующие операции:

- извлечение элементов из JIMAGE-файла;
- печать краткого содержания JIMAGE-файла;
- печать подробного списка элементов с указанием имени, размера, смещения от начала файла и т. д.;
- верификация файлов классов.

Команда `jimage` находится в каталоге `JDK_HOME\bin` и запускается следующим образом:

```
jimage <subcommand> <options> <jimage-file-list>
```

Здесь `<subcommand>` – одна из подкоманд, перечисленные в табл. 7.2, `<options>` – один или несколько параметров, перечисленных в табл. 7.3, а `<jimage-file-list>` – список JIMAGE-файлов через запятую.

Таблица 7.2. Список подкоманд команды jimage

Подкоманда	Описание
`extract`	Извлекает все элементы из указанных JIMAGE-файлов в текущий каталог. Если нужно извлечь в другой каталог, задайте параметр `--dir`.
`info`	Печатает подробную информацию из заголовка указанного JIMAGE-файла.
`list`	Печатает список всех модулей и их элементов в указанном JIMAGE-файле. Чтобы включить в распечатку размер, смещение и степень сжатия, задайте параметр `--verbose`.
`verify`	Печатает список элементов с расширением `.class`, не являющихся классами.

Таблица 7.3. Параметры команды jimage

Параметр	Описание
`--dir <dir-name>`	Задает каталог, в который подкоманда `extract` извлекает элементы JIMAGE-файлов.
`-h, --help`	Напечатать информацию о порядке вызова `jimage`.

`--include <pattern-list>`	Задает список масок для фильтрации элементов. Значением параметра является список разделенных запятыми строк вида:
	`<glob-pattern>`
	`glob:<glob-pattern>`
	`regex:<regex-pattern>`

`--full-version`	Напечатать полную информацию о версии `jimage`.

Параметр	Описание
`--verbose`	При использовании в подкоманде `list` печатает размер, смещение и степень сжатия каждого элемента.
`--version`	Напечатать информацию о версии `jimage`.

Приведу несколько примеров использования команды `jimage` с образом среды выполнения JDK 9, хранящимся на моем компьютере в каталоге `C:\java9\lib\modules`. Можете вместо него взять любой пользовательский образ, созданный командой `jlink`.

Следующая команда извлекает все элементы из образа среды выполнения и копирует их в каталог `extracted_jdk`. Работает она несколько секунд.

```
C:\Java9Revealed>jimage extract --dir extracted_jdk C:\java9\lib\modules
```

Следующая команда извлекает из образа все элементы с расширением `.png` в каталог `extracted_images`:

```
C:\Java9Revealed>jimage extract --include regex:.+\.png --dir extracted_images
C:\java9\lib\modules
```

Следующая команда печатает список всех элементов в образе. Показана только часть распечатки:

```
C:\Java9Revealed>jimage list C:\java9\lib\modules
```

```
jimage: C:\java9\lib\modules

Module: java.activation
    META-INF/mailcap.default
    META-INF/mimetypes.default
...
Module: java.annotations.common
    javax/annotation/Generated.class
...
```

Следующая команда печатает подробные сведения обо всех элементах в образе (обратите внимание на параметр `-verbose`). Показана только часть распечатки:

```
C:\Java9Revealed>jimage list --verbose C:\java9\lib\modules
```

```
jimage: C:\java9\lib\modules

Module: java.activation
Offset    Size   Compressed Entry
34214466   292           0 META-INF/mailcap.default
34214758   562           0 META-INF/mimetypes.default
...
Module: java.annotations.common
```

```
Offset    Size   Compressed Entry
34296622   678           0 javax/annotation/Generated.class
...
```

Следующая команда печатает список всех некорректных файлов классов. Вы можете спросить, каким образом в образе может оказаться некорректный файл класса. Хакер может записать! Однако же для этого примера мне надо было иметь такой некорректный файл. Я поступил просто: взял нормальный файл класса – копию главного класса под названием `Main2.class`, – открыл его в редакторе и удалил часть содержимого. А потом поместил испорченный файл `Main2.class` в модуль `com.jdojo.prime.client` в тот же каталог, где находился файл `Main.class`. После этого я заново создал образ среды выполнения для приложения проверки чисел на простоту. Теперь можно запустить команду:

```
C:\Java9Revealed>jimage verify primechecker\lib\modules
```

```
jimage: primechecker\lib\modules
Error(s) in Class: /com.jdojo.prime.client/com/jdojo/prime/client/Main2.class
```

При запуске для образа, поставляемого в составе JDK 9, она ничего не напечатает, т. к. в нем некорректных файлов классов нет.

Резюме

В JDK 9 образ среды выполнения хранится в специальном формате JIMAGE, оптимизированном с точки зрения памяти и быстродействия. Формат JIMAGE поддерживается только на этапе выполнения. Это контейнер, в котором хранятся индексированные модули, классы и ресурсы JDK. Поиск и загрузка классов из JIMAGE-файла производится гораздо быстрее, чем из файлов в формате JAR и JMOD. Формат JIMAGE является внутренним для JDK и разработчикам нечасто приходится иметь с ним дело напрямую.

В составе JDK 9 поставляется программа `jlink`, позволяющая создать образ среды выполнения в формате JIMAGE для вашего приложения; в него будут включены модули приложения и только те платформенные модули, которые нужны приложению. Команда `jlink` умеет создавать образы из модулей, хранящихся в виде модульных JAR-файлов, развернутых каталогов или JMOD-файлов. В составе JDK 9 имеется также программа `jimage` для анализа содержимого JIMAGE-файлов.

Глава 8

Несовместимые изменения в JDK 9

Краткое содержание главы:

- новая схема нумерации версий JDK;
- разбор строки версии JDK с помощью класса `Runtime.Version`;
- новая структура каталогов JDK/JRE 9;
- как работает механизм переопределения утвержденных стандартов в JDK 9;
- изменения в использовании механизма расширений в JDK 9;
- как работают загрузчики классов в JDK 9 и как загружаются модули;
- инкапсуляция ресурсов в JDK 9;
- доступ к ресурсам в модулях с помощью методов поиска ресурсов в классах `Module`, `Class` и `ClassLoader`;
- что такое схема URL `jrt` и как ее использовать для доступа к ресурсам в образе среды выполнения;
- обращение к внутренним API в JDK 9 и перечень удаленных API;
- как заменить классы и ресурсы в модуле с помощью параметра командной строки `--patch-module`.

Некоторые изменения в JDK 9 могут сделать неработоспособными приложения, которые правильно работали в JDK 8. В этой главе такие изменения описаны во всех подробностях.

Новая схема нумерации версий JDK

Раньше схема нумерации версий JDK была непонятна разработчикам и неудобна для программного разбора. Трудно было сказать, в чем тонкие различия между двумя версиями JDK. Нелегко было ответить даже на простой вопрос: «Какая версия содержит самые последние исправления безопасности: JDK 7 Update 55 или

JDK 7 Update 60?». Хочется сказать «JDK 7 Update 60», но на самом деле ответ не столь очевиден: в обеих версиях исправления безопасности одни и те же. А в чем разница между версиями JDK 8 Update 66, 1.8.0_66 и JDK 8u66? Да ни в чем, это одна и та же версия. Чтобы разобраться в деталях, заключенных в строке версии, нужно было хорошо понимать схему нумерации версий. В JDK 9 сделана попытка стандартизировать эту схему, чтобы она была понятна человеку, допускала простой разбор в программах и следовала отраслевым стандартам нумерации версий.

В JDK 9 имеется статический вложенный класс `Runtime.Version`, который представляет строку версии платформы Java SE. Он позволяет разбирать, проверять правильность и сравнивать строки версий.

Строка версии состоит из следующих элементов в указанном ниже порядке. Обязателен только первый элемент.

- номер версии;
- признак предварительной версии;
- номер сборки;
- дополнительная информация.

Формат строки версии определяется таким регулярным выражением:

```
$vnum(-$pre)?(\+($build)?(-$opt)?)?
```

Короткая строка версии состоит только из номера версии и необязательного признака предварительной версии:

```
$vnum(-$pre)?
```

Строка версии может состоять из одной цифры `"9"`, определяющей основной номер версии, или быть довольно длинной: `"9.0.1-ea+154-20170130.07.36am"`, если включены все части.

Номер версии

Номер версии – это последовательность элементов, разделенных точками. Ее длина произвольна, а формат такой:

```
^[1-9][0-9]*((\.0)*\.[1-9][0-9]*)*)*$
```

Номер версии может содержать от одного до четырех элементов:

```
$major.$minor.$security(.$addtionalInfo)
```

Элемент `$major` – основная версия JDK. Он увеличивается на единицу при выпуске версии, содержащей новые важные функциональные возможности. Например, для JDK 8 основной номер версии равен 8, а для JDK 9 – 9. При увеличении основного номера версии все остальные части удаляются. Например, если имелась версия с номером 9.2.2.1, то при переходе на следующую основную версию новый номер будет равен 10.

Элемент `$minor` – дополнительный номер версии JDK. Он увеличивается на единицу при выпуске обновлений, например: исправлений ошибок, новых сборщиков мусора, новых API, относящихся к JDK и т. д.

Элемент `$security` относится к обновлениям JDK, связанным с безопасностью, и увеличивается при каждом обновлении безопасности. Он не сбрасывается при

увеличении дополнительного номера версии. Чем больше величина `$security` для данного номера `$major`, тем безопаснее версия. Например, JDK 9.1.7 так же безопасна, как JDK 9.5.7, поскольку уровень безопасности в обеих версиях одинаков и равен 7. Напротив, JDK 9.2.2 безопаснее 9.2.1, потому что при одном и том же основном номере версии 9 уровень безопасности 2 больше чем 1.

К номерам версий применимы следующие правила:

- каждый элемент должен быть неотрицательным целым числом;
- первые три элемента интерпретируются как основной номер, дополнительный номер и уровень безопасности, а остальные, если присутствуют, — как дополнительная информация, например, номер исправления;
- обязателен только основной номер версии;
- элементы номера версии не могут содержать начальных нулей; например, основной номер версии JDK 9 равен 9, а не 09;
- последний элемент не может быть равен нулю, т. е. номер версии 9.0.0 недопустим. Разрешены номера вида 9, 9.2 или 9.0.*x*, где *x* – целое положительное число.

Признак предварительной версии

Элемент `$pre` в строке версии – идентификатор предварительной версии, например: `ea` для ознакомительной (early access) версии, `snapshot` для снимка предварительной версии и `internal` для внутренней сборки, предназначенной для разработчиков. Это необязательный элемент. Если он присутствует, то начинается дефисом (`-`) и должен быть строкой букв и цифр, удовлетворяющей регулярному выражению `([a-zA-Z0-9]+)`. В следующей строке версии номер версии равен 9 и указан признак `ea`:

```
9-ea
```

Информация о сборке

Элемент `$build` в строке версии увеличивается на 1 после каждой одобренной (promoted) сборки. Это необязательный элемент. Он сбрасывается в 1 при увеличении любой части номера версии. Если он присутствует, то начинается знаком `+` и должен удовлетворять регулярному выражению `(0|[1-9][0-9]*)`. В следующей строке сборки указан номер сборки 154:

```
9-ea+154
```

Дополнительная информация

Элемент `$opt` в строке версии содержит дополнительную информацию о сборке, например, дату и время внутренней сборки. Это необязательный элемент, который может содержать буквы, цифры, дефисы и точки. Если он присутствует, то начинается знаком дефиса (`-`) и должен удовлетворять регулярному выражению `([-a-zA-Z0-9\.]+)`. Если элемент `$build` отсутствует, то `$opt` должен предваряться последовательностью `+-`. Например, в строке `9-ea+132-2016-08-23` `$build` равно 132, а `$opt` равно `2016-08-23`; в строке `9+-123` элементы `$pre` и `$build` отсутствуют, а `$opt` рав-

но 123. В следующую строку версии включены дата и время выпуска в составе дополнительной информации:

```
9-ea+154-20170130.07.36am
```

Разбор старой и новой строки версии

Выпуски JDK были двух видов: ограниченное обновление (Limited Update), включающее новую функциональность и исправления, не относящиеся к безопасности, и критическое исправление (Critical Patch Updates), включающее только исправления обнаруженных уязвимостей. Строка версии включала номер версии, содержащий номер обновления и номер сборки. Ограниченным обновлениям присваивались номера, кратные 20. Для нумерации критических обновлений использовались нечетные числа, получающиеся путем прибавления к номеру предшествующего ограниченного обновления числа, кратного 5, и, возможно, единицы, чтобы результат был нечетным. Примером может служить строка `1.8.0_31-b13` – обновление 31 основной версии 8. Здесь номер сборки равен 13. Отметим, что до JDK 9 строка версии всегда начиналась с 1.

> **Совет.** Ранее написанный код разбора строки версии для выделения основного номера может перестать работать в JDK 9, хотя это зависит от логики алгоритма. Например, если алгоритм считал, что основной номер – второй элемент и пропускал первый, равный 1, то работать он не будет. Так, для строки 1.8.0 такой алгоритм возвращал 8, а для строки 9.0.1 вернет 0, хотя правильное значение — 9.

Изменение значений системных свойств

В JDK 9 изменились значения системных свойств, содержащих версию JDK. В табл. 8.1 перечислены такие системные свойства и их формат. `$vstr`, `$vnum` и `$pre` обозначают строку версии, номер версии и признак предварительной версии соответственно.

Таблица 8.1. Значения системных свойств в JDK 9

Системное свойство	Значение
`java.version`	`$vnum(\-$pre)?`
`java.runtime.version`	`$vstr`
`java.vm.version`	`$vstr`
`java.specification.version`	`$vnum`
`java.vm.specification.version`	`$vnum`

Использование класса Runtime.Version

В JDK 9 добавлен статический вложенный класс `Runtime.Version`, объекты которого представляют строки версий. В классе `Version` нет открытого конструктора. Получить его экземпляр можно, только вызвав статический метод `parse(String vstr)`.

Этот метод возбуждает исключение, если строка версии равна `null` или недопустима.

```
import java.lang.Runtime.Version;
...
// Разобрать строку версии "9.0.1-ea+132"
Version version = Version.parse("9.0.1-ea+132");
```

Следующие методы класса `Runtime.Version` возвращают элементы строки версии:

- `int major()`
- `int minor()`
- `int security()`
- `Optional<String> pre()`
- `Optional<Integer> build()`
- `Optional<String> optional()`

Отметим, что для необязательных элементов `$pre`, `$build` и `$opt` возвращается значение типа `Optional`. А для необязательных элементов `$minor` и `$security` возвращается значение типа `int`, а не `Optional`; если какой-то из них отсутствует в строке версии, то будет возвращено значение 0.

Напомним, что номер версии может содержать дополнительную информацию после третьего элемента. В классе `Version` нет метода для получения дополнительной информации непосредственно, но есть метод `version()`, который возвращает список типа `List<Integer>`, содержащий все элементы номера версии. Первые три элемента списка – `$major`, `$minor` и `$security`, а остальные содержат дополнительную информацию.

В классе `Runtime.Version` есть методы для сравнения двух строк версий на равенство и на предшествование. При сравнении можно учитывать или не учитывать дополнительную информацию о сборке (`$opt`).

- `int compareTo(Version v)`
- `int compareToIgnoreOptional(Version v)`
- `boolean equals(Object v)`
- `boolean equalsIgnoreOptional(Object v)`

Вызов `v1.compareTo(v2)` возвращает отрицательное, нулевое или положительное число, если `v1` меньше, равна или больше `v2`. Метод `compareToIgnoreOptional()` работает так же, как `compareTo()`, но игнорирует информацию о сборке. Методы `equals()` и `equalsIgnoreOptional()` сравнивают две строки версий на равенство с учетом и без учета факультативной информации о сборке.

Какая из двух строк версий относится к более поздней версии: `9.1.1` или `9.1.1-ea`? Первая не содержит признака предварительной версии, а вторая содержит, так что первая – более поздняя. А если взять такие строки: `9.1.1` и `9.1.1.1-ea`? Теперь позднее вторая версия. Сравнение производится в такой последовательности: `$vnum`, `$pre`, `$build` и `$opt`. Если номер версии уже определяет порядок, то остальные элементы не сравниваются.

Исходный код, приведенный в этом разделе, находится в модуле `com.jdojo.version.string`, объявление которого показано в листинге 8.1. В листинге 8.2 приве-

ден полный код программы, показывающей, как выделять части строки версии с помощью класса `Runtime.Version`.

Листинг 8.1. Объявление модуля com.jdojo.version.string

```
// module-info.java
module com.jdojo.version.string {
    exports com.jdojo.version.string;
}
```

Листинг 8.2. Класс VersionTest для демонстрации работы с классом `Runtime.Version`

```
// VersionTest.java
package com.jdojo.version.string;

import java.util.List;
import java.lang.Runtime.Version;

public class VersionTest {
   public static void main(String[] args) {
      String[] versionStrings = {
         "9", "9.1", "9.1.2", "9.1.2.3.4", "9.0.0",
         "9.1.2-ea+153", "9+132", "9-ea+132-2016-08-23", "9+-123",
         "9.0.1-ea+132-2016-08-22.10.56.45am"};

      for (String versonString : versionStrings) {
         try {
            Version version = Version.parse(versonString);

            // Получить дополнительные элементы номера версии,
            // начиная с четвертого
            String vnumAdditionalInfo = getAdditionalVersionInfo(version);
            System.out.printf("Version String=%s%n", versonString);
            System.out.printf("Major=%d, Minor=%d, Security=%d, " +
               "Additional Version=%s, Pre=%s, Build=%s, Optional=%s %n%n",
               version.major(),
               version.minor(),
               version.security(),
               vnumAdditionalInfo,
               version.pre().orElse(""),
               version.build().isPresent() ? version.build().get().toString() : "",
               version.optional().orElse(""));
         } catch (Exception e) {
            System.out.printf("%s%n%n", e.getMessage());
         }
      }
```

```
    }

    // Возвращает элементы номера версии с четвертого до конца
    public static String getAdditionalVersionInfo(Version v) {
        String str = "";

        List<Integer> vnum = v.version();
        int size = vnum.size();
        if (size >= 4) {
            str = str + String.valueOf(vnum.get(3));
        }
        for (int i = 4; i < size; i++) {
            str = str + "." + String.valueOf(vnum.get(i));
        }

        return str;
    }
}
```

```
Version String=9
Major=9, Minor=0, Security=0, Additional Version=, Pre=, Build=, Optional=

Version String=9.1
Major=9, Minor=1, Security=0, Additional Version=, Pre=, Build=, Optional=

Version String=9.1.2
Major=9, Minor=1, Security=2, Additional Version=, Pre=, Build=, Optional=

Version String=9.1.2.3.4
Major=9, Minor=1, Security=2, Additional Version=3.4, Pre=, Build=, Optional=

Invalid version string: '9.0.0'

Version String=9.1.2-ea+153
Major=9, Minor=1, Security=2, Additional Version=, Pre=ea, Build=153, Optional=

Version String=9+132
Major=9, Minor=0, Security=0, Additional Version=, Pre=, Build=132, Optional=

Version String=9-ea+132-2016-08-23
Major=9, Minor=0, Security=0, Additional Version=, Pre=ea, Build=132, Optional=2016-08-23

Version String=9+-123
Major=9, Minor=0, Security=0, Additional Version=, Pre=, Build=, Optional=123

Version String=9.0.1-ea+132-2016-08-22.10.56.45am
Major=9, Minor=0, Security=1, Additional Version=, Pre=ea, Build=132, Optional=2016-08-22.10.56.45am
```

Изменения в JDK и JRE

В Java SE 9 JDK и JRE преобразованы в модульную форму. Поэтому произошли некоторые изменения в их структуре. Был внесен и ряд других изменений для повышения производительности, безопасности и удобства сопровождения. В большинстве своем эти изменения затрагивают разработчиков библиотек и IDE, а не разработчиков приложений. Для удобства обсуждения я разбил их на три категории:

- структурные изменения;
- изменения поведения;
- изменения API.

Структурные изменения JDK и JRE

Структурные изменения касаются организации каталогов и файлов в образах среды выполнения. До Java SE 9 система сборки JDK порождала образы двух видов: Java Runtime Environment (JRE) и Java Development Kit (JDK). JRE – это полная реализация платформы Java SE, а JDK помимо JRE включал средства разработки и библиотеки. Можно было установить только JRE или JDK, включающий JRE. На рис. 8.1 изображены основные каталоги в развернутом дистрибутиве JDK до выхода Java SE 9. Переменная среды `JDK_HOME` содержит путь к каталогу, в который был установлен JDK. Если устанавливалась только JRE, то присутствовал лишь каталог `jre` и все, что под ним.

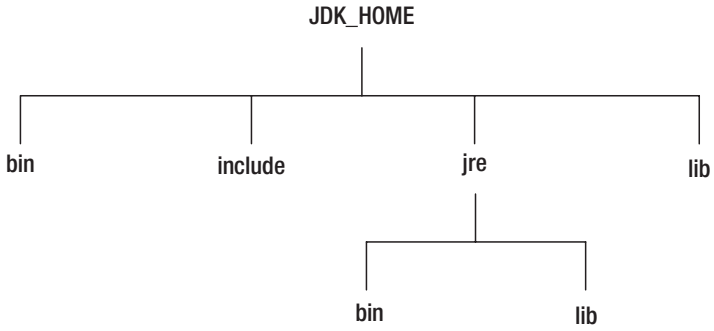

Рис. 8.1. Организация каталогов JDK и JRE до Java SE 9

Раньше в JDK были такие каталоги:

- каталог `bin` содержал командные программы для разработки и отладки, включая `javac`, `jar` и `javadoc`. Здесь же находилась команда `java` для запуска Java-приложений;
- каталог `include` содержал файлы-заголовки C/C++, необходимые для компиляции платформенного кода;
- каталог `lib` содержал несколько JAR-файлов и файлы других типов, необходимые инструментам JDK. В частности, здесь находился файл `tools.jar` с Java-классами для компилятора `javac`;

- каталог `jre\bin` содержал абсолютно необходимые команды, в т. ч. `java`. На платформе Windows здесь же находились системные динамически загружаемые библиотеки (DLL-файлы);
- каталог `jre\lib` содержал редактируемые пользователем конфигурационные файлы, в т. ч. с расширениями `.properties` и `.policy`;
- каталог `jre\lib\endorsed` содержал JAR-файлы, поддерживающие механизм переопределения утвержденных стандартов (Endorsed Standards Override Mechanism). Это позволяло включать в платформу Java более поздние версии классов и интерфейсов, реализующие технологию Endorsed Standards, но созданные не в рамках процесса Java Community Process. Эти JAR-файлы добавляются в начало пути к классам на этапе начальной загрузки JVM и тем самым переопределяют определения этих классов и интерфейсов, присутствующие в среде выполнения Java;
- каталог `jre\lib\ext` содержал JAR-файлы, обеспечивающие механизм расширения. Все находящиеся в этом каталоге JAR-файлы загружались загрузчиком классов расширения, который является потомком базового загрузчика классов и родителем системного загрузчика, загружающего все классы приложения. Помещая JAR-файлы в этом каталог, вы могли расширить платформу Java SE. Содержимое таких JAR-файлов видно всем приложениям, которые компилируются или выполняются в данной среде;
- каталог `jre\lib` содержал несколько JAR-файлов. Файл `rt.jar` содержал Java-классы и ресурсы, необходимые на этапе выполнения. Многие инструментальные средства зависели от местоположения файла `rt.jar`;
- каталог `jre\lib` содержал платформенные динамические библиотеки на платформах, отличных от Windows;
- каталог `jre\lib` содержал несколько других подкаталогов, где хранились файлы, необходимые на этапе выполнения, в т. ч. шрифты и изображения.

Корневой каталог JDK и JRE, содержал несколько файлов, в т. ч. `COPYRIGHT`, `LICENSE` и `README.html`. Файл `release` в корневом каталоге содержал пары ключ-значение, описывающие образ среды выполнения: версию Java, версию ОС, архитектуру ОС и т. д. Ниже показан фрагмент файла `release` из JDK 8:

```
JAVA_VERSION="1.8.0_66"
OS_NAME="Windows"
OS_VERSION="5.2"
OS_ARCH="amd64"
BUILD_TYPE="commercial"
```

В Java SE 9 иерархия каталогов сделана плоской, а различие между JDK и JRE устранено. На рис. 8.2 показаны каталоги установки JDK в Java SE 9. Отметим, что JRE в версии JDK 9 не содержит каталогов `include` и `jmods`.

В Java SE 9 в JDK:

- нет подкаталога `jre`;
- каталог `bin` содержит все команды. На платформе Windows этот каталог по-прежнему содержит динамически загружаемые библиотеки;

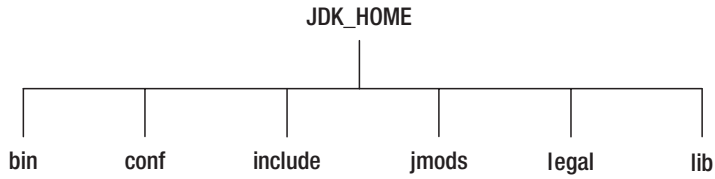

Рис. 8.2. Организация каталогов JDK в Java SE 9

- каталог `conf` содержит редактируемые пользователями конфигурационные файлы, в т. ч. с расширениями `.properties` и `.policy`, которые раньше находились в каталоге `jre\lib`;
- каталог `include`, как и раньше, содержит файлы-заголовки C/C++, необходимые для компиляции платформенного кода. Он присутствует только в JDK;
- каталог `jmods` содержит платформенные модули в формате JMOD. Он необходим при создании пользовательского образа среды выполнения и присутствует только в JDK;
- каталог `legal` содержит правовые уведомления;
- каталог `lib` содержит платформенные динамические библиотеки на платформах, отличных от Windows. Находящиеся в нем подкаталоги и файлы не предназначены для редактирования или использования разработчиками.

Корневой каталог в JDK 9 по-прежнему содержит файлы `COPYRIGHT` и `README`. В файле `release` появился новый элемент `MODULES`, значением которого является список модулей, включенных в образ. Ниже показана часть файла `release` в JDK 9:

```
MODULES=java.rmi,jdk.jdi,jdk.policytool
OS_VERSION="5.2"
OS_ARCH="amd64"
OS_NAME="Windows"
JAVA_VERSION="9"
JAVA_FULL_VERSION="9-ea+133"
```

Я оставил в списке только три модуля. В действительности в полном JDK этот список включает все платформенные модули. А в пользовательском образе среды выполнения он будет содержать только те модули, которые вы включите.

> **Совет.** В Java SE 9 удалены файлы `lib\tools.jar` в JDK и `lib\rt.jar` в JRE. Классы и ресурсы, которые раньше находились в этих JAR-файлах, теперь хранятся в каталоге `lib` во внутреннем формате в файле `modules`. Для извлечения этих классов и ресурсов из образа служит новая схема `jrt`. Приложения, которые зависели от местоположения этих JAR-файлов, перестанут работать.

Изменения поведения

Эти изменения влияют на поведение приложения на этапе выполнения и описываются в следующих разделах.

Механизм переопределения утвержденных стандартов

До Java SE 9 у нас была возможность использовать механизм переопределения утвержденных стандартов, позволяющий воспользоваться новыми версия-

ми классов и интерфейсов, реализующих такие API, как пакет `javax.rmi.CORBA` или Java API для обработки XML (JAXP), которые были созданы не в рамках процесса Java Community Process. Эти JAR-файлы помещались в начало пути к классам на этапе начальной загрузки JVM и тем самым замещали определения классов и интерфейсов, присутствующие в JRE. Местоположения этих JAR-файлов задавались системным свойством `java.endorsed.dirs`, в котором каталоги отделялись друг от друга платформенно-зависимым символом-разделителем. Если это свойство не было установлено, то среда выполнения искала классы и интерфейсы в JAR-файлах в каталоге `jre\lib\endorsed`.

В Java SE 9 механизм переопределения утвержденных стандартов по-прежнему поддерживается. Но среда выполнения теперь состоит из модулей и, чтобы воспользоваться этим механизмом, необходимо иметь новые версии модулей, содержащих соответствующие API. Необходимо задать параметр командной строки `--upgrade-module-path`, значением которого является список каталогов, содержащих такие модули. Следующая команда на платформе Windows переопределяет модули утвержденных стандартов, в т. ч. `java.corba` в JDK 9. Модули в каталогах `umod1` и `umod2` будут использоваться вместо соответствующих модулей в образе среды выполнения:

```
java --upgrade-module-path umod1;umod2 <other-options>
```

> **Совет.** В Java SE 9 считается ошибкой создание каталога `JAVA_HOME\lib\endorsed` и задание системного свойства `java.endorsed.dirs`.

Механизм расширения

В предыдущих версиях Java SE существовал механизм, позволяющий расширить образ среды выполнения, поместив JAR-файлы в каталоги, перечисленные в системном свойстве `java.ext.dirs`. Если это свойство не было задано, то по умолчанию подразумевался каталог `jre\lib\ext`. Все JAR-файлы, находившиеся в этом каталоге, загружались загрузчиком классов расширения, который являлся потомком базового загрузчика классов и родителем системного загрузчика, загружающего все классы приложения. Содержимое таких JAR-файлов было видно всем приложениям, которые компилируются или выполняются в данной среде.

В Java SE 9 механизм расширения не поддерживается. Если подобная функциональность вам необходима, поместите JAR-файлы в начало пути к классам. Наличие каталога `JAVA_HOME\lib\ext` или установка системного свойства `java.ext.dirs` приводит к ошибке в JDK 9.

Изменения в загрузчиках классов

На этапе выполнения все типы загружаются некоторым загрузчиком классов, представленным экземпляром класса `java.lang.ClassLoader`. Имея ссылку на объект `obj`, можно получить ссылку на ассоциированный с ним загрузчик классов, вызвав метод `obj.getClass().getClassLoader()`. А для получения родительского загрузчика классов нужно вызвать метод `getParent()`.

До версии 9 в JDK использовалось три загрузчика классов, показанных на рис. 8.3. Стрелкой обозначено направление делегирования. Эти три загрузчика загружают классы, различающиеся местоположением и типом.

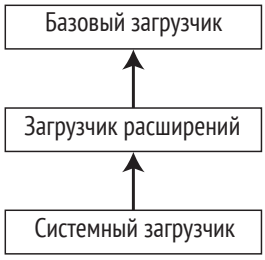

Рис. 8.3. Иерархия загрузчиков классов до JDK 9

Загрузчики классов организованы иерархически, и базовый находится на вершине иерархии. Загрузчик делегирует запрос на загрузку класса загрузчику, находящемуся выше него в иерархии. Так, если системному загрузчику нужно загрузить класс, то сначала он делегирует запрос загрузчику расширений, а тот – базовому загрузчику. Если базовый загрузчик не может загрузить класс, то это попытается сделать загрузчик расширений. Если и тот потерпит неудачу, то настанет очередь системного загрузчика. Если и у него не получится, то возбуждается исключение `ClassNotFoundException`.

Базовый загрузчик классов является родителем загрузчика расширений, а загрузчик расширений – родителем системного загрузчика. У базового загрузчика нет родителя. По умолчанию родителем дополнительных пользовательских загрузчиков классов является системный загрузчик.

Базовый загрузчик загружает базовые классы, включая классы из файла `JAVA_HOME\lib\rt.jar` и нескольких других JAR-файлов, образующих среду выполнения. Он реализован на уровне виртуальной машины. С помощью параметров командной строки `-Xbootclasspath/p` и `-Xbootclasspath/a` можно добавить в начало и в конец пути к базовым классам дополнительные каталоги. Параметр `-Xbootclasspath` позволяет полностью переопределить путь к базовым классам. На этапе выполнения системное свойство `sun.boot.class.path` содержит путь к базовым классам, допускающий только чтение. В JDK этот загрузчик классов представлен значением `null`, т. е. получить ссылку на него невозможно. Например, класс `Object` загружается базовым загрузчиком, поэтому вызов `Object.class.getClassLoader()` возвращает `null`.

Загрузчик расширений служит для загрузки классов, доступных посредством механизма расширения, т. е. из JAR-файлов, находящихся в каталогах, которые перечислены в системном свойстве `java.ext.dirs`. Чтобы получить ссылку на загрузчик расширений, необходимо получить ссылку на системный загрузчик, а затем вызвать для нее метод `getParent()`.

Системный загрузчик загружает классы, находящиеся на пути к классам приложения, который определен переменной среды `CLASSPATH` или параметром командной строки `-cp` (либо `-classpath`). Название *системный загрузчик* не вполне корректно, поскольку предполагает, что загружаются системные классы, что на самом деле не так – загружаются классы приложения. Ссылку на системный загрузчик возвращает статический метод `getSystemClassLoader()` класса `ClassLoader`.

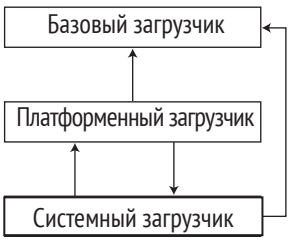

Рис. 8.4. Иерархия загрузчиков классов в JDK 9

В JDK 9 ради обратной совместимости сохранена трехуровневая иерархия загрузчиков классов, но в нее внесены некоторые изменения, касающиеся загрузки классов из системы модулей. На рис. 8.4 показана иерархия загрузчиков в JDK 9.

Отметим, что в JDK 9 системный загрузчик может делегировать запрос платформенному или базовому загрузчику, а платформенный загрузчик может делегировать запрос базовому или системному.

В JDK 9 базовый загрузчик реализован как в библиотеке, так и в виртуальной машине. В целях обратной совместимости он по-прежнему представлен значением `null`, т. е. вызов `Object.class.getClassLoader()` возвращает `null`, как и раньше. Однако не все модули платформы Java SE и JDK загружаются базовым загрузчиком. Среди прочих базовым загрузчиком загружаются модули `java.base`, `java.logging`, `java.prefs` и `java.desktop`. Прочие модули платформы Java SE и JDK загружаются платформенным и системным загрузчиком, как описано ниже. Параметры для задания пути к базовым классам, `-Xbootclasspath` и `-Xbootclasspath/p`, а также системное свойство `sun.boot.class.path` в JDK 9 не поддерживаются. Но параметр `-Xbootclasspath/a` поддерживается, и его значение хранится в системном свойстве `jdk.boot.class.path.append`.

В JDK 9 больше не поддерживается механизм расширения. Но загрузчик расширений сохранен, хотя и называется теперь *платформенным загрузчиком классов*. В классе `ClassLoader` появился новый статический метод `getPlatformClassLoader()`, возвращающий ссылку на этот загрузчик. В табл. 8.2 перечислены модули, загружаемые платформенным загрузчиком. На этот загрузчик возложена еще одна функция. Классам, загруженным базовым загрузчиком, по умолчанию предоставляются все права. Однако не всем классам необходимы такие широкие полномочия. В JDK 9 для повышения безопасности такие классы лишены части привилегий и теперь загружаются платформенным загрузчиком.

Таблица 8.2. Список модулей, загружаемых платформенным загрузчиком в JDK 9

java.activation	java.xml.ws.annotation	jdk.desktop
java.compiler	javafx.base	jdk.dynalink
java.corba	javafx.controls	jdk.javaws
java.jnlp	javafx.deploy	jdk.jsobject
java.scripting	javafx.fxml	jdk.localedata
java.se	javafx.graphics	jdk.naming.dns
java.se.ee	javafx.media	jdk.plugin
java.security.jgss	javafx.swing	jdk.plugin.dom
java.smartcardio	javafx.web	jdk.plugin.server
java.sql	jdk.accessibility	jdk.scripting.nashorn
java.sql.rowset	jdk.charsets	jdk.security.auth
java.transaction	jdk.crypto.cryptoki	jdk.security.jgss
java.xml.bind	jdk.crypto.ec	jdk.xml.dom
java.xml.crypto	jdk.crypto.mscapi	jdk.zipfs
java.xml.ws	jdk.deploy	

Системный загрузчик загружает модули приложения, находящиеся на пути к модулям, и несколько модулей JDK, предоставляющих инструментальные средства или экспортирующих инструментальные API (см. табл. 8.3). Для получения ссылки на системный загрузчик по-прежнему можно использовать статический метод `getSystemClassLoader()` класс `ClassLoader`.

Таблица 8.3. Список модулей, загружаемых системным загрузчиком в JDK 9

jdk.attach	jdk.jartool	jdk.jstatd
jdk.compiler	jdk.javadoc	jdk.pack
jdk.deploy.controlpanel	jdk.jcmd	jdk.packager
jdk.editpad	jdk.jconsole	jdk.packager.services
jdk.hotspot.agent	jdk.jdeps	jdk.policytool
jdk.internal.ed	jdk.jdi	jdk.rmic
jdk.internal.jvmstat	jdk.jdwp.agent	jdk.scripting.nashorn.shell
jdk.internal.le	jdk.jlink	jdk.xml.bind
jdk.internal.opt	jdk.jshell	jdk.xml.ws

Совет. Раньше загрузчик расширений и системный загрузчик были экземплярами класса `java.net.URLClassLoader`. В JDK 9 платформенный загрузчик (бывший загрузчик расширений) и системный загрузчик – экземпляры внутреннего класса JDK. Программа, полагавшаяся на методы класса `URLClassLoader`, может перестать работать в JDK 9.

Механизм загрузки классов в JDK 9 немного изменился. Три встроенных загрузчика теперь работают сообща. Когда системному загрузчику необходимо загрузить класс, он просматривает модули, назначенные всем загрузчикам. Если подходящий модуль назначен какому-то из них, то этот загрузчик и загружает класс – то есть теперь системный загрузчик может делегировать запрос как базовому, так и платформенному загрузчику. Если класс не найден в именованном модуле, назначенном какому-нибудь загрузчику, то системный загрузчик делегирует запрос своему родителю, т. е. платформенному загрузчику. Если класс по-прежнему не загружен, то системный загрузчик просматривает путь к классам. Если класс найден, то он загружается как часть безымянного модуля. В противном случае возбуждается исключение `ClassNotFoundException`.

Когда платформенному загрузчику необходимо загрузить класс, он просматривает модули, назначенные всем загрузчикам. Если подходящий модуль назначен какому-то из них, то этот загрузчик и загружает класс – то есть платформенный загрузчик может делегировать запрос как базовому, так и системному загрузчику. Если класс не найден в именованном модуле, назначенном какому-нибудь загрузчику, то платформенный загрузчик делегирует запрос своему родителю, т. е. базовому загрузчику.

Когда базовому загрузчику необходимо загрузить класс, он просматривает собственный список именованных модулей. Если класс не найден, то просматривается список файлов и каталогов, заданный с помощью параметра командной строки `-Xbootclasspath/a`. Если класс найден на пути к базовым классам, то он загружается как часть безымянного модуля.

Загрузчики классов, а также загружаемые ими модули и классы, можно наблюдать в динамике. В JDK 9 появился новый параметр `-Xlog:modules`, который позволяет протоколировать отладочные или трассировочные сообщения, отправляемые виртуальной машины по мере загрузки модулей:

```
-Xlog:modules=<debug|trace>
```

При этом выводится очень много информации. Я рекомендую перенаправлять вывод в файл, где с ним можно будет комфортно ознакомиться. Следующая команда в Windows запускает клиентскую программу, обращающуюся к службе простых чисел, и протоколирует сообщения о загрузке модулей в файле `test.txt`. Показана только часть вывода. Видно, каким загрузчикам классов назначены модули.

```
C:\Java9Revealed>java -Xlog:modules=trace --module-path lib
--module com.jdojo.prime.client/com.jdojo.prime.client.Main > test.txt
```

```
[0.022s][trace][modules] Setting package: class: java.lang.Object, package: java/lang,
loader: <bootloader>, module: java.base
[0.022s][trace][modules] Setting package: class: java.io.Serializable, package: java/io,
loader: <bootloader>, module: java.base
...
[0.855s][debug][modules] define_module(): creation of module: com.jdojo.prime.client,
version: NULL, location: file:///C:/Java9Revealed/lib/com.jdojo.prime.client.jar, class
loader 0x00000049ec86dd90 a 'jdk/internal/loader/ClassLoaders$AppClassLoader'{0x00000000895
d1c98}, package #: 1
[0.855s][trace][modules] define_module(): creation of package com/jdojo/prime/client for
module com.jdojo.prime.client
...
```

Доступ к ресурсам

Ресурсами называются данные, используемые приложением: изображения, звуковые файлы, видео, текст и т. д. Java всегда предоставлял независимый от местоположения способ доступа к ресурсам. Для этого достаточно было упаковать их в JAR-файлы точно так же, как классы, и разместить эти файлы на пути к классам. Обычно файлы классов и ресурсы упаковываются в одни и те же JAR-файлы. Доступ к ресурсам – важная операция, которую приходится выполнять любому разработчику на Java. В следующих разделах я расскажу о том, какие API были доступны в JDK раньше и в JDK 9.

Доступ к ресурсам до JDK 9

В этом разделе я объясню, как осуществлялся доступ к ресурсам до выхода JDK 9. Если вам все это знакомо, то можете сразу перейти к следующему разделу.

В Java-программе ресурс идентифицируется своим именем – последовательностью строк, разделенных знаком `/`. Для ресурсов, хранящихся в JAR-файлах, имя ресурса – это просто путь к файлу внутри JAR. Например, до выхода JDK 9, файл `Object.class` из пакета `java.lang`, хранившийся в файле `rt.jar`, считался ресурсом с именем `java/lang/Object.class`.

До JDK 9 для доступа к ресурсам можно было использовать методы двух классов:

- `java.lang.Class`
- `java.lang.ClassLoader`

Поиск ресурсов производится классом ClassLoader. Методы поиска в классе Class делегируют работу классу ClassLoader. Поэтому, поняв, как устроен процесс поиска ресурсов в ClassLoader, вы не затруднитесь использованием методов класса Class. В обоих классах есть два метода экземпляра с разными именами:

- URL getResource(String name)
- InputStream getResourceAsStream(String name)

Тот и другой ищут ресурс одинаково, а отличается только тип возвращаемого значения. В первом случае возвращается объект типа URL, а во втором – типа InputStream. Второй метод эквивалентен вызову первого с последующим вызовом openStream() для возвращенного объект URL.

Совет. Все методы поиска ресурса возвращают null, если запрошенный ресурс не найден.

В классе ClassLoader есть три дополнительных статических метода поиска ресурса:

- static URL getSystemResource(String name)
- static InputStream getSystemResourceAsStream(String name)
- static Enumeration<URL> getSystemResources(String name)

Для поиска ресурса они пользуются системным загрузчиком классов. Первый метод возвращает URL первого найденного ресурса, второй возвращает первый найденный ресурс в виде объекта InputStream, а третий – перечисление URL-адресов всех найденных ресурсов с указанным именем.

Существует два вида методов для поиска ресурса: getSystemResource* и getResource*. Прежде чем говорить о том, какой лучше, важно понимать, что есть два вида ресурсов:

- системные;
- несистемные.

Системный ресурс ищется на пути к классам – в каталогах базовых классов, в JAR-файлах в каталогах расширений, в каталогах классов приложения. Несистемный ресурс может находиться где угодно: в указанных каталогах, в сети или в базе данных. Метод getSystemResource() ищет ресурс, пользуясь системным загрузчиком классов, который делегирует запрос своему родителю – загрузчику расширений, а тот, в свою очередь, собственному родителю – базовому загрузчику. Если приложение автономное и пользуется только тремя встроенными в JDK загрузчиками классов, то статических методов вида getSystemResource* достаточно. Приложение найдет все ресурсы на пути к классам, включая и находящиеся в образе среды выполнения, например, в файле rt.jar. Если речь идет об аплете, работающем в браузере, или о корпоративном приложении, работающем в составе сервера приложений или веб-сервера, то следует использовать методы экземпляра вида getResource*, которые ищут ресурс с помощью конкретного загрузчика классов. Если вызвать метод вида getResource* от имени объекта Class, то для поиска будет использован тот загрузчик, который загрузил этот объект.

Всем методам класса ClassLoader передаются абсолютные имена ресурсов, не начинающиеся знаком /. Так, при вызове метода getSystemResource() в качестве имени ресурса указывается java/lang/Object.class.

Методы поиска ресурсов в классе `Class` позволяют задавать как абсолютные, так и относительные имена. Абсолютное имя начинается знаком `/`. Если задано абсолютное имя, то метод класса `Class` удаляет начальный символ `/` и делегирует задачу поиска ресурса загрузчику, который загрузил объект `Class`. Вызов

```
Test.class.getResource("/resources/test.config");
```

преобразуется в такой:

```
Test.class.getClassLoader()
          .getResource("resources/test.config");
```

Если задано относительное имя, то методы класса `Class` добавляют в начало имя пакета, заменяя точки в нем знаками косой черты и дописывая косую черту в конец, а затем делегируют поиск ресурса загрузчику, который загрузил объект `Class`, В предположении, что класс `Test` находится в пакете `com.jdojo.test`, вызов

```
Test.class.getResource("resources/test.config");
```

преобразуется в такой:

```
Test.class.getClassLoader()
          .getResource("com/jdojo/test/resources/test.config");
```

Рассмотрим пример поиска ресурсов в версии, предшествующей JDK 9, в данном случае JDK 8. Исходный код примера вместе с проектом NetBeans имеется в сопроводительном коде для книги. Проект NetBeans называется `com.jdojo.resource.preJDK9`. Если будете создавать проект сами, не забудьте изменить платформу Java и формат исходного файла на JDK 8. Классы и ресурсы организованы следующим образом:

- `word_to_number.properties`
- `com/jdojo/resource/prejdk9/ResourceTest.class`
- `com/jdojo/resource/prejdk9/resources/number_to_word.properties`

В проекте есть два ресурсных файла: `word_to_number.properties` в корне и `number_to_word.properties` в каталоге `com/jdojo/resource/prejdk9/resources`. Их содержимое показано в листингах 8.3 и 8.4.

Листинг 8.3. Содержимое файла word_to_number.properties

```
One=1
Two=2
Three=3
Four=4
Five=5
```

Листинг 8.4. Содержимое файла number_to_word.properties

```
1=One
2=Two
3=Three
```

```
4=Four
5=Five
```

В листинге 8.5 приведен полный код программы, демонстрирующей поиск ресурсов с помощью методов разных классов. Показано, что файлы классов приложения можно трактовать как ресурсы и искать теми же методами, что и другие ресурсы.

Листинг 8.5. Тестовый класс, демонстрирующий поиск ресурсов до JDK 9

```
// ResourceTest.java
package com.jdojo.resource.prejdk9;

import java.io.IOException;
import java.net.URL;
import java.util.Properties;

public class ResourceTest {
   public static void main(String[] args) {
      System.out.println("Поиск ресурсов с помощью системного загручика классов:");
      findSystemResource("java/lang/Object.class");
      findSystemResource("com/jdojo/resource/prejdk9/ResourceTest.class");
      findSystemResource("com/jdojo/prime/PrimeChecker.class");
      findSystemResource("sun/print/resources/duplex.png");

      System.out.println("\nПоиск ресурсов с помощью класса Class:");

      // Относительное имя ресурса - Object.class не будет найден
      findClassResource("java/lang/Object.class");

      // Абсолютное имя ресурса - Object.class будет найден
      findClassResource("/java/lang/Object.class");

      // Относительное имя ресурса - класс будет найден
      findClassResource("ResourceTest.class");

      // Загрузить файл wordtonumber.properties
      loadProperties("/wordtonumber.properties");

      // Файл свойств не будет найден, потому что используется
      // аболютное имя ресурса
      loadProperties("/resources/numbertoword.properties");

      // Файл свойств будет найден
      loadProperties("resources/numbertoword.properties");
   }

   public static void findSystemResource(String resource) {
```

```
        URL url = ClassLoader.getSystemResource(resource);
        System.out.println(url);
    }

    public static URL findClassResource(String resource) {
        URL url = ResourceTest.class.getResource(resource);
        System.out.println(url);
        return url;
    }

    public static Properties loadProperties(String resource) {
        Properties p1 = new Properties();
        URL url = ResourceTest.class.getResource(resource);
        if (url == null) {
            System.out.println("Свойства не найдены: " + resource);
            return p1;
        }

        try {
            p1.load(url.openStream());
            System.out.println("Свойства загружены из " + resource);
            System.out.println(p1);
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }

        return p1;
    }
}
```

```
Поиск ресурсов с помощью системного загручика классов:
jar:file:/C:/java8/jre/lib/rt.jar!/java/lang/Object.class
file:/C:/Java9Revealed/com.jdojo.resource.prejdk9/build/classes/com/jdojo/resource/
prejdk9/
ResourceTest.class
null
jar:file:/C:/java8/jre/lib/resources.jar!/sun/print/resources/duplex.png

Поиск ресурсов с помощью класса Class:
null
jar:file:/C:/java8/jre/lib/rt.jar!/java/lang/Object.class
file:/C:/Java9Revealed/com.jdojo.resource.prejdk9/build/classes/com/jdojo/resource/
prejdk9/
ResourceTest.class
Свойства загружены из /wordtonumber.properties
{One=1, Three=3, Four=4, Five=5, Two=2}
Свойства не найдены: /resources/numbertoword.properties
```

```
Свойства загружены из resources/numbertoword.properties
{5=Five, 4=Four, 3=Three, 2=Two, 1=One}
```

Доступ к ресурсам в JDK 9

Раньше можно было получить доступ к ресурсам в любом JAR-файле на пути к классам. В JDK 9 классы и ресурсы инкапсулированы в модулях. При первой попытке проектировщики JDK 9 применили к ресурсам в модуле правила инкапсуляции: ресурсы считались частным владением модуля и, следовательно, были доступны *только* коду внутри этого модуля. Теоретически это правило выглядит замечательно, но создает проблемы каркасам, которые разделяли ресурсы между модулями и загружали файлы классов как ресурсы из других модулей. В результате был выбран компромисс – разрешить *ограниченный* доступ к ресурсам в модулях, оставив в силе инкапсуляцию. В JDK 9 методы поиска ресурсов находятся в трех классах:

- `java.lang.Class`
- `java.lang.ClassLoader`
- `java.lang.Module`

Классы `Class` и `ClassLoader` не получили никаких новых методов. В классе `Module` имеется метод `getResourceAsStream(String name)`, который возвращает объект `InputStream`, если ресурс найден, и `null` в противном случае.

Синтаксис именования ресурсов

Имя ресурса – последовательность строк, разделенных знаком `/`, например: `com/jdojo/states.png`, `/com/jdojo/words.png`, `logo.png`. Если имя начинается знаком `/`, то оно считается абсолютным.

Имя пакета образуется по имени ресурса по следующим правилам:

- если имя ресурса начинается знаком `/`, удалить этот знак. Например, имя `/com/jdojo/words.png` преобразуется в `com/jdojo/words.png`;
- удалить из имени ресурса все символы, начиная с последнего знака `/`. Имя `com/jdojo/words.png` преобразуется в `com/jdojo`;
- заменить все оставшиеся знаки `/` точкой. Строка `com/jdojo` преобразуется в `com.jdojo`. Получившаяся строка и является именем пакета.

Бывает, что после этих шагов получается пустое или недопустимое имя пакета. Напомним, что имя пакета должно состоять из допустимых идентификаторов Java. Если получилось пустое имя, то пакет называется безымянным. Рассмотрим имя ресурса `META-INF/resource/logo.png`. В результате применения правил получается строка `META-INF.resources`, которая не является допустимым именем пакета, но является допустимым путем к ресурсу.

Правила поиска ресурсов

Из-за необходимости поддерживать обратную совместимость и строгую инкапсуляцию модулей новые правила поиска ресурсов в JDK 9 сложны и учитывают несколько факторов:

- тип модуля, содержащего ресурса: именованный, раскрытый, безымянный или автоматический;
- модуль, обращающийся к ресурсу: содержащий ресурс или другой;
- имя пакета требуемого ресурса: допустимое или недопустимое, является ли пакет безымянным;
- инкапсуляция пакета, содержащего ресурс: является ли экспортированным, раскрытым или инкапсулированным с точки зрения модуля, обращающегося к ресурсу;
- расширение имения файла ресурса: это файл с расширением `.class` или каким-то другим;
- метод какого класса используется для доступа к ресурсу: `Class`, `ClassLoader` или `Module`.

Следующие правила применяются к ресурсу, находящемуся в именованном модуле.

- Если имя ресурса оканчивается на `.class`, то к ресурсу разрешен доступ из любого модуля, т. е. любой модуль может получить доступ к файлам классов в любом именованном модуле.
- Если имя пакета, образованное из имени ресурса, не является допустимым именем пакета Java, как, например, `META-INF.resources`, то к ресурсу разрешен доступ из любого модуля.
- Если имя пакета, образованное из имени ресурса, пусто, как, например, в случае `words.png`, то к ресурсу разрешен доступ из любого модуля.
- Если пакет, содержащий ресурс, раскрыт для модуля, пытающегося получить доступ к ресурсу, то доступ разрешен. Пакет может быть раскрыт для модуля, потому что модуль, в котором находится пакет, является раскрытым или потому что этот модуль раскрывает пакет только определенным модулям в квалифицированном предложении `opens`. Если пакет не раскрыт ни одним из этих способов, то содержащиеся в этом пакете ресурсы недоступны коду вне данного модуля.
- Это правило является следствием предыдущего. Любой пакет в безымянном, автоматическом или раскрытом модуле раскрыт, поэтому все ресурсы в таких модулях доступны из любого другого модуля.

Совет. Чтобы были доступны ресурса пакета, находящегося в именованном модуле, этот пакет должен быть именно раскрыт, а не экспортирован. Экспорт пакета дает другим модулям доступ к открытым типам в этом пакете, но не к ресурсам.

Различные методы поиска ресурсов в классах `Module`, `Class` и `ClassLoader` ведут себя по-разному при доступе к ресурсам в именованных модулях.

- Для доступа к ресурсу в модуле можно использовать метод `getResourceAsStream()` класса `Module`. Поведение этого метода зависит от того, кто его вызвал. Если он вызван из другого модуля, то метод применяет все описанные выше правила доступа к ресурсам.
- Методы `getResource*()` класса `Class` для класса, определенного в именованном модуле, находят ресурсы только в этом модуле. То есть их нельзя использо-

вать для поиска класса вне именованного модуля, где определен класс, от имени которого метод вызван.

- Методы `getResource*()` класса `ClassLoader` находят ресурсы в именованных модулях, опираясь на сформулированные выше правила. Эти методы не зависят от того, кто их вызвал. Загрузчик класса делегирует поиск ресурса своему родителю до того, как пытается найти ресурс сам. Для этих методов есть два исключения. 1) Эти методы находят ресурсы только в безусловно раскрытых пакетах. Если пакет раскрыт конкретным модулям с помощью квалифицированного предложения `opens`, то эти методы не найдут в нем ресурсов. 2) Они производят поиск в модулях, назначенных загрузчику класса.

Объект `Class` находит ресурсы только в модуле, частью которого является. Он поддерживает абсолютные и относительные имена ресурсов. Приведем несколько примеров использования объекта `Class`:

```
// Ресурс будет найден
URL url1 = Test.class.getResource("Test.class");

// Ресурс не будет найден, т. к. классы Test и Object находятся в разных модулях
URL url2 = Test.class.getResource("/java/lang/Object.class");

// Ресурс будет найден, т. к. классы Object и Class находятся в одном модуле,
// java.base
URL url3 = Object.class.getResource("/java/lang/Class.class");

// Ресурс не будет найден, т. к. класс Object находится в модуле java.base,
// а класс Driver – в модуле java.sql
URL url4 = Object.class.getResource("/java/sql/Driver.class");
```

Чтобы использовать класс `Module` для поиска ресурсов, необходимо иметь ссылку на модуль. Если у вас есть доступ к какому-нибудь классу из этого модуля, то вызов метода `getModule()` для этого объекта `Class` вернет ссылку на модуль. Это самый простой способ получить ссылку. Но иногда есть имя модуля в строковом виде, а ссылки на класс из этого модуля нет. Тогда можно получить ссылку по имени. Модули организованы в несколько слоев, представленных экземплярами класса `ModuleLayer` из пакета `java.lang`. JVM содержит по меньшей мере один слой, называемый начальным (boot layer). Модули начального слоя отображаются на встроенные загрузчики классов – базовый, платформенный и системный. Ссылку на начальный слой возвращает статический метод `boot()` класса `ModuleLayer`:

```
// Получить ссылку на начальный слой
ModuleLayer bootLayer = ModuleLayer.boot();
```

Имея ссылку на начальный слой, мы можем использовать его метод `findModule(String moduleName)` для получения ссылки на модуль:

```
// Найти модуль com.jdojo.resource в начальном слое
Optional<Module> m = bootLayer.findModule("com.jdojo.resource");

// Если модуль найден, искать в нем ресурс
```

```
if(m.isPresent()) {
    Module testModule = m.get();
    String resource = "com/jdojo/resource/opened/opened.properties";
    InputStream input = module.getResourceAsStream(resource);
    if (input != null) {
        System.out.println(resource + " найден.");
    } else {
        System.out.println(resource + " не найден.");
    }
} else {
    System.out.println("Модуль com.jdojo.resource не существует");
}
```

Пример доступа к ресурсам в именованных модулях

В этом разделе мы продемонстрируем правила поиска ресурсов в действии. Ресурсы будут упакованы в модуле `com.jdojo.resource`, объявление которого приведено в листинге 8.6.

Листинг 8.6. Объявление модуля com.jdojo.resource

```
// module-info.java
module com.jdojo.resource {
    exports com.jdojo.exported;

    opens com.jdojo.opened;
}
```

Модуль экспортирует пакет `com.jdojo.exported` и раскрывает пакет `com.jdojo.opened`. Ниже перечислены все файлы в модуле `com.jdojo.resource`:

- `module-info.class`
- `unnamed.properties`
- `META-INF\invalid_pkg.properties`
- `com\jdojo\encapsulated\encapsulated.properties`
- `com\jdojo\encapsulated\EncapsulatedTest.class`
- `com\jdojo\exported\AppResource.class`
- `com\jdojo\exported\exported.properties`
- `com\jdojo\opened\opened.properties`
- `com\jdojo\opened\OpenedTest.class`

Существует четыре файла класса. Интерес представляет только файл `module-info.class`, а в остальных файлах определен одноименный класс без какого-либо содержимого. Все файлы с расширением `.properties` — ресурсные файлы, содержимое которых не играет роли. В исходном коде, прилагаемом к книге, эти файлы находятся в каталоге `Java9Revealed\com.jdojo.resource`.

Файл `unnamed.properties` находится в безымянном пакете, поэтому доступен из любого модуля. Файл `invalid_pkg.properties` находится в каталоге `META-INF`, которому соответствует недопустимое имя пакета, поэтому этот файл также доступен из

любого модуля. Пакет `com.jdojo.encapsulated` не раскрытый, поэтому файл `encapsulated.properties` недоступен из других модулей. Пакет `com.jdojo.exported` не раскрытый, поэтому файл `exported.properties` недоступен из других модулей. Пакет `com.jdojo.opened` раскрытый, поэтому файл `opened.properties` доступен из любого модуля. Все файлы классов в этом модуле могут быть найдены кодом, находящимся в других модулях.

В листинге 8.7 приведено объявление модуля `com.jdojo.resource.test`. Содержащийся в нем код пытается получить доступ к ресурсам в самом этом модуле и в модуле `com.jdojo.resource`. Для компиляции модуля `com.jdojo.resource` его нужно поместить на путь к модулям. На рис. 8.5 показано диалоговое окно свойств проекта `com.jdojo.resource.test` в NetBeans. В нем модуль `com.jdojo.resource` помещается на путь к модулям.

Листинг 8.7. Объявление модуля com.jdojo.resource.test

```
// module-info.java
module com.jdojo.resource.test {
    requires com.jdojo.resource;

    exports com.jdojo.resource.test;
}
```

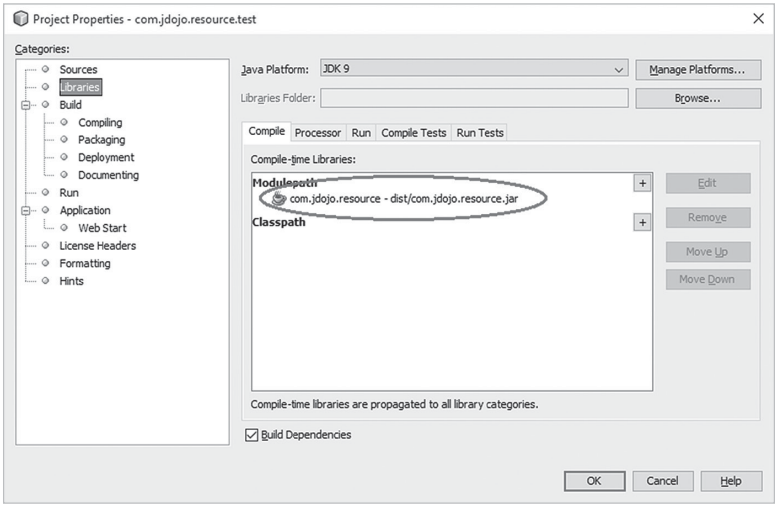

Рис. 8.5. Помещение модуля `com.jdojo.resource` на путь к модулям в проекте NetBeans `com.jdojo.resource.test`

Файлы в модуле `com.jdojo.resource.test` организованы следующим образом:

- `module-info.class`
- `com\jdojo\resource\test\own.properties`
- `com\jdojo\resource\test\ResourceTest.class`

Модуль содержит ресурсный файл `own.properties` в пакете `com.jdojo.resource.test`. Этот файл пуст. В листинге 8.8 приведен код класса `ResourceTest`.

Листинг 8.8. Класс ResourceTest для демонстрации доступа к ресурсам в именованных модулях

```
// ResourceTest
package com.jdojo.resource.test;

import com.jdojo.exported.AppResource;
import java.io.IOException;
import java.io.InputStream;

public class ResourceTest {
   public static void main(String[] args) {
     // Список ресурсов
     String[] resources = {
        "java/lang/Object.class",
        "com/jdojo/resource/test/own.properties",
        "com/jdojo/resource/test/ResourceTest.class",
        "unnamed.properties",
        "META-INF/invalid_pkg.properties",
        "com/jdojo/opened/opened.properties",
        "com/jdojo/exported/AppResource.class",
        "com/jdojo/resource/exported.properties",
        "com/jdojo/encapsulated/EncapsulatedTest.class",
        "com/jdojo/encapsulated/encapsulated.properties"
     };

     System.out.println("Используется Module:");
     Module otherModule = AppResource.class.getModule();
     for (String resource : resources) {
        lookupResource(otherModule, resource);
     }

     System.out.println("\nИспользуется Class:");
     Class cls = ResourceTest.class;
     for (String resource : resources) {
        // Добавить / в начало всех имен ресурсов, чтобы сделать их абсолютными
        lookupResource(cls, "/" + resource);
     }

     System.out.println("\nИспользуется системный ClassLoader:");
     ClassLoader clSystem = ClassLoader.getSystemClassLoader();
     for (String resource : resources) {
        lookupResource(clSystem, resource);
     }

     System.out.println("\nИспользуется платформенный ClassLoader:");
     ClassLoader clPlatform = ClassLoader.getPlatformClassLoader();
```

```
    for (String resource : resources) {
       lookupResource(clPlatform, resource);
    }
  }

  public static void lookupResource(Module m, String resource) {
     try {
        InputStream in = m.getResourceAsStream(resource);
        print(resource, in);
     } catch (IOException e) {
        System.out.println(e.getMessage());
     }
  }

  public static void lookupResource(Class cls, String resource) {
     InputStream in = cls.getResourceAsStream(resource);
     print(resource, in);
  }

  public static void lookupResource(ClassLoader cl, String resource) {
     InputStream in = cl.getResourceAsStream(resource);
     print(resource, in);
  }

  private static void print(String resource, InputStream in) {
     if (in != null) {
        System.out.println("Найден: " + resource);
     } else {
        System.out.println("Не найден: " + resource);
     }
  }
}
```

```
Используется Module:
Не найден: java/lang/Object.class
Не найден: com/jdojo/resource/test/own.properties
Не найден: com/jdojo/resource/test/ResourceTest.class
Найден: unnamed.properties
Найден: META-INF/invalid_pkg.properties
Найден: com/jdojo/opened/opened.properties
Найден: com/jdojo/exported/AppResource.class
Не найден: com/jdojo/resource/exported.properties
Найден: com/jdojo/encapsulated/EncapsulatedTest.class
Не найден: com/jdojo/encapsulated/encapsulated.properties

Используется Class:
```

```
Не найден: /java/lang/Object.class
Найден: /com/jdojo/resource/test/own.properties
Найден: /com/jdojo/resource/test/ResourceTest.class
Не найден: /unnamed.properties
Не найден: /META-INF/invalid_pkg.properties
Не найден: /com/jdojo/opened/opened.properties
Не найден: /com/jdojo/exported/AppResource.class
Не найден: /com/jdojo/resource/exported.properties
Не найден: /com/jdojo/encapsulated/EncapsulatedTest.class
Не найден: /com/jdojo/encapsulated/encapsulated.properties

Используется системный ClassLoader:
Найден: java/lang/Object.class
Найден: com/jdojo/resource/test/own.properties
Найден: com/jdojo/resource/test/ResourceTest.class
Найден: unnamed.properties
Найден: META-INF/invalid_pkg.properties
Найден: com/jdojo/opened/opened.properties
Найден: com/jdojo/exported/AppResource.class
Не найден: com/jdojo/resource/exported.properties
Найден: com/jdojo/encapsulated/EncapsulatedTest.class
Не найден: com/jdojo/encapsulated/encapsulated.properties

Используется платформенный ClassLoader:
Найден: java/lang/Object.class
Не найден: com/jdojo/resource/test/own.properties
Не найден: com/jdojo/resource/test/ResourceTest.class
Не найден: unnamed.properties
Не найден: META-INF/invalid_pkg.properties
Не найден: com/jdojo/opened/opened.properties
Не найден: com/jdojo/exported/AppResource.class
Не найден: com/jdojo/resource/exported.properties
Не найден: com/jdojo/encapsulated/EncapsulatedTest.class
Не найден: com/jdojo/encapsulated/encapsulated.properties
```

Метод `lookupResource()` перегружен. Его варианты ищут ресурсы с помощью трех классов: `Module`, `Class` и `ClassLoader`. Имя ресурса и ссылка на него передаются методу `print()`, который печатает сообщение.

Метод `main()` готовит массив строк, содержащий список имен ресурсов, которые хочет искать различными методами:

```
// Список ресурсов
String[] resources = {/* List of resources */};
```

Метод `main()` пытается найти ресурсы, пользуясь ссылкой на модуль `com.jdojo.resource`. Отметим, что класс `AppResource` находится в модуле `com.jdojo.resource`, так что вызов `AppResource.class.getModule()` возвращает ссылку на этот модуль.

```
System.out.println("Using a Module:");
Module otherModule = AppResource.class.getModule();
for (String resource : resources) {
    lookupResource(otherModule, resource);
}
```

Этот код нашел все файлы классов и все ресурсы в безымянных, недопустимых и раскрытых пакетах модуля `com.jdojo.resource`. Отметим, что файл `java/lang/Object.class` не был найден, потому что он находится в модуле `java.base` а не `com.jdojo.resource`. Ресурсы, находящиеся в модуле `com.jdojo.resource.test`, не найдены по той же причине.

Далее метод `main()` ищет те же ресурсы с помощью объекта `Class`, представляющего класс `ResourceTest`, находящийся в модуле `com.jdojo.resource.test`.

```
Class cls = ResourceTest.class;
for (String resource : resources) {
    // Добавить / в начало всех имен ресурсов, чтобы сделать их абсолютными
    lookupResource(cls, "/" + resource);
}
```

Как показывает распечатка, этот объект находит ресурсы только в модуле `com.jdojo.resource.test`. Я дописал в начало имени ресурса знак `/`, поскольку методы поиска ресурсов в классе `Class` считают имя, не начинающееся знаком косой черты, относительным и добавляют к нему имя пакета, содержащего класс.

Наконец, метод `main()` применяет системный и платформенный загрузчики для поиска всё тех же ресурсов:

```
ClassLoader clSystem = ClassLoader.getSystemClassLoader();
for (String resource : resources) {
    lookupResource(clSystem, resource);
}

ClassLoader clPlatform = ClassLoader.getPlatformClassLoader();
for (String resource : resources) {
    lookupResource(clPlatform, resource);
}
```

Загрузчик классов находит ресурсы во всех модулях, известных ему самому или одному из его предков. Системный загрузчик загружает модули `com.jdojo.resource` и `com.jdojo.resource.test`, поэтому находит в них ресурсы, соблюдая ограничения, налагаемые правилами поиска. Его дед, базовый загрузчик, загрузил класс `Object` из модуля `java.base`, поэтому системный загрузчик сможет найти файл `java/lang/Object.class`.

Платформенный загрузчик не загружает модули приложения `com.jdojo.resource` и `com.jdojo.resource.test`. Из распечатки видно, что платформенный загрузчик нашел только один ресурс `java/lang/Object.class`, который был загружен его родителем, базовым загрузчиком.

Доступ к ресурсам в образе среды выполнения

Рассмотрим несколько примеров доступа к ресурсам в образе среды выполнения. Раньше для этого можно было использовать статический метод `getSystemResource()` класса `ClassLoader`. Ниже приведен код поиска файла `Object.class` в JDK 8:

```
import java.net.URL;
...
String resource = "java/lang/Object.class";
URL url = ClassLoader.getSystemResource(resource);
System.out.println(url);
```

```
jar:file:/C:/java8/jre/lib/rt.jar!/java/lang/Object.class
```

Как видим, возвращенный URL-адрес имеет схему `jar` и указывает на файл `rt.jar`. В JDK 9 образ среды выполнения хранится не в JAR-файлах, а во внутреннем формате, который в будущем может измениться. JDK предоставляет независимый от формата и местоположения способ доступа к ресурсам среды выполнения с помощью схемы `jrt`. Приведенный выше код работает в JDK 9, но возвращаемый URL имеет схему `jrt`, а не `jar`:

```
jrt:/java.base/java/lang/Object.class
```

> **Совет.** Программу, которая обращается к ресурсам из образа среды выполнения и ожидает получить URL со схемой `jar`, необходимо модифицировать в JDK 9, т. к. возвращенный URL имеет схему `jrt`.

Синтаксически схема `jrt` имеет вид:

```
jrt:/<module-name>/<path>
```

Здесь `<module-name>` – имя модуля, а `<path>` – путь к конкретному файлу класса или ресурса в этом модуле. И `<module-name>`, и `<path>` факультативны. URL `jrt:/` обозначает все файлы классов и ресурсов в образе среды выполнения, `jrt:/<module-name>` – все файлы классов и ресурсов в модуле `<module-name>`, а `jrt:/<module-name>/<path>` – файл класса или ресурса с именем `<path>` в модуле `<module-name>`. Ниже приведено два примера URL со схемой `jrt`, адресующих файл класса и файл ресурса:

- `jrt:/java.sql/java/sql/Driver.class`
- `jrt:/java.desktop/sun/print/resources/duplex.png`

Первый URL адресует файл класса `java.sql.Driver` в модуле `java.sql`, второй – файл изображения `sun/print/resources/duplex.png` в модуле `java.desktop`.

> **Совет.** Для доступа к ресурсам в образе среды выполнения с применением схемы `jrt` можно воспользоваться методами поиска ресурсов в классах `Module`, `Class` и `ClassLoader`.

Объект URL со схемой jrt можно создать. В следующем фрагменте показано, как прочитать из образа среды выполнения файл изображения в объект Image и файл класса в массив байтов.

```
// Загрузить файл duplex.png в объект Image
URL imageUrl = new URL("jrt:/java.desktop/sun/print/resources/duplex.png");
Image image = ImageIO.read(imageUrl);

// Использовать объект изображения
System.out.println(image);

// Загрузить содержимое файла Object.class
URL classUrl = new URL("jrt:/java.base/java/lang/Object.class");
InputStream input = classUrl.openStream();
byte[] bytes = input.readAllBytes();
System.out.println("Размер файла Object.class: " + bytes.length);
```

```
BufferedImage@3e57cd70: type = 6 ColorModel: #pixelBits = 32 numComponents = 4 color space =
java.awt.color.ICC_ColorSpace@67b467e9 transparency = 3 has alpha = true isAlphaPre =
false ByteInterleavedRaster: width = 41 height = 24 #numDataElements 4 dataOff[0] = 3
Размер файла Object.class: 1859
```

Когда имеет смысл использовать другие варианты схемы jrt: для представления всех файлов в образе и всех файлов в модуле? URL, адресующий модуль, можно использовать для предоставления прав в файле политики Java. Следующая запись в файле политики предоставляет все права коду в модуле java.activation:

```
grant codeBase "jrt:/java.activation" {
    permission java.security.AllPermission;
}
```

Во многих инструментальных средствах и IDE возникает необходимость перечислить все модули, пакеты и файлы в образе среды выполнения. В состав JDK 9 входит поставщик предназначенной только для чтения файловой системы NIO для схемы jrt. Его можно использовать для построения списка всех файлов классов и ресурсов в образе. Существуют инструменты и IDE, которые работают в JDK 8, но должны поддерживать разработку кода для JDK 9. Этим инструментам также нужен список классов и ресурсов в образе среды выполнения JDK 9. После установки JDK 9 в каталоге lib появится файл jrt-fs.jar. Вы можете добавить этот JAR-файл на путь к классам для инструментов, работающих в JDK 8 и использовать объект FileSystem со схемой jrt, как описано ниже.

В файловой системе jrt имеется корневой каталог, представленный косой чертой (/), который содержит два подкаталога:

```
/
/packages
/modules
```

Следующий код создает объект `FileSystem` для схемы `jrt`:

```
// Создать объект FileSystem со схемой jrt
FileSystem fs = FileSystems.getFileSystem(URI.create("jrt:/"));
```

Далее мы читаем файл изображения и содержимое файла `Object.class`:

```
// Загрузить изображение из модуля
Path imagePath = fs.getPath("modules/java.desktop",
                            "sun/print/resources/duplex.png");
Image image = ImageIO.read(Files.newInputStream(imagePath));

// Использовать объект image
System.out.println(image);

// Прочитать содержимое файла Object.class
Path objectClassPath = fs.getPath("modules/java.base", "java/lang/Object.class");
byte[] bytes = Files.readAllBytes(objectClassPath);
System.out.println("Размер файла Object.class: " + bytes.length);
```

```
BufferedImage@5f3a4b84: type = 6 ColorModel: #pixelBits = 32 numComponents = 4 color space =
java.awt.color.ICC_ColorSpace@5204062d transparency = 3 has alpha = true isAlphaPre =
false ByteInterleavedRaster: width = 41 height = 24 #numDataElements 4 dataOff[0] = 3
image Object.class: 1859
```

Следующий код печатает все элементы – файлы классов и ресурсов – из всех модулей в образе среды выполнения. Можно создать аналогичный объект `Path` для перечисления всех пакетов в образе.

```
// Перечислить все модули в образе среды выполнения
Path modules = fs.getPath("modules");
Files.walk(modules)
     .forEach(System.out::println);
```

```
/modules
/modules/java.base
/modules/java.base/java
/modules/java.base/java/lang
/modules/java.base/java/lang/Object.class
/modules/java.base/java/lang/AbstractMethodError.class
...
```

Рассмотрим полную программу, которая обращается к ресурсам в образе среды выполнения. В листинге 8.9 приведено объявление модуля `com.jdojo.resource.jrt`, а в листинге 8.10 – исходный код класса `JrtFileSystem` в этом модуле.

Листинг 8.9. Объявление модуля com.jdojo.resource.jrt

```
// module-info.java
module com.jdojo.resource.jrt {
    requires java.desktop;
}
```

Листинг 8.10. Класс JrtFileSystem, демонстрирующий использование схемы URL jrt для доступа к ресурсам в образе среды выполнения

```
// JrtFileSystem.java
package com.jdojo.resource.jrt;

import java.awt.Image;
import java.io.IOException;
import java.net.URI;
import java.nio.file.FileSystem;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;
import javax.imageio.ImageIO;

public class JrtFileSystem {
   public static void main(String[] args) throws IOException {
      // Создать объект FileSystem со схемой jrt
      FileSystem fs = FileSystems.getFileSystem(URI.create("jrt:/"));

      // Загрузить изображение из модуля
      Path imagePath = fs.getPath("modules/java.desktop",
                                  "sun/print/resources/duplex.png");
      Image image = ImageIO.read(Files.newInputStream(imagePath));

      // Использовать объект image
      System.out.println(image);

      // Прочитать содержимое файла Object.class
      Path objectClassPath = fs.getPath("modules/java.base",
                                        "java/lang/Object.class");
      byte[] bytes = Files.readAllBytes(objectClassPath);
      System.out.println("Object.class file size: " + bytes.length);

      // Перечислить 5 пакетов в образе среды выполнения
      Path packages = fs.getPath("packages");
      Files.walk(packages)
           .limit(5)
           .forEach(System.out::println);
```

```
        // Перечислить 5 модулей в образе среды выполнения
        Path modules = fs.getPath("modules");
        Files.walk(modules)
             .limit(5)
             .forEach(System.out::println);
    }
}
```

```
BufferedImage@5bfbf16f: type = 6 ColorModel: #pixelBits = 32 numComponents = 4 color space =
java.awt.color.ICC_ColorSpace@27d415d9 transparency = 3 has alpha = true isAlphaPre =
false ByteInterleavedRaster: width = 41 height = 24 #numDataElements 4 dataOff[0] = 3
Object.class file size: 1859
packages
packages/com
packages/com/java.activation
packages/com/java.base
packages/com/java.corba
modules
modules/java.desktop
modules/java.desktop/sun
modules/java.desktop/sun/print
modules/java.desktop/sun/print/resources
```

Отметим, что эта программа печатает только по пять элементов, относящихся к пакетам и модулям. Кроме того, заметим, что мы смогли получить доступ к файлу `sun/print/resources/duplex.png`, находящемуся в модуле `java.desktop`, хотя он не раскрывает пакет `sun.print.resources`. Никакой метод поиска ресурсов из классов `Module`, `Class` и `ClassLoader` не смог бы найти файл `sun/print/resources/duplex.png`.

Внутренние API JDK

JDK содержит открытые и внутренние API. Открытые API предназначены для разработки переносимых Java-приложений. Пакеты `java.*`, `javax.*` и `org.*` содержат открытые API. Гарантируется, что приложение, пользующееся только открытыми API, будет работать во всех операционных системах, поддерживающих Java. Кроме того, гарантируется, что если такое приложение работало в версии JDK *n*, то оно будет работать и в версии *n* + 1.

Пакеты `com.sun.*`, `sun.*` и `jdk.*` используются в реализации самого JDK и составляют набор внутренних API, не предназначенных для разработчиков. Не гарантируется, что внутренние API переносимы с одной операционной системы на другую. Пакеты `com.sun.*` и `sun.*` – часть Oracle JDK. В JDK других поставщиков их нет, но есть другие пакеты, реализующие внутренние API. На рис. 8.6 показаны различные категории JDK API.

До перехода на модули в JDK 9 можно было использовать любые открытые классы из любого JAR-файла, даже составляющие внутренний API. Разработчики, а также авторы нескольких широко распространенных библиотек использовали

внутренние API JDK ради удобства или потому что предоставляемую ими функциональность трудно было реализовать вне JDK. Примерами могут служить классы `BASE64Encoder` и `BASE64Decoder`. Они использовались, потому что уже существовали в пакете `sun.misc`, хотя реализовать их самостоятельно совсем нетрудно. Другой широко используемый класс – `Unsafe` из пакета `sun.misc`. Разработать его замену вне JDK трудно, потому что он обращается к внутренним механизмам JDK.

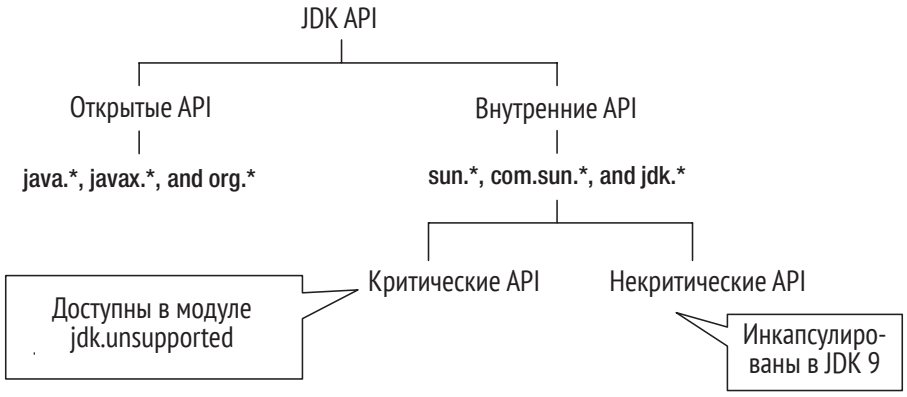

Рис. 8.6. Распределение JDK API по категориям на основе предназначения

Те внутренние API, которые используются исключительно для удобства, не используются вне JDK или для которых существует официально поддерживаемая замена, в JDK 9 отнесены к категории некритических и инкапсулированы. К таковым относятся классы `BASE64Encoder` и `BASE64Decoder` в пакете `sun.misc`, которые в JDK 8 были включены в пакет `java.util` под именами `Base64.Encoder` и `Base64.Decoder` и стали частью открытого API.

Внутренние API, которые широко использовались, но вряд ли могут быть разработаны вне JDK, отнесены к критическим. В JDK 9 они инкапсулированы, если существует замена. Критические внутренние API, которые инкапсулированы в JDK 9, но могут быть использованы, если задать параметр командной строки, снабжены аннотацией `@jdk.Exported`. В JDK 9 нет замены следующим классам, отнесенным к критическим внутренним API. Они доступны посредством модуля `jdk.unsupported`.

- `com.sun.nio.file.ExtendedCopyOption`
- `com.sun.nio.file.ExtendedOpenOption`
- `com.sun.nio.file.ExtendedWatchEventModifier`
- `com.sun.nio.file.SensitivityWatchEventModifier`
- `sun.misc.Signal`
- `sun.misc.SignalHandler`
- `sun.misc.Unsafe`
- `sun.reflect.Reflection`
- `sun.reflect.ReflectionFactory`

Совет. В JDK 9 большинство внутренних API инкапсулировано в модулях и по умолчанию недоступно. Однако к ним все же можно получить доступ, воспользовавшись нестандартным параметром командной строки `--add-reads`.

Методы `addPropertyChangeListener()` и `removePropertyChangeListener()` в перечисленных ниже классах были объявлены нерекомендуемыми в JDK 8 и исключены в JDK 9:

- `java.util.logging.LogManager`
- `java.util.jar.Pack200.Packer`
- `java.util.jar.Pack200.Unpacker`

Программа `jdeps`, находящаяся в каталоге `JAVA_HOME\bin`, позволяет найти зависимости вашего кода от внутренних API JDK на уровне классов. При этом следует задать еще параметр `--jdk-internals`:

```
jdeps --jdk-internals --class-path <class-path> <input-path>
```

Здесь `<input-path>` – путь к файлу класса, каталогу или JAR-файлу. Программа анализирует все классы на путях `<input-path>` и `<class-path>`. Следующая команда печатает сведения об использовании внутренних API в файле `jersey-common.jar` в предположении, что этот JAR-файл находится в каталоге `C:\Java9Revealed\extlib`. Ниже показана часть вывода:

```
C:\Java9Revealed>jdeps --jdk-internals extlib\jersey-common.jar

jersey-common.jar -> jdk.unsupported
   org.glassfish.jersey.internal.util.collection.ConcurrentHashMapV8 -> sun.misc.
Unsafe                   JDK internal API (jdk.unsupported)

org.glassfish.jersey.internal.util.collection.ConcurrentHashMapV8$TreeBin -> sun.misc.
Unsafe                   JDK internal API (jdk.unsupported)
...
```

Замена модуля

Иногда возникает необходимость заменить файлы классов и ресурсов в некоторых модулях другой версией с целью тестирования и отладки. Раньше для этого можно было воспользоваться параметром `Xbootclasspath/p`. В JDK 9 этот параметр не поддерживается и следует использовать нестандартный параметр `--patch-module` в командах `javac` и `java`:

```
--patch-module <module-name>=<path-list>
```

Здесь `<module-name>` – имя заменяемого модуля, а `<path-list>` – список JAR-файлов или каталогов, где находится его новое содержимое; элементы списка должны быть разделены платформенно-зависимым символом-разделителем (точка с запятой в Windows и двоеточие в UNIX-подобных системах).

Параметр `--patch-module` можно указывать несколько раз в одной строке и тем самым заменить содержимое сразу нескольких модулей. Заменять можно прикладные, библиотечные и платформенные модули.

Совет. Параметр `--patch-module` не позволяет заменить файлы `module-info.class`. Попытка сделать это молчаливо игнорируется.

Рассмотрим пример замены модуля `com.jdojo.intro`. Мы заменим в нем файл `Welcome.class`. Напомним, что класс `Welcome` был написан в главе 3. Новый класс будет выводить другое сообщение. Его объявление показано в листинге 8.11. Исходный код находится в проекте NetBeans `com.jdojo.intro.patch`.

Листинг 8.11. Новое объявление класса Welcome

```
// Welcome.java
package com.jdojo.intro;

public class Welcome {
   public static void main(String[] args) {
      System.out.println("Привет, система модулей.");

      // Напечатать имя модуля, содержащего класс Welcome
      Class<Welcome> cls = Welcome.class;
      Module mod = cls.getModule();
      String moduleName = mod.getName();
      System.out.format("Имя модуля: %s%n", moduleName);
   }
}
```

Откомпилируйте этот класс командой

```
C:\Java9Revealed>javac -Xmodule:com.jdojo.intro
 --module-path com.jdojo.intro\dist
 -d patches\com.jdojo.intro.patch com.jdojo.intro.patch\src\com\jdojo\intro\Welcome.
java
```

Эта команда будет работать даже без параметров `-Xmodule` и `--module-path`. Но эти параметры нужны, если вы собираетесь компилировать платформенный класс, например `java.util.Arrays`. Параметр `-Xmodule` задает имя модуля, которому принадлежит компилируемый класс, а параметр `--module-path` говорит, где искать этот модуль. Эти параметры используются, чтобы найти другие классы, необходимые для компиляции нового. В данном случае класс `Welcome` не зависит ни от каких классов в модуле `com.jdojo.intro`, и потому-то исключение этих параметров не повлияет на результат. Параметр `-d` говорит, где сохранять откомпилированный файл `Welcome.class`.

Следующая команда выполняет первоначальный класс `Welcome` из модуля `com.jdojo.intro`:

```
C:\Java9Revealed>java --module-path com.jdojo.intro\dist
 --module com.jdojo.intro/com.jdojo.intro.Welcome
```

```
Добро пожаловать в систему модулей.
Имя модуля: com.jdojo.intro
```

Теперь выполним измененный вариант класса `Welcome`:

```
C:\Java9Revealed>java --module-path com.jdojo.intro\dist
 --patch-module com.jdojo.intro=patches\com.jdojo.intro.patch
 --module com.jdojo.intro/com.jdojo.intro.Welcome
```

```
Привет, система модулей.
Имя модуля: com.jdojo.intro
```

Если задан параметр `--patch-module`, то система модулей просматривает указанные в нем пути раньше, чем путь к модулям. Отметим, что указанные пути ведут к содержимому некоторого модуля, но не являются путями к модулям.

Резюме

В JDK 9 сделано несколько несовместимых изменений, о которых следует знать при переносе унаследованных приложений.

Интуитивно неочевидная схема нумерации версий JDK в JDK 9 полностью переработана. Строка версии JDK состоит из четырех элементов в указанном порядке: номер версии, признак предварительной версии, информация о сборке и дополнительная информация. Обязателен только номер версии. Формат строки версии определяется регулярным выражением `$vnum(-$pre)?(\+($build)?(-$opt)?)?`. Короткая строка версии включает только первые два элемента: номер версии и, возможно, признак предварительной версии. Строка версии может содержать как всего лишь основной номер версии – `"9"`, так и все элементы – `"9.0.1-ea+154-20170130.07.36am"`.

В JDK 9 появился статический вложенный класс `Runtime.Version`, экземпляр которого представляет строку версии JDK. В этом классе нет открытого конструктора и единственный способ получить его экземпляр – вызвать статический метод `parse(String vstr)`. Если строка версии равна `null` или недопустима, то этот метод возбуждает исключение. Класс предоставляет несколько методов для получения различных частей версии.

В JDK 9 изменена структура каталогов JDK и JRE. Теперь единственное различие между установкой JDK и JRE состоит в том, что в JDK входят инструменты разработчика и платформенные модули в формате JMOD, а в JRE их нет. Вы можете построить собственную среду выполнения JRE (с помощь программы `jlink`), включающую любую необходимую вам часть JDK.

До Java SE 9 существовал механизм переопределения утвержденных стандартов, который позволял использовать обновленные версии классов и интерфейсов реализующих соответствующие API, в частности, пакет `javax.rmi.CORBA` и Java API для обработки XML (JAXP), созданные не в рамках процесса Java Community Process. В Java SE 9 этот механизм по-прежнему поддерживается, но нужно задавать параметр командной строки `--upgrademodule-path`. Значением этого параметра является список каталогов, содержащих модули, реализующие API утвержденных стандартов и автономные API.

До Java SE 9 существовал механизм расширения, позволявший расширить образ среды выполнения посредством размещения JAR-файлов в каталогах, заданных системным свойством `java.ext.dirs`. Если это свойство не было задано, то по умолчанию подразумевался каталог `jre\lib\ext`. В Java SE 9 механизм расширения не поддерживается. Если необходима аналогичная функциональность, то можно поместить JAR-файлы в начало пути к классам.

Раньше в JDK существовало три загрузчика классов: базовый загрузчик, загрузчик расширений и системный загрузчик. Они образовывали иерархию – у базового загрузчика не было родителя, родителем загрузчика расширений был базовый загрузчик, а родителем системного загрузчик – загрузчик расширений. Каждый загрузчик классов сначала делегировал запрос своему родителю, а затем уже пытался загрузить тип сам. В JDK 9 ради обратной совместимости сохранена трехзвенная иерархия загрузчиков, но, поскольку механизм расширения не поддерживается, то загрузчик расширений не имеет смысла. Вместо него в JDK 9 появился платформенный загрузчик, ссылку на который можно получить от статического метода `getPlatformClassLoader()` класса `ClassLoader`. Каждый загрузчик загружает модули разных типов.

В JDK 9 ресурсы в именованном модуле по умолчанию инкапсулированы. К ресурсу в именованном модуле может получить доступ код в другом модуле, только если ресурс находится в безымянном, недопустимом или раскрытом пакете. Все ресурсы в именованном модуле, имена которые заканчиваются расширением `.class` (файлы классов), доступны любому другому модулю. Для доступа к ресурсам в образе среды выполнения можно использовать URL-адрес со схемой `jrt`.

Раньше можно было использовать внутренние API JDK. В JDK 9 большинство внутренних API инкапсулировано, но некоторые сделаны доступными посредством модуля `jdk.unsupported`. Чтобы узнать, от каких внутренних API на уровне классов зависит ваша программа, воспользуйтесь программой `jdeps` с параметром `--jdk-internals`.

Иногда возникает необходимость заменить классы файлов и ресурсов в определенных модулях другими версиями с целью тестирования и отладки. Раньше это можно было сделать с помощью параметра `-Xbootclasspath/p`, но в JDK 9 он не поддерживается. В JDK 9 для этой цели используется нестандартный параметр `--patch-module`, доступный в командах `javac` и `java`.

Глава 9

Нарушение инкапсуляции модуля

Краткое содержание главы:

- что такое нарушение инкапсуляции модуля;
- как добавить модулю зависимость с помощью параметра командной строки;
- как экспортировать неэкспортированные пакеты модуля с помощью параметра `--add-exports` и файла `MANIFEST.MF` исполняемого JAR-файла;
- как раскрыть нераскрытые пакеты модуля с помощью параметра `--add-opens` и файла `MANIFEST.MF` исполняемого JAR-файла;
- как расширить список зависимостей модуля с помощью параметра `--add-reads`.

Что такое нарушение инкапсуляции модуля?

Одна из основных целей JDK 9 – инкапсулировать типы и ресурсы в модулях и экспортировать только те пакеты, в которых находятся открытые типы, предназначенные для использования другими модулями. Но иногда требуется нарушить инкапсуляцию, например, чтобы тестировать модуль как белый ящик или чтобы воспользоваться неподдерживаемыми внутренними API JDK или библиотек. Это возможно благодаря нестандартным параметрам командной строки, задаваемым на этапах компиляции и выполнения. Еще одна причина существования этих параметров – обеспечение обратной совместимости. Не все существующие приложения будут в полном объеме перенесены на JDK 9 и приведены к модульному виду. Если такое приложение нуждается в API JDK или библиотеки, которые раньше были открыты, а в JDK 9 стали инкапсулированными, то необходимо сделать так, чтобы оно все же могло работать. Для некоторых предназначенных для этой цели параметров существуют соответственные атрибуты, которые можно включить в файл `MANIFEST.MF` исполняемых JAR-файлов и тем самым избежать задания параметров в командной строке.

Совет. Для каждого параметра командной строки, предназначенного для нарушения инкапсуляции, поддерживается возможность сделать то же самое из программы с помощью API модулей, рассматриваемого в главе 10.

Может показаться, что эти параметры просто восстанавливают положение, существовавшее до JDK 9, но следует предостеречь от неограниченного доступа к внутренним API JDK. Если пакет в модуле не экспортирован и не раскрыт, значит, у проектировщика были причины не предоставлять этот пакет для использования вне модуля. Такие пакеты могут быть модифицированы или даже исключены без предупреждения. Если вы все же настаиваете на их экспорте или раскрытии с помощью параметров командной строки, то принимаете на себя всю ответственность за последствия – ваше приложение в будущем может перестать работать!

Параметры командной строки

Существуют три предложения в объявлении модуля, которые обеспечивают инкапсуляцию типов и ресурсов и предоставляют доступ к ним другим модулям: `exports`, `opens` и `requires`. Каждому предложению соответствует параметр командной строки, а предложениям `exports` и `opens` еще и атрибуты в файле манифеста внутри JAR-файла. Все они перечислены в табл. 9.1.

Таблица 9.1. Предложения модуля и соответствующие параметры командной строки и атрибуты манифеста

Предложение модуля	Параметр командной строки	Атрибут манифеста
`exports`	`--add-exports`	`Add-Exports`
`opens`	`--add-opens`	`Add-Opens`
`requires`	`--add-reads`	`Нет атрибута`

Совет. В одной команде может встречаться несколько параметров `--add-exports`, `--add-opens`, `--add-reads`.

Параметр --add-exports

Предложение `exports` экспортирует пакет из модуля, так чтобы другие модули могли пользоваться открытыми API этого пакета. Если модуль не экспортирует пакет, то это можно сделать с помощью параметра командной строки `--add-exports`:

```
--add-exports <source-module>/<package>=<target-module-list>
```

Здесь `<source-module>` – модуль, который экспортирует пакет `<package>` модулям из списка `<target-module-list>`, перечисленным через запятую. Это эквивалентно добавлению квалифицированного предложения `exports` в объявление модуля `<source-module>`:

```
module <source-module> {
    exports <package> to <target-module-list>;

    // Другие предложения
}
```

> **Совет.** Если вместо списка модулей в параметре `--add-exports` указано специальное значение `ALL-UNNAMED`, то пакет экспортируется всем безымянным модулям. Параметр `--add-exports` поддерживается командами `javac` и `java`.

Следующий параметр экспортирует пакет `sun.util.logging` из модуля `java.base` модулям `com.jdojo.test` и `com.jdojo.prime`:

```
--add-exports java.base/sun.util.logging=com.jdojo.test,com.jdojo.prime
```

А следующий параметр экспортирует пакет `sun.util.logging` из модуля `java.base` всем безымянным модулям:

```
--add-exports java.base/sun.util.logging=ALL-UNNAMED
```

Параметр --add-opens

Предложение `opens` раскрывает пакет из модуля, так чтобы все или только некоторые модули могли использовать глубокую рефлексию для доступа ко всем определенным в этом пакете типам на этапе выполнения. Если модуль не раскрывает пакет, то это можно сделать с помощью параметра командной строки `--add-opens`:

```
--add-opens <source-module>/<package>=<target-module-list>
```

Здесь `<source-module>` – модуль, который раскрывает пакет `<package>` модулям из списка `<target-module-list>`, перечисленным через запятую. Это эквивалентно добавлению квалифицированного предложения `opens` в объявление модуля `<source-module>`:

```
module <source-module> {
    opens <package> to <target-module-list>;

    // Другие предложения
}
```

> **Совет.** Если вместо списка модулей в параметре `--add-opens` указано специальное значение `ALL-UNNAMED`, то пакет раскрывается всем безымянным модулям. Параметр `--add-opens` поддерживается командой `java`. Попытка указать его на этапе компиляции в команде `javac` приводит к выдаче предупреждения и не дает никакого эффекта.

Следующий параметр раскрывает пакет `sun.util.logging` из модуля `java.base` модулям `com.jdojo.test` и `com.jdojo.prime`:

```
--add-opens java.base/sun.util.logging=com.jdojo.test,com.jdojo.prime
```

А следующий параметр экспортирует пакет `sun.util.logging` из модуля `java.base` всем безымянным модулям:

```
--add-opens java.base/sun.util.logging=ALL-UNNAMED
```

Параметр --add-reads

Параметр `--add-reads` призван не нарушить инкапсуляцию модуля, а расширить список его зависимостей. В процессе тестирования и отладки модуля иногда необходимо, чтобы он мог прочесть другой модуль, пусть даже и не зависит от него.

Предложение `requires` служит для объявления зависимости данного модуля от другого. Параметр `--add-reads` можно использовать, чтобы добавить ребро чтения между модулями. Это эквивалентно добавлению предложения `requires` в объявление первого модуля. Синтаксически параметр имеет вид:

```
--add-reads <source-module>=<target-module-list>
```

Здесь `<source-module>` — модуль, чье определение модифицируется с целью чтения модулей из списка `<target-module-list>`, перечисленных через запятую. Это эквивалентно добавлению предложения `requires` в объявление модуля `<source-module>` для каждого модуля из списка `<target-module-list>`:

```
module <source-module> {
    requires <target-module1>;
    requires <target-module2>;

    // Другие предложения
}
```

> **Совет.** Если вместо списка модулей в параметре `--add-reads` указано специальное значение `ALL-UNNAMED`, то модуль читает все безымянные модули. Это единственный способ сделать так, чтобы именованный модуль прочитал безымянные. Никакого эквивалентного предложения модуля для этой цели не существует. Параметр можно указывать на этапах компиляции и выполнения.

Следующий параметр добавляет ребро чтения между модулем `com.jdojo.common` и модулем `jdk.accessibility`:

```
--add-reads com.jdojo.common=jdk.accessibility
```

Параметр --permit-illegal-access

Три вышеупомянутых параметра предназначены только для обеспечения обратной совместимости. Но их использование оказывается трудоемким, если требуется «незаконный» доступ (рефлексивный доступ к недоступным членам типов в модуле). Для таких случаев в команде `java` предусмотрен параметр `--permit-illegal-access`, который разрешает незаконный доступ из любого безымянного модуля (кода, найденного на пути к классам) к членам типов в любом именованном модуле с помощью глубокой рефлексии:

```
java --permit-illegal-access <other-options-and-arguments>
```

Параметр `--permit-illegal-access` не разрешает незаконный доступ из именованного модуля к членам типов в других именованных модулях. В таких случаях необходимо комбинировать этот параметр с параметрами `--add-exports`, `--add-opens` и `--add-reads options`.

> **Совет.** Параметр `--permit-illegal-access` поддерживается в JDK 9, но будет исключен в JDK 10. При его использовании в стандартный поток ошибок выводятся предупреждения. Одно из них говорит, что параметр будет удален в будущих версиях, а в другом приводятся сведения о коде, которому был разрешен незаконный доступ, о коде, к которому предоставлен незаконный доступ, и о параметре, благодаря которому это разрешение дано.

В следующем разделе я приведу пример использования всех параметров для нарушения инкапсуляции.

Пример

Рассмотрим пример нарушения инкапсуляции. Несмотря на тривиальность, он послужит для демонстрации всех соответствующих концепций и параметров командной строки.

В самом начале книги мы создали модуль `com.jdojo.intro`, содержащий класс `Welcome` в пакете `com.jdojo.intro`. Поскольку модуль не экспортирует пакет, класс `Welcome` инкапсулирован и доступ к нему извне запрещен. В этом примере мы вызовем метод `main()` класса `Welcome` из модуля `com.jdojo.intruder`, объявление которого приведено в листинге 9.1. В листинге 9.2 приведен код класса `TestNonExported` в этом модуле.

Листинг 9.1. Объявление модуля com.jdojo.intruder

```
// module-info.java
module com.jdojo.intruder {
    // Предложений модуля нет
}
```

Листинг 9.2. Класс TestNonExported

```
// TestNonExported.java
package com.jdojo.intruder;

import com.jdojo.intro.Welcome;

public class TestNonExported {
   public static void main(String[] args) {
      Welcome.main(new String[]{});
   }
}
```

Класс `TestNonExported` содержит всего одну строку кода – вызывает метод `main()` класса `Welcome`, передавая ему пустой массив строк. Если откомпилировать и выполнить этот класс, то будет напечатано то же сообщение, что при выполнении самого класса `Welcome` в главе 3:

```
Добро пожаловать в систему модулей.
Имя модуля: com.jdojo.intro
```

Попытавшись откомпилировать код модуля `com.jdojo.intruder`, мы получим сообщение об ошибке:

```
C:\Java9Revealed>javac --module-path com.jdojo.intro\dist
 -d com.jdojo.intruder\build\classes
```

```
 com.jdojo.intruder\src\module-info.java com.jdojo.intruder\src\com\jdojo\intruder\
 TestNonExported.java
```

```
com.jdojo.intruder\src\com\jdojo\intruder\TestNonExported.java:4: error: package com.
jdojo.intro is not visible
import com.jdojo.intro.Welcome;
               ^
 (package com.jdojo.intro is declared in module com.jdojo.intro, but module com.
jdojo.
 intruder does not read it)
1 error
```

В этой команде используется параметр `--module-path`, чтобы поместить `com.jdojo.intro` на путь к модулям. Сообщение указывает на предложение `import`, в котором импортируется класс `com.jdojo.intro.Welcome`. Оно говорит, что пакет `com.jdojo.intro` невидим модулю `com.jdojo.intruder`, т. е. модуль `com.jdojo.intro` не экспортирует пакет, содержащий класс `Welcome`. Чтобы исправить эту ошибку, мы должны экспортировать пакет `com.jdojo.intro` из модуля `com.jdojo.intro` модулю `com.jdojo.intruder`, воспользовавшись параметром `--add-exports`:

```
C:\Java9Revealed>javac --module-path com.jdojo.intro\dist
 --add-exports com.jdojo.intro/com.jdojo.intro=com.jdojo.intruder
 -d com.jdojo.intruder\build\classes
 com.jdojo.intruder\src\module-info.java com.jdojo.intruder\src\com\jdojo\intruder\
 TestNonExported.java
```

```
warning: [options] module name in --add-exports option not found: com.jdojo.intro
com.jdojo.intruder\src\com\jdojo\intruder\TestNonExported.java:4: error: package
com.jdojo.intro is not visible
import com.jdojo.intro.Welcome;
               ^
  (package com.jdojo.intro is declared in module com.jdojo.intro, but module
  com.jdojo.intruder does not read it)
1 error
1 warning
```

На этот раз мы получили предупреждение и ошибку – ту же, что и раньше. Предупреждение говорит, что компилятор не смог найти модуль `com.jdojo.intro`. Поскольку зависимости от этого модуля не существует, то он не разрешается, хотя и находится на пути к модулям. Чтобы избавиться от предупреждения, нужно добавить модуль `com.jdojo.intro` в набор корневых модулей по умолчанию, воспользовавшись параметром `--add-modules`:

```
C:\Java9Revealed>javac --module-path com.jdojo.intro\dist
 --add-modules com.jdojo.intro
 --add-exports com.jdojo.intro/com.jdojo.intro=com.jdojo.intruder
```

```
 -d com.jdojo.intruder\build\classes
 com.jdojo.intruder\src\module-info.java
 com.jdojo.intruder\src\com\jdojo\intruder\TestNonExported.java
```

Эта команда `javac` завершается успешно, хотя модуль `com.jdojo.intruder` не читает `com.jdojo.intro`. Похоже на ошибку. Как бы то ни было, я не смог найти описание такого поведения в документации. Позже мы увидим, что команда `java` для тех же модулей не работает. Если эта команда выдаст сообщение о том, что класс `TestNonExported` не может получить доступ к классу `Welcome`, то добавьте такой параметр:

```
--add-reads com.jdojo.intruder=com.jdojo.intro
```

Попробуем снова выполнить класс `TestNonExported` с помощью команды, которая помещает `com.jdojo.intruder` на путь к модулям:

```
C:\Java9Revealed>java --module-path com.jdojo.intro\dist;com.jdojo.intruder\build\
classes
 --add-modules com.jdojo.intro
 --add-exports com.jdojo.intro/com.jdojo.intro=com.jdojo.intruder
 --module com.jdojo.intruder/com.jdojo.intruder.TestNonExported
```

```
Exception in thread "main" java.lang.IllegalAccessError: class com.jdojo.intruder.
TestNonExported (in module com.jdojo.intruder) cannot access class com.jdojo.intro.
Welcome (in module com.jdojo.intro) because module com.jdojo.intruder does not read
module com.jdojo.intro at com.jdojo.intruder/com.jdojo.intruder.TestNonExported.
main(TestNonExported.java:8)
```

Сообщение об ошибке совершенно недвусмысленно. Оно говорит, что модуль `com.jdojo.intruder` должен прочитать модуль `com.jdojo.intro`, чтобы использовать входящий в него класс `Welcome`. Ошибку исправит параметр `--add-reads`, который добавляет ребро чтения (эквивалент предложения `requires`) между модулями `com.jdojo.intruder` и `com.jdojo.intro`:

```
C:\Java9Revealed>java --module-path com.jdojo.intro\dist;com.jdojo.intruder\build\classes
 --add-modules com.jdojo.intro
 --add-exports com.jdojo.intro/com.jdojo.intro=com.jdojo.intruder
 --add-reads com.jdojo.intruder=com.jdojo.intro
 --module com.jdojo.intruder/com.jdojo.intruder.TestNonExported
```

```
Добро пожаловать в систему модулей.
Имя модуля: com.jdojo.intro
```

Вот теперь мы получили желаемый результат. На рис. 9.1 показан граф модулей, созданный в процессе выполнения этой команды.

И `com.jdojo.intruder`, и `com.jdojo.intro` – корневые модули. Модуль `com.jdojo.intruder` добавлен в набор корневых, потому что в нем находится исполняемый главный класс, а модуль `com.jdojo.intro` – благодаря параметру `--add-modules`. Ребро чтения от модуля `com.jdojo.intruder` к модулю `com.jdojo.intro` добавлено благодаря парамет-

ру `--add-reads`. Я изобразил это ребро штриховой линией, чтобы показать, что оно добавлено уже после построения графа модулей. Чтобы увидеть, как разрешаются модули при выполнении этой команды, добавьте параметр `-Xdiag:resolver`.

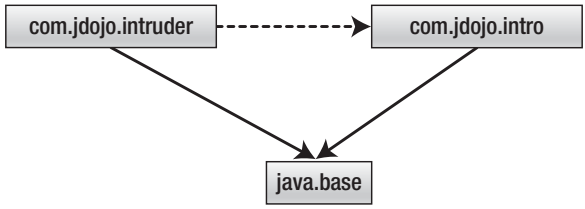

Рис. 9.1. Граф модулей после добавления параметров `--add-modules` и `--add-reads`

Рассмотрим другой пример, демонстрирующий, как раскрыть пакет другому модулю с помощью параметра `--add-opens`. В главе 4 мы написали модуль `com.jdojo.address`, содержащий класс `Address` в пакете `com.jdojo.address`. Этот модуль экспортирует пакет `com.jdojo.address`. Класс содержит закрытое поле `line1`, в котором хранится первая строка адреса. Открытый метод `getLine1()` возвращает значение этого поля.

Класс `TestNonOpen`, показанный в листинге 9.3, пытается загрузить класс `Address`, создать его экземпляр и обратиться к открытым и закрытым членам. Класс `TestNonOpen` входит в модуль `com.jdojo.intruder`. Я указал несколько исключений в объявлении метода `main()`, чтобы не усложнять его код. В реальной программе следовало бы обработать их в блоке `try-catch`.

Листинг 9.3. Класс TestNonOpen

```
// TestNonOpen.java
package com.jdojo.intruder;

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class TestNonOpen {
   public static void main(String[] args)
         throws IllegalAccessException, IllegalArgumentException,
         NoSuchMethodException, ClassNotFoundException,
         InvocationTargetException, InstantiationException,
         NoSuchFieldException {

      String className = "com.jdojo.address.Address";

      // Получить ссылку на класс
      Class<?> cls = Class.forName(className);

      // Получить конструктор без аргументов
```

```
        Constructor constructor = cls.getConstructor();

        // Создать объект класса Address
        Object address = constructor.newInstance();

        // Вызвать метод getLine1() и получить значение line1
        Method getLine1Ref = cls.getMethod("getLine1");
        String line1 = (String)getLine1Ref.invoke(address);
        System.out.println("С помощью ссылки на метод, Line1: " + line1);

        // Прочитать значение из закрытого поля line1
        Field line1Field = cls.getDeclaredField("line1");
        line1Field.setAccessible(true);
        String line11 = (String)line1Field.get(address);
        System.out.println("С помощью ссылки на закрытое поле, Line1: " + line11);
    }
}
```

Попробуем откомпилировать класс TestNonOpen:

```
C:Java9revealed> javac -d com.jdojo.intruder\build\classes
 com.jdojo.intruder\src\com\jdojo\intruder\TestNonOpen.java
```

Компилируется без ошибок. Отметим, что код пытается обратиться к классу `Address` с помощью глубокой рефлексии, а компилятор ничего не знает о том, что этому классу не разрешено читать класс `Address` и его закрытые поля. Теперь попробуем выполнить класс `TestNonOpen`:

```
C:Java9revealed> java --module-path com.jdojo.address\dist;com.jdojo.intruder\build\classes
 --add-modules com.jdojo.address
 --module com.jdojo.intruder/com.jdojo.intruder.TestNonOpen
```

```
С помощью ссылки на метод, Line 1: 1111 Main Blvd.
Exception in thread "main" java.lang.reflect.InaccessibleObjectException: Unable to make
field private java.lang.String com.jdojo.address.Address.line1 accessible: module com.
jdojo.address does not "opens com.jdojo.address" to module com.jdojo.intruder
        at java.base/java.lang.reflect.AccessibleObject.checkCanSetAccessible(Accessible
        Object.java:207)
        at java.base/java.lang.reflect.Field.checkCanSetAccessible(Field.java:171)
        at java.base/java.lang.reflect.Field.setAccessible(Field.java:165)
        at com.jdojo.intruder/com.jdojo.intruder.TestNonOpen.main(TestNonOpen.java:35)
```

Я добавил `com.jdojo.address` в набор корневых модулей с помощью параметра `--add-modules`. Мы сумели создать экземпляр класса `Address`, хотя `com.jdojo.intruder` не читает модуль `com.jdojo.address`. Тому есть две причины.

- Модуль `com.jdojo.address` экспортирует пакет `com.jdojo.address`, содержащий класс `Address`. Следовательно, класс `Address` доступен другим модулям при условии, что они читают модуль `com.jdojo.address`.

- API рефлексии в Java предполагает возможность чтения для всех рефлексивных операций. Это правило предполагает, что `com.jdojo.intruder` читает `com.jdojo.address`, даже если в объявлении модуля `com.jdojo.intruder` нет предложения `reads com.jdojo.address`. Если бы мы захотели использовать типы из пакета `com.jdojo.address` на этапе выполнения, например, объявили бы переменную класса `Address`, то модуль `com.jdojo.intruder` должен был бы прочитать `com.jdojo.address` либо в объявлении, либо на уровне командной строки.

Как видим, класс `TestNonOpen` смог вызвать открытый метод `getLine1()` класса `Address`. Однако при попытке обратиться к закрытому полю `line1` возбуждено исключение. Напомним, что если модуль экспортирует тип, то другие модули могут обращаться к открытым членам этого типа с помощью рефлексии. Но чтобы другой модуль мог получить доступ к закрытым членам типа, пакет, содержащий этот тип, должен быть раскрыт. Пакет `com.jdojo.address` не раскрыт, поэтому модуль `com.jdojo.intruder` не может получить доступ к закрытому полю `line1` класса `Address`. Чтобы исправить ситуацию, укажем параметр `--add-opens`, который раскроет пакет `com.jdojo.address` модулю `com.jdojo.intruder`:

```
C:Java9revealed> java --module-path com.jdojo.address\dist;com.jdojo.intruder\build\classes
 --add-modules com.jdojo.address
 --add-opens com.jdojo.address/com.jdojo.address=com.jdojo.intruder
 --module com.jdojo.intruder/com.jdojo.intruder.TestNonOpen
```

```
С помощью ссылки на метод, Line1: 1111 Main Blvd.
С помощью ссылки на закрытое поле, Line1: 1111 Main Blvd.
```

Настало время посмотреть на параметр `--permit-illegal-access` в действии. Попробуем выполнить класс `TestNonOpen`, поместив его на путь к классам:

```
C:\Java9Revealed>java --module-path com.jdojo.address\dist
 --class-path com.jdojo.intruder\build\classes
 --add-modules com.jdojo.address com.jdojo.intruder.TestNonOpen
```

```
С помощью ссылки на метод, Line1: 1111 Main Blvd.
Exception in thread "main" java.lang.reflect.InaccessibleObjectException: Unable to
make field private java.lang.String com.jdojo.address.Address.line1 accessible:
module com.jdojo.address does not "opens com.jdojo.address" to unnamed module @9f70c54
    at java.base/java.lang.reflect.AccessibleObject.checkCanSetAccessible(Accessible
    Object.java:337)
    at java.base/java.lang.reflect.AccessibleObject.checkCanSetAccessible(Accessible
    Object.java:281)
    at java.base/java.lang.reflect.Field.checkCanSetAccessible(Field.java:175)
    at java.base/java.lang.reflect.Field.setAccessible(Field.java:169)
    at com.jdojo.intruder.TestNonOpen.main(TestNonOpen.java:34)
```

Как видим, класс TestNonOpen, загруженный в безымянный модуль, поскольку находится на пути к классам, смог прочитать экспортированный тип и его открытые методы в модуле com.jdojo.address. Однако закрытую переменную-член ему прочитать не удалось. Это исправляет параметр --permit-illegal-access:

```
C:\Java9Revealed>java --module-path com.jdojo.address\dist
 --class-path com.jdojo.intruder\build\classes
 --add-modules com.jdojo.address
 --permit-illegal-access com.jdojo.intruder.TestNonOpen
```

```
WARNING: --permit-illegal-access will be removed in the next major release
Using method reference, Line1: 1111 Main Blvd.
WARNING: Illegal access by com.jdojo.intruder.TestNonOpen (file:/C:/Java9Revealed/
com.jdojo.intruder/build/classes/) to field com.jdojo.address.Address.line1 (permitted
by -permit-illegal-access)
С помощью ссылки на закрытое поле, Line1: 1111 Main Blvd.
```

Обратите внимание, что сообщения класса TestNonOpen и предупреждения касательно параметра --permit-illegal-access перемешаны.

Атрибуты манифеста JAR-файла

Исполняемым называется JAR-файл, который можно выполнить непосредственно, указав параметр -jar:

```
java -jar myapp.jar
```

В файле MANIFEST.MF внутри исполняемого JAR-файла имеется атрибут Main-Class, значением которого является полное имя главного класса, который будет выполнен показанной выше командой java. Напомним, что существуют и другие виды JAR-файлов, например, модульный и многоверсионный. Но для исполняемости это не играет роли – она определяется только в контексте использования параметра -jar.

Рассмотрим существующее приложение, поставляемое в виде исполняемого JAR-файла. Предположим, что в нем используется глубокая рефлексия для доступа к внутренним API JDK, и в JDK 8 оно прекрасно работало. Теперь мы хотим запустить это JAR-файл в JDK 9, где внутренние API инкапсулированы. Помимо параметра -jar, мы еще должны будем указать параметры --add-exports и --add-opens, так что решение есть. Однако для конечных пользователей исполняемого JAR-файла оно не очень удобно, поскольку они должны знать о новых параметрах командной строки в JDK 9. Чтобы упростить миграцию, было добавлено два новых атрибута в файл MANIFEST.MF:

- Add-Exports
- Add-Opens

Эти атрибуты задаются в главной секции манифеста и соответствуют параметрам --add-exports и --add-opens. Однако существует одно различие. Эти атрибуты

экспортируют и раскрывают пакеты всем безымянным модулям. То есть мы указываем список исходных модулей и их пакетов, но не указываем целевые модули. Иными словами, с помощью манифеста можно экспортировать или раскрыть пакет либо всем безымянным модулям, либо ни одному. Значением каждого атрибута является список пар имя-модуля/имя-пакета через запятую, например:

```
Add-Exports: m1/p1 m2/p2 m3/p3 m1/p1
```

Здесь мы экспортируем пакет `p1` из модуля `m1`, пакет `p2` из модуля `m2` и пакет `p3` из модуля `m3` всем безымянным модулям. Правила разбора файла манифеста не слишком строги и допускают наличие дубликатов, потому что так сложилось исторически. Обратите внимание на повторяющуюся пару `m1/p1` в значении атрибута.

Рассмотрим пример. Я специально сделал его простым, чтобы сосредоточиться на новых атрибутах. В классе `java.lang.Long` имеется закрытое статическое поле `serialVersionUID`, объявленное следующим образом:

```
private static final long serialVersionUID = 4290774380558885855L;
```

В листинге 9.4 приведен код класса `TestManifestAttributes`, в котором для доступа к этому полю используется глубокая рефлексия. Класс находится в модуле `com.jdojo.intruder`. В существующем приложении модули не используются, оно было разработано для версии JDK 8 или более ранней. Однако сейчас это неважно.

Листинг 9.4. Класс TestManifestAttributes

```
// TestManifestAttributes.java
package com.jdojo.intruder;

import java.lang.reflect.Field;

public class TestManifestAttributes {
   public static void main(String[] args) throws NoSuchFieldException,
                 IllegalArgumentException, IllegalAccessException {
      Class<Long> cls = Long.class;
      Field svUid = cls.getDeclaredField("serialVersionUID");
      svUid.setAccessible(true);
      long svUidValue = (long)svUid.get(null);
      System.out.println("Long.serialVersionUID=" + svUidValue);
   }
}
```

Класс `TestManifestAttributes` компилируется без ошибок. Давайте упакуем его в исполняемый JAR-файл. В листинге 9.5 показано, как выглядел бы файла `MANIFEST.MF` до версии JDK 9. Напомним, что этот файл хранится в каталоге `META-INF` в корне JAR-файла. До поры до времени не будем в нем ничего менять.

Листинг 9.5. Содержимое файла MANIFEST.MF

```
Manifest-Version: 1.0
Main-Class: com.jdojo.intruder.TestManifestAttributes
```

Следующая команда создает исполняемый JAR-файл `com.jdojo.intruder.jar` в каталоге `com.jdojo.intruder\dist`. То же самое можно было бы сделать, очистив и собрав проект `com.jdojo.intruder` в NetBeans.

```
C:\Java9Revealed>jar --create --file com.jdojo.intruder\dist\com.jdojo.intruder.jar
 --manifest=com.jdojo.intruder\src\META-INF\MANIFEST.MF
 -C com.jdojo.intruder\build\classes.

C:\Java9Revealed>java -jar com.jdojo.intruder\dist\com.jdojo.intruder.jar
```

```
Exception in thread "main" java.lang.reflect.InaccessibleObjectException: Unable to
make field private static final long java.lang.Long.serialVersionUID accessible:
module java.base does not "opens java.lang" to unnamed module @224aed64
    at java.base/java.lang.reflect.AccessibleObject.checkCanSetAccessible(Accessible
    Object.java:207)
    at java.base/java.lang.reflect.Field.checkCanSetAccessible(Field.java:171)
    at java.base/java.lang.reflect.Field.setAccessible(Field.java:165)
    at com.jdojo.intruder.TestManifestAttributes.main(TestManifestAttributes.java:10)
```

В сообщении об ошибке говорится, что приложение не может получить доступ к закрытому статическому полю `serialVersionUID`, потому что пакет `java.lang` в модуле `java.base` не раскрыт. В предыдущем разделе вы узнали, как раскрыть пакет с помощью параметра командной строки `-add-opens`. Сначала попробуем этот вариант:

```
C:\Java9Revealed>java --add-opens java.base/java.lang=ALL-UNNAMED
 -jar com.jdojo.intruder\dist\com.jdojo.intruder.jar
```

```
Long.serialVersionUID=4290774380558885855
```

Так команда работает. А теперь попробуем вместо этого воспользоваться атрибутом `Add-Opens` в файле `MANIFEST.MF`, как показано в листинге 9.6.

Листинг 9.6. Окончательный вариант файла MANIFEST.MF

```
Manifest-Version: 1.0
Main-Class: com.jdojo.intruder.TestManifestAttributes
Add-Opens: java.base/java.lang
```

Пересоздайте исполняемый JAR-файл той же командой, что и раньше, и выполните его:

```
C:\Java9Revealed>java -jar com.jdojo.intruder\dist\com.jdojo.intruder.jar
```

```
Long.serialVersionUID=4290774380558885855
```

Приложение работает. Убедимся, что атрибут `Add-Opens` игнорируется, если JAR-файл не используется как исполняемый. Проверить это просто. Если запустить приложение, поместив исполняемый JAR-файл на путь к классам или путь к модулям, то должна возникнуть ошибка. Отметим, что мы сможем выполнить это приложение из пути к модулям, потому что работаем в среде JDK 9 и в JAR-файле есть дескриптор модуля. Для более старых приложений существует только одна возможность – запуск из пути к классам:

```
C:\Java9Revealed>java --class-path com.jdojo.intruder\dist\com.jdojo.intruder.jar
com.jdojo.intruder.TestManifestAttributes
```

```
Exception in thread "main" java.lang.reflect.InaccessibleObjectException: Unable to
make field private static final long java.lang.Long.serialVersionUID accessible:
module java.base does not "opens java.lang" to unnamed module @17ed40e0
    at java.base/java.lang.reflect.AccessibleObject.checkCanSetAccessible(Accessible
    Object.java:207)
    at java.base/java.lang.reflect.Field.checkCanSetAccessible(Field.java:171)
    at java.base/java.lang.reflect.Field.setAccessible(Field.java:165)
    at com.jdojo.intruder.TestManifestAttributes.main(TestManifestAttributes.java:10)
```

Чтобы исправить эту ошибку, добавим параметр `--add-opens`:

```
C:\Java9Revealed>java --add-opens java.base/java.lang=ALL-UNNAMED
--class-path com.jdojo.intruder\dist\com.jdojo.intruder.jar com.jdojo.intruder.
TestManifestAttributes
```

```
Long.serialVersionUID=4290774380558885855
```

Резюме

Одна из основных целей JDK 9 – инкапсулировать типы и ресурсы в модулях и экспортировать только те пакеты, в которых находятся открытые типы, предназначенные для использования другими модулями. Но иногда требуется нарушить инкапсуляцию, например, чтобы тестировать модуль как белый ящик или чтобы воспользоваться неподдерживаемыми внутренними API JDK или библиотек. Это возможно благодаря нестандартным параметрам командной строки, задаваемым на этапах компиляции и выполнения. Еще одна причина существования этих параметров – обеспечение обратной совместимости.

В JDK 9 появилось два новых параметра командной строки, `--add-exports` и `--add-opens`, позволяющих нарушить инкапсуляцию, определенную в объявлении модуля. Параметр `--add-exports` служит для экспорта неэкспортированного пакета на этапах компиляции и выполнения, а параметр `--add-opens` – для раскрытия нераскрытого пакета на этапе выполнения. Значения этих параметров задаются в виде `<source-module>/<package>=<target-module-list>`, где `<source-module>` – модуль, который экспортирует или раскрывает пакет `<package>` модулям, перечисленным через

запятую в списке <target-module-list>. Вместо списка можно указать специальное значение ALL-UNNAMED, означающее, что пакет экспортируется или раскрывается всем безымянным модулям.

В главной секции манифеста исполняемого JAR-файла можно использовать два новых атрибута, Add-Exports и Add-Opens. Они дают тот же результат, что соответствующие параметры командной строки, но позволяют экспортировать или раскрывать указанные пакеты только всем безымянным модулям. Значение атрибута задается в виде списка пар имя-модуля/имя-пакета через запятую. Например, запись Add-Opens: java.base/java.lang означает, что пакет java.lang из модуля java.base раскрывается всем безымянным модулям.

В процессе тестирования и отладки иногда возникает необходимость, чтобы один модуль прочитал другой, хотя в объявлении первого модуля нет предложения requires для чтения второго. Сделать это можно, задав в командной строке параметр --add-reads, значение которого имеет вид <sourcemodule>=<target-module-list>, где <source-module> – модуль, чье определение модифицируется, так чтобы он читал модули, перечисленные в списке <target-module-list> через запятую. Если вместо списка указать специальное значение ALL-UNNAMED, то модуль будет читать все безымянные модули.

Глава 10

API модулей

Краткое содержание главы:

- что такое API модулей;
- представление модуля и его дескриптора в программе;
- чтение дескриптора модуля из программы;
- представление версии модуля;
- чтение свойств модуля с помощью классов `Module` и `ModuleDescriptor`;
- изменение определения модуля во время выполнения с помощью класса `Module`;
- создание и чтение аннотаций модулей;
- слои и конфигурации модулей;
- создание пользовательских слоев и загрузка в них модулей.

Что такое API модулей?

API модулей состоит из классов и интерфейсов, дающих программный доступ к модулям. С его помощью можно выполнять следующие операции:

- чтение, модификация и построение дескрипторов модулей;
- загрузка модулей;
- чтение содержимого модуля;
- поиск загруженных модулей;
- создание новых слоев модулей.

API модулей невелик – 15 классов и интерфейсов, распределенных между двумя пакетами: `java.lang` и `java.lang.module`.

Классы `Module`, `ModuleLayer` и `LayerInstantiationException` находятся в пакете `java.lang`, а все остальное – в пакете `java.lang.module`. В табл. 10.1 перечислены классы API модулей с краткими пояснениями. Список не отсортирован по именам. Классы `Module` и `ModuleDescriptor` поставлены на первое место, потому что на практике именно они применяются чаще всего. Остальные классы обычно используются в контейнерах и в библиотеках. В списке нет классов исключений, их я подробно рассмотрю в следующих разделах.

Таблица 10.1. Классы API модулей и их описания

Класс	Описание
`Module`	Представляет модуль на этапе выполнения
`ModuleDescriptor`	Представляет дескриптор модуля. Неизменяемый
`ModuleDescriptor.Builder`	Вложенный класс для построения дескрипторов модулей из программы
`ModuleDescriptor.Exports`	Вложенный класс, представляющий предложения `exports` в объявлении модуля
`ModuleDescriptor.Opens`	Вложенный класс, представляющий предложения `opens` в объявлении модуля
`ModuleDescriptor.Provides`	Вложенный класс, представляющий предложения `peovides` в объявлении модуля
`ModuleDescriptor.Requires`	Вложенный класс, представляющий предложения `requires` в объявлении модуля
`ModuleDescriptor.Version`	Вложенный класс, представляющий строку версии модуля. Включает фабричный метод `parse(String v)`, возвращающий объект этого класса по строке версии
`ModuleDescriptor.Modifier`	Перечисление, элементы которого представляют модификаторы в объявлении модуля, например, `OPEN` для раскрытого модуля
`ModuleDescriptor.Exports.Modifier`	Перечисление, элементы которого представляют модификаторы в предложения `exports` в объявлении модуля
`ModuleDescriptor.Opens.Modifier`	Перечисление, элементы которого представляют модификаторы в предложения `opens` в объявлении модуля
`ModuleDescriptor.Requires.Modifier`	Перечисление, элементы которого представляют модификаторы в предложения `requires` в объявлении модуля
`ModuleReference`	Ссылка на содержимое модуля. Содержит дескриптор модуля и его местоположение
`ResolvedModule`	Представляет разрешенный модуль в графе модулей. Содержит имя модуля, его зависимости и ссылку на содержимое. Может использоваться для обхода всех транзитивных зависимостей модуля в графе
`ModuleFinder`	Интерфейс для поиска модулей на указанных путях или системных модулей. Найденные модули возвращаются в виде экземпляров класса `ModuleReference`. Содержит фабричные методы для получения экземпляров
`ModuleReader`	Интерфейс для чтения содержимого модуля. Объект типа `ModuleReader` можно получить от класса `ModuleReference`
`Configuration`	Представляет граф разрешенных модулей
`ModuleLayer`	Содержит граф модулей (объект `Configuration`) и отображение между модулями в графе и загрузчиками классов
`ModuleLayer.Controller`	Вложенный класс для управления модулями в `ModuleLayer`. Экземпляры этого класса возвращаются методами класса `ModuleLayer`

Представление модулей

Экземпляр класса `Module` представляет модуль на этапе выполнения. Любой тип, загруженный в JVM, принадлежит какому-то модулю. В JDK 9 в классе `Class` появился новый метод `getModule()`, возвращающий модуль, которому принадлежит класс. Ниже показано, как получить модуль класса `BasicInfo`:

```
// Получить объект Class для класса BasicInfo
Class<BasicInfo> cls = BasicInfo.class;

// Получить ссылку на модуль
Module module = cls.getModule();
```

Модуль может быть именованным или безымянным. Метод `isNamed()` класса `Module` возвращает `true` для именованного модуля и `false` для безымянного.

Каждый загрузчик классов содержит безымянный модуль, в который помещаются все типы, найденные этим загрузчиком на пути к классам. Типы, найденные загрузчиком на пути к модулям, принадлежат именованным модулям. Метод `getModule()` класса `Class` может вернуть как именованный, так и безымянный модуль. В класс `ClassLoader` в JDK 9 добавлен метод `getUnnamedModule()`, который возвращает безымянный модуль загрузчика классов. Ниже показано, что в предположении, что класс `BasicInfo` найден на пути к классам, объекты `m1` и `m2` ссылаются на один и тот же экземпляр `Module`:

```
Class<BasicInfo> cls = BasicInfo.class;
Module m1 = cls.getClassLoader().getUnnamedModule();
Module m2 = cls.getModule();
```

Метод `getName()` класса `Module` возвращает имя модуля, а для безымянных модулей `null`.

```
// Получить имя модуля
String moduleName = module.getName();
```

Метод `getPackages()` класса `Module` возвращает множество `Set<String>`, содержащее все пакеты, принадлежащие модулю. Метод `getClassLoader()` возвращает загрузчик классов, ассоциированный с данным модулем.

Метод `getLayer()` возвращает объект `ModuleLayer`, содержащий модуль; если модуль не принадлежит никакому слою, то возвращается `null`. Слой содержит только именованные модули, поэтому для безымянных модулей этот метод всегда возвращает `null`.

Описание модулей

Экземпляр класса `ModuleDescriptor` представляет определение модуля, созданное по его объявлению, – обычно содержимое файла `module-info.class`. Дескриптор модуля можно создать динамически с помощью класса `ModuleDescriptor.Builder`. Объявление модуля можно пополнить, пользуясь параметрами командной строки `--add-reads`, `--add-exports` и `--add-opens` методами класса `Module`: `addReads()`, `addOpens()` и `addExports()`. Класс `ModuleDescriptor` представляет дескриптор, созданный в момент

объявления модуля и еще не пополненный. Объект `ModuleDescriptor` возвращается методом `getDescriptor()` класса `Module`:

```
Class<BasicInfo> cls = BasicInfo.class;
Module module = cls.getModule();

// Получить дескриптор модуля
ModuleDescriptor desc = module.getDescriptor();
```

> **Совет.** Объект `ModuleDescriptor` неизменяемый. У безымянного модуля нет дескриптора. Для безымянного модуля метод `getDescriptor()` возвращает `null`.

Объект `ModuleDescriptor` можно также создать, прочитав двоичную форму объявления модуля из файла `module-info.class` одним из статических методов `read()` класса `ModuleDescriptor`. Показанный ниже код читает файл `module-info.class` из текущего каталога. Обработка исключений для простоты опущена.

```
String moduleInfoPath = "module-info.class";
ModuleDescriptor desc = ModuleDescriptor.read(new FileInputStream(moduleInfoPath));
```

Представление предложений модуля

Класс `ModuleDescriptor` содержит следующие статические вложенные классы, экземпляры которых представляют одноименные предложения в объявлении модуля:

- `ModuleDescriptor.Exports`
- `ModuleDescriptor.Opens`
- `ModuleDescriptor.Provides`
- `ModuleDescriptor.Requires`

Отметим, что не существует класса `ModuleDescriptor.Uses`, представляющего предложение `uses`, поскольку оно описывает имя интерфейса службы, для представления которого достаточно обычной строки.

Представление предложения exports

Экземпляр класса `ModuleDescriptor.Exports` представляет предложение `exports`. Следующие методы этого класса возвращают компоненты предложения:

- `boolean isQualified()`
- `Set<ModuleDescriptor.Exports.Modifier> modifiers()`
- `String source()`
- `Set<String> targets()`

Метод `isQualified()` возвращает `true`, если экспорт квалифицированный, иначе `false`. Метод `source()` возвращает имя экспортированного пакета. В случае квалифицированного экспорта метод `targets()` возвращает неизменяемое множество имен модулей, которым экспортирован пакет, а в случае неквалифицированного – пустое множество. Метод `modifiers()` возвращает множество модификаторов предложения `exports` – элементов вложенного перечисления `ModuleDescriptor.Exports.Modifier`, которое содержит две константы:

- MANDATED: экспорт был неявно объявлен в объявлении модуля;
- SYNTHETIC: экспорт не был явно или неявно объявлен в объявлении модуля.

Представление предложения opens

Экземпляр класса ModuleDescriptor.Opens представляет предложение opens. Следующие методы этого класса возвращают компоненты предложения:

- boolean isQualified()
- Set<ModuleDescriptor.Opens.Modifier> modifiers()
- String source()
- Set<String> targets()

Метод isQualified() возвращает true, если раскрытие квалифицированное, иначе false. Метод source() возвращает имя раскрытого пакета. В случае квалифицированного раскрытия метод targets() возвращает неизменяемое множество имен модулей, которым раскрыт пакет, а в случае неквалифицированного – пустое множество. Метод modifiers() возвращает множество модификаторов предложения opens – элементов вложенного перечисления ModuleDescriptor.Opens.Modifier, которое содержит две константы:

- MANDATED: раскрытие было неявно объявлено объявлении модуля;
- SYNTHETIC: раскрытие не было явно или неявно объявлено в объявлении модуля.

Представление предложения provides

Экземпляр класса ModuleDescriptor.Provides представляет одно или несколько предложений provides для определенного типа службы. Следующие два предложения provides описывают два класса реализации одного и того же типа службы X.Y:

- provides X.Y with A.B;
- provides X.Y with Y.Z;

Оба предложения представлены одним экземпляром класса ModuleDescriptor.Provides. Следующие методы этого класса возвращают компоненты предложения provides:

- List<String> providers()
- String service()

Метод providers() возвращает список полных имен классов поставщиков. В нашем примере список будет содержать имена A.B и Y.Z. Метод service()возвращает полное имя типа службы, в данном случае X.Y.

Представление предложения requires

Экземпляр класса ModuleDescriptor. Requires представляет предложение requires. Следующие методы этого класса возвращают компоненты предложения:

- Optional<ModuleDescriptor.Version> compiledVersion()
- Optional<String> rawCompiledVersion()
- String name()
- Set<ModuleDescriptor.Requires.Modifier> modifiers()

Предположим, что компилируется модуль M, в объявлении которого присутствует предложение requires N. Если версия модуля N доступна в момент компиляции, то она записывается в дескриптор модуля M. Метод compiledVersion() возвращает эту запомненную версию в виде значения типа Optional. Если версия N не была доступна, то этот метод вернет пустой объект Optional. Версия модуля, указанного в предложении requires, запоминается в дескрипторе только для справки. Ни в каком месте системы модулей она не используется. Однако она может применяться инструментальными средствами и каркасами в целях диагностики. Например, инструмент может проверить, что для всех модулей, указанных в предложении requires, версия не ниже, чем была зафиксирована во время компиляции.

Метод rawCompiledVersion() возвращает версию модуля N в виде объекта Optional<String>. В большинстве случае методы compiledVersion() и rawCompiledVersion() возвращают одно и то же значение, но в разных форматах: Optional<ModuleDescriptor.Version> и Optional<String> соответственно. Бывает так, что версия модуля имеет недопустимый формат. Такой модуль можно создать и откомпилировать вне системы модулей Java и даже загрузить как Java-модуль. Тогда метод compiledVersion() вернет пустой объект Optional<ModuleDescriptor.Version>, а rawCompiledVersion() – объект Optional<String>, содержащий недопустимую версию.

> **Совет.** Метод rawCompiledVersion() класса ModuleDescriptor.Requires может вернуть строку версии затребованного модуля в формате, не допускающем разбора.

Метод name() возвращает имя модуля, указанного в предложении requires. Метод modifiers() возвращает множество модификаторов предложения requires – элементов вложенного перечисления ModuleDescriptor.Requires.Modifier, которое содержит следующие константы:

- MANDATED: зависимость была неявно объявлена в объявлении модуля;
- STATIC: зависимость обязательна на этапе компиляции и факультативна на этапе выполнения;
- SYNTHETIC: зависимость не была явно или неявно объявлена в объявлении модуля.
- TRANSITIVE: транзитивная зависимость – любой модуль, зависящий от данного, неявно будет объявлять зависимость от модуля, указанного в этом предложении requires.

Представление версии модуля

Экземпляр класса ModuleDescriptor.Version представляет версию модуля. Он содержит статический фабричный метод parse(String version), который возвращает разобранную версию. Напомним, что версия модуля не указывается в его объявлении, а добавляется на этапе упаковки в модульный JAR-файл, обычно командой jar. Компилятор javac также позволяет указать версию при компиляции модуля.

Строка версии модуля состоит из трех компонент:

- обязательный номер версии;
- факультативный признак предварительной версии;
- факультативная информация о сборке.

Версия модуля имеет вид:

```
vNumToken+ ('-' preToken+)? ('+' buildToken+)?
```

Каждая компонента представляет собой последовательность лексем; каждая лексема – неотрицательное целое число или строка. Лексемы разделены знаками препинания `.`, `-` или `+` либо переходом от последовательности цифр к последовательности знаков, не являющихся ни цифрами, ни знаками препинания, либо наоборот. Строка версии должна начинаться цифрой. Номер версии – это последовательность лексем, разделенных точками и заканчивающаяся первым знаком `-` или `+`. Предварительная версия – это последовательность лексем, разделенных знаками `.` или `-` и заканчивающаяся первым знаком `+`. Версия сборки – последовательность лексем, разделенных знаками `.`, `-` или `+`.

Метод `version()` класса `ModuleDescriptor` возвращает объект типа `Optional<ModuleDescriptor.Version>`.

Другие свойства модулей

Во время упаковки в модульный JAR-файл в файле `module-info.class` можно задать и другие свойства: имя главного класса, название ОС и т. п. Класс `ModuleDescriptor` содержит методы для возврата всех таких свойств, а именно:

- `Set<ModuleDescriptor.Exports> exports()`
- `boolean isAutomatic()`
- `boolean isOpen()`
- `Optional<String> mainClass()`
- `String name()`
- `Set<ModuleDescriptor.Opens> opens()`
- `Set<String> packages()`
- `Set<ModuleDescriptor.Provides> provides()`
- `Optional<String> rawVersion()`
- `Set<ModuleDescriptor.Requires> requires()`
- `String toNameAndVersion()`
- `Set<String> uses()`

Имена методов ясно сообщают об их назначении. Я остановлюсь только на двух, требующих пояснений: `packages()` и `provides()`.

В классе `ModuleDescriptor` есть метод `packages()`, а в классе `Module` – метод `getPackages()`. Тот и другой возвращает множество имен пакетов. Так зачем же два метода для одной цели? На самом деле, цели у них разные. Метод из класса `ModuleDescriptor` возвращает множество пакетов, перечисленных в объявлении модуля, – экспортированных и не экспортированных. Напомним, что получить объект `ModuleDescriptor` для безымянного модуля невозможно, так что имена пакетов в безымянном модуле может вернуть только метод `getPackages()` класса `Module`. Еще одно различие состоит в том, что множество имен пакетов, возвращенное объектом `ModuleDescriptor`, статическое, а возвращенное классом `Module` – динамическое, т. е. содержит те пакеты, что были загружены в модуль в момент вызова `getPackages()`. Класс `Module` сообщает обо всех пакетах, загруженные на этапе выполнения.

Метод `provides()` возвращает объект типа `Set<ModuleDescriptor.Provides>`. Рассмотрим следующие предложения `provides` в объявлении модуля:

```
provides A.B with X.Y1;
provides A.B with X.Y2;
provides P.Q with S.T1;
```

В данном случае множество содержит два элемента: для служб типа `A.B` и `P.Q`. Методы `service()` и `providers()` первого элемента возвращают `A.B` и список, содержащий `X.Y1` и `X.Y2` соответственно. А те же методы второго элемента возвращают `P.Q` и список из одного элемента `S.T1`.

Базовая информация о модуле

В этом разделе я покажу, как получить базовую информацию о модуле во время выполнения. В листинге 10.1 приведено объявление модуля `com.jdojo.module.api`. Он читает три модуля и экспортирует один пакет. Два из читаемых модулей, `com.jdojo.prime` и `com.jdojo.intro`, написаны в предыдущих главах. Поместите их на путь к модулям, затем откомпилируйте и выполните код из модуля `com.jdojo.module.api`. Модуль `java.sql` принадлежит JDK.

Листинг 10.1. Объявление модуля com.jdojo.module.api

```
// module-info.java
module com.jdojo.module.api {
   requires com.jdojo.prime;
   requires com.jdojo.intro;
   requires java.sql;
   exports com.jdojo.module.api;
}
```

В листинге 10.2 приведен код класса `ModuleBasicInfo`, который печатает сведения о трех модулях, пользуясь классами `Module` и `ModuleDescriptor`.

Листинг 10.2. Класс `ModuleBasicInfo`

```
// ModuleBasicInfo.java
package com.jdojo.module.api;

import com.jdojo.prime.PrimeChecker;
import java.lang.module.ModuleDescriptor;
import java.lang.module.ModuleDescriptor.Exports;
import java.lang.module.ModuleDescriptor.Provides;
import java.lang.module.ModuleDescriptor.Requires;
import java.sql.Driver;
import java.util.Set;

public class ModuleBasicInfo {
   public static void main(String[] args) {
      // Получить модуль текущего класса
      Class<ModuleBasicInfo> cls = ModuleBasicInfo.class;
```

```
        Module module = cls.getModule();

        // Напечатать сведения о модуле
        printInfo(module);
        System.out.println("------------------");

        // Напечатать сведения о модуле
        printInfo(PrimeChecker.class.getModule());
        System.out.println("------------------");

        // Напечатать сведения о модуле
        printInfo(Driver.class.getModule());
    }

    public static void printInfo(Module m) {
        String moduleName = m.getName();
        boolean isNamed = m.isNamed();

        // Напечатать тип и имя модуля
        System.out.printf("Имя модуля: %s%n", moduleName);
        System.out.printf("Именованный модуль: %b%n", isNamed);

        // Получить дескриптор модуля
        ModuleDescriptor desc = m.getDescriptor();

        // desc будет равен null, если модуль безымянный
        if (desc == null) {
            Set<String> currentPackages = m.getPackages();
            System.out.printf("Пакеты: %s%n", currentPackages);
            return;
        }

        Set<Requires> requires = desc.requires();
        Set<Exports> exports = desc.exports();
        Set<String> uses = desc.uses();
        Set<Provides> provides = desc.provides();
        Set<String> packages = desc.packages();

        System.out.printf("Requires: %s%n", requires);
        System.out.printf("Exports: %s%n", exports);
        System.out.printf("Uses: %s%n", uses);
        System.out.printf("Provides: %s%n", provides);
        System.out.printf("Пакеты: %s%n", packages);
    }
}
```

Выполним класс `ModuleBasicInfo` в режиме модуля и по старинке. Следующая команда выполняет его в режиме модуля:

```
C:\Java9Revealed>java --module-path com.jdojo.module.api\dist;com.jdojo.prime\dist;com.
jdojo.intro\dist
--module com.jdojo.module.api/com.jdojo.module.api.ModuleBasicInfo
```

```
Имя модуля: com.jdojo.module.api
Именованный модуль: true
Requires: [mandated java.base (@9-ea), com.jdojo.intro, java.sql (@9-ea), com.jdojo.prime]
Exports: [com.jdojo.module.api]
Uses: []
Provides: []
Пакеты: [com.jdojo.module.api]
-----------------
Имя модуля: com.jdojo.prime
Именованный модуль: true
Requires: [mandated java.base (@9-ea)]
Exports: [com.jdojo.prime]
Uses: [com.jdojo.prime.PrimeChecker]
Provides: []
Пакеты: [com.jdojo.prime]
-----------------
Имя модуля: java.sql
Именованный модуль: true
Requires: [transitive java.logging, transitive java.xml, mandated java.base]
Exports: [javax.transaction.xa, java.sql, javax.sql]
Uses: [java.sql.Driver]
Provides: []
Packages: [javax.sql, java.sql, javax.transaction.xa]
```

Теперь выполним класс `ModuleBasicInfo` в унаследованном режиме, воспользовавшись путем к классам:

```
C:\Java9Revealed>java -cp com.jdojo.module.api\dist\com.jdojo.module.api.jar;com.jdojo
.prime\dist\com.jdojo.prime.jar com.jdojo.module.api.ModuleBasicInfo
```

```
Имя модуля: null
Именованный модуль: false
Пакеты: [com.jdojo.module.api]
-----------------
Имя модуля: null
Именованный модуль: false
Пакеты: [com.jdojo.module.api, com.jdojo.prime]
-----------------
Имя модуля: java.sql
Именованный модуль: true
Requires: [mandated java.base, transitive java.logging, transitive java.xml]
```

```
Exports: [javax.transaction.xa, javax.sql, java.sql]
Uses: [java.sql.Driver]
Provides: []
Пакеты: [java.sql, javax.transaction.xa, javax.sql]
```

На этот раз классы `ModuleBasicInfo` и `PrimeChecker` загружены в безымянный модуль системного загрузчика классов, это видно из того, что метод `isNamed()` возвращает в обоих случаях `false`. Обратите внимание на динамическую природу метода `getPackages()` класса `Module`. При первом вызове он вернул только одно имя пакета – `com.jdojo.module.api`, а при втором уже два имени – `com.jdojo.module.api` и `com.jdojo.prime`. Это объясняется тем, что в безымянный модуль были загружены новые пакеты в процессе загрузки типов. Для модуля `java.sql` вывод в обоих случаях одинаков, поскольку платформенные типы всегда загружаются в один и тот же модуль независимо от того, как была запущена программа `java`.

Запросы к модулям

Вот несколько типичных вопросов, касающихся модулей:

- может ли модуль `M` читать модуль `N`?
- может ли модуль использовать службу определенного типа?
- экспортирует ли модуль определенный пакет всем или некоторым модулям?
- раскрывает ли модуль определенный пакет всем или некоторым модулям?
- является ли данный модуль именованным или безымянным?
- является ли данный модуль автоматическим?
- является ли данный модуль раскрытым?

Дескриптор модуля можно пополнить из командной строки или из программы, пользуясь API модулей. Все запросы, касающиеся свойств модулей, можно отнести к одной из двух категорий: для одних ответ может измениться после загрузки модуля, для других не может. Класс `Module` предоставляет методы для запросов из первой категории, а класс `ModuleDescriptor` – из второй. Вот список интересующих нас методов класса `Module`:

- `boolean canRead(Module other)`
- `boolean canUse(Class<?> service)`
- `boolean isExported(String packageName)`
- `boolean isExported(String packageName, Module other)`
- `boolean isOpen(String packageName)`
- `boolean isOpen(String packageName, Module other)`
- `boolean isNamed()`

Имена методов говорят сами за себя. Метод `isNamed()` возвращает `true`, если модуль именованный, и `false` – если безымянный. Правда, свойство именованный-безымянный не изменяется после загрузки модуля, но поскольку получить объект `ModuleDescriptor` для безымянного модуля невозможно, этот метод включен в класс `Module`.

Класс `ModuleDescriptor` содержит три метода, сообщающие о типе модуля и о том, как был сгенерирован его дескриптор. Метод `isOpen()` возвращает `true`, если модуль раскрыт, и `false` в противном случае. Метод `isAutomatic()`возвращает `true`, если модуль автоматический, иначе `false`. В листинге 10.3 приведен код класса `QueryModule`, входящего в модуль `com.jdojo.module.api`. В нем показано, как запросить у модуля информацию о зависимостях и как узнать, экспортирован ли (раскрыт ли) пакет всем модулям или только одному.

Листинг 10.3. Класс QueryModule, демонстрирующий запросы к модулю на этапе выполнения

```
// QueryModule.java
package com.jdojo.module.api;

import java.sql.Driver;

public class QueryModule {
   public static void main(String[] args) throws Exception {
      Class<QueryModule> cls = QueryModule.class;
      Module m = cls.getModule();

      // Проверить, может ли этот модуль читать модуль java.sql
      Module javaSqlModule = Driver.class.getModule();
      boolean canReadJavaSql = m.canRead(javaSqlModule);

      // Проверить, экспортирует ли этот модуль пакет com.jdojo.module.api всем
      // модулям
      boolean exportsModuleApiPkg = m.isExported("com.jdojo.module.api");

      // Проверить, экспортирует ли этот модуль пакет com.jdojo.module.api модулю
      // java.sql
      boolean exportsModuleApiPkgToJavaSql =
              m.isExported("com.jdojo.module.api", javaSqlModule);

      // Проверить, раскрывает ли этот модуль пакет com.jdojo.module.api модулю
      // java.sql
      boolean openModuleApiPkgToJavaSql = m.isOpen("com.jdojo.module.api",
                                                   javaSqlModule);

      // Напечатать тип и имя модуля
      System.out.printf("Именованный модуль: %b%n", m.isNamed());
      System.out.printf("Имя модуля: %s%n", m.getName());
      System.out.printf("Может читать java.sql? %b%n", canReadJavaSql);
      System.out.printf("Экспортирует com.jdojo.module.api? %b%n",
                        exportsModuleApiPkg);
      System.out.printf("Экспортирует com.jdojo.module.api модулю java.sql? %b%n",
                        exportsModuleApiPkgToJavaSql);
```

```
        System.out.printf("Раскрывает com.jdojo.module.api модулю java.sql? %b%n",
                          openModuleApiPkgToJavaSql);
    }
}
```

```
Именованный модуль: true
Имя модуля: com.jdojo.module.api
Может читать java.sql? true
Экспортирует com.jdojo.module.api? true
Экспортирует com.jdojo.module.api модулю java.sql? true
Раскрывает com.jdojo.module.api модулю java.sql? false
```

Модификация модулей

В предыдущих главах мы видели, как заставить модуль экспортировать пакеты или читать другие модули с помощью параметров командной строки `--add-exports`, `--add-opens` и `--add-reads`. В этом разделе мы узнаем, как то же самое можно сделать из программы. Класс `Module` содержит следующие методы, позволяющие изменять его объявление во время выполнения:

- `Module addExports(String packageName, Module other)`
- `Module addOpens(String packageName, Module other)`
- `Module addReads(Module other)`
- `Module addUses(Class<?> serviceType)`

Между параметром командной строки и соответствующим методом есть одно важное различие. Из командной строки можно модифицировать объявление любого метода, тогда как методы зависят от того, кто их вызывает. Код, из которого вызван метод, должен находиться в том модуле, объявление которого модифицируется, – за исключением метода `addOpens()`. Таким образом, если у вас нет доступа к исходному коду модуля, то вы не сможете воспользоваться этими методами для модификации его объявления. Эти методы предназначены для использования в каркасах, которые могут адаптироваться к потребностям среды выполнения для взаимодействия с другими модулями.

Все эти методы возбуждают исключение `IllegalCallerException`, когда вызывающей стороне не разрешено их использовать.

Метод `addExports()` изменяет модуль так, чтобы он экспортировал указанный пакет указанному модулю. Он ничего не делает, если указанный пакет уже экспортируется или раскрыт указанному модулю или если вызван для безымянного или раскрытого модуля. Если указанный пакет равен `null` или отсутствует в модуле, то возбуждается исключение `IllegalArgumentException`. Результат этого метода такой же, как при добавлении квалифицированного экспорта в объявление модуля:

```
exports <packageName> to <other>;
```

Метод `addOpens()` работает так же, как `addExports()`, только указанный пакет не экспортируется, а раскрывается указанному модулю. Его действие эквивалентно добавлению следующего предложения в объявление модуля:

```
opens <packageName> to <other>;
```

Метод `addOpens()` является исключением из правила, касающегося вызывающей стороны. Все остальные методы должны вызываться только из модифицируемого модуля, а `addOpens()` — из других тоже. Допустим, что модуль `M` раскрывает пакет `P` модулю `N` с помощью такого объявления:

```
module M {
    opens P to N;
}
```

В таком случае модулю `N` разрешено вызывать метод `addOpens("P", S)` для модуля `M`, заставив его раскрыть пакет `P` модулю `S`. Так делается, когда автор модуля раскрывает пакет известному абстрактному модулю каркаса, а тот обнаруживает и использует конкретную реализацию модуля во время выполнения. Динамически обнаруживаемый модуль может нуждаться в глубоком рефлексивном доступе к объявляемому. В этой ситуации автор модуля должен знать только имя абстрактного каркасного модуля и раскрыть пакет ему. А абстрактный модуль раскроет этот пакет динамически обнаруженному модулю реализации. Возьмем, к примеру, абстрактный каркас JPA, в котором определен модуль `java.persistence` и который на этапе выполнения обнаруживает другие реализации JPA, скажем, Hibernate или EclipseLink. В таком случае автор модуля может раскрыть пакет только модулю `java.persistence`, а тот раскроет его модулям Hibernate или EclipseLink во время выполнения.

Метод `addReads()` добавляет ребро чтения между данным модулем и указанным. Он ничего не делает, если указан модуль, от имени которого он вызван, потому что любой модуль может читать самого себя, или если вызван для безымянного модуля, потому что безымянный модуль может читать все остальные. Вызов этого метода эквивалентен добавлению следующего предложения `requires` в объявление модуля:

```
requires <other>;
```

Метод `addUses()` добавляет зависимость от службы, так что модуль получает возможность использовать класс `ServiceLoader` для загрузки службы указанного типа. Он ничего не делает при вызове для безымянного или автоматического модуля. Его действие эквивалентно добавлению следующего предложения `uses` в объявление модуля:

```
uses <serviceType>;
```

В листинге 10.4 приведен код класса `UpdateModule`, принадлежащего модулю `com.jdojo.module.api` (см. листинг 10.1). Отметим, что в объявлении модуля нет предложения `uses`. В классе имеется метод `findFirstService()`, который принимает в качестве аргумента тип службы. Он проверяет, может ли модуль загрузить службу такого типа. Напомним, что для загрузки службы некоторого типа с помощью класса `ServiceLoader` в объявлении метода должно присутствовать предложение `uses` с данным типом службы. Метод `findFirstService()` вызывает метод `addUses()` класса `Module`, чтобы добавить предложение `uses` с нужным типом службы, если оно отсутствует. В самом конце метод загружает и возвращает первого найденного поставщика службы.

Листинг 10.4. Класс UpdateModule, демонстрирующий добавление предложения uses в объявление модуля на этап выполнения

```
// UpdateModule.java
package com.jdojo.module.api;

import java.util.ServiceLoader;

public class UpdateModule {
   public static <T> T findFirstService(Class<T> service) {
      /* Прежде чем загружать поставщиков службы, проверим, может ли этот модуль
         использовать (или загружать) службу. Если нет, модифицируем модуль, чтобы
         он мог использовать службу.
      */
      Module m = UpdateModule.class.getModule();
      if (!m.canUse(service)) {
         m.addUses(service);
      }

      return ServiceLoader.load(service)
                          .findFirst()
                          .orElseThrow(
      () -> new RuntimeException("No service provider found for the service: " +
                                 service.getName()));
   }
}
```

Теперь протестируем метод `findFirstService()` класса `UpdateModule`. В листинге 10.5 приведено объявление модуля `com.jdojo.module.api.test`, где объявляется зависимость от модуля `com.jdojo.prime`, чтобы можно было использовать интерфейс службы `PrimeChecker`. Объявляется также зависимость от модуля `com.jdojo.module.api`, чтобы можно было использовать класс `UpdateModule` для загрузки службы. Оба эти модуля необходимо поместить на путь к модулям в проекте NetBeans `com.jdojo.module.api.test`.

Листинг 10.5. Объявление модуля com.jdojo.module.api.test

```
// module-info.java
module com.jdojo.module.api.test {
    requires com.jdojo.prime;
    requires com.jdojo.module.api;
}
```

В листинге 10.6 приведен код класса `Main` в модуле `com.jdojo.module.api.test`.

Листинг 10.6. Класс Main в модуле com.jdojo.module.api.test

```
// Main.java
```

```
package com.jdojo.module.api.test;

import com.jdojo.module.api.UpdateModule;
import com.jdojo.prime.PrimeChecker;

public class Main {
   public static void main(String[] args) {
      long[] numbers = {3, 10};

      try {
         // Получить поставщика службы com.jdojo.prime.PrimeChecker
         PrimeChecker pc = UpdateModule.findFirstService(PrimeChecker.class);

         // Проверить несколько чисел на простоту
         for (long n : numbers) {
            boolean isPrime = pc.isPrime(n);
            System.out.printf("%d простое: %b%n", n, isPrime);
         }
      } catch (RuntimeException e) {
         System.out.println(e.getMessage());
      }
   }
}
```

Попробуйте выполнить класс `Main`, как показано ниже. Не забудьте поместить модуль `com.jdojo.intro` на путь к модулям, потому что `com.jdojo.module.api.test` читает модуль `com.jdojo.module.api`, а тот читает модуль `com.jdojo.intro`.

```
C:\Java9Revealed>java --module-path com.jdojo.prime\dist;com.jdojo.intro\dist;com.jdojo
 .module.api\dist;com.jdojo.module.api.test\dist
 --module com.jdojo.module.api.test/com.jdojo.module.api.test.Main
```

```
No service provider found for the service: com.jdojo.prime.PrimeChecker
```

Как видим, программа выполнилась нормально. Она не нашла поставщика службы `com.jdojo.prime.PrimeChecker` на пути к модулям, о чем и сообщила. Добавим поставщика, модуль `com.jdojo.prime.generic`, в путь к модулям и выполним программу еще раз. Если вы поместите на путь к модулям другого поставщика, то результат может быть иным.

```
C:\Java9Revealed>java --module-path com.jdojo.prime\dist;com.jdojo.intro\dist;com.jdojo
 .module.api\dist;com.jdojo.module.api.test\dist;com.jdojo.prime.generic\dist
 --module com.jdojo.module.api.test/com.jdojo.module.api.test.Main
```

```
3 простое: true
10 простое: false
```

Доступ к ресурсам модуля

Модуль может содержать ресурсы: изображения, аудио- или видеоклипы, файлы свойств и политик. Файлы классов (с расширением `.class`) тоже считаются ресурсами. В классе `Module` имеется метод `getResourceAsStream()`, который возвращает ресурс по имени:

```
InputStream getResourceAsStream(String name) throws IOException
```

Доступ к ресурсам модуля подробно описывается в главе 8.

Аннотации модулей

Объявления модулей можно снабжать аннотациями. В перечислении `java.lang.annotation.ElementType` появился новый элемент `MODULE`. Если целевым типом в объявлении аннотации является `MODULE`, то аннотацию разрешено применять к модулям. В Java 9 две аннотации – `java.lang.Deprecated` и `java.lang.SuppressWarnings` – модифицированы и теперь могут применяться к объявлениям модулей тоже, например:

```
@Deprecated(since="1.2", forRemoval=true)
@SuppressWarnings("unchecked")
module com.jdojo.myModule {
    // Предложения модуля
}
```

Если модуль объявлен нерекомендуемым (deprecated), то всякий раз как он встречается в предложении `requires` (но не `exports` или `opens`), выдается предупреждение. Объясняется это тем, что пользователям нерекомендуемого модуля надо дать знать о том, что вскоре он может быть исключен. Что же касается предложений `exports` и `opens`, то они относятся к самому нерекомендуемому модулю, так что предупреждать некого. При использовании типов, содержащихся в нерекомендуемом модуле, никакие предупреждения не выдаются. Аналогично, если в объявлении модуля подавляется некоторое предупреждение, то это относится к элементам самого объявления, а не к типам, содержащимся внутри модуля.

Класс `Module` реализует интерфейс `java.lang.reflect.AnnotatedElement`, поэтому для чтения аннотаций доступны все предназначенные для этой цели методы. Тип аннотации, применимой к модулям, должен содержать в качестве одной из целей константу `ElementType.MODULE`.

> **Совет.** Нельзя аннотировать отдельные предложения модуля. Например, нельзя снабдить предложение `exports` аннотацией `@Deprecated`, указав тем самым, что экспортируемый пакет будет удален в будущей версии. Эта идея рассматривалась на ранней стадии проектирования и была отвергнута на том основании, что ее реализация займет много времени, а полезность неочевидна. Такая возможность может появиться в будущем, если возникнет потребность. Ну а пока в классе `ModuleDescriptor` вы не найдете никаких методов, связанных с аннотациями.

Теперь давайте создадим новый тип аннотации и применим его к объявлению модуля. В листинге 10.7 приведено объявление модуля `com.jdojo.module.api.annotation`, снабженное тремя аннотациями. Тип аннотации `Version` объявлен в том же

модуле, а его исходный код приведен в листинге 10.8. Для нового типа задана политика сохранения RUNTIME.

Листинг 10.7. Объявление модуля com.jdojo.module.api.annotation

```
// module-info.java

import com.jdojo.module.api.annotation.Version;

@Deprecated(since="1.2", forRemoval=false)
@SuppressWarnings("unchecked")
@Version(major=1, minor=2)
module com.jdojo.module.api.annotation {
    // Предложений модуля нет
}
```

Листинг 10.8. Тип аннотации Version, применимый к пакетам, модулям и типам

```
// Version.java
package com.jdojo.module.api.annotation;

import static java.lang.annotation.ElementType.MODULE;
import static java.lang.annotation.ElementType.PACKAGE;
import static java.lang.annotation.ElementType.TYPE;
import java.lang.annotation.Retention;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Target;

@Retention(RUNTIME)
@Target({PACKAGE, MODULE, TYPE})
public @interface Version {
   int major();
   int minor();
}
```

В листинге 10.9 приведен код класса AnnotationTest. Он читает аннотации модуля com.jdojo.module.api.annotation. В выводе нет аннотации @SuppressWarnings, которой снабжен модуль, потому что для нее указана политика сохранения RetentionPolicy.RUNTIME, т. е. на этапе выполнения аннотация не сохраняется.

Листинг 10.9. Класс AnnotationTest, демонстрирующий, как читать аннотации модуля

```
// AnnotationTest.java
package com.jdojo.module.api.annotation;

import java.lang.annotation.Annotation;

public class AnnotationTest {
```

```
   public static void main(String[] args) {
      // Получить ссылку на модуль com.jdojo.module.api.annotation
      Module m = AnnotationTest.class.getModule();

      // Напечатать все аннотации
      Annotation[] a = m.getAnnotations();
      for(Annotation ann : a) {
         System.out.println(ann);
      }

      // Прочитать аннотацию Deprecated
      Deprecated d = m.getAnnotation(Deprecated.class);
      if (d != null) {
         System.out.printf("Deprecated: since=%s, forRemoval=%b%n",
                           d.since(), d.forRemoval());
      }

      // Прочитать аннотацию Version
      Version v = m.getAnnotation(Version.class);
      if (v != null) {
         System.out.printf("Version: major=%d, minor=%d%n", v.major(), v.minor());
      }
   }
}
```

```
@java.lang.Deprecated(forRemoval=false, since="1.2")
@com.jdojo.module.api.annotation.Version(major=1, minor=2)
Deprecated: since=1.2, forRemoval=false
Version: major=1, minor=2
```

Загрузка классов

Для загрузки и инициализации класса можно использовать один из статических методов `forName()` класса `Class`:

- `Class<?> forName(String className) throws ClassNotFoundException`
- `Class<?> forName(String className, boolean initialize, ClassLoader loader) throws ClassNotFoundException`
- `Class<?> forName(Module module, String className)`

В этих методах параметр `className` – полное имя загружаемого класса или интерфейса, например, `java.lang.Thread` или `com.jdojo.intro.Welcome`. Если параметр `initialize` равен `true`, то класс инициализируется.

Метод `forName(String className)` инициализирует класс после загрузки и использует текущий загрузчик классов, т. е. тот, который загрузил класс, в котором вызван этот метод. Выражение

```
Class.forName("P.Q")
```

эквивалентно следующему

```
Class.forName("P.Q", true, this.getClass().getClassLoader())
```

Метод `forName(Module module, String className)` добавлен в JDK 9. Он не инициализирует класс и не возбуждает исключение `ClassNotFoundException`, если класс не найден, а возвращает `null`.

Ни один из этих методов не проверяет, разрешен ли вызывающей стороне доступ к указанному классу. Таким образом, с помощью этих методов можно загрузить не экспортированные из модуля классы. Однако это не означает, что можно создать экземпляр такого класса и обратиться к его членам. Проверка доступности производится при попытке создать экземпляр класса, и вот тогда может возникнуть исключение.

В листинге 10.10 приведен код класса `LoadingClass`, принадлежащего модулю `com.jdojo.module.api`. Он содержит два варианта метода `loadClass()`. Оба они пытаются загрузить указанный класс и в случае успеха создать его экземпляр, вызвав конструктор без аргументов. Отметим, что модуль `com.jdojo.intro` не экспортирует пакет `com.jdojo.intro`, содержащий класс `Welcome`. В этом примере мы пытаемся загрузить и инстанцировать класс `Welcome` и два несуществующих класса.

Листинг 10.10. Класс, демонстрирующий загрузку классов

```
// LoadingClass.java
package com.jdojo.module.api;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.util.Optional;

public class LoadingClass {
   public static void main(String[] args) {
      loadClass("com.jdojo.intro.Welcome");
      loadClass("com.jdojo.intro.XYZ");

      String moduleName = "com.jdojo.intro";
      Optional<Module> m = ModuleLayer.boot().findModule(moduleName);
      if (m.isPresent()) {
         Module introModule = m.get();
         loadClass(introModule, "com.jdojo.intro.Welcome");
         loadClass(introModule, "com.jdojo.intro.ABC");
      } else {
         System.out.println("Модуль не найден: " + moduleName +
         ". Поместите модуль на путь к модулям.");
      }
   }

   public static void loadClass(String className) {
```

```
      try {
         Class<?> cls = Class.forName(className);
         System.out.println("Класс найден: " + cls.getName());
         instantiateClass(cls);
      } catch (ClassNotFoundException e) {
         System.out.println("Класс не найден: " + className);
      }
   }

   public static void loadClass(Module m, String className) {
      Class<?> cls = Class.forName(m, className);
      if (cls == null) {
         System.out.println("Класс не найден: " + className);
      } else {
         System.out.println("Класс найден: " + cls.getName());
         instantiateClass(cls);
      }
   }

   public static void instantiateClass(Class<?> cls) {
      try {
         // Получить конструктор без аргументов
         Constructor<?> c = cls.getConstructor();
         Object o = c.newInstance();
         System.out.println("Создан экземпляр класса: " + cls.getName());
      } catch (InstantiationException | IllegalAccessException |
               IllegalArgumentException | InvocationTargetException e) {
         System.out.println(e.getMessage());
      } catch (NoSuchMethodException e) {
         System.out.println("Нет конструктора без аргументов в классе: " +
                            cls.getName());
      }
   }
}
```

Попробуем выполнить класс `LoadingClass`, поместив три необходимых ему модуля на путь к классам:

```
C:\Java9Revealed>java
 --module-path com.jdojo.module.api\dist;com.jdojo.prime\dist;com.jdojo.intro\dist
 --module com.jdojo.module.api/com.jdojo.module.api.LoadingClass
```

```
Класс найден: com.jdojo.intro.Welcome
class com.jdojo.module.api.LoadingClass (in module com.jdojo.module.api) cannot
access class com.jdojo.intro.Welcome (in module com.jdojo.intro) because module com.
jdojo.intro does not export com.jdojo.intro to module com.jdojo.module.api
Класс не найден: com.jdojo.intro.XYZ
```

```
Класс найден: com.jdojo.intro.Welcome
class com.jdojo.module.api.LoadingClass (in module com.jdojo.module.api) cannot
access class com.jdojo.intro.Welcome (in module com.jdojo.intro) because module com.
jdojo.intro does not export com.jdojo.intro to module com.jdojo.module.api
Класс не найден: com.jdojo.intro.ABC
```

Как видим, класс `com.jdojo.intro.Welcome` загрузить удалось. Однако мы не можем создать его экземпляр, потому что модуль `com.jdojo.intro` его не экспортирует. В следующей команде добавлен флаг `--add-exports` для экспорта пакета `com.jdojo.intro` модулю `com.jdojo.module.api`. Теперь мы можем и загрузить, и инстанцировать класс `Welcome`.

```
c:\Java9Revealed>java
 --module-path com.jdojo.module.api\dist;com.jdojo.prime\dist;com.jdojo.intro\dist
 --add-exports com.jdojo.intro/com.jdojo.intro=com.jdojo.module.api
 --module com.jdojo.module.api/com.jdojo.module.api.LoadingClass
```

```
Класс найден: com.jdojo.intro.Welcome
Создан экземпляр класса: com.jdojo.intro.Welcome
Класс не найден: com.jdojo.intro.XYZ
Класс найден: com.jdojo.intro.Welcome
Создан экземпляр класса: com.jdojo.intro.Welcome
Класс не найден: com.jdojo.intro.ABC
```

Слои модулей

Работа со слоями модулей считается продвинутой темой. Обычному Java-разработчику не приходится иметь дело со слоями модулей напрямую. В существующих приложениях слои модулей не используются. При переносе приложения на JDK 9 или разработке нового приложения на этой платформе хочешь не хочешь придется использовать по меньшей мере один слой, который JVM создает на этапе инициализации. Обычно слоями пользуются приложения с архитектурой плагинов или контейнеров. В этом разделе я приведу общие сведения о слоях модулей на простом примере. Термины «слой» и «слой модулей» употребляются как синонимы.

Слоем называется множество разрешенных модулей (граф модулей) вместе с функцией, которая отображает каждый модуль на загрузчик классов, отвечающий за загрузку всех типов из этого модуля. Множество разрешенных модулей называется *конфигурацией*. Связь между модулями, загрузчиками классов, конфигурациями и слоями можно схематически представить в виде:

```
Конфигурация = Граф модулей
Слой модулей = Конфигурация + (Модуль -> Загрузчик классов)
```

Модули собраны в слои. Слои образуют иерархию. У каждого слоя имеется хотя бы один родительский слой; исключение составляет *пустой* слой, который не со-

держит никаких модулей, его основное назначение — быть родителем *начального* слоя. Начальный слой создается JVM на этапе инициализации путем разрешения корневых модулей приложения относительно множества видимых модулей. Принцип загрузки типов с помощью загрузчиков классов в JDK 9 не изменился. Обычно загрузчики применяют модель делегирования «сначала родитель», т. е. запрос на загрузку типа передается родителю, который передает его своему родителю – и так до базового загрузчика. Если ни один из родителей не загрузил тип, то это делает загрузчик, получивший исходный запрос. На рис. 10.1 приведен пример организации модулей, загрузчиков классов и слоев.

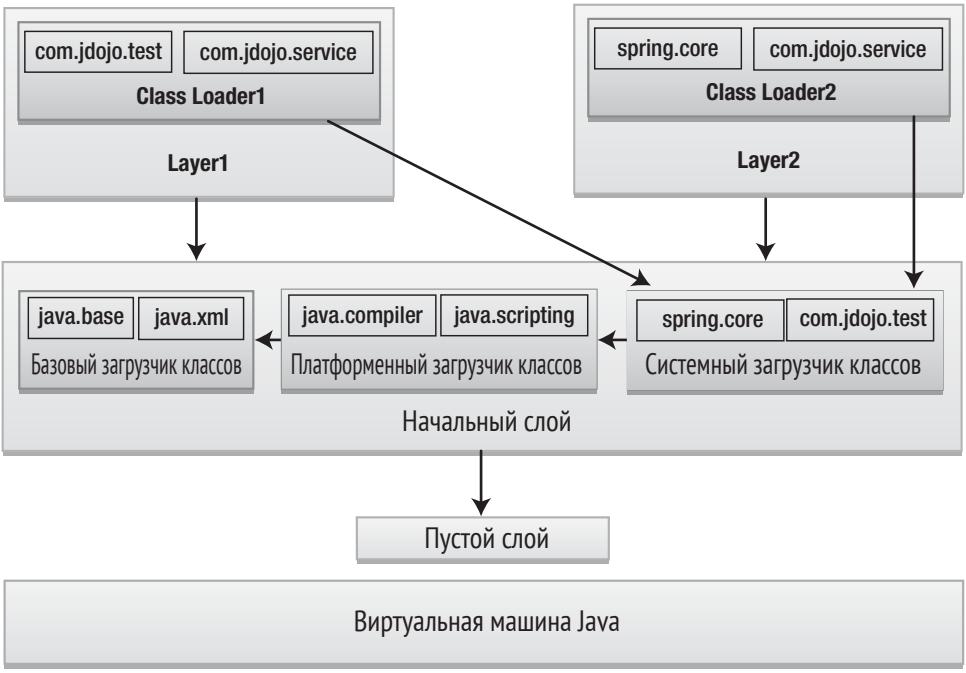

Рис. 10.1. Пример организации слоев модулей в приложении

Стрелка, направленная вниз от X к Y, означает, что X является родителем Y, где X и Y могут быть загрузчиками классов или слоями. Слои расположены друг над другом, и в самом низу находятся пустой и начальный слой. Далее я забуду про пустой слой и буду считать нижним начальный слой. Начальный слой является родителем двух пользовательских слоев, Layer1 и Layer2.

Модули, принадлежащие данному слою, могут читать модули из слоев, расположенных ниже. Следовательно, слои Layer1 и Layer2 могут читать модули в начальном слое. Однако Layer1 не может читать модули в слое Layer2, поскольку эти слои расположены на одном уровне. Точно так же начальный слой не может читать модули в слоях Layer1 и Layer2, потому что является для них родителем. Как видно из рис. 10.1, родителем загрузчиков классов в обоих пользовательских слоях является системный загрузчик, именно так чаще всего и бывает. Сделав системный загрузчик родителем пользовательского, мы гарантируем, что тот сможет прочитать все типы, находящиеся в модулях начального слоя. Возможность чтения сохраняется, когда модули одного слоя читают модули нижерасположенного слоя.

Организация слоев модулей полезна в двух ситуациях, часто встречающихся в сложных Java-приложениях типа серверов приложений и веб-серверов в среде Java EE, которые служат контейнерами размещенных приложений, а именно: механизм переопределения и механизм расширения. В случае механизма переопределения размещенное приложение должно заместить функциональность, предоставляемую контейнером, например, использовать другую версию того же модуля. В случае механизма расширения размещенное приложение должно дополнить функциональность, уже предоставляемую контейнером, например, предоставить дополнительных поставщиков служб. На рис. 10.1 модуль `com.jdojo.test` находится как в начальном слое, так и в слое Layer1. Это случая переопределения модуля. Слой Layer1 будет пользоваться версией из Layer1, а слой Layer2 – версией из начального слоя.

Часто контейнер должен позволять размещенным приложениям предоставлять собственный набор модулей, возможно, переопределяющих встроенные в контейнер модули. Для этого модули размещенных приложений должны загружаться в слой над слоем контейнера. Тогда модули, загруженные в прикладные слои, будут замещать модули в серверных слоях, и, следовательно, мы сможем использовать несколько версий одного модуля в контексте одной и той же JVM.

Размещенное приложение может захотеть использовать не того поставщика службы, которого предлагает контейнер. Чтобы удовлетворить это желание, можно поместить модули, содержащие нужного приложению поставщика, в слой, расположенный выше контейнерного слоя. Для загрузки поставщиков служб предназначен метод `load(ModuleLayer layer, Class<S> service)` класса `ServiceLoader`. При этом параметр `layer` должен быть слоем размещенного приложения. Метод загружает поставщиков службы из указанного слоя и его родителей.

> **Совет.** Слои неизменяемы. После того как слой создан, модули нельзя ни добавить, ни удалить. Чтобы добавить новые модули или заменить версию модуля, необходимо изъять слой из иерархии и заново создать его.

Процесс создания слоя состоит из нескольких шагов:

- создать локаторы модулей;
- создать набор корневых модулей;
- создать объект конфигурации;
- создать слой.

Созданный слой можно использовать для загрузки типов. В следующих разделах все эти шаги описаны подробно. А в конце я покажу, как с помощью слоев можно использовать несколько версий модуля.

Поиск модулей

Локатором модулей называется экземпляр интерфейса `ModuleFinder`. Он служит для поиска объектов `ModuleReference` в процессе разрешения модулей и привязки служб. Интерфейс содержит два фабричных метода для создания локаторов:

- `static ModuleFinder of(Path... entries)`
- `static ModuleFinder ofSystem()`

Метод `of()` ищет модули, просматривая заданную упорядоченную последовательность путей к каталогам или упакованным модулям. Метод возвращает первый встретившийся модуль с запрошенным именем. Ниже показано, как создать локатор, который будет искать модули в каталогах `C:\Java9Revealed\lib` и `C:\Java9Revealed\customLib`:

```
// Создать пути  к модулям
Path mp1 = Paths.get("C:\\Java9Revealed\\lib");
Path mp2 = Paths.get("C:\\Java9Revealed\\customLib");

// Создать локатор модулей на двух путях
ModuleFinder finder = ModuleFinder.of(mp1, mp2);
```

Иногда требуется ссылка на объект `ModuleFinder`, например, чтобы передать ее какому-то методу, но искать при этом ничего не надо. Такую ссылку можно создать, вызвав метод `ModuleFinder.of()` без параметров.

Метод `ofSystem()` возвращает локатор, который ищет системные модули, связанные со средой выполнения. Он всегда находит модуль `java.base`. Отметим, что с образом среды выполнения можно связать пользовательский набор модулей, поэтому какие именно модули найдет этот метод, зависит от образа. Пользовательский образ среды выполнения содержит как модули JDK, так и модули приложения. Этот метод находит те и другие.

Локатор можно также составить из последовательности нуля или более других локаторов с помощью метода `compose()`:

```
static ModuleFinder compose(ModuleFinder... finders)
```

Этот локатор будет вызывать каждый из составляющих его локаторов по очереди. Второй локатор будет искать все модули, не найденные первым, третий – модули, не найденные вторым, и т. д.

В интерфейсе `ModuleFinder` определены следующие методы поиска модулей:

- `Optional<ModuleReference> find(String name)`
- `Set<ModuleReference> findAll()`

Метод `find()` находит модуль с указанным именем, а метод `findAll()` – все модули, которые может найти локатор.

В листинге 10.11 приведен код класса `FindingModule`, показывающий, как использовать `ModuleFinder`. Я использовал пути в Windows вида `C:\Java9Revealed\lib` к каталогам, где на моей машине хранятся модули, но вы можете подставить свои собственные. Класс является частью модуля `com.jdojo.module.api`.

Листинг 10.11. Использование класса ModuleFinder для поиска модулей

```
// FindingModule.java
package com.jdojo.module.api;

import java.lang.module.ModuleDescriptor;
import java.lang.module.ModuleFinder;
import java.lang.module.ModuleReference;
```

```
import java.net.URI;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Optional;
import java.util.Set;

public class FindingModule {
   public static void main(String[] args) {
      // Создать пути к модулям
      Path mp1 = Paths.get("C:\\Java9Revealed\\lib");
      Path mp2 = Paths.get("C:\\Java9Revealed\\customLib");

      // Создать локатор модулей
      ModuleFinder finder = ModuleFinder.of(mp1, mp2);

      // Найти все модули, которые может найти локатор
      Set<ModuleReference> moduleRefs = finder.findAll();

      // Напечатать сведения о найденных модулях
      moduleRefs.forEach(FindingModule::printInfo);
   }

   public static void printInfo(ModuleReference mr) {
      ModuleDescriptor md = mr.descriptor();
      Optional<URI> location = mr.location();
      URI uri = null;
      if(location.isPresent()) {
         uri = location.get();
      }
      System.out.printf("Модуль: %s, положение: %s%n", md.name(), uri);
   }
}
```

```
Модуль: com.jdojo.prime.probable, положение: file:///C:/Java9Revealed/lib/com.jdojo.prime.
probable.jar
Модуль: com.jdojo.person, положение: file:///C:/Java9Revealed/lib/com.jdojo.person.jar
Модуль: com.jdojo.address, положение: file:///C:/Java9Revealed/lib/com.jdojo.address.jar
...
```

Чтение содержимого модуля

В предыдущем разделе мы научились использовать объект `ModuleFinder` для поиска ссылок на модули – экземпляров класса `ModuleReference`, которые инкапсулируют объект `ModuleDescriptor` и местоположение модуля. Для получения экземпляра интерфейса `ModuleReader` предназначен метод `open()` класса `ModuleReference`. Класс `ModuleReader` используется для перечисления, поиска и чтения содержимого моду-

ля. В следующем фрагменте показано, как получить объект ModuleReader для модуля java.base:

```
// Создать локатор системных модулей
ModuleFinder finder = ModuleFinder.ofSystem();

// Модуль java.base гарантированно существует
Optional<ModuleReference> omr = finder.find("java.base");
ModuleReference moduleRef = omr.get();

// Получить читатель модуля
ModuleReader reader = moduleRef.open();
```

Метод open() класса ModuleReference может возбуждать исключение IOException. Для простоты я опустил обработку исключения.

Для работы с содержимым модуля используются следующие методы класса ModuleReader, имена которых говорят сами за себя:

- void close() throws IOException
- Optional<URI> find(String resourceName) throws IOException
- Stream<String> list() throws IOException
- default Optional<InputStream> open(String resourceName) throws IOException
- default Optional<ByteBuffer> read(String resourceName) throws IOException
- default void release(ByteBuffer bb)

Передаваемое этим методам имя ресурса представляет собой строку пути, компоненты которого разделены знаком /. Так, ресурс, соответствующий классу java.lang.Object в модуле java.base, называется java/lang/Object.class.

По завершении работы с объектом ModuleReader его необходимо закрыть его методом close(). При попытке прочитать содержимое модуля с помощью закрытого объекта ModuleReader будет возбуждено исключение IOException. Метод read() возвращает значение типа Optional<ByteBuffer>. Закончив работу с буфером байтов, вызовите его метод release(ByteBuffer bb), чтобы предотвратить утечку памяти.

В листинге 10.12 приведена программа чтения содержимого модуля. Она читает содержимое класса Object в объект ByteBuffer и печатает его размер в байтах. Кроме того, печатаются имена пяти ресурсов в модуле java.base. На вашей машине результат может быть другим.

Листинг 10.12. Применение объекта ModuleReader для чтения содержимого модуля

```
// ReadingModuleContents.java
package com.jdojo.module.api;

import java.io.IOException;
import java.lang.module.ModuleFinder;
import java.lang.module.ModuleReader;
import java.lang.module.ModuleReference;
import java.nio.ByteBuffer;
```

```
import java.util.Optional;

public class ReadingModuleContents {
   public static void main(String[] args) {
      // Создать локатор системных модулей
      ModuleFinder finder = ModuleFinder.ofSystem();

      // Модуль java.base гарантированно существует
      Optional<ModuleReference> omr = finder.find("java.base");
      ModuleReference moduleRef = omr.get();

      // Получить и использовать читатель модуля
      try (ModuleReader reader = moduleRef.open()) {
         // Прочитать класс Object и напечатать его размер
         Optional<ByteBuffer> bb = reader.read("java/lang/Object.class");

         bb.ifPresent(buffer -> {
            System.out.println("Размер Object.class: " + buffer.limit());

            // Освободить буфер байтов
            reader.release(buffer);
         });
         System.out.println("\nПять ресурсов модуля java.base:");
         reader.list()
               .limit(5)
               .forEach(System.out::println);
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

```
Размер Object.class: 1859
Пять ресурсов модуля java.base:
module-info.class
sun/util/BuddhistCalendar.class
sun/util/PreHashedMap$1$1.class
sun/util/PreHashedMap$1.class
sun/util/PreHashedMap$2$1$1.class
```

Создание конфигураций

Конфигурация представляет набор разрешенных модулей. Разрешенным называется модуль, для которого вычислены зависимости, заданные в предложениях `requires`. В процессе разрешения используются два набора модулей: корневых и видимых. Каждый модуль из набора корневых считается начальным, и присутст-

вующие в его объявлении предложения `requires` разрешаются относительно набора видимых модулей. Корневой модуль может затребовать другой модуль, который, в свою очередь, затребует третий и т. д. В процессе разрешения вычисляется цепочка зависимостей для всех корневых модулей. Получающийся граф модулей называется графом *зависимостей*.

В графе зависимостей учитываются только предложения `requires`. Если в модуле присутствует предложение `requires transitive`, то модули, зависящие от данного, неявно зависят также от модулей, упомянутых в этом предложении. Граф зависимостей пополняется дополнительными ребрами чтения, обусловленными предложениями `requires transitive`, и в результате получается *граф чтения*.

Предложения `uses` и `provides` в объявлениях модулей также образуют зависимости. Если модуль `M` использует службу типа `S`, а модуль `N` предоставляет реализацию `S` в виде `T`, то `M` зависит от `N` в части использования службы `S`. Граф чтения пополняется модулями, вычисленными для таких зависимостей по службам.

Конфигурация, созданная для начального слоя, содержит модули, полученные в результате разрешения зависимостей (предложения `requires`), добавления неявных ребер чтения (`requires transitive`) и учета зависимостей по службам (предложения `uses` и `provides`). При создании конфигурации для пользовательского слоя вы вправе включать или не включать зависимости по службам.

Конфигурация представлена экземпляром класса `Configuration`. У каждой конфигурации, кроме пустой, есть по меньшей мере один родитель.

Разрешенный модуль в конфигурации представлен классом `ResolvedModule`. Его метод `reads()` возвращает множество `Set<ResolvedModule>` модулей, которые читает разрешенный модуль. Метод `configuration()` возвращает объект конфигурации `Configuration`, частью которой является разрешенный модуль. Метод `reference()` возвращает объект `ModuleReference`, от которого можно получить объект `ModuleReader` для чтения содержимого модуля.

Для создания объекта `Configuration` служат следующие методы класса `Configuration`:

- `static Configuration empty()`
- `Configuration resolve(ModuleFinder before, ModuleFinder after, Collection<String> roots)`
- `Configuration resolveAndBind(ModuleFinder before, ModuleFinder after, Collection<String> roots)`
- `static Configuration resolve(ModuleFinder before, List<Configuration> parents, ModuleFinder after, Collection<String> roots)`
- `static Configuration resolveAndBind(ModuleFinder before, List<Configuration> parents, ModuleFinder after, Collection<String> roots)`

Метод `empty()` возвращает пустой объект `Configuration`. Его основное назначение – служить родителем конфигурации начального слоя.

У методов `resolve()` и `resolveAndBind()` по два варианта: метод экземпляра и статический метод. Между ними есть только одно различие: метод экземпляра создает новую конфигурацию, используя текущую в качестве родительской, а статическому методу передается список родителей новой конфигурации.

Метод `resolve()` создает новую конфигурацию путем разрешения зависимостей, обусловленных предложениями `requires` и `requires transitive` в объявлениях модулей. В роли корневых выступают модули, заданные в аргументе `roots`.

В процессе разрешения модули сначала ищутся с помощью переданного локатора `before`. Если модуль не найден, то производится поиск в родительских конфигурациях. Если модуль и там не найден, то применяется переданный локатор `after`. Если ваше намерение состоит в том, чтобы переопределить модуль в родительской конфигурации, то следует поместить модуль на путь, просматриваемый локатором `before`.

Метод `resolveAndBind()` работает так же, как `resolve()`, но разрешает еще и зависимости по службам. В следующем фрагменте показано, как создать конфигурацию, используя конфигурацию начального слоя в качестве родительской:

```
// Определить локаторы модулей
String modulePath = "C:\\Java9Revealed\\customLib";
Path path = Paths.get(modulePath);

ModuleFinder beforFinder = ModuleFinder.of(path);

// Наш локатор after пустой
ModuleFinder afterFinder = ModuleFinder.of();

// Задать корневые модули
Set<String> rootModules = Set.of("com.jdojo.layer");

// Создать конфигурацию, используя конфигурацию начального слоя в качестве
// родительской
Configuration parentConfig = ModuleLayer.boot().configuration();
Configuration config = parentConfig.resolve(beforFinder, afterFinder, rootModules);
```

Для получения сведений о разрешенных модулях в конфигурации служат следующие методы класса `Configuration`:

- `Optional<ResolvedModule> findModule(String name)`
- `Set<ResolvedModule> modules()`
- `List<Configuration> parents()`

Имена и сигнатуры этих методов говорят сами за себя. В следующем разделе я покажу, как использовать объект `Configuration` для создания слоя модулей.

Создание слоя модулей

Слой модулей – это конфигурация плюс функция, ставящая в соответствие каждому модулю загрузчик классов. Для создания слоя сначала необходимо создать конфигурацию и иметь один или несколько загрузчиков классов. Загрузчик, соответствующий модулю, отвечает за загрузку всех типов в этом модуле. Мы можем отобразить все модули в конфигурации на один и тот же загрузчик или поставить в соответствие каждому модулю свой загрузчик. Возможно также определить пользовательскую стратегию отображения. Обычно загрузчики классов применяют стратегия делегирования запроса родительскому загрузчику. Мы также можем воспользоваться этой стратегией при определении соответствия между модулями слоя и загрузчиками классов.

Слой представляется экземпляром класса `ModuleLayer` из пакета `java.lang`. Метод `empty()` этого класса возвращает пустой слой с пустой конфигурацией, а метод `boot()` – начальный слой. Для создания пользовательского слоя используются следующие методы:

- `ModuleLayer defineModules(Configuration cf, Function<String,ClassLoader> clf)`
- `static ModuleLayer.Controller defineModules(Configuration cf, List<ModuleLayer> parentLayers, Function<String,ClassLoader> clf)`
- `ModuleLayer defineModulesWithManyLoaders(Configuration cf, ClassLoader parentClassLoader)`
- `static ModuleLayer.Controller defineModulesWithManyLoaders(Configuration cf, List<ModuleLayer> parentLayers, ClassLoader parentLoader)`
- `ModuleLayer defineModulesWithOneLoader(Configuration cf, ClassLoader parentClassLoader)`
- `static ModuleLayer.Controller defineModulesWithOneLoader(Configuration cf, List<ModuleLayer> parentLayers, ClassLoader parentLoader)`

У каждого из методов `defineModulesXxx()` по два варианта: метод экземпляра и статический метод. Методы экземпляра считают слой, от имени которого вызваны, родительским, а статические методы позволяют задать список родительских слоев. Статические методы возвращают объект `ModuleLayer.Controller`, который можно использовать для работы с модулями в новом слое. Этот вложенный класс, принадлежащий пакету `java.lang`, располагает следующими методами:

- `ModuleLayer.Controller addOpens(Module source, String packageName, Module target)`
- `ModuleLayer.Controller addReads(Module source, Module target)`
- `ModuleLayer layer()`

Метод `addOpens()` позволяет раскрыть принадлежащий этому модулю пакет для другого модуля, а метод `addReads()` добавляет ребро чтения между этим и другим модулем. Метод `layer()` возвращает объект `ModuleLayer`, которым управляет данный контроллер.

Метод `defineModules(Configuration cf, Function<String,ClassLoader> clf)` принимает в качестве первого аргумента конфигурацию, а в качестве второго – функцию отображения, которая принимает имя модуля в конфигурации и возвращает соответствующий ему загрузчик классов. Метод завершается с ошибкой в следующих случаях:

- несколько модулей, содержащих один и тот же пакет, отображаются на один и тот же загрузчик;
- модуль отображается на загрузчик, которому уже назначен модуль с таким же именем;
- модуль отображается на загрузчик, которому уже назначены типы, определенные в каком-либо пакете из этого модуля.

Метод `defineModulesWithManyLoaders(Configuration cf, ClassLoader parentClassLoader)` создает слой по заданной конфигурации. Каждому модулю в конфигурации сопоставляется отдельный загрузчик классов, создаваемый этим же методом. Заданный во втором аргументе загрузчик становится родительским для всех созданных загрузчиков. Обычно в роли родительского выступает системный загрузчик. Если

во втором аргументе передать `null`, то родителем всех созданных загрузчиков будет базовый загрузчик.

Метод `defineModulesWithOneLoader(Configuration cf, ClassLoader parentClassLoader)` создает слой по заданной конфигурации. Он создает единственный загрузчик классов, родителем которого является загрузчик, заданный во втором аргументе, и сопоставляет его всем модулям конфигурации. Если во втором аргументе передан `null`, то родителем созданного загрузчика будет базовый загрузчик.

В следующем фрагменте создается дочерний слой начального слоя. Все модули этого слоя будут загружаться одним и тем же загрузчиком, родителем которого является системный загрузчик.

```
Configuration config = /* создать конфигурацию... */
ClassLoader sysClassLoader = ClassLoader.getSystemClassLoader();
ModuleLayer parentLayer = ModuleLayer.boot();
ModuleLayer layer = parentLayer.defineModulesWithOneLoader(config, sysClassLoader);
```

После того как слой создан, нужно будет загружать классы из находящихся в нем модулей. Все принадлежащие модулю типы загружаются загрузчиком, сопоставленным этому модулю. Отметим, что один и тот же модуль может быть определен в нескольких слоях, но тогда этим модулям будут сопоставлены разные загрузчики. В классе `ModuleLayer` имеется метод `findLoader(String moduleName)`, который принимает имя модуля в качестве аргумента и возвращает сопоставленный ему объект `ClassLoader`. Если ни в данном слое, ни в его предках не существует модуля с таким именем, то возбуждается исключение `IllegalArgumentException`. Получив `ClassLoader`, мы можем вызвать его метод `loadClass(String className)` для загрузки класса из данного модуля. В следующем фрагменте кода показано, как загрузить класс из слоя (обработка исключений опущена):

```
ModuleLayer layer = /* create a layer... */

// Загрузить класс, зная слой
String moduleName = "com.jdojo.layer";
String className = "com.jdojo.layer.LayerInfo";
Class<?> cls = layer.findLoader(moduleName)
                    .loadClass(className);
```

Имея объект `Class`, мы можем использовать его для создания объектов и вызова их методов. В следующем фрагменте создается объект загруженного класса и вызывается метод `printInfo` этого объекта:

```
// Имя метода, печатающего сведения об объекте
String methodName = "printInfo";

// Создать экземпляр класса с помощью конструктора без аргументов
Object obj = cls.getConstructor().newInstance();

// Найти метод
Method method = cls.getMethod(methodName);

// Вызвать метод, печатающий сведения
method.invoke(obj);
```

Следующие методы класса `ModuleLayer` позволяют получить информацию о самом слое или содержащихся в нем модулях:

- `Optional<Module> findModule(String moduleName)`
- `Set<Module> modules()`
- `List<ModuleLayer> parents()`

Метод `findModule()` ищет модуль с заданным именем в данном слое или его родителях. Метод `modules()` возвращает множество модулей слоя, которое может быть и пустым, если слой не содержит ни одного модуля. Метод `parent()` возвращает список родителей данного слоя, который будет пустым для пустого слоя.

Далее мы рассмотрим полный пример, демонстрирующий, как создать пользовательский модуль и как загрузить две версии одного и того же модуля в два слоя одного приложения.

Модуль называется `com.jdojo.layer` и содержит единственный пакет `com.jdojo.layer`, в котором находится один класс `LayerInfo`. У нас будет две версии одного модуля, так что код почти полностью повторяется. Я создал в NetBeans два проекта с именами `com.jdojo.layer.v1` и `com.jdojo.layer.v2`.

В листингах 10.13 и 10.14 находятся соответственно объявления версии 1.0 модуля `com.jdojo.layer` и класса `LayerInfo`.

Листинг 10.13. Версия 1.0 модуля com.jdojo.layer

```
// module-info.com версия 1.0
module com.jdojo.layer {
    exports com.jdojo.layer;
}
```

Листинг 10.14. Класс LayerInfo из версии 1.0 модуля com.jdojo.layer

```
// LayerInfo.java
package com.jdojo.layer;

public class LayerInfo {
   private final static String VERSION = "1.0";

   static {
      System.out.println("Загружается LayerInfo версии " + VERSION);
   }

   public void printInfo() {
      Class cls = this.getClass();
      ClassLoader loader = cls.getClassLoader();
      Module module = cls.getModule();
      String moduleName = module.getName();
      ModuleLayer layer = module.getLayer();

      System.out.println("Версия класса: " + VERSION);
      System.out.println("Имя класса: " + cls.getName());
```

```
        System.out.println("Загрузчик класса: " + loader);
        System.out.println("Имя модуля: " + moduleName);
        System.out.println("Имя слоя: " + layer);
    }
}
```

В классе `LayerInfo` нет ничего сложного. В статической переменной `VERSION` хранится информация о версии. В статическом инициализаторе печатается сообщение, включающее эту информацию. Из него станет понятно, какая версия класса загружается. Метод `printInfo()` печатает подробные сведения: версию, имя класса, загрузчик класса, имя модуля и имя слоя.

В листингах 10.15 и 10.16 приведена версия 2.0 объявлений модуля `com.jdojo.layer` и класса `LayerInfo`. По сравнению с версией 1.0 изменилось только значение статической переменной `VERSION`.

Листинг 10.15. Версия 2.0 модуля com.jdojo.layer

```
// module-info.com версия 2.0
module com.jdojo.layer {
    exports com.jdojo.layer;
}
```

Листинг 10.16. Класс LayerInfo из версии 2.0 модуля com.jdojo.layer

```
// LayerInfo.java
package com.jdojo.layer;

public class LayerInfo {
    private final static String VERSION = "2.0";

    static {
        System.out.println("Загружается LayerInfo версии " + VERSION);
    }

    public void printInfo() {
        Class cls = this.getClass();
        ClassLoader loader = cls.getClassLoader();
        Module module = cls.getModule();
        String moduleName = module.getName();
        ModuleLayer layer = module.getLayer();

        System.out.println("Версия класса: " + VERSION);
        System.out.println("Имя класса: " + cls.getName());
        System.out.println("Загрузчик класса: " + loader);
        System.out.println("Имя модуля: " + moduleName);
        System.out.println("Имя слоя: " + layer);
    }
}
```

Мы готовы протестировать слои и загрузить обе версии модуля `com.jdojo.layer` в два разных слоя в контексте одной и той же JVM. Создайте модульный JAR-файл для версии 2.0, назовите его `com.jdojo.layer.v2.jar` или еще как-нибудь и поместите в каталог `C:\Java9Revealed\customLib`. Если вы решите поместить этот файл в какой-то другой каталог, то не забудьте изменить путь в листинге 10.18.

Программа тестирования слоев находится в модуле `com.jdojo.layer.test`, объявление которого приведено в листинге 10.17, где объявлена зависимость от версии 1.0 модуля `com.jdojo.layer`. Как проверить, что с модулем `com.jdojo.layer.test` используется именно версия 1.0 модуля `com.jdojo.layer`? Всего-то и нужно поместить код версии 1.0 на путь к модулям при выполнении модуля `com.jdojo.layer.test`. В NetBeans для этого следует поместить проект `com.jdojo.layer.v1` на путь к модулям в проекте `com.jdojo.layer.test`.

В листинге 10.18 приведен код класса `LayerTest`, в котором создается пользовательский слой и в него загружаются модули. Подробное объяснение логики следует ниже.

Листинг 10.17. Объявление модуля com.jdojo.later.test

```
// module-info.java
module com.jdojo.layer.test {
    // Этот модуль читает версию 1.0 модуля com.jdojo.layer
    requires com.jdojo.layer;
}
```

Листинг 10.18. Класс LayerTest

```
// LayerTest.java
package com.jdojo.layer.test;

import java.lang.module.Configuration;
import java.lang.module.ModuleFinder;
import java.lang.reflect.Method;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Set;

public class LayerTest {
   public static void main(String[] args) {
      /* Местоположение пользовательского модуля. Измените путь, так чтобы он
         указывал на каталог, содержащий модульный JAR-файл модуля
         com.jdojo.layer (версии 2.0) на вашей машине.
      */
      final String CUSTOM_MODULE_LOCATION = "C:\\Java9Revealed\\customLib";

      // Определить набор корневых модулей, которые должны быть разрешены
      // в пользовательском слое
```

```
    Set<String> rootModules = Set.of("com.jdojo.layer");

    // Создать пользовательский слой
    ModuleLayer customLayer = createLayer(CUSTOM_MODULE_LOCATION,
                                          rootModules);

    // Протестировать класс в начальном слое
    ModuleLayer bootLayer = ModuleLayer.boot();
    testLayer(bootLayer);
    System.out.println();

    // Протестировать класс в пользовательском слое
    testLayer(customLayer);
}

public static ModuleLayer createLayer(String modulePath,
                                      Set<String> rootModules) {
    Path path = Paths.get(modulePath);

    // Определить локаторы модулей, используемые при создании
    // конфигурации для пользовательского слоя
    ModuleFinder beforFinder = ModuleFinder.of(path);
    ModuleFinder afterFinder = ModuleFinder.of();

    // Создать конфигурацию для пользовательского слоя
    Configuration parentConfig = ModuleLayer.boot().configuration();
    Configuration config =
         parentConfig.resolve(beforFinder, afterFinder, rootModules);

    /* Создать пользовательский слой с одним загрузчиком классов. Родителем
       этого загрузчика будет системный загрузчик, а родителем
       пользовательского слоя — начальный слой.
    */
    ClassLoader sysClassLoader = ClassLoader.getSystemClassLoader();
    ModuleLayer parentLayer = ModuleLayer.boot();
    ModuleLayer layer = parentLayer.defineModulesWithOneLoader(config,
            sysClassLoader);

    // Проверить, загружен ли модуль в этот слой
    if (layer.modules().isEmpty()) {
       System.out.println("\nНе найден модуль " + rootModules
          + " на пути " + modulePath + ". "
          + "Проверьте, существует ли файл com.jdojo.layer.v2.jar "
          + "в этом месте." + "\n");
    }

    return layer;
```

```
    }

    public static void testLayer(ModuleLayer layer) {
        final String moduleName = "com.jdojo.layer";
        final String className = "com.jdojo.layer.LayerInfo";
        final String methodName = "printInfo";
        try {
            // Загрузить класс
            Class<?> cls = layer.findLoader(moduleName)
                                .loadClass(className);

            // Создать экземпляр класса с помощью конструктора без аргументов
            Object obj = cls.getConstructor().newInstance();

            // Найти метод
            Method method = cls.getMethod(methodName);

            // Вызвать метод, который напечатает подробные сведения
            method.invoke(obj);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Я объясню логику каждого метода класса `LayerTest`. В методе `main()` объявлена переменная `CUSTOM_MODULE_LOCATION`, в которой хранится местоположение версии 2.0 модуля `com.jdojo.layer`. Если на вашей машине откомпилированный код хранится в другом месте, измените значение этой переменной.

```
final String CUSTOM_MODULE_LOCATION = "C:\\Java9Revealed\\customLib";
```

Модуль `com.jdojo.layer` является единственным корневым модулем в конфигурации пользовательского слоя:

```
Set<String> rootModules = Set.of("com.jdojo.layer");
```

Для создания пользовательского слоя вызывается метод `createLayer()`, который помещает в слой версию 2.0 модуля `com.jdojo.layer`, находящуюся в каталоге `CUSTOM_MODULE_LOCATION`:

```
ModuleLayer customLayer = createLayer(CUSTOM_MODULE_LOCATION, rootModules);
```

Метод `main()` получает ссылку на начальный слой:

```
ModuleLayer bootLayer = ModuleLayer.boot();
```

Теперь вызывается метод `testLayer()`, сначала для начального слоя, а затем для пользовательского. Метод находит загрузчик классов, соответствующий модулю `com.jdojo.layer` в слое, и загружает класс `com.jdojo.layer.LayerInfo`.

```
final String moduleName = "com.jdojo.layer";
```

```
final String className = "com.jdojo.layer.LayerInfo";
final String methodName = "printInfo";
Class<?> cls = layer.findLoader(moduleName)
                    .loadClass(className);
```

С помощью конструктора без аргументов создается объект класса `LayerInfo`:

```
Object obj = cls.getConstructor().newInstance();
```

Наконец, мы получаем ссылку на метод `printInfo()` класса `LayerInfo` и вызываем его:

```
Method method = cls.getMethod(methodName);
method.invoke(obj);
```

Класс `LayerTest` можно выполнить в `NetBeans` или с помощью показанной ниже команды. На вашей машине результат может отличаться. Имя слоя – это список всех модулей в этом слое, именно такое значение возвращает метод `toString()` класса `ModuleLayer`.

```
C:\Java9Revealed>java --module-path com.jdojo.layer.v1\dist;com.jdojo.layer.test\dist
--module com.jdojo.layer.test/com.jdojo.layer.test.LayerTest
```

```
Загружается LayerInfo версии 1.0
Версия класса: 1.0
Имя класса: com.jdojo.layer.LayerInfo
Загрузчик класса: jdk.internal.loader.ClassLoaders$AppClassLoader@6e3c1e69
Имя модуля: com.jdojo.layer
Имя слоя: java.security.jgss, jdk.unsupported, jdk.jlink, jdk.security.jgss, jdk.javadoc,
jdk.crypto.cryptoki, java.naming, jdk.jartool, java.xml.crypto, jdk.deploy, java.
logging, jdk.snmp, jdk.zipfs, jdk.crypto.mscapi, jdk.naming.dns, java.smartcardio,
java.base, jdk.crypto.ec, jdk.dynalink, jdk.compiler, java.compiler, jdk.jdeps,
java.rmi, java.xml, com.jdojo.layer.test, jdk.management, java.datatransfer, jdk.
scripting.nashorn, java.desktop, java.management, jdk.naming.rmi, java.scripting,
jdk.localedata, jdk.accessibility, jdk.charsets, com.jdojo.layer, java.security.sasl,
jdk.security.auth, jdk.internal.opt, java.prefs

Загружается LayerInfo версии 2.0
Версия класса: 2.0
Имя класса: com.jdojo.layer.LayerInfo
Загрузчик класса: jdk.internal.loader.Loader@4cb2c100
Имя модуля: com.jdojo.layer
Имя слоя: com.jdojo.layer
```

Резюме

API модулей состоит из классов и интерфейсов, дающих доступ к модулям из программы. С его помощью можно программно читать, модифицировать и строить

дескрипторы модулей, загружать модули, читать содержимое модуля, создавать слои и т. д. API модулей невелик, он содержит полтора десятка классов и интерфейсов, распределенных между двумя пакетами: `java.lang` и `java.lang.module`. Классы `Module`, `ModuleLayer` и `LayerInstantiationException` находятся в пакете `java.lang.module`.

Экземпляр класса `Module` представляет модуль во время выполнения. Каждый загруженный в JVM тип принадлежит какому-то модулю. В JDK 9 в класс `Class` добавлен метод `getModule()`, возвращающий класс, которому принадлежит модуль.

Экземпляр класса `ModuleDescriptor` представляет определение модуля, созданное по его объявлению, обычно прочитанное из файла `module-info.class`. Дескриптор модуля можно создать и динамически с помощью класса `ModuleDescriptor.Builder`. Объявление модуля можно пополнить из командной строки, задавая параметры `--add-reads`, `--add-exports` и `--add-opens`, или программно, с помощью методов класса `Module`: `addReads()`, `addOpens()` и `addExports()`. Класс `ModuleDescriptor` представляет дескриптор модуля в том виде, в каком он существовал в момент объявления, т. е. еще не пополненный. Метод `getDescriptor()` класса `Module` возвращает объект `ModuleDescriptor`. Объект `ModuleDescriptor` неизменяемый. У безымянного модуля нет дескриптора, для него метод `getDescriptor()` возвращает `null`. В классе `ModuleDescriptor` несколько вложенных классов, в частности, `ModuleDescriptor.Requires`; каждый из них представляет одно предложение модуля.

Дескриптор модуля можно пополнить из командной строки и с помощью API модулей. Все запросы, касающиеся свойств модуля, можно отнести к одной из двух категорий: для одних ответ может измениться после загрузки модуля, для других не может. Класс `Module` предоставляет методы для запросов из первой категории, а класс `ModuleDescriptor` – из второй.

Определение модуля можно модифицировать во время выполнения с помощью следующих методов класса `Module`: `addExports()`, `addOpens()`, `addReads()` и `addUses()`.

Объявления модулей можно снабжать аннотациями. В перечисление `java.lang.annotation.ElementType` добавлен новый элемент `MODULE`. Если целевым типом в объявлении аннотации является `MODULE`, то аннотацию разрешено применять к модулям. В Java 9 две аннотации – `java.lang.Deprecated` и `java.lang.SuppressWarnings` – модифицированы и теперь могут применяться к объявлениям модулей тоже. Эти аннотации влияют только на объявление самого модуля, но не содержащихся в нем типов.

Модули собраны в слои. Слой – это множество разрешенных модулей плюс функция, сопоставляющая каждому модулю загрузчик классов, отвечающий за загрузку всех типов в этом модуле. Множество разрешенных модулей называется *конфигурацией*. Слои образуют иерархию. У каждого слоя имеется хотя бы один родительский слой; исключение составляет *пустой* слой, который не содержит никаких модулей, его основное назначение – быть родителем *начального* слоя. Начальный слой создается JVM на этапе инициализации путем разрешения корневых модулей приложения относительно множества видимых модулей. Можно создавать пользовательские слои. Благодаря слоям мы можем загружать разные версии одного модуля в разные слои и использовать в контексте одной и той же JVM.

Глава 11

Оболочка Java

Краткое содержание главы:

- что такое оболочка Java;
- что такое программа `JShell` и `JShell` API;
- конфигурирование `JShell`;
- использование `JShell` для выполнения фрагментов Java-кода;
- использование `JShell` API для выполнения фрагментов Java-кода.

Что такое оболочка Java?

Оболочка Java, которая в JDK 9 называется `JShell`, – это командная утилита, позволяющая интерактивно работать с языком Java, т. е. выполнять фрагменты Java-кода без написания законченной программы. Это цикл REPL (Read-Eval-Print loop – цикл чтения-выполнения-печати) для Java. Вместе с тем `JShell` – еще и API, дающий возможность разрабатывать приложения с такой же функциональностью, как у программы `JShell`.

Название REPL происходит от трех примитивных функций языка Lisp – `read`, `eval`, `print` – выполняемых в цикле. Функция `read` читает то, что вводит пользователь, и строит из прочитанного структуру данных; функция `eval` выполняет разобранную структуру и возвращает результат, а функция `print` этот результат печатает. Затем программа готова снова принимать данные от пользователя, т. е. возвращается к началу цикла. Акронимом REPL теперь обозначают любую интерактивную программу, позволяющую взаимодействовать с языком программирования. На рис. 11.1 показана принципиальная схема REPL. Именно так работает оболочка UNIX и консоль Windows: читают команду операционной системы, выполняют ее, печатают результат и ждут следующей команды.

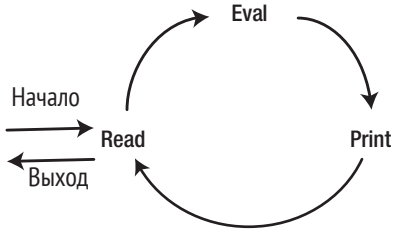

Рис. 11.1. Принципиальная схема цикла чтения-выполнения-печати

Зачем в JDK 9 включили `JShell`? Одна из основных причин – жалобы на сложность изучения Java. В других языках, например Lisp, Python, Ruby, Groovy, Clojure, цикл REPL уже давно поддерживается. А для того чтобы написать

даже простейшую программу «Hello, world!» на Java, приходится прибегать к другому циклу – Edit-Compile-Execute (ECEL – редактирование, компиляция, выполнение). И если потребуется внести изменение, то эти шаги придется повторить. Даже без учета служебных операций – создания структуры каталогов, компиляции и запуска программы – вот минимальный текст, который необходимо написать, чтобы модульная Java-программа в JDK 9 напечатала сообщение «Hello, world!»:

```
// module-info.java
module HelloWorld {
}

// HelloWorld.java
package com.jdojo.intro;

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

Написание полной программы только для того, чтобы вычислить такое простое выражение, – это перебор. Потому-то в учебных заведениях и перестают преподавать Java в качестве первого языка программирования. Проектировщики Java прислушались к мнению академических кругов и включили `JShell` в JDK 9. Теперь для достижения того же эффекта достаточно написать всего одну строчку кода в ответ на приглашение `jshell`:

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

jshell>
```

Первая строка – введенный вами код, вторая – напечатанный им результат. После печати снова появляется приглашение `jshell` и можно вводить следующее выражение.

> **Совет.** JShell – это не новый язык и не новый компилятор. Это инструментальная программа и API для интерактивной работы с языком Java. Для начинающих она предлагает способ быстро исследовать возможности языка, а для опытных разработчиков – быстро ознакомиться с результатами фрагмента кода без необходимости компилировать и запускать полную программу. Она также позволяет быстро разработать прототип приложения, применяя инкрементный подход, – добавить кусок, посмотреть, что получилось, добавить следующий кусок – и так, пока не будет готов прототип.

В состав JDK 9 входит команда `jshell` и `JShell` API. Все, что программа поддерживает на уровне командной строки, поддерживает и API. То есть фрагмент кода можно выполнить из командной строки или с помощью API. В обсуждении ниже вы должны научиться различать эти контексты. Большая часть главы посвящена командной программе, но в последнем разделе на примере описывается API.

Архитектура JShell

Компилятор Java не распознает такие фрагменты, как объявление метода или переменной, в отрыве от контекста. На верхнем уровне могут находиться только классы или предложения импорта, и лишь они могут существовать сами по себе. Любой другой код должен быть частью класса. А `JShell` позволяет выполнять и постепенно усложнять фрагменты Java-кода.

Руководящий принцип современной архитектуры `JShell` – применять имеющиеся в Java языковые средства и другие технологии, включенные в JDK, чтобы сохранять совместимость с настоящей и будущими версиями языка. По мере развития языка будет развиваться и `JShell`, причем в сам код `JShell` не нужно будет вносить никаких или почти никаких изменений. На рис. 11.2 показана высокоуровневая архитектура `JShell`.

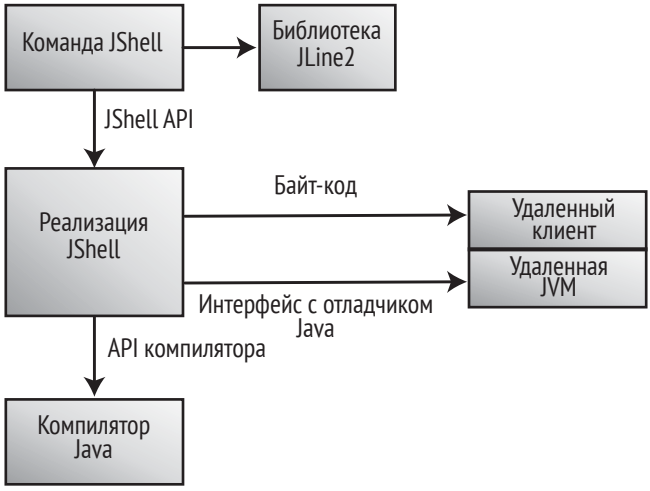

Рис. 11.2. Архитектура `JShell`

В `JShell` используется версия 2 библиотеки JLine для обработки ввода с консоли. Стандартный компилятор Java не знает, как разбирать и компилировать фрагмент кода. Поэтому в `JShell` имеется собственный анализатор, который разбирает фрагмент кода и определяет его тип, например: объявление метода, объявление переменной и т. д. После того как тип фрагмента определен, он обертывается синтетическим классом по следующим правилам:

- предложения импорта оставляются в неизменном виде и размещаются в начале синтетического класса;
- переменные, методы и объявления классов становятся статическими членами синтетического класса;
- выражения и предложения обертываются синтетическим методом синтетического класса.

Все синтетические классы входят в пакет `REPL`. Обернутые фрагменты компилируются стандартным компилятором Java с применением API компилятора. Компилятор принимает исходный код обертки в виде строки и преобразует его

в байт-код, хранящийся в памяти. Сгенерированный байт-код посылается через сокет удаленному процессу, исполняющему JVM, для загрузки и выполнения. Иногда фрагменты кода, загруженные в удаленную JVM, должны быть заменены командой `JShell`, для этого используется API отладчика Java.

Запуск команды JShell

В состав JDK 9 входит команда `jshell`, находящаяся в каталоге `JDK_HOME\bin`. Если JDK 9 установлен в каталог `C:\java9` в Windows, то исполняемый файл будет называться `C:\java9\bin\jshell.exe`. Для запуска `JShell` откройте окно команд и введите команду `jshell`[1]:

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro
jshell>
```

При вводе `jshell` в командной строке может возникнуть ошибка:

```
C:\Java9Revealed>jshell
"jshell" не является внутренней или внешней
командой, исполняемой программой или пакетным файлом.
C:\Java9Revealed>
```

Это значит, что каталог `JDK_HOME\bin` не включен в переменную среды `PATH`. Я установил JDK 9 в каталог `C:\java9`, поэтому на моем компьютере переменная `JDK_HOME` равна `C:\java9`. Чтобы исправить ошибку, либо включите каталог `C:\java9\bin` в переменную `PATH`, либо при запуске `jshell` указывайте полный путь, в моем случае `C:\java9\bin\jshell`. Ниже показано, как установить переменную среду `PATH` в Windows и выполнить `JShell`:

```
C:\Java9Revealed>SET PATH=C:\java9\bin;%PATH%
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro
jshell>
```

А вот как `jshell` запускается с указанием полного пути:

```
C:\Java9Revealed>C:\java9\bin\jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro
jshell>
```

Если `jshell` успешно запустилась, то она печатает приветствие, содержащее сведения о версии, и предложение ввести команду `/help intro` для получения краткого введения:

---

[1] На момент перевод книги команда `jshell` не была локализована. Тем не менее для удобства читателя ее сообщения переведены, хотя перевод может отличаться от окончательного. – *Прим. перев.*

```
jshell> /help intro
|
| intro
|
|  Команда jshell позволяет исполнять Java-код и сразу же получать результат.
|  Разрешается ввести определение (переменной, метода, класса и т. д.), например:
|  int x = 8
|  или выражение, например: x + x
|  или предложение Java, или предложение import.
|  Эти небольшие куски Java-кода называются 'фрагментами'.
|
|  Некоторые команды jshell предназначены для того, чтобы вы лучше понимали,
|  что делаете, и могли управлять работой программы, например: /list
|
|  Для получения списка команд введите /help

jshell>
```

Для получения справки вы всегда можете ввести команду `/help`, которая напечатает список команд с краткими пояснениями:

```
jshell> /help
<<Вывод не показан.>>
jshell>
```

Команда `jshell` понимает несколько аргументов. Например, можно передать информацию компилятору фрагментов или удаленной JVM, которая эти фрагменты выполняет. Укажите при запуске `jshell` параметр `-help`, чтобы получить список всех поддерживаемых стандартных параметров. Если указать параметр `--help-extra` или `-X`, то будет выведен список всех нестандартных параметров. С помощью параметров можно, к примеру, задать путь к классам и путь к модулям. Более подробно мы познакомимся с параметрами ниже.

Можно также настроить под себя скрипты запуска команды `jshell`, воспользовавшись параметром `--start`. В качестве его аргумента можно задать `DEFAULT` или `PRINTING`. Аргумент `DEFAULT` запускает `jshell` с несколькими предопределенными предложениями `import`, чтобы вам не нужно было импортировать общеупотребительные классы самостоятельно. Следующие две команды эквивалентны:

- `jshell`
- `jshell --start DEFAULT`

Для вывода сообщения в стандартный вывод используется метод `System.out.println()`. Можно запустить `jshell` с параметром `--start PRINTING`, и тогда все варианты методов `System.out.print()`, `System.out.println()` и `System.out.printf()` будут включены как методы верхнего уровня `print()`, `println()` и `printf()` соответственно. В результате можно будет писать просто `print()`, `println()` и `printf()`, опуская `System.out.`

```
C:\Java9Revealed>jshell --start PRINTING
|  Добро пожаловать в JShell -- версия 9-ea
```

```
|  Для получения вводной справки введите /help intro

jshell> println("hello")
hello

jshell>
```

Параметр `--start` можно указать дважды, чтобы включить и импорт по умолчанию, и сокращение методов печати:

```
C:\Java9Revealed>jshell --start DEFAULT --start PRINTING
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell>
```

Выход из JShell

Для выхода из `jshell` введите команду `/exit` и нажмите **Enter**:

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> /exit
|  До свидания

C:\Java9Revealed>
```

`JShell` снисходительна к ошибкам. Если вы введете незнакомое ключевое слово в конструкции Java, оно будет просто проигнорировано. Допустимы частичные команды. Если введенную частичную команду можно однозначно дополнить до полного имени команды, то так и будет сделано. Например, команды `/edit` и `/exit` обе начинаются с `/e`. Если ввести `/ex` вместо `/exit`, то `jshell` будет действовать, как будто введена команда `/exit`:

```
jshell> /ex
|  До свидания

C:\Java9Revealed>
```

Но если ввести `/e`, то будет выдано сообщение об ошибке, поскольку продолжить команду можно двумя способами:

```
jshell> /e
|  Команда: '/e' неоднозначна: /edit, /exit
|  Для получения справки введите /help.

jshell>
```

Что такое фрагменты и команды?

JShell можно использовать для двух целей:

- выполнение фрагментов Java-кода, которые в JShell называются просто фрагментами;
- выполнение команд, которые служат для опроса внутреннего состояния JShell и задания среды JShell.

Чтобы отличать команды от фрагментов, все команды начинаются знаком /. Выше мы уже встречались с командами, например, /exit и /help. Команды нужны для взаимодействия с самой программой, например, настройки вывода, печати справки, печати прошлых команд и фрагментов. Для получения полного перечня команд наберите команду /help.

Фрагменты кода должны следовать синтаксическим правилам, определенным в Спецификации языка Java Language Specification. Допустимы следующие фрагменты:

- объявление импорта;
- объявление класса;
- объявление интерфейса;
- объявление метода;
- объявление поля;
- предложение;
- выражение.

> **Совет.** В JShell можно использовать все конструкции языка Java за исключением пакетов. Все фрагменты в JShell обернуты внутренним пакетом REPL и синтетическим внутренним классом.

JShell знает, когда ввод фрагмента завершен. После нажатия **Enter** программа либо выполняет фрагмент, если он уже завершен, либо переходит на следующую строку и ждет завершения фрагмента. Если строка начинается приглашением ...>, значит, фрагмент еще не завершен. Приглашение к продолжению ввода, по умолчанию равное ...>, можно настроить. Приведем несколько примеров:

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> 2 + 2
$1 ==> 4

jshell> 2 +
  ...> 2
$2 ==> 4

jshell> 2
$3 ==> 2

jshell>
```

Если ввести `2 + 2` и нажать **Enter**, то `jshell` сочтет, что фрагмент завершен (это выражение). Она вычислит выражение, напечатает результат 4 и присвоит его переменной `$1`. Переменная `$1` автоматически генерируется программой `JShell`. О сгенерированных переменных мы подробнее поговорим ниже. Если ввести `2 +` и нажать **Enter**, то `jshell` предложит продолжить ввод, потому что `2 +` – незавершенный фрагмент Java-кода. После ввода `2` в следующей строке фрагмент становится завершенным; `jshell` вычисляет его и печатает результат. Если ввести `2` и нажать **Enter**, то `jshell` вычисляет фрагмент, потому что `2` – завершенное выражение.

Вычисление выражений

В `jshell` можно вычислить любое допустимое выражение Java. В следующих примерах вычисляется сумма и произведение чисел:

```
jshell> 2 + 2
$1 ==> 4

jshell> 9.0 * 6
$2 ==> 54.0
```

При вычислении выражения `jshell` печатает результат, если у выражения есть значение. В примерах выше `2 + 2` равно `4`, а `9.0 * 6` равно `54.0`. Значение выражения присваивается переменной. Напечатанный результат содержит имя переменной и значение выражения. Имена этих переменных можно использовать в других выражениях. Для печати значения достаточно ввести имя переменной:

```
jshell> $1
$1 ==> 4

jshell> $2
$2 ==> 54.0

jshell> System.out.println($1)
4

jshell> System.out.println($2)
54.0
```

> **Совет.** В `jshell` не нужно завершать предложение точкой с запятой, как в обычной Java-программе. `JShell` самостоятельно вставит недостающие точки с запятой.

В Java у каждой переменной имеется тип данных. А какого типа переменные `$1` и `$2` в примерах выше? В Java результатом вычисления выражения `2 + 2` является число типа `int`, а выражения `9.0 * 6` – число типа `double`. Поэтому типами данных `$1` и `$2` должны быть соответственно `int` и `double`. Как это проверить? Сначала пойдем по трудному пути. Приведем `$1` и `$2` к типу `Object` и вызовем метод `getClass()`, в результате должно получиться `Integer` и `Double`. Отметим, что примитивные типы `int` и `double` при приведении к типу `Object` обертываются ссылочными типами `Integer` и `Double`:

```
jshell> 2 + 2
$1 ==> 4

jshell> 9.0 * 6
$2 ==> 54.0

jshell> ((Object)$1).getClass()
$3 ==> class java.lang.Integer

jshell> ((Object)$2).getClass()
$4 ==> class java.lang.Double

jshell>
```

Есть и более простой способ узнать тип переменной, созданной `jshell`, – попросить `jshell` перейти в режим подробной выдачи, в котором печатаются типы создаваемых переменных и многое другое. Ниже показано, как установить этот режим и что в нем выдается при вычислении тех же самых выражений:

```
jshell> /set feedback verbose
|  Режим выдачи: verbose

jshell> 2 + 2
$1 ==> 4
|  создана временная переменная $1 : int

jshell> 9.0 * 6
$2 ==> 54.0
|  создана временная переменная $2 : double

jshell>
```

Как видим, типы созданных переменных `$1` и `$2` – `int` и `double` соответственно. Начинающим рекомендуется выполнять эту команду с параметром `-retain`, чтобы режим подробной выдачи сохранялся между сеансами `jshell`:

```
jshell> /set feedback -retain verbose
```

Можно также воспользоваться командой `/vars` для вывода списка всех переменных, определенных в `jshell`:

```
jshell> /vars
|  int $1 = 4
|  double $2 = 54.0

jshell>
```

Чтобы вернуться в режим нормальной выдачи, выполните такую команду:

```
jshell> /set feedback -retain normal
|  Режим выдачи: normal

jshell>
```

Вычислением простых выражений вида `2 + 2` возможности `jshell` не исчерпываются. Вычислить можно любое выражение Java. В примере ниже вычисляется конкатенация строк и используются методы класса `String`. Демонстрируется также использование циклов `for`:

```
jshell> "Hello " + "world! " + 2016
$1 ==> "Hello world! 2016"

jshell> $1.length()
$2 ==> 17

jshell> $1.toUpperCase()
$3 ==> "HELLO WORLD! 2016"

jshell> $1.split(" ")
$4 ==> String[3] { "Hello", "world!", "2016" }

jshell> for(String s : $4) {
  ...> System.out.println(s);
  ...> }
Hello
world!
2016

jshell>
```

Вывод списка фрагментов

Все введенное в `jshell` в конечном итоге становится частью какого-то фрагмента. Каждому фрагменту присваивается уникальный идентификатор, по которому впоследствии можно сослаться на фрагмент, например, чтобы его удалить. Команда `/list` выводит список фрагментов:

- `/list`
- `/list -all`
- `/list -start`
- `/list <snippet-name>`
- `/list <snippet-id>`

Без аргументов команда `/list` печатает все введенные пользователем активные фрагменты, в т. ч. прочитанные из файла командой `/open`.

При наличии аргумента `-all` печатаются все фрагменты – активные, неактивные, ошибочные и стартовые.

При наличии аргумента `-start` печатаются только стартовые фрагменты, которые кэшируются. Стартовые фрагменты печатаются, даже если в текущем сеансе они были удалены.

У некоторых фрагментов есть имя (например, у объявлений переменных и методов), и у любого фрагмента есть идентификатор. Если указать в команде `/list` имя или идентификатор, то будет напечатан только этот фрагмент.

Команда `/list` печатает список фрагментов в таком формате:

```
<snippet-id> : <snippet-source-code>
<snippet-id> : <snippet-source-code>
<snippet-id> : <snippet-source-code>
...
```

`JShell` генерирует уникальные идентификаторы фрагментов. Стартовые фрагменты имеют идентификаторы `s1`, `s2`, `s3...`, правильные фрагменты – `1`, `2`, `3...`, ошибочные – `e1`, `e2`, `e3....` В следующем примере сеанса `jshell` показано, как вывести список фрагментов командой `/list`. В частности, демонстрируется использование команды `/drop` для удаления фрагментов по имени и по идентификатору:

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> /list

jshell> 2 + 2
$1 ==> 4

jshell> /list

  1 : 2 + 2

jshell> int x = 100
x ==> 100

jshell> /list

  1 : 2 + 2
  2 : int x = 100;

jshell> /list -all

  s1 : import java.io.*;
  s2 : import java.math.*;
  s3 : import java.net.*;
  s4 : import java.nio.file.*;
  s5 : import java.util.*;
  s6 : import java.util.concurrent.*;
  s7 : import java.util.function.*;
  s8 : import java.util.prefs.*;
  s9 : import java.util.regex.*;
 s10 : import java.util.stream.*;
   1 : 2 + 2
```

```
   2 : int x = 100;

jshell> /list -start

  s1 : import java.io.*;
  s2 : import java.math.*;
  s3 : import java.net.*;
  s4 : import java.nio.file.*;
  s5 : import java.util.*;
  s6 : import java.util.concurrent.*;
  s7 : import java.util.function.*;
  s8 : import java.util.prefs.*;
  s9 : import java.util.regex.*;
 s10 : import java.util.stream.*;

jshell> string str = "using invalid type string"
|  Ошибка:
|  не найден символ
|    символ: class string
|  string str = "using invalid type string";
|  ^----^

jshell> /list

  1 : 2 + 2
  2 : int x = 100;

jshell> /list -all

  s1 : import java.io.*;
  s2 : import java.math.*;
  s3 : import java.net.*;
  s4 : import java.nio.file.*;
  s5 : import java.util.*;
  s6 : import java.util.concurrent.*;
  s7 : import java.util.function.*;
  s8 : import java.util.prefs.*;
  s9 : import java.util.regex.*;
 s10 : import java.util.stream.*;
   1 : 2 + 2
   2 : int x = 100;
  e1 : string str = "using invalid type string";

jshell> /drop 1
|  удалена переменная $1

jshell> /list
```

```
   2 : int x = 100;

jshell> /drop x
|  удалена переменная x

jshell> /list

jshell> /list -all

  s1 : import java.io.*;
  s2 : import java.math.*;
  s3 : import java.net.*;
  s4 : import java.nio.file.*;
  s5 : import java.util.*;
  s6 : import java.util.concurrent.*;
  s7 : import java.util.function.*;
  s8 : import java.util.prefs.*;
  s9 : import java.util.regex.*;
 s10 : import java.util.stream.*;
   1 : 2 + 2
   2 : int x = 100;
  e1 : string str = "using invalid type string";

jshell>
```

Имена переменных, методов и классов становятся именами фрагментов. Отметим, что в Java переменная, метод и класс могут иметь одинаковое имя, потому что они принадлежат разным пространствам имен. Эти сущности можно напечатать, указав имя в команде `/list`:

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> /list x
|  Нет такого фрагмента: x

jshell> int x = 100
x ==> 100

jshell> /list x

   1 : int x = 100;

jshell> void x(){}
|  создан метод x()
```

```
jshell> /list x

   1 : int x = 100;
   2 : void x(){}

jshell> void x(int n) {}
|  создан метод x(int)

jshell> /list x

   1 : int x = 100;
   2 : void x(){}
   3 : void x(int n) {}

jshell> class x{}
|  создан класс x

jshell> /list x
   1 : int x = 100;
   2 : void x(){}
   3 : void x(int n) {}
   4 : class x{}

jshell>
```

Редактирование фрагментов

`JShell` предлагает несколько способов редактирования фрагментов и команд. Для перемещения по командной строке во время ввода фрагментов и команд можно пользоваться клавишами, перечисленными в табл. 11.1, а для редактирования текста в строке – клавишами из табл. 11.2

Таблица 11.1. Клавиши навигации для редактирования в JShell

Клавиша	Описание
`Enter`	Завершить ввод текущей строки
`Left-Arrow`	Назад на один символ
`Right-Arrow`	Вперед на один символ
`Ctrl-A`	В начало строки
`Ctrl-E`	В конец строки
`Ctrl-B` (или `Alt-B`)	Назад на одно слово
`Meta-F` (или `Alt-F`)	Вперед на одно слово

Таблица 11.2. Клавиши модификации текста в JShell

Клавиша	Описание
`Delete`	Удалить символ под курсором
`Backspace`	Удалить символ до курсора
`Ctrl-K`	Удалить текст от курсора до конца строки
`Meta-D` (или `Alt-D`)	Удалить текст от курсора до конца слова
`Ctrl-W`	Удалить текст от курсора до предыдущего пробельного символа
`Ctrl-Y`	Вставить последний удаленный текст
`Meta-Y` (или `Alt-Y`)	После `Ctrl-Y` эта комбинация клавиш в цикле перебирает ранее удаленные куски текста

В `JShell` трудно редактировать многострочные фрагменты тексты, несмотря даже на широкий набор комбинаций клавиш. Проектировщики понимали эту проблему и предоставили встроенный редактор фрагментов. Но можно настроить `JShell`, так чтобы использовался ваш любимый редактор (см. раздел «Задание редактора фрагментов» ниже).

Чтобы начать редактирование фрагмента, введите команду `/edit`. У нее есть три варианта:

- `/edit <snippet-name>`
- `/edit <snippet-id>`
- `/edit`

Можно указать имя или идентификатор фрагмента. Если аргумент вообще не задан, то команда `/edit` открывает в редакторе все фрагменты. По умолчанию используется встроенный редактор JShell Edit Pad, показанный на рис. 11.3.

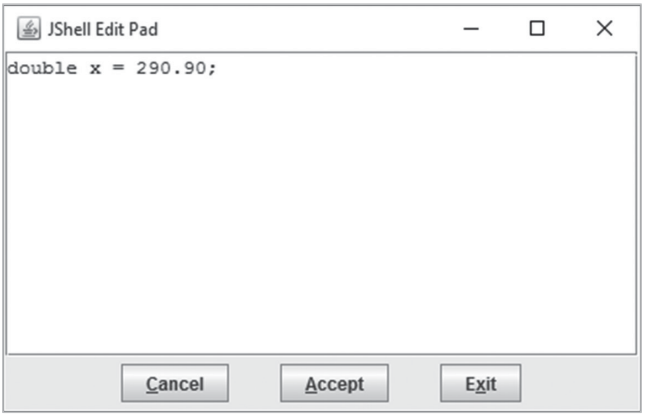

Рис. 11.3. Встроенный в JShell редактор JShell Edit Pad

JShell Edit Pad написан на Swing, он состоит из объекта `JFrame` с текстовой областью `JTextArea` и тремя кнопками `JButton`. Если будете редактировать в нем фрагменты, не забывайте нажать кнопку **Accept**, перед тем как выйти из окна, – только тогда изменения вступят в силу. Если нажать кнопку **Cancel** или **Exit**, не подтвердив изменения, то они будут потеряны.

Если вы знаете имя переменной, метода или класса, то можете редактировать фрагмент по имени. В показанном ниже сеансе `jshell` создается переменная, методы и класс с одним и тем же именем `x`, а затем с помощью команды `/edit x` они редактируются все сразу:

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> int x = 100
x ==> 100

jshell> void x(){}
|  создан метод x()

jshell> void x (int n) {}
|  создан метод x(int)

jshell> class x{}
|  создан класс x

jshell> 2 + 2
$5 ==> 4

jshell> /edit x
```

Команда `/edit x` открывает все фрагменты с именем `x` в JShell Edit Pad, как показано на рис. 11.4. Вы можете отредактировать их, подтвердить изменения, выйти из редактора и продолжить сеанс `jshell`.

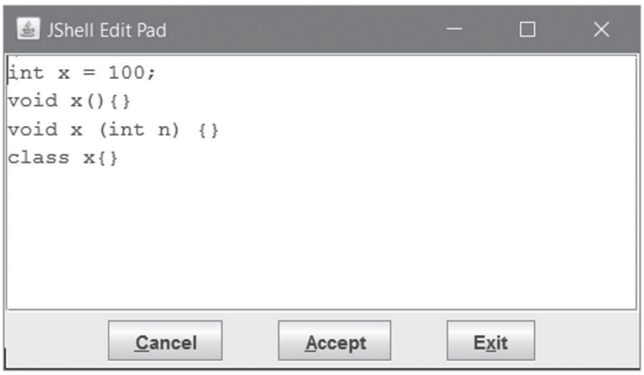

Рис. 11.4. Редактирование фрагмента по имени

Повторное выполнение предыдущих фрагментов

Часто возникает необходимость повторно выполнить введенные ранее фрагменты. Для перемещения по истории фрагментов и команд предназначены клавиши со стрелками вверх и вниз. Найдя нужный фрагмент или команду, нажмите **Enter**. Можно вместо этого воспользоваться следующими командами (но только для повторного выполнения фрагментов, а не команд):

- `/!`
- `/<snippet-id>`
- `/-<n>`

Команда `/!` повторно выполняет последний фрагмент. Команда `/<snippet-id>` выполняет фрагмент с идентификатором `<snippet-id>`. Команда `/-<n>` выполняет *n*-ый предыдущий фрагмент, например: `/-1` выполняет последний фрагмент, `/-2` – предпоследний и т. д. Команды `/!` и `/-1` эквивалентны.

Объявление переменных

В `jshell` переменные объявляются так же, как в Java-программе. Переменную можно объявить на верхнем уровне, внутри метода или внутри класса (поле). Модификаторы `static` и `final` в объявлениях переменных верхнего уровня не допускаются. Если они все-таки присутствуют, то игнорируются с выдачей предупреждения. Модификатор `static` означает, что контекстом переменной является класс, а `final` запрещает последующую модификацию значения переменной. Они не допускаются, потому что `jshell` позволяет объявлять свободные (не связанные с классом) переменные, значения которых можно изменять. В примерах ниже демонстрируется объявление переменных:

```
c:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> int x
x ==> 0

jshell> int y = 90
y ==> 90

jshell> side = 90
|  Ошибка:
|  символ не найден
|  символ: variable side
|  side = 90
|  ^--^

jshell> static double radius = 2.67
```

```
|  Предупреждение:
|  Модификатор 'static' не разрешен в объявлениях верхнего уровня, игнорируется
|  static double radius = 2.67;
|  ^----^
radius ==> 2.67

jshell> String str = new String("Hello")
str ==> "Hello"

jshell>
```

Использование необъявленной переменной в выражениях верхнего уровня приводит к ошибке, как в случае переменной `side` в примере выше. Ниже я покажу, как можно использовать необъявленную переменную в теле метода.

Разрешается также изменять тип переменной. Можно объявить переменную `x` сначала как `int`, а затем ее же – как `double` или `String`:

```
jshell> int x = 10;
x ==> 10

jshell> int y = x + 2;
y ==> 12

jshell> double x = 2.71
x ==> 2.71

jshell> y
y ==> 12

jshell> String x = "Hello"
x ==> "Hello"

jshell> y
y ==> 12

jshell>
```

Обратите внимание, что значение переменной `y` не изменилось и не пересчиталось после изменения значения `x`.

Можно также удалить переменную командой `/drop`, которая принимает имя переменной в качестве аргумента. Следующая команда удаляет переменную `x`:

```
jshell> /drop x
```

Команда `/vars` выводит список переменных – объявленных пользователем и автоматически сгенерированных `jshell` при вычислении выражений, возвращающих результат. У этой команды есть несколько вариантов:

- `/vars`
- `/vars <variable-name>`

- `/vars <variable-snippet-id>`
- `/vars -start`
- `/vars -all`

Команда без аргументов выводит список всех активных переменных в текущем сеансе. Если указано имя или идентификатор фрагмента, то выводится объявление переменной в этом фрагменте. Если задан аргумент `-start`, то выводятся все переменные, объявленные в стартовом скрипте. При наличии аргумента `-all` выводятся все вообще переменные, включая ошибочные, переопределенные, удаленные и стартовые. Ниже приведены примеры использования команды `/vars`:

```
c:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> /vars

jshell> 2 + 2
$1 ==> 4

jshell> /vars
|  int $1 = 4

jshell> int x = 20;
x ==> 20

jshell> /vars
|  int $1 = 4
|  int x = 20

jshell> String str = "Hello";
str ==> "Hello"

jshell> /vars
|  int $1 = 4
|  int x = 20
|  String str = "Hello"

jshell> double x = 90.99;
x ==> 90.99

jshell> /vars
|  int $1 = 4
|  String str = "Hello"
|  double x = 90.99

jshell> /drop x
```

```
|  удалена переменная x

jshell> /vars
|  int $1 = 4
|  String str = "Hello"

jshell>
```

Предложения import

В `jshell` можно использовать предложения `import`. Напомним, что в Java-программе все типы из пакета `java.lang` импортируются по умолчанию. А чтобы воспользоваться типами из других пакетов, необходимо включить в единицу компиляции соответствующие предложения `import`. Начнем с примера – попробуем создать три объекта: `String`, `List<Integer>` и `ZonedDateTime`. Класс `String` находится в пакете `java.lang`, классы `List` и `Integer` – в пакетах `java.util` и `java.lang` соответственно, а класс `ZonedDateTime` – в пакете `java.time`.

```
jshell> String str = new String("Hello")
str ==> "Hello"

jshell> List<Integer> nums = List.of(1, 2, 3, 4, 5)
nums ==> [1, 2, 3, 4, 5]

jshell> ZonedDateTime now = ZonedDateTime.now()
|  Ошибка:
|  символ не найден
|    символ: class ZonedDateTime
|  ZonedDateTime now = ZonedDateTime.now();
|  ^-----------^
|  Ошибка:
|    символ не найден
|  символ: variable ZonedDateTime
|  ZonedDateTime now = ZonedDateTime.now();
|                      ^-----------^

jshell>
```

При попытке воспользоваться классом `ZonedDateTime` из пакета `java.time` возникла ошибка. Мы ожидали, что аналогичная ошибка возникнет и при попытке создать объект `List`, поскольку он находится в пакете `java.util`, который по умолчанию не импортируется.

Но главная и единственная цель JShell – облегчить жизнь разработчикам, когда им нужно выполнить фрагмент кода. Для этого программа по умолчанию импортирует типы из нескольких пакетов. Из каких именно? Это легко узнать, напечатав список всех активных предложений `import` командой `/imports`:

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

jshell>
```

Как видим, в этот список входит пакет `java.util`. Именно поэтому при создании списка импортировать его необязательно. Мы можем включить и свои предложения `import`. В следующем примере показано, как импортировать и использовать класс `ZonedDateTime`.

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

jshell> import java.time.*

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

jshell> ZonedDateTime now = ZonedDateTime.now()
now ==> 2016-11-11T10:39:10.497234400-06:00[America/Chicago]

jshell>
```

Все предложения `import`, добавленные в сеансе `jshell`, забываются при выходе из сеанса. Можно также явно удалить предложения `import` – включенные по умолчанию или добавленные вами позже. Для этого нужно только знать идентификатор фрагмента. Напомним, что стартовые фрагменты нумеруются `s1`, `s2`, `s3`, …, а пользовательские – `1`, `2`, `3`, … . В следующих примерах показано, как добавить и удалить предложения `import`:

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> import java.time.*

jshell> List<Integer> list = List.of(1, 2, 3, 4, 5)
list ==> [1, 2, 3, 4, 5]

jshell> ZonedDateTime now = ZonedDateTime.now()
now ==> 2017-02-19T21:08:08.802099-06:00[America/Chicago]
jshell> /list -all

  s1 : import java.io.*;
  s2 : import java.math.*;
  s3 : import java.net.*;
  s4 : import java.nio.file.*;
  s5 : import java.util.*;
  s6 : import java.util.concurrent.*;
  s7 : import java.util.function.*;
  s8 : import java.util.prefs.*;
  s9 : import java.util.regex.*;
 s10 : import java.util.stream.*;
   1 : import java.time.*;
   2 : List<Integer> list = List.of(1, 2, 3, 4, 5);
   3 : ZonedDateTime now = ZonedDateTime.now();

jshell> /drop s5

jshell> /drop 1

jshell> /list -all

  s1 : import java.io.*;
  s2 : import java.math.*;
  s3 : import java.net.*;
  s4 : import java.nio.file.*;
  s5 : import java.util.*;
  s6 : import java.util.concurrent.*;
  s7 : import java.util.function.*;
```

```
  s8 : import java.util.prefs.*;
  s9 : import java.util.regex.*;
 s10 : import java.util.stream.*;
   1 : import java.time.*;
   2 : List<Integer> list = List.of(1, 2, 3, 4, 5);
   3 : ZonedDateTime now = ZonedDateTime.now();

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

jshell> List<Integer> list2 = List.of(1, 2, 3, 4, 5)
|   Ошибка:
|   символ не найден
|     символ: class List
|   List<Integer> list2 = List.of(1, 2, 3, 4, 5);
|   ^--^
|   Ошибка:
|   символ не найден
|     символ: variable List
|   List<Integer> list2 = List.of(1, 2, 3, 4, 5);
|                         ^--^

jshell> import java.util.*
|  update replaced variable list, reset to null

jshell> List<Integer> list2 = List.of(1, 2, 3, 4, 5)
list2 ==> [1, 2, 3, 4, 5]

jshell> /list -all

  s1 : import java.io.*;
  s2 : import java.math.*;
  s3 : import java.net.*;
  s4 : import java.nio.file.*;
  s5 : import java.util.*;
  s6 : import java.util.concurrent.*;
  s7 : import java.util.function.*;
  s8 : import java.util.prefs.*;
  s9 : import java.util.regex.*;
```

```
 s10 : import java.util.stream.*;
   1 : import java.time.*;
   2 : List<Integer> list = List.of(1, 2, 3, 4, 5);
   3 : ZonedDateTime now = ZonedDateTime.now();
  e1 : List<Integer> list2 = List.of(1, 2, 3, 4, 5);
   4 : import java.util.*;
   5 : List<Integer> list2 = List.of(1, 2, 3, 4, 5);

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

jshell>
```

Объявление методов

В `jshell` можно объявлять и вызывать методы. Методы верхнего уровня вводятся прямо в `jshell` и не находятся внутри какого-нибудь класса. Можно также объявлять классы (см. следующий раздел), содержащие методы. В этом разделе мы увидим, как объявлять и вызывать методы верхнего уровня. Разрешается вызывать и методы существующих классов. В следующем примере объявляется и вызывается метод `square()`:

```
jshell> long square(int n) {
   ...> return n * n;
   ...> }
|  создан метод square(int)

jshell> square(10)
$2 ==> 100

jshell> long n2 = square(37)
n2 ==> 1369

jshell>
```

В теле метода разрешены опережающие ссылки, т. е. можно ссылаться на еще не объявленные методы или переменные. Объявляемый метод нельзя вызывать, пока не будут определены все методы и переменные, на которые он ссылается.

```
jshell> long multiply(int n) {
   ...> return multiplier * n;
   ...> }
|  метод multiply(int) создан, но не может быть вызван, пока не объявлена
|  переменная multiplier

jshell> multiply(10)
|  попытка вызвать метод multiply(int), который не может быть вызван, пока не
|  объявлена переменная multiplier

jshell> int multiplier = 2
multiplier ==> 2

jshell> multiply(10)
$6 ==> 20

jshell> void printCube(int n) {
   ...> System.out.printf("%d в кубе равно %d.%n", n, cube(n));
   ...> }
|  метод printCube (int) создан, но не может быть вызван, пока не объявлен
|  метод cube(int)

jshell> long cube(int n) {
   ...> return n * n * n;
   ...> }
|   создан метод cube(int)

jshell> printCube(10)
10 в кубе равно 1000.

jshell>
```

В этом примере объявляется метод `multiply(int n)`. Он умножает аргумент на переменную `multiplier`, которая еще не объявлена. После объявления печатается предупреждение, в котором ясно говорится, что метод `multiply()` нельзя вызывать до объявления переменной `multiplier`. Попытка все же вызвать метод приводит к ошибке. Ниже мы объявили переменную `multiplier`, и теперь метод `multiply()` вызывается успешно.

Совет. Благодаря опережающим ссылкам можно объявлять рекурсивные методы.

Объявление типов

Как и в Java, в `jshell` можно объявлять любые типы: классы, интерфейсы, перечисления и аннотации. В следующем сеансе создается класс `Counter`, затем создается объект этого класса и вызываются его методы:

```
jshell> class Counter {
   ...> private int counter;
   ...> public synchronized int next() {
   ...>     return ++counter;
   ...> }
   ...>
   ...> public int current() {
   ...>     return counter;
   ...> }
   ...> }
|  создан класс Counter
jshell> Counter c = new Counter();
c ==> Counter@25bbe1b6

jshell> c.current()
$3 ==> 0

jshell> c.next()
$4 ==> 1

jshell> c.next()
$5 ==> 2

jshell> c.current()
$6 ==> 2

jshell>
```

Для вывода списка объявленных в `jshell` типов служит команда `/types`, имеющая несколько вариантов:

- `/types`
- `/types <type-name>`
- `/types <snippet-id>`
- `/types -start`
- `/types -all`

Без аргументов команда печатает активные классы, интерфейсы и перечисления, объявленные в `jshell`. Если задано имя типа или идентификатор фрагмента, то печатается только запрошенный тип. Если задан флаг `-start`, то печатаются автоматически добавленные стартовые типы. С флагом `-all` команда печатает все вообще типы, включая ошибочные, переопределенные, удаленные и стартовые. Ниже приводится продолжение предыдущего сеанса – показано, как напечатать все активные в текущем сеансе типы:

```
jshell> /types
|   class Counter

jshell>
```

Класс `Counter` невелик. Но вводить в командной строке исходный код класса побольше – удовольствие ниже среднего. Хотелось бы, конечно, вводить код в своем любимом редакторе, например, NetBeans, а для быстрого тестирования классов использовать `jshell`. Чтобы открыть исходный код в редакторе, не выходя из `jshell`, воспользуйтесь командой `/open`:

```
/open <file-path>
```

Исходный код класса `Counter` находится в файле `Java9Revealed/com.jdojo.jshell/src/Counter.java`. В следующем примере сеанса показано, как открыть сохраненный файл `Counter.java` в `jshell`. Предполагается, что исходный код сохранен на диске `C:\` в Windows. Если вы работаете в другой операционной системе, замените путь к файлу в соответствии с принятыми с ней соглашениями.

```
jshell> /open C:\Java9Revealed\com.jdojo.jshell\src\Counter.java

jshell> Counter c = new Counter()
c ==> Counter@25bbe1b6

jshell> c.current()
$3 ==> 0

jshell> c.next()
$4 ==> 1

jshell> c.next()
$5 ==> 2

jshell> c.current()
$6 ==> 2

jshell>
```

Отметим, что в исходном коде класса `Counter` нет объявления пакета, потому что `jshell` не позволяет объявлять класс (да и любой другой тип) в составе пакета. Все типы, объявленные в `jshell`, считаются статическими и находящимися во внутреннем синтетическом классе. Однако иногда хочется протестировать собственный класс, для которого пакет указан. В `jshell` разрешается использовать уже откомпилированный класс, находящийся в пакете. Обычно это необходимо, когда мы используем библиотеки при разработке приложения и хотим поэкспериментировать с небольшими фрагментами, в которых используются библиотечные классы. Чтобы найти классы, установить путь к ним командой `/env`.

Класс `Person` находится в пакете `com.jdojo.jshell`, включенном в исходный код, прилагаемый к книге. Его объявление показано в листинге 11.1.

Листинг 11.1. Исходный код класса Person

```
// Person.java
package com.jdojo.jshell;

public class Person {
```

```
   private String name;

   public Person() {
      this.name = "Unknown";
   }

   public Person(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
}
```

Показанная ниже команда `jshell` задает путь к классам в Windows в предположении, что исходный код к книге сохранен в каталоге `C:\`. Если эти предположения не выполняются, то команду следует модифицировать:

```
jshell> /env -class-path C:\Java9Revealed\com.jdojo.jshell\build\classes
|   Установка новых параметров и восстановление состояния.

jshell> Person guy = new Person("Martin Guy Crawford")
|   Ошибка:
|   символ не найден
|     символ: class Person
|   Person guy = new Person("Martin Guy Crawford");
|   ^----^
|   Ошибка:
|   символ не найден
|     символ: class Person
|   Person guy = new Person("Martin Guy Crawford");
|                    ^----^
```

Понимаете, в чем причина ошибки? Мы указали простое имя класса, `Person`, не импортировав класс, и `jshell` не смогла его найти. Необходимо либо импортировать класс `Person`, либо указать полное имя. Далее показано продолжение сеанса, в котором ошибка исправлена:

```
jshell> import com.jdojo.jshell.Person

jshell> Person guy = new Person("Martin Guy Crawford")
guy ==> com.jdojo.jshell.Person@192b07fd

jshell> guy.getName()
```

```
$9 ==> "Martin Guy Crawford"

jshell> guy.setName("Forrest Butts")

jshell> guy.getName()
$11 ==> "Forrest Butts"

jshell>
```

Установка среды выполнения

В предыдущем разделе мы узнали, как установить путь к классам командой `/env`. Эта команда применяется и для установки многих других частей контекста выполнения, в частности, пути к модулям. А также для разрешения модулей, чтобы в `jshell` можно было использовать типы, находящиеся в модулях. Полностью ее синтаксис выглядит так:

```
/env [-class-path <path>] [-module-path <path>] [-add-modules <modules>]
[-add-exports <m/p=n>]
```

Команда `/env` без аргументов печатает значения в текущем контексте выполнения. Параметр `-classpath` задает путь к классам, а `-module-path` – путь к модулям. Параметр `-add-modules` добавляет модули в набор корневых модулей, чтобы их можно было разрешить. Специальные значения этого параметра – `ALL-DEFAULT`, `ALL-SYSTEM` и `ALL-MODULE-PATH` – описаны в главе 2. Параметр `-add-exports` служит для экспорта неэкспортированных пакетов из модуля другим модулям. Семантика этих параметров такая же, как в командах `javac` и `java`.

> **Совет.** В командной строке этим параметрам предшествуют два дефиса, например, `--module-path`. В `jshell` можно использовать один или два дефиса, т. е. разрешены обе формы: `-module-path` и `--module-path`.

При задании параметров контекста выполнения текущий сеанс сбрасывается, и все ранее выполненные фрагменты повторно выполняются в безмолвном режиме, т. е. выполняемый фрагмент не отображается. Однако ошибки, возникшие при повторном выполнении, показываются.

Для установки контекста выполнения предназначены команды `/env`, `/reset` и `/reload`, оказывающие различное действие. Но семантика параметров контекста, например `-class-path` и `-module-path`, всегда одинакова. Чтобы получить список всех параметров, которые можно использовать для установки контекста, выполните команду `-/help context`.

Для примера рассмотрим использование относящихся к модулям параметров команды `/env`. В главе 3 мы создали модуль `com.jdojo.intro`. Этот модуль содержит, но не экспортирует пакет `com.jdojo.intro`. Мы хотим вызвать статический метод `main(String[] args)` класса `Welcome` из неэкспортированного пакета. В `jshell` для этого следует проделать следующие действия.

Задать путь к модулям, чтобы система нашла модуль.

- Разрешить модуль, добавив его в набор корневых. Для этого нужно воспользоваться параметром `-add-modules` команды `/env`.
- Экспортировать пакет командой `-add-exports`. Введенные в `jshell` фрагменты выполняются в безымянном модуле, поэтому пакет следует экспортировать всем безымянным модулям, задав специальное значение параметра, `ALL-UNNAMED`. Если в параметре `-add-exports` целевые модули не указаны, то по умолчанию подразумевается `ALL-UNNAMED`.
- Факультативно импортировать класс `com.jdojo.intro.Welcome`, если вы хотите ссылаться на него во фрагментах по короткому имени.
- Теперь в `jshell` можно вызывать метод `Welcome.main()`.

В показанном ниже сеансе `jshell` демонстрируется выполнение этих шагов. Предполагается, что при запуске сеанса текущим каталогом был `C:\Java9Revealed`, а откомпилированный код модуля `com.jdojo.intro` находится в каталоге `C:\Java9Revealed\com.jdojo.intro\build\classes`. Если на вашей машине это не так, измените пути, указанные в сеансе.

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> /env -module-path com.jdojo.intro\build\classes
|   Установка новых параметров и восстановление состояния.

jshell> /env -add-modules com.jdojo.intro
|   Установка новых параметров и восстановление состояния.

jshell> /env -add-exports com.jdojo.intro/com.jdojo.intro=ALL-UNNAMED
|   Установка новых параметров и восстановление состояния.

jshell> import com.jdojo.intro.Welcome

jshell> Welcome.main(null)
Добро пожаловать в систему модулей.
Имя модуля: com.jdojo.intro

jshell> /env
|   --module-path com.jdojo.intro\build\classes
|   --add-modules com.jdojo.intro
|   --add-exports com.jdojo.intro/com.jdojo.intro=ALL-UNNAMED

jshell>
```

Отсутствие контролируемых исключений

Если в Java-программе метод возбуждает контролируемое исключение, то необходимо либо обработать его в блоке `try-catch`, либо добавить фразу `throws` в объявление метода. Поскольку задача `JShell` – предоставить простой и быстрый способ

выполнения фрагментов, обрабатывать контролируемые исключения необязательно. Если во время выполнения фрагмента возникнет контролируемое исключение, то `jshell` напечатает трассу стека и продолжит работу.

```
jshell> FileReader fr = new FileReader("secrets.txt")
| java.io.FileNotFoundException thrown: secrets.txt (The system cannot find the file
specified)
|       at FileInputStream.open0 (Native Method)
|       at FileInputStream.open (FileInputStream.java:196)
|       at FileInputStream.<init> (FileInputStream.java:139)
|       at FileInputStream.<init> (FileInputStream.java:94)
|       at FileReader.<init> (FileReader.java:58)
|       at (#1:1)

jshell>
```

Этот фрагмент возбуждает исключение `FileNotFoundException`, потому что в текущем каталоге нет файла `secrets.txt`. Если бы файл существовал, то мы могли бы создать объект `FileReader`, не прибегая к блоку `try-catch`. Отметим, что если бы этот фрагмент встретился внутри метода, то было бы применено обычно правило Java, и объявление метода не откомпилировалось бы:

```
jshell> void readSecrets() {
   ...> FileReader fr = new FileReader("secrets.txt");
   ...> // Еще какой-то код
   ...> }
|  Ошибка:
|  unreported exception java.io.FileNotFoundException; must be caught or declared to
be thrown
|  FileReader fr = new FileReader("secrets.txt");
|                  ^--------------------------^

jshell>
```

Автозавершение

В `JShell` поддерживается автозавершение: можно ввести часть текста и нажать клавишу **Tab**. Эта возможность доступна при вводе команд и фрагментов. Программа распознает контекст и помогает автоматически завершить команду. Если есть несколько вариантов продолжения, то выводятся все, и вы должны будете выбрать нужную самостоятельно. Если продолжение единственное, то текст будет завершен без вашего вмешательства.

> **Предупреждение.** На момент написания книги клавиши автозавершения, описанные в этом разделе, как раз пересматривались. Было предложено объединить функциональность клавиш **Tab** и **Tab+Shift**, оставив только **Tab**, а функциональность клавиши <fix> (**Alt+Enter** или **Alt+F1**) повесить на **Tab+Shift**. О состоянии работ в этом направлении можно узнать по адресу https://bugs.openjdk.java.net/browse/JDK-8177076. Команда `/help shortcuts` выводит текущие назначения клавиш автозавершения.

В примере ниже показано, как работает автозавершение с несколькими вариатами продолжения. Введите `/e` и нажмите **Tab**:

```
jshell> /e
/edit    /exit

jshell> /e
```

Программа поняла, что вы вводите команду, потому что строка начинается знаком `/`. Существуют две команды (`/edit` и `/exit`), начинающиеся с `/e`, они и напечатаны. Теперь вы должны сами завершить команду, введя дополнительные символы. Если введено достаточно символов, чтобы команду можно было определить однозначно, то после нажатия **Enter** команда будет выполнена. В данном примере достаточно ввести `/ed` или `/ex` и нажать **Enter**. Чтобы получить список всех команд, введите `/` и нажмите **Tab**:

```
jshell> /
/!          /?          /drop       /edit       /env        /exit       /help
/history    /imports    /list       /methods    /open       /reload     /reset
/save       /set        /types      /vars
```

В следующем фрагменте создается переменная `str` типа `String` с начальным значением `"GoodBye"`:

```
jshell> String str = "GoodBye"
str ==> "GoodBye"
```

В том же сеансе `jshell` введите `str.` и нажмите **Tab**:

```
jshell> str.
charAt(                 chars()                 codePointAt(
codePointBefore(        codePointCount(         codePoints()
compareTo(              compareToIgnoreCase(    concat(
contains(               contentEquals(          endsWith(
equals(                 equalsIgnoreCase(       getBytes(
getChars(               getClass()              hashCode()
indexOf(                intern()                isEmpty()
lastIndexOf(            length()                matches(
notify()                notifyAll()             offsetByCodePoints(
regionMatches(          replace(                replaceAll(
replaceFirst(           split(                  startsWith(
subSequence(            substring(              toCharArray()
toLowerCase(            toString()              toUpperCase(
trim()                  wait(
```

Здесь напечатаны имена всех методов класса `String`, которые можно вызвать для переменной `str`. Обратите внимание, что одни имена заканчиваются двумя скобками `()`, тогда как другие – только одной. Это не ошибка. Если у метода нет аргументов, то его имя заканчивается двумя скобками, а если есть – то только одной.

Продолжая тот же пример, введите `str.sub` и нажмите **Tab**:

```
jshell> str.sub
subSequence( substring(
```

Теперь программа нашла два метода, начинающиеся с `sub`. Вы можете ввести выражение целиком, `str.substring(0, 4)`, и нажать **Enter** для его вычисления:

```
jshell> str.substring(0, 4)
$2 ==> "Good"
```

А можно вместо этого ввести `str.subs` и нажать **Tab** – тогда программа завершит имя метода, добавит `(` и будет ждать ввода аргументов:

```
jshell> str.substring(
substring(

jshell> str.substring(
```

Теперь введите аргумент метода и нажмите **Enter**, чтобы вычислить выражение:

```
jshell> str.substring(0, 4)
$3 ==> "Good"

jshell>
```

Если метод принимает аргументы, то, наверное, вы захотите увидеть их типы. Если после ввода имени метода или конструктора и открывающей скобки нажать **Shift+Tab**, то программа выведет сигнатуры всех перегруженных вариантов. В примере выше нажатие **Shift+Tab** после ввода `str.substring(` приводит к выводу следующей информации:

```
jshell> str.substring(
String String.substring(int beginIndex)
String String.substring(int beginIndex, int endIndex)
<чтобы увидеть javadoc, нажмите shift-tab еще раз>
```

Если еще раз нажать **Shift+Tab**, то будет показана документация по методу `substring()`. Ниже показано, что произойдет в нашем примере. Чтобы вывести продолжение документации, нажмите пробел, а чтобы вернуться к приглашению `jshell`, нажмите **Q**:

```
jshell> str.substring(
String String.substring(int beginIndex)
Returns a string that is a substring of this string.The substring begins with
the character at the specified index and extends to the end of this string.
Examples:
    "unhappy".substring(2) returns "happy"
    "Harbison".substring(3) returns "bison"
    "emptiness".substring(9) returns "" (an empty string)

Parameters:
```

```
beginIndex - the beginning index, inclusive.

Returns:
the specified substring.
String String.substring(int beginIndex, int endIndex)
Returns a string that is a substring of this string.The substring begins at the
specified beginIndex and extends to the character at index endIndex - 1 . Thus
the length of the substring is endIndex-beginIndex .
Examples:
    "hamburger".substring(4, 8) returns "urge"
    "smiles".substring(1, 5) returns "mile"

Parameters:
beginIndex - the beginning index, inclusive.
endIndex - the ending index, exclusive.

Returns:
the specified substring.

jshell> str.substring(
```

Иногда мы хотим присвоить значение выражения переменной подходящего типа. Бывает, что тип известен, а бывает, что и нет. `JShell` помогает завершить присваивание после ввода всего выражения. Введите выражение целиком и нажмите **Alt+Enter** или **Alt+F1** в зависимости от платформы. Затем нажмите **Alt+V**, чтобы завершить присваивание – программа добавит правильный тип переменной и установит курсор в позицию, где вы сможете ввести ее имя. Проработаем на примере. Введите выражение `2 + 2`:

```
jshell> 2 + 2
```

Если вы работаете в Windows, нажмите **Alt+Enter** и затем **Alt+V**. `jshell` автоматически завершит выражение присваивания и будет ждать, пока вы введете имя переменной:

```
jshell> int  = 2 + 2
```

Курсор находится перед знаком `=`. Введите имя переменной, `x`, и нажмите **Enter**:

```
jshell> int x = 2 + 2
x ==> 4

jshell>
```

При выполнении эти команд в Windows окно консоли разворачивается на весь экран. Чтобы вернуться в обычный режим, нажмите **Alt+Enter**.

История фрагментов и команд

JShell хранит историю команд и фрагментов, введенных во всех сеансах. По ней можно перемещаться вперед и назад клавишами со стрелками вверх и

вниз. Кроме того, команда `/history` печатает историю действий в текущем сеансе:

```
jshell> 2 + 2
$1 ==> 4

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

jshell> /history
2 + 2
System.out.println("Hello")
/history

jshell>
```

Если в этот момент нажать клавишу со стрелкой вверх один раз, то будет показана команда `/history`, два раза – `System.out.println("Hello")`, три раза – `2 + 2`. Четвертое нажатие той же клавиши покажет последнюю команду или фрагмент, введенный в предыдущем сеансе `jshell`. Чтобы выполнить ранее введенную команду или фрагмент, найдите его с помощью клавиши со стрелкой вверх, а затем нажмите **Enter**. Клавиша со стрелкой вниз переходит к следующей команде или фрагменту. Если вы дошли до первой (последней) команды/фрагмента в списке, то нажатие клавиши со стрелкой вверх (вниз) игнорируется.

Чтение трассы стека в JShell

Фрагменты, введенные в `jshell`, – часть синтетического класса. Java не позволяет объявлять метод на верхнем уровне программы. Объявление метода должно быть частью какого-то типа. Когда в программе возникает исключение, в трассе стека печатаются имена типов и номера строк. В `jshell` исключение может быть возбуждено внутри фрагмента. В таком случае имя синтетического класса и номера строки в нем ничего не скажут разработчику. Местоположение кода внутри фрагмента указывается в трассе стека в следующем формате:

```
at <snippet-name> (#<snippet-id>:<line-number-in-snippet>)
```

Отметим, что фрагмент может и не иметь имени. Так, у фрагмента `2 + 2` имени нет. Но фрагменту, в котором объявляется переменная, присваивается такое же имя, как у самой переменной, и то же самое относится к объявлению метода и типа. Иногда у нескольких фрагментов оказывается одинаковое имя, например, если объявлены одноименные переменная и метод. `jshell` присваивает всем фрагментам уникальные идентификаторы. Узнать идентификатор фрагмента поможет команда `/list -all`.

В следующем сеансе `jshell` объявлен метод `divide()` и напечатана трасса стека исключения `ArithmeticException`, возникшего в результат деления целого числа на нуль:

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
```

```
|  Для получения вводной справки введите /help intro

jshell> int divide(int x, int y) {
   ...> return x/y;
   ...> }
|  создан метод divide(int,int)

jshell> divide(10, 2)
$2 ==> 5

jshell> divide(10, 0)
|  java.lang.ArithmeticException thrown: / by zero
|        at divide (#1:2)
|        at (#3:1)

jshell> /list -all
  s1 : import java.io.*;
  s2 : import java.math.*;
  s3 : import java.net.*;
  s4 : import java.nio.file.*;
  s5 : import java.util.*;
  s6 : import java.util.concurrent.*;
  s7 : import java.util.function.*;
  s8 : import java.util.prefs.*;
  s9 : import java.util.regex.*;
 s10 : import java.util.stream.*;
   1 : int divide(int x, int y) {
       return x/y;
       }
   2 : divide(10, 2)
   3 : divide(10, 0)

jshell>
```

Попробуем прочитать эту трассу стека. В последней строке, `at (#3:1)`, говорится, что исключение возникло в строке 1 фрагмент номер 3. В списке, напечатанном командой `/list -all`, под этим номером значится выражение `divide(10, 0)`, которое и стало причиной исключения. Вторая строка, `at divide (#1:2)`, говорит, что на следующем уровне стека находится строка 2 фрагмента с именем `divide` и идентификатором 1.

Повторное использование сеансов JShell

В сеансе `jshell` могло быть введено много фрагментов и команд, которые вы, возможно, захотите снова использовать в других сеансах. Команда `/save` сохраняет команды и фрагменты в файле, а команда `/open` загружает ранее сохраненные команды и фрагменты. Синтаксически команда `/save` выглядит так:

```
/save <option> <file-path>
```

Здесь `<option>` может принимать значения `-all`, `-history` и `-start`, а `<file-path>` – путь к файлу, в котором будут сохранены фрагменты и команды.

Команда `/save` без параметров сохраняет все активные фрагменты в текущем сеансе. Команды и ошибочные фрагменты не сохраняются.

Команда `/save` с параметром `-all` сохраняет все фрагменты в текущем сеансе в указанном файле, включая ошибочные и стартовые. Команды не сохраняются.

Команда `/save` с параметром `-history` сохраняет все, что было введено в текущем сеансе с момента его начала.

Команда `/save` с параметром `-start` сохраняет стартовые определения в указанном файле.

Загрузить фрагменты из файла поможет команда `/open`, принимающая имя файла в качестве параметра.

В показанном ниже сеансе `jshell` объявляется класс `Counter`, создается объект этого класса и вызываются его методы. Затем все активные фрагменты сохраняются в файле `jshell.jsh`. Для файлов `jshell` рекомендуется расширение `.jsh`, но вы можете предпочесть любое другое.

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> class Counter {
   ...>   private int count;
   ...>   public synchronized int next() {
   ...>     return ++count;
   ...>   }
   ...>   public int current() {
   ...>     return count;
   ...>   }
   ...> }
|  создан класс Counter

jshell> Counter counter = new Counter()
counter ==> Counter@25bbe1b6

jshell> counter.current()
$3 ==> 0

jshell> counter.next()
$4 ==> 1

jshell> counter.next()
$5 ==> 2

jshell> counter.current()
```

```
$6 ==> 2

jshell> /save jshell.jsh

jshell> /exit
|  Goodbye
```

В этот момент в текущем каталоге должен появиться файл `jshell.jsh`, содержимое которого приведено в листинге 11.2.

Листинг 11.2. Содержимое файла jshell.jsh

```
class Counter {
    private int count;
    public synchronized int next() {
      return ++count;
    }
    public int current() {
      return count;
    }
}
Counter counter = new Counter();
counter.current()
counter.next()
counter.next()
counter.current()
```

В следующем сеансе `jshell` открывается файл `jshell.jsh` и заново выполняются все фрагменты, сохраненные в предыдущем сеансе. Открыв файл, вы можете вызывать методы переменной `counter`.

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> /open jshell.jsh

jshell> counter.current()
$7 ==> 2

jshell> counter.next()
$8 ==> 3

jshell>
```

Сброс состояния JShell

Для сброса состояния `JShell` предназначена команда `/reset`. При ее выполнении происходит следующее:

- все фрагменты, введенные в текущем сеансе, забываются, так что будьте осторожны;
- стартовые фрагменты выполняются заново;
- состояние выполнения заново инициализируется;
- настройки `jshell`, установленные командой `/set`, сохраняются;
- среда выполнения, установленная командой `/env`, сохраняется.

В показанном ниже сеансе объявляется переменная, затем сеанс сбрасывается и производится попытка напечатать значение переменной. Но, поскольку после сброса все переменные теряются, поиск ранее объявленной переменной завершается неудачно:

```
jshell> int x = 987
x ==> 987

jshell> /reset
|  Состояние сбрасывается.

jshell> x
|  Ошибка:
|  символ не найден
|    символ: variable x
|  x
|  ^

jshell>
```

Перезагрузка состояния JShell

Допустим, вы ввели много фрагментов, после чего вышли из сеанса. А теперь хотите вернуться и выполнить эти фрагменты заново. Можно, конечно, начать новый сеанс `jshell` и повторно ввести все фрагменты. Но это долго и скучно. Есть более простой способ – воспользоваться командой `/reload`. Она сбрасывает состояние `jshell` и заново выполняет все правильные фрагменты и команды `/drop` в той последовательности, в которой они вводились. Для настройки ее поведения имеются параметры `-restore` и `-quiet`.

Без параметров команда `/reload` сбрасывает состояние `jshell` и воспроизводит историю (исключая ошибочные фрагменты), начиная с того из следующих действий (событий), которое произошло последним:

- начало нового сеанса;
- выполнение последней команды `/reset`;
- выполнение последней команды `/reload`.

Если в команде `/reload` был задан параметр `-restore`, то состояние сбрасывается и воспроизводится история между двумя последними из следующих действий (событий):

- запуск `jshell`;
- выполнение команды `/reset`;

- выполнение команды /reload.

Разберемся, что происходит в результате команды /reload с параметром -restore. Ее основная задача – восстановить предыдущее состояние выполнения. Если выполнять ее в начале каждого сеанса jshell, начиная со второго, то сеанс будет содержать все фрагменты, когда-либо введенные в сеансах jshell! Очень полезная штука! Получается, что можно выполнить фрагменты, закрыть jshell, снова открыть, выполнить команду /reload -restore – и ни один из ранее введенных фрагментов никогда не потеряется! Иногда мы выполняем команду /reset дважды в одном сеансе и хотим восстановить состояние между сбросами. Нужного результата можно достичь с помощью этой команды.

В каждом из следующих сеансов jshell создается переменная и с помощью команды /reload -restore восстанавливается предыдущий сеанс. Как видим, в четвертом сеансе используется переменная x1, объявленная в первом.

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> int x1 = 10
x1 ==> 10

jshell> /exit
|  До свидания

C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> /reload -restore
|  Рестарт и восстановление предыдущего состояния
-: int x1 = 10;

jshell> int x2 = 20
x2 ==> 20

jshell> /exit
|  До свидания

C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> /reload -restore
|  Рестарт и восстановление предыдущего состояния
-: int x1 = 10;
-: int x2 = 20;

jshell> int x3 = 30
```

```
x3 ==> 30

jshell> /exit
|  До свидания

C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> /reload -restore
|  Рестарт и восстановление предыдущего состояния
-: int x1 = 10;
-: int x2 = 20;
-: int x3 = 30;

jshell> System.out.println("x1 is " + x1)
x1 is 10

jshell>
```

Команда `/reload` отображает воспроизводимую историю. Чтобы подавить отображение, задайте параметр `-quiet`. Этот параметр можно использовать вместе с `-restore` или отдельно. Он не подавляет вывод сообщений об ошибках, если таковые возникнут в процессе воспроизведения истории. Ниже приведены два сеанса `jshell`: в первом объявляется переменная `x1`, а во втором выполняется команда `/reload` с параметром `-quiet`. Обратите внимание, что теперь информация о перезагрузке переменной `x1` во втором сеансе не отображается.

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> int x1 = 10
x1 ==> 10

jshell> /exit
|  До свидания

C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> /reload -restore -quiet
|  Рестарт и восстановление предыдущего состояния

jshell> x1
x1 ==> 10

jshell>
```

Конфигурирование JShell

Команда `/set` позволяет конфигурировать сеанс `jshell` разными способами: от задания стартовых фрагментов до задания зависящего от платформы редактора фрагментов.

Задание редактора фрагментов

В состав `JShell` входит редактор фрагментов по умолчанию. Для редактирования всех или одного фрагмента служит команда `/edit`, которая открывает фрагмент в редакторе. Редактор фрагментов зависит от платформы, например, в Windows это может быть `notepad.exe`. Чтобы установить или отменить установку редактора, воспользуйтесь командой `/set editor`, имеющей следующие варианты:

- `/set editor [-retain] [-wait] <command>`
- `/set editor [-retain] -default`
- `/set editor [-retain] -delete`

При наличии параметра `-retain` новое значение параметра сохраняется между сеансами `jshell`.

Указанная команда `<command>` зависит от платформы: в Windows одна, в UNIX другая и т. д. Команда может содержать флаги. `JShell` сохраняет подлежащие редактированию фрагменты во временном файле и дописывает его имя в конец командной строки. Пока редактор открыт, работать с `jshell` нельзя. Если редактор сразу завершает процесс, то следует задать параметр `-wait`, который заставит `jshell` дожидаться закрытия редактора. Следующая команда задает Блокнот в качестве редактора в Windows:

```
jshell> /set editor -retain notepad.exe
```

Если задан параметр `-default`, то заданный редактор становится редактором фрагментов по умолчанию. Параметр `-delete` означает, что текущую установку редактора следует отменить. При задании `-delete` совместно с `-retain` стирается запомненная установка редактора:

```
jshell> /set editor -retain -delete
|  Установлен редактор: -default

jshell>
```

Редактор, заданный в одной из переменных среды `JSHELLEDITOR`, `VISUAL` или `EDITOR`, имеет приоритет над редактором по умолчанию. Эти переменные просматриваются в указанном порядке. Если ни одна из них не установлена, то используется редактор по умолчанию. Команда `/set editor` без параметров печатает сведения о текущем установленном редакторе.

В следующем сеансе в качестве редактора устанавливается Блокнот в Windows. На других платформах этот пример работать не будет.

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
```

```
|  Для получения вводной справки введите /help intro

jshell> /set editor
|  /set editor -default

jshell> /set editor -retain notepad.exe
|  Установлен редактор: notepad.exe
|  Сохраненная установка редактора: notepad.exe

jshell> /exit
|  До свидания

C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> /set editor
| /set editor -retain notepad.exe

jshell> 2 + 2
$1 ==> 4

jshell> /edit

jshell> /set editor -retain -delete
|  Установлен редактор: -default

jshell> /exit
|  До свидания

C:\Java9Revealed>SET JSHELLEDITOR=notepad.exe

C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> /set editor
| /set editor notepad.exe

jshell>
```

Задание режима выдачи

После выполнения фрагмента или команд `jshell` что-то печатает в ответ. Объем и формат печатаемой информации называется *режимом выдачи*. Можно задать один из четырех предопределенных режимов или определить свой собствен-ный:

- silent
- concise
- normal
- verbose

В режиме silent не выдается вообще ничего, а в режиме verbose выдается максимально подробная информация. В режиме concise выдается та же информация, что в режиме normal, но в компактном виде. Режим выдачи устанавливается такой командой:

```
/set feedback [-retain] <mode>
```

Здесь <mode> – один из четырех предопределенных режимов. Если задан параметр -retain, то установленный режим будет сохраняться между сеансами.

Можно также запустить jshell в определенном режиме выдачи:

```
jshell --feedback <mode>
```

Следующая команда запускает jshell в режиме verbose:

```
C:\Java9Revealed>jshell --feedback verbose
```

Ниже показано, как устанавливать различные режимы выдачи:

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> 2 + 2
$1 ==> 4

jshell> /set feedback verbose
|  Режим выдачи: verbose

jshell> 2 + 2
$2 ==> 4
|  создана временная переменная $2 : int

jshell> /set feedback concise

jshell> 2 + 2
$3 ==> 4

jshell> /set feedback silent
-> 2 + 2
-> System.out.println("Hello")
Hello

-> /set feedback verbose
|  Режим выдачи: verbose

jshell> 2 + 2
```

```
$6 ==> 4
| создана временная переменная $6 : int
```

Установленный режим выдачи действует только в текущем сеансе `jshell`. Если вы хотите, чтобы он сохранялся между сеансами, выполните команду `/set feedback` с параметром `-retain`:

```
jshell> /set feedback -retain
```

Теперь при повторном входе в `jshell` будет установлен режим, действовавший перед выполнением этой команды. Но, как и раньше, можно временно сменить режим в текущем сеансе. А чтобы сделать новый режим постоянным, нужно выполнить команду `/set feedback <mode>`, а затем еще раз `/set feedback -retain`.

Есть также возможность сразу установить новый режим и сделать его постоянным, например:

```
jshell> /set feedback -retain verbose
```

Чтобы узнать, какой режим действует сейчас, выполните команду `/set feedback`. Она напечатает текущий режим выдачи и список всех возможных режимов:

```
jshell> /set feedback
|  /set feedback normal
|
|  Допустимые режимы выдачи:
|    concise
|    normal
|    silent
|    verbose

jshell>
```

Совет. При изучении `jshell` рекомендуется запускать ее в режиме `verbose`, чтобы выдавалась максимально подробная информация о выполнении команд и фрагментов. Так вы освоите программу быстрее.

Создание пользовательских режимов выдачи

Четырех предопределенных режимов выдачи обычно достаточно. Они дают различные уровни подробности выдаваемой информации. Можно определить и пользовательский режим выдачи, хотя я сомневаюсь, что вам это когда-нибудь понадобится. Создание пользовательского режима – не вполне тривиальное дело. Нужно проделать несколько шагов. Пользовательский режим можно создать с нуля или скопировать существующий режим и изменить несколько настроек в нем. Скорее всего, вы захотите пойти по второму пути. Синтаксически команда создания пользовательского режима выдачи выглядит так:

```
/set mode <mode> [<old-mode>] [-command|-quiet|-delete]
```

Здесь `<mode>` – имя пользовательского режима выдачи, например, `kverbose`, а `<old-mode>` – имя существующего режима, настройки которого копируются в новый.

Если задан параметр -command, то выводятся сведения об установленном режиме, а если -quiet, то ничего не выводится. Параметр -delete позволяет удалить режим.

Следующая команда создает режим выдачи kverbose, копируя все настройки предопределенного режима verbose:

```
/set mode kverbose verbose -command
```

Следующая команда сохраняет режим kverbose для использования в будущих сеансах:

```
/set mode kverbose -retain
```

Чтобы удалить пользовательский режим выдачи, задайте параметр -delete. Удалить предопределенный режим невозможно. Если вы сохранили пользовательский режим, то, добавив параметр -retain, сможете удалить его как из текущего, так и из будущих сеансов. Следующая команда удаляет режим kverbose:

```
/set mode kverbose -delete -retain
```

Пока что между предопределенным режимом verbose и пользовательским режимом kverbose нет никакой разницы. После создания режима выдачи необходимо настроить три аспекта:

- приглашения;
- порог отсечения вывода;
- формат вывода.

Совет. Определив пользовательский режим выдачи, вы должны выполнить команду /set feedback <new-mode>, чтобы начать им пользоваться.

Можно настроить два приглашения: основное и продолжающее. Основное приглашение отображается, когда jshell готова читать новую команду или фрагмент, а продолжающее – в начале строки при вводе многострочного фрагмента. Приглашения задаются следующим образом:

```
/set prompt <mode> "<prompt>" "<continuation-prompt>"
```

Здесь <prompt> — основное приглашение, а <continuation-prompt> – продолжающее. Следующая команда задает приглашения для режима kverbose:

```
/set prompt kverbose "\njshell-kverbose> " "more... "
```

Можно настроить максимальное число знаков, отображаемых для каждого типа действий или событий:

```
/set truncation <mode> <length> <selectors>
```

Здесь <mode> – режим выдачи, <length> – максимальное число знаков, отображаемых для указанных селекторов, а <selectors> – список селекторов (через запятую), определяющих контекст, к которому применим данный порог отсечения. Селекторы – это предопределенные ключевые слова, представляющие контексты. Например, селектор vardecl соответствует объявлению переменной без инициализа-

ции. Для получения дополнительной информации о задании порогов отсечения и о селекторах выполните команду

```
/help /set truncation
```

Следующие команды задают порог отсечения 80 знаков для всего и 5 знаков для значения переменной или выражения:

```
/set truncation kverbose 80
/set truncation kverbose 5 expression,varvalue
```

Применимый порог отсечения определяется самым специфичным селектором. Выше заданы пороги для двух случаев: фрагменты всех типов (80 знаков) и значения переменных и выражений (5 знаков). В случае выражения наиболее специфичным является селектор `expression`, поэтому если значение выражения длиннее пяти знаков, то будут показаны только первые 5 знаков.

Задание формата вывода – дело непростое. Формат необходимо задать для всех ожидаемых типов вывода. Я не стану подробно описывать типы форматов вывода. Интересующиеся читатели могут выполнить команду

```
/help /set format
```

Формат вывода задается следующим образом:

```
/set format <mode> <field> "<format>" <selectors>
```

Здесь `<mode>` – имя режима выдачи, для которого задается формат вывода, `<field>` – имя контекстного поля, а `"<format>"` – формат, используемый для отображения выводимой информации. Строка `<format>` может содержать имена предопределенных полей в фигурных скобках, например: `{name}`, `{type}`, `{value}` и т. д., вместо которых будут подставлены значения, зависящие от контекста. `<selectors>` – селекторы, определяющие контекст, в котором применим этот формат.

Следующая команда задает формат отображения в случае добавления, модификации или замены выражения во введенном фрагменте. Вся команда должна записываться в одной строке.

```
/set format kverbose display "{result}{pre}created a temporary variable named {name} of type
{type} and initialized it with {value}{post}" expression-added,modified,replaced-primary
```

В следующем сеансе `jshell` создается новый формат выдачи `kverbose` путем копирования всех настроек предопределенного режима `verbose`. Затем настраиваются приглашения, пороги отсечения и форматы вывода. Сравнивается поведение `jshell` в режимах `verbose` и `kverbose`. Все команды следует вводить в одной строке, даже если в книге они занимают несколько строк.

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> /set feedback
```

```
|  /set feedback -retain normal
|
|  Доступные режимы выдачи:
|     concise
|     normal
|     silent
|     verbose

jshell> /set mode kverbose verbose -command
|  Создан новый режим выдачи: kverbose

jshell> /set mode kverbose -retain

jshell> /set prompt kverbose "\njshell-kverbose> " "more... "

jshell> /set truncation kverbose 5 expression,varvalue

jshell> /set format kverbose display "{result}{pre}created a temporary variable named {name}
of type {type} and initialized it with {value}{post}" expression-added,modified,replacedprimary

jshell> /set feedback kverbose
|  Режим выдачи: kverbose

jshell-kverbose> 2 +
more... 2
$2 ==> 4
|   created a temporary variable named $2 of type int and initialized it with 4

jshell-kverbose> 111111 + 222222
$3 ==> 33333
| created a temporary variable named $3 of type int and initialized it with 33333

jshell-kverbose> /set feedback verbose
|  Режим выдачи: verbose
jshell> 2 +
   ...> 2
$4 ==> 4
|  создана временная переменная $4 : int

jshell> 111111 + 222222
$5 ==> 333333
|  создана временная переменная $5 : int

jshell> /exit
|  До свидания

C:\Java9Revealed>jshell
```

```
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> /set feedback
|  /set feedback -retain normal
|
|  Сохраненные режимы выдачи:
|    kverbose
|  Доступные режимы выдачи:
|    concise
|    kverbose
|    normal
|    silent
|    verbose

jshell>
```

В этом сеансе для режима выдачи `kverbose` задан порог отсечения 5 знаков для значений выражений и переменных. Именно поэтому в режиме `kverbose` в качестве значения выражения `111111 + 222222` отображается `33333`, а не `333333`. Это не ошибка, а результат настройки формата.

Обратите внимание, что команда `/set feedback` показывает команду, установившую текущий режим выдачи, а также список доступных режимов, в котором присутствует и созданный нами режим `kverbose`.

При создании пользовательского режима выдачи полезно знать настройки существующих режимов. Для печати всех настроек всех режимов предназначена команда

```
/set mode
```

Можно также напечатать список настроек одного режима выдачи, передав его имя в качестве аргумента. Следующая команда печатает все настройки режима `silent`. А первой строке напечатана команда создания режима `silent`.

```
jshell> /set mode silent
| /set mode silent -quiet
| /set prompt silent "-> " ">> "
| /set format silent display ""
| /set format silent err "%6$s"
| /set format silent errorline "    {err}%n"
| /set format silent errorpost "%n"
| /set format silent errorpre "| "
| /set format silent errors "%5$s"
| /set format silent name "%1$s"
| /set format silent post "%n"
| /set format silent pre "|  "
| /set format silent type "%2$s"
```

```
| /set format silent unresolved "%4$s"
| /set format silent value "%3$s"
| /set truncation silent 80
| /set truncation silent 1000 expression,varvalue

jshell>
```

Задание стартовых фрагментов

Команда `/set` с аргументом `start` задает стартовые фрагменты и команды, автоматически выполняемые сразу после запуска `jshell`. Мы уже встречались со стартовыми фрагментами, в которых импортируются типы из нескольких часто используемых пакетов. Как правило, в стартовый скрипт помещаются команды `/env` для задания путей к классам и модулям и предложения `import`.

Список стартовых фрагментов по умолчанию печатает команда `/list -start`. Отметим, что она выводит именно фрагменты по умолчанию, а не текущие стартовые фрагменты. Напомним, что стартовые фрагменты можно удалять. Стартовые фрагменты по умолчанию – это то, что вы получаете в момент запуска `jshell`. А текущие стартовые фрагменты – это стартовые фрагменты по умолчанию за вычетом удаленных в текущем сеансе.

Для задания стартовых фрагментов и команд применяются следующие варианты команды `/set`:

- `/set start [-retain] <file>`
- `/set start [-retain] -default`
- `/set start [-retain] -none`

Параметр `-retain` факультативный. Если он задан, то настройка сохраняется между сеансами `jshell`.

Первая форма используется для чтения стартовых фрагментов и команд из файла. Когда в текущем сеансе выполняется команда `/reset` или `/reload`, читается и выполняется содержимое этого файла. Если стартовый код читается из файла, то `jshell` кэширует его содержимое для использования в будущем. Модификация файла не оказывает влияния на стартовый код до тех пор, пока команда `set` не будет выполнена снова.

Вторая форма позволяет восстановить встроенные стартовые фрагменты и команды.

Третья форма стирает весь стартовый код, т. е. после запуска не будут выполняться никакие фрагменты и команды.

Команда `/set start` без параметров и файла выводит текущие стартовые настройки. Если стартовый код читается из файла, то будет показано имя файла, стартовые фрагменты и время установки этих фрагментов.

Рассмотрим следующий сценарий. В каталоге `com.jdojo.jshell` в прилагаемом к книге исходном коде имеется класс `com.jdojo.jshell.Person`. Протестируем его в `jshell` и будем использовать типы из пакета `java.time`. Для этого стартовые настройки нужно задать, как показано в листинге 11.3.

Листинг 11.3. Содержимое файла startup.jsh

```
/env -class-path C:\Java9Revealed\com.jdojo.jshell\build\classes
import java.io.*
import java.math.*
import java.net.*
import java.nio.file.*
import java.util.*
import java.util.concurrent.*
import java.util.function.*
import java.util.prefs.*
import java.util.regex.*
import java.util.stream.*
import java.time.*;
import com.jdojo.jshell.*;
void printf(String format, Object... args) { System.out.printf(format, args); }
```

Сохраните эти настройки в файле `startup.jsh` в текущем каталоге. Если вы сохраните их в каком-то другом каталоге, то перед выполнением примера нужно будет задать абсолютный путь к файлу. Отметим, что в самом начале находится команда `/env -class-path` для Windows, предполагающая, что исходный код сохранен в каталоге `C:\`. Измените путь к классам в соответствии со своей платформой и местом, куда вы поместили прилагаемый к книге код.

Обратите внимание на последний фрагмент в файле `startup.jsh file`. В нем определена функция верхнего уровня `printf()`, обертывающая метод `System.out.printf()`. В первых сборках `JShell` эта функция включалась по умолчанию, но впоследствии была удалена. Если вы хотите использовать короткое имя метода, например `printf()` вместо `System.out.printf()` для печати сообщений на стандартный вывод, то можете включить этот фрагмент в свой стартовый скрипт. Если вы хотите использовать методы верхнего уровня `println()` и `printf()` по умолчанию, то запустите `jshell` следующим образом:

```
C:\Java9Revealed>jshell --start DEFAULT --start PRINTING
```

Аргумент `DEFAULT` включает все предложения `import`, подразумеваемые по умолчанию, а аргумент `PRINTING` – все варианты `print()`, `println()` и `printf()`. Запустив `jshell` этой командой, выполните команду `/list -start`, чтобы увидеть, какие стартовые предложения `import` и методы были добавлены в результате.

В следующих примерах демонстрируется задание стартовых настроек из файла и их использование в последующих сеансах.

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> /set start
| /set start -default
```

```
jshell> /set start -retain startup.jsh

jshell> Person p;
|  создана переменная p, но на нее нельзя ссылаться, пока не будет объявлен класс Person

jshell> /reset
|  Сбрасывается состояние.

jshell> Person p;
p ==> null
jshell> /exit
|  До свидания

C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> /set start
|  /set start -retain startup.jsh
|  ---- startup.jsh @ Feb 20, 2017, 10:06:47 AM ----
|  /env -class-path C:\Java9Revealed\com.jdojo.jshell\build\classes
|  import java.io.*
|  import java.math.*
|  import java.net.*
|  import java.nio.file.*
|  import java.util.*
|  import java.util.concurrent.*
|  import java.util.function.*
|  import java.util.prefs.*
|  import java.util.regex.*
|  import java.util.stream.*
|  import java.time.*;
|  import com.jdojo.jshell.*;
|  void printf(String format, Object... args) { System.out.printf(format, args); }

jshell> Person p
p ==> null

jshell> LocalDate.now()
$2 ==> 2016-11-15

jshell>

jshell> printf("2 + 2 = %d%n", 2 + 2)
2 + 2 = 4

jshell>
```

Совет. Задание стартовых фрагментов и команд вступает в силу только после перезапуска `jshell` либо выполнения команды `/reset` или `/reload`. Не включайте команды `/reset` и `/reload` в стартовый файл, т. к. это приведен к зацикливанию на стадии его загрузки.

Существует три предопределенных стартовых скрипта:

- `DEFAULT`
- `PRINTING`
- `JAVASE`

Скрипт `DEFAULT` содержит общеупотребительные предложения `import`. В скрипте `PRINTING` определены верхнеуровневые методы `JShell`, обертывающие методы `print()`, `println()` и `printf()` класса `PrintStream`. Скрипт `JAVASE` импортирует все пакеты Java SE, это большой скрипт, он работает несколько секунд. Ниже показано, как использовать эти скрипты в качестве стартовых:

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> println("Hello")
|  Ошибка:
|  символ не найден
|    symbol: method println(java.lang.String)
|  println("Hello")
|  ^-----^

jshell> /set start -retain DEFAULT PRINTING

jshell> /exit
|  До свидания

C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> println("Hello")
Hello

jshell>
```

В первом случае обращение к методу `println()` привело к ошибке. Но после того как мы установили скрипт `PRINTING` в качестве стартового и перезапустили программу, метод заработал.

Использование документации по JShell

В состав программы JShell входит подробная документация. Но поскольку это командный инструмент, читать ее не очень просто. Чтобы вывести список команд с краткими описаниями, воспользуйтесь командой `/help` или `/?`.

```
jshell> /help
|  Введите выражение, предложение или объявление на языке Java.
|  Или введите одну из следующих команд:
|  /list [<имя или ид>|-all|-start] - вывести указанный исходный код
|  /edit <имя или ид> -- редактировать исходный код по имени или ид
|  /drop <имя или ид> -- удалить исходный код по имени или ид
...
```

В качестве аргумента команды `/help` можно указать имя команды – тогда будет напечатана справка по ней. Следующая команда печатает справку по самой команде `/help`:

```
jshell> /help /help
|
|  /help
|
|  Вывод сведений о jshell.
|  /help
|     Выводит список команд jshell и тем справки.
|
|  /help <команда>
|     Выводит сведения об указанной команде. Знак / обязателен.
|     Достаточно ввести несколько первых букв имени команды -- если таких
|     команд несколько, будут выведены все. Пример: /help /li
|
|  /help <тема>
|     Выводит содержимое указанной темы справки. Пример: /help intro
```

Следующие команды выводят сведения о командах `/list` и `/set`. Вывод занимает много места и потому опущен:

```
jshell> /help /list
|...

jshell> /help /set
|...
```

Иногда команда относится к нескольким темам, например, команда `/set` задает режим выдачи, редактор фрагментов, стартовые скрипты и т. д. Если вам нужна информация по конкретной сфере применения команды, воспользуйтесь следующим вариантом команды `/help`:

```
/help /<command> <topic-name>
```

Следующая команда печатает справку о задании режима выдачи:

```
jshell> /help /set feedback
```

А эта команда печатает сведения о создании пользовательского режима выдачи:

```
jshell> /help /set mode
```

Команда `/help` с указанием темы в качестве аргумента выводит справку по этой теме. В настоящее время определены три темы: `intro`, `shortcuts` и `context`. Следующая команда выводит введение в работу с программой `JShell`:

```
jshell> /help intro
```

Следующая команда выводит список всех комбинаций клавиш с описаниями:

```
jshell> /help shortcuts
```

Следующая команда выводит список параметров установки контекста выполнения. Эти параметры используются в командах `/env`, `/reset` и `/reload`.

```
jshell> /help context
```

JShell API

JShell API дает доступ к движку выполнения фрагментов из программы. Обычному разработчику этот API вряд ли пригодится. Он предназначен для таких инструментов, как NetBeans IDE, в которые может быть включен пользовательский интерфейс, эквивалентный командной утилите `JShell`, чтобы разработчик мог выполнять фрагменты кода, не выходя из IDE. В этом разделе я кратко опишу JShell API и продемонстрирую его на простом примере.

JShell API находится в пакете `jdk.jshell` из модуля `jdk.jshell`. Если вы собираетесь использовать JShell API, то ваш модуль должен читать модуль `jdk.jshell`. JShell API состоит в основном из трех абстрактных классов и одного интерфейса:

- `JShell`
- `Snippet`
- `SnippetEvent`
- `SourceCodeAnalysis`

Экземпляр класса `JShell` представляет движок выполнения фрагментов. Это основной класс JShell API. В нем хранится состояние всех выполняемых фрагментов.

Фрагмент представляется экземпляром класса `Snippet`. Объект `JShell` генерирует события фрагментов в процессе их выполнения.

Событие фрагмента представляется экземпляром интерфейса `SnippetEvent`. Событие содержит текущее и предыдущее состояние фрагмента, значение для тех фрагментов, которые вычисляют значение, исходный код фрагмента, приведший к возникновению события, объект `Exception`, если при выполнении фрагмента возникло исключение, и т. д.

Экземпляр класса `SourceCodeAnalysis` предоставляет средства анализа кода и рекомендации для фрагмента. Он отвечает на вопросы вида:

- Является ли фрагмент полным?
- Можно ли сделать фрагмент полным, добавив в конец точку с запятой?

Объект `SourceCodeAnalysis` предлагает также список рекомендаций, например, для автозавершения по нажатию **Tab** и для доступа к документации. Этот класс

предназначен для инструментальных средств, предоставляющих ту же функциональность, что `JShell`. Больше я к нему возвращаться не буду. Интересующиеся читатели могут ознакомиться с документацией по этому классу.

На рис. 11.5 показана диаграмма прецедентов для различных компонентов JShell API. В следующих разделах я подробнее расскажу об этих классах и их использовании, а в последнем разделе приведу пример.

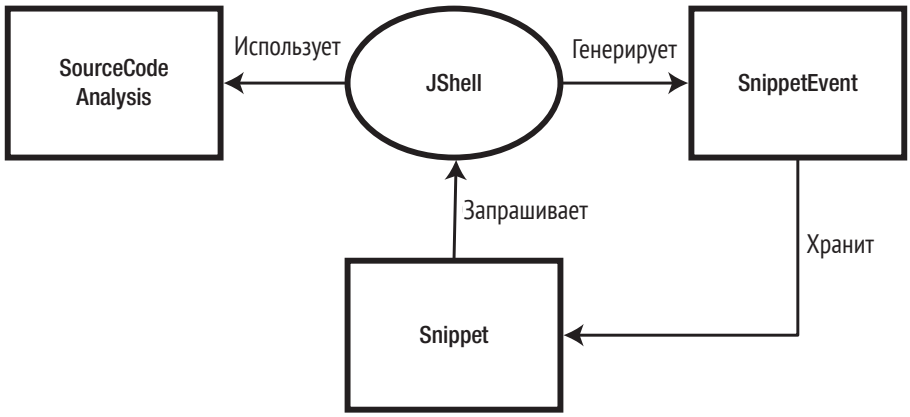

Рис. 11.5. Диаграмма прецедентов для компонентов JShell API

Создание объекта JShell

Класс `JShell` абстрактный. Создать его экземпляры можно двумя способами:

- с помощью статического метода `create()`;
- с помощью статического построителя `JShell.Builder`.

Метод `create()` возвращает предварительно сконфигурированный экземпляр класса `JShell`:

```
// Создать экземпляр JShell
JShell shell = JShell.create()
```

Класс `JShell.Builder` позволяет сконфигурировать экземпляр `JShell`, задав генератор идентификаторов фрагментов, генератор имен временных переменных, поток `PrintStream` для вывода, поток ввода для чтения фрагментов и поток для вывода ошибок. Экземпляр класса `JShell.Builder` возвращается статическим методом `builder()` класса `JShell`. Ниже показано, как с помощью `JShell.Builder` создать экземпляр `JShell`, при этом `myXXXStream` – ссылки на объекты потоков:

```
// Create a JShell instance
JShell shell = JShell.builder()
                     .in(myInputStream)
                     .out(myOutputStream)
                     .err(myErrorStream)
                     .build();
```

Имея экземпляр `JShell`, можно приступить к выполнению фрагментов с помощью его метода `eval(String snippet)`. Для удаления фрагмента служит ме-

тод `drop(PersistentSnippet snippet)`, для добавления пути к классам – метод `addToClasspath(String path)`. Каждый из этих методов изменяет состояние объекта `JShell`.

> **Совет.** По завершении работы с объектом `JShell` необходимо вызвать его метод `close()`, чтобы освободить ресурсы. Класс `JShell` реализует интерфейс `AutoCloseable`, поэтому лучший способ гарантировать закрытие после использования – погрузить создание объекта в блок `try` с ресурсами. Объект `JShell` изменяемый и не является потокобезопасным.

Методы `onSnippetEvent(Consumer<SnippetEvent> listener)` и `onShutdown(Consumer<JShell> listener)` класса `JShell` позволяют зарегистрировать обработчики событий фрагментов и обработчик события останова `JShell`. Событие фрагмента генерируется при изменении состояния фрагмента: после первого выполнения или из-за обновления в результате выполнения какого-то другого фрагмента.

Метод `sourceCodeAnalysis()` класса `JShell` возвращает экземпляр класса `SourceCodeAnalysis`, оказывающего помощь в анализе кода.

Другие методы класса `JShell` служат для опроса его состояния. Так, методы `snippets()`, `types()`, `methods()` и `variables()` возвращают соответственно список всех фрагментов, список активных фрагментов с объявлением типов, список активных фрагментов с объявлением методов и список активных фрагментов с объявлением переменных.

Метод `eval()` используется чаще всего. Он выполняет (или вычисляет) переданный фрагмент и возвращает список `List<SnippetEvent>`. У событий в списке можно запрашивать состояние выполнения. Ниже приведен простой пример использования метода `eval()`:

```
String snippet = "int x = 100;";

// Выполнить фрагмент
List<SnippetEvent> events = shell.eval(snippet);

// Обработать результаты
events.forEach((SnippetEvent se) -> {
    /* Здесь должен быть код обработки событий фрагмента */
});
```

Работа с фрагментами

Экземпляр класса `Snippet` представляет фрагмент. Класс не дает возможности создавать свои экземпляры. Фрагмент передается объекту `JShell` в виде строки, а экземпляры класса `Snippet` мы получаем в процессе обработки событий фрагмента. В событии фрагмента хранится также предыдущее и текущее состояние фрагмента. Имея объект `Snippet`, мы можем запросить его текущее состояние методом `status(Snippet s)` класса `JShell`, который возвращает объект `Snippet.Status`.

> **Совет.** Класс `Snippet` неизменяемый и потокобезопасный.

Существует несколько типов фрагментов, например: объявление переменной, объявление переменной с инициализацией, объявление метода, объявление типа и т. д. Сам класс `Snippet` абстрактный, но у него есть подклассы для представления каждого типа фрагментов. Ниже показана иерархия наследования этих классов:

- `Snippet`
- `ErroneousSnippet`
- `ExpressionSnippet`
- `StatementSnippet`
- `PersistentSnippet`
  - `ImportSnippet`
  - `DeclarationSnippet`
    - `MethodSnippet`
    - `TypeDeclSnippet`
    - `VarSnippet`

Имена подклассов класса `Snippet` не нуждаются в пояснениях. Так, экземпляр класса `PersistentSnippet` представляет фрагмент, который хранится в `JShell` и допускает повторное использование, например, объявление класса или метода. В классе `Snippet` имеются следующие методы:

- `String id()`
- `String source()`
- `Snippet.Kind kind()`
- `Snippet.SubKind subKind()`

Метод `id()` возвращает уникальный идентификатор фрагмента, а метод `source()` – его исходный код. Методы `kind()` и `subKind()` возвращают тип и подтип фрагмента.

Тип фрагмента определяется элементом перечисления `Snippet.Kind`, например: `IMPORT`, `TYPE_DECL`, `METHOD`, `VAR` и т. д. Подтип несет уточняющую информацию о типе, например, если фрагмент является объявлением типа, то подтип говорит, является ли тип классом, интерфейсом, перечислением или аннотацией. Подтип фрагмента определяется элементом перечисления `Snippet.SubKind`, например: `CLASS_SUBKIND`, `ENUM_SUBKIND` и т. д. В перечислении `Snippet.Kind` имеется свойство `isPersistent`, равное `true`, если фрагмент этого типа сохраняемый, и `false` в противном случае.

Подклассы `Snippet` обладают дополнительными методами для возврата информации о фрагментах конкретных типов. Так, в классе `VarSnippet` имеется метод `typeName()`, возвращающий тип переменной. А в классе `MethodSnippet` есть методы `parameterTypes()` и `signature()`, возвращающие типы параметров и полную сигнатуру метода в виде строки.

Фрагмент ничего не знает о своем состоянии. Объект `JShell` выполняет фрагмент и хранит его состояние. Отметим, что выполнение фрагмента может оказать влияние на состояние других фрагментов. Например, в результате выполнения фрагмента с объявлением переменной состояние фрагмента, в котором объявлен метод, может измениться с правильного на неправильное или наоборот, если этот метод ссылается на переменную. Чтобы узнать текущее состояние фрагмента, вызовите метод `status(Snippet s)` объект `JShell`. Этот метод возвращает элемент перечисления `Snippet.Status`.

- DROPPED: фрагмент неактивен, поскольку был удален методом drop() класса JShell.
- NONEXISTENT: фрагмент неактивен, т. к. еще не существует.
- OVERWRITTEN: фрагмент неактивен, т. к. был заменен другим.
- RECOVERABLE_DEFINED: фрагмент представляет собой объявление, содержащее неразрешенные ссылки. Сигнатура объявления правильна, другие фрагменты видят это объявление. Его можно будет использовать, после того как другие фрагменты переведут его в состояние VALID.
- RECOVERABLE_NOT_DEFINED: фрагмент представляет собой объявление, содержащее неразрешенные ссылки. Сигнатура объявления неправильна, другие фрагменты его не видят. Можно будет использовать впоследствии, когда состояние изменится на VALID.
- REJECTED: фрагмент неактивен, т. к. компиляция завершилась с ошибкой. Никакие будущие изменения состояния JShell не сделают фрагмент правильным.
- VALID: фрагмент правильный в контексте текущего состояния JShell.

Обработка событий фрагмента

JShell генерирует события фрагментов в процессе выполнения или вычисления фрагмента. Для обработки событий следует зарегистрировать обработчики с помощью метода onSnippetEvent() класса JShell или перебрать события из списка List<SnippetEvent>, который возвращает метод eval() класса JShell. В следующем примере показано, как обрабатывать события фрагментов:

```
try (JShell shell = JShell.create()) {
   // Создать фрагмент
   String snippet = "int x = 100;";

   shell.eval(snippet)
        .forEach((SnippetEvent se) -> {
           Snippet s = se.snippet();
           System.out.printf("Фрагмент: %s%n", s.source());
           System.out.printf("Тип: %s%n", s.kind());
           System.out.printf("Подтип: %s%n", s.subKind());
           System.out.printf("Предыдущее состояние: %s%n", se.previousStatus());
           System.out.printf("Текущее состояние: %s%n", se.status());
           System.out.printf("Значение: %s%n", se.value());
   });
}
```

Пример

Рассмотрим JShell API в действии. В листинге 11.4 приведено объявление модуля com.jdojo.jshell.api, а в листинге 11.5 – полный код класса JShellApiTest, являющегося частью этого модуля.

Листинг 11.4. Объявление модуля com.jdojo.jshell.api

```
// module-info.java
   module com.jdojo.jshell.api {
   requires jdk.jshell;
}
```

Листинг 11.5. Класс JShellApiTest, демонстрирующий использование JShell API

```
// JShellApiTest.java
package com.jdojo.jshell.api;

import jdk.jshell.JShell;
import jdk.jshell.Snippet;
import jdk.jshell.SnippetEvent;

public class JShellApiTest {
   public static void main(String[] args) {
      // Создать массив фрагментов, которые будут выполняться последовательно
      String[] snippets = {
           "int x = 100;",
           "double x = 190.89;",
           "long multiply(int value) {return value * multiplier;}",
           "int multiplier = 2;",
           "multiply(200)",
           "mul(99)"
      };

      try (JShell shell = JShell.create()) {
         // Зарегистрировать обработчик события фрагмента
         shell.onSnippetEvent(JShellApiTest::snippetEventHandler);

         // Выполнить все фрагменты
         for(String snippet : snippets) {
            shell.eval(snippet);
            System.out.println("------------------------");
         }
      }
   }

   public static void snippetEventHandler(SnippetEvent se) {
      // Напечатать сведения о событии фрагмента
      Snippet snippet = se.snippet();
      System.out.printf("Фрагмент: %s%n", snippet.source());

      // Напечатать причину события фрагмента
      Snippet causeSnippet = se.causeSnippet();
```

```
      if (causeSnippet != null) {
         System.out.printf("Фрагмент-причина: %s%n", causeSnippet.source());
      }
      System.out.printf("Тип: %s%n", snippet.kind());
      System.out.printf("Подтип: %s%n", snippet.subKind());
      System.out.printf("Предыдущее состояние: %s%n", se.previousStatus());
      System.out.printf("Текущее состояние: %s%n", se.status());
      System.out.printf("Значение: %s%n", se.value());
      Exception e = se.exception();
      if (e != null) {
         System.out.printf("Исключение: %s%n", se.exception().getMessage());
      }
   }
}
```

```
Фрагмент: int x = 100;
Тип: VAR
Подтип: VAR_DECLARATION_WITH_INITIALIZER_SUBKIND
Предыдущее состояние: NONEXISTENT
Текущее состояние: VALID
Значение: 100
-----------------------
Фрагмент: double x = 190.89;
Тип: VAR
Подтип: VAR_DECLARATION_WITH_INITIALIZER_SUBKIND
Предыдущее состояние: VALID
Текущее состояние: VALID
Значение: 190.89
Фрагмент: int x = 100;
Фрагмент-причина: double x = 190.89;
Тип: VAR
Подтип: VAR_DECLARATION_WITH_INITIALIZER_SUBKIND
Предыдущее состояние: VALID
Текущее состояние: OVERWRITTEN
Значение: null
-----------------------
Фрагмент: long multiply(int value) {return value * multiplier;}
Тип: METHOD
Подтип: METHOD_SUBKIND
Предыдущее состояние: NONEXISTENT
Текущее состояние: RECOVERABLE_DEFINED
Значение: null
-----------------------
Фрагмент: int multiplier = 2;
Тип: VAR
```

```
Подтип: VAR_DECLARATION_WITH_INITIALIZER_SUBKIND
Предыдущее состояние: NONEXISTENT
Текущее состояние: VALID
Значение: 2
Фрагмент: long multiply(int value) {return value * multiplier;}
Фрагмент-причина: int multiplier = 2;
Тип: METHOD
Подтип: METHOD_SUBKIND
Предыдущее состояние: RECOVERABLE_DEFINED
Текущее состояние: VALID
Значение: null
-----------------------
Фрагмент: multiply(200)
Тип: VAR
Подтип: TEMP_VAR_EXPRESSION_SUBKIND
Предыдущее состояние: NONEXISTENT
Текущее состояние: VALID
Значение: 400
-----------------------
Фрагмент: mul(99)
Тип: ERRONEOUS
Подтип: UNKNOWN_SUBKIND
Предыдущее состояние: NONEXISTENT
Текущее состояние: REJECTED
Значение: null
-----------------------
```

Метод `main()` создает следующие шесть фрагментов и сохраняет их в массиве строк:

1. `"int x = 100;"`
2. `"double x = 190.89;"`
3. `"long multiply(int value) {return value * multiplier;}"`
4. `"int multiplier = 2;"`
5. `"multiply(200)"`
6. `"mul(99)"`

Экземпляр `JShell` создается в блоке `try` с ресурсами. В качестве обработчика событий фрагментов регистрируется метод `snippetEventHandler()`, который печатает сведения о фрагменте: его исходный код, исходный код фрагмента, ставшего причиной изменения состояния данного фрагмента, предыдущее и текущее состояние фрагмента, его значение и т. д. Наконец, в цикле `for-each` мы обходим все фрагменты и вызываем метод `eval()` для выполнения каждого.

Рассмотрим подробнее состояние движка `JShell` при выполнении каждого фрагмента.

- Когда выполняется фрагмент 1, состояние меняется с NONEXISTENT на VALID, потому что раньше фрагмент не существовал. В этом фрагменте объявляется переменная и результатом его вычисления является значение 100.
- В момент выполнения фрагмента 2 он уже существовал. В нем объявляется та же самая переменная, но с другим типом данных. Предыдущее и текущее состояние – VALID. В результате выполнения этого фрагмента состояние фрагмента 1 меняется с VALID на OVERWRITTEN, потому что двух переменных с одним именем быть не может.
- Во фрагменте 3 объявляется метод multiply(), в теле которого используется необъявленная переменная multiplier, поэтому состояние изменяется с NONEXISTENT на RECOVERABLE_DEFINED. Метод определен, т. е. на него можно ссылаться, но нельзя вызывать до тех пор, пока не будет определена переменная multiplier подходящего типа.
- Во фрагменте 4 определяется переменная multiplier, в результате чего фрагмент 3 становится правильным.
- Во фрагменте 5 вычисляется выражение, в состав которого входит вызов метода multiply(). Выражение правильное и его значение равно 400.
- Во фрагменте 6 вычисляется выражение, в состав которого входит вызов метода mul(), который ранее не был определен. Этот фрагмент ошибочен и отклоняется.

Обычно JShell API и программа JShell вместе не используются. Однако забавы ради попробуем это сделать. JShell API – это всего лишь еще один API в Java, и ничто не мешает воспользоваться им в программе JShell. В следующем сеансе создается экземпляр класса JShell, регистрируется обработчик событий фрагментов и вычисляется два фрагмента.

```
C:\Java9Revealed>jshell
|  Добро пожаловать в JShell -- версия 9-ea
|  Для получения вводной справки введите /help intro

jshell> /set feedback silent
-> import jdk.jshell.*
-> JShell shell = JShell.create()
-> shell.onSnippetEvent(se -> {
>>  System.out.printf("Фрагмент: %s%n", se.snippet().source());
>>  System.out.printf("Предыдущее состояние: %s%n", se.previousStatus());
>>  System.out.printf("Текущее состояние: %s%n", se.status());
>>  System.out.printf("Значение: %s%n", se.value());
>> });
-> shell.eval("int x = 100;");
Фрагмент: int x = 100;
Предыдущее состояние: NONEXISTENT
Текущее состояние: VALID
Значение: 100
-> shell.eval("double x = 100.89;");
Фрагмент: double x = 100.89;
Предыдущее состояние: VALID
```

```
Текущее состояние: VALID
Значение: 100.89
Фрагмент: int x = 100;
Предыдущее состояние: VALID
Текущее состояние: OVERWRITTEN
Значение: null
-> shell.close()
-> /exit

C:\Java9Revealed>
```

Резюме

Оболочка Java, которая в JDK 9 называется JShell, — это командная утилита, позволяющая интерактивно работать с языком Java, т. е. выполнять фрагменты Java-кода без написания законченной программы. Это цикл REPL для Java. Вместе с тем JShell – еще и API, предоставляющий доступ к функциональности REPL для Java-кода другим инструментальным программам, например, IDE.

Функциональность JShell реализована в программе `jshell`, которая находится в каталоге `JDK_HOME\bin directory`. Она поддерживает выполнение фрагментов и команд. Фрагментом называется кусок Java-кода. В процессе выполнения фрагментов `JShell` изменяет свое состояние. Программа также хранит состояние всех введенных фрагментов. Для опроса состояния `JShell` и конфигурирования среды выполнения имеется набор команд. Чтобы отличать команды от фрагментов, все команды начинаются знаком `/`.

`JShell` поддерживает ряд механизмов, позволяющих разработчикам работать более продуктивно и с большим комфортом, например, автозавершение кода и показ документации прямо в программе. `JShell` стремится использовать уже имеющуюся в JDK функциональность, например, API компилятора для разбора, анализа и компиляции фрагментов и API отладчика Java для замены существующего фрагмента новым в JVM. Благодаря дизайну JShell становится возможно вводить новые конструкции в язык Java, не внося никаких или почти никаких изменений в программу `jshell`.

Глава 12

Изменения API процессов

Краткое содержание главы:

- что такое API процессов;
- создание платформенного процесса;
- получение информации о новом процессе;
- получение информации о текущем процессе;
- получение информации обо всех системных процессах;
- задание прав для создания, опроса и управления платформенными процессами.

Что такое API процессов?

API процессов включает классы и интерфейсы для работы с платформенными процессами. Этот API позволяет выполнять следующие действия:

- создавать новые платформенные процессы из Java-программы;
- получать описатели платформенных процессов – неважно, созданы они из Java-программы или другими способами;
- завершать работающие процессы;
- узнавать, работает ли процесс, и запрашивать другие атрибуты;
- получать список дочерних процессов и родительский процесс;
- получать идентификатор платформенного процесса (PID);
- получать потоки ввода, вывода и ошибок вновь созданного процесса;
- ждать завершения процесса;
- выполнять некоторую операцию, когда процесс завершается.

В состав API процессов входят следующие классы и интерфейсы из пакета `java.lang`:

- `Runtime`
- `ProcessBuilder`
- `ProcessBuilder.Redirect`
- `Process`
- `ProcessHandle`
- `ProcessHandle.Info`

Работа с процессами поддерживалась, начиная с Java 1.0. Платформенный процесс, созданный в Java-программе, представляется классом `Process`. Процесс создается методом `exec()` класса `Runtime`.

В JDK 5.0 был добавлен класс `ProcessBuilder`, а в JDK 7.0 – вложенный класс `ProcessBuilder.Redirect`. В объекте `ProcessBuilder` хранится набор атрибутов процесса. Его метод `start()` запускает новый процесс и возвращает представляющий его объект `Process`. Метод `start()` можно вызывать многократно, и всякий раз он будет возвращать новый процесс с атрибутами, хранящимися в объекте `ProcessBuilder`. В Java 5.0 класс `ProcessBuilder` взял на себя работу метода `Runtime.exec()` по созданию нового процесса.

В Java 7 и Java 8 API процессов подвергся усовершенствованиям: было добавлено нескольких новых методов в классы `Process` и `ProcessBuilder`.

До Java 9 API процессов недоставало базовых средств для работы с платформенными процессами, в т. ч. получения идентификатора и владельца процесса, времени запуска процесса, потребленного процессом времени ЦП, количества работающих процессов и т. д. И раньше была возможность запускать платформенные процессы и работать с их потоками ввода, вывода и ошибок. Но невозможно было работать с процессами, запущенными не вами, и нельзя было получить сведения о процессах. Чтобы приблизиться к уровню процессов, разработчикам приходилось писать платформенный код, прибегая к технологии Java Native Interface (JNI). В Java 9 эти долгожданные средства работы с платформенными процессами наконец реализованы.

В Java 9 в API процессов добавлен интерфейс `ProcessHandle`. Его экземпляр идентифицирует платформенный процесс и позволяет опрашивать состояние процесса и управлять процессом.

Сравним класс `Process` с интерфейсом `ProcessHandle`. Экземпляр класса `Process` представляет платформенный процесс, запущенный Java-программой, а интерфейс `ProcessHandle` – процесс, запущенный как данной Java-программой, так и любым другим способом. В Java 9 в класс `Process` добавлено несколько методов, имеющихся также в новом интерфейсе `ProcessHandle`. Кроме того, класс `Process` содержит метод `toHandle()`, возвращающий значение типа `ProcessHandle`.

Экземпляр интерфейса `ProcessHandle.Info` представляет мгновенный снимок атрибутов процесса. Поскольку в разных операционных системах процессы реализованы по-разному, состав атрибутов зависит от системы. Состояние процесса может измениться в любой момент, например, потребленное время ЦП меняется всякий раз, как процесс получает квант времени. Для получения актуальной информации о процессе следует вызывать метод `info()` интерфейса `ProcessHandle` в тот момент, когда информация нужна. Этот метод возвращает новый экземпляр `ProcessHandle.Info`.

Все примеры в этой главе выполнялись в ОС Windows 10. На вашей машине могут получиться другие результаты.

Текущий процесс

Статический метод `current()` интерфейса `ProcessHandle` возвращает описатель текущего процесса. Это всегда тот процесс, который выполняет данную Java-программу.

```
// Получить описатель текущего процесса
ProcessHandle current = ProcessHandle.current();
```

Имея описатель текущего процесса, мы можем вызывать методы интерфейса `ProcessHandle` для получения сведений о процессе, как описано в следующем разделе.

> **Совет.** Невозможно снять текущий процесс. Попытка сделать это методом `destroy()` или `destroyForcibly()` интерфейса `ProcessHandle` приводит к исключению `IllegalStateException`.

Опрос состояния процесса

Для опроса состояния процесса предназначены методы интерфейса `ProcessHandle`. В табл. 12.1 перечислены наиболее употребительные методы этого интерфейса с краткими описаниями. Отметим, что многие методы возвращают мгновенный снимок состояния процесса на момент вызова. Нет никакой гарантии, что процесс будет находиться в том же состоянии тогда, когда вы соберетесь использовать его атрибуты, потому что процессы создаются, работают и завершаются асинхронно по отношению к вашей программе.

Таблица 12.1. Методы интерфейса ProcessHandle

Метод	Описание
`static Stream<ProcessHandle> allProcesses()`	Возвращает мгновенный снимок всех процессов ОС, видимых текущему процессу.
`Stream<ProcessHandle> children()`	Возвращает мгновенный снимок непосредственных потомков данного процесса. Для получения потомков любого уровня (детей, внуков, правнуков и т. д.) воспользуйтесь методом `descendants()`.
`static ProcessHandle current()`	Возвращает описатель `ProcessHandle` текущего процесса, т. е. процесс, в котором исполняется Java-программа, вызвавшая этот метод.
`Stream<ProcessHandle> descendants()`	Возвращает мгновенный снимок всех потомков процесса. Сравните с методом `children()`, который возвращает только непосредственных потомков.
`boolean destroy()`	Пытается завершить процесс. Возвращает `true`, если процесс успешно завершен, и `false` в противном случае. Разрешено ли программе завершать процесс, зависит от контроля доступа в операционной системе.
`boolean destroyForcibly()`	Пытается принудительно завершить процесс. Возвращает `true`, если процесс успешно завершен, и `false` в противном случае. В результате этого вызова процесс завершается немедленно, тогда как при нормальном завершении ему предоставляется шанс завершиться корректно. Разрешено ли программе завершать процесс, зависит от контроля доступа в операционной системе.

Метод	Описание
`long getPid()`	Возвращает идентификатор платформенного процесса (PID), присвоенный ему операционной системой. PID может использоваться повторно, поэтому два процесса с одинаковыми идентификаторами необязательно совпадают.
`ProcessHandle.Info info()`	Возвращает мгновенный снимок информации о процессе.
`boolean isAlive()`	Возвращает `true`, если процесс, представленный данным экземпляром `ProcessHandle`, еще не завершился, иначе `false`. Этот метод может возвращать `true` в течение некоторого времени после успешного запроса о завершении процесса, потому что процессы завершаются асинхронно.
`static Optional<ProcessHandle> of(long pid)`	Возвращает значение типа `Optional<ProcessHandle>`, представляющее существующий платформенный процесс. Возвращает пустой объект `Optional`, если процесс с указанным `pid` не существует
`CompletableFuture <ProcessHandle> onExit()`	Возвращает объект типа `CompletableFuture <ProcessHandle>`, соответствующее завершению процессу. Этот объект можно использовать, чтобы добавить задачу, которая будет выполнена, когда процесс завершится. Вызов этого метода для текущего процесса приводит к исключению `IllegalStateException`.
`Optional<ProcessHandle> parent()`	Возвращает значение типа `Optional<ProcessHandle>` для родительского процесса.
`boolean supportsNormalTermination()`	Возвращает `true`, если реализация `destroy()` завершает процесс нормально.

В табл. 12.2 перечислены методы вложенного интерфейса `ProcessHandle.Info`. Экземпляр этого интерфейса содержит мгновенный снимок информации о процессе. Для его получения следует вызвать метод `info()` интерфейса `ProcessHandle` или класса `Process`. Все методы этого интерфейса возвращают объекты типа `Optional`.

Таблица 12.2. Методы интерфейса ProcessHandle.Info

Метод	Описание
`Optional<String[]> arguments()`	Возвращает аргументы процесса. Процесс может изменить переданные ему аргументы после запуска. В таком случае возвращаются измененные аргументы.
`Optional<String> command()`	Возвращает путь к исполняемому файлу процесса.
`Optional<String> commandLine()`	Вспомогательный метод для объединения команды с аргументами. Возвращает командную строку процесса, полученную путем объединения значений, возвращенных методами `command()` и `arguments()`, если оба метода возвращают непустые объекты `Optional`.

Метод	Описание
`Optional<Instant> startInstant()`	Возвращает время запуска процесса. Если операционная система не позволяет узнать время запуска, то возвращается пустой объект `Optional`.
`Optional<Duration> totalCpuDuration()`	Возвращает общее время ЦП, потребленное процессом. Отметим, что процесс может работать очень долго, но потреблять при этом совсем мало времени ЦП.
`Optional<String> user()`	Возвращает пользователя процесса.

Пора уже посмотреть на интерфейсы `ProcessHandle` и `ProcessHandle.Info` в действии. Все рассматриваемые в этой главе классы находятся в модуле `com.jdojo.process.api`, объявление которого приведено в листинге 12.1.

Листинг 12.1. Объявление модуля com.jdojo.process.api

```
// module-info.java
module com.jdojo.process.api {
    exports com.jdojo.process.api;
}
```

В листинге 12.2 приведен код класса `CurrentProcessInfo`. Его метод `printInfo()` принимает в качестве аргумента объект типа `ProcessHandle` и печатает сведения о процессе. Метод `main()` получает описатель текущего процесса, в котором исполняется Java-программа, и вызывает метод `printInfo()`. Программа была запущена в ОС Windows 10.

Листинг 12.2. Класс CurrentProcessInfo печатает сведения о текущем процессе

```
// CurrentProcessInfo.java
package com.jdojo.process.api;

import java.time.Duration;
import java.time.Instant;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.util.Arrays;

public class CurrentProcessInfo {
   public static void main(String[] args) {
      // Получить описатель текущего процесса
      ProcessHandle current = ProcessHandle.current();

      // Напечатать сведения о процессе
      printInfo(current);
   }
```

```
   public static void printInfo(ProcessHandle handle) {
      // Получить идентификатор процесса
      long pid = handle.getPid();

      // Процесс еще работает?
      boolean isAlive = handle.isAlive();

      // Получить остальную информацию о процессе
      ProcessHandle.Info info = handle.info();
      String command = info.command().orElse("");
      String[] args = info.arguments()
                          .orElse(new String[]{});
      String commandLine = info.commandLine().orElse("");
      ZonedDateTime startTime = info.startInstant()
                                    .orElse(Instant.now())
                                    .atZone(ZoneId.systemDefault());
      Duration duration = info.totalCpuDuration()
                              .orElse(Duration.ZERO);
      String owner = info.user().orElse("Unknown");
      long childrenCount = handle.children().count();

      // Напечатать сведения о процессе
      System.out.printf("PID: %d%n", pid);
      System.out.printf("IsAlive: %b%n", isAlive);
      System.out.printf("Команда: %s%n", command);
      System.out.printf("Аргументы: %s%n", Arrays.toString(args));
      System.out.printf("Командная строка: %s%n", commandLine);
      System.out.printf("Время запуска: %s%n", startTime);
      System.out.printf("Время ЦП: %s%n", duration);
      System.out.printf("Владелец: %s%n", owner);
      System.out.printf("Число прямых потомков: %d%n", childrenCount);
   }
}
```

```
PID: 8692
IsAlive: true
Команда: C:\java9\bin\java.exe
Аргументы: []
Командная строка:
Время запуска: 2016-11-27T12:28:20.611-06:00[America/Chicago]
Время ЦП: PT0.296875S
Владелец: kishori\ksharan
Число прямых потомков: 1
```

Сравнение процессов

Сравнить процессы на совпадение не так-то просто. Полагаться только на PID'ы нельзя, т. к. операционная система повторно использует PID'ы завершившихся процессов. Можно проверить PID'ы и время запуска; если то и другое совпадает, то, вероятно, это один и тот процесс. Метод `equals()` реализации интерфейса `ProcessHandle` по умолчанию проверяет три вещи:

- реализации интерфейса `ProcessHandle` должны быть одинаковы для обоих процессов;
- PID'ы процессов должны совпадать;
- процессы должны быть запущены в одно и то же время.

Совет. Использование метода `compareTo()` для упорядочивания процессов из предлагаемой по умолчанию реализации интерфейса `ProcessHandle` не слишком полезно, т. к. в ней просто сравниваются PID'ы обоих процессов.

Создание процесса

Для запуска нового процесса необходим экземпляр класса `ProcessBuilder`, который содержит ряд методов для задания атрибутов процесса. Метод `start()` запускает новый процесс и возвращает объект `Process`, который можно использовать для работы с потоками ввода, вывода и ошибок процесса. В следующем фрагменте создается объект `ProcessBuilder` для запуска JVM в Windows:

```
ProcessBuilder pb = new ProcessBuilder()
            .command("C:\\java9\\bin\\java.exe",
                     "--module-path",
                     "myModulePath",
                     "--module",
                     "myModule/className")
            .inheritIO();
```

Задать команду и аргументы нового процесса можно двумя способами:

- передать их конструктору класса `ProcessBuilder`;
- воспользоваться методом `command()`.

Метод `command()` без аргументов возвращает команду, заданную в `ProcessBuilder`. Два других варианта – с переменным числом аргументов типа `String` и с аргументом типа `List<String>` – используются для задания команды и ее аргументов. Первый аргумент – путь к исполняемому файлу команды, остальные – аргументы этой команды.

У нового процесса свои собственные потоки ввода, вывода и ошибок. Метод `inheritIO()` делает эти потоки совпадающими с потоками текущего процесса. В классе `ProcessBuilder` есть несколько методов `redirectXxx()` для переопределения стандартных потоков ввода-вывода нового процесса. Например, можно перенаправить стандартный поток ошибок в файл для протоколирования. Задав все атрибуты процесса, можно вызывать метод `start()`, запускающий процесс:

```
// Запустить новый процесс
Process newProcess = pb.start();
```

Метод `start()` класса `ProcessBuilder` можно вызывать многократно, тогда будет запущено несколько процессов с одинаковыми атрибутами. При этом мы экономим на создании объектов `ProcessBuilder`.

Для получения описателя процесса служит метод `toHandle()` класса `Process`:

```
// Получить описатель процесса
ProcessHandle handle = newProcess.toHandle();
```

Описатель процесса можно использовать, чтобы снять процесс, дождаться его завершения или опросить состояние и атрибуты процесса, например, получить только прямых или всех потомков, родителей, потребленное время ЦП и т. д. Состав информации о процессе и доступные механизмы управления им зависят от операционной системы и ее средств контроля доступа.

Довольно трудно написать пример создания процесса, который работал бы во всех операционных системах. Я остановился на создании нового процесса, в котором запускается новый экземпляр JVM, выполняющей класс. Если вы смогли выполнить на своей машине другие примеры из этой книги, то будет работать и этот.

В листинге 12.3 приведен код класса `Job`. Его метод `main()` принимает два аргумента: длительность одного периода сна и общая продолжительность сна в секундах. Если они не переданы, то по умолчанию подразумеваются значения 5 и 60 секунд. Сначала метод пытается выделить первый и второй аргумент, если они заданы. Затем он получает описатель текущего процесса методом `ProcessHandle.current()`. Читается PID текущего процесса и печатается сообщение, содержащее PID, длительность одного периода сна и общую продолжительность сна. Наконец, метод входит в цикл, где на каждой итерации спит в течение одного периода, пока не истечет общая продолжительность сна. На каждой итерации печатается сообщение.

Листинг 12.3. Объявление класса Job

```
// Job.java
package com.jdojo.process.api;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;
import java.util.stream.Collectors;

/**
* Экземпляр этого класса – задача, которая периодически засыпает на заданное
* время, пока не будет превышена общая продолжительность сна. Длительность
* одного периода в секундах задается в качестве первого аргумента, а общая
* продолжительность сна – в качестве второго. По умолчанию подразумеваются
* значения 5 и 60. Если задано значение меньше 0, то оно заменяется на 0.
```

```
*/
public class Job {
   // Длительность одного периода сна
   public static final long DEFAULT_SLEEP_INTERVAL = 5;

   // Общая продолжительность сна
   public static final long DEFAULT_SLEEP_DURATION = 60;

   public static void main(String[] args) {
      long sleepInterval = DEFAULT_SLEEP_INTERVAL;
      long sleepDuration = DEFAULT_SLEEP_DURATION;

      // Получить переданную длительность одного периода
      if (args.length >= 1) {
         sleepInterval = parseArg(args[0], DEFAULT_SLEEP_INTERVAL);
         if (sleepInterval < 0) {
            sleepInterval = 0;
         }
      }

      // Получить переданную общую продолжительность сна
      if (args.length >= 2) {
         sleepDuration = parseArg(args[1], DEFAULT_SLEEP_DURATION);
         if (sleepDuration < 0) {
            sleepDuration = 0;
         }
      }

      long pid = ProcessHandle.current().getPid();
      System.out.printf("Сведения о задаче (pid=%d): длительность периода" +
                        "=%d секунд, продолжительность сна=%d " +
                        "секунд.%n",
                        pid, sleepInterval, sleepDuration);

      for (long sleptFor = 0; sleptFor < sleepDuration;
                              sleptFor += sleepInterval) {
         try {
            System.out.printf("Задача (pid=%d) будет спать" +
                              " %d секунд.%n",
                              pid, sleepInterval);
            // Спать в течение одного периода
            TimeUnit.SECONDS.sleep(sleepInterval);
         } catch (InterruptedException ex) {
            System.out.printf("Задача (pid=%d) была " +
                              "прервана.%n", pid);
         }
      }
```

```
}

/**
* Запускает новую JVM для выполнения класса Job.
* @param sleepInterval длительность периода сна. Передается
* JVM в первом аргументе.
* @param sleepDuration общая продолжительность сна. Передается
* JVM во втором аргументе.
* @return ссылка на процесс вновь запущенной JVM или
* null, если запустить JVM не удалось.
*/
public static Process startProcess(long sleepInterval,
                                   long sleepDuration) {
   // Команда запуска новой JVM сохраняется в List<String>
   List<String> cmd = new ArrayList<>();

   // Добавить компоненты команды по порядку
   addJvmPath(cmd);
   addModulePath(cmd);
   addClassPath(cmd);
   addMainClass(cmd);

   // Добавить аргументы
   cmd.add(String.valueOf(sleepInterval));
   cmd.add(String.valueOf(sleepDuration));

   // Построить атрибуты процесса
   ProcessBuilder pb = new ProcessBuilder()
              .command(cmd)
              .inheritIO();
   String commandLine = pb.command()
                          .stream()
                          .collect(Collectors.joining(" "));
   System.out.println("Выполняется команда:\n" + commandLine);

   // Запустить процесс
   Process p = null;
   try {
      p = pb.start();
   } catch (IOException e) {
      e.printStackTrace();
   }
   return p;
}

/**
* Разбирает аргументы, переданные JVM и передаемые затем методу main().
```

```
* @param valueStr строковое значение аргумента
* @param defaultValue значение аргумента, подразумеваемое по умолчанию,
* если valueStr не целое число.
* @return valueStr в виде long или defaultValue, если
* valueStr не целое число.
*/
private static long parseArg(String valueStr,
                             long defaultValue) {
   long value = defaultValue;
   if (valueStr != null) {
      try {
         value = Long.parseLong(valueStr);
      } catch (NumberFormatException e) {
         // ничего не делать
      }
   }
   return value;
}

/**
* Добавляет в список компонентов команды путь к JVM. Сначала пробует
* атрибут command текущего процесса, а если он отсутствует, то
* берет системное свойство java.home.
* @param cmd список компонентов команды
*/
private static void addJvmPath(List<String> cmd) {
   // Сначала пробуем получить команду запуска текущей JVM
   String jvmPath = ProcessHandle.current()
                                 .info()
                                 .command().orElse("");
   if(jvmPath.length() > 0) {
      cmd.add(jvmPath);
   } else {
      // Пробуем составить путь к JVM, пользуясь системным
      // свойством java.home
      final String FILE_SEPARATOR =
      System.getProperty("file.separator");
      jvmPath = System.getProperty("java.home") +
                           FILE_SEPARATOR + "bin" +
                           FILE_SEPARATOR + "java";
      cmd.add(jvmPath);
   }
}

/**
* Добавляет в список компонентов команды путь к модулям.
* @param cmd список компонентов команды
```

```
    */
    private static void addModulePath(List<String> cmd) {
        String modulePath =
        System.getProperty("jdk.module.path");
        if(modulePath != null && modulePath.trim().length() > 0) {
            cmd.add("--module-path");
            cmd.add(modulePath);
        }
    }
    /**
    * Добавляет в список компонентов команды путь к классам.
    * @param cmd список компонентов команды
    */
    private static void addClassPath(List<String> cmd) {
        String classPath = System.getProperty("java.class.path");
        if(classPath != null && classPath.trim().length() > 0) {
            cmd.add("--class-path");
            cmd.add(classPath);
        }
    }

    /**
    *
    * Добавляет в список компонентов команды главный класс. Добавляет
    * module/className или просто className в зависимости от того,
    * загружен класс Job в именованный или в безымянный модуль.
    * @param cmd список компонентов команды
    */
    private static void addMainClass(List<String> cmd) {
        Class<Job> cls = Job.class;
        String className = cls.getName();
        Module module = cls.getModule();
        if(module.isNamed()) {
            String moduleName = module.getName();
            cmd.add("--module");
            cmd.add(moduleName + "/" + className);
        } else {
        cmd.add(className);
        }
    }
}
```

В классе `Job` имеется метод `startProcess(long sleepInterval, long sleepDuration)`, запускающий новый процесс – JVM с классом `Job` в качестве главного класса. JVM передаются аргументы: длительность одного периода сна и общая продолжительность сна. Метод пытается построить команду для запуска `java` из каталога `JDK_HOME\bin`. Если класс `Job` был загружен в именованный модуль, то строится команда вида

```
JDK_HOME\bin\java --module-path <module-path> --module com.jdojo.process.api/com.jdojo.
process.api.Job <sleepInterval> <sleepDuration>
```

а если в безымянный – то команда вида

```
JDK_HOME\bin\java -class-path <class-path> com.jdojo.process.api.Job <sleepInterval>
<sleepDuration>
```

Метод `startProcess()` печатает команду, запускающую процесс, пытается запустить процесс и возвращает ссылку на него.

Метод `addJvmPath()` добавляет путь к исполняемому файлу JVM в список компонентов команды. Сначала он пытается использовать путь к текущей JVM. Если получить его не удается, то путь строится на основе системного свойства `java.home`.

В классе `Job` есть еще несколько вспомогательных методов для построения компонентов команды и разбора аргументов, переданных методу `main()`. Подробности смотрите в комментариях.

Если вы хотите запустить процесс, который будет работать 15 секунд и просыпаться каждые 5 секунд, то нужно вызвать метод `startProcess()` следующим образом:

```
// Запустить процесс, который будет работать 15 секунд
Process p = Job.startProcess(5, 15);
```

Для печати сведений о процессе воспользуемся методом `printInfo()` класса `CurrentProcessInfo` из листинга 12.2:

```
// Получить описатель текущего процесса
ProcessHandle handle = p.toHandle();

// Напечатать сведения о процессе
CurrentProcessInfo.printInfo(handle);
```

Значение, возвращенное методом `onExit()` интерфейса `ProcessHandle` можно использовать для выполнения операции по завершении процесса.

```
CompletableFuture<ProcessHandle> future = handle.onExit();

// Напечатать сообщение, когда процесс завершится
future.thenAccept((ProcessHandle ph) -> {
    System.out.printf("Задача (pid=%d) завершилась.%n", ph.getPid());
});
```

Можно дождаться завершения нового процесса:

```
// Ждать завершения процесса
future.get();
```

В этом примере вызов `future.get()` возвращает описатель процесса `ProcessHandle`. Нам он не нужен, потому что мы его уже и так знаем.

В листинге 12.4 приведен код класса `StartProcessTest`, из которого видно, как создать новый процесс с помощью класса `Job`. В методе `main()` создается новый процесс, печатаются сведения о нем, определяется операция, выполняемая при

завершении процесса, и снова печатаются сведения о процессе. Отметим, что процесс работает 15 секунд, но потребляет только 0.359375 секунд времени ЦП, поскольку большую часть времени спит. На вашей машине может получиться другой результат. Я прогонял программу в Windows 10.

Листинг 12.4. Класс StartProcessTest, создающий новые процессы

```
// StartProcessTest.java
package com.jdojo.process.api;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class StartProcessTest {
   public static void main(String[] args) {
      // Запустить процесс, работающий 15 секунд
      Process p = Job.startProcess(5, 15);

      if (p == null) {
         System.out.println("Не удалось создать новый процесс.");
         return;
      }

      // Получить описатель текущего процесса
      ProcessHandle handle = p.toHandle();

      // Напечатать сведения о процессе
      CurrentProcessInfo.printInfo(handle);

      CompletableFuture<ProcessHandle> future = handle.onExit();

      // Напечатать сообщение, когда процесс завершится
      future.thenAccept((ProcessHandle ph) -> {
         System.out.printf("Задача (pid=%d) завершилась.%n", ph.getPid());
      });

      try {
         // Ждать завершения процесса
         future.get();
      } catch (InterruptedException | ExecutionException e) {
         e.printStackTrace();
      }

      // Снова напечатать сведения о процессе
      CurrentProcessInfo.printInfo(handle);
   }
}
```

```
Выполняется команда:
C:\java9\bin\java.exe --module-path
C:\Java9Revealed\com.jdojo.process.api\build\classes --class-path
C:\Java9Revealed\com.jdojo.process.api\build\classes --module
com.jdojo.process.api/com.jdojo.process.api.Job 5 15
PID: 10928
IsAlive: true
Команда: C:\java9\bin\java.exe
Аргументы: []
Командная строка:
Время запуска: 2016-11-28T13:43:28.318-06:00[America/Chicago]
Время ЦП: PT0S
Владелец: kishori\ksharan
Число прямых потомков: 1
Сведения о задаче (pid=10928): длительность периода=5 секунд, продолжительность
сна=15 секунд.
Задача (pid=10928) заснет на 5 секунд.
Задача (pid=10928) заснет на 5 секунд.
Задача (pid=10928) заснет на 5 секунд.
Задача (pid=10928) завершилась.
PID: 10928
IsAlive: false
Команда:
Аргументы: []
Командная строка:
Время запуска: 2016-11-28T13:43:28.318-06:00[America/Chicago]
Время ЦП: PT0.359375S
Владелец: kishori\ksharan
Число прямых потомков: 0
```

Получение описателя процесса

Существует несколько способов получить описатель платформенного процесса. Если процесс создан Java-программой, то получить `ProcessHandle` можно от метода `toHandle()` класса `Process`. Но платформенный процесс можно создать и вне JVM. Для получения описателя платформенного процесса в интерфейсе `ProcessHandle` имеются следующие методы:

- `static Optional<ProcessHandle> of(long pid)`
- `static ProcessHandle current()`
- `Optional<ProcessHandle> parent()`
- `Stream<ProcessHandle> children()`
- `Stream<ProcessHandle> descendants()`
- `static Stream<ProcessHandle> allProcesses()`

Статический метод `of()` возвращает значение типа `Optional<ProcessHandle>` по заданному `pid`'у. Если процесса с таким `pid`'ом не существует, то возвращается пустой `Optional`. Чтобы воспользоваться этим методом, нужно знать PID процесса.

```
// Получить описатель процесса с pid = 1234
Optional<ProcessHandle> handle = ProcessHandle.of(1234L);
```

Статический метод `current()` возвращает описатель текущего процесса, т. е. процесса, исполняющего Java-программу. Пример мы уже видели в листинге 12.2.

Метод `parent()` возвращает описатель родительского процесса или пустой `Optional`, если у процесса нет родителя или получить родителя невозможно.

Метод `children()` возвращает мгновенный снимок всех прямых потомков процесса. Нет никакой гарантии, что процесс, возвращенный этим методом, все еще существует. Отметим, что уже завершившийся процесс не имеет потомков.

Метод `descendants()` возвращает мгновенный снимок всех потомков процесса прямых и отдаленных.

Метод `allProcesses()` возвращает мгновенный снимок всех процессов, видимых данному процессу. Не гарантируется, что к моменту обработки поток будет содержать все работающие процессы операционной системы. Возможно, что после создания мгновенного снимка какие-то процессы уже завершились, а какие-то были созданы. В следующем фрагменте печатаются отсортированные идентификаторы всех процессов.

```
System.out.printf("PID'ы всех процессов:%n");
ProcessHandle.allProcesses()
             .map(ph -> ph.getPid())
             .sorted()
             .forEach(System.out::println);
```

Для работающих процессов можно вычислять различные статистические показатели. Можно также написать на Java диспетчер задач, отображающий все работающие процессы и их атрибуты. В листинге 12.5 показано, как получить сведения о процессе, работающем дольше всех, и процессе, потребляющем больше всех времени ЦП. Для получения первого процесса я сравниваю время запуска, а для получения второго – общее потребленное время ЦП.

Листинг 12.5. Вычисление статистики процессов

```
// ProcessStats.java
package com.jdojo.process.api;

import java.time.Duration;
import java.time.Instant;

public class ProcessStats {
   public static void main(String[] args) {
      System.out.printf("Процесс, потребляющий максимальное время ЦП:%n");
      ProcessHandle.allProcesses()
                   .max(ProcessStats::compareCpuTime)
```

```
                .ifPresent(CurrentProcessInfo::printInfo);

        System.out.printf("%nПроцесс, работающий дольше всех:%n");
        ProcessHandle.allProcesses()
                .max(ProcessStats::compareStartTime)
                .ifPresent(CurrentProcessInfo::printInfo);
    }

    public static int compareCpuTime(ProcessHandle ph1,
                                     ProcessHandle ph2) {
        return ph1.info()
                .totalCpuDuration()
                .orElse(Duration.ZERO)
                .compareTo(ph2.info()
                            .totalCpuDuration()
                            .orElse(Duration.ZERO));
    }

    public static int compareStartTime(ProcessHandle ph1,
                                       ProcessHandle ph2) {
        return ph1.info()
                .startInstant()
                .orElse(Instant.now())
                .compareTo(ph2.info()
                            .startInstant()
                            .orElse(Instant.now()));
    }
}
```

```
Процесс, потребляющий максимальное время ЦП:
PID: 10696
IsAlive: true
Команда: C:\Program Files (x86)\Google\Chrome\Application\chrome.exe
Аргументы: []
Командная строка:
Время запуска: 2016-11-28T10:12:08.537-06:00[America/Chicago]
Время ЦП: PT14M26.5S
Владелец: kishori\ksharan
Число прямых потомков: 0

Процесс, работающий дольше всех:
PID: 0
IsAlive: false
Команда:
Аргументы: []
Командная строка:
```

```
Время запуска: 2016-11-29T13:18:22.262776600-06:00[America/Chicago]
Время ЦП: PT0S
Владелец: Unknown
Число прямых потомков: 127
```

Завершение процесса

Для завершения процесса служат методы `destroy()` и `destroyForcibly()` интерфейса `ProcessHandle` и класса `Process`. Оба метода возвращают `true`, если удалось завершить процесс, и `false` в противном случае. Метод `destroy()` пытается завершить процесс нормально, а метод `destroyForcibly()` принудительно. Может случиться, что метод `isAlive()` возвращает `true` в течение короткого времени после запроса о завершении.

> **Совет.** Завершить текущий процесс нельзя. Попытка вызвать метод `destroy()` или `destroyForcibly()` для текущего процесса приводит к исключению `IllegalStateException`. Кроме того, средства контроля доступа операционной системы могут запретить завершение процесса.

В случае нормального завершения процесс получает возможность завершиться корректно, а в случае принудительного – снимается немедленно. Действительно ли процесс завершается нормально, зависит от реализации. Метод `supportsNormalTermination()` интерфейса `ProcessHandle` и класса `Process` позволяет узнать, поддерживается ли нормальное завершение. Если да, то метод возвращает `true`, иначе `false`.

Вызов любого из этих методов для завершения уже завершенного процесса не дает никакого эффекта. По завершении процесса объект `CompletableFuture<Process>`, возвращенный методом `onExit()` класса `Process`, и объект `CompletableFuture<ProcessHandle>`, возвращенный методом `onExit()` интерфейса `ProcessHandle`, становятся завершенными.

Управление правами процесса

В примерах из предыдущего раздела предполагалось, что диспетчер безопасности Java не установлен. Если же диспетчер безопасности установлен, то для запуска, управления и опроса платформенных процессов необходимо иметь права.

- Для создания нового процесса необходимо иметь право `FilePermission(cmd,"execute")`, где `cmd` – абсолютный путь к программе, исполняемой в процессе. Если `cmd` – не абсолютный путь, то нужно иметь право `FilePermission("<<ALL FILES>>","execute")`.
- Для опроса состояния платформенных процессов и снятия процессов методами интерфейса `ProcessHandle`, приложение должно обладать правом `RuntimePermission("manageProcess")`.

В листинге 12.6 приведена программа, которая получает число процессов и создает новый процесс. Она выполняет обе задачи без диспетчера безопасности и с ним.

Листинг 12.6. Управление процессами с диспетчером безопасности

```
// ManageProcessPermission.java
package com.jdojo.process.api;

import java.util.concurrent.ExecutionException;

public class ManageProcessPermission {
   public static void main(String[] args) {
      // Получить число процессов
      long count = ProcessHandle.allProcesses().count();
      System.out.printf("Число процессов: %d%n", count);

      // Запустить новый процесс
      Process p = Job.startProcess(1, 3);

      try {
         p.toHandle().onExit().get();
      } catch (InterruptedException | ExecutionException e) {
         System.out.println(e.getMessage());
      }

      // Установить диспетчер безопасности
      SecurityManager sm = System.getSecurityManager();
      if(sm == null) {
         System.setSecurityManager(new SecurityManager());
         System.out.println("Диспетчер безопасности установлен.");
      }

      // Получить число процессов
      try {
         count = ProcessHandle.allProcesses().count();
         System.out.printf("Число процессов: %d%n", count);
      } catch(RuntimeException e) {
         System.out.println("Не удалось получить число процессов: " +
                            e.getMessage());
      }
      // Запустить новый процесс
      try {
         p = Job.startProcess(1, 3);
         p.toHandle().onExit().get();
      } catch (InterruptedException | ExecutionException |
               RuntimeException e) {
         System.out.println("Не удалось запустить новый процесс: " +
                            e.getMessage());
      }
   }
}
```

Попробуйте выполнить класс `ManageProcessPermission` следующей командой, не изменяя файлов политик Java:

```
C:\Java9Revealed>java --module-path
C:\Java9Revealed\com.jdojo.process.api\build\classes --module
com.jdojo.process.api/com.jdojo.process.api.ManageProcessPermission
```

```
Число процессов: 126
Выполняется команда:
C:\java9\bin\java.exe --module-path
C:\Java9Revealed\com.jdojo.process.api\build\classes --module
com.jdojo.process.api/com.jdojo.process.api.Job 1 3
Сведения о задаче (pid=6320): длительность периода=1 секунд, продолжительность сна=3 секунд.
Задача (pid=6320) заснет на 5 секунд.
Задача (pid=6320) заснет на 5 секунд.
Задача (pid=6320) заснет на 5 секунд.
Диспетчер безопасности установлен.
Не удалось получить число процессов: access denied ("java.lang.RuntimePermission"
"manageProcess")
Не удалось запустить новый процесс: access denied ("java.lang.RuntimePermission"
"manageProcess")
```

Как видим, до установки диспетчера безопасности мы могли получить число процессов и создать новый процесс, а после установки среда выполнения Java в обоих случаях возбуждает исключения. Чтобы исправить ошибку, необходимо предоставить следующие права:

- `RuntimePermission "manageProcess"`, позволяющее приложению опрашивать платформенный процесс и создавать новые процессы;
- `FilePermission "execute"` для пути к Java-программе, позволяющее запускать JVM;
- `PropertyPermission "read"` на системные свойства `"jdk.module.path"` и `"java.class.path"`, чтобы класс `Job` мог читать эти свойства при построении командной строки для запуска JVM.

В листинге 12.7 показан скрипт, предоставляющий эти права всем программам. Вы должны записать этот скрипт в файл `JDK_HOME\conf\security\java.policy` на своей машине. Путь к `C:\\java9\\bin\\java.exe` правилен только в Windows и только если JDK 9 установлен в каталог `C:\java9 directory`. В остальных случаях измените этот путь, чтобы он указывал на исполнитель Java-программ.

Листинг 12.7. Добавление к файлу `JDK_HOME|conf\security\java.policy`

```
grant {
    permission java.lang.RuntimePermission "manageProcess";
    permission java.io.FilePermission "C:\\java9\\bin\\java.exe", "execute";
    permission java.util.PropertyPermission "jdk.module.path", "read";
```

```
    permission java.util.PropertyPermission "java.class.path", "read";
};
```

Если теперь выполнить ту же команду, что и раньше, то будет напечатано примерно следующее:

```
Число процессов: 133
Выполняется команда:
C:\java9\bin\java.exe --module-path
C:\Java9Revealed\com.jdojo.process.api\build\classes --module
com.jdojo.process.api/com.jdojo.process.api.Job 1 3
Сведения о задаче (pid=3108): длительность периода=1 секунд, продолжительность сна=3 секунд.
Задача (pid=3108) заснет на 5 секунд.
Задача (pid=3108) заснет на 5 секунд.
Задача (pid=3108) заснет на 5 секунд.
Диспетчер безопасности установлен.
Число процессов: 133
Выполняется команда:
C:\java9\bin\java.exe --module-path
C:\Java9Revealed\com.jdojo.process.api\build\classes --module
com.jdojo.process.api/com.jdojo.process.api.Job 1 3
Сведения о задаче (pid=3684): длительность периода=1 секунд, продолжительность сна=3 секунд.
Задача (pid=3684) заснет на 5 секунд.
Задача (pid=3684) заснет на 5 секунд.
Задача (pid=3684) заснет на 5 секунд.
```

Резюме

API процессов включает классы и интерфейсы для работы с платформенными процессами. API процессов существовал в Java SE со времен версии 1.0 в виде классов `Runtime` и `Process`. Он позволял создавать платформенные процессы, управлять их потоками ввода-вывода и завершать их. В последующих версиях Java SE этот API был усовершенствован. Но до Java 9 разработчикам приходилось писать платформенный код для выполнения таких простых задач, как получение идентификатора процесса, команды, запустившей процесс и т. п. В Java 9 добавлен интерфейс `ProcessHandle` для представления описателя процесса. Описатели позволяют опрашивать платформенные процессы и управлять ими.

API процессов составляют следующие классы и интерфейсы: `Runtime`, `ProcessBuilder`, `ProcessBuilder.Redirect`, `Process`, `ProcessHandle` и `ProcessHandle.Info`.

Для запуска платформенного процесса можно использовать метод `exec()` класса `Runtime`, но предпочтительнее метод `start()` класса `ProcessBuilder`. Объект класса `ProcessBuilder.Redirect` представляет источник ввода процесса или его вывод.

Экземпляр класса `Process` представляет платформенный процесс, созданный Java-программой.

Экземпляр интерфейса `ProcessHandle` представляет процесс, созданный Java-программой или иными способами. Этот интерфейс добавлен в Java 9 и предоставляет несколько методов для опроса процессов и управления ими. Экземпляр интерфейса `ProcessHandle.Info` представляет мгновенный снимок информации о процессах, его можно получить от метода `info()` класса `Process` или интерфейса `ProcessHandle`. Если имеется экземпляр `Process`, то можно воспользоваться его методом `toHandle()` для получения `ProcessHandle`.

Метод `onExit()` интерфейса `ProcessHandle` возвращает объект `CompletableFuture<ProcessHandle>` для выполнения некоторой операции после завершения процесса. Этот метод нельзя использовать для текущего процесса.

Если установлен диспетчер безопасности, то для опроса платформенных процессов и управления ими необходимо иметь право `RuntimePermission "manageProcess"`, а для запуска процесса – право `FilePermission "execute"` на исполняемый файл.

Глава 13

Изменения API коллекций

Краткое содержание главы:

- создание немодифицируемых списков, множеств и отображений до JDK 9 и проблемы, возникавшие при их использовании;
- создание немодифицируемых списков с помощью статического фабричного метода `of()` интерфейса `List` в JDK 9;
- создание немодифицируемых множеств с помощью статического фабричного метода `of()` интерфейса `Set` в JDK 9;
- создание немодифицируемых отображений с помощью статических фабричных методов `of()`, `ofEntries()` и `entry()` интерфейса `Map` в JDK 9.

Общие сведения

API коллекций включает классы и интерфейсы, предоставляющие возможность хранить и манипулировать различными типами коллекций объектов: списками, множествами и отображениями. Этот API был добавлен в Java SE 1.2. В языке Java не поддерживаются литеральные коллекции – самый простой и быстрый способ объявления коллекций с одновременной инициализацией. Если бы литералы поддерживались, то список `List`, содержащий числа 100 и 200, можно было бы инициализировать так:

```
List<Integer> list = [100, 200];
```

Литеральная коллекция компактно записывается и с ней легко обращаться. Ее можно эффективно разместить в памяти, т. к. число элементов заранее известно. Ее можно сделать неизменяемой и, следовательно, потокобезопасной.

Включение литеральных коллекций в Java несколько раз обсуждалось до JDK 9. Но проектировщики решили не делать этого, по крайней мере, в JDK 9. Добавление литеральных коллекций на данной стадии потребовало бы слишком много усилий при малой отдаче. Было решено достичь той же цели, расширив API коллекций путем добавления статических фабричных методов в интерфейсы `List`, `Set` и `Map`, и тем открыть возможность для простого и эффективного создания небольших немодифицируемых коллекций.

Существующий API позволяет создавать только изменяемые коллекции. Чтобы сделать коллекцию неизменяемой (или немодифицируемой), ее нужно обернуть другим объектом. Так, для создания немодифицируемого списка с двумя целыми числами в JDK 8 и более ранних версиях использовался примерно такой код:

```
// Создать пустой изменяемый список
List<Integer> list = new ArrayList<>();

// Добавить в него два элемента
list.add(100);
list.add(200);

// Создать неизменяемый список, обернув изменяемый
List<Integer> list2 = Collections.unmodifiableList(list);
```

У такого подхода есть серьезный недостаток. Немодифицируемый список – всего лишь обертка модифицируемого. Я сознательно оставил переменную `list` видимой. Да, мы не можем модифицировать список через переменную `list2`, но через переменную `list` мы можем внести в него изменения, которые будут видны и при доступе через переменную `list2`. В листинге 13.1 приведен полный код программы, которая создает немодифицируемый список, а затем изменяет его.

Листинг 13.1. Создание немодифицируемого списка до JDK 9

```
// PreJDK9UnmodifiableList.java
package com.jdojo.collection;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class PreJDK9UnmodifiableList {
   public static void main(String[] args) {
      List<Integer> list = new ArrayList<>();
      list.add(100);
      list.add(200);
      System.out.println("list = " + list);

      // Создать немодифицируемый список
      List<Integer> list2 = Collections.unmodifiableList(list);
      System.out.println("list2 = " + list2);

      // Добавить элемент через переменную list
      list.add(300);

      // Напечатать содержимое списка через переменные list и list2
      System.out.println("list = " + list);
      System.out.println("list2 = " + list2);
```

```
    }
}
```

```
list = [100, 200]
list2 = [100, 200]
list = [100, 200, 300]
list2 = [100, 200, 300]
```

Как видим, сохранив ссылку на исходный список, мы можем изменять его, и якобы немодифицируемый список на деле становится модифицируемым! Проблему можно решить, записав в исходную переменную `list` ссылку на немодифицируемый список:

```
List<Integer> list = new ArrayList<>();
list.add(100);
list.add(200);

// Создать немодифицируемый список и сохранить его в list
list = Collections.unmodifiableList(list);
```

Отметим, что в этом примере для создания и заполнения немодифицируемого списка понадобилось несколько предложений. Если требуется объявить и инициализировать список в переменной экземпляра или статической переменной класса, то этот подход не годится, потому что такое объявление должно быть простым и компактным – занимающим одно предложение. Если использовать предыдущий фрагмент кода для переменной экземпляра, то получится что-то типа:

```
public class Test {
    private List<Integer> list = new ArrayList<>();
    {
list.add(100);
list.add(200);
list = Collections.unmodifiableList(list);
    }

    // ...
}
```

Существуют и другие способы объявить и инициализировать немодифицируемый список, например, воспользоваться массивом, а затем преобразовать его в список. Ниже показано три таких способа:

```
public class Test {
// Преобразование массива в список
private List<Integer> list2 = Collections.unmodifiableList(
    new ArrayList<>( Arrays.asList(100, 200)));

// Использование анонимного класса
```

```
private List<Integer> list3 = Collections.unmodifiableList(
    new ArrayList<>(){{add(100); add(200);}});

// Использование потока
private List<Integer> list4 = Collections.unmodifiableList(
    Stream.of(100, 200).collect(Collectors.toList()));

// Прочий код
}
```

Этот пример доказывает, что создать немодифицируемый список в одном предложении можно. Но все эти варианты громоздки и неэффективны: чтобы всего лишь поместить в список два целых числа, мы создаем несколько объектов и храним значения в промежуточном массиве.

В JDK 9 эти проблемы решены путем добавления статических фабричных методов в интерфейсы `List`, `Set` и `Map`. Метод называется `of()` и имеет несколько перегруженных вариантов. В JDK 9 для объявления и инициализации немодифицируемого списка из двух элементов можно написать:

```
// Создать немодифицируемый список с двумя целыми числами
List<Integer> list = List.of(100, 200);
```

Немодифицируемые списки

В JDK 9 в интерфейс `List` добавлен перегруженный статический фабричный метод `of()`, предлагающий простой и компактный способ создания немодифицируемых списков. Вот все варианты метода `of()`:

- `static <E> List<E> of()`
- `static <E> List<E> of(E e1)`
- `static <E> List<E> of(E e1, E e2)`
- `static <E> List<E> of(E e1, E e2, E e3)`
- `static <E> List<E> of(E e1, E e2, E e3, E e4)`
- `static <E> List<E> of(E e1, E e2, E e3, E e4, E e5)`
- `static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6)`
- `static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)`
- `static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)`
- `static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)`
- `static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)`
- `static <E> List<E> of(E... elements)`

Одиннадцать вариантов метода `of()` предназначены для создания списков, содержащих от 0 до 10 элементов. Последний вариант принимает переменное число аргументов и создает немодифицируемый список с произвольным числом элементов. Спрашивается, зачем так много вариантов, раз есть возможность создать список с произвольным числом элементов? Только из соображений производительности. Проектировщики API хотели, чтобы списки с небольшим числом элементов создавались эффективно. А переменное число аргументов реализовано

с помощью массива. Поскольку методы с фиксированным числом аргументов не упаковывают элементы в массив, они работают эффективнее. Для небольших списков в реализации используются специальные классы.

Списки, возвращаемые любым из методов `of()`, обладают следующими характеристиками:

- они структурно неизменяемы; попытка добавить, заменить или удалить элемент приводит к исключению `UnsupportedOperationException`;
- в них не допускаются элементы, равные `null`; при попытке поместить в список `null` возбуждается исключение `NullPointerException`;
- они сериализуемы, если все элементы сериализуемы;
- порядок элементов списка такой же, как порядок аргументов метода `of()`;
- не дается никаких гарантий о конкретном классе возвращенного объекта, т. е. не следует думать, что это объект класса `ArrayList` или какого-то другого класса, реализующего интерфейс `List`. В реализациях методов используются внутренние классы, об именах которых не следует делать никаких предположений. Так, вызовы `List.of()` и `List.of("A")` могут возвращать объекты разных классов.

В классе `Collections` имеется статическое поле `EMPTY_LIST`, представляющее неизменяемый пустой список. Имеется также статический метод `emptyList()` для получения неизменяемого пустого списка. А метод `singletonList(T object)` возвращает неизменяемый одноэлементный список, содержащий заданный элемент. Ниже показано, как создается неизменяемый пустой список и одноэлементный список в прежних версиях и в JDK 9:

```
// Создание пустого неизменяемого списка до JDK 9
List<Integer> emptyList1 = Collections.EMPTY_LIST;
List<Integer> emptyList2 = Collections.emptyList();

// Создание пустого списка в JDK 9
List<Integer> emptyList = List.of();

// Создание неизменяемого одноэлементного списка до JDK 9
List<Integer> singletonList1 = Collections.singletonList(100);

// Создание неизменяемого одноэлементного списка в JDK 9
List<Integer> singletonList = List.of(100);
```

Как использовать метод `of()` для создания немодифицируемого списка из массива? Ответ зависит от того, какой список вам нужен: содержащий те же элементы, что массив, или содержащий сам массив в качестве единственного элемента. Конструкция `List.of(array)` приводит к вызову метода `of(E... elements)` и возвращенный список, будет содержать те же элементы, что массив. Если же нужно поместить в список сам массив, то поможет конструкция `List.<array-type>of(array)` – тогда будет вызван метод `of(E e1)`, так что в возвращенном списке будет всего один элемент – сам массив. В следующем примере сказанное демонстрируется для массива объектов `Integer`:

```
Integer[] nums = {100, 200};

// Создать список, содержащий те же элементы, что массив
List<Integer> list1 = List.of(nums);
System.out.println("list1 = " + list1);
System.out.println("list1.size() = " + list1.size());

// Создать список, единственным элементом которого является сам массив
List<Integer[]> list2 = List.<Integer[]>of(nums);
System.out.println("list2 = " + list2);
System.out.println("list2.size() = " + list2.size());
```

```
list1 = [100, 200]
list1.size() = 2
list2 = [[Ljava.lang.Integer;@27efef64]
list2.size() = 1
```

В листинге 13.2 приведен полный код программы, демонстрирующей использование статического фабричного метода `of()` интерфейса `List` для создания немодифицируемых списков.

Листинг 13.2. Использование статического метода List.of() для создания немодифицируемых списков

```
// ListTest.java
package com.jdojo.collection;

import java.util.List;

public class ListTest {
   public static void main(String[] args) {
      // Создать несколько немодифицируемых списков
      List<Integer> emptyList = List.of();
      List<Integer> luckyNumber = List.of(19);
      List<String> vowels = List.of("A", "E", "I", "O", "U");

      System.out.println("emptyList = " + emptyList);
      System.out.println("singletonList = " + luckyNumber);
      System.out.println("vowels = " + vowels);

      try {
         // Пробуем включить в список null
         List<Integer> list = List.of(1, 2, null, 3);
      } catch(NullPointerException e) {
```

```
            System.out.println("Наличие null в List.of() запрещено.");
        }

        try {
            // Пробуем добавить в список элемент
            luckyNumber.add(8);
        } catch(UnsupportedOperationException e) {
            System.out.println("Добавить элемент невозможно.");
        }

        try {
            // Пробуем удалить элемент
            luckyNumber.remove(0);
        } catch(UnsupportedOperationException e) {
            System.out.println("Удалить элемент невозможно.");
        }
    }
}
```

```
emptyList = []
singletonList = [19]
vowels = [A, E, I, O, U]
Наличие null в List.of() запрещено.
Добавить элемент невозможно.
Удалить элемент невозможно.
```

Немодифицируемые множества

В JDK 9 в интерфейс `Set` добавлен перегруженный статический фабричный метод `of()`, предлагающий простой и компактный способ создания немодифицируемых множеств. Вот все варианты метода `of()`:

- `static <E> Set<E> of()`
- `static <E> Set<E> of(E e1)`
- `static <E> Set<E> of(E e1, E e2)`
- `static <E> Set<E> of(E e1, E e2, E e3)`
- `static <E> Set<E> of(E e1, E e2, E e3, E e4)`
- `static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5)`
- `static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6)`
- `static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)`
- `static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)`
- `static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)`
- `static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)`
- `static <E> Set<E> of(E... elements)`

Все варианты методы `of()` оптимизированы для достижения максимальной производительности. Первые одиннадцать вариантов предназначены для создания множеств, содержащих от 0 до 10 элементов. Они существуют только для того, чтобы избежать упаковки аргументов в массив для множеств небольшого размера. Вариант с переменным числом аргументов позволяет создать немодифицируемое множество с произвольным числом элементов.

Множества, возвращаемые любым из методов `of()`, обладают следующими характеристиками:

- они структурно неизменяемы; попытка добавить, заменить или удалить элемент приводит к исключению `UnsupportedOperationException`;
- в них не допускаются элементы, равные `null`; при попытке поместить в множество `null` возбуждается исключение `NullPointerException`;
- они сериализуемы, если все элементы сериализуемы;
- в них не допускаются дубликаты; попытка задать повторяющиеся элементы приводит к исключению `IllegalArgumentException`;
- не дается никаких гарантий о конкретном классе возвращенного объекта, т. е. не следует думать, что это объект класса `HashSet` или какого-то другого класса, реализующего интерфейс `Set`. В реализациях методов используются внутренние классы, об именах которых не следует делать никаких предположений. Так, вызовы `Set.of()` и `Set.of("A")` могут возвращать объекты разных классов.

В классе `Collections` имеется статическое поле `EMPTY_SET`, представляющее неизменяемое пустое множество. Имеется также статический метод `emptySet()` для получения неизменяемого пустого множества. А метод `singleton(T object)` возвращает неизменяемое одноэлементное множество, содержащее заданный элемент. Ниже показано, как создается неизменяемое пустое множество и одноэлементное множество в прежних версиях и в JDK 9:

```
// Создание неизменяемого пустого множества до JDK 9
Set<Integer> emptySet1 = Collections.EMPTY_SET;
Set<Integer> emptySet2 = Collections.emptySet();

// Создание пустого множества в JDK 9
Set<Integer> emptySet = Set.of();

// Создание неизменяемого одноэлементного множества до JDK 9
Set<Integer> singletonSet1 = Collections.singleton(100);

// Создание неизменяемого одноэлементного множества в JDK 9
Set<Integer> singletonSet = Set.of(100);
```

Ниже демонстрируется, как создать немодифицируемое множество из массива: содержащее те же элементы, что в массиве, и содержащее сам массив в качестве единственного элемента. Отметим, что в первом случае массив не должен содержать дубликатов, иначе будет возбуждено исключение `IllegalArgumentException`.

```
Integer[] nums = {100, 200};

// Создать множество, содержащее те же элементы, что массив
Set<Integer> set1 = Set.of(nums);
System.out.println("set1 = " + set1);
System.out.println("set1.size() = " + set1.size());

// Создать множество, единственным элементом которого является массив
Set<Integer[]> set2 = Set.<Integer[]>of(nums);
System.out.println("set2 = " + set2);
System.out.println("set2.size() = " + set2.size());

// Создать массив с повторяющимися элементами
Integer[] nums2 = {101, 201, 101};

// Попытаться создать множество, единственным элементом которого является этот
// массив
Set<Integer[]> set3 = Set.<Integer[]>of(nums2);
System.out.println("set3 = " + set3);
System.out.println("set3.size() = " + set3.size());

try {
   // Попытаться создать множество, содержащее те же элементы,
   // что этот массив. Возбуждается исключение IllegalArgumentException.
   Set<Integer> set4 = Set.of(nums2);
   System.out.println("set4 = " + set4);
} catch(IllegalArgumentException e) {
   System.out.println(e.getMessage());
}
```

```
set1 = [100, 200]
set1.size() = 2
set2 = [[Ljava.lang.Integer;@47c62251]
set2.size() = 1
set3 = [[Ljava.lang.Integer;@3e6fa38a]
set3.size() = 1
duplicate element: 101
```

В листинге 13.3 приведен полный код программы, демонстрирующей использование статического фабричного метода `of()` интерфейса `Set` для создания немодифицируемых множеств. Элементы множества могут выводиться не в том же порядке, в котором были указаны при создании, потому что порядок элементов в множестве не гарантирован.

Листинг 13.3. Использование статического метода Set.of() для создания немодифицируемых множеств

```
// SetTest.java
package com.jdojo.collection;

import java.util.Set;

public class SetTest {
   public static void main(String[] args) {
      // Создать несколько немодифицируемых множеств
      Set<Integer> emptySet = Set.of();
      Set<Integer> luckyNumber = Set.of(19);
      Set<String> vowels = Set.of("A", "E", "I", "O", "U");

      System.out.println("emptySet = " + emptySet);
      System.out.println("singletonSet = " + luckyNumber);
      System.out.println("vowels = " + vowels);

      try {
         // Пробуем включить в множество null
         Set<Integer> set = Set.of(1, 2, null, 3);
      } catch(NullPointerException e) {
         System.out.println("Наличие null в Set.of() запрещено.");
      }

      try {
         // Пробуем включить дубликаты
         Set<Integer> set = Set.of(1, 2, 3, 2);
      } catch(IllegalArgumentException e) {
         System.out.println(e.getMessage());
      }

      try {
         // Пробуем добавить в множество элемент
         luckyNumber.add(8);
      } catch(UnsupportedOperationException e) {
         System.out.println("Добавить элемент невозможно.");
      }

      try {
         // Пробуем удалить элемент
         luckyNumber.remove(0);
      } catch(UnsupportedOperationException e) {
         System.out.println("Удалить элемент невозможно.");
```

```
        }
    }
}
```

```
emptySet = []
singletonSet = [19]
vowels = [E, O, A, U, I]
Наличие null в Set.of() запрещено.
duplicate element: 2
Добавить элемент невозможно.
Удалить элемент невозможно.
```

Немодифицируемые отображения

В JDK 9 в интерфейс `Map` добавлен перегруженный статический фабричный метод `of()`, предлагающий простой и компактный способ создания немодифицируемых отображений. Следующие 11 вариантов этого метода позволяют создать немодифицируемый объект `Map`, содержащий от 0 до 10 пар ключ-значение:

- `static <K,V> Map<K,V> of()`
- `static <K,V> Map<K,V> of(K k1, V v1)`
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2)`
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3)`
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4)`
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5)`
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6)`
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, k5, V v5, K k6, V v6, K k7, V v7)`
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8)`
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9)`
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9, K k10, V v10)`

Обратите внимание на порядок аргументов. Первый и второй аргумент – соответственно ключ и значение первой пары отображения, третий и четвертый – второй пары и т. д. Отметим, что для `Map` нет варианта метода `of()` с переменным числом аргументов, как для `List` и `Set`. Это связано с тем, что один элемент отображения содержит пару (ключ и значение), а в Java возможен только один аргумент переменной длины. Ниже показано, как создаются отображения с помощью метода `of()`:

```
// Пустое немодифицируемое отображение
Map<Integer, String> emptyMap = Map.of();
```

```
// Немодифицируемое одноэлементное отображение
Map<Integer, String> singletonMap = Map.of(1, "One");

// Немодифицируемое отображение с двумя элементами
Map<Integer, String> luckyNumbers = Map.of(1, "One", 2, "Two");
```

Для создания немодифицируемого отображения с произвольным числом элементов в JDK 9 предоставляется статический метод `ofEntries()` интерфейса `Map`:

```
<K,V> Map<K,V> ofEntries(Map.Entry<? extends K,? extends V>... entries)
```

Для использования метода `ofEntries()` каждый элемент отображения следует обернуть объектом `Map.Entry`. В JDK 9 в интерфейсе `Map` имеется вспомогательный статический метод `entry()` для создания объектов `Map.Entry`:

```
<K,V> Map.Entry<K,V> entry(K k, V v)
```

Чтобы выражение было понятным и компактным, следует статически импортировать метод `Map.entry` и создавать немодифицируемое отображение следующим образом:

```
import java.util.Map;
import static java.util.Map.entry;

// ...

// Использование методов Map.ofEntries() и Map.entry() для создания
// немодифицируемого отображения
Map<Integer, String> numberToWord =
       Map.ofEntries(entry(1, "One"),
                     entry(2, "Two"),
                     entry(3, "Three"));
```

Множества, возвращаемые методами `of()` и `ofEntries()`, обладают следующими характеристиками:

- они структурно неизменяемы; попытка добавить, заменить или удалить элемент приводит к исключению `UnsupportedOperationException`;
- в них не допускаются ключи и значения, равные `null`; нарушение этого ограничения влечет за собой исключение `NullPointerException`;
- они сериализуемы, если все ключи и значения сериализуемы;
- в них не допускаются дубликаты ключей; попытка задать повторяющиеся ключи приводит к исключению `IllegalArgumentException`;
- порядок обхода отображения не определен;
- не дается никаких гарантий о конкретном классе возвращенного объекта, т. е. не следует думать, что это объект класса `HashMap` или какого-то другого класса, реализующего интерфейс `Map`. В реализациях методов используются внутренние классы, об именах которых не следует делать никаких предположений. Так, вызовы `Mao.of()` и `Map.of(1, "One")` могут возвращать объекты разных классов.

В классе `Collections` имеется статическое поле `EMPTY_MAP`, представляющее неизменяемое пустое отображение. Имеется также статический метод `emptyMap()` для получения неизменяемого пустого отображения. А метод `singletonMap(K key, V value)` возвращает неизменяемое одноэлементное отображение, содержащее заданные ключ и значение. Ниже показано, как создается неизменяемое пустое отображение и одноэлементное отображение в прежних версиях и в JDK 9:

```
// Создание неизменяемого пустого отображения до JDK 9
Map<Integer,String> emptyMap1 = Collections.EMPTY_MAP;
Map<Integer,String> emptyMap2 = Collections.emptyMap();

// Создание пустого отображения в JDK 9
Map<Integer,String> emptyMap = Map.of();

// Создание неизменяемого одноэлементного отображения до JDK 9
Map<Integer,String> singletonMap1 =
Collections.singletonMap(1, "One");

// Создание неизменяемого одноэлементного отображения в JDK 9
Map<Integer,String> singletonMap = Map.of(1, "One");
```

В листинге 13.4 приведен полный код программы, демонстрирующей использование статических методов `of()`, `ofEntries()` и `entry()` интерфейса `Map` для создания немодифицируемых отображений. Обратите внимание на порядок задания аргументов метода и порядок вывода элементов отображения. Они могут не совпадать, поскольку отображение, как и множество, не гарантирует извлечение элементов в определенном порядке.

Листинг 13.4. Использование статических методов `Map.of()`, `Map.ofEntries()` и `Map.entry()` для создания немодифицируемых отображений

```
// MapTest.java
package com.jdojo.collection;

import java.util.Map;
import static java.util.Map.entry;

public class MapTest {
   public static void main(String[] args) {
      // Создать несколько немодифицируемых отображений
      Map<Integer,String> emptyMap = Map.of();
      Map<Integer,String> luckyNumber = Map.of(19, "Nineteen");
      Map<Integer,String> numberToWord =
           Map.of(1, "One", 2, "Two", 3, "Three");
      Map<String,String> days = Map.ofEntries(
           entry("Mon", "Monday"),
           entry("Tue", "Tuesday"),
           entry("Wed", "Wednesday"),
```

```
            entry("Thu", "Thursday"),
            entry("Fri", "Friday"),
            entry("Sat", "Saturday"),
            entry("Sun", "Sunday"));

        System.out.println("emptyMap = " + emptyMap);
        System.out.println("singletonMap = " + luckyNumber);
        System.out.println("numberToWord = " + numberToWord);
        System.out.println("days = " + days);

        try {
           // // Пробуем включить в отображение null
           Map<Integer,String> map = Map.of(1, null);
        } catch(NullPointerException e) {
           System.out.println("Наличие null в Map.of() запрещено.");
        }

        try {
           // Пробуем использовать повторяющиеся ключи
           Map<Integer,String> map = Map.of(1, "One", 1, "On");
        } catch(IllegalArgumentException e) {
           System.out.println(e.getMessage());
        }

        try {
           // Пробуем добавить элемент
           luckyNumber.put(8, "Eight");
        } catch(UnsupportedOperationException e) {
           System.out.println("Добавить элемент невозможно.");
        }

        try {
           // Пробуем удалить элемент
           luckyNumber.remove(0);
        } catch(UnsupportedOperationException e) {
           System.out.println("Удалить элемент невозможно.");
        }
    }
}
```

```
emptyMap = {}
singletonMap = {19=Nineteen}
numberToWord = {1=One, 3=Three, 2=Two}
days = {Sat=Saturday, Tue=Tuesday, Thu=Thursday, Sun=Sunday, Wed=Wednesday,
Fri=Friday, Mon=Monday}
Наличие null в Map.of() запрещено.
```

```
duplicate key: 1
Добавить элемент невозможно.
Удалить элемент невозможно.
```

Резюме

Поддержка литеральных коллекций в Java была очень востребована. Но вместо этого в JDK 9 модернизирован API коллекций – в интерфейсы `List`, `Set` и `Map` добавлены статические фабричные методы, возвращающие соответственно немодифицируемые списки, множества и отображения. Этот метод – он называется `of()` – перегружен и позволяет создавать коллекции, содержащие от 0 до 10 элементов. В интерфейсах `List` и `Set` имеется также вариант метода `of()`, принимающий переменное число аргументов и позволяющий создавать списки и множества с произвольным числом элементов. В интерфейсе `Map` имеется статический фабричный метод `ofEntries()` для создания немодифицируемого отображения с произвольным числом элементов. В интерфейсе `Map` определен также статический метод `entry()`, который принимает ключ и значение и возвращает экземпляр класса `Map.Entry`. Методы `ofEntries()` и `entry()` используются совместно.

Новые статические фабричные методы оптимизированы с целью достижения максимальной производительности. Методы `List.of()` и `Set.of()` не допускают элементов `null`. Метод `Set.of()` не допускает повторяющих элементов. Методы `Map.of()` и `Map.ofEntries()` не допускают повторяющихся ключей, а также задания `null` в качестве ключа или значения.

Глава 14

Клиентский API HTTP/2

Краткое содержание главы:

- что такое клиентский API HTTP/2;
- создание HTTP-клиентов;
- создание HTTP-запросов;
- получение HTTP-запросов;
- создание оконечных точек WebSocket;
- отправка данных от сервера клиенту без запроса.

В JDK 9 клиентский API HTTP/2 поставляется в инкубаторном модуле `jdk.incubator.httpclient`. Этот модуль экспортирует пакет `jdk.incubator.http`, содержащий весь открытый API. Инкубаторный модуль не является частью Java SE. В Java SE 10 он будет либо стандартизован и станет частью Java SE, либо исключен. На странице http://openjdk.java.net/jeps/11 подробно рассказывается об инкубаторных модулях в JDK.

Инкубаторные модули не разрешаются по умолчанию на этапах компиляции и выполнения, поэтому необходимо добавить `jdk.incubator.httpclient` в набор корневых модулей по умолчанию с помощью параметра `--add-modules`:

```
<javac|java|jmod...> -add-modules jdk.incubator.httpclient ...
```

Инкубаторный модуль разрешается, если его читает другой, разрешенный, модуль. В этой главе мы создадим модуль, который читает `jdk.incubator.httpclient`, так что параметр `--add-modules` не понадобится.

Поскольку API, предоставляемый инкубаторным модулем, не окончателен, при его использовании на этапе компиляции или выполнения в стандартный поток ошибок выводится предупреждение:

```
WARNING: Using incubator modules: jdk.incubator.httpclient
```

Имена инкубаторных модулей и пакетов, содержащих инкубаторный API, начинаются строкой `jdk.incubator`. После стандартизации и включения в Java SE имена будут приведены в соответствии с принятыми в Java соглашениями. Так, модуль `jdk.incubator.httpclient` может быть переименован в `java.httpclient`.

Поскольку модуль `jdk.incubator.httpclient` еще не вошел в Java SE, документации по нему вы не найдете. Я сгенерировал документацию в формате Javadoc и включил ее в исходный код, прилагаемый к книге. Вы можете обратиться к ней с помощью файла `Java9Revealed/jdk.incubator.httpclient/dist/javadoc/index.html`. Документация была сгенерирована по ознакомительной сборке 158. Может статься, что с тех пор API изменился и документацию нужно сгенерировать заново. Ниже описано, как это сделать:

1. В исходном коде к книге имеется проект NetBeans `jdk.incubator.httpclient`, находящийся в одноименном каталоге.
2. После установки JDK 9 исходный код оказывается в файле `src.zip` в установочном каталоге. Скопируйте содержимое каталога `jdk.incubator.httpclient` из архива `src.zip` в каталог `Java9revealed\jdk.incubator.httpclient\src` скачанного исходного кода к книге.
3. Откройте проект `jdk.incubator.httpclient` в NetBeans.
4. Щелкните правой кнопкой мыши по узлу проекта и выберите из контекстного меню команду **Generate Javadoc** (Создать документацию Java). Сообщения об ошибках и предупреждения можно игнорировать. Документация будет создана в каталоге `Java9Revealed/jdk.incubator.httpclient/dist/javadoc`. Чтобы получить доступ к документации по модулю `jdk.incubator.httpclient`, откройте файл `index.html` в этом каталоге.

Что такое клиентский API HTTP/2?

Протокол HTTP/1.1 поддерживался в Java, начиная с версии JDK 1.0. HTTP API включает ряд типов в пакете `java.net`. У существующего API имеются следующие проблемы:

- он проектировался для поддержки нескольких протоколов, в частности, http, ftp, gopher, и т. д., многие из которых уже не употребляются;
- он слишком абстрактный и трудный для использования;
- он содержит много недокументированных особенностей;
- поддерживается только блокирующий режим, и, следовательно, каждый запрос следует обрабатывать в отдельном потоке.

В мае 2015 Инженерный совет Интернет (IETF) опубликовал спецификацию HTTP/2, ее полный текст можно найти по адресу https://tools.ietf.org/html/rfc7540. В HTTP/2 не изменилась семантика на уровне приложения, т. е. все, что вы знали о протоколе HTTP и что использовали в своих приложениях, осталось в силе. Но стал эффективнее способ подготовки пакетов данных и передачи их по сети между клиентом и сервером. Заголовки, методы, коды состояния HTTP, URL-адреса и т. п. остались прежними. HTTP/2 стремится решить многочисленные проблемы производительности, характерные для соединений по протоколу HTTP/1.

- HTTP/2 поддерживает обмен двоичными данными, а не только текстовыми, как HTTP/1.1.
- HTTP/2 поддерживает мультиплексирование и конкурентность, т. е. по одному и тому же TCP-соединению можно одновременно передавать не-

сколько пакетов данных в обоих направлениях, и ответы на запросы могут приходить не по порядку. Тем самым исключаются накладные расходы на установление нескольких соединений между сторонами, что часто приходилось делать при работе с HTTP/1.1. В HTTP/1.1 ответы должны приходить в том порядке, в каком отправлялись запросы – это называется *блокирующей очередностью* (head-of-line blocking). А HTTP/2 эта проблема решена благодаря возможности мультиплексирования одного TCP-соединения.

- Клиент вправе внести предложение о приоритете запроса, которое может быть учтено или проигнорировано сервером.
- Заголовки HTTP сжимаются, что заметно уменьшает размер заголовка, а, значит, и задержку.
- Разрешена инициативная (unsolicited) отправка ресурсов сервером клиенту.

Вместо модификации существующего API для HTTP/1.1 в JDK 9 предложен новый клиентский API HTTP/2, поддерживающий обе версии протокола. Предполагается, что этот API в конечном итоге заменит старый. Новый API содержит также классы и интерфейсы для разработки клиентских приложений, основанных на протоколе WebSocket. Полную спецификацию протокола WebSocket см. по адресу `https://tools.ietf.org/html/rfc6455`. У нового клиентского API HTTP/2 имеются следующие преимущества по сравнению с существующим API:

- он прост для изучения и использования в большинстве типичных случаев;
- он предоставляет уведомления в виде событий. Например, генерируются уведомления при получении заголовков, получении тела и возникновении ошибки;
- он поддерживает инициативную отправку ресурсов сервером без явного запроса со стороны клиента. Это упрощает настройку обмена с серверами по протоколу WebSocket;
- он поддерживает протоколы HTTP/2 и HTTPS/TLS;
- он работает в синхронном (блокирующем) и асинхронном (неблокирующем) режиме.

Новый API содержит менее 20 типов, из которых четыре – основные. Остальные типы используются вместе с основными. Кроме того, в новом API используется несколько типов из старого. Новый API находится в пакете `jdk.incubator.http` из модуля `jdk.incubator.httpclient`. К основным типам относятся три абстрактных класса и один интерфейс:

- класс `HttpClient`
- класс `HttpRequest`
- класс `HttpResponse`
- интерфейс `WebSocket`

Экземпляром класса `HttpClient` является контейнер для хранения конфигурационных данных, который можно использовать для отправки нескольких HTTP-запросов, вместо того чтобы каждый раз настраивать все заново. Экземпляр класса `HttpRequest` представляет HTTP-запрос, который можно отправить серверу. Экземпляр класса `HttpResponse` представляет HTTP-ответ. Экземпляр интерфейса `WebSocket` представляет клиента WebSocket. Создать WebSocket-сервер можно с по-

мощью Java EE 7 WebSocket API. В конце главы я продемонстрирую создание клиента и сервера WebSocket.

Экземпляры `HttpClient`, `HttpRequest` и `WebSocket` создаются с помощью построителей. В каждом типе имеется вложенный класс или интерфейс с именем `Builder`, который служит для построения экземпляров этого типа. Отметим, что экземпляр `HttpResponse` не создается, а возвращается в ответ на отправленный HTTP-запрос. Использовать новый клиентский API HTTP/2 так просто, что для чтения HTTP-ресурса достаточно одного предложения! В следующем фрагменте ресурс с URL-адресом `https://www.google.com/` читается в виде строки в ответ на HTTP-запрос GET:

```
String responseBody = HttpClient.newHttpClient()
          .send(HttpRequest.newBuilder(new URI("https://www.google.com/"))
              .GET()
              .build(), BodyHandler.asString())
          .body();
```

Вот типичные шаги обработки HTTP-запроса:

- создать объект `HTTPClient` для хранения конфигурационной информации;
- создать объект `HttpRequest` и поместить в него информацию, которую нужно отправить серверу;
- отправить `HttpRequest` серверу;
- получить от сервера ответ в виде объекта `HttpResponse`;
- обработать ответ.

Настройка среды для примеров

В этой главе много примеров взаимодействия с сервером. Вместо того чтобы развертывать веб-приложение в Интернете, я создал в NetBeans проект, который вы можете развернуть локально. Если вы предпочитаете какое-то другое веб-приложение, то замените в примерах URL-адреса.

Веб-приложение, созданное в NetBeans, находится в каталоге `webapp` приложенного к книге исходного кода. Я тестировал примеры, развернув веб-приложение на сервере GlassFish 4.1.1 и Tomcat 8/9. Скачать NetBeans IDE с сервером GlassFish можно по адресу `https://netbeans.org/`. HTTP-прослушиватель в сервере GlassFish работает на порту 8080. Если вы зададите другой порт, то измените номер порта в URL-адресах, встречающихся в примерах.

Все клиентские программы из этой главы находятся в каталоге `com.jdojo.http.client` приложенного к книге кода. Они принадлежат модулю `com.jdojo.http.client`, объявление которого приведено в листинге 14.1.

Листинг 14.1. Объявление модуля com.jdojo.http.client

```
// module-info.java
module com.jdojo.http.client {
    requires jdk.incubator.httpclient;
}
```

Создание HTTP-клиентов

Отправляемый серверу HTTP-запрос необходимо сконфигурировать, сообщив серверу следующую информацию: какой использовать аутентификатор, детали конфигурации SSL, используемый диспетчер куков, сведения о прокси-сервере, политика перенаправления, если сервер перенаправит запрос, и т. д. Все эти конфигурационные данные хранятся в классе `HttpClient` и могут использоваться для нескольких запросов. Для отдельных запросов часть конфигурационных данных можно переопределить. При отправке HTTP-запроса необходимо указать объект `HttpClient`, содержащий конфигурационные данные для этого запроса. В объекте `HttpClient` хранятся следующие данные: аутентификатор, диспетчер куков, исполнитель, политика перенаправления, приоритет запроса, селектор прокси, контекст SSL, параметры SSL и версия HTTP.

Аутентификатор – это экземпляр класса `java.net.Authenticator`, используемый для аутентификации по протоколу HTTP. По умолчанию аутентификатор отсутствует.

Диспетчер куков управляет куками. Это экземпляр класса `java.net.CookieManager`. По умолчанию отсутствует.

Исполнитель – экземпляр интерфейса `java.util.concurrent.Executor`, используемый для асинхронной отправки HTTP-запросов и получения HTTP-ответов. Если исполнитель не задан, используется исполнитель по умолчанию.

Политика перенаправления – один из элементов перечисления `HttpClient.Redirect`, определяет, как обрабатывать полученное от сервера, указание о перенаправлении. По умолчанию подразумевается `NEVER`, т. е. клиент не переходит по адресу, указанному сервером.

Приоритет запроса – число от 1 до 256 (включительно), которое в HTTP/2 является пожеланием серверу о том, в каком порядке производить обработку запросов. Чем больше значение, тем выше приоритет.

Селектор прокси – экземпляр класса `java.net.ProxySelector`, решающий, какой прокси-сервер использовать. По умолчанию прокси-сервер не используется.

Контекст SSL – экземпляр класса `javax.net.ssl.SSLContext`, предоставляющего реализацию протокола SSL. Подразумеваемый по умолчанию объект `SSLContext` работает, когда не требуется согласовывать протоколы и не нужна аутентификация клиента.

Параметры SSL – это параметры соединений по протоколам SSL/TLS/DTLS. Они хранятся в экземпляре класса `javax.net.ssl.SSLParameters`.

Версия HTTP может быть равна 1.1 или 2. Задается в виде элемента перечисления `HttpClient.Version`: `HTTP_1_1` или `HTTP_2`. Означает, что по возможности следует использовать указанный протокол. По умолчанию подразумевается `HTTP_1_1`.

> **Совет.** Объект `HttpClient` неизменяемый. Часть конфигурационных данных, хранящихся в `HttpClient`, может быть переопределена при построении конкретного запроса.

Класс `HttpClient` абстрактный, поэтому создать его экземпляр непосредственно невозможно. Для получения объекта `HttpClient` есть два способа:

- статический метод `newHttpClient()` класса `HttpClient`;
- метод `build()` класса `HttpClient.Builder`.

Ниже показано, как получить объект `HttpClient` по умолчанию:

```
// Получить HttpClient по умолчанию
HttpClient defaultClient = HttpClient.newHttpClient();
```

Для создания объекта `HttpClient` можно также воспользоваться статическим методом `HttpClient.newBuilder()`, который возвращает экземпляр класса `HttpClient.Builder`. Этот класс содержит методы для задания всех конфигурационных данных. Конфигурационное значение задается в виде параметра метода, а метод возвращает ссылку на сам объект-построитель, так что вызовы нескольких методов можно соединять в цепочку. В следующем фрагменте создается объект `HttpClient`, в котором установлена политика перенаправления `ALWAYS` и версия `HTTP_2`:

```
// Создать специальный объект HttpClient
HttpClient httpClient = HttpClient.newBuilder()
          .followRedirects(HttpClient.Redirect.ALWAYS)
          .version(HttpClient.Version.HTTP_2)
          .build();
```

В классе `HttpClient` имеются методы для получения каждого конфигурационного значения:

- `Optional<Authenticator> authenticator()`
- `Optional<CookieManager> cookieManager()`
- `Executor executor()`
- `HttpClient.Redirect followRedirects()`
- `Optional<ProxySelector> proxy()`
- `SSLContext sslContext()`
- `Optional<SSLParameters> sslParameters()`
- `HttpClient.Version version()`

Поскольку класс `HttpClient` неизменяемый, методов установки в нем нет. Сам по себе объект `HttpClient` не используется. Для отправки запроса серверу необходим еще объект `HttpRequest`, о котором речь пойдет в следующем разделе. Класс `HttpClient` содержит следующие методы для отправки запроса серверу:

- `<T> HttpResponse<T> send(HttpRequest req, HttpResponse.BodyHandler<T> responseBodyHandle)`
- `<T> CompletableFuture<HttpResponse<T>> sendAsync(HttpRequest req, HttpResponse.BodyHandler<T> responseBodyHandler)`
- `<U,T> CompletableFuture<U> sendAsync(HttpRequest req, HttpResponse.MultiProcessor<U,T> multiProcessor)`

Метод `send()` отправляет запрос синхронно, а методы `sendAsync()` асинхронно. Подробнее они будут рассмотрены в следующих разделах.

Обработка HTTP-запросов

Клиентское приложение взаимодействует с веб-сервером путем отправки HTTP-запросов, на которые сервер присылает HTTP-ответы. HTTP-запрос представляется экземпляром класса `HttpRequest`. Обработка HTTP-запроса состоит из следующих шагов:

- получить построитель HTTP-запроса;
- задать параметры запроса;

- с помощью построителя создать HTTP-запрос;
- отправить HTTP-запрос серверу синхронно или асинхронно;
- обработать ответ от сервера.

Получение построителя HTTP-запроса

Чтобы создать объект `HttpRequest`, необходим построитель – экземпляр класса `HttpRequest.Builder`. Для получения объекта `HttpRequest.Builder` можно воспользоваться следующими статическими методами класса `HttpRequest`:

- `HttpRequest.Builder newBuilder()`
- `HttpRequest.Builder newBuilder(URI uri)`

Ниже показано, как эти методы используются:

```
// URI-адрес указывает на google
URI googleUri = new URI("http://www.google.com");

// Получить построитель для этого URI
HttpRequest.Builder builder1 = HttpRequest.newBuilder(googleUri);

// Получить построитель, не указывая URI
HttpRequest.Builder builder2 = HttpRequest.newBuilder();
```

Задание параметров HTTP-запроса

Имея построитель HTTP-запроса, мы можем задать различные параметры запроса. Все методы возвращают сам построитель, так что их можно сцеплять. Ниже перечислены эти методы:

- `HttpRequest.Builder DELETE(HttpRequest.BodyProcessor body)`
- `HttpRequest.Builder expectContinue(boolean enable)`
- `HttpRequest.Builder GET()`
- `HttpRequest.Builder header(String name, String value)`
- `HttpRequest.Builder headers(String... headers)`
- `HttpRequest.Builder method(String method, HttpRequest.BodyProcessor body)`
- `HttpRequest.Builder POST(HttpRequest.BodyProcessor body)`
- `HttpRequest.Builder PUT(HttpRequest.BodyProcessor body)`
- `HttpRequest.Builder setHeader(String name, String value)`
- `HttpRequest.Builder timeout(Duration duration)`
- `HttpRequest.Builder uri(URI uri)`
- `HttpRequest.Builder version(HttpClient.Version version)`

HTTP-запрос отправляется серверу с помощью объекта `HttpClient`. При построении объекта `HttpRequest` версию HTTP можно задать с помощью метода `version()` его построителя, если необходимо переопределить значение, заданное в `HttpClient`. Ниже показано, как установить версию 2.0 протокола HTTP для конкретного запроса, переопределив значение по умолчанию, заданное в объекте `HttpClient`:

```
// По умолчанию установлена версия HTTP 1.1. Она и будет использоваться, если
// не переопределена для конкретного запроса
```

```
HttpClient client = HttpClient.newHttpClient();

// URI-адрес, указывающий на google
URI googleUri = new URI("http://www.google.com");

// Получить HttpRequest с версией HTTP 2.0
HttpRequest request = HttpRequest.newBuilder(googleUri)
          .version(HttpClient.Version.HTTP_2)
          .build();

// Объект клиента содержит версию HTTP 1.1, а объект запроса – версию 2.0.
// В следующем предложении запрос посылается по протоколу HTTP 2.0, как
// указано в объекте HttpRequest.
HttpResponse<String> r = client.send(request, BodyHandler.asString());
```

Метод `timeout()` задает таймаут для запроса. Если ответ не получен в течение указанного периода, то возбуждается исключение `HttpTimeoutException`.

HTTP-запрос может содержать заголовок `expect`, имеющий значение `"100-Continue"`. Если заголовок присутствует, то клиент посылает серверу только заголовки и ожидается, что сервер вернет ошибку или ответ `100-Continue`. Получив этот ответ, клиент посылает серверу тело запроса. Клиент применяет эту технику, чтобы проверить, сможет ли сервер обработать запрос, прежде чем посылать само тело. По умолчанию этот заголовок не посылается. Чтобы это сделать, построитель запроса должен вызвать метод `expectContinue(true)`. Заметим, что вызвать метод `header("expect", "100-Continue")` для активации этого механизма недостаточно.

```
// Разрешить отправку заголовка expect=100-Continue в запросе
HttpRequest.Builder builder = HttpRequest.newBuilder()
                                        .expectContinue(true);
```

В следующих разделах описано, как задавать заголовки, тело и метод HTTP-запроса.

Задание заголовков запроса

Заголовком HTTP-запроса является пара имя-значение. Заголовков может быть несколько. Для добавления заголовков предназначены методы `header()`, `headers()` и `setHeader()` класса `HttpRequest.Builder`. Методы `header()` и `headers()` добавляют заголовки, если их еще нет в запросе, в противном случае ничего не делают. Метод `setHeader()` заменяет заголовок, если он присутствует, иначе добавляет заголовок.

Методы `header()` и `setHeader()` позволяют добавлять или изменять по одному заголовку, а метод `headers()` может добавить сразу несколько заголовков. Метод `headers()` принимает переменное число аргументов, которые должны быть чередующимися парами ключ-значение. Ниже показано, как задать заголовки HTTP-запроса:

```
   // Создать URI
URI calc = new URI("http://localhost:8080/webapp/Calculator");

// Использовать метод header()
```

```
HttpRequest.Builder builder1 = HttpRequest.newBuilder(calc)
      .header("Content-Type", "application/x-www-form-urlencoded")
      .header("Accept", "text/plain");

// Использовать метод headers()
HttpRequest.Builder builder2 = HttpRequest.newBuilder(calc)
      .headers("Content-Type", "application/x-www-form-urlencoded",
               "Accept", "text/plain");

// Использовать метод setHeader()
HttpRequest.Builder builder3 = HttpRequest.newBuilder(calc)
      .setHeader("Content-Type", "application/x-www-form-urlencoded")
      .setHeader("Accept", "text/plain");
```

Задание тела запроса

В некоторых HTTP-запросах, например POST и PUT, имеется тело. Тело HTTP-запроса устанавливается процессором тела – статическим вложенным интерфейсом `HttpRequest.BodyProcessor`, который содержит следующие статические фабричные методы, конструирующие тело запроса из различных источников, например `String`, `byte[]` или `File`:

- `HttpRequest.BodyProcessor fromByteArray(byte[] buf)`
- `HttpRequest.BodyProcessor fromByteArray(byte[] buf, int offset, int length)`
- `HttpRequest.BodyProcessor fromByteArrays(Iterable<byte[]> iter)`
- `HttpRequest.BodyProcessor fromFile(Path path)`
- `HttpRequest.BodyProcessor fromInputStream(Supplier<? extends InputStream> streamSupplier)`
- `HttpRequest.BodyProcessor fromString(String body)`
- `HttpRequest.BodyProcessor fromString(String s, Charset charset)`

Первый аргумент каждого метода указывает на источник данных для тела запроса. Например, метод `fromString(String body)` возвращает процессор тела в случае, когда тело запроса хранится в объекте типа `String`.

> **Совет.** В классе `HttpRequest` имеется статический метод `noBody()`, возвращающий объект `HttpRequest.BodyProcessor`, который обрабатывает запрос без тела. Обычно этот метод используется совместно с методом `method()`, когда HTTP-метод не предполагает наличия тела, но `method()` требует, чтобы процессор тела был передан.

Должно ли в запросе быть тело, зависит от HTTP-метода отправки запроса. В запросах `DELETE`, `POST` и `PUT` тело есть, в запросе `GET` – нет. В классе `HttpRequest.Builder` имеются методы с такими же именами, как у HTTP-методов; они служат для задания метода и тела запроса. Например, для отправки POST-запроса предназначен метод `POST(HttpRequest.BodyProcessor body)`.

Существует много HTTP-методов, например, `HEAD` и `OPTIONS`, для которых в классе `HttpRequest.Builder` нет соответствующего метода. Но имеется метод `method(String method, HttpRequest.BodyProcessor body)`, позволяющий задать любой HTTP-метод. Используя метод `method()`, не забывайте, что имя метода должно быть записано заглавными буквами: `GET`, `POST`, `HEAD` и т. д. Ниже приведен список таких методов:

- `HttpRequest.Builder DELETE(HttpRequest.BodyProcessor body)`
- `HttpRequest.Builder method(String method, HttpRequest.BodyProcessor body)`
- `HttpRequest.Builder POST(HttpRequest.BodyProcessor body)`
- `HttpRequest.Builder PUT(HttpRequest.BodyProcessor body)`

В следующем фрагменте тело HTTP-запроса читается из строки, обычно так поступают, когда на URL-адрес отправляется HTML-форма. В данном случае форма содержит три поля `n1`, `n2` и `op`.

```
URI calc = new URI("http://localhost:8080/webapp/Calculator");

// Создать строку, содержащую данные формы с полями n1 = 10, n2 = 20, op = +.
String formData = "n1=" + URLEncoder.encode("10","UTF-8") +
                  "&n2=" + URLEncoder.encode("20","UTF-8") +
                  "&op=" + URLEncoder.encode("+","UTF-8") ;

HttpRequest.Builder builder = HttpRequest.newBuilder(calc)
    .header("Content-Type", "application/x-www-form-urlencoded")
    .header("Accept", "text/plain")
    .POST(HttpRequest.BodyProcessor.fromString(formData));
```

Создание HTTP-запроса

В предыдущих разделах мы узнали, как создать объект `HttpRequest.Builder` и использовать его для задания различных свойств HTTP-запроса. Для создания же HTTP-запроса нужно просто вызвать метод `build()` объекта `HttpRequest.Builder`. В следующем фрагменте создается объект `HttpRequest`, в котором задан HTTP-метод `GET`:

```
HttpRequest request = HttpRequest.newBuilder()
                                 .uri(new URI("http://www.google.com"))
                                 .GET()
                                 .build();
```

Ниже создается объект `HttpRequest` с заголовками, телом и HTTP-методом `POST`:

```
// Задать URI и данные формы
URI calc = new URI("http://localhost:8080/webapp/Calculator");
String formData = "n1=" + URLEncoder.encode("10","UTF-8") +
                  "&n2=" + URLEncoder.encode("20","UTF-8") +
                  "&op=" + URLEncoder.encode("+","UTF-8");

// Построить объект HttpRequest
HttpRequest request = HttpRequest.newBuilder(calc)
    .header("Content-Type", "application/x-www-form-urlencoded")
    .header("Accept", "text/plain")
    .POST(HttpRequest.BodyProcessor.fromString(formData))
    .build();
```

Отметим, что создание объекта `HttpRequest` еще не приводит к его отправке серверу. Для отправки следует вызвать метод `send()` или `sendAsync()` класса `HttpClient`. Об этом пойдет речь в следующем разделе.

Ниже создается объект HttpRequest, в котором задан HTTP-метод HEAD. Обратите внимание, что для задания метода используется метод method() класса HttpRequest.Builder.

```
HttpRequest request =
    HttpRequest.newBuilder(new URI("http://www.google.com"))
               .method("HEAD", HttpRequest.noBody())
               .build();
```

Обработка HTTP-ответов

Имея объект HttpRequest, мы можем послать запрос серверу и получить ответ синхронно или асинхронно. Ответ сервера представляется экземпляром класса HttpResponse<T>, где T – тип тела ответа, например: String, byte[] или Path. Для отправки запроса и получения ответа предназначены следующие методы класса HttpRequest:

- <T> HttpResponse<T> send(HttpRequest req, HttpResponse.BodyHandler<T> responseBodyHandler)
- <T> CompletableFuture<HttpResponse<T>> sendAsync(HttpRequest req, HttpResponse.BodyHandler<T> responseBodyHandler)
- <U,T> CompletableFuture<U> sendAsync(HttpRequest req, HttpResponse.MultiProcessor<U,T> multiProcessor)

Метод send() синхронный, т. е. блокирует выполнение программы до получения ответа. Метод sendAsync() асинхронный, он немедленно возвращает объект CompletableFuture<HttpResponse>, который станет завершенным, когда ответ будет готов к обработке.

Обработка состояния и заголовков ответа

HTTP-ответ содержит код состояния, заголовки и тело. Объект HttpResponse становится доступен сразу после получения кода состояния и заголовков, но до получения тела. Метод statusCode() класса HttpResponse возвращает код состояния в виде числа типа int. Метод headers() возвращает заголовки ответа в виде экземпляра интерфейса HttpHeaders. Интерфейс HttpHeaders содержит следующие методы для получения значений заголовка с указанным именем или всех заголовков в виде Map<String,List<String>>:

- List<String> allValues(String name)
- Optional<String> firstValue(String name)
- Optional<Long> firstValueAsLong(String name)
- Map<String,List<String>> map()

В листинге 14.2 приведен полный текст программы, которая отправляет HEAD-запрос на URL http://www.google.com и печатает код состояния и заголовки полученного ответа.

Листинг 14.2. Обработка кода состояния и заголовков ответа

```
// GoogleHeadersTest.java
```

```
package com.jdojo.http.client;

import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;
import jdk.incubator.http.HttpClient;
import jdk.incubator.http.HttpRequest;
import jdk.incubator.http.HttpResponse;

public class GoogleHeadersTest {
   public static void main(String[] args) {
      try {
         URI googleUri = new URI("http://www.google.com");
         HttpClient client = HttpClient.newHttpClient();

         HttpRequest request =
            HttpRequest.newBuilder(googleUri)
                       .method("HEAD", HttpRequest.noBody())
                       .build();

         HttpResponse<?> response =
            client.send(request, HttpResponse.BodyHandler.discard(null));

         // Напечатать код состояния и заголовки ответа
         Print the response status code and headers
         System.out.println("Код состояния ответа:" +
                            response.statusCode());

         System.out.println("Заголовки ответа:");
         response.headers()
                 .map()
                 .entrySet()
                 .forEach(System.out::println);
      } catch (URISyntaxException | InterruptedException |
               IOException e) {
         e.printStackTrace();
      }
   }
}
```

```
WARNING: Using incubator modules: jdk.incubator.httpclient
Код состояния ответа:200
Заголовки ответа:
accept-ranges=[none]
cache-control=[private, max-age=0]
content-type=[text/html; charset=ISO-8859-1]
date=[Sun, 26 Feb 2017 16:39:36 GMT]
expires=[-1]
p3p=[CP="This is not a P3P policy! See https://www.google.com/support/accounts/
answer/151657?hl=en for more info."]
server=[gws]
```

```
set-cookie=[NID=97=Kmz52m8Zdf4lsNDsnMyrJomx_2kD7lnWYcNEuwPWsFTFUZ7yli6DbCB98Wv-
SlxOfKA0OoOBIBgysuZw3ALtgJjX67v7-mC5fPv88n8VpwxrNcjVGCfFrxVro6gRNIrye4dAWZvUVfY28eOM;
expires=Mon, 28-Aug-2017 16:39:36 GMT; path=/; domain=.google.com; HttpOnly]
transfer-encoding=[chunked]
vary=[Accept-Encoding]
x-frame-options=[SAMEORIGIN]
x-xss-protection=[1; mode=block]
```

Обработка тела ответа

Обработка тела HTTP-ответа состоит из двух шагов.

- В момент отправки запроса методом `send()` или `sendAsync()` класса `HttpClient` необходимо задать обработчик тела ответа – экземпляр интерфейса `HttpResponse.BodyHandler<T>`.
- Когда будут получены код состояния и заголовки, вызывается метод `apply()` этого обработчика. Методу `apply()` передаются код состояния и заголовки, а он возвращает экземпляр интерфейса `HttpResponse.BodyProcessor<T>`, который читает тело ответа и преобразует данные к типу `T`.

Не забивайте голову деталями обработки тела ответа. Предоставляется несколько реализаций интерфейса `HttpResponse.BodyHandler<T>`. Для получения экземпляра интерфейса для разных параметрических типов `T` имеются следующие статические фабричные методы:

- `HttpResponse.BodyHandler<byte[]> asByteArray()`
- `HttpResponse.BodyHandler<Void> asByteArrayConsumer(Consumer<Optional<byte[]>> consumer)`
- `HttpResponse.BodyHandler<Path> asFile(Path file)`
- `HttpResponse.BodyHandler<Path> asFile(Path file, OpenOption... openOptions)`
- `HttpResponse.BodyHandler<Path> asFileDownload(Path directory, OpenOption... openOptions)`
- `HttpResponse.BodyHandler<String> asString()`
- `HttpResponse.BodyHandler<String> asString(Charset charset)`
- `<U> HttpResponse.BodyHandler<U> discard(U value)`

Сигнатуры методов ясно дают понять, тело какого типа они обрабатывают. Например, если тело ответа требуется получить в виде строки, то для получения обработчика тела следует воспользоваться методом `asString()`. Метод `discard(U value)` возвращает обработчик, который просто отбрасывает тело ответа и возвращает вместо него переданное значение `value`.

Метод `body()` класса `HttpResponse<T>` возвращает тело ответа в виде объекта типа `T`.

В следующем фрагменте на URL-адрес `http://google.com` отправляется GET-запрос, а ответ преобразуется в строку. Обработка ошибок опущена.

```
import java.net.URI;
import jdk.incubator.http.HttpClient;
import jdk.incubator.http.HttpRequest;
import jdk.incubator.http.HttpResponse;
import static jdk.incubator.http.HttpResponse.BodyHandler.asString;
...
```

```
// Построить запрос
HttpRequest request = HttpRequest.newBuilder()
                  .uri(new URI("http://google.com"))
                  .GET()
                  .build();

// Отправить запрос и получить ответ
HttpResponse<String> response = HttpClient.newHttpClient()
                                          .send(request, asString());

// Получить и напечатать тело ответа
String body = response.body();
System.out.println(body);
```

```
WARNING: Using incubator modules: jdk.incubator.httpclient
<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="http://www.google.com/">here</A>.
</BODY></HTML>
```

В примере возвращен ответ с кодом состояния 301, означающим, что ресурс перемещен. В теле ответа содержится новый URL-адрес. Если бы в объекте `HttpClient` была установлена политика перенаправления `ALWAYS`, то запрос был бы автоматически отправлен на новый `URL`. Ниже показано, как это сделать:

```
// Запрос будет перенаправляться на URL, указанный сервером
HttpResponse<String> response = HttpClient.newBuilder()
    .followRedirects(HttpClient.Redirect.ALWAYS)
    .build()
    .send(request, asString());
```

В листинге 14.3 приведен полный текст программы, которая отправляет POST-запрос, имеющий тело, и асинхронно обрабатывает ответ. Веб-приложение в прилагаемом к книге исходном коде содержит сервлет `Calculator`, исходный код которого не приводится для экономии места. Сервлет принимает три параметра: `n1`, `n2` и `op`, где `n1` и `n2` – числа, а `op` – оператор (+, –, * или /). В ответ возвращается простой текст, содержащий оба операнда, оператор и результат операции. URL в тексте программы задан в предположении, что сервлет развернут локально, а веб-сервер прослушивает порт 8080. Если это не так, внесите соответствующие изменения. Показанный результат будет получен, если сервлет вызван успешно, иначе результат будет иным.

Листинг 14.3. Асинхронная обработка тела HTTP-ответа

```
// CalculatorTest.java
```

```
package com.jdojo.http.client;

import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;
import java.net.URLEncoder;
import jdk.incubator.http.HttpClient;
import jdk.incubator.http.HttpRequest;
import static jdk.incubator.http.HttpRequest.BodyProcessor.fromString;
import jdk.incubator.http.HttpResponse;

public class CalculatorTest {
   public static void main(String[] args) {
      try {
         URI calcUri =
            new URI("http://localhost:8080/webapp/Calculator");

         String formData = "n1=" + URLEncoder.encode("10","UTF-8") +
                           "&n2=" + URLEncoder.encode("20","UTF-8") +
                           "&op=" + URLEncoder.encode("+","UTF-8") ;

         // Создать запрос
         HttpRequest request = HttpRequest.newBuilder()
            .uri(calcUri)
            .header("Content-Type", "application/x-www-form-urlencoded")
            .header("Accept", "text/plain")
            .POST(fromString(formData))
            .build();

         // Обработать запрос асинхронно. После получения ответа
         // будет вызван метод processResponse() этого класса
         HttpClient.newHttpClient()
                   .sendAsync(request, HttpResponse.BodyHandler.asString())
                   .whenComplete(CalculatorTest::processResponse);
         try {
            // Текущий поток засыпает на 5 секунд, чтобы
            // завершилась асинхронная обработка ответа
            Thread.sleep(5000);
         } catch (InterruptedException ex) {
            ex.printStackTrace();
         }
      } catch (URISyntaxException | IOException e) {
         e.printStackTrace();
      }
   }

   private static void processResponse(HttpResponse<String> response,
```

```
                                        Throwable t) {
      if (t == null ) {
         System.out.println("Код состояния ответа: " +
                            response.statusCode());
         System.out.println("Тело ответа: " + response.body());
      } else {
         System.out.println("Произошло исключение во время " +
         "обработки HTTP-запроса. Ошибка: " + t.getMessage());
      }
   }
}
```

```
WARNING: Using incubator modules: jdk.incubator.httpclient
Код состояния ответа: 200
Тело ответа: 10 + 20 = 30.0
```

Использование обработчика тела ответа экономит время программиста. В одном предложении можно скачать и сохранить в файле содержимое ресурса. В следующем фрагменте содержимое ресурса по адресу `http://www.google.com` сохраняется в файле `google.html` в текущем каталоге. По завершении загрузки печатается путь к скачанному файлу. В случае ошибки печатается трасса стека.

```
HttpClient.newBuilder()
          .followRedirects(HttpClient.Redirect.ALWAYS)
          .build()
          .sendAsync(HttpRequest.newBuilder()
                                .uri(new URI("http://www.google.com"))
                                .GET()
                                .build(),
                                asFile(Paths.get("google.html")))
          .whenComplete((HttpResponse<Path> response,
                         Throwable exception) -> {
               if(exception == null) {
                   System.out.println("Сохранено в файле " +
                                      response.body().toAbsolutePath());
               } else {
                   exception.printStackTrace();
               }
           });
```

Обработка концевиков запроса

HTTP-концевики – это список пар имя-значение, аналогичный HTTP-заголовкам, но отправляемый сервером в конце HTTP-ответа, после того как тело отправлено. Многие серверы не используют концевики вовсе. В классе `HttpResponse`

имеется метод `trailers()`, который возвращает концевики в виде объекта `CompletableFuture<HttpHeaders>`. Обратите внимание на тип, которым параметризован этот объект, – `HttpHeaders`. В клиентском API HTTP/2 нет типа `HttpTrailers`. Прежде чем получать концевики, необходимо получить тело ответа. На момент написания книги клиентский API HTTP/2 не поддерживал обработку концевиков. В следующем фрагменте показано, как можно будет напечатать все концевики запроса, когда API начнет это поддерживать:

```
// Получить HTTP-ответ
HttpResponse<String> response = HttpClient.newBuilder()
          .followRedirects(HttpClient.Redirect.ALWAYS)
          .build()
          .send(HttpRequest.newBuilder()
                           .uri(new URI("http://www.google.com"))
                           .GET()
                           .build(),
                           asString());
// Прочитать тело ответа
String body = response.body();

// Обработать концевики
response.trailers()
        .whenComplete((HttpHeaders trailers, Throwable t) -> {
if(t == null) {
trailers.map()
        .entrySet()
        .forEach(System.out::println);
} else {
    t.printStackTrace();
}
         });
```

Задание политики перенаправления запроса

В ответ на HTTP-запрос веб-сервер может вернуть код состояния 3XX, где X – цифра от 0 до 9. Этот код означает, что для завершения запроса требуются дополнительные действия со стороны клиента. Например, код состояния 301 означает, что ресурс постоянно перемещен на другой адрес. В теле ответа содержатся альтернативные адреса. По умолчанию после получения кода 3XX запрос не перенаправляется на новый адрес. Но есть возможность задать политику, определяющую действия объекта `HttpClient` в этом случае. Политика задается в виде элемента перечисления `HttpClient.Redirect`:

- `ALWAYS`
- `NEVER`
- `SAME_PROTOCOL`
- `SECURE`

ALWAYS означает, что нужно всегда перенаправлять запрос в соответствии с указаниями сервера.

NEVER означает, что перенаправлять никогда не нужно. Это политика по умолчанию.

SAME_PROTOCOL означает, что перенаправлять нужно, когда в старом и в новом адресе одинаковый протокол – HTTP в HTTP, HTTPS в HTTPS.

SECURE означает, что перенаправлять нужно всегда за исключением случая, когда в старом адресе указан протокол HTTPS, а в новом – HTTP.

О том, как использовать политику перенаправления, было рассказано выше.

Использование протокола WebSocket

Протокол WebSocket обеспечивает двусторонний обмен данными между двумя оконечными точками: клиентской и серверной. Термином *оконечная точка* обозначаются обе стороны соединения по протоколу WebSocket. Клиентская оконечная точка инициирует соединение, а серверная принимает его. Поскольку соединение двустороннее, серверная оконечная точка может отправлять сообщения клиентской по собственной инициативе. В этом контексте встречается и другой термин: *равноправный участник*, или *пир* (*peer*). Пир – это просто другая сторона соединения. Так, для клиентской оконечной точки пиром является серверная, а для серверной – клиентская. *Сеанс* WebSocket – это последовательность взаимодействий между концевой точкой и одним пиром.

В протоколе WebSocket можно выделить три фазы:

- начальное квитирование;
- обмен данными;
- завершающее квитирование.

Клиент инициирует начальное квитирование. Процедура начинается по протоколу HTTP отправкой запроса о переходе на протокол WebSocket. Сервер отвечает согласием на переход. После того как квитирование успешно завершено, клиент и сервер обмениваются сообщениями. Обмен может быть инициирован как клиентом, так и сервером. Затем любая оконечная точка может отправить сообщение о завершении сеанса, и другая сторона отвечает таким же сообщением. По окончании завершающего квитирования соединение WebSocket закрывается.

Клиентский API HTTP/2 в JDK 9 поддерживает создание клиентских оконечных точек WebSocket. В полном примере работы с протоколом WebSocket необходимы обе оконечные точки: серверная и клиентская. В следующих разделах рассматриваются обе.

Создание серверной оконечной точки

Для создания серверной оконечной точки необходимо издание Java EE, а эта книга посвящена SE 9. Я кратко объясню, как создать серверную оконечную точку для использования в этом разделе. Не вдаваясь в детали, просто скажу, что для этой цели применяются аннотации Java EE 7.

В листинге 14.4 приведен код класса `TimeServerEndPoint`. Этот класс – часть веб-приложения в каталоге `webapp` прилагаемого к книге исходного кода. При раз-

вертывании веб-приложения на веб-сервере этот класс становится серверной оконечной точкой.

Листинг 14.4. Серверная оконечная точка WebSocket

```
// TimeServerEndPoint.java
package com.jdojo.ws;

import java.io.IOException;
import java.time.ZonedDateTime;
import java.util.concurrent.TimeUnit;
import javax.websocket.CloseReason;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;
import static javax.websocket.CloseReason.CloseCodes.NORMAL_CLOSURE;

@ServerEndpoint("/servertime")
public class TimeServerEndPoint {
   @OnOpen
   public void onOpen(Session session) {
      System.out.println("Клиент подключен. ");
   }

   @OnClose
   public void onClose(Session session) {
      System.out.println("Соединение закрыто.");
   }

   @OnError
   public void onError(Session session, Throwable t) {
      System.out.println("Произошла ошибка:" + t.getMessage());
   }

   @OnMessage
   public void onMessage(String message, Session session) {
      System.out.println("Клиент: " + message);

      // Отправить сообщение клиенту
      sendMessages(session);
   }

   private void sendMessages(Session session) {
      /* Запустить новый поток и отправить клиенту 3 сообщения.
         Каждое сообщение содержит текущую дату и время с часовым поясом.
      */
```

```
        new Thread(() -> {
            for(int i = 0; i < 3; i++) {
                String currentTime = ZonedDateTime.now().toString();
                try {
                    session.getBasicRemote()
                           .sendText(currentTime, true);
                    TimeUnit.SECONDS.sleep(5);
                } catch(InterruptedException | IOException e) {
                    e.printStackTrace();
                    break;
                }
            }

            try {
                // Закрываем WebSocket
                session.close(new CloseReason(NORMAL_CLOSURE,
                                              "Done"));
            } catch (IOException e) {
                e.printStackTrace();
            }
        })
        .start();
    }
}
```

Благодаря использованию аннотации `@ServerEndpoint("/servertime")` класс `TimeServerEndPoint` становится серверной оконечной точкой при развертывании в веб-сервере. Значение `/servertime` в аннотации определяет URL-адрес, по которому эта оконечная точка будут опубликована сервером.

Класс содержит четыре метода, снабженных аннотациями `@onOpen`, `@onMessage`, `@onClose` и `@onError`. Сами методы я назвал так же, как аннотации. Эти методы вызываются в различные моменты жизненного цикла серверной оконечной точки. В качестве аргумента им передается объект `Session`, представляющий взаимодействие этой оконечной точки со своим пиром – клиентской оконечной точкой.

Метод `onOpen()` вызывается после успешного начального квитирования и печатает сообщение о том, что клиент подключился.

Метод `onMessage()` вызывается при получении сообщения от пира. Он печатает сообщение и вызывает закрытый метод `sendMessages()`, который запускает новый поток и посылает пиру три сообщения. После отправки каждого сообщения поток спит 3 секунды. Сообщение содержит текущую дату и время с часовым поясом. Сообщения пирам можно посылать синхронно и асинхронно. Для отправки сообщения необходимо иметь ссылку на интерфейс `RemoteEndpoint`, представляющий диалог с пиром. Для ее получения предназначены методы `getBasicRemote()` и `getAsyncRemote()` объекта `Session`, которые возвращают соответственно экземпляры `RemoteEndpoint.Basic` и `RemoteEndpont.Async`, умеющие посылать сообщения синхронно и асинхронно. Имея ссылку на пира (удаленную оконечную точку), мы можем вызывать методы `sendXxx()` для отправки ему данных различных типов.

```
// Отправить пиру синхронное текстовое сообщение
```

```
session.getBasicRemote()
       .sendText(currentTime, true);
```

Второй аргумент метода `sendText()` говорит, является ли эта часть многочастного сообщения последней. Если сообщение состоит всего из одной части, следует передать `true`.

После отправки пиру всех сообщений вызывается метод `sendClose()`, который отправляет сообщение о закрытии. Метод принимает объект класса `CloseReason`, который инкапсулирует код и причину закрытия. Получив сообщение о закрытии, пир должен ответить таким же сообщением, после чего соединение WebSocket закрывается. Отметим, что после отправки сообщения о закрытии серверная оконечная точка не должна посылать пиру никаких сообщений.

Метод `onError()` вызывается в случае ошибки, не обработанной протоколом WebSocket.

Использовать эту оконечную точку саму по себе невозможно. Необходима клиентская оконечная точка, создание которой подробно рассматривается в следующем разделе. А работа этой оконечной точки будет продемонстрирована в разделе «Выполнение программы» ниже.

Создание клиентской оконечной точки

При разработке клиентской оконечной точки WebSocket используется интерфейс `WebSocket`, входящий в состав клиентского API HTTP/2 в JDK 9. Этот интерфейс содержит следующие вложенные типы:

- `WebSocket.Builder`
- `WebSocket.Listener`
- `WebSocket.MessagePart`

Экземпляр интерфейса `WebSocket` представляет клиентскую оконечную точку WebSocket. Для создания экземпляра применяется построитель – экземпляр интерфейса `WebSocket.Builder`. Экземпляр `WebSocket.Builder` возвращается методом `newWebSocketBuilder(URI uri, WebSocket.Listener listener)` класса `HttpClient`.

Когда на клиентской оконечной точке происходит событие, например, завершение начального квитирования, поступление сообщения, завершающее квитирование и т. д., посылается уведомление прослушивателю – экземпляру интерфейса `WebSocket.Listener`. Этот интерфейс содержит по одному методу по умолчанию для каждого типа уведомления. Написать класс, реализующий этот интерфейс, должны вы. Но реализовывать необходимо только методы, соответствующие событиям, для которых вы хотите получать уведомления. Прослушиватель указывается при создании экземпляра `WebSocket`.

Отправляя пиру сообщение о закрытии, можно задать код состояния. В интерфейсе `WebSocket` определены для этого следующие константы:

- `CLOSED_ABNORMALLY`: сообщение о закрытии с кодом состояния 1006, означающим, что соединение закрыто аномально, например, без отправки или получения сообщения о закрытии;
- `NORMAL_CLOSURE`: сообщение о закрытии с кодом состояния 1000, означающим, что соединение закрыто нормально, т. е. цель, для которого оно открывалось, достигнута.

Серверная оконечная точка может посылать многочастные сообщения. Каждая часть помечается признаком: первое, промежуточное, последнее или полное сообщение. В перечислении `WebSocket.MessagePart` определены соответствующие константы: `FIRST`, `PART`, `LAST`, `WHOLE`. Эти значения будут получены вместе с сообщением, когда прослушиватель получит уведомление о поступлении сообщения.

В следующих разделах подробно рассматриваются все этапы подготовки клиентской оконечной точки.

Создание прослушивателя

Прослушиватель – это экземпляр интерфейса `WebSocket.Listener`. Для его создания необходимо написать класс, реализующий этот интерфейс. Интерфейс содержит следующие методы:

- `CompletionStage<?> onBinary(WebSocket webSocket, ByteBuffer message, WebSocket.MessagePart part)`
- `CompletionStage<?> onClose(WebSocket webSocket, int statusCode, String reason)`
- `void onError(WebSocket webSocket, Throwable error)`
- `void onOpen(WebSocket webSocket)`
- `CompletionStage<?> onPing(WebSocket webSocket, ByteBuffer message)`
- `CompletionStage<?> onPong(WebSocket webSocket, ByteBuffer message)`
- `CompletionStage<?> onText(WebSocket webSocket, CharSequence message, WebSocket.MessagePart part)`

Метод `onOpen()` вызывается, когда клиентская оконечная точка подключается к пиру, ссылка на который передается методу в первом аргументе. Реализация по умолчанию запрашивает одно сообщение, т. е. прослушиватель может получить еще только одно сообщение. Запрос сообщений производится методом `request(long n)` интерфейса `WebSocket`:

```
// Разрешить получение еще одного сообщения
webSocket.request(1);
```

> **Совет.** Если сервер отправляет больше сообщений, чем запрошено, то сообщения накапливаются в очереди TCP-соединения, и в конечном итоге это приведет к тому, что механизм управления потоком, встроенный в TCP, запретит серверу отправлять последующие сообщения. Важно вызывать метод `request(long n)` в подходящее время с подходящим аргументом, чтобы прослушиватель продолжал получать сообщения от сервера.

Метод `onClose()` вызывается, когда оконечная точка получает от пира сообщение о закрытии. Это последнее уведомление прослушивателя. Возбуждаемые этим методом исключения игнорируются. Реализация по умолчанию не делает ничего. Как правило, пиру необходимо отправить свое сообщение о закрытии, чтобы довести до конца завершающее квитирование.

Метод `onPing()` вызывается, когда оконечная точка получает от пира сообщение `Ping`. Такое сообщение может послать как клиент, так и сервер. Реализация по умолчанию отправляет в ответ сообщение `Pong` с такой же полезной нагрузкой, как в полученном сообщении.

Метод `onPong()` вызывается, когда оконечная точка получает от пира сообщение `Pong`. Обычно такое сообщение приходит в ответ на ранее отправленное сообще-

ние Ping, но может быть также получено инициативное сообщение Pong. Реализация метода onPong() по умолчанию запрашивает еще одно сообщение для прослушивателя и больше ничего не делает.

Метод onError() вызывается в случае ошибки ввода-вывода или ошибки протокола WebSocket. Возбуждаемые этим методом исключения игнорируются. После вызова этого метода прослушиватель не получает больше никаких уведомлений. Реализация по умолчанию не делает ничего.

Методы onBinary() и onText() вызываются, когда от пира приходит соответственно двоичное или текстовое сообщение. Не забывайте проверять последний аргумент метода, который указывает позицию сообщения в потоке. Если приходят части сообщения, то из них необходимо собрать полное сообщение. Если любой из этих методов возвращает null, значит, обработка сообщения завершена. В противном случае возвращается объект CompletionStage<?>, который становится завершенным, когда завершится обработка сообщения.

В следующем фрагменте создается прослушиватель WebSocket, готовый к приему текстовых сообщений:

```
WebSocket.Listener listener = new WebSocket.Listener() {
   @Override
   public CompletionStage<?> onText(WebSocket webSocket,
                                    CharSequence message,
                                    WebSocket.MessagePart part) {

      // Разрешить получение прослушивателем одного сообщения
      webSocket.request(1);

      // Напечатать сообщение, полученное от сервера
      System.out.println("Сервер: " + message);

      // Вернуть null, показав, что сообщение обработано полностью
      return null;
   }
};
```

Построение оконечной точки

Мы должны построить экземпляр интерфейса WebSocket, выступающий в роли клиентской оконечной точки. Этот экземпляр подключается к серверной оконечной точке и обменивается с ней сообщениями. Для построения экземпляра WebSocket служит интерфейс WebSocket.Builder. Построитель возвращается следующим методом класса HttpClient:

```
WebSocket.Builder newWebSocketBuilder(URI uri, WebSocket.Listener listener)
```

Экземпляр HttpClient, от которого получен построитель WebSocket, предоставляет необходимые для подключения конфигурационные данные. В аргументе uri передается URI серверной оконечной точки, а в аргументе listener прослушиватель для строящейся оконечной точки (см. предыдущий раздел). Имея построитель, мы можем вызывать его методы для конфигурирования оконечной точки:

- `WebSocket.Builder connectTimeout(Duration timeout)`
- `WebSocket.Builder header(String name, String value)`
- `WebSocket.Builder subprotocols(String mostPreferred, String... lesserPreferred)`

Метод `connectTimeout()` задает таймаут начального квитирования. Если в течение указанного времени начальное квитирование не будет доведено до конца, то объект `CompletableFuture`, возвращенный методом `buildAsync()` интерфейса `WebSocket.Builder`, завершается с исключением `HttpTimeoutException`. Мы можем добавить собственные заголовки для процедуры начального квитирования с помощью метода `header()`. Метод `subprotocols()` позволяет специфицировать запрос на указанные подпротоколы в процессе начального квитирования – лишь один из них будет выбран сервером. Подпротоколы определяются приложением. Клиент и сервер должны согласовать между собой подпротоколы и их параметры.

Наконец, метод `buildAsync()` интерфейса `WebSocket.Builder` строит оконечную точку и возвращает объект `CompletableFuture<WebSocket>`, который нормально завершается, когда эта оконечная точка оказывается подключенной к серверной; в случае ошибки будущий объект завершается с исключением. В следующем фрагменте показано, как построить и подключить клиентскую оконечную точку. Отметим, что URI сервера имеет схему `ws`, зарезервированную для протокола WebSocket.

```
URI serverUri = new URI("ws://localhost:8080/webapp/servertime");

// Получить прослушиватель
WebSocket.Listener listener = ...;

// Построить оконечную точку, используя HttpClient по умолчанию
HttpClient.newHttpClient()
          .newWebSocketBuilder(serverUri, listener)
          .buildAsync()
          .whenComplete((WebSocket webSocket, Throwable t) -> {
             // Здесь должен быть ваш код
  });
```

Отправка сообщений пиру

После подключения клиентской оконечной точки к пиру можно начинать обмен сообщениями. Клиентская оконечная точка представлена экземпляром интерфейса `WebSocket`, который содержит следующие методы для отправки сообщений:

- `CompletableFuture<WebSocket> sendBinary(ByteBuffer message, boolean isLast)`
- `CompletableFuture<WebSocket> sendClose()`
- `CompletableFuture<WebSocket> sendClose(int statusCode, String reason)`
- `CompletableFuture<WebSocket> sendPing(ByteBuffer message)`
- `CompletableFuture<WebSocket> sendPong(ByteBuffer message)`
- `CompletableFuture<WebSocket> sendText(CharSequence message)`
- `CompletableFuture<WebSocket> sendText(CharSequence message, boolean isLast)`

Метод `sendText()` служит для отправки пиру текстового сообщения. Для отправки части сообщения используется вариант с двумя аргументами. Если второй аргумент равен `false`, значит, это промежуточная часть многочастного сообщения, а

если `true`, то последняя часть. Если ранее не отправлялись части этого сообщения, то `true` во втором аргументе означает, что это полное сообщение. Вспомогательный метод `sendText(CharSequence message)` вызывает второй вариант, передавая `true` во втором аргументе.

Метод `sendBinary()` отправляет пиру двоичное сообщение.

Методы `sendPing()` и `sendPong()` отправляют пиру соответственно сообщения `Ping` и `Pong`.

Метод `sendClose()` отправляет пиру сообщение `Close`. Это сообщение можно отправить в ответ на аналогичное сообщение от пира в процессе завершающего квитирования или инициативно, чтобы начать процедуру завершающего квитирования.

> **Совет.** Для немедленного закрытия соединения WebSocket предназначен метод `abort()` интерфейса `WebSocket`.

Выполнение программы

Настало время посмотреть, как серверная и клиентская оконечная точка WebSocket обмениваются сообщениями. В листинге 14.5 приведен код класса `WebSocketClient`, инкапсулирующего клиентскую оконечную точку. Предполагается, что он будет использоваться следующим образом:

```
// Создать клиентскую оконечную точку WebSocket
WebSocketClient wsClient = new WebSocketClient(new URI("<server-uri>"));

// Подключиться к серверу и начать обмен сообщениями
wsClient.connect();
```

Листинг 14.5. Класс, инкапсулирующий клиентскую оконечную точку

```
// WebSocketClient.java
package com.jdojo.http.client;

import java.net.URI;
import java.util.concurrent.CompletionStage;
import jdk.incubator.http.HttpClient;
import jdk.incubator.http.WebSocket;

public class WebSocketClient {
   private WebSocket webSocket;
   private final URI serverUri;
   private boolean inError = false;

   public WebSocketClient(URI serverUri) {
      this.serverUri = serverUri;
   }

   public boolean isClosed() {
```

```
    return (webSocket != null && webSocket.isClosed())
           ||
           this.inError;
}

public void connect() {
    HttpClient.newHttpClient()
              .newWebSocketBuilder(serverUri, this.getListener())
              .buildAsync()
              .whenComplete(this::statusChanged);
}

private void statusChanged(WebSocket webSocket, Throwable t) {
    this.webSocket = webSocket;

    if (t == null) {
        this.talkToServer();
    } else {
        this.inError = true;
        System.out.println("Не удалось подключиться к серверу." +
                           " Ошибка: " + t.getMessage());
    }
}

private void talkToServer() {
    // Разрешить прослушивателю получение одного сообщения
    webSocket.request(1);

    // Отправить серверу запрос о текущем времени
    webSocket.sendText("Hello");
}

private WebSocket.Listener getListener() {
    return new WebSocket.Listener() {
        @Override
        public void onOpen(WebSocket webSocket) {
        // Разрешить прослушивателю получение еще одного сообщения
            webSocket.request(1);

            // Уведомить пользователя об успешном подключении
            System.out.println("WebSocket открыт.");
        }

        @Override
        public CompletionStage<?> onClose(WebSocket webSocket,
                          int statusCode, String reason) {
            // Сервер закрыл веб-сокет. Ответить на поступившее от
```

```
            // сервера сообщение о закрытии
            webSocket.sendClose();
            System.out.println("WebSocket закрыт." +
                               " Код закрытия: " + statusCode +
                               ", причина закрытия: " + reason);

            // Вернуть null, означающий, что этот WebSocket можно закрывать
            return null;
         }

         @Override
         public void onError(WebSocket webSocket, Throwable t) {
            System.out.println("Произошла ошибка: " + t.getMessage());
         }

         @Override
         public CompletionStage<?> onText(WebSocket WebSocket,
                CharSequence message, WebSocket.MessagePart part) {
         // Разрешить прослушивателю получение еще одного сообщения
            webSocket.request(1);

            // Напечатать сообщение, полученное от сервера
            System.out.println("Сервер: " + message);

            // Вернуть null, означающий, что этот WebSocket можно закрывать
            return null;
         }
      };
   }
}
```

Класс `WebSocketClient` работает следующим образом.

- В переменной экземпляра `webSocket` сохраняется ссылка на клиентскую оконечную точку.
- В переменной экземпляра `serverUri` сохраняется URI серверной оконечной точки.
- В переменной экземпляра `isError` запоминается, имела ли место ошибка при работе этой оконечной точки.
- Метод `isClosed()` проверяет, была ли оконечная точка закрыта или имела ли место ошибка.
- Переменная экземпляра `webSocket` равна `null` до момента успешного завершения начального квитирования. Ее значение изменяется в методе `statusChanged()`.
- Метод `connect()` строит `WebSocket` и инициирует процедуру начального квитирования. По ее завершении он вызывает метод `statusChanged()` вне зависимости от состояния соединения.

- Метод `statusChanged()` начинает диалог с сервером, вызывая метод `talkToServer()` в случае успешного завершения начального квитирования. В противном случае он печатает сообщение об ошибке и устанавливает флаг `isError` в `true`.
- Метод `talkToServer()` разрешает прослушивателю получить еще одно сообщение и отправляет текстовое сообщение серверной оконечной точке. Отметим, что по получении текстового сообщения от клиента серверная оконечная точка отправляет три сообщения с пятисекундным интервалом, т. е. отправка сообщения из метода `talkToServer()` инициирует обмен сообщениями между двумя оконечными точками.
- Метод `getListener()` создает и возвращает экземпляр `WebSocket.Listener`. Серверная оконечная точка отправляет три сообщения, а вслед за ними сообщение о закрытии. Метод `onClose()` прослушивателя отвечает на сообщение о закрытии от сервера отправкой пустого сообщения о закрытии, и на этом работа клиентской оконечной точки завершается.

В листинге 14.6 приведена программа запуска клиентской оконечной точки. Перед тем как выполнять класс `WebSocketClientTest`, убедитесь, что веб-приложение, содержащее серверную оконечную точку, работает. Кроме того, нужно будет изменить статическую переменную `SERVER_URI` в соответствии с серверной оконечной точкой вашего веб-приложения. Если у вас получится совсем не такой результат, как у меня, обратитесь к разделу «Устранение неполадок в приложении WebSocket». Поскольку программа печатает текущую дату и время с часовым поясом, то в этом отношении ваш результат заведомо будет другим.

Листинг 14.6. Программа запуска клиентской оконечной точки

```
// WebSocketClientTest.java
package com.jdojo.http.client;

import java.net.URI;
import java.net.URISyntaxException;
import java.util.concurrent.TimeUnit;

public class WebSocketClientTest {
   // Не забудьте указать URI вашей серверной оконечной точки
   static final String SERVER_URI ="ws://localhost:8080/webapp/servertime";

   public static void main(String[] args)
         throws URISyntaxException, InterruptedException {
      // Создать клиентский WebSocket
      WebSocketClient wsClient = new WebSocketClient(new URI(SERVER_URI));

      // Подключиться к серверу
      wsClient.connect();
```

```
        // Дождаться закрытия WebSocket
        while(!wsClient.isClosed()) {
           TimeUnit.SECONDS.sleep(1);
        }

        // Выйти
        System.exit(0);
    }
}
```

```
WebSocket открыт.
Сервер: 2016-12-15T14:19:53.311-06:00[America/Chicago]
Сервер: 2016-12-15T14:19:58.312-06:00[America/Chicago]
Сервер: 2016-12-15T14:20:03.313-06:00[America/Chicago]
WebSocket закрыт. Код закрытия: 1000, код причины: Done
```

Устранение неполадок в приложении WebSocket

При тестировании приложения WebSocket многое может пойти не так, как ожидалось. В табл. 14.1 перечислены некоторые проблемы и способы их решения.

Таблица 14.1. Возможные ошибки приложения WebSocket и способы их устранения

Сообщение об ошибке	Решение
`Не удалось подключиться к серверу. Ошибка: java.net.ConnectException: Connection refused: no further information`	Веб-сервер не работает или указан неправильный URI. Запустите веб-сервер и проверьте URI сервера, заданный в статической переменной `SERVER_URI` класса `WebSocketClientTest`
`Не удалось подключиться к серверу. Ошибка: java.net.http.WebSocketHandshakeException: 404: RFC 6455 1.3. Unable to complete handshake; HTTP response status code 404`	URI сервера указывает на неправильную оконечную точку. Проверьте значение статической переменной `SERVER_URI` класса `WebSocketClientTest`

`WebSocket открыт`
`Dec 15, 2016 2:58:03 PM java.net.http.WS$1 onError`
`WARNING: Failing connection java.net.http.WS@162532d6[CONNECTED], reason: 'RFC 6455 7.2.1. Stream ended before a Close frame has been received'`
`Произошла ошибка: null`
	После начального квитирования сервер автоматически закрывает серверную оконечную точку. Обычно это результат работы антивирусной программы на вашей машине. Настройте антивирус, так чтобы он разрешал подключения по HTTP к указанному порту, или запустите веб-сервер на другом порту, который не блокируется антивирусом

Сообщение об ошибке	Решение
`WebSocket открыт.` `Сервер: 2016-12-16T07:15:04.586-06:00[America/Chicago]`	Приложение печатает одну или две строки, а затем зависает. Так бывает, когда в коде клиентской оконечной точки нет вызовов `webSocket.request(1)`. Сервер отправляет сообщения, которые ставятся в очередь, потому что дополнительные сообщения не разрешены. Для решения проблемы включите вызовы метода `request(n)` в обработчики событий `onOpen`, `onText` и других

Резюме

В JDK 9 добавлен клиентский API HTTP/2, который позволяет работать с HTTP-запросами и ответами и Java-приложениях. API содержит классы и интерфейсы для разработки клиентских оконечных точек WebSocket с поддержкой аутентификации и TLS. API находится в пакете `jdk.incubator.http`, который является частью модуля `jdk.incubator.httpclient`.

Наиболее важные компоненты клиентского API HTTP/2 – три абстрактных класса, `HttpClient`, `HttpRequest`, `HttpResponse`, и интерфейс `WebSocket`. Экземпляры этих типов создаются построителями. Класс `HttpClient` неизменяемый. В объекте `HttpClient` хранятся конфигурационные данные для HTTP-соединения, его можно использовать для отправки нескольких HTTP-запросов. HTTP-запрос представляется экземпляром класса `HttpRequest`, а полученный от сервера HTTP-ответ – экземпляром класса `HttpResponse`. Отправлять запросы и получать ответы можно синхронно или асинхронно.

Экземпляр интерфейса `WebSocket` представляет клиентскую оконечную точку протокола WebSocket. Обмен данными с серверной оконечной точкой производится асинхронно. WebSocket API основан на событиях. Для клиентской оконечной точки необходимо задать прослушиватель – экземпляр интерфейса `WebSocket.Listener`. Прослушиватель получает уведомления – путем вызова соответствующих методов – когда происходит событие, например, после успешного завершения начального квитирования вызывается метод `onOpen()`. API поддерживает обмен текстовыми и двоичными сообщениями между пирами. Сообщение может состоять из нескольких частей.

Глава 15

Модифицированный тип Deprecated

Краткое содержание главы:

- как объявлять API нерекомендуемым;
- роль тега `@deprecated` в документации Java и аннотации `@Deprecated` в объявлении API нерекомендуемыми;
- точные правила генерации предупреждений о нерекомендованности;
- изменение аннотации `@Deprecated` в JDK 9;
- новые предупреждения о нерекомендованности в JDK 9;
- использование аннотации `@SuppressWarnings` для подавления различных типов предупреждений о нерекомендованности в JDK 9;
- использование инструмента статического анализа `jdeprscan` для проверки откомпилированного кода на наличие нерекомендуемых API.

Что такое нерекомендуемый API?

Объявление нерекомендуемым – этап жизненного цикла API. Нерекомендуемыми могут быть модули, пакеты, типы, конструкторы, методы, поля, параметры и локальные переменные. Объявляя API нерекомендуемым, вы тем самым говорите пользователю:

- не использовать API, потому что это опасно;
- прекратить использование этого API, потому что ему есть более удачная замена;
- прекратить использование этого API, потому что в будущей версии он будет исключен.

Как объявить API нерекомендуемым

В JDK есть две конструкции для объявления API нерекомендуемым:

- тег `@deprecated` в документации Java;
- аннотация `@Deprecated`.

Тег `@deprecated`, добавленный в документацию Java в версии JDK 1.1, позволяет подробно описать, почему именно API объявлен нерекомендуемым, применяя развитые средства форматирования, имеющиеся в языке HTML. Тип аннотации `java.lang.Deprecated`, добавленный в JDK 5.0, можно применять к любому элементу API, чтобы объявить его нерекомендуемым. До JDK 9 аннотация не содержала никаких элементов. Она видна на этапе выполнения.

Предполагается, что тег `@deprecated` и аннотация `@Deprecated` будут использоваться совместно, т. е. либо оба присутствуют, либо оба отсутствуют. Аннотация `@Deprecated` не позволяет задать описание причины нерекомендованности, для этой цели нужно использовать тег `@deprecated`.

> **Совет.** Если для некоторого элемента API тег `@deprecated` используется, а аннотация `@Deprecated` – нет, то компилятор выдает предупреждение. До JDK 9, чтобы увидеть эти предупреждения, нужно было запускать компилятор с флагом `-Xlint:dep-ann`.

В листинге 15.1 приведено объявление класса `FileCopier`. Предположим, что этот класс поставляется в составе некоторой библиотеки. Он объявлен нерекомендуемым с помощью аннотации `@Deprecated`. В документации присутствует тег `@deprecated` с подробной информацией: когда объявлен, чем заменить и уведомлением о планируемом исключении. До JDK 9 аннотация `@Deprecated` не содержала никаких элементов, поэтому все подробности нужно было описывать в теге `@deprecated`. Отметим, что тег `@since`, встречающийся в документации, говорит, что класс `FileCopier` существует, начиная с версии 1.2 библиотеки, тогда как в теге `@deprecated` говорится, что он объявлен нерекомендуемым, начиная с версии 1.4.

Листинг 15.1. Служебный класс FileCopier

```
// FileCopier.java
package com.jdojo.deprecation;

import java.io.File;

/**
* Класс содержит только статические методы для копирования файлов
* и каталогов.
*
* @deprecated Объявлен нерекомендуемым в версии 1.4. Небезопасен.
* Используйте вместо него класс <code>java.nio.file.Files</code>.
* Класс будет исключен из будущей версии библиотеки.
*
* @since 1.2
*/

@Deprecated
public class FileCopier {
   // Непосредственное создание экземпляров не поддерживается.
   private FileCopier() {
   }
```

```
    /**
    * Копирует содержимое src в dst.
    * @param src Исходный файл
    * @param dst Конечный файл
    * @return true, если копирование завершено успешно, иначе false.
    */
    public static boolean copy(File src, File dst) {
        // Здесь находится содержательный код
        return true;
        }

    // Другие методы
}
```

Программа Javadoc перемещает содержимое тега `@deprecated` в начало сгенерированной документации, чтобы привлечь внимание читателя. Компилятор выдает предупреждение, если нерекомендуемый API встречается в коде, который не объявлен нерекомендуемым. Отметим, что само присутствие аннотации `@Deprecated` не является поводом для предупреждения, оно выдается, когда API, снабженный такой аннотацией, действительно используется в программе. Если бы мы использовали класс `FileCopier` в другом месте программы, то получили бы от компилятора предупреждение.

Модификация аннотации @Deprecated в JDK 9

Предположим, что мы откомпилировали код и развернули его в производственной среде. Если затем заменить в этой среде JDK или библиотеки новыми версиями, в которых некоторые используемые в нашем приложении API объявлены нерекомендуемыми, то никаких предупреждений мы не получим и, стало быть, упустим шанс вовремя отказаться от нерекомендуемых API. Чтобы получить предупреждения, код необходимо перекомпилировать. Раньше не было никакого инструмента для сканирования и анализа откомпилированного кода (например, JAR-файлов) на предмет использования нерекомендуемых API. Хуже того – если нерекомендуемый API исключается из новой версии библиотеки, то старый откомпилированный код вообще перестает работать. Разработчикам также не хватало информации в документации по нерекомендуемым элементам – не было формального способа сообщить, когда API объявлен нерекомендуемым и в какой версии его планируется удалить. Эти сведения можно было включить лишь в текстовую часть тега `@deprecated`. В JDK 9 сделана попытка решить эти проблемы путем расширения аннотации `@Deprecated`. В ней появилось два новых элемента: `since` и `forRemoval`.

Раньше объявление аннотации выглядело так:

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE})
public @interface Deprecated {
}
```

В JDK 9 внесены изменения, выделенные полужирным шрифтом:

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, MODULE, PARAMETER, TYPE})
public @interface Deprecated {
String since() default "";
boolean forRemoval() default false;
}
```

Для обоих новых элементов заданы значения по умолчанию, так что ранее написанные аннотации не «поломаются». В элементе `since` указано, в какой версии аннотированный элемент API объявлен нерекомендуемым. Это строка, которая должна следовать соглашениям, используемым для именования версий JDK, например, «9». По умолчанию строка пуста. Отметим, что в JDK 9 в аннотации `@Deprecated` не появилось элемента для описания причины нерекомендованности. Тому есть две причины.

- Эта аннотация видна на этапе выполнения. Добавление текстового описания увеличило бы потребление памяти.
- Описание может и не быть простым текстом. Например, нужно дать ссылку на замену нерекомендуемого API. Тег `@deprecated` в документации Java уже предоставляет такую функциональность.

В элементе `forRemoval` указывается, что аннотированный элемент API будет исключен из будущей версии и следует вовремя заменить его. По умолчанию этот элемент равен `false`.

> **Совет.** Тег `@since` в документации говорит, когда элемент API был добавлен, а элемент `since` в аннотации `@Deprecated` – когда элемент был объявлен нерекомендуемым. В JDK 9 были предприняты разумные усилия, чтобы задним число проставить значения того и другого для большинства, если не всех случаев употребления аннотации `@Deprecated` в API, входящих в состав Java SE .

До JDK 9 предупреждения о нерекомендованности выдавались, исходя из наличия аннотации `@Deprecated` для элемента API и места, в котором элемент используется (см. табл. 15.1). Предупреждение выдавалось, если нерекомендуемый элемент API использовался в коде, который не был объявлен нерекомендуемым. Если нерекомендуемыми были и элемент API, и место его использования, то предупреждение не выдавалось. Существовала возможность подавить предупреждения, снабдив место использования аннотацией `@SuppressWarnings("deprecation")`.

Таблица 15.1. Матрица предупреждений о нерекомендованности до JDK 9

Место использования API	Место объявления API	
	Не нерекомендуемое	Нерекомендуемое
Не нерекомендуемое	Н	П
Нерекомендуемое	Н	Н

Н = Предупреждение не выдается, П = Предупреждение выдается

После включения элемента `forRemoval` в тип аннотации `@Deprecated` появилось пять новых случаев. Если в объявлении API нерекомендуемым элемент `forRemoval` равен `false`, то говорят об *обыкновенном объявлении нерекомендуемым*, и в этом случае предупреждения о нерекомендованности называются *обыкновенными*. Если же элемент `forRemoval` равен `true`, то говорят о *терминальном объявлении нерекомендуемым*, а предупреждения называются *терминальными* или *предупреждениями об исключении*. В табл. 15.2 показана матрица предупреждений о нерекомендованности в JDK 9.

Таблица 15.2. Матрица предупреждений о нерекомендованности до JDK 9

Место использования API	Место объявления API		
	Не нерекомендуемое	Обыкновенно нерекомендуемое	Терминально нерекомендуемое
Не нерекомендуемое	Н	ОП	ТП
Обыкновенно нерекомендуемое	Н	Н	ТП
Терминально нерекомендуемое	Н	Н	ТП

Н = Предупреждение не выдается, ОП = Выдается обыкновенное предупреждение, ТП = Выдается терминальное предупреждение

В целях сохранения обратной совместимости четыре случая в левом верхнем углу оставлены такими же, как в табл. 15.1. Это значит, что если для вашей программы выдавалось предупреждение о нерекомендованности в JDK 8, то и в JDK 9 будет выдаваться обыкновенное предупреждение о нерекомендованности. Если же API объявлен терминально нерекомендуемым, то во всех местах использования будет выдаваться предупреждение об исключении.

В JDK 9 требует пояснений один случай – когда и сам API, и место его использования объявлены терминально нерекомендуемыми. Раз и API, и код нерекомендуемые и будут удалены, то какой смысл в предупреждении? Так сделано, чтобы охватить случай, когда терминально нерекомендуемый API и место его использования находятся в двух разных кодовых базах, сопровождаемых независимо. Если код, где API используется, будет существовать дольше самого API, то в нем начнут возникать неожиданные ошибки во время выполнения. Выдача предупреждения в месте использования даст сопровождающим программистам возможность спланировать альтернативу на случай, если терминально нерекомендуемый API будет исключен раньше, чем код в месте использования.

Подавление предупреждений о нерекомендованности

Появление предупреждений об удалении в JDK 9 привело к возникновению нового случая в подавлении таких предупреждений. До JDK 9 можно было подавить все предупреждения о нерекомендованности, снабдив место использования аннотацией `@SuppressWarnings("deprecation")`. Рассмотрим такую ситуацию:

- в JDK 8 API объявлен нерекомендуемым, но в месте использования предупреждение подавлено;
- в JDK 9 объявление этого API нерекомендуемым из обыкновенного стало терминальным;
- код в месте использования успешно компилируется в JDK 9, поскольку в JDK 8 предупреждения были подавлены;
- API исключен, и в месте использования стали возникать ошибки во время выполнения, хотя никаких предупреждений об исключении не выдавалось.

Чтобы избежать такого развития событий, в JDK 9 аннотация `@SuppressWarnings("deprecation")` не подавляет предупреждения об исключении, а подавляет только обыкновенные предупреждения. Чтобы подавить предупреждения об исключении, нужна аннотация `@SuppressWarnings("removal")`. А чтобы подавить любые предупреждения о нерекомендованности, добавьте аннотацию `@SuppressWarnings({"deprecation", "removal"})`.

Пример нерекомендуемого API

В этом разделе будут показаны все случаи объявления API нерекомендуемым – с подавлением и без подавления предупреждений. Я объявляю нерекомендуемыми только методы, но, конечно, все сказанное относится и к другим элементам API. В комментариях описано ожидаемое поведение. В листинге 15.2 приведен код класса `Box`. Класс содержит три метода, по одному в каждой категории: не объявленный нерекомендуемым, объявленный обыкновенно нерекомендуемым и объявленный терминально нерекомендуемым. При компиляции класса `Box` никаких предупреждений не выдается, потому что нерекомендуемые API в нем не используются, а только объявляются.

Листинг 15.2. Класс Box с методами трех типов: не нерекомендуемые, обыкновенно нерекомендуемые и терминально нерекомендуемые

```
// Box.java
package com.jdojo.deprecation;

/**
* Класс служит для демонстрации объявления API нерекомендуемым.
*/
public class Box {
   /**
   * Не является нерекомендуемым
   */
   public static void notDeprecated() {
      System.out.println("notDeprecated...");
   }

   /**
   * Обыкновенно нерекомендуемый.
```

```
    * @deprecated Не использовать.
    */
    @Deprecated(since="2")
    public static void deprecatedOrdinarily() {
        System.out.println("deprecatedOrdinarily...");
    }

    /**
    * Терминально нерекомендуемый.
    * @deprecated Будет исключен в следующей версии
    * Внесите изменения в код.
    */
    @Deprecated(since="2", forRemoval=true)
    public static void deprecatedTerminally() {
        System.out.println("deprecatedTerminally...");
    }
}
```

В листинге 15.3 приведен код класса `BoxTest`, в котором используются все методы класса `Box`. Некоторые методы класса `BoxTest` объявлены обыкновенно или терминально нерекомендуемыми. Первые девять методов соответствуют девяти случаям в табл. 15.2, когда генерируются четыре предупреждения о нерекомендованности: одно обыкновенное и три терминальных. Методы с именами вида `m4X()`, демонстрируют подавление обыкновенных и терминальных предупреждений.

Листинг 15.3. Класс BoxTest, демонстрирующий использование нерекомендуемых API и подавление предупреждений о нерекомендованности

```
// BoxTest.java
package com.jdojo.deprecation;

public class BoxTest {
    /**
    * API: не нерекомендуемый
    * Место использования: не нерекомендуемый
    * Предупреждение о нерекомендованности: нет
    */
    public static void m11() {
        Box.notDeprecated();
    }

    /**
    * API: обыкновенно нерекомендуемый
    * Место использования: не нерекомендуемый
    * Предупреждение о нерекомендованности: нет
    */
    public static void m12() {
```

```
    Box.deprecatedOrdinarily();
}

/**
* API: терминально нерекомендуемый
* Место использования: не нерекомендуемый
* Предупреждение о нерекомендованности: предупреждение об исключении
*/
public static void m13() {
    Box.deprecatedTerminally();
}

/**
* API: не нерекомендуемый
* Место использования: обыкновенно нерекомендуемый
* Предупреждение о нерекомендованности: нет
* @deprecated использование опасно.
*/
@Deprecated(since="1.1")
public static void m21() {
    Box.notDeprecated();
}

/**
* API: обыкновенно нерекомендуемый
* Место использования: обыкновенно нерекомендуемый
* Предупреждение о нерекомендованности: нет
* @deprecated использование опасно.
*/
@Deprecated(since="1.1")
public static void m22() {
    Box.deprecatedOrdinarily();
}

/**
* API: терминально нерекомендуемый
* Место использования: обыкновенно нерекомендуемый
* Предупреждение о нерекомендованности: предупреждение об исключении
* @deprecated использование опасно.
*/
@Deprecated(since="1.1")
public static void m23() {
    Box.deprecatedTerminally();
}

/**
* API: не нерекомендуемый
```

```
* Место использования: терминально нерекомендуемый
* Предупреждение о нерекомендованности: нет
* @deprecated будет удалено.
*/
@Deprecated(since="1.1", forRemoval=true)
public static void m31() {
   Box.notDeprecated();
}

/**
* API: обыкновенно нерекомендуемый
* Место использования: терминально нерекомендуемый
* Предупреждение о нерекомендованности: нет
* @deprecated будет удалено.
*/
@Deprecated(since="1.1", forRemoval=true)
public static void m32() {
   Box.deprecatedOrdinarily();
}

/**
* API: терминально нерекомендуемый
* Место использования: терминально нерекомендуемый
* Предупреждение о нерекомендованности: предупреждение об исключении
* @deprecated будет удалено.
*/
@Deprecated(since="1.1", forRemoval=true)
public static void m33() {
   Box.deprecatedTerminally();
}

/**
* API: обыкновенно и терминально нерекомендуемый
* Место использования: не нерекомендуемый
* Предупреждение о нерекомендованности: обыкновенное предупреждение
*    и предупреждение об исключении
*/
public static void m41() {
   Box.deprecatedOrdinarily();
   Box.deprecatedTerminally();
}

/**
* API: обыкновенно и терминально нерекомендуемый
* Место использования: не нерекомендуемый
* Предупреждение о нерекомендованности: обыкновенные предупреждения
*/
```

```
    @SuppressWarnings("deprecation")
    public static void m42() {
        Box.deprecatedOrdinarily();
        Box.deprecatedTerminally();
    }

    /**
    * API: обыкновенно и терминально нерекомендуемый
    * Место использования: не нерекомендуемый
    * Предупреждение о нерекомендованности: предупреждения об исключении
    */
    @SuppressWarnings("removal")
    public static void m43() {
        Box.deprecatedOrdinarily();
        Box.deprecatedTerminally();
    }

    /**
    * API: обыкновенно и терминально нерекомендуемый
    * Место использования: не нерекомендуемый
    * Предупреждение о нерекомендованности: предупреждения об исключении
    */
    @SuppressWarnings({"deprecation", "removal"})
    public static void m44() {
        Box.deprecatedOrdinarily();
        Box.deprecatedTerminally();
    }
}
```

Класс `BoxTest` следует откомпилировать с флагом `-Xlint:deprecation`, чтобы видеть предупреждения о нерекомендованности. Следующую команду нужно вводить в одной строке, а не в двух.

```
C:\Java9Revealed\com.jdojo.deprecation\src>javac -Xlint:deprecation
 -d ..\build\classes com\jdojo\deprecation\BoxTest.java
```

```
com\jdojo\deprecation\BoxTest.java:20: warning: [deprecation] deprecatedOrdinarily()
in Box has been deprecated
    Box.deprecatedOrdinarily();
       ^

com\jdojo\deprecation\BoxTest.java:29: warning: [removal] deprecatedTerminally() in
Box has been deprecated and marked for removal
    Box.deprecatedTerminally();
       ^

com\jdojo\deprecation\BoxTest.java:62: warning: [removal] deprecatedTerminally() in
```

```
Box has been deprecated and marked for removal
    Box.deprecatedTerminally();
       ^

com\jdojo\deprecation\BoxTest.java:95: warning: [removal] deprecatedTerminally() in
Box has been deprecated and marked for removal
    Box.deprecatedTerminally();
       ^

com\jdojo\deprecation\BoxTest.java:105: warning: [deprecation] deprecatedOrdinarily() in
Box has been deprecated
    Box.deprecatedOrdinarily();
       ^

com\jdojo\deprecation\BoxTest.java:106: warning: [removal] deprecatedTerminally() in
Box has been deprecated and marked for removal
    Box.deprecatedTerminally();
       ^

com\jdojo\deprecation\BoxTest.java:117: warning: [removal] deprecatedTerminally() in
Box has been deprecated and marked for removal
    Box.deprecatedTerminally();
       ^

com\jdojo\deprecation\BoxTest.java:127: warning: [deprecation] deprecatedOrdinarily() in
Box has been deprecated
    Box.deprecatedOrdinarily();
       ^
8 warnings
```

Статический анализ нерекомендуемых API

Напомним, что предупреждения о нерекомендованности выдаются во время компиляции. Вы не увидите никаких предупреждений, если в уже развернутом приложении используется обыкновенно нерекомендуемый API или если на этапе выполнения возникает ошибка из-за того, что когда-то допустимый API объявлен терминально нерекомендуемым и исключен. До выпуска JDK 9 при обновлении версии JDK или других библиотек необходимо было перекомпилировать собственный код, чтобы увидеть предупреждения о нерекомендованности. В JDK 9 ситуация стала лучше, поскольку появился инструмент *статического* анализа `jdeprscan`, позволяющий просканировать откомпилированный код в поисках встречающихся в нем нерекомендуемых API. В настоящее время инструмент сообщает только о нерекомендуемых API самого JDK. Если в откомпилированном коде используются нерекомендуемые API из других библи-

отек, например Spring, Hibernate или ваших собственных, то программа ничего о них не скажет.

Программа `jdeprscan` находится в каталоге `JDK_HOME\bin` и запускается следующим образом:

```
jdeprscan [options] {dir|jar|class}
```

Здесь `[options]` – нуль или более параметров. Можно задать через запятую список каталогов, JAR-файлов или полных имен классов, подлежащих сканированию. Ниже перечислены параметры программы:

- `-l, --list`
- `--class-path <CLASSPATH>`
- `--for-removal`
- `--release <6|7|8|9>`
- `-v, --verbose`
- `--version`
- `--full-version`
- `-h, --help`

Если задан параметр `-list`, то выводится список нерекомендуемых API в Java SE. В этом случае не следует задавать местоположение откомпилированных классов.

Параметр `--class-path` задает путь к классам, для нахождения зависимостей в процессе сканирования.

Параметр `--for-removal` ограничивает сканирование только теми нерекомендуемыми API, которые подлежат исключению в будущем. Его можно использовать только для версии 9 или более поздних, поскольку раньше тип аннотации `@Deprecated` не содержал элемента `forRemoval`.

Параметр `--release` задает версию Java SE, в которой API были объявлены нерекомендуемыми. Например, чтобы ввести список API, объявленных нерекомендуемыми в версии JDK 6, запустите программу следующим образом:

```
jdeprscan --list --release 6
```

Если указан параметр `-verbose`, то в процессе сканирования печатаются дополнительные сообщения.

При задании параметров `--version` и `--full-version` печатается соответственно сокращенная и полная версия программы `jdeprscan`.

Параметр `--help` печатает подробную справку о программе.

В листинге 15.4 приведен код класса `JDeprScanTest`. Код тривиальный, поскольку интерес представляет не его выполнение, а компиляция. Создается два потока, один из них останавливается методом `stop()` класса `Thread`, а другой уничтожается методом `destroy()`. Методы `stop()` и `destroy()` объявлены обыкновенно нерекомендуемыми в версиях JDK 1.2 и JDK 1.5 соответственно. В JDK 9 метод `destroy()` объявлен терминально нерекомендуемым, а метод `stop()` оставлен обыкновенно нерекомендуемым. В последующих примерах я воспользуюсь этим классом.

Листинг 15.4. Класс JDeprScanTest, в котором используется обыкновенно нерекомендуемый метод stop() и терминально нерекомендуемый метод destroy() класса Thread

```
// JDeprScanTest.java
package com.jdojo.deprecation;

public class JDeprScanTest {
   public static void main(String[] args) {
      Thread t = new Thread(() -> System.out.println("Test"));
      t.start();
      t.stop();

      Thread t2 = new Thread(() -> System.out.println("Test"));
      t2.start();
      t2.destroy();
   }
}
```

Следующая команда печатает список всех нерекомендуемых API в JDK 9. Список длинный, и до начала печати проходит несколько секунд, в течение которых команда сканирует весь JDK.

```
C:\Java9Revealed>jdeprscan -list
```

```
@Deprecated java.lang.ClassLoader
javax.tools.ToolProvider.getSystemToolClassLoader()
...
```

Следующая команда печатает все терминально нерекомендуемые API в JDK 9, т. е. те, что планируется исключить в будущей версии:

```
C:\Java9Revealed>jdeprscan --list --for-removal
```

```
...
@Deprecated(since="9", forRemoval=true) class java.lang.Compiler
...
```

Следующая команда печатает список всех API, объявленных нерекомендуемыми в JDK 8:

```
C:\Java9Revealed>jdeprscan --list --release 8
```

```
@Deprecated class javax.swing.text.TableView.TableCell
...
```

Следующая команда печатает список нерекомендуемых API, используемых в классе `java.lang.Thread`.

```
C:\Java9Revealed>jdeprscan java.lang.Thread
```

```
class java/lang/Thread uses deprecated method java/lang/Thread::resume()V
```

Отметим, что эта команда не печатает список нерекомендуемых API в классе `Thread`. Она печатает список тех API в классе `Thread`, в которых *используются* нерекомендуемые API.

Следующая команда печатает список нерекомендуемых API JDK, используемых в откомпилированном коде данной главы, который находится в каталоге `Java9Revealed/com.jdojo.deprecation/build/classes` прилагаемого к книге кода.

```
C:\Java9Revealed>jdeprscan com.jdojo.deprecation\build\classes
```

```
Directory com.jdojo.deprecation\build\classes:
class com/jdojo/deprecation/JDeprScanTest uses deprecated method
java/lang/Thread::stop()V
class com/jdojo/deprecation/JDeprScanTest uses deprecated method
java/lang/Thread::destroy()V (forRemoval=true)
```

```
C:\Java9Revealed>jdeprscan --for-removal com.jdojo.deprecation\build\classes
```

```
Directory com.jdojo.deprecation\build\classes:
class com/jdojo/deprecation/JDeprScanTest uses deprecated method
java/lang/Thread::destroy()V (forRemoval=true)
```

Динамический анализ нерекомендуемых API

Программа `jdeprscan` – инструмент статического анализа, поэтому случаи динамического использования нерекомендуемых API она не видит. Например, нерекомендуемый метод можно вызвать с помощью рефлексии, и `jdeprscan` этого не заметит. Проигнорирует она также нерекомендуемые методы в поставщиках служб, загружаемых классом `ServiceLoader`.

В будущей версии в JDK, возможно, будет включен инструмент динамического анализа `jdeprdetect`, который будет отслеживать все случаи использования нерекомендуемых API на этапе выполнения. Он будет полезен для нахождения мертвого кода, ссылающегося на нерекомендуемые API, о которых сообщает программа статического анализа `jdeprscan`.

Отказ от предупреждений о нерекомендованности при импорте

До JDK 9 компилятор выдавал предупреждение при попытке импортировать нерекомендуемые конструкции с помощью предложений `import`, даже если места их использования были помечены аннотацией `@SuppressWarnings`. Это нервировало разработчика, стремящегося избавиться от всех предупреждений при компиляции программы. Ведь подавить эти предупреждения было нельзя, т. к. предложения `import` не аннотируются. В JDK 9 эта проблема решена – предупреждения о нерекомендованности при импорте *вообще* не выдаются.

Рассмотрим класс `ImportDeprecationWarning`, показанный в листинге 15.5. В нем нерекомендуемый класс `StringBufferInputStream` используется в трех местах:

- в предложении `import`;
- в объявлении переменной;
- в выражении создания экземпляра.

Листинг 15.5. Класс ImportDeprecationWarning

```
// ImportDeprecationWarning.java
package com.jdojo.deprecation;

import java.io.StringBufferInputStream;

public class ImportDeprecationWarning {
   @SuppressWarnings("deprecation")
   public static void main(String[] args) {
      StringBufferInputStream sbis =
           new StringBufferInputStream("Hello");

      for(int c = sbis.read(); c != -1; c = sbis.read()) {
         System.out.println((char)c);
      }
   }
}
```

Метод `main()` снабжен аннотацией `@SuppressWarnings`, чтобы подавить предупреждения о нерекомендованности. При компиляции этого класса в JDK 8 с флагом `-Xlint:deprecation` выдается показанное ниже предупреждение, а при компиляции в JDK 9 не выдается.

```
C:\Java9Revealed\com.jdojo.deprecation\src>javac -Xlint:deprecation -d ..\build\classes com\
jdojo\deprecation\ImportDeprecationWarning.java

com\jdojo\deprecation\ImportDeprecationWarning.java:4: warning: [deprecation]
StringBufferInputStream in java.io has been deprecated
```

```
import java.io.StringBufferInputStream;
           ^
1 warning
```

Если бы при компиляции в JDK 8 мы убрали аннотацию `@SuppressWarnings` с метода `main()`, то компилятор выдал бы три предупреждения – по одному для каждого случая использования нерекомендуемого класса `StringBufferInputStream`, тогда как в JDK 9 предупреждения было бы только два – предложение `import` не считается основанием для предупреждения.

Резюме

Объявление нерекомендуемым в Java – этап жизненного цикла API. Это предупреждение разработчику: откажись от API, потому что его использование небезопасно, потому что существует лучшая альтернатива или потому что он будет исключен в следующей версии. Использование нерекомендуемых API приводит к выдаче предупреждений на этапе компиляции.

Тег `@deprecated` в документации и аннотация `@Deprecated` используются совместно для объявления нерекомендуемыми модулей, пакетов, типов, конструкторов, методов, полей и локальных переменных. До JDK 9 в этой аннотации не было никаких элементов. Она видна на этапе выполнения.

В JDK 9 в аннотацию добавлено два элемента: `since` и `forRemoval`. Элемент `since`, по умолчанию равный пустой строке, содержит версию, в которой API был объявлен нерекомендуемым. Булев элемент `forRemoval` по умолчанию равен `false`. Значение `true` означает, что данный элемент API будет исключен в будущей версии.

Компилятор в JDK 9 выдает предупреждения двух типов в зависимости от значения элемента `forRemoval` в аннотации `@Deprecated`: *обыкновенные предупреждения о нерекомендованности*, если `forRemoval=false`, и *предупреждения об исключении*, если `forRemoval=true`.

До JDK 9 предупреждения о нерекомендованности можно было подавить, снабдив места использования нерекомендуемых API аннотацией `@SuppressWarnings("deprecation")`. В JDK 9 необходимо использовать аннотацию `@SuppressWarnings("deprecation")` для подавления обыкновенных предупреждений, аннотацию `@SuppressWarnings("removal")` для подавления предупреждений об исключении и аннотацию `@SuppressWarnings({"deprecation", "removal"})` для подавления предупреждений обоих видов.

До JDK 9 импорт нерекомендуемой конструкции с помощью предложения `import` приводил к выдаче предупреждения на этапе компиляции. В JDK 9 такие предупреждения не выдаются.

Глава 16

Навигация по стеку

Краткое содержание главы:

- что такое стек и кадр стека;
- как осуществлялась навигация по стеку до JDK 9;
- API навигации по стеку в JDK 9;
- как узнать вызывающий класс в JDK 9.

Что такое стек?

У каждого потока в JVM имеется собственный стек, создаваемый одновременно с потоком. Стек – это структура данных, обслуживаемая в соответствии с дисциплиной «последним пришел – первым ушел» (LIFO). Стек состоит из кадров. При каждом вызове метода создается новый кадр, который помещается на вершину стека. Кадр уничтожается (выталкивается из стека), когда метод возвращает управление. В кадре стека хранятся локальные переменные метода, а также операнды, возвращаемое значение и ссылка на пул констант класса текущего метода. В конкретной реализации JVM в стеке может храниться дополнительная информация.

Кадр стека JVM представляет вызов метода в данном потоке. В любой момент времени в каждом потоке активен только один кадр, который называется *текущим кадром*, а соответствующий метод называется *текущим методом*. Класс, в котором определен текущий метод, называется *текущим классом*. Когда текущий метод вызывает другой метод, его кадр перестает быть текущим; в стек помещается новый кадр, который становится текущим, а текущим методом становится только что вызванный. Когда метод возвращает управление, текущим вновь становится предыдущий кадр. Дополнительные сведения о стеках и кадрах стеков см. в «Спецификации виртуальной машины Java» по адресу `https://docs.oracle.com/javase/specs/jvms/se8/html/index.html`.

Совет. Если JVM поддерживает платформенные методы, то у потока также имеется стек платформенных методов, в котором находятся кадры, соответствующие вызовам таких методов.

На рис. 16.1 показано два потока и их стеки JVM. В стеке первого потока находится четыре кадра, а в стеке второго – три кадра. В потоке Thread-1 активен кадр 4, а в потоке Thread-2 – кадр 3.

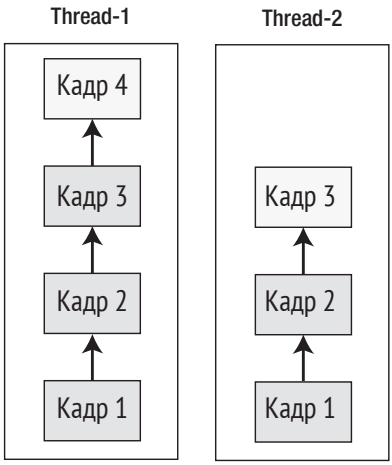

Рис. 16.1. Потоки и их стеки в JVM

Что такое навигация по стеку?

Навигацией по стеку (или обходом стека) называется процедура перемещения по кадрам стека потока и исследования их содержимого. Начиная с версии Java 1.4, можно было получить мгновенный снимок стека потока и сведения о каждом кадре: имя класса и вызванного метода, имя исходного файла, номер строки в файле и т. д. Классы и интерфейсы, предназначенные для навигации по стеку, составляют API навигации по стеку.

Навигация по стеку JDK 8

До JDK 9 для обхода всех кадров в стеке потока необходимо было пользоваться следующими классами из пакета `java.lang`:

- `Throwable`
- `Thread`
- `StackTraceElement`

Экземпляр класса `StackTraceElement` представляет один кадр стека. Метод `getStackTrace()` класса `Throwable` возвращает массив `StackTraceElement[]`, содержащий кадры стека текущего потока. Метод `getStackTrace()` класса `Thread` возвращает массив `StackTraceElement[]`, содержащий кадры стека указанного потока. В первом элементе массива находится верхний кадр стека, представляющий последний по порядку вызов метода. В некоторых реализациях JVM в возвращаемом массиве некоторые кадры могут отсутствовать.

Класс `StackTraceElement` содержит следующие методы, возвращающие сведения о вызове метода, представленном данным кадром:

- `String getClassLoaderName()`
- `String getClassName()`
- `String getFileName()`
- `int getLineNumber()`
- `String getMethodName()`
- `String getModuleName()`
- `String getModuleVersion()`
- `boolean isNativeMethod()`

Совет. Методы `getModuleName()`, `getModuleVersion()` и `getClassLoaderName()` добавлены в этот класс в JDK 9.

У большинства методов класса `StackTraceElement` имена говорят сами за себя, например, `getMethodName()` возвращает имя метода, вызов которого представлен данным кадром. Метод `getFileName()` возвращает имя исходного файла, содержащего код вызванного метода, а `getLineNumber()` – номер строки в этом файле, в которой находится вызов метода.

Ниже показано, как обойти стек текущего потока с применением классов `Throwable` и `Thread`:

```
// С использованием класса Throwable
StackTraceElement[] frames = new Throwable().getStackTrace();

// С использованием класса Thread
StackTraceElement[] frames2 = Thread.currentThread().getStackTrace();

// Обработать кадры...
```

Весь код, приведенный в этой главе, находится в модуле `com.jdojo.stackwalker`, объявление которого показано в листинге 16.1.

Листинг 16.1. Объявление модуля com.jdojo.stackwalker

```
// module-info.java
module com.jdojo.stackwalker {
    exports com.jdojo.stackwalker;
}
```

В листинге 16.2 приведен код класса `LegacyStackWalk`, а ниже – результат его работы в JDK 8.

Листинг 16.2. Обход стека потока до JDK 9

```
// LegacyStackWalk.java
package com.jdojo.stackwalker;

import java.lang.reflect.InvocationTargetException;

public class LegacyStackWalk {
   public static void main(String[] args) {
      m1();
   }

   public static void m1() {
      m2();
   }

   public static void m2() {
```

```
        // Вызвать m3() напрямую
        System.out.println("\nБез использования рефлексии: ");
        m3();

        // Вызвать m3() с использованием рефлексии
        try {
           System.out.println("\С использованием рефлексии: ");
           LegacyStackWalk.class
                          .getMethod("m3")
                          .invoke(null);
        } catch (NoSuchMethodException |
                 InvocationTargetException |
                 IllegalAccessException |
                 SecurityException e) {
           e.printStackTrace();
        }
     }

     public static void m3() {
        // Напечатать сведения о стеке вызовов
        StackTraceElement[] frames = Thread.currentThread()
                                           .getStackTrace();
        for(StackTraceElement frame : frames) {
           System.out.println(frame.toString());
        }
     }
}
```

```
Без использования рефлексии:
java.lang.Thread.getStackTrace(Thread.java:1552)
com.jdojo.stackwalker.LegacyStackWalk.m3(LegacyStackWalk.java:37)
com.jdojo.stackwalker.LegacyStackWalk.m2(LegacyStackWalk.java:18)
com.jdojo.stackwalker.LegacyStackWalk.m1(LegacyStackWalk.java:12)
com.jdojo.stackwalker.LegacyStackWalk.main(LegacyStackWalk.java:8)

С использованием рефлексии:
java.lang.Thread.getStackTrace(Thread.java:1552)
com.jdojo.stackwalker.LegacyStackWalk.m3(LegacyStackWalk.java:37)
sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
java.lang.reflect.Method.invoke(Method.java:498)
com.jdojo.stackwalker.LegacyStackWalk.m2(LegacyStackWalk.java:25)
com.jdojo.stackwalker.LegacyStackWalk.m1(LegacyStackWalk.java:12)
com.jdojo.stackwalker.LegacyStackWalk.main(LegacyStackWalk.java:8)
```

Метод `main()` класса `LegacyStackWalk` вызывает метод `m1()`, который вызывает метод `m2()`. Метод `m2()`вызывает метод `m3()` дважды: один раз напрямую, а другой с использованием рефлексии. Метод `m3()` получает снимок стека текущего потока с помощью метода `getStrackTrace()` класса `Thread` и печатает сведения о кадрах, обращаясь к методу `toString()` класса `StackTraceElement`. Можно было бы получить ту же самую информацию о кадре, вызывая методы этого класса. При выполнении класса `LegacyStackWalk` в JDK 9 в сведения о кадре включается еще имя и версия модуля:

```
Без использования рефлексии:
java.base/java.lang.Thread.getStackTrace(Thread.java:1654)
com.jdojo.stackwalker/com.jdojo.stackwalker.LegacyStackWalk.m3(LegacyStackWalk.java:37)
com.jdojo.stackwalker/com.jdojo.stackwalker.LegacyStackWalk.m2(LegacyStackWalk.java:18)
com.jdojo.stackwalker/com.jdojo.stackwalker.LegacyStackWalk.m1(LegacyStackWalk.java:12)
com.jdojo.stackwalker/com.jdojo.stackwalker.LegacyStackWalk.main(LegacyStackWalk.java:8)

С использованием рефлексии:
java.base/java.lang.Thread.getStackTrace(Thread.java:1654)
com.jdojo.stackwalker/com.jdojo.stackwalker.LegacyStackWalk.m3(LegacyStackWalk.java:37)
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
java:62)
java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.
java:43)
java.base/java.lang.reflect.Method.invoke(Method.java:538)
com.jdojo.stackwalker/com.jdojo.stackwalker.LegacyStackWalk.m2(LegacyStackWalk.java:25)
com.jdojo.stackwalker/com.jdojo.stackwalker.LegacyStackWalk.m1(LegacyStackWalk.java:12)
com.jdojo.stackwalker/com.jdojo.stackwalker.LegacyStackWalk.main(LegacyStackWalk.java:8)
```

Недостатки навигации по стеку

До JDK 9 у API навигации по стеку было несколько недостатков.

- Неэффективность. Метод `getStrackTrace()` класса `Throwable` возвращал снимок всего стека. Невозможно было получить всего несколько верхних кадров.
- Кадры содержали имена методов и классов, но не ссылки на классы. Ссылки на классы – это экземпляры класса `Class<?>`, тогда как имена классов – просто строки.
- Спецификация JVM позволяет реализации пропускать некоторые кадры в целях эффективности. В таких случаях получить для инспектирования весь стек было невозможно.
- Поведение многих API – в JDK и в других библиотеках – зависит от того, кто их вызывает. Например, для вызова метода `addExports()` класса `Module` вызывающий класс должен находиться в том же модуле, иначе будет возбуждено исключение `IllegalCallerException`. В существующих API не было прос-

того и эффективного способа получить ссылку на вызывающий класс. Для этого нужно было прибегать к внутреннему API JDK – статическому методу `getCallerClass()` класса `sun.reflect.Reflection`.

- Не было простого способа отфильтровать кадры стека, соответствующие конкретным классам.

Навигация по стеку в JDK 9

В JDK 9 появился новый API навигации по стеку, состоящий из единственного класса `StackWalker` в пакете `java.lang`. Класс предоставляет последовательный поток кадров стека текущего потока. Кадры поступают по порядку – от верхнего к нижнему. Класс `StackWalker` очень эффективен, потому что вычисляет кадры стека лениво. К тому же, в нем есть вспомогательный метод для получения ссылки на вызывающий класс. Ниже перечислены члены класса:

- вложенное перечисление `StackWalker.Option`
- вложенный интерфейс `StackWalker.StackFrame`;
- методы для получения экземпляра класса `StackWalker`;
- методы для обработки кадров стека;
- метод для получения вызывающего класса.

В следующих разделах описываются все компоненты класса `StackWalker` и порядок их использования.

Параметры навигации по стеку

Поведение класса `StackWalker` можно настроить с помощью параметров – элементов перечисления `StackWalker.Option`:

- `RETAIN_CLASS_REFERENCE`
- `SHOW_HIDDEN_FRAMES`
- `SHOW_REFLECT_FRAMES`

Если задан параметр `RETAIN_CLASS_REFERENCE`, то кадры, возвращенные классом `StackWalker`, будут содержать ссылку на объект `Class`, описывающий класс, в котором объявлен метод, представленный данным кадром. Этот параметр следует также задавать, если требуется получить ссылку на объект `Class`, соответствующую вызываемому методу. По умолчанию параметр не задан.

По умолчанию кадры, зависящие от реализации, и кадры рефлексии не включаются в поток, возвращаемый классом `StackWalker`. Чтобы включались все кадры, задайте параметр `SHOW_HIDDEN_FRAMES`.

Если задан параметр `SHOW_REFLECT_FRAMES`, то поток кадров включает кадры рефлексии. Но кадры, зависящие от реализации, остаются скрытыми; чтобы их увидеть, нужно задать параметр `SHOW_HIDDEN_FRAMES`.

Практическое применение этих параметров будет продемонстрировано ниже.

Представление кадра стека

До JDK 9 для представления кадра стека использовался класс `StackTraceElement`. В JDK 9 для этой цели применяется интерфейс `StackWalker.StackFrame`.

Совет. Имена конкретных классов, реализующих интерфейс `StackWalker.StackFrame`, с которыми можно было бы работать напрямую, не сообщаются. API навигации по стеку возвращает экземпляры интерфейса в ответ на запрос кадра стека.

Интерфейс `StackWalker.StackFrame` содержит следующие методы, большинство которых совпадает с методами класса `StackTraceElement`:

- `int getByteCodeIndex()`
- `String getClassName()`
- `Class<?> getDeclaringClass()`
- `String getFileName()`
- `int getLineNumber()`
- `String getMethodName()`
- `boolean isNativeMethod()`
- `StackTraceElement toStackTraceElement()`

Каждый метод описывается в файле класса структурой `method_info`, в которой имеется таблица атрибутов, содержащая атрибут переменной длины `Code`. В этом атрибуте хранится массив `code` с байт-кодом метода. Метод `getByteCodeIndex()` возвращает индекс в массиве `code` метода, содержащего точку выполнения, представленную данным кадром. Для платформенных методов возвращается -1. Дополнительные сведения о массиве `code` и атрибуте `Code` см. в разделе 4.7.3 «Спецификации виртуальной машины Java» по адресу https://docs.oracle.com/javase/specs/jvms/se8/html/.

Как работать с массивом `code` метода? Прикладному программисту индекс точки выполнения в байт-коде не нужен. JDK поддерживает чтение файла класса и всех его атрибутов с помощью внутренних API. Узнать индекс каждой команды в байт-коде метода позволяет программа `javap`, находящаяся в каталоге `JDK_HOME\bin`. Для распечатки массива `code` методов нужно запустить ее с флагом `-c`. Следующая команда выводит массив `code` для всех методов класса `LegacyStackWalk`:

```
C:\Java9Revealed>javap -c com.jdojo.stackwalker\build\classes\com\jdojo\stackwalker\
LegacyStackWalk.class
```

```
Compiled from "LegacyStackWalk.java"
public class com.jdojo.stackwalker.LegacyStackWalk {
public com.jdojo.stackwalker.LegacyStackWalk();
  Code:
    0: aload_0
    1: invokespecial   #1          // Method java/lang/Object."<init>":()V
    4: return

public static void main(java.lang.String[]);
  Code:
    0: invokestatic    #2          // Method m1:()V
    3: return

public static void m1();
```

```
  Code:
    0: invokestatic    #3          // Method m2:()V
    3: return

public static void m2();
  Code:
    0: getstatic       #4      // Field java/lang/System.out:Ljava/io/PrintStream;
    3: ldc #5                  // String \nБез использования рефлексии:
    5: invokevirtual   #6      // Method java/io/PrintStream.println:(Ljava/
lang/String;)V
    8: invokestatic    #7      // Method m3:()V
...
    32: anewarray      #13     // class java/lang/Object
    35: invokevirtual  #14     // Method java/lang/reflect/Method.
invoke:(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;
...
public static void m3();
  Code:
    0: invokestatic    #20     // Method java/lang/Thread.currentThread:()
Ljava/lang/Thread;
    3: invokevirtual   #21     // Method java/lang/Thread.getStackTrace:()
[Ljava/lang/StackTraceElement;
...
}
```

Полужирным шрифтом выделены точки выполнения в каждом методе `main()`, `m1()`, `m2()`, `m3()`, когда снимок стека вызовов сделан в методе `m3()`. Обратите внимание, что `m2()` вызывает `m3()` дважды: в первый раз индекс байт-кода равен 8, а во второй – 35.

Метод `getDeclaringClass()` возвращает ссылку на объект `Class` класса, в котором объявлен метод, представленный кадром. Он возбуждает исключение `UnsupportedOperationException`, если для данного объекта `StackWalker` не был задан параметр `RETAIN_CLASS_REFERENCE`.

Метод `toStackTraceElement()` возвращает экземпляр класса `StackTraceElement`, представляющий тот же кадр стека. Этот метод удобен, если вы хотите использовать API из JDK 9 для получения `StackWalker.StackFrame`, но при этом сохранить старый код анализа кадра, в котором использовался класс `StackTraceElement`.

Получение экземпляра StackWalker

В классе `StackWalker` имеются следующие статические фабричные методы, возвращающие экземпляр `StackWalker`:

- `StackWalker getInstance()`
- `StackWalker getInstance (StackWalker.Option option)`
- `StackWalker getInstance (Set<StackWalker.Option> options)`
- `StackWalker getInstance (Set<StackWalker.Option> options, int estimateDepth)`

Различные варианты метода `getInstance()` позволяют по-разному сконфигурировать объект `StackWalker`. По умолчанию исключаются все скрытые кадры и не сохраняются ссылки на классы. Для включения этих режимов пользуйтесь вариантами с параметром `StackWalker.Option`.

Аргумент `estimateDepth` – предположение об ожидаемом числе кадров, чтобы `StackWalker` мог оптимизировать размер внутреннего буфера.

В следующем фрагменте создается четыре экземпляра класса `StackWalker` с различными конфигурациями:

```
import java.util.Set;
import static java.lang.StackWalker.Option.*;
...

// Получить StackWalker в конфигурации по умолчанию
StackWalker sw1 = StackWalker.getInstance();

// Получить StackWalker, показывающий кадры рефлексии
StackWalker sw2 = StackWalker.getInstance(SHOW_REFLECT_FRAMES);

// Получить StackWalker, показывающий все скрытые кадры
StackWalker sw3 = StackWalker.getInstance(SHOW_HIDDEN_FRAMES);

// Получить StackWalker, показывающий кадры рефлексии и сохраняющий ссылки на классы
StackWalker sw4 = StackWalker.getInstance(
Set.of(SHOW_REFLECT_FRAMES, RETAIN_CLASS_REFERENCE));
```

> **Совет.** Объект `StackWalker` потокобезопасный и допускает повторное использование. Несколько потоков могут использовать один и тот же экземпляр для навигации по своим стекам.

Навигация по стеку

Теперь посмотрим, как обойти кадры стека потока. В классе `StackWalker` есть два метода для навигации по стеку *текущего потока*:

- `void forEach(Consumer<? super StackWalker.StackFrame> action)`
- `<T> T walk(Function<? super Stream<StackWalker.StackFrame>,? extends T> function)`

Метод `forEach()` используется, когда нужно обойти весь стек. Указанному объекту `Consumer` передается по одному кадру, начиная с верхнего. В следующем фрагменте печатаются сведения о каждом кадре, возвращенном объектом `StackWalker`:

```
// Напечатать сведения обо всех кадрах стека текущего потока
Prints the details of all stack frames of the current thread
StackWalker.getInstance()
           .forEach(System.out::println);
```

Метод `walk()` применяется, если к потоку нужно применить, например, средства фильтрации и отображения. Он принимает объект `Function`, который получает

в качестве аргумента поток `Stream<StackWalker.StackFrame>` и может вернуть объект любого типа. `StackWalker` создает поток кадров стека и передает его вашей функции. По выходе из функции поток закрывается. Поток, переданный методу `walk()`, можно обойти только один раз. При попытке обойти его снова возбуждается исключение `IllegalStateException`.

В следующем фрагменте мы с помощью метода `walk()` обходим весь стек и печатаем сведения о каждом кадре. Результат тот же самый, что в предыдущем фрагменте, где использовался метод `forEach()`.

```
// Напечатать сведения обо всех кадрах стека текущего потока
StackWalker.getInstance()
           .walk(s -> {
               s.forEach(System.out::println);
               return null;
           });
```

> **Совет.** Метод `forEach()` класса `StackWalker` используется для обработки кадров по одному, а метод `walk()` — для обработки всего стека как потока кадров. Метод `walk()` позволяет смоделировать функциональность `forEach()`, но обратное неверно.

Спрашивается, почему метод `walk()` передает поток кадров заданной функции, вместо того чтобы просто вернуть его? Это сделано сознательно. Элементы потока вычисляются лениво. После того как поток кадров создан, JVM вправе реорганизовать стек, и не будет никакого способа обнаружить, что стек изменился за время, пока вы удерживаете ссылку на него. Именно поэтому создание и закрытие потока кадров стека контролируется классом `StackWalker`.

Сейчас я покажу несколько иллюстративных примеров использования метода `walk()`, а полная программа будет приведена ниже. В следующем фрагменте мы получаем снимок кадров стека текущего потока в списке `List`.

```
import java.lang.StackWalker.StackFrame;
import java.util.List;
import static java.util.stream.Collectors.toList;
...
List<StackFrame> frames = StackWalker.getInstance()
                                     .walk(s -> s.collect(toList()));
```

А здесь мы помещаем в список строковые представления всех кадров стека за исключением тех, что относятся к вызовам методов с именами, начинающимися с `m2`:

```
import java.util.List;
import static java.util.stream.Collectors.toList;
...
List<String> list = StackWalker.getInstance()
  .walk(s -> s.filter(f -> !f.getMethodName().startsWith("m2"))
              .map(f -> f.toString())
              .collect(toList())
        );
```

А этот код помещает в список строковые представления всех кадров стека за исключением тех, что относятся к вызовам методов, объявленных в классах с именами, оканчивающимися на `Test`:

```
import static java.lang.StackWalker.Option.RETAIN_CLASS_REFERENCE;
import java.util.List;
import static java.util.stream.Collectors.toList;
...
List<String> list = StackWalker
  .getInstance(RETAIN_CLASS_REFERENCE)
  .walk(s -> s.filter(f -> !f.getDeclaringClass()
                            .getName().endsWith("Test"))
              .map(f -> f.toString())
              .collect(toList())
        );
```

А теперь соберем весь стек в одну строку, разделяя кадры платформенно-зависимым разделителем:

```
import static java.util.stream.Collectors.joining;
...
String stackStr = StackWalker.getInstance()
  .walk(s -> s.map(f -> f.toString())
              .collect(joining (System.getProperty("line.separator"))
        ));
```

В листинге 16.3 приведен полный код программы, демонстрирующей использование класса `StackWalker` и его метода `walk()`. Метод `main()` вызывает метод `m1()` дважды, с разными параметрами `StackWalker`. Метод `m2()` применяет рефлексию для вызова метода `m3()`, который печатает сведения о кадре стека. При первом вызове `m1()` кадры рефлексии скрыты и ссылки на классы опущены.

Листинг 16.3. Использование StackWalker для обхода кадров стека текущего потока

```
// StackWalking.java
package com.jdojo.stackwalker;

import java.lang.StackWalker.Option;
import static java.lang.StackWalker.Option.RETAIN_CLASS_REFERENCE;
import static java.lang.StackWalker.Option.SHOW_REFLECT_FRAMES;
import java.lang.StackWalker.StackFrame;
import java.lang.reflect.InvocationTargetException;
import java.util.Set;
import java.util.stream.Stream;

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

```
        m1(Set.of());

        System.out.println();

        // Оставить ссылки на классы и показать кадры рефлексии
        m1(Set.of(RETAIN_CLASS_REFERENCE, SHOW_REFLECT_FRAMES));
    }

    public static void m1(Set<Option> options) {
        m2(options);
    }

    public static void m2(Set<Option> options) {
        // Вызвать m3() с помощью рефлексии
        try {
            System.out.println("Параметры StackWalker: " + options);
            StackWalking.class
                 .getMethod("m3", Set.class)
                 .invoke(null, options);
        } catch (NoSuchMethodException
                 | InvocationTargetException
                 | IllegalAccessException
                 | SecurityException e) {
            e.printStackTrace();
        }
    }

    public static void m3(Set<Option> options) {
        // Напечатать сведения о стеке вызовов
        StackWalker.getInstance(options)
                    .walk(StackWalking::processStack);
    }

    public static Void processStack(Stream<StackFrame> stack) {
        stack.forEach(frame -> {
            int bci = frame.getByteCodeIndex();
            String className = frame.getClassName();
            Class<?> classRef = null;
            try {
                classRef = frame.getDeclaringClass();
            } catch (UnsupportedOperationException e) {
                // Ничего не делать
            }

            String fileName = frame.getFileName();
            int lineNumber = frame.getLineNumber();
            String methodName = frame.getMethodName();
```

```
            boolean isNative = frame.isNativeMethod();

            StackTraceElement sfe = frame.toStackTraceElement();

            System.out.printf("Платформенный метод=%b", isNative);
            System.out.printf(", индекс байт-кода=%d", bci);
            System.out.printf(", имя модуля=%s", sfe.getModuleName());
            System.out.printf(", версия модуля=%s", sfe.getModuleVersion());
            System.out.printf(", имя класса=%s", className);
            System.out.printf(", ссылка на класс=%s", classRef);
            System.out.printf(", имя файла=%s", fileName);
            System.out.printf(", номер строки=%d", lineNumber);
            System.out.printf(", имя метода=%s.%n", methodName);
        });

        return null;
    }
}
```

```
Параметры StackWalker: []
Платформенный метод=false, Индекс байт-кода=9, Имя модуля=null, Версия модуля=null, Имя
класса=com.jdojo.stackwalker.StackWalking, Ссылка на класс=null, Имя файла=StackWalking.java,
Номер строки=44, Имя метода=m3.
Платформенный метод=false, Индекс байт-кода=37, Имя модуля=null, Версия модуля=null, Имя
класса=com.jdojo.stackwalker.StackWalking, Ссылка на класс=null, Имя файла=StackWalking.java,
Номер строки=32, Имя метода=m2.
Платформенный метод=false, Индекс байт-кода=1, Имя модуля=null, Версия модуля=null, Имя
класса=com.jdojo.stackwalker.StackWalking, Ссылка на класс=null, Имя файла=StackWalking.java,
Номер строки=23, Имя метода=m1.
Платформенный метод=false, Индекс байт-кода=3, Имя модуля=null, Версия модуля=null, Имя
класса=com.jdojo.stackwalker.StackWalking, Ссылка на класс=null, Имя файла=StackWalking.java,
Номер строки=14, Имя метода=main.

Параметры StackWalker: [SHOW_REFLECT_FRAMES, RETAIN_CLASS_REFERENCE]
Платформенный метод=false, Индекс байт-кода=9, Имя модуля=null, Версия модуля=null, Имя
класса=com.jdojo.stackwalker.StackWalking, Ссылка на класс=class com.jdojo.stackwalker.
StackWalking, Имя файла=StackWalking.java, Номер строки=44, Имя метода=m3.
Платформенный метод=true, Индекс байт-кода=-1, Имя модуля=java.base, Версия модуля=9-ea,
Имя класса=jdk.internal.reflect.NativeMethodAccessorImpl, Ссылка на класс=class jdk.
internal.reflect.NativeMethodAccessorImpl, Имя файла=NativeMethodAccessorImpl.java,
Номер строки=-2, Имя метода=invoke0.
Платформенный метод=false, Индекс байт-кода=100, Имя модуля=java.base, Версия модуля=9-ea,
Имя класса=jdk.internal.reflect.NativeMethodAccessorImpl, Ссылка на класс=class jdk.internal.
reflect.NativeMethodAccessorImpl, Имя файла=NativeMethodAccessorImpl.java, Номер строки=62,
Имя метода=invoke.
Платформенный метод=false, Индекс байт-кода=6, Имя модуля=java.base, Версия модуля=9-ea,
```

```
Имя класса=jdk.internal.reflect.DelegatingMethodAccessorImpl, Ссылка на класс=class jdk.
internal.reflect.DelegatingMethodAccessorImpl, Имя файла=DelegatingMethodAccessorImpl.
java, Номе строки=43, Имя метода=invoke.
Платформенный метод=false, Индекс байт-кода=59, Имя модуля=java.base, Версия модуля=9-ea,
Имя класса=java.lang.reflect.Method, Ссылка на класс=class java.lang.reflect.Method,
Имя файла=Method.java, Номер строки=538, Имя метода=invoke.
Платформенный метод=false, Индекс байт-кода=37, Имя модуля=null, Версия модуля=null, Имя
класса=com.jdojo.stackwalker.StackWalking, Ссылка на класс=class com.jdojo.stackwalker.
StackWalking, Имя файла=StackWalking.java, Номер строки=32, Имя метода=m2.
Платформенный метод=false, Индекс байт-кода=1, Имя модуля=null, Версия модуля=null, Имя
класса=com.jdojo.stackwalker.StackWalking, Ссылка на класс=class com.jdojo.stackwalker.
StackWalking, Имя файла=StackWalking.java, Номер строки=23, Имя метода=m1.
Платформенный метод=false, Индекс байт-кода=21, Имя модуля=null, Версия модуля=null, Имя
класса=com.jdojo.stackwalker.StackWalking, Ссылка на класс=class com.jdojo.stackwalker.
StackWalking, Имя файла=StackWalking.java, Номер строки=19, Имя метода=main.
```

Получение вызывающего класса

До JDK 9 для получения вызывающего класса у разработчика были следующие возможности:

- метод `getClassContext()` класса `SecurityManager`, который приходилось расширять, потому что этот метод защищенный;
- метод `getCallerClass()` внутреннего класса JDK `sun.reflect.Reflection`.

В JDK 9 получение ссылки на вызывающий класс стало проще благодаря добавлению метода `getCallerClass()` в класс `StackWalker`. Метод возвращает значение типа `Class<?>`. Если при создании экземпляра `StackWalker` не был задан параметр `RETAIN_CLASS_REFERENCE`, то этот метод возбуждает исключение `UnsupportedOperationException`. Если в стеке нет вызывающего кадра, например, когда этот метод вызван из `main()`, то возбуждается исключение `IllegalStateException`.

Какой класс считается вызывающим? В языке Java есть две конструкции, допускающие вызов, – методы и конструкторы. В обсуждении ниже употребляется термин «метод», но все сказанное в равной мере относится и к конструкторам. Предположим, что метод `getCallerClass()` вызван рефлексивно из метода `S`, а тот вызван из метода `T`. Предположим также, что метод `T` находится в классе `C`. В таком случае вызывающим является класс `C`.

> **Совет.** Метод `getCallerClass()` класса `StackWalker` отфильтровывает все скрытые кадры и кадры рефлексии при поиске вызывающего класса вне зависимости от параметров, указанных при получении экземпляра `StackWalker`.

В листинге 16.4 приведен полный код программы, демонстрирующей получение вызывающего класса. Метод `main()` вызывает метод `m1()`, тот вызывает `m2()`, который, в свою очередь, вызывает `m3()`. Метод `m3()` получает экземпляр класса `StackWalker` и находит вызывающий класс. Отметим, что метод `m2()` пользуется для вызова `m3()` рефлексией. В самом конце получить вызывающий класс пытается

метод `main()`. При выполнении класса `CallerClassTest` его метод `main()` вызывается из JVM, поэтому в стеке нет вызывающего кадра и возникает исключение `IllegalStateException`.

Листинг 16.4. Получение ссылки на вызывающий класс с помощью класса StackWalker

```
// CallerClassTest.java
package com.jdojo.stackwalker;

import java.lang.StackWalker.Option;
import static java.lang.StackWalker.Option.RETAIN_CLASS_REFERENCE;
import static java.lang.StackWalker.Option.SHOW_REFLECT_FRAMES;
import java.lang.reflect.InvocationTargetException;
import java.util.Set;

public class CallerClassTest {
   public static void main(String[] args) {
      /* Получить вызывающий класс не удастся, потому что не задан параметр
         RETAIN_CLASS_REFERENCE.
      */
      m1(Set.of());

      // Будет напечатан вызывающий класс
      m1(Set.of(RETAIN_CLASS_REFERENCE, SHOW_REFLECT_FRAMES));

      try {
         /* Следующее предложение возбуждает исключение IllegalStateException, потому
            что вызывающего класса нет; этот метод вызывается JVM. Но если бы метод
            main() вызывался из программы, то исключение не возникло бы.
         */
         Class<?> cls = StackWalker.getInstance(RETAIN_CLASS_REFERENCE)
                                   .getCallerClass();
         System.out.println("В методе main, вызывающий класс: " + cls.getName());
      } catch (IllegalCallerException e) {
         System.out.println("В методе main, исключение: " + e.getMessage());
      }
   }

   public static void m1(Set<Option> options) {
      m2(options);
   }

   public static void m2(Set<Option> options) {
      // Вызов m3() с помощь рефлексии
      try {
         CallerClassTest.class
```

```
                        .getMethod("m3", Set.class)
                        .invoke(null, options);
        } catch (NoSuchMethodException | InvocationTargetException
                 | IllegalAccessException | SecurityException e) {
            e.printStackTrace();
        }
    }

    public static void m3(Set<Option> options) {
        try {
            // Напечатать имя вызывающего класса
            Class<?> cls = StackWalker.getInstance(options)
                                      .getCallerClass();
            System.out.println("Вызывающий класс: " + cls.getName());
        } catch (UnsupportedOperationException e) {
            System.out.println("В m3(): " + e.getMessage());
        }
    }
}
```

```
В m3(): This stack walker does not have RETAIN_CLASS_REFERENCE access
Вызывающий класс: com.jdojo.stackwalker.CallerClassTest
В методе main, исключение: no caller frame
```

В этом примере метод, печатающий сведения о кадрах стека, вызывался из другого метода того же класса. А теперь вызовем этот метод из метода другого класса и посмотрим, что получится. В листинге 16.5 приведен код класса `CallerClassTest2`.

Листинг 16.5. Другой пример получения вызывающего класса

```
// CallerClassTest2.java
package com.jdojo.stackwalker;

import java.lang.StackWalker.Option;
import java.util.Set;
import static java.lang.StackWalker.Option.RETAIN_CLASS_REFERENCE;

public class CallerClassTest2 {
    public static void main(String[] args) {
        Set<Option> options = Set.of(RETAIN_CLASS_REFERENCE);
        CallerClassTest.m1(options);
        CallerClassTest.m2(options);
        CallerClassTest.m3(options);

        System.out.println("\nВызывается метод main():");
        CallerClassTest.main(null);
```

```
        System.out.println("\nАнонимный класс:");
        new Object() {
           {
              CallerClassTest.m3(options);
           }
        };

        System.out.println("\nЛямбда-выражение:");
        new Thread(() -> CallerClassTest.m3(options))
           .start();
    }
}
```

```
Вызывающий класс: com.jdojo.stackwalker.CallerClassTest
Вызывающий класс: com.jdojo.stackwalker.CallerClassTest
Вызывающий класс: com.jdojo.stackwalker.CallerClassTest2

Вызывается метод main():
В m3(): This stack walker does not have RETAIN_CLASS_REFERENCE access
Вызывающий класс: com.jdojo.stackwalker.CallerClassTest
В методе main, вызывающий класс: com.jdojo.stackwalker.CallerClassTest2

Анонимный класс:
Вызывающий класс: com.jdojo.stackwalker.CallerClassTest2$1

Лямбда-выражение:
Вызывающий класс: com.jdojo.stackwalker.CallerClassTest2
```

Метод `main()` класса `CallerClassTest2` вызывает все четыре метода `CallerClassTest`. Если `CallerClassTest.m3()` вызван из `CallerClassTest2` непосредственно, то вызывающим классом является `CallerClassTest2`. Если метод `CallerClassTest.main()` вызывается из класса `CallerClassTest2`, то вызывающий кадр существует и вызывающим классом является `CallerClassTest2`. Сравните с тем, что получилось в предыдущем примере, когда мы выполняли класс `CallerClassTest`. Тогда метод `CallerClassTest.main()` вызывался из JVM, и мы не могли получить вызывающий класс, потому что в стеке не было вызывающего кадра. В конце метод `CallerClassTest.m3()` вызывается из анонимного класса и из лямбда-выражения. Анонимный класс считается вызывающим. А в случае лямбда-выражения вызывающим считается объемлющий выражение класс.

Права для навигации по стеку

Если присутствует диспетчер безопасности Java и при создании объекта `StackWalker` был задан параметр `RETAIN_CLASS_REFERENCE`, то проверяется, предоставлено ли

программе право `java.lang.StackFramePermission` со значением `retainClassReference`. Если нет, то возбуждается исключение `SecurityException`. Проверка производится в момент создания объекта `StackWalker`, а не в начале обхода стека.

В листинге 16.6. приведен код класса `StackWalkerPermissionCheck`. Метод `printStackFrames()` создает экземпляр `StackWalker` с параметром `RETAIN_CLASS_REFERENCE`. Метод `main()` вызывает этот метод, который печатает трассу стека, не испытывая никаких проблем. Затем устанавливается диспетчер безопасности и метод `printStackFrames()` вызывается снова. На этот раз возбуждается исключение `SecurityException`.

Листинг 16.6. Создание объекта `StackWalker` в режиме сохранения ссылок на классы в присутствии диспетчера безопасности

```
// StackWalkerPermissionCheck.java
package com.jdojo.stackwalker;

import static java.lang.StackWalker.Option.RETAIN_CLASS_REFERENCE;

public class StackWalkerPermissionCheck {
   public static void main(String[] args) {
      System.out.println("До установки диспетчера безопасности:");
      printStackFrames();

      SecurityManager sm = System.getSecurityManager();
      if (sm == null) {
         sm = new SecurityManager();
         System.setSecurityManager(sm);
      }

      System.out.println("\nПосле установки диспетчера безопасности:");
      printStackFrames();
   }

   public static void printStackFrames() {
      try {
         StackWalker.getInstance(RETAIN_CLASS_REFERENCE)
                    .forEach(System.out::println);
         } catch(SecurityException e){
            System.out.println("Не удалось создать " +
               "StackWalker. Ош: " + e.getMessage());
      }
   }
}
```

```
До установки диспетчера безопасности:
com.jdojo.stackwalker/com.jdojo.stackwalker.StackWalkerPermissionCheck.
```

```
printStackFrames(Stac
kWalkerPermissionCheck.java:24)
com.jdojo.stackwalker/com.jdojo.stackwalker.StackWalkerPermissionCheck.
main(StackWalkerPermissionCheck.java:9)

После установки диспетчера безопасности:
Не удалось создать StackWalker. Ошибка: access denied ("java.lang.StackFramePermission"
"retainClassReference")
```

В листинге 16.7 показано, как предоставить право, требуемое для создания `StackWalker` с параметром `RETAIN_CLASS_REFERENCE`. Право предоставляется всем базам кода. Показанный блок необходимо добавить в конец файла `java.policy` в каталоге `JAVA_HOME\conf\security` на своей машине.

Листинг 16.7. Предоставление права java.lang.StackFramePermission со значением "retainClassReference"

```
grant {
    permission java.lang.StackFramePermission "retainClassReference";
};
```

Если запустить класс из листинга 16.6 с правом, предоставленным в листинге 16.7, то получится такой результат:

```
До установки диспетчера безопасности:
com.jdojo.stackwalker/com.jdojo.stackwalker.StackWalkerPermissionCheck.
printStackFrames(StackWalkerPermissionCheck.java:24)
com.jdojo.stackwalker/com.jdojo.stackwalker.StackWalkerPermissionCheck.
main(StackWalkerPermissionCheck.java:9)

После установки диспетчера безопасности:
com.jdojo.stackwalker/com.jdojo.stackwalker.StackWalkerPermissionCheck.
printStackFrames(StackWalkerPermissionCheck.java:24)
com.jdojo.stackwalker/com.jdojo.stackwalker.StackWalkerPermissionCheck.
main(StackWalkerPermissionCheck.java:18)
```

Резюме

У каждого потока в JVM имеется собственный стек, создаваемый одновременно с потоком. Стек состоит из кадров. Кадр стека JVM представляет вызов метода в данном потоке. При каждом вызове метода создается новый кадр, который помещается на вершину стека. Кадр уничтожается (выталкивается из стека), когда метод возвращает управление. В любой момент времени в каждом потоке активен только один кадр, который называется *текущим кадром*, а соответствующий

метод называется *текущим методом*. Класс, в котором определен текущий метод, называется *текущим классом*.

До JDK 9 для обхода всех кадров в стеке потока можно было пользоваться классами `Throwable`, `Thread` и `StackTraceElement`. Экземпляр класса `StackTraceElement` представляет кадр стека. Метод `getStrackTrace()` класса `Throwable` возвращает массив `StackTraceElement[]`, содержащий все кадры стека текущего потока. Метод `getStackTrace()` класса `Thread` возвращает массив `StackTraceElement[]`, содержащий кадры стека указанного потока. В первом элементе массива находится верхний кадр стека, представляющий последний по порядку вызов метода. В некоторых реализациях JVM в возвращаемом массиве часть кадров может отсутствовать.

В JDK 9 навигация по стеку упростилась. В пакете `java.lang` появился новый класс StackWalker. Для получения экземпляра `StackWalker` используется один из вариантов статического фабричного метода `getInstance()`. Поведение `StackWalker` можно настроить с помощью параметров – элементов перечисления `StackWalker.Option`. Кадр стека представляется экземпляром вложенного интерфейса `StackWalker.StackFrame`. В этом интерфейсе определен метод `toStackTraceElement()`, с помощью которого можно получить объект `StackTraceElement`, соответствующий `StackWalker.StackFrame`.

Для обхода кадров стека текущего потока предназначены методы `forEach()` и `walk()`. Метод `getCallerClass()` экземпляра `StackWalker` возвращает ссылку на вызывающий класс. Чтобы получать ссылки на класс, представленный самим кадром стека, и на вызывающий класс, при создании экземпляра необходимо задать параметр `RETAIN_CLASS_REFERENCE`. По умолчанию `StackWalker` не сообщает о кадрах рефлексии и кадрах, зависящих от реализации. Чтобы включить и их тоже, необходимо при конфигурировании `StackWalker` задавать параметры `SHOW_REFLECT_FRAMES` и `SHOW_HIDDEN_FRAMES`. Параметр `SHOW_HIDDEN_FRAMES` подразумевает также включение кадров рефлексии.

Если присутствует диспетчер безопасности Java и при создании экземпляра `StackWalker` задается параметр `RETAIN_CLASS_REFERENCE`, то кодовой базе должно быть предоставлено право `java.lang.StackFramePermission` со значением `retainClassReference`. В противном случае будет возбуждено исключение `SecurityException`. Наличие права проверяется в момент создания объекта `StackWalker`, а не в начале обхода стека.

Глава 17

Реактивные потоки

Краткое содержание главы:

- что такое поток;
- в чем суть инициативы Reactive Streams, ее спецификации и соответствующего Java API;
- API реактивных потоков в JDK;
- создание издателей, подписчиков и процессоров с помощью API реактивных потоков в JDK 9.

Что такое поток?

Потоком называется последовательность элементов, порождаемых производителем и потребляемых одним или несколькими потребителями. У модели производитель-потребитель есть и другие названия: источник–сток или издатель–подписчик. В этой главе я буду употреблять название издатель–подписчик. Слова «элемент», «элемент данных» и «данные» в этом контексте являются синонимами и обозначают квант информации, публикуемый издателем и получаемый подписчиками.

Существует несколько механизмов обработки потоков, наиболее распространенными являются модель вытягивания (pull) и модель проталкивания (push). В случае проталкивания издатель отправляет данные подписчику, а в случае вытягивания подписчик сам забирает данные у издателя. Эти модели прекрасно приспособлены к идеальной ситуации, когда издатель и подписчик работают в одинаковом темпе. Мы рассмотрим несколько ситуаций, когда это не так, опишем, какие при этом возникают проблемы и как их можно решать.

Если издатель быстрее подписчика, то последний должен располагать неограниченным буфером для хранения быстро поступающих элементов или отбрасывать те, которые не успевает обработать. Другое решение – применить стратегию *противодавления* (backpressure), когда подписчик просит издателя снизить темп и придержать данные, пока подписчик не будет готов их обработать. Стратегия противодавления может вынудить издателя поддерживать неограниченный буфер, если он продолжает производить данные и должен их сохранять. Но издатель может вместо этого завести ограниченный буфер и отбрасывать новые элементы

в случае его переполнения. Другая стратегия издателя – повторно публиковать элементы для подписчика, который не смог принять данные в момент первоначального опубликования.

Что может сделать подписчик, если в момент запроса данных у издателя ничего нет? Если запрос синхронный, то подписчик должен ждать, быть может, неопределенно долго, пока данные не появятся. Если издатель отправляет подписчику данные синхронно и подписчик обрабатывает их также синхронно, то издатель будет заблокирован до момента завершения обработки. Решение состоит в том, чтобы организовать асинхронную обработку на обеих сторонах, так чтобы подписчик мог запросить данные у издателя и заняться другими делами, пока они не пришли. Когда у издателя появляются новые данные, он асинхронно отправляет их подписчику.

Что такое реактивные потоки?

В 2013 году была выдвинута инициатива Reactive Streams (реактивные потоки), ставившая целью разработку стандарта *асинхронной* обработки потоков с *неблокирующим противодавлением*. Ее предметом было решение задач обработки потоков данных: как передать поток от издателя подписчику, избавив издателя от блокировки, а подписчика от неограниченного роста буфера или необходимости отбрасывать данные.

Модель реактивных потоков очень проста. Подписчик отправляет издателю асинхронный запрос на `N` элементов данных. Издатель асинхронно отправляет подписчику `N` или менее элементов.

> **Совет.** В реактивных потоках происходит динамическое переключение между моделями проталкивания и вытягивания. Модель вытягивания применяется, когда быстрее работает подписчик, а модель проталкивания – когда быстрее издатель.

В 2015 году была опубликована спецификация и Java API для работы с реактивными потоками. Дополнительные сведения об инициативе Reactive Streams см. на сайте `http://www.reactive-streams.org/`. API реактивных потоков в Java состоит всего из четырех интерфейсов:

- `Publisher<T>`
- `Subscriber<T>`
- `Subscription`
- `Processor<T,R>`

Издатель порождает потенциально неограниченное число *последовательных* элементов данных. Он публикует (отправляет) элементы своим текущим подписчикам в соответствии с их требованиями.

Подписчик подписывается на получение данных от издателя. Издатель отправляет подписчику маркер подписки. Имея этот маркер, подписчик запрашивает `N` элементов данных у издателя. Когда данные будут готовы, издатель отправляет подписчику `N` или меньше элементов. Подписчик может запросить дополнительные элементы. У издателя может быть более одного необработанного запроса от подписчика.

Подписка – это представление маркера, отправленного издателем подписчику. Издатель отправляет маркер, если запрос на подписку удовлетворен. Подписчик использует подписку – экземпляр `Subscription` – для взаимодействия с издателем, когда хочет запросить дополнительные элементы или отказаться от подписки.

На рис. 17.1 показано типичное взаимодействие между издателем и подписчиком. Сама подписка не показана. Не показаны также ошибки и события отмены.

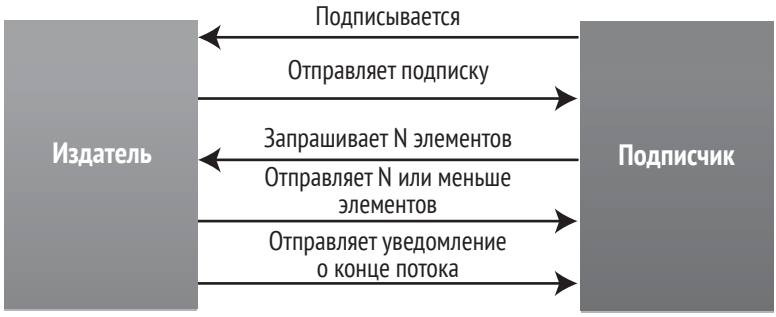

Рис. 17.1. Типичная последовательность взаимодействий между издателем и подписчиком

Процессор представляет этап обработки, выступающий одновременно в роли подписчика и издателя. Интерфейс `Processor` расширяет оба интерфейса: `Publisher` и `Subscriber`. Он служит для преобразования элементов данных в конвейере между издателем и подписчиком. Объект `Processor<T,R>` подписывается на элементы типа `T`, получает их, преобразует в тип `R` и публикует. На рис. 17.2 показано место процессора в конвейере. Процессоров в конвейере может быть несколько.

Рис. 17.2. Процессор как преобразователь данных в конвейере издатель-подписчик

API реактивных потоков в Java, согласующийся с инициативой Reactive Streams, показан в листинге 17.1. Отметим, что все методы имеют тип `void`, поскольку представляют либо асинхронный запрос, либо асинхронное уведомление о событии. В следующем разделе я объясню, как этот API включен в JDK 9.

Листинг 17.1. API реактивных потоков в Java

```
public interface Publisher<T> {
  public void subscribe(Subscriber<? super T> s);
}

public interface Subscriber<T> {
   public void onSubscribe(Subscription s);
   public void onNext(T t);
   public void onError(Throwable t);
   public void onComplete();
```

```
}

public interface Subscription {
   public void request(long n);
   public void cancel();
}

public interface Processor<T,R> extends Subscriber<T>, Publisher<R> {
}
```

API реактивных потоков очень просто понять. Но вот реализовать его далеко не просто. Реализация усложняется из-за асинхронной природы всех взаимодействий между издателями и подписчиками и необходимостью учитывать противодавление. Предполагается, что широкий круг практических ситуаций будет реализован в библиотеках, и прикладным программистам не придется этим заниматься. В JDK 9 предлагается простая реализация интерфейса `Publisher`, которую можно использовать как есть или расширить. Одну из полных реализаций реактивных потоков дает библиотека RxJava (https://github.com/ReactiveX/RxJava)[1].

API реактивных потоков в JDK 9

JDK 9 предлагает API, совместимый со стандартом Reactive Streams, в пакете `java.util.concurrent`, находящемся в модуле `java.base`. API включает два класса:

- `Flow`
- `SubmissionPublisher<T>`

Класс `Flow` финальный. Он инкапсулирует API реактивных потоков и статический метод. Все четыре интерфейса, определенные в спецификации API, включены в класс `Flow` в виде вложенных статических интерфейсов:

- `Flow.Processor<T,R>`
- `Flow.Publisher<T>`
- `Flow.Subscriber<T>`
- `Flow.Subscription`

Они содержат методы, перечисленные в листинге 17.1. В классе `Flow` имеется также статический метод `defaultBufferSize()`, который возвращает подразумеваемый по умолчанию размер буфера, используемого издателями и подписчиками. В настоящее время он равен 256.

Класс `SubmissionPublisher<T>` реализует интерфейс `Flow.Publisher<T>`. Он также реализует интерфейс `AutoCloseable`, поэтому его экземплярами можно управлять с помощью блока `try` с ресурсами. В JDK 9 нет реализации интерфейса `Flow.Subscriber<T>`; это вам придется делать самостоятельно. Однако в классе `SubmissionPublisher<T>` имеется метод `consume(Consumer<? super T> consumer)`, которым можно воспользоваться для обработки всех элементов, публикуемых данным издателем. Примеры будут приведены позже.

---

[1] См. также Нуркевич Т., Кристенсен Б. «Реактивное программирование с применением RxJava». – ДМК Пресс, 2017. – *Прим. перев.*

Взаимодействия между издателем и подписчиком

Прежде чем начинать использовать API, важно разобраться в последовательности событий, возникающих в типичном сеансе работы с реактивными потоками. Я включил в описание имена методов, принимающих участие в каждом событии. У издателя может быть нуль или более подписчиков, но в обсуждении ниже предполагается, что подписчик ровно один.

- Мы создаем издателя и подписчика, являющихся соответственно экземплярами интерфейсов `Flow.Publisher` и `Flow.Subscriber`.
- Подписчик пытается подписаться, вызывая метод издателя `subscribe()`. В случае успеха издатель асинхронно вызывает метод подписчика `onSubscribe()`, передавая ему объект `Flow.Subscription`. Если подписаться не получилось, то вызывается метод подписчика `onError()`, которому передается исключение `IllegalStateException`, и на этом взаимодействие между издателем и подписчиком завершается.
- Подписчик отправляет издателю запрос на получение `N` элементов данных, вызывая метод `request(N)` объекта `Subscription`. Подписчик может отправить несколько таких запросов, не дожидаясь выполнения ранее отправленных.
- Издатель вызывает метод подписчика `onNext(T item)` столько раз, сколько элементов запросил подписчик (или меньше), и при каждом вызове отправляет один элемент. Если у издателя больше нет элементов для подписчика, то он вызывает метод подписчика `onComplete()`, сигнализируя о конце потока, и на этом взаимодействие завершается. Если подписчик запросил `Long.MAX_VALUE` элементов, то это означает по существу неограниченный запрос, и мы имеем модель проталкивания.
- Если издатель в какой-то момент обнаруживает ошибку, то он вызывает метод подписчика `onError()`.
- Подписчик может отказаться от подписки, вызвав метод `cancel()` объекта `Flow.Subscription`. После отказа от подписки взаимодействие между издателем и подписчиком завершается. Однако и после отказа от подписки подписчик может получать данные, если к моменту отказа еще оставались не полностью обработанные запросы.

Таким образом, после вызова метода `onComplete()` или `onError()` подписчик больше не получает уведомлений от издателя.

После вызова метода издателя `subscribe()` гарантируется, что подписчик будет видеть следующую последовательность событий, если только не откажется от подписки:

```
onSubscribe onNext* (onError | onComplete)?
```

Здесь символы `*` и `?` интерпретируются, как в регулярном выражении: `*` означает «нуль или больше», а `?` – «нуль или один».

Первым вызванным методом подписчика всегда является `onSubscribe()`, это уведомление издателя об успешной подписке. Метод `onNext()` может вызываться нуль или более раз – при каждой публикации элемента данных. Может быть вызван ровно один из методов `onComplete()` или `onError()` – и только один раз, это означает

переход в конечное состояние; но эти методы вызываются, только если подписчик не отказался от подписки.

Создание издателя

Для создания издателя должен быть реализован интерфейс `Flow.Publisher<T>`. Я рассмотрю реализующий его класс `SubmissionPublisher<T>`. В этом классе имеются следующие конструкторы:

- `SubmissionPublisher()`
- `SubmissionPublisher(Executor executor, int maxBufferCapacity)`
- `SubmissionPublisher(Executor executor, int maxBufferCapacity, BiConsumer<? super Flow.Subscriber<? super T>,? super Throwable> handler)`

Класс `SubmissionPublisher` использует переданный конструктору объект `Executor` для доставки данных подписчикам. Если публикуемые данные порождаются несколькими потоками и число подписчиков можно оценить заранее, то можно использовать `Executor` с фиксированным пулом потоков, который возвращается статическим методом `newFixedThreadPool(int nThread)` класса `Executor`. В противном случае используется `Executor` по умолчанию, полученный от метода `commonPool()` класса `ForkJoinPool`.

В классе `SubmissionPublisher` для каждого подписчика создается отдельный буфер, размер которого определяется аргументом конструктора `maxBufferCapacity`. Размер буфера по умолчанию возвращает статический метод `defaultBufferSize()` класса `Flow`, он равен 256. Если число опубликованных элементов превышает размер буфера подписчика, то не помещающиеся элементы отбрасываются. Текущий размер буфера каждого подписчика возвращает метод `getMaxBufferCapacity()` класса `SubmissionPublisher`.

Если какой-то метод подписчика возбуждает исключение, то его подписка аннулируется. Если исключение возбуждает метод подписчика `onNext()`, то перед аннулированием подписки вызывается обработчик, заданный в аргументе `handler` конструктора. По умолчанию обработчик равен `null`.

В следующем фрагменте создается объект `SubmissionPublisher`, который публикует элементы типа `Long`; все его атрибуты принимают значения по умолчанию:

```
// Создать издателя, который публикует значение типа Long
SubmissionPublisher<Long> pub = new SubmissionPublisher<>();
```

Класс `SubmissionPublisher` реализует интерфейс `AutoCloseable`. При вызове его метода `close()` вызывается метод `onComplete()` всех текущих подписчиков. Попытка опубликовать элемент после вызова метода `close()` приводит к исключению `IllegalStateException`.

Публикация данных

В классе `SubmissionPublisher<T>` имеются следующие методы для публикации данных:

- `int offer(T item, long timeout, TimeUnit unit, BiPredicate<Flow.Subscriber<? super T>,? super T> onDrop)`

- `int offer(T item, BiPredicate<Flow.Subscriber <? super T>,? super T> onDrop)`
- `int submit(T item)`

Метод `submit()` блокирует выполнение, пока не появятся ресурсы для публикации элемента текущим подписчикам. Рассмотрим случай, когда для каждого подписчика выделен буфер на 10 элементов. Подписчик оформляет подписку, но не запрашивает данные. Издатель публикует 10 элементов и помещает все их в буфер. Попытка издателя опубликовать еще один элемент методом `submit()` приведет к блокировке, потому что буфер подписчика полон.

Метод `offer()` неблокирующий. Его первый вариант позволяет задать таймаут, по истечении которого элемент отбрасывается. Можно также задать обработчик отбрасывания – объект тип `BiPredicate`. Перед тем как отбросить предназначенный подписчику элемент, вызывается метод `test()` обработчика. Если он возвращает `true`, то производится еще одна попытка отправить элемент, а если `false`, то элемент отбрасывается без повторной попытки. Отрицательное число, возвращенное методом `offer()`, равно число неудачных попыток доставить элемент подписчику, а положительное дает оценку максимального числа элементов, которые были отправлены всем текущим подписчикам, но еще не потреблены ими.

Каким методом публиковать элемент: `submit()` или `offer()`? Зависит от требований. Если каждый опубликованный элемент необходимо доставить всем подписчикам, то следует предпочесть `submit()`. Если вы готовы подождать в течение некоторого времени и повторить попытку, то лучше выбрать `offer()`.

Простой пример

Рассмотрим простой пример использования класса `SubmissionPublisher`. Объект `SubmissionPublisher` может публиковать данные методом `submit(T item)`. В следующем фрагменте порождается и публикуется пять целых чисел (1, 2, 3, 4, 5) в предположении, что `pub` – ссылка на объект `SubmissionPublisher`:

```
// Порождается и публикуется 5 целых чисел
LongStream.range(1L, 6L)
          .forEach(pub::submit);
```

Для потребления элементов, публикуемых издателем, необходим подписчик. В классе `SubmissionPublisher` имеется метод `consume(Consumer<? super T> consumer)`, позволяющий добавить подписчика, готового обрабатывать все опубликованные элементы и не интересующегося никакими уведомлениями об ошибках и конце потока. Этот метод возвращает объект `CompletableFuture<Void>`, который становится завершенным, когда издатель вызывает метод `onComplete()` подписчика. Ниже показано, как добавить потребителя, который является внутренним подписчиком:

```
// Добавить подписчика, который печатает опубликованные элементы
CompletableFuture<Void> subTask = pub.consume(System.out::println);
```

Код в этой главе, – часть модуля `com.jdojo.stream`, объявление которого приведено в листинге 17.2.

Листинг 17.2. Объявление модуля com.jdojo.stream

```
// module-info.java
module com.jdojo.stream {
    exports com.jdojo.stream;
}
```

В листинге 17.3 показан код класса `NumberPrinter`, демонстрирующий использование класса `SubmissionPublisher` для публикации целых чисел.

Листинг 17.3. Пример издателя и подписчика — публикуется и печатается пять целых чисел

```
// NumberPrinter.java
package com.jdojo.stream;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.SubmissionPublisher;
import java.util.stream.LongStream;

public class NumberPrinter {
   public static void main(String[] args) {
      CompletableFuture<Void> subTask = null;

      // По выходе из блока try издатель закрывается
      try (SubmissionPublisher<Long> pub = new SubmissionPublisher<>()) {
         // Напечатать размер буфера каждого подписчика
         System.out.println("Размер буфера подписчика: " +
                            pub.getMaxBufferCapacity());

         // Добавить подписчика, печатающего опубликованные элементы
         subTask = pub.consume(System.out::println);

         // Сгенерировать и опубликовать пять целых чисел
         LongStream.range(1L, 6L)
                   .forEach(pub::submit);
      }

      if (subTask != null) {
         try {
            // Ждать завершения подписчика
            subTask.get();
         } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
         }
      }
```

```
    }
}
```

```
Размер буфера подписчика: 256
1
2
3
4
5
```

Метод `main()` объявляет переменную `subTask`, в которой будет храниться ссылка на задачу подписчика. Метод `subTask.get()` блокирует выполнение, пока подписчик не завершит работу.

```
CompletableFuture<Void> subTask = null;
```

Издатель, публикующий элементы типа `Long`, создается в блоке `try` с ресурсами. Издатель является экземпляром класса `SubmissionPublisher<Long>`. Он автоматически закрывается при выходе из блока `try`.

```
try (SubmissionPublisher<Long> pub = new SubmissionPublisher<>()) {
    //...
}
```

Программа печатает размер буфера каждого подписчика.

```
System.out.println("Размер буфера подписчика: " + pub.getMaxBufferCapacity());
```

Для добавления подписчика вызывается метод `consume()`. Отметим, что созданный им потребитель под капотом преобразуется в объект `Subscriber`. Подписчик получает уведомление о каждом опубликованном элементе и просто печатает его.

```
subTask = pub.consume(System.out::println);
```

Теперь можно публиковать целые числа. Программа генерирует пять чисел, от 1 до 5, и публикует их методом `submit()`.

```
LongStream.range(1L, 6L)
          .forEach(pub::submit);
```

Уведомления об опубликованных числах асинхронно отправляются подписчику. По выходе из блока `try` издатель закрывается. Чтобы программа продолжала работать, до тех пор пока подписчик не закончит обработку всех опубликованных данных, необходимо вызвать метод `subTask.get()`. Если этого не сделать, то могут быть напечатаны не все пять чисел.

Создание подписчиков

Для создания подписчика необходим класс, реализующий интерфейс `Flow.Subscriber<T>`. Как реализовать методы этого интерфейса, зависит от приложения. В этом разделе мы создадим класс `SimpleSubscriber`, реализующий интерфейс `Flow.Subscriber<Long>`. В листинге 17.4 приведен код этого класса.

Листинг 17.4. Класс SimpleSubscriber, реализующий интерфейс Flow.Subscriber<Long>

```
// SimpleSubscriber.java
package com.jdojo.stream;

import java.util.concurrent.Flow;

public class SimpleSubscriber implements Flow.Subscriber<Long> {
   private Flow.Subscription subscription;

   // Имя подписчика
   private String name = "Unknown";

   // Максимальное число элементов, обрабатываемых подписчиком
   private final long maxCount;

   // Счетчик уже обработанных элементов
   private long counter;

   public SimpleSubscriber(String name, long maxCount) {
      this.name = name;
      this.maxCount = maxCount <= 0 ? 1 : maxCount;
   }

   public String getName() {
      return name;
   }

   @Override
      public void onSubscribe(Flow.Subscription subscription) {
      this.subscription = subscription;
      System.out.printf("%s подписался, макс число элементов %d.%n", name, maxCount);

      // Обработать все элементы за один присест
      subscription.request(maxCount);
   }

   @Override
   public void onNext(Long item) {
      counter++;
      System.out.printf("%s получил %d.%n", name, item);
      if (counter >= maxCount) {
         System.out.printf("%s отписывается. Обработано элементов: %d.%n", name, counter);

         // Отказаться от подписки
         subscription.cancel();
```

```
        }
    }

    @Override
    public void onError(Throwable t) {
        System.out.printf("Произошла ошибка в подписчике %s: %s.%n", name, t.getMessage());
    }

    @Override
    public void onComplete() {
        System.out.printf("%s закончил работу.%n", name);
    }
}
```

Экземпляр класса `SimpleSubscriber` представляет подписчика, для которого определены имя и максимальное число обрабатываемых элементов. Аргументы `name` и `maxCount` передаются конструктору класса. Если `maxCount` меньше 1, то конструктор присваивает значение 1.

В методе `onSubscribe()` подписка, полученная от издателя, сохраняется в переменной экземпляра `subscription`. Метод печатает сообщение о подписке и разом обрабатывает все элементы. В этом подписчике применена модель проталкивания, потому что после первоначального запроса он больше ничего не запрашивает у издателя. Издатель отправит этому подписчику не более `maxCount` элементов данных.

В методе `onNext()` переменная экземпляра `counter` увеличивается на 1. В этой переменной хранится число полученных подписчиком элементов. Метод печатает сообщения со сведениями о каждом элементе. Получив последний из запрошенных элементов, подписчик отказывается от подписки. После этого он не получит от издателя новых элементов.

В методах `onError()` и `onComplete()` печатаются соответствующие сообщения.

В следующем фрагменте создается объект `SimpleSubscriber` с именем `S1`, готовый обработать не более 10 элементов.

```
SimpleSubscriber sub1 = new SimpleSubscriber("S1", 10);
```

Теперь посмотрим на `SimpleSubscriber` в действии. В листинге 17.5 приведен полный код программы. Она публикует элементы и после публикации каждого элемента ждет от 1 до 3 секунд. Продолжительность ожидания выбирается случайным образом. Поскольку в программе используется асинхронная обработка, на вашей машине может получиться несколько иной результат.

Листинг 17.5. Издатель периодически публикует данные, на которые подписано несколько объектов SimpleSubscriber

```
// PeriodicPublisher.java
package com.jdojo.stream;

import java.util.Random;
```

```
import java.util.concurrent.Flow.Subscriber;
import java.util.concurrent.SubmissionPublisher;
import java.util.concurrent.TimeUnit;

public class PeriodicPublisher {
   final static int MAX_SLEEP_DURATION = 3;

   // Для генерации случайного времени ожидания
   final static Random sleepTimeGenerator = new Random();

   public static void main(String[] args) {
      SubmissionPublisher<Long> pub = new SubmissionPublisher<>();

      // Создать четырех подписчиков
      SimpleSubscriber sub1 = new SimpleSubscriber("S1", 2);
      SimpleSubscriber sub2 = new SimpleSubscriber("S2", 5);
      SimpleSubscriber sub3 = new SimpleSubscriber("S3", 6);
      SimpleSubscriber sub4 = new SimpleSubscriber("S4", 10);

      // Подписаться на издателя
      pub.subscribe(sub1);
      pub.subscribe(sub2);

      pub.subscribe(sub3);

      // Четвертый подписчик подписывается спустя 2 секунды
      subscribe(pub, sub4, 2);

      // Начать публикацию данных
      Thread pubThread = publish(pub, 5);
      try {
         // Ждать завершения подписчика
         pubThread.join();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }

   public static Thread publish(SubmissionPublisher<Long> pub, long count) {
      Thread t = new Thread(() -> {
         for (long i = 1; i <= count; i++) {
            pub.submit(i);
            sleep(i);
         }

         // Закрыть подписчика
         pub.close();
```

```
        });

        // Запустить поток
        t.start();

        return t;
    }

    private static void sleep(Long item) {
        // Ждать от 1 до 3 секунд
        int sleepTime = sleepTimeGenerator.nextInt(MAX_SLEEP_DURATION) + 1;
        try {
            System.out.printf("Опубликовал %d. Жду %d с.%n", item, sleepTime);
            TimeUnit.SECONDS.sleep(sleepTime);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private static void subscribe(SubmissionPublisher<Long> pub, Subscriber<Long> sub,
                                  long delaySeconds) {
        new Thread(() -> {
            try {
                TimeUnit.SECONDS.sleep(delaySeconds);
                pub.subscribe(sub);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

```
S2 подписался, макс число элементов 5.
Опубликовал 1. Жду 1 с.
S3 подписался, макс число элементов 6.
S1 подписался, макс число элементов 2.
S1 получил 1.
S3 получил 1.
S2 received 1.
Опубликовал 2. Жду 1 с.
S1 получил 2.
S2 получил 2.
S3 получил 2.
S1 отписывается. Обработано элементов: 2.
S4 подписался, макс число элементов 10.
Опубликовал 3. Жду 1 с.
```

```
S4 получил 3.
S3 получил 3.
S2 получил 3.
Опубликовал 4. Жду 2 с.
S4 получил 4.
S3 получил 4.
S2 получил 4.
Опубликовал 5. Жду 2 с.
S2 получил 5.
S2 отписывается. Обработано элементов:  5.
S4 получил 5.
S3 получил 5.
S3 завершил работу.
S4 завершил работу.
```

В классе `PeriodicPublisher` есть две статические переменные. В переменной `MAX_SLEEP_DURATION` хранится максимальное число секунд ожидания после публикации очередного элемента (3). В переменной `sleepTimeGenerator` хранится ссылка на объект `Random`, который используется в методе `sleep()` для генерации следующего интервала ожидания.

В методе `main()` класса `PeriodicPublisher` выполняются следующие действия:

- создается издатель – экземпляр класса `SubmissionPublisher<Long>`;
- создается четыре подписчика с именами `S1`, `S2`, `S3`, `S4`. Каждый подписчик готов обрабатывать разное число элементов;
- три подписчика подписываются сразу;
- подписчик `S4` подписывается в отдельном потоке спустя 2 секунды. Это делается в методе `subscribe()`. В выходной распечатке видно, что `S4` подписывается после того, как два элемента (1 и 2) уже опубликованы, поэтому он их не получит;
- вызывается метод `publish()`, который запускает новый поток для публикации пяти элементов и возвращает ссылку на этот поток;
- вызывается метод `join()` потока, публикующего элементы, чтобы программа не завершилась раньше, чем будут опубликованы все элементы.

Метод `publish()` публикует все пять элементов и в конце закрывает издателя. После публикации каждого элемента вызывается метод `sleep()`, приостанавливающий текущий поток на случайное время от 1 до `MAX_SLEEP_DURATION` секунд.

Обратите внимание, что несколько подписчиков отказались от подписки, потому что уже получили заказанное число элементов от издателя.

Эта программа гарантирует, что все данные будут опубликованы до завершения, но не гарантирует, что все подписчики эти данные получат. Из распечатки видно, что подписчики получили все опубликованные данные. Это произошло, потому что издатель ждет не менее одной секунды после публикации последнего элемента, а в такой маленькой программе этого достаточно, чтобы все подписчики успели получить и обработать элемент.

В программе не демонстрируется противодавление, потому что все подписчики применяют модель проталкивания, запрашивая сразу все данные. Можете в качестве домашнего задания модифицировать класс `SimpleSubscriber`, чтобы был виден эффект противодавления:

- Запросите в методе `onSubscribe()` один элемент методом `subscription.request(1)`.
- В методе `onNext()` запросите дополнительные элементы после задержки. Смысл задержки в том, чтобы подписчик работал медленнее, чем издатель публикует элементы.
- Нужно будет либо опубликовать более 256 элементов – размера буфера подписчика по умолчанию, либо уменьшить этот размер, воспользовавшись другим конструктором класса `SubmissionPublisher`. Это вынудит издателя опубликовать больше элементов, чем могут обработать подписчики.
- Подпишите подписчиков, задав обработчик отбрасывания, чтобы было видно, когда издатель встречает противодавление.
- Для публикации элементов используйте метод `offer()` класса `SubmissionPublisher`, чтобы издатель не ждал бесконечно подписчиков, отказывающихся обрабатывать последующие элементы.

Использование процессоров

Процессор одновременно является подписчиком и издателем. Чтобы воспользоваться процессором, нужен класс, реализующий интерфейс `Flow.Processor<T,R>`, где `T` – тип элемента, на который оформлена подписка, а `R` – тип публикуемого элемента. В этом разделе я создам простой процессор, который фильтрует элементы с помощью предиката `Predicate<T>`. Процессор подписывается на издателя, публикующего шесть целых чисел – 1, 2, 3, 4, 5, 6. Подписчик подписывается на процессор. Процессор получает данные от издателя и повторно публикует те элементы, которые удовлетворяют критерию, заданному предикатом. В листинге 17.6 приведен код класса `FilterProcessor<T>`, экземпляры которого играют роль процессоров.

Листинг 17.6. Процессор, который отфильтровывает элементы по предикату и повторно публикует оставшиеся

```
// FilterProcessor.java
package com.jdojo.stream;

import java.util.concurrent.Flow;
import java.util.concurrent.Flow.Processor;
import java.util.concurrent.SubmissionPublisher;
import java.util.function.Predicate;

public class FilterProcessor<T> extends SubmissionPublisher<T> implements Processor<T,T>{
   private Predicate<? super T> filter;

   public FilterProcessor(Predicate<? super T> filter) {
```

```
        this.filter = filter;
    }

    @Override
    public void onSubscribe(Flow.Subscription subscription) {
        // Запросить неограниченное число элементов
        subscription.request(Long.MAX_VALUE);
    }

    @Override
    public void onNext(T item) {
        // Если элемент прошел фильтр, опубликовать его, иначе ничего не делать.
        System.out.println("Фильтр получил: " + item);
        if (filter.test(item)) {
            this.submit(item);
        }
    }

    @Override
    public void onError(Throwable t) {
        // Асинхронно передать сообщение onError всем подписчикам
        this.getExecutor().execute(() -> this.getSubscribers()
            .forEach(s -> s.onError(t)));
    }

    @Override
    public void onComplete() {
        System.out.println("Фильтр завершил работу.");

        // Закрыть издателя, чтобы все подписчики получили сообщение onComplete
        this.close();
    }
}
```

Класс `FilterProcessor<T>` наследует классу `SubmissionPublisher<T>` и реализует интерфейс `Flow.Processor<T,T>`. Процессор должен выполнять функции как издателя, так и подписчика. Я сделал его наследником `SubmissionPublisher<T>`, чтобы не пришлось писать самому код издателя. Класс реализует все методы интерфейса `Processor<T,T>`, т. е. будет получать и публиковать данные одного и того же типа.

Конструктор принимает объект `Predicate<? super T>` и сохраняет его в переменной экземпляра `filter`, которая будет использоваться в методе `onNext()` для фильтрации элементов.

Метод `onNext()` применяет фильтр. Если фильтр возвращает `true`, то элемент публикуется подписчикам процессора методом `submit()`, унаследованным от суперкласса `SubmissionPublisher`.

Метод `onError()` асинхронно транслирует ошибку подписчикам. Он пользуется методами `getExecutor()` и `getSubscribers()` класса `SubmissionPublisher`, которые возвра-

щают соответственно исполнителя (объект `Executor`) и список подписчиков. Исполнитель асинхронно публикует сообщения всем имеющимся подписчикам.

Метод `onComplete()` закрывает издательскую часть процессора, в результате всем подписчикам будет послано сообщение `onComplete`.

Теперь посмотрим на этот процессор в действии. В листинге 17.7 приведен код класса `ProcessorTest`, а объяснение его работы находится после распечатки результатов.

Листинг 17.7. Использование процессора в конвейере издатель-подписчик

```
// ProcessorTest.java
package com.jdojo.stream;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.SubmissionPublisher;
import java.util.concurrent.TimeUnit;
import java.util.stream.LongStream;

public class ProcessorTest {
   public static void main(String[] args) {
      CompletableFuture<Void> subTask = null;

      // Издатель закрывается по выходе из блока try
      try (SubmissionPublisher<Long> pub = new SubmissionPublisher<>()) {
         // Создать подписчика
         SimpleSubscriber sub = new SimpleSubscriber("S1", 10);

         // Создать процессор
         FilterProcessor<Long> filter = new FilterProcessor<>(n -> n % 2 == 0);

         // Подписать фильтр на издателя, а подписчика на фильтр
         pub.subscribe(filter);
         filter.subscribe(sub);

         // Сгенерировать и опубликовать 6 целых чисел
         LongStream.range(1L, 7L)
                   .forEach(pub::submit);
      }

      try {
         // Подождать 2 секунды, чтобы подписчики успели обработать все данные
         TimeUnit.SECONDS.sleep(2);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}
```

```
S1 подписался, макс число элементов 10.
Фильтр получил: 1
Фильтр получил: 2
Фильтр получил: 3
S1 получил 2.
Фильтр получил: 4
S1 получил 4.
Фильтр получил: 5
Фильтр получил: 6
Фильтр завершил работу.
S1 получил 6.
S1 завершил работу.
```

Метод `main()` класса `ProcessorTest` производит следующие действия:

- создает издателя, публикующего шесть целых чисел от 1 до 6 в блоке `try` с ресурсами, так чтобы он автоматически закрывался при выходе из блока;
- создает подписчика `S1` – экземпляр класса `SimpleSubscriber`, – который готов обработать не более 10 элементов данных;
- создает процессор – экземпляр класса `FilterProcessor<Long>`, – передавая ему предикат `Predicate<Long>`, который позволяет ретранслировать четные числа и отбрасывать нечетные;
- процессор подписывается на издателя, а подписчик – на процессор. Тем самым создается конвейер издатель–фильтр–подписчик;
- в конце первого блока `try` программа генерирует целые числа от 1 до 6 и публикует их с помощью издателя;
- в конце метода `main()` программа ждет 2 секунды, чтобы дать фильтру и подписчику возможность обработать события. Если убрать этот кусок, то программа может выйти, вообще ничего не напечатав. Необходимость в нем вызвана асинхронной природой обработки событий. Издатель заканчивает отправку всех уведомлений фильтру в момент выхода из первого блока `try`. Однако фильтру и подписчику нужно некоторое время для получения и обработки этих уведомлений.

Резюме

Потоком называется последовательность элементов, порождаемых производителем и потребляемых одним или несколькими потребителями. У модели производитель–потребитель есть и другие названия: источник–сток или издатель–подписчик.

Существует несколько механизмов обработки потоков, наиболее распространенными являются модель вытягивания (pull) и модель проталкивания (push). В случае проталкивания издатель отправляет данные подписчику, а в случае вытягивания подписчик сам забирает данные у издателя. Эти модели испытывают проблемы, когда стороны работают в разном темпе. Решение состоит в том, чтобы

поток адаптировался к скорости работы издателя и подписчика. Для этой цели применяется стратегия *противодавления*, когда подписчик уведомляет издателя о том, сколько элементов данных он готов обработать, а издатель отправляет именно столько элементов.

В 2013 году была выдвинута инициатива Reactive Streams, ставившая целью разработку стандарта *асинхронной* обработки потоков с *неблокирующим противодавлением*. Ее предметом было решение задач обработки потоков данных: как передать поток от издателя подписчику, избавив издателя от блокировки, а подписчика от неограниченного роста буфера или необходимости отбрасывать данные. В реактивных потоках происходит динамическое переключение между моделями проталкивания и вытягивания. Модель вытягивания применяется, когда быстрее работает подписчик, а модель проталкивания – когда быстрее издатель.

В 2015 году была опубликована спецификация и Java API для работы с реактивными потоками. API реактивных потоков в Java состоит из четырех интерфейсов: `Publisher<T>`, `Subscriber<T>`, `Subscription` и `Processor<T,R>`.

Издатель публикует элементы данных своим текущим подписчикам в соответствии с их требованиями. *Подписчик* подписывается на получение данных от издателя. Издатель отправляет подписчику маркер подписки. Имея этот маркер, подписчик запрашивает `N` элементов данных у издателя. Когда данные будут готовы, издатель отправляет подписчику `N` или меньше элементов. Подписчик может запросить дополнительные элементы.

JDK 9 предлагает API, совместимый со стандартом Reactive Streams, в пакете `java.util.concurrent`, находящемся в модуле `java.base`. API включает два класса: `Flow` и `SubmissionPublisher<T>`.

Класс `Flow` инкапсулирует API реактивных потоков и статический метод. Все четыре интерфейса, определенные в спецификации API, включены в класс `Flow` в виде вложенных статических интерфейсов: `Flow.Processor<T,R>`, `Flow.Publisher<T>`, `Flow.Subscriber<T>` и `Flow.Subscription`.

Глава 18

Изменения API потоков

Краткое содержание главы:

- новые вспомогательные методы, добавленные в интерфейс `Stream`;
- новые коллекторы, добавленные в класс `Collectors`.

В JDK 9 в API потоков пополнился несколькими вспомогательными методами. Я буду классифицировать их по тому типу, в который они добавлены:

- интерфейс `Stream`;
- класс `Collectors`.

Методы интерфейса `Stream` определяют новые потоковые операции, а методы класса `Collectors` – новые коллекторы. В последующих разделах те и другие описаны подробно. Исходный код, приведенный в этой главе, находится в модуле `com.jdojo.streams`, объявление которого показано в листинге 18.1.

Листинг 18.1. Объявление модуля com.jdojo.streams

```
// module-info.java
module com.jdojo.streams {
    exports com.jdojo.streams;
}
```

Новые потоковые операции

В JDK 9 в интерфейс `Stream` добавлены следующие методы:

- `default Stream<T> dropWhile(Predicate<? super T> predicate)`
- `default Stream<T> takeWhile(Predicate<? super T> predicate)`
- `static <T> Stream<T> ofNullable(T t)`
- `static <T> Stream<T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)`

В JDK 8 в интерфейсе `Stream` было два метода: `skip(long count)` и `limit(long count)`. Метод `skip()` пропускает заданное число начальных элементов и возвращает

остальные элементы потока. Метод `limit()` возвращает указанное или меньшее число начальных элементов. Один метод отбрасывает начальные элементы, второй оставляет только начальные элементы. Тому и другому передается число элементов. Методы `dropWhile()` и `takeWhile()` аналогичны методам `skip()` и `limit()` соответственно, только получают они не число элементов, а предикат.

Эти методы похожи на метод `filter()` с одним исключением. Метод `filter()` вычисляет предикат для всех элементов потока, а методы `dropWhile()` и `takeWhile()` – только для начальных, пока предикат не вернет `false`.

Для упорядоченного потока метод `dropWhile()` возвращает все элементы, кроме тех начальных, для которых предикат равен `true`. Рассмотрим такой упорядоченный поток целых чисел:

```
1, 2, 3, 4, 5, 6, 7
```

Если передать методу `dropWhile()` предикат, который возвращает `true` для чисел, меньших 5, то будут отброшены первые четыре элемента:

```
5, 6, 7
```

Для неупорядоченного потока поведение метода `dropWhile()` не детерминировано. Может быть отброшено любое подмножество элементов, удовлетворяющих предикату. В текущей реализации отбрасываются начальные элементы, пока не будет найден первый элемент, не удовлетворяющий предикату.

У метода `dropWhile()` существуют два крайних случая. Если первый элемент не удовлетворяет предикату, то возвращается исходный поток. Если все элементы удовлетворяют предикату, то возвращается пустой поток.

Метод `takeWhile()` работает аналогично `dropWhile()`, только начальные элементы возвращаются, а остальные отбрасываются.

> **Совет.** Применять методы `dropWhile()` и `takeWhile()` к упорядоченным параллельным потокам следует с осторожностью, т. к. может пострадать производительность. Прежде чем они смогут вернуть управление, придется получить и упорядочить элементы из всех потоков выполнения[1]. Лучше применять эти методы к последовательным потокам.

Метод `ofNullable(T t)` возвращает поток из одного указанного элемента, если этот элемент не равен `null`. Если же передан `null`, то возвращается пустой поток. Этот метод полезен при использовании метода `flatMap()`. Рассмотрим следующее отображение, в котором встречаются значения `null`:

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "One");
map.put(2, "Two");
map.put(3, null);
map.put(4, "four");
```

Как получить множество всех значений, отличных от `null`, т. е. содержащее элементы `"One"`, `"Two"` и `"Four"`? Вот как мы сделали бы это в JDK 8:

---

[1] Слово «поток» неоднозначно: оно обозначает поток данных (stream), поток выполнения (thread) и поток управления (flow). Хочется надеяться, что из контекста ясно, о чем идет речь. – *Прим. перев.*

```
// В JDK 8
Set<String> nonNullvalues = map.entrySet()
          .stream()
          .flatMap(e -> e.getValue() == null ? Stream.empty() : Stream.of(e.getValue()))
          .collect(toSet());
```

Обратите внимание на тернарный оператор внутри лямбда-выражения в методе `flatMap()`. В JDK 9 это выражение можно упростить, воспользовавшись методом `ofNullable()`:

```
// В JDK 9
Set<String> nonNullvalues = map.entrySet()
          .stream()
          .flatMap(e -> Stream.ofNullable(e.getValue()))
          .collect(toSet());
```

Новый метод `iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)` позволяет создать последовательный (возможно, бесконечный) поток, итеративно применяя к начальному значению `seed` заданную функцию `next`. Итерирование прекращается, когда предикат `hasNext` вернет `false`. Вызов этого метода эквивалентен циклу `for`:

```
for (T n = seed; hasNext.test(n); n = next.apply(n)) {
    // элемент n добавляется в поток
}
```

Следующий код порождает поток, содержащий целые числа от 1 до 10:

```
Stream.iterate(1, n -> n <= 10, n -> n + 1)
```

В листинге 18.2 приведен полный код программы, демонстрирующей использование новых методов интерфейса `Stream`.

Листинг 18.2. Класс StreamTest, демонстрирующий использование новых методов интерфейса Stream

```
// StreamTest.java
package com.jdojo.streams;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;
import static java.util.stream.Collectors.toList;
import static java.util.stream.Collectors.toSet;
import java.util.stream.Stream;

public class StreamTest {
   public static void main(String[] args) {
      System.out.println("Использование Stream.dropWhile() и Stream.takeWhile():");
```

```
        testDropWhileAndTakeWhile();

        System.out.println("\nИспользование Stream.ofNullable():");
        testOfNullable();

        System.out.println("\nИспользование Stream.iterator():");
        testIterator();
    }

    public static void testDropWhileAndTakeWhile() {
        List<Integer> list = List.of(1, 3, 5, 4, 6, 7, 8, 9);
        System.out.println("Исходный поток: " + list);
        List<Integer> list2 = list.stream()
                                  .dropWhile(n -> n % 2 == 1)
                                  .collect(toList());
        System.out.println("После вызова dropWhile(n -> n % 2 == 1): " + list2);
        List<Integer> list3 = list.stream()
                                  .takeWhile(n -> n % 2 == 1)
                                  .collect(toList());
        System.out.println("После вызова takeWhile(n -> n % 2 == 1): " + list3);
    }

    public static void testOfNullable() {
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "One");
        map.put(2, "Two");
        map.put(3, null);
        map.put(4, "Four");

        Set<String> nonNullValues = map.entrySet()
                                       .stream()
                                       .flatMap(e -> Stream.ofNullable(e.getValue()))
                                       .collect(toSet());
        System.out.println("Отображение: " + map);
        System.out.println("Значения, отличные от null: " + nonNullValues);
    }

    public static void testIterator() {
        List<Integer> list = Stream.iterate(1, n -> n <= 10, n -> n + 1)
                                   .collect(toList());
        System.out.println("Целые числа от 1 до 10: " + list);
    }
}
```

```
Использование Stream.dropWhile() и Stream.takeWhile():
Исходный поток: [1, 3, 5, 4, 6, 7, 8, 9]
```

```
После вызова dropWhile(n -> n % 2 == 1): [4, 6, 7, 8, 9]
После вызова takeWhile(n -> n % 2 == 1): [1, 3, 5]
Использование Stream.ofNullable():
Отображение: {1=One, 2=Two, 3=null, 4=Four}
Значения, отличные от null: [One, Four, Two]
Использование Stream.iterator():
Целые числа от 1 до 10: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Новые коллекторы

В класс `Collectors` добавлено два метода, возвращающих значение типа `Collector`:

- `<T,A,R> Collector<T,?,R> filtering(Predicate<? super T> predicate, Collector<? super T,A,R> downstream)`
- `<T,U,A,R> Collector<T,?,R> flatMapping(Function<? super T,? extends Stream<? extends U>> mapper, Collector<? super U,A,R> downstream)`

Метод `filtering()` возвращает объект `Collector`, который применяет фильтр перед сбором элементов. Если переданный предикат возвращает `true`, то элемент помещается в коллекцию, иначе пропускается.

Метод `flatMapping()` возвращает объект `Collector`, который применяет функцию разглаживания перед сбором элемента. Заданная функция применяется к каждому элементу потока, и возвращенные ей элементы аккумулируются.

Оба метода возвращают коллектор, который полезнее всего в многоуровневой редукции, например, для обработки результатов методов `groupingBy` или `partitioningBy`. В листинге 18.3 демонстрируется их применение.

Листинг 18.3. Класс Employee

```
// Employee.java
package com.jdojo.streams;

import java.util.List;

public class Employee {
   private String name;
   private String department;
   private double salary;
   private List<String> spokenLanguages;

   public Employee(String name, String department, double salary,
                   List<String> spokenLanguages) {
      this.name = name;
      this.department = department;
      this.salary = salary;
      this.spokenLanguages = spokenLanguages;
```

```
}

public String getName() {
   return name;
}

public void setName(String name) {
   this.name = name;
}

public String getDepartment() {
   return department;
}

public void setDepartment(String department) {
   this.department = department;
}

public double getSalary() {
   return salary;
}

public void setSalary(double salary) {
   this.salary = salary;
}

public List<String> getSpokenLanguages() {
   return spokenLanguages;
}

public void setSpokenLanguages(List<String> spokenLanguages) {
   this.spokenLanguages = spokenLanguages;
}

@Override
public String toString() {
   return "[" + name + ", " + department + ", " + salary + ", " + spokenLanguages +
          "]";
}

public static List<Employee> employees() {
   return List.of(
      new Employee("John", "Sales", 1000.89, List.of("English", "French")),
      new Employee("Wally", "Sales", 900.89, List.of("Spanish", "Wu")),
      new Employee("Ken", "Sales", 1900.00, List.of("English", "French")),
      new Employee("Li", "HR", 1950.89, List.of("Wu", "Lao")),
      new Employee("Manuel", "IT", 2001.99, List.of("English", "German")),
```

```
            new Employee("Tony", "IT", 1700.89, List.of("English"))
        );
    }
}
```

Для объекта `Employee` (работник) определены имя, отдел, зарплата и список языков, которыми он владеет. Метод `toString()` возвращает строку, содержащую все эти свойства. Статический метод `employees()` возвращает список работников, показанных в табл. 18.1.

Таблица 18.1. Работники, используемые в примере

Имя	Отдел	Зарплата	Иностранные языки
John	Sales	1000.89	English, French
Wally	Sales	900.89	Spanish, Wu
Ken	Sales	1900.00	English, French
Li	HR	1950.89	Wu, Lao
Manuel	IT	2001.89	English, German
Tony	IT	1700.89	English

Вот как можно сгруппировать работников по отделам:

```
Map<String,List<Employee>> empGroupedByDept = Employee.employees()
        .stream()
        .collect(groupingBy(Employee::getDepartment, toList()));
System.out.println(empGroupedByDept);
```

```
{Sales=[[John, Sales, 1000.89, [English, French]], [Wally, Sales, 900.89, [Spanish, Wu]],
[Ken, Sales, 1900.0, [English, French]]], HR=[[Li, HR, 1950.89, [Wu, Lao]]], IT=[[Manuel,
IT, 2001.99, [English, German]], [Tony, IT, 1700.89, [English]]]}
```

Такая возможность существовала в API потоков еще в JDK 8. Но пусть мы хотим получить сгруппированный по отделам список работников с зарплатой больше 1900. Первая попытка применить фильтр могла бы выглядеть так:

```
Map<String, List<Employee>> empSalaryGt1900GroupedByDept = Employee.employees()
        .stream()
        .filter(e -> e.getSalary() > 1900)
        .collect(groupingBy(Employee::getDepartment, toList()));
System.out.println(empSalaryGt1900GroupedByDept);
```

```
{HR=[[Li, HR, 1950.89, [Wu, Lao]]], IT=[[Manuel, IT, 2001.99, [English, German]]]}
```

В каком-то смысле цель достигнута. Однако в список не вошли отделы, в которых нет ни одного работника, получающего больше 1900. Так произошло, потому что мы отфильтровали все такие отделы еще до того, как начали собирать результаты. Желаемый результат можно получить, воспользовавшись коллектором, который возвращает метод `filtering()`. Теперь, даже если в отделе нет работников, получающих больше 1900, отдел все равно будет включен, но с пустым списком работников.

```
Map<String, List<Employee>> empGroupedByDeptWithSalaryGt1900 = Employee.employees()
        .stream()
        .collect(groupingBy(Employee::getDepartment,
                            filtering(e -> e.getSalary() > 1900.00, toList())));
System.out.println(empGroupedByDeptWithSalaryGt1900);
```

```
{Sales=[], HR=[[Li, HR, 1950.89, [Wu, Lao]]], IT=[[Manuel, IT, 2001.99, [English,
German]]]}
```

Как видим, отдел `Sales` присутствует, хотя в нем нет работников с зарплатой больше 1900.

Попробуем получить множество языков, на которых говорят работники каждого отдела. В следующем фрагменте производится попытка воспользоваться коллектором, который возвращает метод `mapping()` класса `Collectors`:

```
Map<String,Set<List<String>>> langByDept = Employee.employees()
      .stream()
      .collect(groupingBy(Employee::getDepartment,
               mapping(Employee::getSpokenLanguages, toSet())));
System.out.println(langByDept);
```

```
{Sales=[[English, French], [Spanish, Wu]], HR=[[Wu, Lao]], IT=[[English, German],
[English]]}
```

Мы получили множество списков `Set<List<String>>`, а не просто `Set<String>`. Необходимо преобразовать список `List<String>` в поток строк, прежде чем собирать строки в множество. Задачу решает коллектор, возвращенный методом `flatMapping()`:

```
Map<String,Set<String>> langByDept2 = Employee.employees()
      .stream()
      .collect(groupingBy(Employee::getDepartment,
               flatMapping(e -> e.getSpokenLanguages().stream(), toSet())));
System.out.println(langByDept2);
```

```
{Sales=[English, French, Spanish, Wu], HR=[Lao, Wu], IT=[English, German]}
```

Вот теперь результат правильный. В листинге 18.4 приведен полный код программы, демонстрирующей использование фильтрации и разглаживания при собирании данных.

Листинг 18.4. Класс StreamCollectorsTest, демонстрирующий использование фильтрации и разглаживания

```
// StreamCollectorsTest.java
package com.jdojo. streams;

import java.util.List;
import java.util.Map;
import java.util.Set;
import static java.util.stream.Collectors.filtering;
import static java.util.stream.Collectors.flatMapping;
import static java.util.stream.Collectors.groupingBy;
import static java.util.stream.Collectors.mapping;
import static java.util.stream.Collectors.toList;
import static java.util.stream.Collectors.toSet;

public class StreamCollectorsTest {
   public static void main(String[] args) {
      System.out.println("Тестирование Collectors.filtering():");
      testFiltering();

      System.out.println("\nТестирование Collectors.flatMapping():");
      testFlatMapping();
   }

   public static void testFiltering() {
      Map<String, List<Employee>> empGroupedByDept = Employee.employees()
              .stream()
              .collect(groupingBy(Employee::getDepartment, toList()));

      System.out.println("Группировка работников по отделам:");
      System.out.println(empGroupedByDept);

      // Группировка по отделам работников с зарплатой > 1900
      Map<String, List<Employee>> empSalaryGt1900GroupedByDept = Employee.employees()
              .stream()
              .filter(e -> e.getSalary() > 1900)
              .collect(groupingBy(Employee::getDepartment, toList()));

      System.out.println("\nГруппировка работников с зарплатой > 1900 по отделам:");
      System.out.println(empSalaryGt1900GroupedByDept);

      // Группировка по отделам работников с зарплатой > 1900
      Map<String, List<Employee>> empGroupedByDeptWithSalaryGt1900 = Employee.employees()
```

```
                .stream()
                .collect(groupingBy(Employee::getDepartment,
                filtering(e -> e.getSalary() > 1900.00, toList())));

        System.out.println("\nГруппировка работников с зарплатой > 1900 по отделам:");
        System.out.println(empGroupedByDeptWithSalaryGt1900);

        // Группировка по отделам работников, владеющих по меньшей мере 2 языками,
        // один из которых английский
        Map<String, List<Employee>> empByDeptWith2LangWithEn = Employee.employees()
                .stream()
                .collect(groupingBy(Employee::getDepartment,
                        filtering(e -> e.getSpokenLanguages().size() >= 2
                        &&
                        e.getSpokenLanguages().contains("English"),
                        toList())));

        System.out.println("\nГруппировка по отделам работников, владеющих мин. 2 языками, " +
        " один из которых английский:");
        System.out.println(empByDeptWith2LangWithEn);
    }

    public static void testFlatMapping(){
        Map<String,Set<List<String>>> langByDept = Employee.employees()
                .stream()
                .collect(groupingBy(Employee::getDepartment,
                        mapping(Employee::getSpokenLanguages, toSet())));

        System.out.println("\nЯзыки, сгруппированные по отделам с помощью mapping():");
        System.out.println(langByDept);

        Map<String,Set<String>> langByDept2 = Employee.employees()
                .stream()
                .collect(groupingBy(Employee::getDepartment,
                        flatMapping(e -> e.getSpokenLanguages().stream(), toSet())));
        System.out.println("\nЯзыки, сгруппированные по отделам с помощью flapMapping():");
        System.out.println(langByDept2) ;
    }
}
```

```
Тестирование Collectors.filtering():
Группировка работников по отделам:
{Sales=[[John, Sales, 1000.89, [English, French]], [Wally, Sales, 900.89, [Spanish, Wu]],
[Ken, Sales, 1900.0, [English, French]]], HR=[[Li, HR, 1950.89, [Wu, Lao]]], IT=[[Manuel,
IT, 2001.99, [English, German]], [Tony, IT, 1700.89, [English]]]}

Группировка работников с зарплатой > 1900 по отделам:
```

```
{HR=[[Li, HR, 1950.89, [Wu, Lao]]], IT=[[Manuel, IT, 2001.99, [English, German]]]}

Группировка работников с зарплатой > 1900 по отделам:
{Sales=[], HR=[[Li, HR, 1950.89, [Wu, Lao]]], IT=[[Manuel, IT, 2001.99, [English,
German]]]}

Группировка по отделам работников, владеющих мин. 2 языками, один из которых английский:
{Sales=[[John, Sales, 1000.89, [English, French]], [Ken, Sales, 1900.0, [English, French]]],
HR=[], IT=[[Manuel, IT, 2001.99, [English, German]]]}

Тестирование Collectors.flatMapping():
Языки, сгруппированные по отделам с помощью mapping():
{Sales=[[English, French], [Spanish, Wu]], HR=[[Wu, Lao]], IT=[[English, German],
[English]]}

Языки, сгруппированные по отделам с помощью flapMapping():
{Sales=[English, French, Spanish, Wu], HR=[Lao, Wu], IT=[English, German]}
```

Резюме

В JDK 9 в API потоков добавлено несколько методов, упрощающих работу с потоками и написание сложных запросов с применением коллекторов.

В интерфейс `Stream` добавлено четыре метода: `dropWhile()`, `takeWhile()`, `ofNullable()`, `iterate()`. Для упорядоченного потока метод `dropWhile()` отбрасывает начальные элементы до тех пор, пока переданный предикат равен `true`, и возвращает оставшиеся. Для неупорядоченного потока поведение метода `dropWhile()` не детерминировано. Он может отбросить любое подмножество элементов, удовлетворяющих предикату. В текущей реализации отбрасываются начальные элементы, пока не будет найден первый элемент, не удовлетворяющий предикату. Метод `takeWhile()` работает аналогично `dropWhile()`, только начальные элементы возвращаются, а остальные отбрасываются. Метод `ofNullable(T t)` возвращает поток из одного указанного элемента, если этот элемент не равен `null`. Если же передан `null`, то возвращается пустой поток. Метод `iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)` позволяет создать последовательный (возможно, бесконечный) поток, итеративно применяя к начальному значению `seed` заданную функцию `next`. Итерирование прекращается, когда предикат `hasNext` вернет `false`.

В класс `Collectors` добавлено два метода: `filtering()` и `flatMapping()`. Метод `filtering()` возвращает объект `Collector`, который применяет фильтр перед сбором элементов. Если переданный предикат возвращает `true`, то элемент помещается в коллекцию, иначе пропускается. Метод `flatMapping()` возвращает объект `Collector`, который применяет функцию разглаживания перед сбором элемента. Заданная функция применяется к каждому элементу потока, и возвращенные ей элементы аккумулируются.

Глава 19

Протоколирование на уровне платформы и JVM

Краткое содержание главы:

- новый API платформенного протоколирования;
- параметр командной строки для протоколирования на уровне JVM.

В JDK 9 полностью переработаны системы протоколирования для платформенных классов (входящих в JDK) и компонент JVM. Новый API позволяет по своему выбору задать систему протоколирования, которая будет использоваться для вывода сообщений из платформенных классов. Введен также новый параметр командной строки, позволяющий получить доступ к сообщениям от всех компонентов JVM. В этой главе подробно описываются оба механизма.

API платформенного протоколирования

В Java SE 9 добавлен API платформенного протоколирования, позволяющий выбрать систему протоколирования, например Log4j, SLF4J или собственный механизм, который будет использоваться платформенными классами Java (входящими в состав JDK) для вывода сообщений. Тут необходимо сделать важную оговорку. Этот API предназначен для использования самим JDK, а не классами приложения. Поэтому использовать его для вывода сообщений приложением не следует. Для этой цели пользуйтесь своей любимой библиотекой, например Log4j. Новый API не дает возможности программно сконфигурировать диспетчер протоколирования. API включает следующие компоненты:

- интерфейс службы `java.lang.System.LoggerFinder` – абстрактный статический класс;
- интерфейс `java.lang.System.Logger`, определяющий API протоколирования;
- перегруженный метод `getLogger()` класса `java.lang.System`, возвращающий объект типа `System.Logger`.

Детали конфигурирования платформы зависят от конкретной системы протоколирования. Например, в случае Log4j библиотеку Log4j и платформенное

протоколирование нужно конфигурировать по отдельности. Для конфигурирования платформенного протоколирования нужно выполнить следующие действия:

- создать класс, реализующий интерфейс `System.Logger`;
- реализовать интерфейс службы `System.LoggerFinder`;
- указать реализацию в объявлении модуля.

В качестве платформенной системы протоколирования я сконфигурирую Log4j 2.0. Конфигурирование и использование Log4j – обширная тема сама по себе. Я коснусь только тех аспектов, которые необходимы для конфигурирования платформенного протоколирования.

> **Совет.** Если вы не сконфигурировали собственный механизм платформенного протоколирования, то JDK пользуется реализацией `System.LoggerFinder` по умолчанию, подразумевающей, что для протоколирования используется механизм `java.util.logging`, если присутствует модуль `java.logging`. Он возвращает объект, который направляет сообщения экземпляру `java.util.logging.Logger`. Если же модуль `java.logging` отсутствует, то реализация по умолчанию возвращает экземпляр простого диспетчера протоколирования, который выводит сообщения уровня `INFO` и выше на консоль `System.err`.

Подготовка библиотеки Log4j 2.0

Для примеров из этого раздела необходимо скачать библиотеку Log4j 2.0. Необходимые JAR-файлы, содержащие Log4j 2.0, имеются в каталоге `Java9Reaveled/extlib` прилагаемого к книге исходного кода. Можно также скачать их непосредственно со страницы `https://logging.apache.org/log4j/2.0/download.html`. Распакуйте скачанный файл и скопируйте следующие два JAR-файла в каталог `C:\Java8Revealed\extlib` (если будете копировать в другой каталог, не забудьте заменить путь, упоминаемый в тексте):

- `log4j-api-2.8.jar`
- `log4j-core-2.8.jar`

Возможно, в скачанных вами файлах версия будет другой. В нашем примере эти JAR-файлы будут использоваться как автоматические модули. Имена автоматических модулей образуются из имени JAR-файла, в данном случае – `log4j.api` и `log4j.core`. Подробно об автоматических модулях написано в главе 4.

Подготовка проекта NetBeans

Я создал проект `com.jdojo.logger` в NetBeans. На путь к модулям в этом проекте помещены оба JAR-файла Log4j, упомянутые в предыдущем разделе (рис. 19.1). Чтобы сделать это в NetBeans, выберите команду **Add JAR/Folder** (Добавить файл JAR/папку) из контекстного меню узла **Libraries** (Библиотеки).

Определение модуля

Все классы и ресурсы, относящиеся к данному примеру, находятся в модуле `com.jdojo.logger`, объявление которого приведено в листинге 19.1.

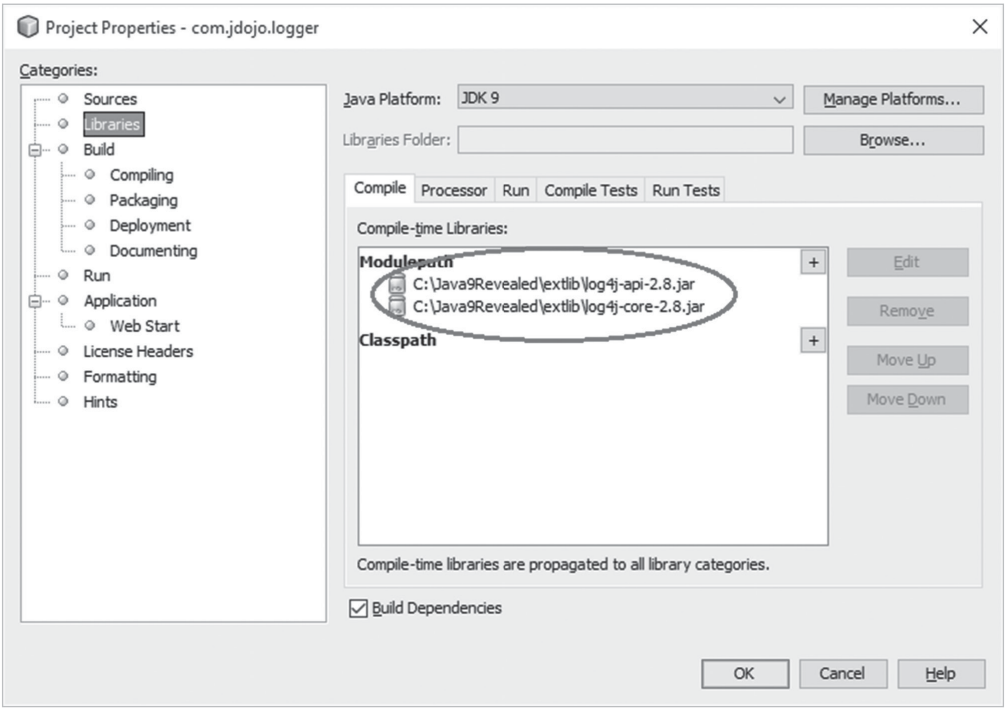

Рис. 19.1. Задание пути к модулям в проекте NetBeans com.jdojo.logger

Листинг 19.1. Объявление модуля com.jdojo.logger

```
// module-info.java
module com.jdojo.logger {
   requires log4j.api;
   requires log4j.core;

   exports com.jdojo.logger;

   provides java.lang.System.LoggerFinder with com.jdojo.logger.Log4jLoggerFinder;
}
```

В первых двух предложениях `requires` объявляется зависимость от JAR-файлов Log4j, которые в данном случае являются автоматическими модулями. В предложении `exports` экспортируются все типы из пакета `com.jdojo.logger` этого модуля. Предложение `provides` важно для настройки платформенного протоколирования. В нем утверждается, что мы предоставляем класс `com.jdojo.logger.Log4jLoggerFinder`, реализующий интерфейс службы `java.lang.System.LoggerFinder`. Этот класс мы скоро напишем. На рис. 19.2 изображен граф модулей для модуля `com.jdojo.logger`.

Обратите внимание на циклические зависимости и безымянный модуль в этом графе. Это связано с присутствием автоматических модулей в объявлении. Модуль `com.jdojo.logger` читает оба автоматических модуля. Каждый автоматический

модуль читает все остальные модули, о чем свидетельствуют стрелки, ведущие из `log4j.code` и `log4j.api` в другие модули. Хотя в этом примере безымянного модуля нет, в графе он присутствует. В данном случае безымянный модуль не содержит ни одного типа, а появился он потому, что автоматический модуль читает все вообще модули, в т. ч. безымянный.

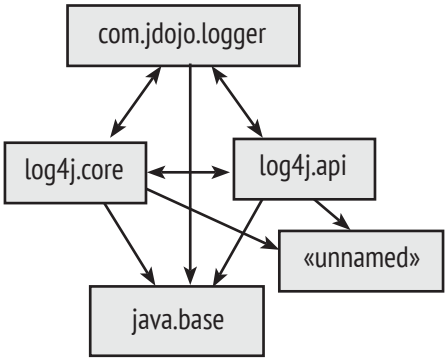

Рис. 19.2. Граф модулей для модуля com.jdojo.logger

Добавление конфигурационного файла Log4j

В листинге 19.2 приведен конфигурационный файл `log4j2.xml`, находящийся в корне исходного кода в проекте NetBeans. Иначе говоря, файл `log4j2.xml` помещен в безымянный пакет. В такой конфигурации Log4j будет помещать сообщения в файл `logs/platform.log` в текущем каталоге. Дополнительные сведения о конфигурировании Log4j см. в документации.

Листинг 19.2. Конфигурационный файл log4j2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="error">
   <Appenders>
      <File name="JdojoLogFile" fileName="logs/platform.log">
         <PatternLayout>
           <Pattern>%d %p %c [%t] %m%n</Pattern>
         </PatternLayout>
      </File>
      <Async name="Async">
         <AppenderRef ref="JdojoLogFile"/>
      </Async>
   </Appenders>
   <Loggers>
      <Root level="info">
      <AppenderRef ref="Async"/>
      </Root>
   </Loggers>
</Configuration>
```

Создание системного диспетчера протоколирования

Нам необходим системный диспетчер протоколирования – класс, реализующий интерфейс `System.Logger`. Этот интерфейс содержит следующие методы:

- `String getName()`
- `boolean isLoggable(System.Logger.Level level)`
- `default void log(System.Logger.Level level, Object obj)`
- `default void log(System.Logger.Level level, String msg)`
- `default void log(System.Logger.Level level, String format, Object... params)`
- `default void log(System.Logger.Level level, String msg, Throwable thrown)`
- `default void log(System.Logger.Level level, Supplier<String> msgSupplier)`
- `default void log(System.Logger.Level level, Supplier<String> msgSupplier, Throwable thrown)`
- `void log(System.Logger.Level level, ResourceBundle bundle, String format, Object... params)`
- `void log(System.Logger.Level level, ResourceBundle bundle, String msg, Throwable thrown)`

Мы должны предоставить реализации четырех абстрактных методов этого интерфейса. Метод `getName()` должен возвращать имя диспетчера, оно может быть произвольным. Метод `isLoggable()` возвращает `true`, если диспетчер может протоколировать сообщение указанного уровня. Два варианта метода `log()` служат для протоколирования сообщений и вызываются остальными методами `log()`, помеченными ключевым словом `default`.

В перечислении `System.Logger.Level` определены константы, задающие уровни сообщений. С уровнем связаны имя и серьезность. Ниже перечислены уровни в порядке возрастания серьезности: `ALL`, `TRACE`, `DEBUG`, `INFO`, `WARNING`, `ERROR`, `OFF`. Для получения имени и серьезности уровня предназначены методы `getName()` и `getSeverity()`.

В листинге 19.3 приведен код класса `Log4jLogger`, реализующего интерфейс `System.Logger`.

Листинг 19.3. Класс Log4jLogger, реализующий интерфейс System.Logger

```
// Log4jLogger.java
package com.jdojo.logger;

import java.util.ResourceBundle;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Log4jLogger implements System.Logger {
   // Настоящий диспетчер протоколирования - Log4j. Наш диспетчер
   // делегирует ему все запросы.
   private final Logger logger = LogManager.getLogger();

   @Override
   public String getName() {
```

```
        return "Log4jLogger";
    }

    @Override
    public boolean isLoggable(Level level) {
        // Получить уровень log4j из System.Logger.Level
        org.apache.logging.log4j.Level log4jLevel = toLog4jLevel(level);

        // Проверить, может ли log4j протоколировать сообщения этого уровня
        return logger.isEnabled(log4jLevel);
    }

    @Override
    public void log(Level level, ResourceBundle bundle, String msg, Throwable thrown) {
        logger.log(toLog4jLevel(level), msg, thrown);
    }

    @Override
    public void log(Level level, ResourceBundle bundle, String format, Object... params) {
        logger.printf(toLog4jLevel(level), format, params);
    }

    private static org.apache.logging.log4j.Level toLog4jLevel(Level level) {
        switch (level) {
            case ALL:
                return org.apache.logging.log4j.Level.ALL;
            case DEBUG:
                return org.apache.logging.log4j.Level.DEBUG;
            case ERROR:
                return org.apache.logging.log4j.Level.ERROR;
            case INFO:
                return org.apache.logging.log4j.Level.INFO;
            case OFF:
                return org.apache.logging.log4j.Level.OFF;
            case TRACE:
                return org.apache.logging.log4j.Level.TRACE;
            case WARNING:
                return org.apache.logging.log4j.Level.WARN;
            default:
                throw new RuntimeException("Unknown Level: " + level);
        }
    }
}
```

В качестве платформенного диспетчера протоколирования будет использоваться объект `Log4jLogger`. Он делегирует все запросы настоящему диспетчеру, в данном случае Log4j. В переменной экземпляра `logger` хранится ссылка на экземпляр класса `Logger` из библиотеки Log4j.

Мы имеем дело с двумя API протоколирования: `System.Logger` и Log4j. В них определены различные уровни, представленные соответственно типами `System.Logger.Level` и `org.apache.logging.log4j.Level`. Для протоколирования сообщения классы JDK передают значение типа `System.Logger.Level` одному из методов `log()` интерфейса `System.Logger`, который должен сопоставить ему уровень Log4j. Такое сопоставление осуществляется в методе `toLog4jLevel()`, который получает значение типа `System.Logger.Level` и возвращает соответствующее ему значение `org.apache.logging.log4j.Level`.

Метод `isLoggable()` отображает системный уровень на уровень Log4j и спрашивает у Log4j, протоколируются ли сообщения этого уровня. В конфигурационном файле Log4j (см. листинг 19.2) можно разрешить или запретить протоколирование на любом уровне.

Я не стал усложнять реализацию методов `log()`. Они просто делегируют запрос Log4j. В этой реализации не используется аргумент `ResourceBundle`. Он нужен, если мы хотим локализовать сообщение перед протоколированием.

Итак, логика платформенного протоколирования написана, но для тестирования она пока не готова. Чтобы увидеть ее в действии, нужно еще немного поработать.

Создание локатора диспетчера протоколирования

Среда выполнения Java должна как-то найти наш диспетчер. Для этого используется паттерн локатора службы. Вспомните следующее предложение в объявлении модуля (листинг 19.1):

```
provides java.lang.System.LoggerFinder with com.jdojo.logger.Log4jLoggerFinder;
```

В этом разделе мы создадим класс реализации `Log4jLoggerFinder`, реализующий интерфейс `System.LoggerFinder`. Напомним, что интерфейс службы может быть не только интерфейсом Java, но и абстрактным классом. В данном случае `System.LoggerFinder` – абстрактный класс, а наш класс `Log4jLoggerFinder` его расширяет. В листинге 19.4 приведен код класса `Log4jLoggerFinder`, реализующего интерфейс `System.LoggerFinder`.

Листинг 19.4. Класс Log4jLoggerFinder, реализующий интерфейс службы `System.LoggerFinder`

```
// Log4jLoggerFinder.java
package com.jdojo.logger;

import java.lang.System.LoggerFinder;

public class Log4jLoggerFinder extends LoggerFinder {
   // Диспетчер платформенного протоколирования
   private final Log4jLogger logger = new Log4jLogger();

   @Override
   public System.Logger getLogger(String name, Module module) {
```

```
        System.out.printf("Log4jLoggerFinder.getLogger(): " +
                          "[имя=%s, модуль=%s]%n", name, module.getName());

        // Использовать один и тот же диспетчер протоколирования для всех модулей
        return logger;
    }
}
```

`Log4jLoggerFinder` создает экземпляр класса `Log4jLogger` (см. листинг 19.3) и сохраняет ссылку на него в переменной экземпляра `logger`. Метод `getLogger()` возвращает эту самую переменную, когда JDK запрашивает диспетчер протоколирования. Аргументы `name` и `module` метода `getLogger()` – это имя запрашиваемого диспетчера и модуль запрашивающей стороны. Например, когда классу `java.util.Currency` необходимо вывести сообщение, он запрашивает диспетчер с именем `java.util.Currency`, указывая свой модуль `java.base`. Если вы хотите использовать отдельный диспетчер для каждого модуля, то можете сделать возвращаемое значение зависящим от аргумента `module`. В этом примере для всех модулей возвращается один и тот же диспетчер, так что все сообщения протоколируются в одно и то же место. Я оставил в методе `getLogger()` предложение `System.out.println()`, чтобы при выполнении примера были видны значения аргументов `name` и `module`.

Тестирование платформенного модуля

Пора уже посмотреть, как все это работает. В листинге 19.5 приведен код класса `PlatformLoggerTest`, используемого для тестирования платформенного протоколирования. На вашей машине результат может быть иным.

Листинг 19.5. Класс PlatformLoggerTest для тестирования платформенного протоколирования

```
// PlatformLoggerTest.java
package com.jdojo.logger;

import java.lang.System.Logger;
import static java.lang.System.Logger.Level.TRACE;
import static java.lang.System.Logger.Level.ERROR;
import static java.lang.System.Logger.Level.INFO;
import java.util.Currency;
import java.util.Set;

public class PlatformLoggerTest {
    public static void main(final String... args) {
        // Загрузить все валюты
        Set<Currency> c = Currency.getAvailableCurrencies();
        System.out.println("Загружено валют: " + c.size());

        // Протестировать платформенный диспетчер протоколирования
        // на нескольких сообщениях
```

```
        Logger logger = System.getLogger("Log4jLogger");
        logger.log(TRACE, "Вход в приложение.");
        logger.log(ERROR, "Произошла неизвестная ошибка.");
        logger.log(INFO, "Для сведения");
        logger.log(TRACE, "Выход из приложения.");
    }
}
```

```
Загружено валют: 225
Log4jLoggerFinder.getLogger(): [name=javax.management.mbeanserver, module=java.management]
Log4jLoggerFinder.getLogger(): [name=javax.management.misc, module=java.management]
Log4jLoggerFinder.getLogger(): [name=Log4jLogger, module=com.jdojo.logger]
```

Метод `main()` пытается получить список символов известных валют и печатает их количество. В чем смысл этого действия, я объясню ниже. Пока считайте, что это просто вызов метода класса `java.util.Currency`.

Хотя этот механизм и не предназначен для протоколирования сообщений приложения, при тестировании поступить так можно. Мы получаем ссылку на платформенный диспетчер протоколирования от метода `System.getLogger()` и начинаем выводить сообщения:

```
Logger logger = System.getLogger("Log4jLogger");
logger.log(TRACE, "Вход в приложение.");
logger.log(ERROR, "Произошла неизвестная ошибка.");
logger.log(INFO, "Для сведения");
logger.log(TRACE, "Выход из приложения.");
```

> **Совет.** В JDK 9 класс `System` содержит два статических метода для получения ссылки на платформенный диспетчер протоколирования: `getLogger(String name)` и `getLogger(String name, ResourceBundle bundle)`. Оба возвращают экземпляр интерфейса `System.Logger`.

Ни одного из четырех сообщений в распечатке нет. Куда же они делись? Напомним, что при конфигурировании Log4j мы указали, что сообщения нужно записывать в файл `logs/platform.log` в текущем каталоге. Что такое текущий каталог, зависит от того, как выполнялся класс `PlatformLoggerTest`: если из NetBeans, то это каталог проекта, т. е. `C:\Java9Revealed\com.jdojo.logger`, а если из командной строки, то текущий каталог определяете вы сами. В предположении, что класс выполнялся из NetBeans, протокол находится в файле `C:\Java9Revealed\com.jdojo.logger\logs\platform.log`. Его содержимое показано в листинге 19.6.

Листинг 19.6. Содержимое файла logs/platform.log

```
2017-02-09 09:58:34,644 ERROR com.jdojo.logger.Log4jLogger [main] Произошла неизвестная ошибка.
2017-02-09 09:58:34,646 INFO com.jdojo.logger.Log4jLogger [main] Для сведения
```

Примечание. При каждом выполнении класса `PlatformLoggerTest` Log4j дописывает сообщения в конец файла `logs\platform.log`. Далее приводятся только строки, относящиеся к одному прогону программы. Можете перед каждым запуском очищать или даже удалять файл журнала, он будет создан заново.

В файле журнала мы видим только два сообщения, уровня `ERROR` и `INFO`, а сообщения уровня `TRACE` отброшены. Именно так было настроено протоколирование в конфигурационном файле Log4j, показанном в листинге 19.2. Мы включили протоколирование уровня `INFO`:

```
<Loggers>
   <Root level="info">
      <AppenderRef ref="Async"/>
   </Root>
</Loggers>
```

С каждым уровнем протоколирования связана серьезность – целое число. Если включено протоколирование уровня `x`, то протоколируются все сообщения, для которых серьезность больше или равна `x`. В табл. 19.1 показаны имена всех уровней, определенные в перечислении `System.Logger.Level`, и соответствующая каждому уровню серьезность. Отметим, что в этом примере мы пользуемся уровнями, определенными в Log4j, а не в перечислении `System.Logger.Level`. Однако относительный порядок уровней Log4j такой же, как в таблице.

Таблица 19.1. Имена и уровни серьезности, определенные в перечислении System.Logger.Level

Имя	Серьезность
ALL	Integer.MIN_VALUE
TRACE	400
DEBUG	500
INFO	800
WARNING	900
ERROR	1000
OFF	Integer.MAX_VALUE

Если включено протоколирование на уровне `INFO`, то протоколируются все сообщения уровней `INFO`, `WARNING` и `ERROR`. Если требуется протоколировать сообщения любого уровня, то в конфигурационном файле Log4j следует задать уровень `trace` или `all` (см. листинг 19.2).

Из приведенной ниже распечатки видно, что платформенные классы трижды обращались к методу `getLogger()` класса `Log4jLoggerFinder`. Первые два обращения были из модуля `javax.management`, а третий связан с запросом диспетчера из метода `main()` класса `PlatformLoggerTest`.

В журнале мы видим собственные сообщения, но нет ни одного сообщения от классов JDK. Уверен, вам было бы очень любопытно увидеть такие сообщения. Да

в конце концов весь пример для того и затевался. А как узнать имя класса JDK, который отправил сообщение через ваш платформенный диспетчер протоколирования, и как заставить эти классы что-то протоколировать? Прямого и простого способа нет. Я искал в исходном коде JDK упоминания класса `sun.util.logging.PlatformLogger`, который используется для протоколирования платформенных сообщений, и нашел, что модуль `javax.management` выводит в протокол сообщения уровня `TRACE`. Чтобы их увидеть, нужно задать уровень `trace` в конфигурационном файле Log4j и снова выполнить класс `PlatformLoggerTest`. Тогда в журнале появится множество сообщений.

Вернемся к использованию класса `Currency` в классе `PlatformLoggerTest`. Я выбрал его, чтобы показать сообщения, протоколируемые классом JDK `java.util.Currency`. Когда мы запрашиваем список всех валют, JDK читает встроенный список и пользовательский файл `currency.properties` в каталоге `JAVA_HOME\lib`. В данном случае мы для выполнения примера использовали JDK, так что `JAVA_HOME` ссылается на `JDK_HOME`. Создайте текстовый файл `JDK_HOME\lib\currency.properties`, содержимое которого показано в листинге 19.7. Это всего одно слово `ABadCurrencyFile`.

Листинг 19.7. Содержимое файла JDK_HOME\lib\currency.properties

```
ABadCurrencyFile
```

Класс `Currency` пытается загрузить файл свойств `currency.properties`, который должен бы содержать пары `имя=значение`. Но созданный нами файл не является допустимым файлом свойств. Когда класс `Currency` пытается загрузить его, возбуждается исключение и класс выводит сообщение, пользуясь платформенным диспетчером протоколирования. Теперь мы знаем, что создали неправильный файл валют, и видим платформенный диспетчер в действии.

Снова выполнив класс `PlatformLoggerTest`, получаем такой результат:

```
Log4jLoggerFinder.getLogger(): [имя=javax.management.mbeanserver, модуль=java.management]
Log4jLoggerFinder.getLogger(): [имя=javax.management.misc, модуль=java.management]
Log4jLoggerFinder.getLogger(): [имя=java.util.Currency, модуль=java.base]
Загружено валют: 225
Log4jLoggerFinder.getLogger(): [имя=Log4jLogger, модуль=com.jdojo.logger]
```

Как видим, модуль `java.base` запросил платформенный диспетчер протоколирования, передав имя `java.util.Currency`, поскольку обнаружил недопустимый файл валют. В листинге 19.8 показано содержимое файла журнала.

Листинг 19.8. Содержимое файла logs/platform.log

```
2017-02-09 10:45:52,413 INFO com.jdojo.logger.Log4jLogger [main] currency.properties entry
for ABADCURRENCYFILE is ignored because of the invalid country code.
2017-02-09 10:45:52,420 ERROR com.jdojo.logger.Log4jLogger [main] Произошла неизвестная ошибка.
2017-02-09 10:45:52,420 INFO com.jdojo.logger.Log4jLogger [main] Для сведения
```

Что дальше

Я показал, как сконфигурировать Log4j 2.0 в качестве системы платформенного протоколирования. Но до использования этого примера в производственной среде еще далеко. Одно из возможных улучшений – включить имя класса, запротоколировавшего сообщение. В листинге 19.8 видно, что во всех сообщениях имя класса одно и то же – `com.jdojo.logger.Log4jLogger`, что, конечно, неправильно, т. к. одно сообщение пришло из класса `com.jdojo.logger.PlatformLoggerTest`, а другое – из класса `java.util.Currency`. Как исправить эту ошибку?

Сначала разберемся, в чем собственно проблема. Имя класса диспетчера определяет библиотека Log4j. Она просто смотрит, кто вызвал метод `log()` и считает, что это и есть класс, запротоколировавший сообщение. В листинге 19.3 метод `log()` из библиотеки Log4j вызывают два метода `log()`, делегирующие свою работу. Log4j видит, что отправителем сообщений является класс `com.jdojo.logger.Log4jLogger`, и подставляет его имя в отформатированное сообщение. Исправить это можно двумя способами.

- В классе Log4jLogger форматировать сообщение самостоятельно, пользуясь API навигации по стеку, добавленным в JDK 9. Этот API дает имя вызывающего класса и другую информацию. При таком подходе нужно будет изменить формат (layout) в конфигурационном файле Log4j, чтобы Log4j не пыталась определить имя вызывающего класса и включить его в сообщение.
- Подождать выхода следующей версии Log4j, которая, *возможно*, будет поддерживать платформенное протоколирование JDK 9 на внутреннем уровне. Но на момент написания книги таких анонсов не было.

Унифицированное протоколирование JVM

В JDK 9 добавлен новый параметр командной строки `-Xlog`, дающий единую точку доступа ко всем сообщениям, протоколируемым компонентами JVM. Порядок его использования довольно сложный, и, прежде чем объяснять его, я опишу, как протоколируются сообщения.

> **Совет.** Для печати справки по параметру `-Xlog` выполните команду `java` с параметром `-Xlog:help`. Справка содержит синтаксис и значения всех параметров с примерами.

Перечислим ряд моментов, которые следует иметь в виду, говоря о протоколировании сообщений в JVM.

- JVM должна определить тему (или компонент JVM), к которой относится сообщение. Например, если сообщение касается сборки мусора, то в нем должна быть соответствующая метка. Сообщение может относиться к нескольким темам, например, сборка мусора и управление кучей. Следовательно, с сообщением может быть ассоциировано несколько меток.
- Как и любой механизм протоколирования, JVM задает уровень сообщения: информационное, предупреждение и т. д.
- Должна быть возможность включить в сообщение дополнительную контекстную информацию (декораторы): текущая дата и время, поток, отправивший сообщение, связанные с сообщением метки и т. д.

- Куда выводить сообщение? На `stdout`, `stderr` или в один или несколько файлов? Должна ли существовать возможность задать параметры файлов журналов, например: имена, размеры и политику ротации?

Теперь можно перейти к конкретным терминам, используемым при описании протоколирования в JVM:

- метки;
- уровни;
- декораторы;
- вывод.

В следующем примере класс `com.jdojo.Welcome`, написанный в главе 3, выполняется с параметром `-Xlog`. В стандартный вывод протоколируются все сообщения с меткой `gc`, уровнем серьезности `trace` и выше и декораторами `level`, `time` и `tags`.

```
C:\Java9revealed> java -Xlog:gc=trace:stdout:level,time,tags
 --module-path com.jdojo.intro\dist
 --module com.jdojo.intro/com.jdojo.intro.Welcome
```

```
[2017-02-10T12:50:11.412-0600][trace][gc] MarkStackSize: 4096k MarkStackSizeMax: 16384k
[2017-02-10T12:50:11.427-0600][debug][gc] ConcGCThreads: 1
[2017-02-10T12:50:11.432-0600][debug][gc] ParallelGCThreads: 4
[2017-02-10T12:50:11.433-0600][debug][gc] Initialize mark stack with 4096 chunks, maximum 16384
[2017-02-10T12:50:11.436-0600][info ][gc] Using G1
Добро пожаловать в систему модулей
Имя модуля: com.jdojo.intro
```

Метки сообщений

С каждым протоколируемым сообщением ассоциирована одна или несколько меток, в совокупности они называются набором меток (tag-set). Ниже перечислены все метки, существовавшие на момент написания книги. В будущем их состав может измениться. Для получения актуального списка поддерживаемых меток выполните команду `java` с параметром `-Xlog:help`.

```
add, age, alloc, aot, annotation, arguments, attach, barrier, biasedlocking, blocks, bot,
breakpoint, census, class, classhisto, cleanup, compaction, constraints, constantpool,
coops, cpu, cset, data, defaultmethods, dump, ergo, exceptions, exit, fingerprint, freelist,
gc, hashtables, heap, humongous, ihop, iklass, init, itables, jni, jvmti, liveness, load,
loader, logging, mark, marking, methodcomparator, metadata, metaspace, mmu, modules,
monitorinflation, monitormismatch, nmethod, normalize, objecttagging, obsolete, oopmap, os,
pagesize, patch, path, phases, plab, promotion, preorder, protectiondomain, ref, redefine,
refine, region, remset, purge, resolve, safepoint, scavenge, scrub, stacktrace, stackwalk,
start, startuptime, state, stats, stringdedup, stringtable, stackmap, subclass, survivor,
sweep, task, thread, tlab, time, timer, update, unload, verification, verify, vmoperation,
vtables, workgang, jfr, system, parser, bytecode, setting, event
```

Если вас интересуют сообщения, относящиеся к сборке мусора и инициализации, укажите метки `gc` и `startuptime`. У большинства меток в этом списке странные имена, и предназначены они для разработчиков JVM, а не для прикладных

программистов. Я не смог найти содержательные описания всех меток. По адресу https://bugs.openjdk.java.net/browse/JDK-8146948 размещен запрос о публикации описаний меток. Ошибка помечена исправленной, но я так и не увидел параметра, который позволил бы вывести описание меток.

Совет. Специальное значение `all` параметра `-Xlog` означает, что JVM должна протоколировать все сообщения вне зависимости от меток. Оно подразумевается по умолчанию.

Уровни сообщений

Уровень серьезности определяет, какие сообщения протоколировать. Определены следующие уровни (в порядке возрастания серьезности): `trace`, `debug`, `info`, `warning`, `error`. Если разрешено протоколирование сообщение с уровнем серьезности `S`, то и сообщения с более высокими уровнями также протоколируются.

Совет. Задание специального уровня серьезности `off` в параметре `-Xlog` вообще выключает протоколирование. По умолчанию подразумевается уровень `info`.

Декораторы сообщений

В сообщения JVM можно включить дополнительную информацию. Такие дополнительные вставки называются декораторами и помещаются в начало сообщения. Каждый декоратор заключен в квадратные скобки. В табл. 19.2 перечислены все декораторы с указанием длинного и короткого имени. В параметре `-Xlog` можно указывать как длинное, так и короткое имя декоратора.

Таблица 19.2. Описание декораторов с указанием длинных и коротких имен

Длинное имя	Короткое имя	Описание
`hostname`	`hn`	Имя компьютера
`level`	`l`	Уровень серьезности сообщения
`pid`	`p`	Идентификатор процесса
`tags`	`tg`	Все ассоциированные с сообщением метки
`tid`	`ti`	Идентификатор потока
`time`	`t`	Текущая дата и время в формате ISO-8601 (например, 2017-02-10T18:42:58.418+0000)
`timemillis`	`tm`	Текущее время в миллисекундах в виде числа; значение, возвращаемое методом `System.currentTimeMillis()`
`timenanos`	`tn`	Текущее время в наносекундах в виде числа; значение, возвращаемое методом `System.nanoTime()`
`uptime`	`u`	Время с момента запуска JVM в секундах и миллисекундах (например, 9.219s)
`uptimemillis`	`um`	Количество миллисекунд с момента запуска JVM
`uptimenanos`	`un`	Количество наносекунд с момента запуска JVM
`utctime`	`utc`	Текущая дата и время в формате UTC (например, 2017-02-10T12:42:58.418-0600)

Совет. Чтобы отключить декораторы, задайте специальный декоратор `none` в параметре `-Xlog`. По умолчанию подразумеваются декораторы `uptime,level,tags` в указанном порядке.

Место назначения сообщения

Можно задать одно из трех мест расположения журнала JVM:

- `stdout`
- `stderr`
- `file=<file-name>`

Задание `stdout` и `stderr` означает, что местом назначения сообщений JVM является стандартный поток вывода и стандартный поток ошибок соответственно. По умолчанию подразумевается `stdout`.

Задание `file` позволяет указать имя текстового файла, в который записывается журнал. В имени файла можно употреблять спецификаторы `%p` и `%t`, вместо которых будет подставлен идентификатор процесса и время запуска JVM соответственно. Например, если задать в качестве места назначения `file=jvm%p_%t.log`, то при каждом запуске JVM сообщения будут протоколироваться в файл с именем вида:

- `jvm2348_2017-02-10_13-26-05.log`
- `jvm7292_2017-02-10_13-26-06.log`

При каждом запуске JVM будет создаваться такой файл журнала. Здесь 2348 и 7292 – идентификаторы процесса JVM.

Совет. Если ключевые слова `stdout` и `stderr` отсутствуют, то считается, что местом назначения является файл. Поэтому вместо `file=jvm.log` можно писать просто `jvm.log`.

При выводе в текстовый файл можно задавать также дополнительные параметры:

- `filecount=<file-count>`
- `filesize=<file-size>`

Они управляют максимальным размером одного файла журнала и максимальным числом журналов. Рассмотрим такое задание параметра:

```
file=jvm.log::filesize=1M,filecount=3
```

Обратите внимание на двойное двоеточие (`::`). Что это значит, я объясню в следующем разделе. В данном случае файл журнала называется `jvm.log`. Максимальный размер журнала составляет `1M` (1 МБ), а максимальное число журналов равно 3. При этом будет создано четыре файла: `jvm.log`, `jvm.log.0`, `jvm.log.1` и `jvm.log.2`. Сообщения всегда пишутся в файл `jvm.log`, а остальные три файла ротируются, когда размер активного файла окажется больше 1 МБ. Для задания размера файла можно использовать единицу `K` (килобайты) или `M` (мегабайты). Если суффикс `K` или `M` не указан, предполагается размер в байтах.

Синтаксис параметра -Xlog

Параметр `-Xlog` задается в следующем общем виде:

```
-Xlog[:<contents>][:[<output>][:[<decorators>][:<output-options>]]]
```

Подпараметры `-Xlog` разделяются знаками двоеточия. Двоеточие, соответствующее отсутствующей предшествующей части, должно присутствовать. Например, `-Xlog::stderr` означает, что для всех подпараметров подразумеваются значения по умолчанию за исключением подпараметра `<output>`, равного `stderr`.

Ниже приведен простейший случай использования `-Xlog`, когда все сообщения JVM выводятся в стандартный вывод:

```
java -Xlog --module-path com.jdojo.intro\dist --module com.jdojo.intro/com.jdojo.intro.
Welcome
```

У `-Xlog` есть два специальных подпараметра: `help` и `disable`. `-Xlog:help` выводит справку по работе с `-Xlog`, `-Xlog:disable` выключает протоколирование JVM полностью. Конечно, вместо `-Xlog:disable` можно было бы просто не употреблять `-Xlog` вовсе, но значение `disable` существует по другой причине. Параметр `-Xlog` может встречаться несколько раз в одной команде. Если в разных вхождениях один и тот же подпараметр устанавливается несколько раз, то действовать будет последнее значение. Поэтому можно в начале задать `-Xlog:disable`, а позже установить конкретный режим протоколирования, т. е. сначала отменить все умолчания, а затем включить те подпараметры, которые нас интересуют.

В части `<contents>` задаются метки и уровни серьезности протоколируемых сообщений. Синтаксически она выглядит так:

```
tag1[+tag2...][*][=level][,...]
```

Знак `+` в части `<contents>` соответствует логическому И. Например, `gc+exit` означает, что нужно протоколировать все сообщения, для которых набор меток состоит ровно из двух меток: `gc` и `exit`. Знак `*` в конце метки означает «по крайней мере». Например, если задано `gc*`, то протоколируются все сообщения, для которых присутствует метка `gc` и, возможно, какие-то еще, т. е. сообщения с наборами меток `[gc]`, `[gc,exit]`, `[gc,remset,exit]` и т. д. Конструкция `gc+exit*` означает, что нужно протоколировать сообщения, для которых набор меток содержит по крайней мере `gc` и `exit`, т. е. `[gc,exit]`, `[gc,remset,exit]` и т. д. Для каждой метки можно также задать уровень серьезности, при котором сообщение с такой меткой протоколируется. Например, `gc=trace` означает, что протоколируются все сообщения, для которых набор меток содержит только `gc` с уровнем серьезности `trace` или выше. Можно задать несколько таких критериев через запятую. Так, `gc=trace,heap=error` означает, что нужно протоколировать все сообщения, для которых набор меток либо содержит метку `gc` и уровень серьезности не ниже `trace`, либо метку `heap` с уровнем серьезности `error`.

Ниже приводится несколько команд протоколирования сообщений JVM, удовлетворяющих различным критериям. Показан результат на моем компьютере, на вашем картина может быть иной. В следующей команде заданы метки `gc` и `startuptime`, а для остальных параметров подразумеваются значения по умолчанию:

```
C:\Java9Revealed>java -Xlog:gc,startuptime --module-path com.jdojo.intro\dist
 --module com.jdojo.intro/com.jdojo.intro.Welcome
```

```
[0.017s][info][startuptime] StubRoutines generation 1, 0.0002258 secs
[0.022s][info][gc ] Using G1
[0.022s][info][startuptime] Genesis, 0.0045639 secs
...
```

Задание `-Xlog` без подпараметров эквивалентно заданию `-Xlog:all=info:stdout:uptime,level,tags`. При этом на `stdout` выводятся все сообщения с уровнем серьезности не ниже `info` с декораторами `uptime`, `level` и `tags`. В следующей команде демонстрируется протоколирование JVM с параметрами по умолчанию. Показана только часть вывода:

```
C:\Java9Revealed>java -Xlog --module-path com.jdojo.intro\dist
 --module com.jdojo.intro/com.jdojo.intro.Welcome
```

```
[0.015s][info][os] SafePoint Polling address: 0x000001195fae0000
[0.015s][info][os] Memory Serialize Page address: 0x000001195fdb0000
[0.018s][info][biasedlocking] Aligned thread 0x000001195fb37f40 to 0x000001195fb38000
[0.019s][info][class,path   ] bootstrap loader class path=C:\java9\lib\modules
[0.019s][info][class,path   ] classpath:
[0.020s][info][class,path   ] opened: C:\java9\lib\modules
[0.020s][info][class,load   ] opened: C:\java9\lib\modules
[0.027s][info][os,thread    ] Thread is alive (tid: 17724).
[0.027s][info][os,thread    ] Thread is alive (tid: 6436).
[0.033s][info][gc           ] Using G1
[0.034s][info][startuptime  ] Genesis, 0.0083975 secs
[0.038s][info][class,load   ] java.lang.Object source: jrt:/java.base
[0.226s][info][os,thread           ] Thread finished (tid: 7584).
[0.226s][info][gc,heap,exit        ] Heap
[0.226s][info][gc,heap,exit        ]  Metaspace      used 6398K, capacity 6510K,
[0.226s][info][safepoint,cleanup   ] mark nmethods, 0.0000057 secs
[0.226s][info][os,thread           ] Thread finished (tid: 3660).
```

Следующая команда приводит к протоколированию всех сообщений, для которых присутствует по крайней мере метка `gc`, а уровень серьезности не ниже `debug`. Сообщения записываются в файл `gc.log` в текущем каталоге и снабжаются декоратором `time`. Отметим, что на `stdout` выводятся еще два сообщения из метода `main()` класса `Welcome`. Но я привожу только часть файла `gc.log`, а не то, что печатается на стандартный вывод.

```
C:\java9revealed>java -Xlog:gc*=trace:file=gc.log:time
 --module-path com.jdojo.intro\dist
 --module com.jdojo.intro/com.jdojo.intro.Welcome
```

```
[2017-02-11T08:40:23.942-0600]   Maximum heap size 2113804288
[2017-02-11T08:40:23.942-0600]   Initial heap size 132112768
[2017-02-11T08:40:23.942-0600]   Minimum heap size 6815736
[2017-02-11T08:40:23.942-0600] MarkStackSize: 4096k MarkStackSizeMax: 16384k
[2017-02-11T08:40:23.966-0600] Heap region size: 1M
[2017-02-11T08:40:23.966-0600] WorkerManager::add_workers() : created_workers: 4
[2017-02-11T08:40:23.966-0600] Initialize Card Live Data
[2017-02-11T08:40:23.966-0600] ParallelGCThreads: 4
[2017-02-11T08:40:23.966-0600] WorkerManager::add_workers() : created_workers: 1
...
```

Следующая команда протоколирует те же сообщения, что предыдущая, но без декораторов:

```
C:\java9revealed>java -Xlog:gc*=trace:file=gc.log:none
 --module-path com.jdojo.intro\dist
 --module com.jdojo.intro/com.jdojo.intro.Welcome
```

```
  Maximum heap size 2113804288
  Initial heap size 132112768
  Minimum heap size 6815736
MarkStackSize: 4096k MarkStackSizeMax: 16384k
Heap region size: 1M
WorkerManager::add_workers() : created_workers: 4
Initialize Card Live Data
ParallelGCThreads: 4
WorkerManager::add_workers() : created_workers: 1
...
```

Следующая команда протоколирует те же сообщения, что предыдущая, но выводит их в ротируемый набор из 10 файлов с базовым именем `gc.log` и размером 5 МБ:

```
C:\Java9Revealed>java -Xlog:gc*=trace:file=gc.log:none:filesize=5m,filecount=10
 --module-path com.jdojo.intro\dist --module com.jdojo.intro/com.jdojo.intro.Welcome
```

Следующая команда протоколирует все сообщения, для которых присутствует метка `gc`, а уровень серьезности не ниже `debug`. Сообщения с меткой `exit` не протоколируются, в частности, не протоколируются сообщения, содержащие обе метки `gc` и `exit`. Сообщения выводятся на `stdout` с декораторами по умолчанию. Показана только часть вывода.

```
C:\Java9Revealed>java -Xlog:gc*=debug,exit*=off --module-path com.jdojo.intro\dist
 --module com.jdojo.intro/com.jdojo.intro.Welcome
```

```
[0.015s][info][gc,heap] Heap region size: 1M
```

```
[0.015s][debug][gc,heap] Minimum heap 8388608 Initial heap 132120576 Maximum heap
2113929216
[0.015s][debug][gc,ergo,refine] Initial Refinement Zones: green: 4, yellow: 12, red: 20, min
yellow size: 8
[0.016s][debug][gc,marking,start] Initialize Card Live Data
[0.016s][debug][gc,marking      ] Initialize Card Live Data 0.024ms
[0.016s][debug][gc              ] ConcGCThreads: 1
[0.018s][debug][gc,ihop         ] Target occupancy update: old: 0B, new: 132120576B
[0.019s][info ][gc              ] Using G1
[0.182s][debug][gc,metaspace,freelist]    space @ 0x000001e7dbeb8260 704K, 99% used
[0x000001e7fe880000, 0x000001e7fedf8400, 0x000001e7fee00000, 0x000001e7ff080000)
[0.191s][debug][gc,refine            ] Stopping 0
...
```

Следующая команда протоколирует сообщения с меткой `startuptime` и уровнем не ниже `info`, снабжая их декораторами `hostname`, `uptime`, `level` и `tags`. Все остальные подпараметры принимают значения по умолчанию. Сообщения выводятся на `stdout`. Обратите внимание на двойное двоеточие (`::`). Оно необходимо, потому что мы не задали место назначения.

```
C:\Java9Revealed>java -Xlog:startuptime::hostname,uptime,level,tags
 --module-path com.jdojo.intro\dist --module com.jdojo.intro/com.jdojo.intro.Welcome
```

```
[0.015s][kishori][info][startuptime] StubRoutines generation 1, 0.0002574 secs
[0.019s][kishori][info][startuptime] Genesis, 0.0038339 secs
[0.019s][kishori][info][startuptime] TemplateTable initialization, 0.0000081 secs
[0.020s][kishori][info][startuptime] Interpreter generation, 0.0010698 secs
[0.032s][kishori][info][startuptime] StubRoutines generation 2, 0.0001518 secs
[0.032s][kishori][info][startuptime] MethodHandles adapters generation, 0.0000229 secs
[0.033s][kishori][info][startuptime] Start VMThread, 0.0001491 secs
[0.055s][kishori][info][startuptime] Initialize java.lang classes, 0.0224295 secs
[0.058s][kishori][info][startuptime] Initialize java.lang.invoke classes, 0.0015945 secs
[0.162s][kishori][info][startuptime] Create VM, 0.1550707 secs
Добро пожаловать в систему модулей.
Имя модуля: com.jdojo.intro
```

Резюме

В JDK 9 полностью переработаны системы протоколирования для платформенных классов (входящих в JDK) и компонентов JVM. Новый API позволяет по своему выбору задать систему протоколирования, которая будет использоваться для вывода сообщений из платформенных классов. Введен также новый параметр командной строки, позволяющий получить доступ к сообщениям от всех компонентов JVM.

API платформенного протоколирования позволяет задать диспетчер протоколирования, который будет использоваться всеми платформенными классами.

Можно взять существующие библиотеки протоколирования, например, Log4j. API состоит из класса `java.lang.System.LoggerFinder` и интерфейса `java.lang.System.Logger`.

Платформенный диспетчер протоколирования представляется экземпляром интерфейса `System.Logger`. Класс `System.LogFinder` – это интерфейс службы. Вы должны предоставить реализацию этого интерфейса, возвращающую экземпляр `System.Logger`. Для получения `System.Logger` можно воспользоваться методом `getLogger()` класса `java.lang.System`. Объявление модуля вашего приложения должно содержать предложение `provides`, указывающее на реализацию интерфейса службы `System.LogFinder`. В противном случае будет использован диспетчер по умолчанию.

JDK 9 позволяет протоколировать сообщения от всех компонентов JVM с помощью единственного параметра `-Xlog`, в котором можно задать типы сообщений, уровень серьезности, место назначения, декораторы и свойства набора журналов. С каждым сообщением связан набор меток. Уровень серьезности сообщения определяется элементами перечисления `System.Logger.Level`. Местом назначения может быть `stdout`, `stderr` или `file`.

Глава 20

Другие изменения в JDK 9

Краткое содержание главы:

- знак подчеркивания как новое ключевое слово;
- улучшенный синтаксис блоков `try` с ресурсами;
- использование ромбовидного оператора в анонимных классах;
- использование закрытых методов в интерфейсах;
- применение аннотации `@SafeVarargs` к закрытым методам;
- отбрасывание вывода подпроцессов;
- новые методы в классах `Math` и `StrictMath`;
- потоки объектов `Optional` и другие новые операции с `Optional`;
- использование уведомлений об активном ожидании;
- улучшения в Time API и классах `Matcher` и `Objects`;
- сравнение массивов и срезок массивов;
- усовершенствования в документации Java и новые возможности поиска;
- поддержка платформенного рабочего стола;
- использование глобальных и локальных фильтров в процессе десериализации объектов;
- передача данных из входного потока в выходной, а также дублирование и срезка буферов.

В Java SE 9 много крупных и мелких изменений. Самые значимые – система модулей, клиентский API HTTP/2 и т. д. Каждому из них посвящена отдельная глава. А в этой главе рассматриваются изменения, которые важны для разработчиков, но не настолько значительны, чтобы отводить под них целую главу. Необязательно читать эту главу последовательно. В каждом разделе излагается одна тема. Если вас интересует что-то конкретное, то можете сразу перейти к нужному разделу.

Исходный код примеров находится в модуле `com.jdojo.misc`, объявление которого приведено в листинге 20.1.

Листинг 20.1. Объявление модуля com.jdojo.misc

```
// module-info.java
module com.jdojo.misc {
   requires java.desktop;
   exports com.jdojo.misc;
}
```

Этот модуль читает модуль `java.desktop`, который понадобится для реализации платформенно-зависимых возможностей рабочего стола.

Знак подчеркивания — ключевое слово

В JDK 9 знак подчеркивания (_) является ключевым словом, поэтому не может выступать в роли односимвольного идентификатора: имени переменной, метода, типа и т. д. Однако в многосимвольных идентификаторах его использование по-прежнему разрешено. Рассмотрим программу в листинге 20.2.

Листинг 20.2. Программа, в которой знак подчеркивания используется в роли идентификатора

```
// UnderscoreTest.java
package com.jdojo.misc;

public class UnderscoreTest {
   public static void main(String[] args) {
      // Используем знак подчеркивания в качестве идентификатора. В JDK 8
      // компилятор при этом выдавал предупреждение, а в JDK 9 это ошибка.
      int _ = 19;
      System.out.println(_);

      // Используем знак подчеркивания в многосимвольном идентификаторе. Это
      // нормально и в JDK 8, и в JDK 9.
      final int FINGER_COUNT = 20;
      final String _prefix = "Sha";
   }
}
```

В JDK 8 компиляция класса `UnderscoreTest` приводит к двум предупреждениям об использовании знака подчеркивания в качестве идентификатора: в объявлении переменной и при вызове метода `System.out.println()`. Предупреждение выдается для каждого такого случая использования знака подчеркивания.

```
com.jdojo.misc\src\com\jdojo\misc\UnderscoreTest.java:8: warning: '_' used as an
identifier
       int _ = 19;
           ^
  (use of '_' as an identifier might not be supported in releases after Java SE 8)
```

```
com.jdojo.misc\src\com\jdojo\misc\UnderscoreTest.java:9: warning: '_' used as an
identifier
        System.out.println(_);
                           ^
  (use of '_' as an identifier might not be supported in releases after Java SE 8)
2 warnings
```

При компиляции класса `UnderscoreTest` в JDK 9 выдаются две ошибки:

```
com.jdojo.misc\src\com\jdojo\misc\UnderscoreTest.java:8: error: as of release 9, '_' is a
keyword, and may not be used as an identifier
        int _ = 19;
            ^
com.jdojo.misc\src\com\jdojo\misc\UnderscoreTest.java:9: error: as of release 9, '_' is a
keyword, and may not be used as an identifier
        System.out.println(_);
                           ^
2 errors
```

Каков специальный смысл знака подчеркивания и как его использовать? В JDK 9 запрещается использовать подчеркивание в качестве идентификатора. Проектировщики JDK собираются наделить его специальным смыслом в будущих версиях, так что подождем до JDK 10 или 11. Закомментируйте приведенные строки или весь код класса, чтобы проект NetBeans компилировался без ошибок.

Усовершенствование блоков try с ресурсами

В JDK 7 в пакет `java.lang` был добавлен интерфейс `AutoCloseable`:

```
public interface AutoCloseable {
    void close() throws Exception;
}
```

Тогда же была добавлена новая синтаксическая конструкция – `try` с ресурсами – для управления автозакрываемыми объектами (или ресурсами). Последовательность действий была такой:

- в начале блока ссылка на ресурс присваивается *вновь объявленной* переменной;
- в теле блока мы работаем с ресурсом;
- по выходе из тела блока будет автоматически вызван метод `close()` объекта, представляющего ресурс.

Это позволило избежать написания трафаретного кода с блоком `finally`. Ниже показано, как разработчики управляли закрываемым ресурсом в предположении, что некий класс `Resource` реализует интерфейс `AutoCloseable`:

```
/* До JDK 7*/
Resource res = null;
try{
```

```
   // Создать ресурс
   res = new Resource();
   // Код работы с res
} finally {
   try {
      if(res != null) {
         res.close();
      }
   } catch(Exception e) {
      e.printStackTrace();
   }
}
```

Блок `try` с ресурсами в JDK 7 позволил существенно улучшить ситуацию. Теперь показанный выше фрагмент можно переписать так:

```
try (Resource res = new Resource()) {
    // Код работы с res
}
```

Здесь метод `close()` объекта `res` вызывается, когда поток управления покидает блок. В блоке `try` можно перечислить несколько ресурсов, разделив их знаками точки с запятой:

```
try (Resource res1 = new Resource(); Resource res2 = new Resource()) {
    // Код работы с res1 и res2
}
```

По выходе из блока будут вызваны методы `close()` обоих ресурсов `res1` и `res2`. Ресурсы закрываются в порядке, обратном созданию. В данном случае сначала будет вызван метод `res2.close()`, а потом `res1.close()`.

В JDK 7 и 8 требуется, чтобы переменная, ссылающаяся на ресурс, была объявлена прямо в блоке `try` с ресурсами. Если ссылка на ресурс передана в аргументе метода, то написать такой код не получится:

```
void useIt(Resource res) {
    // Ошибка компиляции в JDK 7 и 8
    try(res) {
        // Код работы с res
    }
}
```

Чтобы обойти это ограничение, необходимо объявить новую переменную типа `Resource` и инициализировать ее значением аргумента. Это подход демонстрируется в следующем фрагменте, где по выходе из блока будет вызван метод `close()` переменной `res1`:

```
void useIt(Resource res) {
    try(Resource res1 = res) {
```

```
        // Код работы с res
    }
}
```

В JDK 9 указанное ограничение снято. Теперь для ссылки на ресурс, управляемый блоком `try` с ресурсами, можно использовать *финальную* или *эффективно финальную* переменную. Переменная называется финальной, если в ее объявлении присутствует ключевое слово `final`:

```
// res – финальная переменная
final Resource res = new Resource();
```

Переменная называется эффективно финальной, если ее значение не изменяется после инициализации. В следующем фрагменте переменная `res` эффективно финальная, хотя в ее объявлении нет слова `final`. Она инициализируется и больше никогда не изменяется.

```
void doSomething() {
    // res – эффективно финальная переменная
    Resource res = new Resource();
    res.useMe();
}
```

В JDK 9 следующий код корректен:

```
Resource res = new Resource();
try (res) {
    // Код работы с res
}
```

Если мы хотим управлять несколькими ресурсами, то можем поступить так:

```
Resource res1 = new Resource();
Resource res2 = new Resource();
try (res1; res2) {
    // Здесь res1 и res2 используются
}
```

В одном блоке `try` с ресурсами можно сочетать оба подхода. В следующем фрагменте две эффективно финальные переменные объявлены вне блока, а одна – внутри:

```
Resource res1 = new Resource();
Resource res2 = new Resource();
try (res1; res2; Resource res3 = new Resource()) {
    // Здесь res1, res2 и res3 используются
}
```

Начиная с JDK 7, переменные, объявленные внутри блока `try` с ресурсами, неявно являются финальными. В следующем фрагменте переменная объявляется финальной явно:

```
Resource res1 = new Resource();
Resource res2 = new Resource();

// Явно объявить res3 финальной
try (res1; res2; final Resource res3 = new Resource()) {
    // Здесь res1, res2 и res3 используются
}
```

Рассмотрим законченный пример. В JDK есть несколько автозакрываемых классов, например, классы `InputStream` и `OutputStream` в пакете `java.io`. В листинге 20.3 приведен код класса `Resource`, реализующего интерфейс `AutoCloseable`. Объекты этого класса можно использовать как ресурсы, управляемые блоком `try` с ресурсами. Переменная экземпляра `id` служит для идентификации ресурсов. Конструктор и другие методы при вызове просто печатают сообщения.

Листинг 20.3. Класс Resource, реализующий интерфейс AutoCloseable

```
// Resource.java
package com.jdojo.misc;

public class Resource implements AutoCloseable {
   private final long id;

   public Resource(long id) {
      this.id = id;
      System.out.printf("Создан ресурс %d.%n", this.id);
   }

   public void useIt() {
      System.out.printf("Используется ресурс %d.%n", this.id);
   }

   @Override
   public void close() {
      System.out.printf("Закрывается ресурс %d.%n", this.id);
   }
}
```

В листинге 20.4 приведен код класса `ResourceTest`, демонстрирующего использование появившейся в JDK 9 возможности управлять ресурсами с помощью ссылающихся на них финальных или эффективно финальных переменных.

Листинг 20.4. Класс ResourceTest, демонстрирующий использование блоков `try-catch` в JDK 9

```
// ResourceTest.java
package com.jdojo.misc;

public class ResourceTest {
```

```
   public static void main(String[] args) {
      Resource r1 = new Resource(1);
      Resource r2 = new Resource(2);

      try(r1; r2) {
         r1.useIt();
         r2.useIt();
         r2.useIt();
      }

      useResource(new Resource(3));
   }

   public static void useResource(Resource res) {
      try(res; Resource res4 = new Resource(4)) {
         res.useIt();
         res4.useIt();
      }
   }
}
```

```
Создан ресурс 1.
Создан ресурс 2.
Используется ресурс 1.
Используется ресурс 2.
Using resource 2.
Закрывается ресурс 2.
Закрывается ресурс 1.
Создан ресурс 3.
Создан ресурс 4.
Используется ресурс 3.
Используется ресурс 4.
Закрывается ресурс 4.
Закрывается ресурс 3.
```

Ромбовидный оператор в анонимных классах

В JDK 7 появился ромбовидный оператор (<>), используемый для вызова конструктора универсального класса при условии, что компилятор способен вывести универсальный тип. Следующие два предложения эквивалентны, но во втором используется ромбовидный оператор:

```
// Явное задание универсального типа
Specify the generic type explicitly
List<String> list1 = new ArrayList<String>();
```

```
// Компилятор выводит ArrayList<> как ArrayList<String>
List<String> list2 = new ArrayList<>();
```

В JDK 7 не разрешалось использовать ромбовидный оператор при создании анонимного класса. В следующем фрагменте анонимный класс с ромбовидным оператором используется для создания экземпляра интерфейса `Callable<V>`:

```
// Ошибка компиляции в JDK 7 и 8
Callable<Integer> c = new Callable<>() {
   @Override
   public Integer call() {
      return 100;
   }
};
```

В JDK 7 и 8 компилятор выдает такое сообщение об ошибке:

```
error: cannot infer type arguments for Callable<V>
        Callable<Integer> c = new Callable<>() {
                                           ^
  reason: cannot use '<>' with anonymous inner classes
  where V is a type-variable:
    V extends Object declared in interface Callable
1 error
```

Чтобы исправить ошибку, нужно указать конкретный тип вместо ромбовидного оператора:

```
// Работает в JDK 7 и 8
Callable<Integer> c = new Callable<Integer>() {
   @Override
   public Integer call() {
      return 100;
   }
};
```

В JDK 9 добавлена поддержка ромбовидного оператора в анонимных классах при условии, что выведенные типы денотируемые. Использовать ромбовидный оператор в анонимных классах нельзя – даже в JDK 9 – если выведенный тип неденотируемый. Внутри компилятора Java используется много типов, которые невозможно записать с соблюдением синтаксиса Java. Типы, которые можно так записать, называются *денотируемыми* (denotable), а типы, о которых компилятор знает, но которые нельзя записать в Java-программе, – неденотируемыми. Например, `String` – денотируемый тип, поскольку его можно использовать в программе, а `Serializable & CharSequence` – неденотируемый, хотя компилятор понимает, что это такое. Это тип пересечения, реализующий оба интерфейса, `Serializable` и `CharSequence`. Типы пересечения допустимы в определениях универсальных типов, но объявить переменную такого типа невозможно:

```
// Не разрешено в Java-программе. Нельзя объявлять переменную типа пересечения.
Serializable & CharSequence var;

// Разрешено в Java-программе
class Magic<T extends Serializable & CharSequence> {
    // Какой-то код
}
```

В JDK 9 разрешен следующий фрагмент, в котором ромбовидный оператор используется в сочетании с анонимным классом:

```
// Ошибка компиляции в JDK 7 и 8, но разрешено в JDK 9.
Callable<Integer> c = new Callable<>() {
   @Override
   public Integer call() {
      return 100;
   }
};
```

Пользуясь приведенным выше определением класса `Magic`, в JDK 9 мы можем использовать анонимный класс следующим образом:

```
// Разрешено в JDK 9. <> выводится как <String>.
Magic<String> m1 = new Magic<>(){
    // Какой-то код
};
```

А такое использование класса `Magic` не компилируется в JDK 9, потому что компилятор выводит, что универсальный тип является типом пересечения, т. е. неденотируемым:

```
// Ошибка компиляции в JDK 9. <> выводится как <Serializable & CharSequence>,
// а этот тип неденотируемый.
Magic<?> m2 = new Magic<>(){
    // Какой-то код
};
```

При компиляции этого кода выдается такое сообщение об ошибке:

```
error: cannot infer type arguments for Magic<>
        Magic<?> m2 = new Magic<>(){
                              ^
  reason: type argument INT#1 inferred for Magic<> is not allowed in this
    context inferred argument is not expressible in the Signature attribute
  where INT#1 is an intersection type:
    INT#1 extends Object,Serializable,CharSequence
1 error
```

Закрытые методы в интерфейсах

В JDK 8 в интерфейсах стало возможно употреблять статические методы и методы по умолчанию. Но если в таких методах необходимо было несколько раз реализовать одну и ту же логику, то приходилось либо дублировать код, либо выносить логику в другой класс, чтобы скрыть реализацию. Рассмотрим интерфейс `Alphabet`, показанный в листинге 20.5.

Листинг 20.5. Интерфейс Alphabet, в котором два метода по умолчанию имеют общую логику

```
// Alphabet.java
package com.jdojo.misc;

public interface Alphabet {
   default boolean isAtOddPos(char c) {
      if (!Character.isLetter(c)) {
         throw new RuntimeException("Не буква: " + c);
      }

      char uc = Character.toUpperCase(c);
      int pos = uc - 64;

      return pos % 2 == 1;
   }

   default boolean isAtEvenPos(char c) {
      if (!Character.isLetter(c)) {
         throw new RuntimeException("Не буква: " + c);
      }

      char uc = Character.toUpperCase(c);
      int pos = uc - 64;

      return pos % 2 == 0;
   }
}
```

Методы `isAtOddpos()` и `isAtEvenPos()` проверяют, находится ли указанная буква на четной или на нечетной позиции в алфавите, в предположении, что мы имеем дело только с буквами английского алфавита. Считается, что буквы A и a занимают позицию 1, B и b – позицию 2 и т. д. Обратите внимание, что методы отличаются только предложениями `return`, а весь остальной код совпадает. Очевидно, здесь необходим рефакторинг. Идеально было бы вынести общий код в третий метод и вызывать его из обоих. Но в JDK 8 это не очень хорошо, потому что интерфейсы поддерживают только открытые методы. Стало быть, третий метод пришлось бы сделать открытым, видимым всем и каждому, а нам бы этого не хотелось.

JDK 9 приходит на помощь. В этой версии разрешено объявлять закрытые методы в интерфейсах. В листинге 20.6 показан переработанный вариант интерфейса `Alphabet`, в котором используется закрытый метод, содержащий код, общий для обоих методом. Я назвал новый интерфейс `AlphabetJdk9`, чтобы можно было включить в программу как новый, так и старый код. Теперь каждый из прежних методов занимает всего одну строку.

Листинг 20.6. Интерфейс AlphabetJdk9 с закрытым методом

```
// AlphabetJdk9.java
package com.jdojo.misc;

public interface AlphabetJdk9 {
   default boolean isAtOddPos(char c) {
      return getPos(c) % 2 == 1;
   }

   default boolean isAtEvenPos(char c) {
      return getPos(c) % 2 == 0;
   }

   private int getPos(char c) {
      if (!Character.isLetter(c)) {
         throw new RuntimeException("Не буква: " + c);
      }

      char uc = Character.toUpperCase(c);
      int pos = uc - 64;

      return pos;
   }
}
```

До JDK 9 все методы интерфейса неявно были открытыми. Напомним следующие простые правила, действующие для всех Java-программ:

- закрытый метод не наследуется и потому не может быть переопределен;
- финальный метод не может быть переопределен;
- абстрактный метод наследуется и может быть переопределен;
- метод по умолчанию является методом экземпляра и имеет реализацию по умолчанию. Предполагается, что он будет переопределен.

С появлением закрытых методов в JDK 9 стало необходимо соблюдать несколько правил при объявлении методов в интерфейсе. Не все комбинации модификаторов `abstract`, `public`, `private`, `static` и `final` поддерживаются, потому что не все имеют смысл. В табл. 20.1 перечислены поддерживаемые и неподдерживаемые комбинации модификаторов в объявлениях методов интерфейса в JDK 9. Отме-

тим, что модификатор `final` вообще недопустим. Согласно этой таблице, в интерфейсе можно объявить либо неабстрактный метод экземпляра, не являющийся методом по умолчанию, либо статический метод.

Таблица 20.1. Модификаторы, поддерживаемые в объявлениях методов интерфейса

Модификаторы	Поддерживается?	Описание
`public static`	Да	Поддерживается, начиная с JDK 8
`public abstract`	Да	Поддерживается, начиная с JDK 1
`public default`	Да	Поддерживается, начиная с JDK 8
`private static`	Да	Поддерживается, начиная с JDK 9
`private`	Да	Поддерживается, начиная с JDK 9. Неабстрактный метод экземпляра
`private abstract`	Нет	Эта комбинация не имеет смысла. Закрытый метод не наследуется, поэтому его нельзя переопределить, тогда как абстрактный метод полезен, только когда его переопределяют
`private default`	Нет	Эта комбинация не имеет смысла. Закрытый метод не наследуется, поэтому его нельзя переопределить, тогда как метод по умолчанию должен быть переопределен в случае использования

Применение аннотации @SafeVarargs к закрытым методам

Материализуемым (reifiable) называется тип, вся информация о котором доступна на этапе выполнения, как в случае `String`, `Integer`, `List` и т. д. Нематериализуемым называется тип, информация о котором была удалена компилятором посредством стирания типа (type erasure); например, тип `List<String>` после компиляции превращается в `List`.

В случае переменного числа аргументов нематериализуемого типа тип аргумента доходит только до компилятора. Компилятор стирает параметризованный тип и заменяет его массивом фактического типа: `Object[]` для неограниченных типов и массивом объектов верхнего граничного типа для ограниченных типов. Компилятор не может гарантировать безопасность операций, производимых над нематериализуемыми аргументами типа var-args внутри метода. Рассмотрим следующее определение метода:

```
<T> void print(T... args) {
   for(T element : args) {
      System.out.println(element);
   }
}
```

Компилятор заменит `print(T... args)` на `print(Object[] args)`. В теле этого метода нет небезопасных операций над аргументом `args`. Но вот метод, в котором такие операции есть:

```
public static void unsafe(List<Long>... rolls) {
   Object[] list = rolls;
   list[0] = List.of("One", "Two");

   // Небезопасно!!! На этапе выполнения возникнет исключение ClassCastException
   Long roll = rolls[0].get(0);
}
```

Метод `unsafe()` присваивает массив `rolls` типа `List<Long>` массиву `Object[]`, и это нормально. Он сохраняет список типа `List<String>` в первом элементе массива `Object[]`, что тоже разрешено. Последнее предложение компилируется без ошибок, потому что для `rolls[0]` выводится тип `List<Long>`, и предполагается, что метод `get(0)` вернет `Long`. Но на этапе выполнения возбуждается исключение `ClassCastException`, потому что тип фактического значения, возвращенного при вызове `rolls[0].get(0)`, – `String`, а не `Long`.

При объявлении таких методов, как `print()` и `unsafe()`, с переменным числом аргументов нематериализуемого типа компилятор выдается предупреждение вида

```
warning: [unchecked] Possible heap pollution from parameterized vararg type List<Long>
    public static void unsafe(List<Long>... rolls) {
                                            ^
```

Предупреждение выдается при объявлении такого метода и при каждом его вызове. Если метод `unsafe()` вызывается пять раз, то мы получим шесть предупреждений (одно при объявлении и пять при вызовах). Эти предупреждения можно подавить, расставив в местах объявления и вызовов аннотации `@SafeVarargs`. Снабжая такой аннотацией объявление метода, мы уверяем пользователей и компилятор, что в теле метода нет небезопасных операций над нематериализуемыми аргументами типа var-args. Вашего слова достаточно, чтобы компилятор не выдавал предупреждений. Но если во время выполнения раскроется обман, то будет возбуждено соответствующее исключение.

До JDK 9 аннотацию `@SafeVarargs` можно было применять к следующим конструкциям, допускающим исполнение (конструкторам и методам):

- конструкторы;
- статические методы;
- финальные методы.

Конструкторы, статические и финальные методы нельзя переопределить. Смысл применения аннотации `@SafeVarargs` только к непереопределяемым конструкциям состоит в том, чтобы защититься от включения небезопасных операций в переопределенный метод или конструктор. Предположим, что имеется класс `X`, который содержит метод `m1()` с аннотацией `@SafeVarargs`. Предположим далее, что класс `Y` наследует классу `X`. В классе `Y` метод `m1()` можно переопределить, включив в него

небезопасные операции. Это станет сюрпризом на этапе выполнения, поскольку программист, который писал код в терминах суперкласса `X`, не ожидал никаких небезопасных операций – ведь так было обещано в аннотации `m1()`.

Закрытые методы также не допускают переопределения, поэтому в JDK 9 было решено разрешить применение аннотации `@SafeVarargs` и к ним тоже. В листинге 20.7 приведен код класса с закрытым методом, снабженным аннотацией `@SafeVarargs`. Новый перечень исполняемых конструкций, для которых разрешена аннотация `@SafeVarargs`, в JDK 9 выглядит следующим образом:

- конструкторы;
- статические методы;
- финальные методы;
- закрытые методы.

Листинг 20.7. Применение аннотации `@SafeVarargs` к закрытому методу в JDK 9

```
// SafeVarargsTest.java
package com.jdojo.misc;

public class SafeVarargsTest {
   // Разрешено в JDK 9
   @SafeVarargs
   private <T> void print(T... args) {
      for(T element : args) {
         System.out.println(element);
      }
   }
   ...
}
```

При компиляции этого кода в JDK 8 возникает ошибка с сообщением о том, что `@SafeVarargs` неприменима к нефинальным методам, каковым в данном случае является закрытый метод. Чтобы увидеть сообщение, необходимо запустить компилятор с параметром `-Xlint:unchecked`.

```
com\jdojo\misc\SafeVarargsTest.java:6: error: Invalid SafeVarargs annotation. Instance
method <T> print(T...) is not final.
    private <T> void print(T... args) {
                     ^
  where T is a type-variable:
    T extends Object declared in method <T>print(T...)
```

Отбрасывание вывода процесса

В JDK 9 во вложенный класс `ProcessBuilder.Redirect` добавлена новая константа `DISCARD` типа `ProcessBuilder.Redirect`. Это место назначения можно указывать для потоков вывода и ошибок подпроцессов, если требуется отбросить все, что в них выводится. Это реализовано путем записи в «null-файл» операционной системы.

В листинге 20.8 приведен полный код программы, демонстрирующей отбрасывание вывода подпроцесса.

Листинг 20.8. Отбрасывание вывода процесса

```
// DiscardProcessOutput.java
package com.jdojo.misc;

import java.io.IOException;

public class DiscardProcessOutput {
   public static void main(String[] args) {
      System.out.println("Используется Redirect.INHERIT:");
      startProcess(ProcessBuilder.Redirect.INHERIT);

      System.out.println("\nИспользуется Redirect.DISCARD:");
      startProcess(ProcessBuilder.Redirect.DISCARD);
   }

   public static void startProcess(ProcessBuilder.Redirect outputDest) {
      try {
         ProcessBuilder pb = new ProcessBuilder()
               .command("java", "-version")
               .redirectOutput(outputDest)
               .redirectError(outputDest);

         Process process = pb.start();
         process.waitFor();
      } catch (IOException | InterruptedException e) {
         e.printStackTrace();
      }
   }
}
```

```
Используется Redirect.INHERIT:
java version "9-ea"
Java(TM) SE Runtime Environment (build 9-ea+157)
Java HotSpot(TM) 64-Bit Server VM (build 9-ea+157, mixed mode)
Используется Redirect.DISCARD:
```

Метод `startProcess()` запускает процесс, в котором выполняется программа `java` с аргументом `-version`. Методу передается место назначения вывода. При первом обращении местом назначения является `Redirect.INHERIT`, что позволяет подпроцессу использовать для печати сообщений стандартный поток вывода и стандартный поток ошибок. При втором обращении местом назначения является `Redirect.DISCARD`, поэтому мы не видим никакого вывода подпроцесса.

Новые методы класса StrictMath

В пакете `java.lang` есть два класса: `Math` и `StrictMath`. Оба содержат только статические члены и предоставляют методы для выполнения таких базовых операций, как извлечение квадратного корня, вычисление абсолютной величины и знака, тригонометрические и гиперболические функции. Зачем нужно два класса? Дело в том, что от класса `Math` не требуется возврат в точности одинаковых результатов вне зависимости от реализации. Поэтому допускается использование платформенных библиотек, которые могут возвращать несколько различающиеся результаты на разных платформах. Класс `StrictMath` обязан возвращать одинаковые результаты в любой реализации. Многие методы класса `Math` вызывают методы `StrictMath`. В JDK 9 в классы `Math` и `StrictMath` добавлены следующие методы:

- `long floorDiv(long x, int y)`
- `int floorMod(long x, int y)`
- `double fma(double x, double y, double z)`
- `float fma(float x, float y, float z)`
- `long multiplyExact(long x, int y)`
- `long multiplyFull(int x, int y)`
- `long multiplyHigh(long x, long y)`

Метод `floorDiv()` возвращает наибольшее значение типа `long`, меньшее или равное частному от деления `x` на `y`. Если знаки аргументов совпадают, то результат деления округляется в сторону нуля (режим усечения). Если же знаки различны, то результат округляется в сторону минус бесконечности. Если делимое равно `Long.MIN_VALUE`, а делитель `-1`, то метод возвращает `Long.MIN_VALUE`. Если делитель равен 0, то возбуждается исключение `ArithmeticException`.

Метод `floorMod()` возвращает «остаток» от деления, т. е. величину

```
x - (floorDiv(x, y) * y)
```

Знак `floorMod()` совпадает со знаком делителя `y`, а сама величина остатка находится в диапазоне

```
-abs(y) < r < +abs(y).
```

Методы `fma()` соответствуют операции `fusedMultiplyAdd` (умножение-сложение с однократным округлением), определенной в стандарте IEEE 754-2008. Они возвращают результат вычисления `(a * b + c)`, выполненного в предположении неограниченных диапазона и точности с последующим округлением до ближайшего значения типа `double` или `float`. Округление производится в режиме банковского округления (к ближайшему четному). Отметим, что метод `fma()` возвращает более точный результат, чем прямое вычисление выражения `(a * b + c)`, потому что в последнем случае имеют место две ошибки округления – при умножении и при сложении – тогда как в первом только одна.

Метод `multiplyExact()` возвращает произведение двух аргументов и возбуждает исключение `ArithmeticException`, если результат не помещается в значение типа `long`.

Метод `multiplyFull()` возвращает точное произведение двух аргументов в виде значения типа `long`.

Метод `multiplyHigh()` возвращает старшие 64 разряда 128-разрядного произведения двух 64-разрядных аргументов в виде значения типа `long`. В листинге 20.9 приведен полный код программы, демонстрирующей использование новых методов класса `StrictMath`.

Листинг 20.9. Класс StrictMathTest, демонстрирующий использование новых методов класса StrictMath

```
// StrictMathTest.java
package com.jdojo.misc;

import static java.lang.StrictMath.*;

public class StrictMathTest {
  public static void main(String[] args) {
    System.out.println("Метод StrictMath.floorDiv(long, int):");
    System.out.printf("floorDiv(20L, 3) = %d%n", floorDiv(20L, 3));
    System.out.printf("floorDiv(-20L, -3) = %d%n", floorDiv(-20L, -3));
    System.out.printf("floorDiv(-20L, 3) = %d%n", floorDiv(-20L, 3));
    System.out.printf("floorDiv(Long.Min_VALUE, -1) = %d%n", floorDiv(Long.MIN_VALUE, -1));

    System.out.println("\nМетод StrictMath.floorMod(long, int):");
    System.out.printf("floorMod(20L, 3) = %d%n", floorMod(20L, 3));
    System.out.printf("floorMod(-20L, -3) = %d%n", floorMod(-20L, -3));
    System.out.printf("floorMod(-20L, 3) = %d%n", floorMod(-20L, 3));

    System.out.println("\nМетод StrictMath.fma(double, double, double):");
    System.out.printf("fma(3.337, 6.397, 2.789) = %f%n", fma(3.337, 6.397, 2.789));

    System.out.println("\nМетод StrictMath.multiplyExact(long, int):");
    System.out.printf("multiplyExact(29087L, 7897979) = %d%n",
            multiplyExact(29087L, 7897979));
    try {
       System.out.printf("multiplyExact(Long.MAX_VALUE, 5) = %d%n",
       multiplyExact(Long.MAX_VALUE, 5));
    } catch (ArithmeticException e) {
       System.out.println("multiplyExact(Long.MAX_VALUE, 5) = " + e.getMessage());
    }

    System.out.println("\nМетод StrictMath.multiplyFull(int, int):");
    System.out.printf("multiplyFull(29087, 7897979) = %d%n", multiplyFull(29087, 7897979));

    System.out.println("\nМетод StrictMath.multiplyHigh(long, long):");
    System.out.printf("multiplyHigh(29087L, 7897979L) = %d%n",
            multiplyHigh(29087L, 7897979L));
    System.out.printf("multiplyHigh(Long.MAX_VALUE, 8) = %d%n",
            multiplyHigh(Long.MAX_VALUE, 8));
```

```
  }
}
```

```
Метод StrictMath.floorDiv(long, int):
floorDiv(20L, 3) = 6
floorDiv(-20L, -3) = 6
floorDiv(-20L, 3) = -7
floorDiv(Long.Min_VALUE, -1) = -9223372036854775808

Метод StrictMath.floorMod(long, int):
floorMod(20L, 3) = 2
floorMod(-20L, -3) = -2
floorMod(-20L, 3) = 1

Метод StrictMath.fma(double, double, double):
fma(3.337, 6.397, 2.789) = 24.135789

Метод StrictMath.multiplyExact(long, int):
multiplyExact(29087L, 7897979) = 229728515173
multiplyExact(Long.MAX_VALUE, 5) = long overflow

Метод StrictMath.multiplyFull(int, int):
multiplyFull(29087, 7897979) = 229728515173

Метод StrictMath.multiplyHigh(long, long):
multiplyHigh(29087L, 7897979L) = 0
multiplyHigh(Long.MAX_VALUE, 8) = 3
```

Изменения в классе ClassLoader

В JDK 9 в класс `java.lang.ClassLoader` добавлен один конструктор и несколько методов:

- `protected ClassLoader(String name, ClassLoader parent)`
- `public String getName()`
- `protected Class<?> findClass(String moduleName, String name)`
- `protected URL findResource(String moduleName, String name) throws IOException`
- `public Stream<URL> resources(String name)`
- `public final boolean isRegisteredAsParallelCapable()`
- `public final Module getUnnamedModule()`
- `public static ClassLoader getPlatformClassLoader()`
- `public final Package getDefinedPackage(String name)`
- `public final Package[] getDefinedPackages()`

Их имена говорят сами за себя. Я не стану обсуждать в этом разделе все методы, при желании описания можно найти в документации. Защищенные конструктор и методы предназначены для разработки новых загрузчиков классов.

У загрузчика классов может быть необязательное имя, возвращаемое методом `getName()`. Если загрузчик не имеет имени, то этот метод возвращает `null`. Среда выполнения Java включает имя загрузчика классов, если оно задано, в трассу стека и сообщения об исключениях. Это полезно для отладки.

Метод `resources()` возвращает поток URL-адресов всех найденных ресурсов с указанным именем.

С каждым загрузчиком классов ассоциирован безымянный модуль, который содержит все типы, найденные этим загрузчиком на пути к классам. Метод `getUnnamedModule()` возвращает ссылку на безымянный модуль загрузчика классов.

Статический метод `getPlatformClassLoader()` возвращает ссылку на платформенный загрузчик классов.

Новые методы в классе Optional<T>

В JDK 9 в класс `java.util.Optional<T>` добавлено три новых метода:

- `void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)`
- `Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)`
- `Stream<T> stream()`

Прежде чем описывать эти методы и предлагать демонстрационную программу, рассмотрим следующий список объектов типа `Optional<Integer>`:

```
List<Optional<Integer>> optionalList = List.of(Optional.of(1),
                                               Optional.empty(),
                                               Optional.of(2),
                                               Optional.empty(),
                                               Optional.of(3));
```

Список содержит пять элементов, два из которых пустые объекты `Optional`, а остальные три содержат значения 1, 2 и 3. Мы будем обращаться к этому списку по ходу обсуждения.

Метод `ifPresentOrElse()` предлагает два варианта действий. Если значение присутствует, то к нему применяется указанное действие, в противном случае выполняется действие `emptyAction`. Код ниже обходит все элементы списка с помощью потока и печатает значение, если `Optional` не пуст, и строку «Empty» в противном случае:

```
optionalList.stream()
            .forEach(p -> p.ifPresentOrElse(System.out::println,
                                            () -> System.out.println("Empty")));
```

```
1
Empty
2
Empty
3
```

Метод `or()` возвращает сам объект `Optional`, если тот содержит значение, а в противном случае – объект `Optional`, возвращенный указанным объектом `supplier`. В следующем фрагменте из списка объектов `Optional` создается поток и с помощью метода `or()` все пустые объекты отображаются на объект `Optional` со значением 0.

```
optionalList.stream()
            .map(p -> p.or(() -> Optional.of(0)))
            .forEach(System.out::println);
```

```
Optional[1]
Optional[0]
Optional[2]
Optional[0]
Optional[3]
```

Метод `stream()` возвращает последовательный поток, содержащий значение, присутствующее в объекте `Optional`. Если `Optional` пуст, то возвращается пустой поток. Предположим, что имеется список объектов `Optional`, и мы хотим собрать все присутствующие значения в другом списке. В Java 8 это можно было бы сделать так:

```
// list8 будет содержать числа 1, 2, 3
List<Integer> list8 = optionalList.stream()
                                  .filter(Optional::isPresent)
                                  .map(Optional::get)
                                  .collect(toList());
```

Нам пришлось воспользоваться методом `filter`, чтобы отфильтровать все пустые объекты `Optional`, а оставшиеся отобразить на их значения. Благодаря новому методу `stream()` мы можем объединить операции `filter()` и `map()` в одну операцию `flatMap()`:

```
// list9 будет содержать числа 1, 2, 3
List<Integer> list9 = optionalList.stream()
                                  .flatMap(Optional::stream)
                                  .collect(toList());
```

В листинге 20.10 приведен полный код программы для демонстрации этих методов.

Листинг 20.10. Использование новых методов класса Optional

```
// OptionalTest.java
package com.jdojo.misc;

import java.util.List;
import java.util.Optional;
```

```
import static java.util.stream.Collectors.toList;

public class OptionalTest {
   public static void main(String[] args) {
   // Создать список Optional<Integer>
   List<Optional<Integer>> optionalList = List.of(
            Optional.of(1),
            Optional.empty(),
            Optional.of(2),
            Optional.empty(),
            Optional.of(3));

   // Напечатать исходный список
   System.out.println("Исходный список: " + optionalList);

   // Использование метода ifPresentOrElse()
   optionalList.stream()
               .forEach(p -> p.ifPresentOrElse(System.out::println,
                                    () -> System.out.println("Empty")));

   // Использование метода or()
   optionalList.stream()
               .map(p -> p.or(() -> Optional.of(0)))
               .forEach(System.out::println);

   // В Java 8
   List<Integer> list8 = optionalList.stream()
                                     .filter(Optional::isPresent)
                                     .map(Optional::get)
                                     .collect(toList());
   System.out.println("Список в Java 8: " + list8);

   // В Java 9
   List<Integer> list9 = optionalList.stream()
                                     .flatMap(Optional::stream)
                                     .collect(toList());
   System.out.println("Список в Java 9: " + list9);
   }
}
```

```
Исходный список: [Optional[1], Optional.empty, Optional[2], Optional.empty, Optional[3]]
1
Empty
2
Empty
3
Optional[1]
```

```
Optional[0]
Optional[2]
Optional[0]
Optional[3]
Список в Java 8: [1, 2, 3]
Список в Java 9: [1, 2, 3]
```

Новые методы класса CompletableFuture<T>

В класс `CompletableFuture<T>` из пакета `java.util.concurrent` в JDK 9 добавлены следующие методы:

- `<U> CompletableFuture<U> newIncompleteFuture()`
- `Executor defaultExecutor()`
- `CompletableFuture<T> copy()`
- `CompletionStage<T> minimalCompletionStage()`
- `CompletableFuture<T> completeAsync(Supplier<? extends T> supplier, Executor executor)`
- `CompletableFuture<T> completeAsync(Supplier<? extends T> supplier)`
- `CompletableFuture<T> orTimeout(long timeout, TimeUnit unit)`
- `CompletableFuture<T> completeOnTimeout(T value, long timeout, TimeUnit unit)`
- `static Executor delayedExecutor(long delay, TimeUnit unit, Executor executor)`
- `static Executor delayedExecutor(long delay, TimeUnit unit)`
- `static <U> CompletionStage<U> completedStage(U value)`
- `static <U> CompletableFuture<U> failedFuture(Throwable ex)`
- `static <U> CompletionStage<U> failedStage(Throwable ex)`

За дополнительной информацией о них отсылаю читателя к документации.

Уведомления об активном ожидании

В многопоточной программе потокам часто необходимо координировать свою работу. Иногда одному потоку приходится ждать, когда другой изменит некоторую волатильную (объявленную с модификатором `volatile`) переменную. После того как переменная изменена, первый поток может продолжить работу. Если ожидание затягивается, то первому потоку лучше освободить процессор, заснув или перейдя в режим ожидания, – его уведомят о том, когда можно возобновить работу. Однако с засыпанием и ожиданием связана задержка. Если ждать нужно недолго, то для уменьшения задержки поток крутится в цикле, проверяя выполнение некоторого условия. Рассмотрим класс, который в цикле ожидает, когда волатильная переменная `dataReady` станет равна `true`:

```
volatile boolean dataReady;
...

@Override
```

```
public void run() {
   // Ждать готовности данных
   while (!dataReady) {
      // Никакого кода нет
   }
   processData();
}

private void processData() {
   // Здесь обрабатываются данные
}
```

Цикл `while` в этом коде называется спин-циклом, или активным ожиданием. Программа крутится в этом цикле, пока переменная `dataReady` не станет равна `true`.

Вообще говоря, активное ожидание не поощряется, поскольку на него впустую растрачиваются ресурсы, но часто оно бывает необходимо. В этом примере преимущество состоит в том, что поток продолжит обработку сразу, как только переменная `dataReady` примет значение `true`. Но за это приходится расплачиваться потреблением энергии, потому что поток активно использует ресурсы процессора.

Некоторые процессоры можно уведомить о том, что поток находится в режиме активного ожидания, чтобы они по возможности оптимизировали потребление ресурсов. Например, процессоры семейства x86 поддерживают для этой цели команду `PAUSE`, которая откладывает выполнение следующей команды в данном потоке на короткое время.

В JDK 9 в класс `Thread` добавлен статический метод `onSpinWait()`. Он просто уведомляет процессор о том, что вызывающий поток в данный момент не может продолжать работу, так что потребление ресурсов можно оптимизировать. Если оборудование не поддерживает такие уведомления, то реализация метода может быть пустой.

В листинге 20.11 приведен пример кода. Отметим, что использование уведомления об активном ожидании не изменяет семантику программы. Но при наличии аппаратной поддержки программа, возможно, будет работать быстрее.

Листинг 20.11. Пример уведомления процессора об активном ожидании с помощью статического метода `Thread.onSpinWait()`

```
// SpinWaitTest.java
package com.jdojo.misc;

public class SpinWaitTest implements Runnable {
   private volatile boolean dataReady = false;

   @Override
   public void run() {
      // Ждать готовности данных
      while (!dataReady) {
```

```
            // Уведомить об активном ожидании
            Thread.onSpinWait();
        }
        processData();
    }

    private void processData() {
        // Здесь обрабатываются данные
    }

    public void setDataReady(boolean dataReady) {
        this.dataReady = dataReady;
    }
}
```

Улучшения в Time API

В JDK 9 добавлено много новых методов в классы и интерфейсы, составляющие Time API. Подробно рассказать обо всех изменениях невозможно. Я приведу список новых методов с краткими описаниями. Сложные методы сопровождаются примерами. Time API состоит из пакетов `java.time.*`, находящихся в модуле `java.base`.

В этом разделе приводятся только небольшие фрагменты кода. Законченную программу можно найти в классе `com.jdojo.misc.TimeApiTest`, входящем в модуль `com.jdojo.misc` прилагаемого к книге исходного кода. Во многих примерах печатается текущая дата и время, поэтому на вашей машине результаты будут иными.

Класс Clock

В класс `Clock` добавлен следующий метод:

```
static Clock tickMillis(ZoneId zone)
```

Метод `tickMillis()` возвращает текущий отсчет времени в миллисекундах. Используется лучший из доступных в системе источников времени. Тактовый генератор дает разрешение выше миллисекундного. Вызов этого метода эквивалентен такому выражению:

```
Clock.tick(Clock.system(zone), Duration.ofMillis(1))
```

Класс Duration

Новые методы класса `Duration` можно отнести к трем категориям:

- деление одного промежутка времени на другой;
- получение промежутка времени в терминах заданной единицы времени и получение части промежутка времени, например, количества дней, часов, секунд и т. д.;
- усечение промежутка времени до указанной единицы времени.

Я буду использовать промежуток времени длительностью 23 дня 3 часа 45 минут и 30 секунд. В следующем фрагменте создается такой объект `Duration` и ссылка на него запоминается в переменной `compTime`:

```
// Создать промежуток времени 23 дня 3 часа 45 минут и 30 секунд
Duration compTime = Duration.ofDays(23)
                            .plusHours(3)
                            .plusMinutes(45)
                            .plusSeconds(30);
System.out.println("Продолжительность: " + compTime);
```

```
Продолжительность: PT555H45M30S
```

После перевода дней в часы путем умножения на 24 оказывается, что этот промежуток составляет 555 часов, 45 минут и 30 секунд.

Деление одного промежутка времени на другой

В этой категории всего один метод:

```
long dividedBy(Duration divisor)
```

Метод `dividedBy()` позволяет разделить один промежуток времени на другой. Он возвращает число, показывающее, сколько раз промежуток `divisor` помещается в промежутке, от имени которого вызван метод. Чтобы узнать, сколько в промежутке целых недель, мы вызываем метод `dividedBy()`, указав в качестве `divisor` промежуток длительностью 7 дней. Ниже показано, как вычислить число целых дней, недель и часов в промежутке времени:

```
long wholeDays = compTime.dividedBy(Duration.ofDays(1));
long wholeWeeks = compTime.dividedBy(Duration.ofDays(7));
long wholeHours = compTime.dividedBy(Duration.ofHours(7));
System.out.println("Число целых дней: " + wholeDays);
System.out.println("Число целых недель: " + wholeWeeks);
System.out.println("Число целых часов: " + wholeHours);
```

```
Число целых дней: 23
Число целых недель: 3
Число целых часов: 79
```

Преобразование промежутка времени и выделение его частей

В класс `Duration` добавлено несколько методов из этой категории:

- `long toDaysPart()`
- `int toHoursPart()`

- `int toMillisPart()`
- `int toMinutesPart()`
- `int toNanosPart()`
- `long toSeconds()`
- `int toSecondsPart()`

В классе `Duration` есть два набора методов: `toXxx()` и `toXxxPart()`, где `Xxx` может принимать значения `Days`, `Hours`, `Minutes`, `Seconds`, `Millis` и `Nanos`. Обратите внимание, что в приведенном выше перечне метод `toDaysPart()` есть, а метода `toDays()` нет. В этом и подобных случаях соответствующие методы уже существовали в JDK 8. В частности, это относится к методу `toDays()`.

Методы `toXxx()` преобразуют промежуток времени в единицу `Xxx` и возвращают целую часть. Методы `toXxxPart()` разбивают промежуток времени на части – `days:hours:minutes:seconds:milils:nanos` – и возвращают часть `Xxx`. В нашем примере `toDays()` преобразует промежуток в дни и возвращает целую часть – 23. А `toDaysPart()` представляет промежуток в виде `23Days:3Hours:45Minutes:30Seconds:0Millis:0Nanos` и возвращает первую часть – 23. Те же правила применимы к методам `toHours()` и `toHoursPart()`. Метод `toHours()` преобразует промежуток времени в часы и возвращает целое количество часов – 555. Метод `toHoursPart()` представляет промежуток в том же виде, что метод `toDaysPart()` и возвращает часть, относящуюся к часам, – 3. В следующем фрагменте приведено несколько примеров:

```
System.out.println("toDays(): " + compTime.toDays());
System.out.println("toDaysPart(): " + compTime.toDaysPart());
System.out.println("toHours(): " + compTime.toHours());
System.out.println("toHoursPart(): " + compTime.toHoursPart());
System.out.println("toMinutes(): " + compTime.toMinutes());
System.out.println("toMinutesPart(): " + compTime.toMinutesPart());
```

```
toDays(): 23
toDaysPart(): 23
toHours(): 555
toHoursPart(): 3
toMinutes(): 33345
toMinutesPart(): 45
```

Усечение промежутка времени

В класс `Duration` добавлен только один метод из этой категории:

```
Duration truncatedTo(TemporalUnit unit)
```

Метод `truncatedTo()` возвращает копию промежутка времени после отбрасывания частей, соответствующих более мелким единицам времени, чем `unit`. Допускаются только единицы времени не крупнее `DAYS`. Попытка задать более крупную единицу времени, например `WEEKS` или `YEARS`, приводит к исключению.

Совет. В JDK 8 метод `truncatedTo(TemporalUnit unit)` уже существовал в классах `LocalTime` и `Instant`.

Ниже показано, как используется этот метод:

```
System.out.println("Усечено до DAYS: " + compTime.truncatedTo(ChronoUnit.DAYS));
System.out.println("Усечено до HOURS: " + compTime.truncatedTo(ChronoUnit.HOURS));
System.out.println("Усечено до MINUTES: " + compTime.truncatedTo(ChronoUnit.MINUTES));
```

```
Усечено до DAYS: PT552H
Усечено до HOURS: PT555H
Усечено до MINUTES: PT555H45M
```

Рассматривается промежуток времени `23Days:3Hours:45Minutes:30Seconds:0Millis:0Nanos`. Если усечь его до `DAYS`, то все части мельче дней отбрасываются, и метод возвращает 23 дня, или 552 часа, что мы и видим. Если усечь до `HOURS`, то отбрасываются все части мельче часов и возвращается 555 часов. Усечение до `MINUTES` отбрасывает все части мельче минуты, в частности секунды и миллисекунды.

Фабричный метод ofInstant()

При проектировании API основное внимание уделялось эффективности и удобству разработчика. Существует несколько часто встречающихся случаев, когда для преобразования между типами даты и времени программисту приходилось вызывать больше методов, чем необходимо. Вот два примера:

- преобразование `java.util.Date` в `LocalDate`;
- преобразование `Instant` в `LocalDate` и `LocalTime`.

В JDK 9 в классы `LocalDate` и `LocalTime` добавлен статический фабричный метод `ofInstant(Instant instant, ZoneId zone)` для упрощения этих преобразований. В JDK 8 этот метод уже присутствовал в классах `ZonedDateTime`, `OffsetDateTime`, `LocalDateTime` и `OffsetTime`. В следующем фрагменте показаны оба способа (в JDK 8 и JDK 9) преобразования `java.util.Date` в `LocalDate`:

```
// В JDK 8
Date dt = new Date();
LocalDate ld= dt.toInstant()
                .atZone(ZoneId.systemDefault())
                .toLocalDate();
System.out.println("Текущая местная дата: " + ld);

// В JDK 9
LocalDate ld2 = LocalDate.ofInstant(dt.toInstant(), ZoneId.systemDefault());
System.out.println("Текущая местная дата: " + ld2);
```

```
Текущая местная дата: 2017-02-11
```

```
Текущая местная дата: 2017-02-11
```

В следующем фрагменте показаны оба способа (в JDK 8 и JDK 9) преобразования `Instant` в `LocalDate` и `LocalTime`:

```
// В JDK 8
Instant now = Instant.now();
ZoneId zone = ZoneId.systemDefault();
ZonedDateTime zdt = now.atZone(zone);
LocalDate ld3 = zdt.toLocalDate();
LocalTime lt3 = zdt.toLocalTime();
System.out.println("Местная дата: " + ld3 + ", местное время:" + lt3);

// В JDK 9
LocalDate ld4 = LocalDate.ofInstant(now, zone);
LocalTime lt4 = LocalTime.ofInstant(now, zone);
System.out.println("Местная дата: " + ld4 + ", местное время:" + lt4);
```

```
Местная дата: 2017-02-11, местное время:22:13:31.919339400
Местная дата: 2017-02-11, местное время:22:13:31.919339400
```

Получение числа секунд от начала отсчета

Иногда требуется получить число секунд между началом отсчета – моментом времени 1970-01-01T00:00:00Z – и моментом, заданным в виде `LocalDate`, `LocalTime` или `OffsetTime`. В JDK 8 в классе `OffsetDateTime` существовал метод `toEpochSecond()` специально для этой цели. Чтобы получить число секунд от начала отсчета до момента, представленного объектом `ZonedDateTime`, нужно было преобразовать этот объект к типу `OffsetDateTime` методом `toOffsetDateTime()`, а затем вызвать метод `toEpochSecond()`. В JDK 8 не было метода `toEpochSecond()` в классах `LocalDate`, `LocalTime` и `OffsetTime`. В JDK 9 эти методы добавлены:

- `LocalDate.toEpochSecond(LocalTime time, ZoneOffset offset)`
- `LocalTime.toEpochSecond(LocalDate date, ZoneOffset offset)`
- `OffsetTime.toEpochSecond(LocalDate date)`

Почему сигнатуры метода `toEpochSecond()` различны? Чтобы получить число секунд от начала отсчета 1970-01-01T00:00:00Z, нужно определить какой-то другой момент времени. Момент можно составить из трех частей: дата, время, часовой пояс. В классах `LocalDate` и `LocalTime` присутствует только одна из этих трех частей, а в классе `OffsetTime` две части: время и часовой пояс. Недостающие части следует задать в виде аргументов метода. Так что аргументами метода `toEpochSecond()` являются части, которых не хватает для определения момента времени. В следующем фрагменте с помощью трех классов вычисляется число секунд между одним и тем же моментом времени и началом отсчета:

```
LocalDate ld = LocalDate.of(2017, 2, 12);
LocalTime lt = LocalTime.of(9, 15, 45);
ZoneOffset offset = ZoneOffset.ofHours(6);
OffsetTime ot = OffsetTime.of(lt, offset);
long s1 = ld.toEpochSecond(lt, offset);
long s2 = lt.toEpochSecond(ld, offset);
long s3 = ot.toEpochSecond(ld);
System.out.println("LocalDate.toEpochSecond(): " + s1);
System.out.println("LocalTime.toEpochSecond(): " + s2);
System.out.println("OffsetTime.toEpochSecond(): " + s3);
```

```
LocalDate.toEpochSecond(): 1486869345
LocalTime.toEpochSecond(): 1486869345
OffsetTime.toEpochSecond(): 1486869345
```

Поток объектов LocalDate

В JDK 9 упрощен перебор дат между двумя заданными – по одному дню или по одному периоду. В класс `LocalDate` добавлено два метода:

- `Stream<LocalDate> datesUntil(LocalDate endExclusive)`
- `Stream<LocalDate> datesUntil(LocalDate endExclusive, Period step)`

Они порождают последовательный упорядоченный поток объектов `LocalDate`. Первый элемент потока – объект, от имени которого вызван метод. Метод `datesUntil(LocalDate endExclusive)` увеличивает дату следующего объекта на один день, а метод `datesUntil(LocalDate endExclusive, Period step)` – на указанную величину `step`. Конечная дата не включается. С возвращенным потоком можно выполнить несколько полезных вычислений. В следующем фрагменте подсчитывается число воскресений в 2017 году. Отметим, что в качестве последней, не включаемой, даты указано 1 января 2018 года, так что поток возвращает все даты в 2017 году.

```
long sundaysIn2017 = LocalDate.of(2017, 1, 1)
                              .datesUntil(LocalDate.of(2018, 1, 1))
                              .filter(ld -> ld.getDayOfWeek() == DayOfWeek.SUNDAY)
                              .count();
System.out.println("Число воскресений в 2017 году: " + sundaysIn2017);
```

```
Число воскресений в 2017 году: 53
```

В следующем фрагменте печатаются все пятницы, выпадающие на 13 число, между 1 января 2017 (включая) и 1 января 2022 (не включая):

```
System.out.println("Пятницы, выпадающие на 13 число, в период с 2017 по 2021 год: ");
LocalDate.of(2017, 1, 1)
```

```
        .datesUntil(LocalDate.of(2022, 1, 1))
        .filter(ld -> ld.getDayOfMonth() == 13 && ld.getDayOfWeek() == DayOfWeek.FRIDAY)
        .forEach(System.out::println);
```

```
Пятницы, выпадающие на 13 число, в период с 2017 по 2021 год:
2017-01-13
2017-10-13
2018-04-13
2018-07-13
2019-09-13
2019-12-13
2020-03-13
2020-11-13
2021-08-13
```

В следующем фрагменте печатается последний день каждого месяца в 2017 году:

```
System.out.println("Последние дни месяцев в 2017 году:");
LocalDate.of(2017, 1, 31)
        .datesUntil(LocalDate.of(2018, 1, 1), Period.ofMonths(1))
        .map(ld -> ld.format(DateTimeFormatter.ofPattern("EEE MMM dd, yyyy")))
        .forEach(System.out::println);
```

```
Последние дни месяцев в 2017 году:
Tue Jan 31, 2017
Tue Feb 28, 2017
Fri Mar 31, 2017
Sun Apr 30, 2017
Wed May 31, 2017
Fri Jun 30, 2017
Mon Jul 31, 2017
Thu Aug 31, 2017
Sat Sep 30, 2017
Tue Oct 31, 2017
Thu Nov 30, 2017
Sun Dec 31, 2017
```

Новые параметры форматирования

В JDK 9 добавлено несколько параметров форматирования в Time API. В следующих разделах они описываются подробно.

Модифицированная юлианская дата

В форматной строке даты можно использовать спецификатор g, который соответствует модифицированной юлианской дате. Этот спецификатор может повторяться, например ggg, и тогда результат будет дополнен нулями, если число цифр в нем меньше количества повторений g. На странице по адресу http://www.unicode.org/reports/tr35/tr35-41/tr35-dates.html#Date_Format_Patterns семантика спецификатора g описана следующим образом:

> *Модифицированная юлианская дата отличается от традиционной юлианской даты в двух отношениях. Во-первых, сутки начинаются с полуночи по местному, а не по гринвичскому времени. Во-вторых, это число зависит от часового пояса. Можно считать, что одно это число включает в себя все компоненты даты.*

Совет. Заглавной букве G в JDK 8 уже приписана семантика форматирования даты, она преобразуется в эру: AD, Anno Domini, A, BC, Before Christ или B.

В следующем фрагменте демонстрируется использование модифицированной юлианской даты:

```
ZonedDateTime zdt = ZonedDateTime.now();
System.out.println("Текущая ZonedDateTime: " + zdt);
System.out.println("Модифицированная юлианская дата (g): " +
zdt.format(DateTimeFormatter.ofPattern("g")));
System.out.println("Модифицированная юлианская дата (ggg): " +
zdt.format(DateTimeFormatter.ofPattern("ggg")));
System.out.println("Модифицированная юлианская дата (gggggg): " +
zdt.format(DateTimeFormatter.ofPattern("gggggg")));
```

```
Текущая ZonedDateTime: 2017-02-12T11:49:03.364431100-06:00[America/Chicago]
Модифицированная юлианская дата (g): 57796
Модифицированная юлианская дата (ggg): 57796
Модифицированная юлианская дата (gggggg): 057796
```

Обобщенные названия часовых поясов

В JDK 8 две буквы, V и z, были зарезервированы для форматирования часового пояса. Буква V выводила часовой пояс в формате "America/Los_Angeles; Z; -08:30", а буква z – в формате Central Standard Time и CST.

В JDK 9 добавлен спецификатор формата v, который выводит обобщенное название часового пояса без сдвига от UTC, например, Central Time или CT. Он соответствует времени по настенным часам. Так, сдвиг времени 8am Central Time от UTC составляет UTC-06 1 марта 2017 и UTC-05 19 марта 2017. Существует два варианта формата – v и vvvv – для вывода обобщенного названия часового пояса в короткой (например, CT) и длинной (например, Central Time) форме. В следующем фрагменте демонстрируется различие результатов форматирования со спецификаторами V, z и v:

```
ZonedDateTime zdt = ZonedDateTime.now();
System.out.println("Текущая ZonedDateTime: " + zdt);
System.out.println("Формат VV: " +
zdt.format(DateTimeFormatter.ofPattern("MM/dd/yyyy HH:mm VV")));
System.out.println("Формат z: " +
zdt.format(DateTimeFormatter.ofPattern("MM/dd/yyyy HH:mm z")));
System.out.println("Формат zzzz: " +
zdt.format(DateTimeFormatter.ofPattern("MM/dd/yyyy HH:mm zzzz")));
System.out.println("Формат v: " +
zdt.format(DateTimeFormatter.ofPattern("MM/dd/yyyy HH:mm v")));
System.out.println("Формат vvvv: " +
zdt.format(DateTimeFormatter.ofPattern("MM/dd/yyyy HH:mm vvvv")));
```

```
Текущая ZonedDateTime: 2017-02-12T12:30:08.975373900-06:00[America/Chicago]
Формат VV: 02/12/2017 12:30 America/Chicago
Формат z: 02/12/2017 12:30 CST
Формат zzzz: 02/12/2017 12:30 Central Standard Time
Формат v: 02/12/2017 12:30 CT
Формат vvvv: 02/12/2017 12:30 Central Time
```

Класс Scanner и потоковые операции

В JDK 9 в класс `java.util.Scanner` добавлено три метода, возвращающих значение типа `Stream`:

- `Stream<MatchResult> findAll(String patternString)`
- `Stream<MatchResult> findAll(Pattern pattern)`
- `Stream<String> tokens()`

Метод `findAll()` возвращает поток, содержащий все совпадения с образцом. Вызов `findAll(patternString)` эквивалентен вызову `findAll(Pattern.compile(patternString))`. Метод `tokens()` возвращает поток выделенных сканнером лексем, разделенных текущим разделителем. В листинге 20.12 приведена программа, демонстрирующая разбиение строки на слова с помощью метода `findAll()`.

Листинг 20.12. Класс ScannerTest, печатающий составляющие строку слова

```
// ScannerTest.java
package com.jdojo.misc;

import java.util.List;
import java.util.Scanner;
import java.util.regex.MatchResult;
import static java.util.stream.Collectors.toList;

public class ScannerTest {
```

```
   public static void main(String[] args) {
      String patternString = "\\b\\w+\\b";
      String input = "A test string,\n which contains a new line.";
      List<String> words = new Scanner(input)
            .findAll(patternString)
            .map(MatchResult::group)
            .collect(toList());

      System.out.println("Входная строка: " + input);
      System.out.println("Слова: " + words);
   }
}
```

```
Входная строка: A test string,
 which contains a new line.
Слова: [A, test, string, which, contains, a, new, line]
```

Улучшения в классе Matcher

В класс `java.util.regex.Matcher` добавлено несколько новых методов:

- `Matcher appendReplacement(StringBuilder sb, String replacement)`
- `StringBuilder appendTail(StringBuilder sb)`
- `String replaceAll(Function<MatchResult,String> replacer)`
- `String replaceFirst(Function<MatchResult,String> replacer)`
- `Stream<MatchResult> results()`

В JDK 8 в класс `Matcher` уже входили первые четыре из этих методов. Но в JDK 9 они стали перегруженными. Методы `appendReplacement()` и `appendTail()` раньше работали с объектом `StringBuffer`, а теперь – также с объектом `StringBuilder`. Методы `replaceAll()` и `replaceFirst()` раньше принимали в качестве аргумента `String`, а в JDK 9 – еще и `Function<T,R>`.

Метод `results()` возвращает результаты сопоставления в виде потока элементов типа `MatchResult`. Для получения результата в виде строки следует опросить `MatchResult`. В JDK 8 тоже можно было получить результаты работы `Matcher` в виде потока, но логика была не вполне очевидной. Метод `results()` не переводит объект `Matcher` в исходное состояние. Чтобы использовать сопоставитель повторно, нужно самостоятельно вызвать метод `reset()` для установки в нужную позицию. В листинге 20.13 показано несколько интересных применений этого метода.

Листинг 20.13. Класс MatcherTest, демонстрирующий применение метода results()

```
// MatcherTest.java
package com.jdojo.misc;

import java.util.List;
```

```
import java.util.Set;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import static java.util.stream.Collectors.toList;
import static java.util.stream.Collectors.toSet;

public class MatcherTest {
   public static void main(String[] args) {
      // Регулярное выражение для поиска 7- или 10-значных телефонных номеров
      String regex = "\\b(\\d{3})?(\\d{3})(\\d{4})\\b";

      // Входная строка
      String input = "1, 3342229999, 2330001, 6159996666, 123, 3340909090";

      // Создать сопоставитель
      Matcher matcher = Pattern.compile(regex)
                               .matcher(input);

      // Собрать отформатированные телефонные номера в список
      List<String> phones = matcher.results()
              .map(mr -> (mr.group(1) == null ? "" : "(" + mr.group(1) + ") ")
                         + mr.group(2) + "-" + mr.group(3))
              .collect(toList());
      System.out.println("Телефоны: " + phones);

      // Сбросить сопоставитель для повторного использования
      matcher.reset();

      // Получить различные коды регионов
      Set<String> areaCodes = matcher.results()
                                     .filter(mr -> mr.group(1) != null)
                                     .map(mr -> mr.group(1))
                                     .collect(toSet());
      System.out.println("Различные коды регионов: " + areaCodes);
   }
}
```

```
Телефоны: [(334) 222-9999, 233-0001, (615) 999-6666, (334) 090-9090]
Различные коды регионов: [334, 615]
```

В методе `main()` объявлены две локальные переменные, `regex` и `input`. В переменной `regex` хранится регулярное выражение для сопоставления с 7- или 10-значными телефонными номерами. Его можно использовать для поиска телефонов во входной строке. В переменной `input` хранится текст, в котором встречаются телефонные номера.

```
// Регулярное выражение для поиска 7- или 10-значных телефонных номеров
String regex = "\\b(\\d{3})?(\\d{3})(\\d{4})\\b";

// Входная строка
String input = "1, 3342229999, 2330001, 6159996666, 123, 3340909090";
```

Затем мы компилируем регулярное выражение и получаем объект-сопоставитель:

```
Matcher matcher = Pattern.compile(regex)
                         .matcher(input);
```

Мы хотим представить 10-значный номер в формате `(nnn) nnn-nnnn`, а 7-значный – в формате `nnn-nnnn`, а затем собрать все отформатированные телефоны в список `List<String>`. Это делается так:

```
List<String> phones = matcher.results()
                             .map(mr -> (mr.group(1) == null ? "" : "(" + mr.group(1) + ") ")
                                  + mr.group(2) + "-" + mr.group(3))
                             .collect(toList());
```

Обратите внимание, что метод `map()` принимает объект `MatchResult` и возвращает отформатированный номер телефона в виде строки. Если был найден 7-значный номер, то группа 1 будет равна `null`. Далее мы хотим произвести поиск уникальных кодов регионов в 10-значных номерах. Для этого мы предварительно должны сбросить сопоставитель, чтобы поиск начинался с первого символа входной строки.

```
matcher.reset();
```

Первая группа в `MatchResult` содержит код региона. Мы должны отфильтровать 7-значные номера и собрать значения первой группы в множество `Set<String>` уникальных кодов регионов. Вот как это делается:

```
Set<String> areaCodes = matcher.results()
                               .filter(mr -> mr.group(1) != null)
                               .map(mr -> mr.group(1))
                               .collect(toSet());
```

Улучшения в классе Objects

Класс `java.util.Objects` содержит статические методы, применяемые к объектам. Обычно они используются для проверки аргументов метода, например, на совпадение с `null`. В JDK 9 в этот класс добавлены следующие статические методы:

- `<T> T requireNonNullElse(T obj, T defaultObj)`
- `<T> T requireNonNullElseGet(T obj, Supplier<? extends T> supplier)`
- `int checkFromIndexSize(int fromIndex, int size, int length)`
- `int checkFromToIndex(int fromIndex, int toIndex, int length)`
- `int checkIndex(int index, int length)`

В JDK 8 уже было три варианта метода `requireNonNull()`. Он проверяет, что значение не равно `null`, и, если это не так, возбуждает исключение `NullPointerException`. В JDK 9 добавлено еще два варианта.

Метод `requireNonNullElse(T obj, T defaultObj)` возвращает `obj`, если `obj` не равно `null`, и `defaultObj` – если `obj` равно `null`, а `defaultObj` не равно `null`. Если же и `obj`, и `defaultObj` равны `null`, то возбуждается исключение `NullPointerException`.

Метод `requireNonNullElseGet(T obj, Supplier<? extends T> supplier)` работает так же, как `requireNonNullElse(T obj, T defaultObj)`, только для получения значения по умолчанию использует объект-поставщик `supplier`. Метод возвращает `obj`, если `obj` не равен `null`. Если `supplier` не равен `null` и возвращает значение, отличное от `null`, то это значение и возвращается. В противном случае возбуждается исключение `NullPointerException`.

Методы вида `checkXxx()` проверяют, входит ли индекс или поддиапазон в указанный диапазон. Они полезны при работе с массивами и коллекциями, когда доступ к ним производится по индексу или поддиапазону. Методы возбуждают исключение `IndexOutOfBoundsException`, если индекс или поддиапазон находится вне диапазона.

Метод `checkFromIndexSize(int fromIndex, int size, int length)` проверяет, находится ли поддиапазон с границами `fromIndex` (включается) и `fromIndex + size` (не включается) внутри диапазона от 0 (включается) до `length` (не включается). Если какой-то аргумент отрицателен или поддиапазон выходит за границы диапазона, то возбуждается исключение `IndexOutOfBoundsException`. Если поддиапазон находится целиком внутри диапазона, то метод возвращает значение `fromIndex`. Предположим, что в нашей программе имеется метод, который принимает индекс и размер и возвращает поддиапазон массива или списка. Тогда с помощью описываемого метода мы сможем проверить, принадлежит ли запрашиваемый поддиапазон массиву или списку.

Метод `checkFromToIndex(int fromIndex, int toIndex, int length)` проверяет, находится ли поддиапазон с границами `fromIndex` (включается) и `toIndex` (не включается) внутри диапазона от 0 (включается) до `length` (не включается). Если какой-то аргумент отрицателен или поддиапазон выходит за границы диапазона, то возбуждается исключение `IndexOutOfBoundsException`. Если поддиапазон находится целиком внутри диапазона, то метод возвращает значение `fromIndex`.

Метод `checkIndex(int index, int length)` проверяет, принадлежит ли указанный индекс диапазону от 0 до `length` (не включая). Если какой-то аргумент отрицателен или индекс находится вне диапазона, то возбуждается исключение `IndexOutOfBoundsException`. Если индекс находится целиком внутри диапазона, то метод возвращает значение `index`. Это полезно, когда некий метод принимает индекс и возвращает значение в позиции массива или списка с таким индексом.

Сравнение массивов

Класс `java.util.Arrays` содержит только статические методы для выполнения различных операций с массивами, например: сортировка, сравнение, преобразование в поток и т. д. В JDK 9 в него добавлено несколько методов для сравнения массивов и их срезок. Новые методы можно отнести к трем категориям:

- сравнение двух массивов или их срезок на равенство;
- лексикографическое сравнение двух массивов;
- нахождение индекса первого расхождения двух массивов.

Список добавленных в класс методов очень велик. Методы из каждой категории перегружены для всех примитивных типов и типа `Object`. Для экономии места я не стану их перечислять, а отошлю вас к документации по классу `Arrays`. В конце раздела приведен пример для двух массивов чисел типа `int`.

Метод `equals()` сравнивает два массива на равенство. Два массива считаются равными, если число элементов одинаково и все соответственные элементы равны. Для типа `int` имеется два варианта метода `equals()`:

- `boolean equals(int[] a, int[] b)`
- `boolean equals(int[] a, int aFromIndex, int aToIndex, int[] b, int bFromIndex, int bToIndex)`

Первый вариант, сравнивающий два массива на равенство, существовал и раньше. Второй вариант, сравнивающий на равенство срезки двух массивов, добавлен в JDK 9. Аргументы `fromIndex` (включается) и `toIndex` (не включается) определяют диапазоны элементов в сравниваемых массивах. Метод возвращает `true`, если массивы равны и `false` в противном случае. Два массива, равные `null`, считаются равными.

В JDK 9 добавлено несколько вариантов методов `compare()` и `compareUnsigned()`. Оба метода производят лексикографическое сравнение элементов в массивах или срезках массивов. В методе `compareUnsigned()` целые числа рассматриваются как числа без знака. Считается, что массив `null` лексикографически меньше массива, отличного от `null`. Два массива, равные `null`, равны.

Ниже приведено два варианта метода `compare()` для `int`:

- `int compare(int[] a, int[] b)`
- `int compare(int[] a, int aFromIndex, int aToIndex, int[] b, int bFromIndex, int bToIndex)`

Метод `compare()` возвращает 0, если длины массивов (или срезок) равны и массивы содержат одни и те же элементы в одном и том же порядке. Метод возвращает отрицательное значение, если первый массив (или срезка) лексикографически меньше второго массива (или срезки), и положительное значение, если первый массив (или срезка) лексикографически больше второго.

Метод `mismatch()` сравнивает два массива или срезки. Вот два его варианта для типа `int`:

- `int mismatch(int[] a, int[] b)`
- `int mismatch (int[] a, int aFromIndex, int aToIndex, int[] b, int bFromIndex, int bToIndex)`

Метод `mismatch()` возвращает индекс первого несовпадения. Если несовпадений нет, возвращается –1. Если хотя бы один массив равен `null`, возбуждается исключение `NullPointerException`. В листинге 20.14 приведен полный код программы сравнения двух массивов типа `int[]` и их срезок.

Листинг 20.14. Сравнение массивов и срезок массивов методами класса Arrays

```
// ArrayComparision.java
package com.jdojo.misc;

import java.util.Arrays;

public class ArrayComparison {
   public static void main(String[] args) {
      int[] a1 = {1, 2, 3, 4, 5};
      int[] a2 = {1, 2, 7, 4, 5};
      int[] a3 = {1, 2, 3, 4, 5};

      // Напечатать исходные массивы
      System.out.println("Три массива:");
      System.out.println("a1: " + Arrays.toString(a1));
      System.out.println("a2: " + Arrays.toString(a2));
      System.out.println("a3: " + Arrays.toString(a3));

      // Сравнить массивы на равенство
      System.out.println("\nСравнение массивов методом equals():");
      System.out.println("Arrays.equals(a1, a2): " + Arrays.equals(a1, a2));
      System.out.println("Arrays.equals(a1, a3): " + Arrays.equals(a1, a3));
      System.out.println("Arrays.equals(a1, 0, 2, a2, 0, 2): " +
      Arrays.equals(a1, 0, 2, a2, 0, 2));

      // Сравнить массивы лексикографически
      System.out.println("\nСравнение массивов методом compare():");
      System.out.println("Arrays.compare(a1, a2): " + Arrays.compare(a1, a2));
      System.out.println("Arrays.compare(a2, a1): " + Arrays.compare(a2, a1));
      System.out.println("Arrays.compare(a1, a3): " + Arrays.compare(a1, a3));
      System.out.println("Arrays.compare(a1, 0, 2, a2, 0, 2): " +
      Arrays.compare(a1, 0, 2, a2, 0, 2));

      // Найти индекс первого несовпадения
      System.out.println("\nПоиск несовпадения методом mismatch():");
      System.out.println("Arrays.mismatch(a1, a2): " + Arrays.mismatch(a1, a2));
      System.out.println("Arrays.mismatch(a1, a3): " + Arrays.mismatch(a1, a3));
      System.out.println("Arrays.mismatch(a1, 0, 5, a2, 0, 1): " +
      Arrays.mismatch(a1, 0, 5, a2, 0, 1));
   }
}
```

```
Три массива:
a1: [1, 2, 3, 4, 5]
a2: [1, 2, 7, 4, 5]
```

```
a3: [1, 2, 3, 4, 5]

Сравнение массивов методом equals():
Arrays.equals(a1, a2): false
Arrays.equals(a1, a3): true
Arrays.equals(a1, 0, 2, a2, 0, 2): true

Поиск несовпадения методом mismatch():
Arrays.compare(a1, a2): -1
Arrays.compare(a2, a1): 1
Arrays.compare(a1, a3): 0
Arrays.compare(a1, 0, 2, a2, 0, 2): 0

Finding mismatch using the mismatch() method:
Arrays.mismatch(a1, a2): 2
Arrays.mismatch(a1, a3): -1
Arrays.mismatch(a1, 0, 5, a2, 0, 1): 1
```

API аплетов объявлен нерекомендуемым

Для работы Java-аплетов необходимо расширение браузера. Многие производители браузеров уже перестали поддерживать расширение для Java или планируют это сделать в ближайшем будущем. Но если браузер не поддерживает расширение Java, то аплеты работать не будут, а, значит, нет никаких причин использовать API аплетов. Поэтому в JDK 9 этот API объявлен нерекомендуемым. Однако в версии JDK 10 он не будет исключен. Если это и произойдет в какой-то будущей версии, то разработчики получат заблаговременное извещение. Нерекомендуемыми объявлены следующие классы и интерфейсы:

- `java.applet.AppletStub`
- `java.applet.Applet`
- `java.applet.AudioClip`
- `java.applet.AppletContext`
- `javax.swing.JApplet`

В JDK 9 все классы, относящиеся к AWT и Swing, помещены в модуль `java.desktop`. Эти нерекомендуемые классы и интерфейсы находятся там же.

Программа `appletviewer`, которая входит в состав JDK и находится в каталоге `bin`, предназначена для тестирования аплетов. Она также объявлена нерекомендуемой. Ее запуск в JDK 9 сопровождается предупреждением.

Усовершенствования в документации Java

В JDK 9 улучшен порядок написания, генерации и использования документации. В Javadoc теперь поддерживается HTML5. По умолчанию программа `javadoc` по-прежнему генерирует вывод в формате HTML4. Но добавлен параметр `-html5`, при задании которого генерируется документация в формате HTML5:

```
javadoc -html5 <other-options>
```

Программа `javadoc` находится в каталоге `JDK_HOME\bin`. Если запустить ее с параметром `-help`, то будет напечатана справка об использовании с описанием всех параметров. Полное рассмотрение `javadoc` выходит за рамки этой книги.

NetBeans позволяет сгенерировать документацию по проекту. В диалоговом окне **Properties** (Свойства) проекта выберите пункт **Build** → **Documenting** (Собрать → Документирование). В открывшемся окне можно будет задать все параметры программы `javadoc`. Для генерации документации выберите команду **Generate Javadoc** (Создать документацию Java) из контекстного меня проекта, появляющегося при щелчке правой кнопкой мыши.

В JDK 9 сохранена трехфреймовая или бесфреймовая компоновка. В левом верхнем фрейме находятся три ссылки: `ALL CLASSES`, `ALL PACKAGES` и `ALL MODULES`. Ссылка `ALL MODULES` добавлена в JDK 9, при щелчке по ней выводится список всех модулей. Ссылка `ALL CLASSES` позволяет просмотреть все классы в левом нижнем фрейме, а ссылка `ALL PACKAGES` – все пакеты. На рис. 20.1 показано, как изменилась страница документации Java.

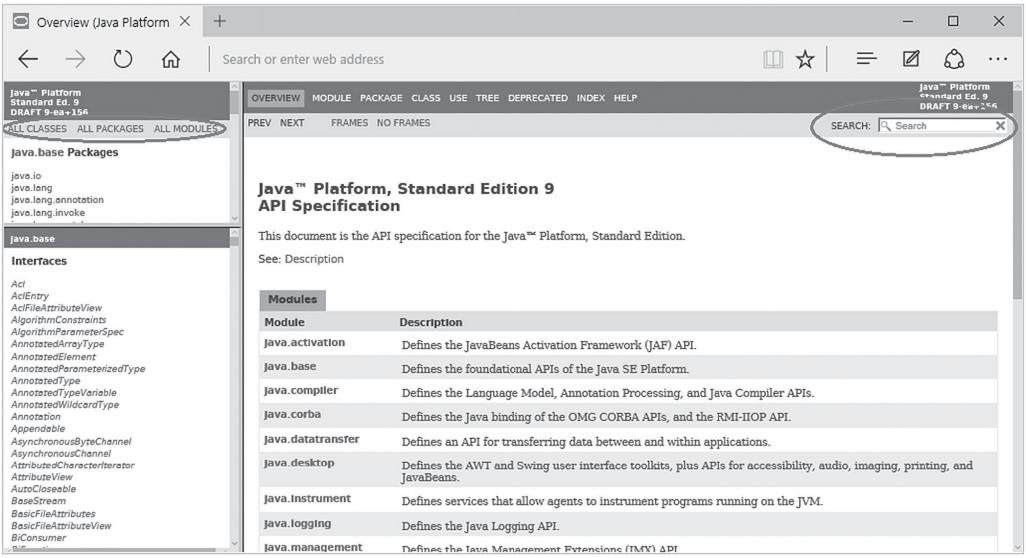

Рис. 20.1. Документация Java в трехфреймовой компоновке в Java SE 9

Можно также посмотреть документацию по конкретному модулю. Выберите модуль в левом верхнем фрейме, щелкните по ссылке с именем модуля, которая находится между верхним и нижним фреймом в левой части окна, – и в правом фрейме появится список всех предложений модуля: `requires`, `exports` и т. д.

Рассмотрим следующую ситуацию. Вы ищете алгоритм реализации какой-то идеи на Java и находите в Интернете код, в котором используется некий класс, но отсутствует соответствующее предложение `import`. У вас есть доступ к документации по Java SE, и вы хотите узнать поподробнее об этом классе. Как узнать имя пакета, необходимое для получения документации по классу? Вы снова ищете в Интернете, на этот раз имя класса, которое, возможно, будет сопровождаться ссылкой на документацию. Вместо этого можно скопировать фрагмент кода в

какую-нибудь Java IDE, например NetBeans или Eclipse, и дать IDE возможность сгенерировать необходимые предложения импорта с именем пакета. К счастью, в JDK 9 можно забыть обо всех неудобствах поиска имени пакета.

В правом фрейме есть еще одно добавление. Во всех отображаемых в нем страницах в правом верхнем углу имеется поле **Search** (Поиск) (см. рис. 20.1). Оно позволяет производить поиск в документации Java. Программа `javadoc` готовит индекс термов, по которому можно искать. Для успешного поиска нужно знать, какие термы индексируются:

- Можно искать по имени модуля, класса, пакета, типа и члена типа. Типы формальных параметров конструкторов и методов индексируются, имена параметров – нет. Таким образом, искать по типу формального параметра можно. Если ввести в поисковом поле `"(String, int, int)"`, то будет найден список конструкторов и методов, принимающих три формальных параметра типа `String`, `int` и `int`. Если в качестве поискового терма ввести `"util"`, то будет показан список всех пакетов, типов и членов, в именах которых встречается строка `"util"`.
- В JDK 9 введен новый внутритекстовый тег `@index`, который означает, что программа `javadoc` должна индексировать ключевое слово. Он может встречаться в виде `{@index <keyword> <description>}`, где `<keyword>` – индексируемое ключевое слово, а `<description>` – его описание. Вот пример использования тега `@index` с ключевым словом `jdojo`: `{@index jdojo Info site (www.jdojo.com) for the Java 9 Revealed book!}`.

Все, что не входит в этот перечень, не может быть объектом поиска по документации. Результаты поиска отображаются в виде списка с несколькими разделами: Modules, Packages, Types, Members и SearchTags. В раздел SearchTags входят результаты поиска по ключевым словам, индексированным благодаря наличию тегов `@index`.

> **Совет.** Поиск по документации не поддерживает регулярные выражения. Введенное поисковое слово ищется среди всех индексированных термов.

На рис. 20.2 показано поле **Search** и список результатов поиска. Я сгенерировал документацию Java для модуля `com.jdojo.misc` и поискал в ней слово `jdojo`, как показано в левой части рисунка. В правой части показан результат поиска терма `Module` в документации по Java SE 9.

Для навигации по результатам поиска можно использовать клавиши со стрелками вверх и вниз. Для просмотра подробных сведений можно поступить двумя способами:

- щелкнув по результату поиска, открыть документацию по этой теме;
- нажатием клавиш со стрелками подвести выделение к результату поиска и нажать клавишу **Enter**.

> **Совет.** Чтобы отказаться от поиска, запустите `javadoc` с параметром `-noindex`. Тогда программа не будет строить индекс и в сгенерированных страницах документации не будет поля поиска.

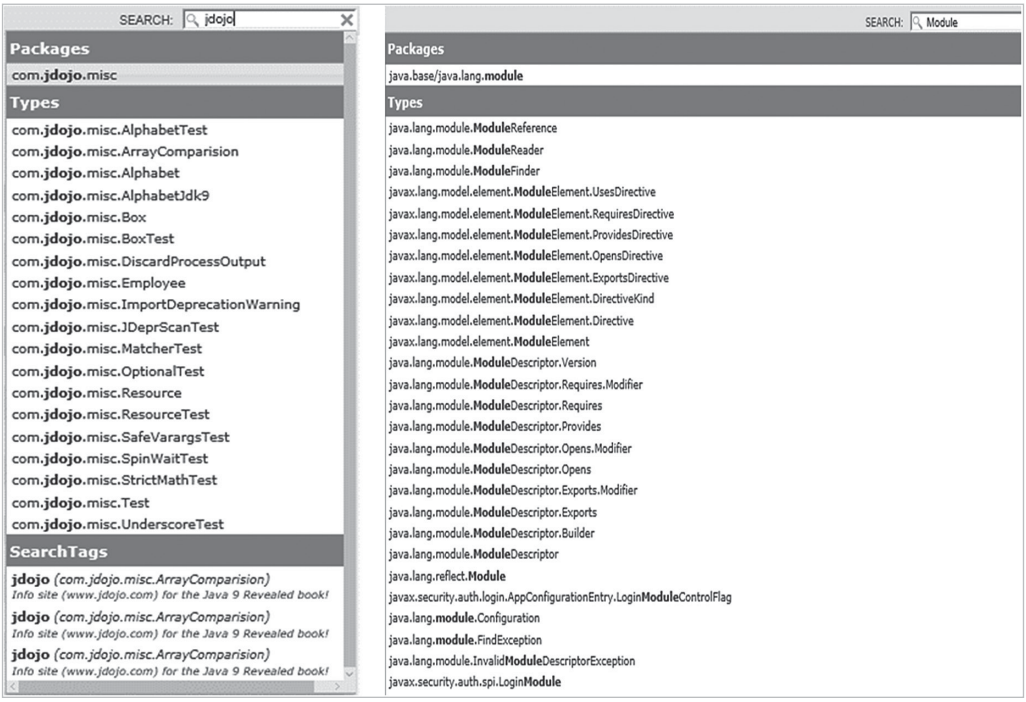

Рис. 20.2. Поле поиска и результаты поиска

Поиск по документации производится локально средствами клиентского JavaScript. Никаких обращений к серверу нет. Отключив JavaScript в браузере, вы не сможете воспользоваться поиском.

Поддержка платформенного рабочего стола

В Java SE 6 была добавлена поддержка платформенного рабочего стола с помощью класса `java.awt.Desktop`, который давал возможность выполнить из Java-приложения следующие операции:

- открытие URI в браузере, заданном пользователем по умолчанию;
- открытие почтового адреса в почтовом клиенте, заданном пользователем по умолчанию;
- открытие, редактирование и печать файлов с помощью зарегистрированных приложений.

В Java SE 9 поддержка платформенного рабочего стола получила дальнейшее развитие, добавлена поддержка открытых API для многих уведомлений о системных и прикладных событиях, если таковые существуют на конкретной платформе. Класс `java.awt.Desktop` по-прежнему является центральным, но для поддержки такого большого числа новых функций рабочего стола в Java SE 9 добавлен новый пакет `java.awt.desktop`, находящийся в модуле `java.desktop`. Класс `java.awt.Desktop` также получил много новых методов. Новый пакет содержит 30 классов и интерфейсов. В JDK 9 API рабочего стола поддерживает 24 действия и уведомления

платформенного рабочего стола, идентифицируемые элементами перечисления `Desktop.Action`. Перечислим некоторые из них:

- уведомления о том, что присоединенные дисплеи входят в режим энергосбережения или выходят из него;
- уведомления о том, что система засыпает или просыпается;
- уведомления об изменении состояния пользовательского сеанса, например, блокировке или разблокировке;
- уведомления об изменении состояния приложения: переходе в приоритетный или фоновый режим;
- уведомления о том, что приложение получило запрос показать диалоговое окно «О программе».

Все эти возможности можно использовать для оптимизации потребления ресурсов приложением. Например, если система засыпает, можно прекратить анимацию и возобновить ее, когда система проснется. Рассмотреть API рабочего стола во всей полноте в этой книге невозможно. Поэтому я только приведу подробный пример. Дополнительные сведения ищите в документации по классу `java.awt.Desktop`, а также классам и интерфейсам в пакете `java.awt.desktop`. Использование возможностей рабочего стола обычно сопровождается следующими шагами.

- Проверить, поддерживается ли класс `Desktop` на данной платформе, вызвав статический метод `isDesktopSupported()` этого класса. Если метод возвращает `false`, значит, все функции рабочего стола недоступны.
- Если класс `Desktop` поддерживается, получить от его статического метода `getDesktop()` ссылку на экземпляр класса `Desktop`.
- Не все функции рабочего стола доступны на всех платформах. Метод `isSupported(Desktop.Action action)` объекта рабочего стола проверяет, поддерживается ли конкретное действие. Перечень действий представлен элементами перечисления `Desktop.Action`.
- Если действие поддерживается, то можно вызвать один из методов класса `Desktop`, чтобы выполнить его, например, открыть файл или зарегистрировать обработчик события методом `addAppEventListener(SystemEventListener listener)`.

Совет. Пакеты `java.awt` и `java.awt.desktop` входят в модуль `java.desktop`. При использовании функций платформенного рабочего стола не забудьте прочитать этот модуль.

В листинге 20.15 приведен полный код программы, демонстрирующей операции с рабочим столом. Приложение регистрирует обработчик события изменения сеанса. Когда состояние пользовательского сеанса изменяется, приложение получает уведомление и печатает сообщение на стандартный вывод. Изменить состояние сеанса можно, войдя или выйдя удаленно либо заблокировав или разблокировав компьютер. Это уведомление от рабочего стола можно использовать для приостановки ресурсоемкой обработки, например анимации, в момент деактивации сеанса и последующего возобновления после активации. Во время работы программы вы должны будете заблокировать и разблокировать компьютер,

чтобы увидеть выведенные программой сообщения. Показанный в тексте результат был получен при прогоне программы в Windows. Программа самостоятельно завершается, проработав две минуты.

Листинг 20.15. Демонстрация функций платформенного рабочего стола

```
// DeskTopFrame.java
package com.jdojo.misc;

import java.awt.Desktop;
import java.awt.desktop.UserSessionEvent;
import java.awt.desktop.UserSessionListener;
import java.util.concurrent.TimeUnit;

public class DeskTopFrame {
   public static void main(String[] args) {
      // Проверить доступность класса Desktop
      if (!Desktop.isDesktopSupported()) {
         System.out.println("На данной платформе Desktop не поддерживается.");
         return;
      }

      System.out.println("На данной платформе Desktop поддерживается.");

      // Получить ссылку на рабочий стол
      Desktop desktop = Desktop.getDesktop();

      // Проверить, поддерживается ли уведомление о пользовательском сеансе
      if (!desktop.isSupported(Desktop.Action.APP_EVENT_USER_SESSION)) {
         System.out.println("Уведомления о пользовательском сеансе не " +
                            "поддерживаются рабочим столом");
         return;
      }

      System.out.println("Заблокируйте и разблокируйте пользовательский " +
                         "сеанс, чтобы увидеть уведомление об изменении.");

      // Добавить обработчик события изменения пользовательского сеанса
      desktop.addAppEventListener(new UserSessionListener() {
         @Override
         public void userSessionDeactivated(UserSessionEvent e) {
            System.out.println("Пользовательский сеанс деактивирован. Причина: " +
                               e.getReason());
         }

         @Override
         public void userSessionActivated(UserSessionEvent e) {
```

```
            System.out.println("Пользовательский сеанс активирован. Причина: " +
                               e.getReason());
         }
      });

      // Текущий поток засыпает на две минуты
      try {
         TimeUnit.SECONDS.sleep(120);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}
```

```
На данной платформе Desktop поддерживается.
Заблокируйте и разблокируйте пользовательский сеанс, чтобы увидеть уведомление об изменении.
Пользовательский сеанс деактивирован. Причина: LOCK
Пользовательский сеанс активирован. Причина: LOCK
```

Метод `main()`, проверяет, доступен ли класс `Desktop` на данной платформе. Если нет, то программа завершается, иначе запрашивается ссылка на рабочий стол.

```
if (!Desktop.isDesktopSupported()) {
    System.out.println("На данной платформе Desktop не поддерживается.");
    return;
}

// Получить ссылку на рабочий стол
Desktop desktop = Desktop.getDesktop();
```

Нас интересует получение уведомлений об изменении пользовательского сеанса, так что нужно проверить, поддерживается ли эта функция. Если нет, то программа завершается.

```
if (!desktop.isSupported(Desktop.Action.APP_EVENT_USER_SESSION)) {
    System.out.println("Уведомления о пользовательском сеансе не " +
                       "поддерживаются рабочим столом");
    return;
}
```

Если уведомления об изменении пользовательского сеанса поддерживаются, то необходимо зарегистрировать обработчик событий типа `UserSessionListener`:

```
desktop.addAppEventListener(new UserSessionListener() {
    @Override
    public void userSessionDeactivated(UserSessionEvent e) {
        System.out.println("Пользовательский сеанс деактивирован. Причина: " +
```

```
                        e.getReason());
    }

    @Override
    public void userSessionActivated(UserSessionEvent e) {
        System.out.println("Пользовательский сеанс активирован. Причина: " +
                        e.getReason());
    }
});
```

Методы `userSessionActivated()` и `userSessionDeactivated()` зарегистрированного обработчика вызываются в момент активации и деактивации пользовательского сеанса соответственно. Обоим передается объект типа `UserSessionEvent`. Метод `getReason()` класса `UserSessionEvent` возвращает значение типа перечисления `UserSessionEvent.Reason`, в котором определены возможные причины изменения сеанса: `CONSOLE`, `LOCK`, `REMOTE` и `UNSPECIFIED`. Константы `CONSOLE` и `REMOTE` означают, что сеанс был подключен/отключен к консоли или к удаленному терминалу соответственно. Константа `LOCK` означает, что сеанс был заблокирован или разблокирован, а константа `UNSPECIFIED` представляет все остальные причины изменения.

В конце метода `main()` текущий поток засыпает на два минуты, чтобы у пользователя был шанс заблокировать или разблокировать сеанс. Если эту часть программы удалить, то программа завершится, не дожидаясь изменения сеанса.

Фильтры десериализации объектов

В Java имеются средства сериализации и десериализации объектов. Чтобы снизить риски, связанные с десериализацией, в JDK 9 введена концепция входного фильтра объектов, который позволяет проверить десериализуемый объект и прекратить десериализацию, если проверка не проходит. Фильтр является экземпляром нового интерфейса `java.io.ObjectInputFilter`. В основу фильтрации могут быть положены следующие критерии:

- длина десериализуемого массива;
- глубина вложенности десериализуемого объекта;
- количество десериализуемых ссылок на объекты;
- класс десериализуемого объекта;
- количество байтов, занятых объектом во входном потоке.

Интерфейс `ObjectInputFilter` включает единственный метод:

```
ObjectInputFilter.Status checkInput(ObjectInputFilter.FilterInfo filterInfo)
```

Можно задать глобальный фильтр, применяемый к десериализации всех объектов. Глобальный фильтр можно переопределить для конкретного потока `ObjectInputStream`, установив для него локальный фильтр. Можно вообще обойтись без глобального фильтра, а задавать локальный фильтр для каждого входного потока объектов. Существует несколько способов создания и установки фильтров. Все они будут рассмотрены ниже на примерах. В первом разделе я расскажу о добавленных в JDK 9 классах и интерфейсах для работы с фильтрами:

- `ObjectInputFilter`
- `ObjectInputFilter.Config`
- `ObjectInputFilter.FilterInfo`
- `ObjectInputFilter.Status`

Экземпляры интерфейса `ObjectInputFilter` представляют фильтры. Чтобы создать фильтр, можно написать класс, реализующий этот интерфейс. А можно вместо этого получить его экземпляры из строки, воспользовавшись методом `createFilter(String pattern)` класса `ObjectInputFilter.Config`.

Вложенный статический служебный класс `ObjectInputFilter.Config` используется для двух целей:

- получить и установить глобальный фильтр;
- создать фильтр из заданной строки-спецификации.

В классе `ObjectInputFilter.Config` имеются три статических метода:

- `ObjectInputFilter createFilter(String pattern)`
- `ObjectInputFilter getSerialFilter()`
- `void setSerialFilter(ObjectInputFilter filter)`

Метод `createFilter()` принимает строку-спецификацию и возвращает экземпляр интерфейса `ObjectInputFilter`. Ниже показано, как создать фильтр, который проверяет, что длина десериализуемого массива не больше 4:

```
String pattern = "maxarray=4";
ObjectInputFilter filter = ObjectInputFilter.Config.createFilter(pattern);
```

В одном фильтра можно задать несколько спецификаций, разделенных точкой с запятой. В следующем фрагменте создается фильтр с двумя условиями. Он прервет десериализацию, если встретит массив длины больше 4 или объект, размер которого превышает 1024 байта.

```
String pattern = "maxarray=4;maxbytes=1024";
ObjectInputFilter filter = ObjectInputFilter.Config.createFilter(pattern);
```

Существует несколько правил задания спецификаций фильтра. Если вы предпочитаете выражать логику фильтра на Java, то можете создать класс, реализующий интерфейс `ObjectInputFilter`, поместив алгоритм проверки в метод `checkInput()`. Ниже перечислены правила, которых следует придерживаться при создании фильтра из строки.

Из пяти критериев фильтрации четыре задают верхние границы: `maxarray`, `maxdepth`, `maxrefs` и `maxbytes`. Они задаются с помощью пар `name=value`, где `name` – одно из вышеперечисленных ключевых слов, а `value` – ограничение сверху. Если в строке встречается знак `=`, то в качестве имени должно фигурировать одно из этих ключевых слов. Пятый критерий служит для задания имени класса в виде:

```
<module-name>/<fully-qualified-class-name>
```

Если класс находится в безымянном модуле, то производится сравнение с именем класса. Если объект является массивом, то под классом понимается тип эле-

мента массива, а не тип самого массива. Ниже перечислены правила сопоставления имени класса с образцом:

- если имя класса соответствует образцу, то десериализация объекта разрешена;
- знак `!` в начале строке трактуется как логическое НЕ;
- если образец содержит знак `/`, то часть до него рассматривается как имя модуля. Если имя модуля совпадает с именем класса, то часть после косой черты рассматривается как образец для сравнения с именем класса. Если в образце нет знака `/`, то имя модуля не участвует в сравнении;
- образец, заканчивающийся символами `".**"`, сопоставляется с любым классом в пакете и всех его подпакетах;
- образец, заканчивающийся символами `".*"`, сопоставляется с любым классом в пакете.
- образец, заканчивающийся символом `"*"`, сопоставляется с любым классом, имя которого содержит образец в качестве префикса;
- если образец совпадает с именем класса, то сопоставление считается успешным;
- в противном случае имя класса считается не соответствующим образцу, и объект отклоняется.

Если задать строку `"com.jdojo.**"` в качестве спецификации фильтра, то будет разрешено десериализовывать все классы в пакете `com.jdojo` и его подпакетах, а объекты других классов будут отклонены. Если задать спецификацию `"!com.jdojo.**"`, то будут отклонены все классы в пакете `com.jdojo` и его подпакетах, а десериализация объектов всех остальных классов будет разрешена.

Методы `getSerialFilter()` и `setSerialFilter()` служат для получения и задания глобального фильтра. Задать глобальный фильтр можно тремя способами:

- задать системное свойство `jdk.serialFilter`, значением которого является последовательность спецификаций фильтра, разделенных точкой с запятой;
- задать свойство `jdk.serialFilter` в файле `java.security`, находящемся в каталоге `JAVA_HOME\conf\security`. Если для выполнения программы используется JDK, то `JAVA_HOME` – это `JDK_HOME`, в противном случае это `JRE_HOME`;
- вызвать статический метод `setSerialFilter()` класса `ObjectInputFilter.Config`.

В следующей командной строке для выполнения класса установлено системное свойство `jdk.serialFilter`. На остальные части командной строки не обращайте внимания.

```
C:\Java9Revealed>java -Djdk.serialFilter=maxarray=100;maxdepth=3;com.jdojo.** --module-path
com.jdojo.misc\build\classes --module com.jdojo.misc/com.jdojo.misc.ObjectFilterTest
```

В листинге 20.16 показана часть гораздо большего конфигурационного файла `JAVA_HOME\conf\security\java.security`, в которой задается фильтр, эквивалентный предыдущему.

Листинг 20.16. Часть файла `JAVA_HOME\conf\security\java.security`, в которой задается свойство `jdk.serialFilter`

```
maxarray=100;maxdepth=3;com.jdojo.**
```

Совет. Если фильтр задан в системном свойстве и в конфигурационном файле, то используется системное свойство.

При выполнении команды `java` с установленным глобальным фильтром в `stderr` будут выводиться такие сообщения:

```
Feb 17, 2017 9:23:45 AM java.io.ObjectInputFilter$Config lambda$static$0
INFO: Creating serialization filter from maxarray=20;maxdepth=3;!com.jdojo.**
```

Это платформенные сообщения, протоколируемые с помощью диспетчера `java.io.serialization` для модуля `java.base`. Если задан диспетчер платформенного протоколирования (см. главу 19), то сообщения будут передаваться ему. В одном из таких сообщений приводится информация о глобальном фильтре, заданном с помощью системного свойства или конфигурационного файла.

Глобальный фильтр можно задать также с помощью статического метода `setSerialFilter()` класса `ObjectInputFilter.Config`:

```
// Создать фильтр
String pattern = "maxarray=100;maxdepth=3;com.jdojo.**";
ObjectInputFilter globalFilter = ObjectInputFilter.Config.createFilter(pattern);

// Установить глобальный фильтр
ObjectInputFilter.Config.setSerialFilter(globalFilter);
```

Совет. Глобальный фильтр можно установить только один раз. Например, если фильтр установлен с помощью системного свойства `jdk.serialFilter`, то вызов метода `Config.setSerialFiter()` приведет к исключению `IllegalStateException`. Если глобальный фильтр устанавливается методом `Config.setSerialFiter()`, то методу следует передать аргумент, отличный от `null`. Эти правила призваны гарантировать, что глобальный фильтр, установленный с помощью системного свойства или конфигурационного файла, не будет переопределен в коде программы.

Для получения глобального фильтра используется статический метод `getSerialFilter()` класса `ObjectInputFilter.Config` – независимо от того, как фильтр был установлен. Если глобального фильтра нет, то этот метод вернет `null`.

`ObjectInputFilter.FilterInfo` – вложенный статический интерфейс, экземпляр которого представляет текущий контекст десериализации и передается методу `checkInput()` вашего фильтра. Вы не должны реализовывать этот интерфейс и создавать его экземпляры самостоятельно. Интерфейс содержит следующие методы, которые вызываются из метода фильтра `checkInput()` для чтения текущего контекста десериализации:

- `Class<?> serialClass()`
- `long arrayLength()`
- `long depth()`
- `long references()`
- `long streamBytes()`

Метод `serialClass()` возвращает класс десериализуемого объекта. В случае массива возвращается класс массива, а не его элемента. Если в процессе десериализации новый объект не создан, то метод возвращает `null`.

Метод `arrayLength()` возвращает длину десериализуемого массива. Если десериализуемый объект – не массив, возвращается –1.

Метод `depth()` возвращает глубину вложенности десериализуемого объекта. Начальное значение глубины равно 1, увеличивается на 1 для каждого уровня вложенности и уменьшается на 1 при выходе из вложенного объекта.

Метод `references()` возвращает текущее число десериализуемых ссылок на объекты.

Метод `streamBytes()` возвращает текущее количество байтов, прочитанных из входного потока.

Объект может удовлетворять или не удовлетворять заданному в фильтре критерию. В зависимости от результатов проверки необходимо вернуть тот или иной элемент перечисления `ObjectInputFilter.Status`. Обычно они возвращаются в качестве значения метода `checkInput()`:

- `ALLOWED`
- `REJECTED`
- `UNDECIDED`

Константа `ALLOWED` означает, что десериализация разрешена, `REJECTED` – что запрещена, `UNDECIDED` – что решение не принято. Обычно значение `UNDECIDED` возвращается в том случае, когда принять решение о десериализации текущего объекта должен какой-то другой фильтр. Если фильтр служит для отклонения некоторых классов, то можно вернуть `REJECTED` для заведомо отклоняемых классов и `UNDECIDED` для остальных. В листинге 20.17 приведен простой фильтр на основе длины массива.

Листинг 20.17. Фильтр десериализации объектов, который разрешает десериализацию массивов указанной длины

```
// ArrayLengthObjectFilter.java
package com.jdojo.misc;

import java.io.ObjectInputFilter;

public class ArrayLengthObjectFilter implements ObjectInputFilter {
   private long maxLenth = -1;

   public ArrayLengthObjectFilter(int maxLength) {
      this.maxLenth = maxLength;
   }

   @Override
   public Status checkInput(FilterInfo info) {
      long arrayLength = info.arrayLength();
      if (arrayLength >= 0 && arrayLength > this.maxLenth) {
         return Status.REJECTED;
      }

      return Status.ALLOWED;
```

```
    }
}
```

А в следующем фрагменте этот фильтр используется, поскольку мы задаем максимальную длину массива 3. Если во входном потоке объектов встретится массив большей длины, то десериализация закончится исключением `java.io.InvalidClassException`. Обработка исключения опущена.

```
ArrayLengthObjectFilter filter = new ArrayLengthObjectFilter(3);
File inputFile = ...
ObjectInputStream in = new ObjectInputStream(new FileInputStream(inputFile))) {
    in.setObjectInputFilter(filter);
    Object obj = in.readObject();
```

В листинге 20.18 приведен код класса `Item`. Для краткости методы чтения и установки опущены. Объекты этого класса используются для демонстрации фильтров десериализации.

Листинг 20.18. Класс Item для демонстрации фильтров десериализации

```
// Item.java
package com.jdojo.misc;

import java.io.Serializable;
import java.util.Arrays;

public class Item implements Serializable {
    private int id;
    private String name;
    private int[] points;

    public Item(int id, String name, int[] points) {
        this.id = id;
        this.name = name;
        this.points = points;
    }

    /* Здесь должны быть методы чтения и установки */
    @Override
    public String toString() {
        return "[id=" + id + ", name=" + name + ", points=" + Arrays.toString(points) + "]";
    }
}
```

В листинге 20.19 приведен код класса `ObjectFilterTest`, демонстрирующего использование фильтров в процессе десериализации объектов.

Листинг 20.19. Класс ObjectFilterTest, в котором используется несколько фильтров десериализации

```
// ObjectFilterTest.java
package com.jdojo.misc;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputFilter;
import java.io.ObjectInputFilter.Config;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class ObjectFilterTest {
   public static void main(String[] args) {
      // Относительный путь к входному/выходному файлу
      File file = new File("serialized", "item.ser");

      // Гарантировать существование каталогов
      ensureParentDirExists(file);

      // Создать объект Item, который будет подвергнут сериализации и десериализации
      Item item = new Item(100, "Pen", new int[]{1,2,3,4});

      // Сериализовать объект item
      serialize(file, item);

      // Распечатать глобальный фильтр
      ObjectInputFilter globalFilter = Config.getSerialFilter();
      System.out.println("Глобальный фильтр: " + globalFilter);

      // Десериализовать объект item
      Item item2 = deserialize(file);
      System.out.println("Десериализован с применением глобального фильтра: " + item2);

      // Использовать фильтр для отклонения массивов длины > 2
      String maxArrayFilterPattern = "maxarray=2";
      ObjectInputFilter maxArrayFilter = Config.createFilter(maxArrayFilterPattern);
      Item item3 = deserialize(file, maxArrayFilter);
      System.out.println("Десериализован с фильтром maxarray=2: " + item3);

      // Создать пользовательский фильтр
      ArrayLengthObjectFilter customFilter = new ArrayLengthObjectFilter(5);
      Item item4 = deserialize(file, customFilter);
      System.out.println("Десериализован с пользовательским фильтром (maxarray=5): " +
                         item4);
```

```
    }

    private static void serialize(File file, Item item) {
        try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(file))) {
            out.writeObject(item);
            System.out.println("Сериализованный объект: " + item);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static Item deserialize(File file) {
        try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(file))) {
            Item item = (Item)in.readObject();
            return item;
        } catch (Exception e) {
            System.out.println("Не удалось десериализовать item. Ошибка: " +
                               e.getMessage());
        }
        return null;
    }

    private static Item deserialize(File file, ObjectInputFilter filter) {
        try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(file))) {
            // Установить переданный фильтр входных объектов
            in.setObjectInputFilter(filter);
            Item item = (Item)in.readObject();
            return item;
        } catch (Exception e) {
            System.out.println("Не удалось десериализовать item. Ошибка: " +
                               e.getMessage());
        }
        return null;
    }

    private static void ensureParentDirExists(File file) {
        File parent = file.getParentFile();
        if(!parent.exists()) {
            parent.mkdirs();
        }
        System.out.println("Входной/выходной файл " + file.getAbsolutePath());
    }
}
```

Класс `ObjectFilterTest` сериализует объект класса `Item`, а затем пытается десериализовать его с применением различных фильтров. Метод `ensureParentDirExists()` принимает объект `File`, проверяет, существует ли родительский каталог и при необходимости создает его. Заодно печатается путь к файлу.

Метод `serialize()` сериализует указанный объект `Item` в указанный файл. Он один раз вызывается из `main()` для сериализации объекта.

Метод `deserialize()` перегружен. Вариант `deserialize(File file)` десериализует объект `Item`, хранящийся в указанном файле, с применением глобального фильтра, а вариант `deserialize(File file, ObjectInputFilter filter)` – с применением указанного фильтра. Обратите внимание, что во втором варианте вызывается метод `in.setObjectInputFilter(filter)`, который замещает глобальный фильтр, если таковой был задан.

Метод `main()` распечатывает глобальный фильтр, создает объект `Item`, сериализует его, создает несколько локальных фильтров и десериализует объект, применяя различные фильтры. Следующая команда выполняет класс `ObjectFilterTest` без глобального фильтра.

```
C:\Java9Revealed>java --module-path com.jdojo.misc\build\classes
 --module com.jdojo.misc/com.jdojo.misc.ObjectFilterTest
```

```
Входной/выходной файл C:\Java9Revealed\serialized\item.ser
Сериализованный объект: [id=100, name=Pen, points=[1, 2, 3, 4]]
Глобальный фильтр: null
Десериализован с применением глобального фильтра: [id=100, name=Pen, points=[1, 2, 3, 4]]
Не удалось десериализовать item. Ошибка: filter status: REJECTED
Десериализован с фильтром maxarray=2: null
Десериализован с пользовательским фильтром (maxarray=5): [id=100, name=Pen, points=[1, 2, 3, 4]]
```

Следующая команда выполняет класс `ObjectFilterTest` с глобальным фильтром `maxarray=1`, который не дает десериализовывать массивы, содержащие более одного элемента. Глобальный фильтр задан с помощью системного свойства `jdk.serialFilter`. Поскольку используется глобальный фильтр, на `stderr` выводятся сообщения от классов JDK. Они выделены полужирным шрифтом.

```
C:\Java9Revealed>java -Djdk.serialFilter=maxarray=1
 --module-path com.jdojo.misc\build\classes
 --module com.jdojo.misc/com.jdojo.misc.ObjectFilterTest
```

```
Входной/выходной файл C:\Java9Revealed\serialized\item.ser
Сериализованный объект: [id=100, name=Pen, points=[1, 2, 3, 4]]
Feb 17, 2017 1:09:57 PM java.io.ObjectInputFilter$Config lambda$static$0
INFO: Creating serialization filter from maxarray=1
Глобальный фильтр: maxarray=1
Не удалось десериализовать item. Ошибка: filter status: REJECTED
Десериализован с применением глобального фильтра: null
Не удалось десериализовать item. Ошибка: filter status: REJECTED
Десериализован с фильтром maxarray=2: null
Десериализован с пользовательским фильтром (maxarray=5): [id=100, name=Pen, points=[1, 2, 3, 4]]
```

Обратите внимание, что выводится в случае применения глобального фильтра. Поскольку объект `Item` содержит массив с четырьмя элементами, глобальный фильтр не дает его десериализовать. Однако тот же объект удалось десериализовать с помощью фильтра `ArrayLengthObjectFilter`, поскольку он заместил глобальный фильтр и допускает массивы, содержащие до 5 элементов.

Добавления в API ввода-вывода

В JDK 9 добавлено несколько вспомогательных методов в API ввода-вывода. Прежде всего, отметим новый метод класса `InputStream`:

```
long transferTo(OutputStream out) throws IOException
```

Вам, наверное, приходилось писать код, который читает все байты из входного потока только для того, чтобы записать их в выходной поток. Теперь можно обойтись без цикла чтения-записи. Метод `transferTo()` читает все байты из входного потока и в том же порядке записывает их в выходной поток. Метод возвращает количество скопированных байтов.

> **Совет.** Метод `transferTo()` не закрывает потоки. После возврата управления будет достигнут конец входного потока.

Ниже показано, как в одной строке скопировать содержимое файла `log.txt` в файл `log_copy.txt` (обработка исключений опущена).

```
new FileInputStream("log.txt").transferTo(new FileOutputStream("log_copy.txt"));
```

В класс `java.nio.Buffer` добавлено два метода:

- `abstract Buffer duplicate()`
- `abstract Buffer slice()`

Тот и другой возвращают объект `Buffer`, разделяющий содержимое с исходным буфером. Возвращаемый буфер будет прямым или допускающим только чтение, только если таковым был исходный буфер. Метод `duplicate()` возвращает буфер с такой же емкостью, пределом, позицией и отметкой, что у исходного. Метод `slice()` возвращает буфер, для которого позиция равна 0, емкость и предел равны числу элементов, оставшихся в буфере, а отметка не определена. Содержимое возвращенного буфера начинается с текущей позиции исходного. В буфере, возвращенном этим методом, позиция, предел и отметка изменяются независимо от исходного. В приведенном ниже фрагменте демонстрируются характеристики дублированного и срезанного буфера.

```
IntBuffer b1 = IntBuffer.wrap(new int[]{1, 2, 3, 4});
IntBuffer b2 = b1.duplicate();
IntBuffer b3 = b1.slice();
System.out.println("b1=" + b1);
System.out.println("b2=" + b2);
```

```
System.out.println("b2=" + b3);

// Сдвинуть b1 на 1 позицию
b1.get();
IntBuffer b4 = b1.duplicate();
IntBuffer b5 = b1.slice();
System.out.println("b1=" + b1);
System.out.println("b4=" + b4);
System.out.println("b5=" + b5);
```

```
b1=java.nio.HeapIntBuffer[pos=0 lim=4 cap=4]
b2=java.nio.HeapIntBuffer[pos=0 lim=4 cap=4]
b2=java.nio.HeapIntBuffer[pos=0 lim=4 cap=4]
b1=java.nio.HeapIntBuffer[pos=1 lim=4 cap=4]
b4=java.nio.HeapIntBuffer[pos=1 lim=4 cap=4]
b5=java.nio.HeapIntBuffer[pos=0 lim=3 cap=3]
```

Резюме

В JDK 9 знак подчеркивания (_) является ключевым словом, поэтому не может выступать в роли односимвольного идентификатора: имени переменной, метода, типа и т. д. Однако в многосимвольных идентификаторах его использование по-прежнему разрешено.

В JDK 9 снято ограничение, требующее объявлять новую переменную для ресурсов, управляемых блоком `try` с ресурсами. Теперь для ссылки на такой ресурс можно использовать *финальную* или *эффективно финальную* переменную.

В JDK 9 добавлена поддержка ромбовидного оператора в анонимных классах при условии, что выведенный класс денотируемый.

В интерфейсах могут присутствовать закрытые методы, если они являются статическими или не абстрактными и не методами по умолчанию.

В JDK 9 разрешено применять аннотацию `@SafeVarargs` к закрытым методам. В JDK 8 это было разрешено для конструкторов, статических и финальных методов.

В JDK 9 во вложенный класс `ProcessBuilder.Redirect` добавлена константа `DISCARD` типа `ProcessBuilder.Redirect`. Это место назначения можно указывать для потоков вывода и ошибок подпроцессов, если требуется отбросить все, что в них выводится. Это реализовано путем записи в «null-файл» операционной системы.

В JDK 9 в классы `Math` и `StrictMath` добавлено несколько методов для поддержки дополнительных математических операций, в т. ч. `floorDiv(long x, int y)`, `floorMod(long x, int y)`, `multiplyExact(long x, int y)`, `multiplyFull(int x, int y)`, `multiplyHigh(long x, long y)` и др.

В JDK 9 в класс `java.util.Optional` добавлены методы `ifPresentOrElse()`, `or()` и `stream()`. Метод `ifPresentOrElse()` предлагает два варианта действий. Если значение присутствует, то к нему применяется указанное действие, в противном случае выполняется действие `emptyAction`. Метод `or()` возвращает сам объект `Optional`, если тот

содержит значение, а в противном случае — объект Optional, возвращенный указанным объектом supplier. Метод stream() возвращает последовательный поток, содержащий значение, присутствующее в объекте Optional. Если Optional пуст, то возвращается пустой поток. Метод stream() полезен в сочетании с методом flatMap.

В JDK 9 добавлен статический метод onSpinWait() в класс Thread. Это просто уведомление процессору о том, что вызывающий поток временно не может продолжать работу, поэтому использование ресурсов можно оптимизировать. Рекомендуется использовать в цикле активного ожидания.

Time API получил в JDK 9 дальнейшее развитие. Добавлено несколько методов в классы Duration, LocalDate, LocalTime и OffsetTime. Новый метод datesUntil() класса LocalDate возвращает поток дат между двумя заданными с шагом, равным одному дню или заданному периоду. Добавлены также новые спецификаторы формата.

Класс Matcher получил перегруженные варианты нескольких существующих методов, которые раньше работали с объектами StringBuffer, а теперь и с объектами StringBuilder. Новый метод results() возвращает поток Stream<MatchResult>. В класс Objects добавлены методы проверки выхода за границы массива и коллекции.

В класс java.util.Arrays добавлены методы для сравнения массивов и срезок массивов на равенство и для поиска первого несовпадения.

Система документации Java в JDK 9 заметно усовершенствована. Теперь поддерживается HTML5. Для генерации документации в формате HTML5 следует запустить программу javadoc с параметром -html5. Имена всех модулей, пакетов, типов, членов типов и типов формальных параметров индексируются, так что по ним можно осуществлять поиск. В правом верхнем углу каждой страницы документации отображается поле поиска по индексированным термам. Кроме того, для поиска по пользовательским термам можно использовать новый тег @index. Поиск выполняется средствами клиентского JavaScript без обращения к какому-либо серверу.

Многие производители браузеров уже исключили поддержку расширения Java или планируют сделать это в ближайшем будущем. В связи с этим в JDK 9 объявлен нерекомендуемым API аплетов: все типы в пакете java.applet и класс javax.swing.JApplet, а также программа appletviewer.

В JDK 6 уже была добавлена ограниченная поддержка платформенного рабочего стола в виде класса java.awt.Desktop: открытие URI в браузере, заданном пользователем по умолчанию, открытие почтового адреса в почтовом клиенте, заданном пользователем по умолчанию, и открытие, редактирование и печать файлов с помощью зарегистрированных приложений. В Java SE 9 поддержка платформенного рабочего стола получила дальнейшее развитие, добавлена поддержка открытых API для многих уведомлений о системных и прикладных событиях, если таковые существуют на конкретной платформе. Для поддержки такого большого числа новых функций рабочего стола в Java SE 9 добавлен новый пакет java.awt.desktop, находящийся в модуле java.desktop. Класс java.awt.Desktop также получил много новых методов. В JDK 9 API рабочего стола поддерживает 24 действия и уведомления платформенного рабочего стола, в т. ч. уведомления о том, что присоединенные дисплеи входят в режим энергосбережения или выходят из него, уведомления о том, что система засыпает или просыпается и т. д.

Чтобы снизить риски, связанные с десериализацией, в JDK 9 введена концепция входного фильтра объектов, который позволяет проверить десериализуе-

мый объект и прекратить десериализацию, если проверка не проходит. Фильтр является экземпляром нового интерфейса `java.io.ObjectInputFilter`. Можно задать глобальный фильтр на уровне системы, который будет применяться к десериализации любого объекта. Глобальный фильтр задается с помощью нового системного свойства `jdk.serialFilter`, либо с помощью свойства `jdk.serialFilter` в конфигурационном файле `JAVA_HOME\conf\security\java.security`, либо путем вызова метода `setSerialFilter()` класса `ObjectInputFilter.Config`. Локальный фильтр для объекта `ObjectInputStream` задается с помощью его метода `setObjectInputFilter()`, при этом глобальный фильтр замещается.

В класс `java.io.InputStream` добавлен метод `transferTo(OutputStream out)`, позволяющий прочитать все байты из входного потока и записать их в том же порядке в выходной поток. Этот метод не закрывает ни тот, ни другой поток. В класс `java.nio.Buffer` добавлены два метода, `duplicate()` и `slice()`, позволяющие продублировать буфер и получить его срезку. Результирующие буферы разделяют содержимое с исходным, но позиция, предел и отметка изменяются независимо.

Предметный указатель

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. (499) 782-38-89

Электронный адрес: **books@alians-kniga.ru**.

Кишори Шаран

Java 9. Полный обзор нововведений
Для быстрого ознакомления и миграции

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16. Печать цифровая.
Усл. печ. л. 44,20. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф