# Программируем на Java

Практический подход к созданию надежных и эффективных приложений

Аарон Плоетц

# Code with Java 21

*A practical approach for building robust and efficient applications*

**Aaron Ploetz**

Аэрон Плоетц

# ПРОГРАММИРУЕМ НА JAVA

2025

**Плоетц А.**

Программируем на Java: пер. с англ. — 2025. — 368 с.: ил.

Книга о современном программировании на Java, охватывающая новые возможности Java 21 и практическое применение языка для создания многофункциональных приложений. Подробно рассматриваются ключевые концепции, структуры данных, работа с реляционными базами данных PostgreSQL и Apache Cassandra®, а также использование фреймворков Spring Boot и Spring Data. Отдельное внимание уделено разработке графики, анимации и даже небольших аркадных игр на Java. Рассматриваются лучшие практики построения программной архитектуры, создания графических интерфейсов и веб-приложений с помощью Vaadin.

Книга будет полезна как начинающим разработчикам, так и опытным специалистам, желающим освоить новые возможности Java 21 и углубить знания в области современного программирования.

*Для программистов*

# Содержание

*Посвящается моим детям:*
*Хадии, Эйвери, Эмили и Виктории.*
*&*
*Моему крестнику:*
*Тиму*

*Дорога к успеху вымощена следами тех,*
*кто не сдался, и сожалениями тех, кто сдался.*

# Об авторе

Аарон Плоетц (Aaron Ploetz) — developer advocate в DataStax. С 1997 года профессионально занимается разработкой программного обеспечения и имеет успешный опыт руководства командами DBA и DevOps как в стартапах, так и в компаниях из списка Fortune 50. Он трижды был назван MVP для Apache Cassandra® и выступал на различных мероприятиях, включая ключевой доклад по финтеху на Data Day Mexico City 2023. Аарон часто отвечает на вопросы других разработчиков на StackOverflow и является автором книг по распределенным базам данных. Он получил степень бакалавра в области управления компьютерных систем в Университете Висконсин-Уайтуотер (University of Wisconsin-Whitewate) и степень магистра в области разработки программного обеспечения (специализация — технология баз данных) в Университете Реджис (Regis University).

В свободное время Аарон увлекается рыбалкой, ретро-видеоиграми и лыжным спортом. Аарон и его жена Корин живут со своими четырьмя детьми в районе Twin[1] Cities.

[1] Агломерация Миннеаполиса и Сент-Пола. — *Прим. перев.*

# О рецензенте

Отавио Сантана (Otavio Santana) — увлеченный архитектор и инженер по программному обеспечению, специализирующийся на облачных и Java-технологиях. Он обладает глубоким опытом в области высокопроизводительных приложений в сфере финансов, социальных сетей и электронной коммерции.

Отавио Сантана внес большой вклад в развитие Java и экосистемы открытого кода. Он помогал в определении направления и целей платформы Java, начиная с Java 8, в качестве исполнительного члена JCP, а также был коммиттером и руководителем нескольких продуктов и спецификаций с открытым исходным кодом.

Отавио заслужил признание за свой вклад в развитие открытого исходного кода и получил множество наград, включая все категории JCP Awards и Duke's Choice Award. Он также является выдающимся участником программы Java Champions и Oracle ACE.

# Благодарности

Я хотел бы выразить признательность моим коллегам, Седрику Лунвену (Cédrick Lunven) и Мэри Григлески (Mary Grygleski), которые оказали мне огромную помощь и сыграли решающую роль в том, что я вернулся к Java. Я также хотел бы поблагодарить Отавио Сантану (Otavio Santana), Шарата Чандера (Sharat Chander) и многих других, кто сделал сообщество Java таким гостеприимным и ободряющим.

Я также хотел бы поблагодарить Урсулу Келлманн (Ursula Kellmann), которая ушла от нас слишком рано. Она была замечательным наставником, который многому научил меня в Java и являлся блестящим примером того, каким должен быть настоящий эксперт в своем деле.

Наконец, я хотел бы поблагодарить свою жену Корин (Coriene), которая постоянно вдохновляет меня принимать новые вызовы и быть лучшей версией самого себя.

# Предисловие

Изучение разработки программного обеспечения всегда было непростым делом. В начале моего пути книги были единственным доступным средством. Когда я приступил к изучению BASIC на своем Tandy 1000, я впервые познакомился с Java на последнем курсе университета Висконсин-Уайтуотер (1998 год). Мне нравилось, что Java провозглашала принцип "код один, выполнение везде". В 1990-х годах этот аспект был действительно важен.

В начале 2010-х годов, во время работы в команде Mid-Tier Cassandra в компании W.W. Grainger, я написал много кода на Java. Это было мое первое знакомство с Java корпоративного уровня. Лежащая в основе нашего сервисного уровня база данных (Apache Cassandra®) также была написана на Java, так что я был буквально брошен в пучину поиска исключений и нюансов Java 7.

Проведя бóльшую часть следующего десятилетия в качестве DBA Cassandra, в 2021 году я наконец вернулся к (почти) полноценному написанию кода на Java. Это был глоток свежего воздуха. Мне нравилось, как сильно развивалась Java вплоть до 17-й версии. К тому времени я уже написал две книги о базах данных NoSQL, и чем больше я работал с Java, тем больше мне хотелось написать о ней книгу.

Сегодня в мире существует огромное количество приложений на языке Java. Он работает на миллиардах устройств и обеспечивает работу всего — от видеоигр до веб-сайтов электронной коммерции, приносящих миллиарды долларов. Невозможно отрицать, что знание кода на Java стало ценным навыком.

Сегодня одним из самых больших препятствий для начинающих разработчиков (помимо выбора языка) является установка и настройка среды разработки. В противоположность этому, многие распространенные в 1980-х годах компьютеры позволяли очень легко начать программировать. Пользователям часто хватало всего нескольких нажатий клавиш, чтобы попасть в среду разработки программного обеспечения.

Некоторые первые домашние компьютеры (например, Tandy Color Computers и Apple II) сразу после загрузки выводили пользователя на экран с приглашением к программированию на BASIC. Эти первые компьютеры по умолчанию просили, чтобы их запрограммировали! С другой стороны, современные компьютеры (осо-

бенно телефоны и планшеты) так уже не делают. Построение подходящей среды программирования на современных компьютерах часто оказывается непростой задачей. Именно поэтому в первой главе этой книги мы уделим время установке и настройке среды разработки Eclipse.

Однако сделать компьютеры снова программируемыми — это не просто преодоление технических препятствий. Это изменение культуры. Нужно посмотреть на свое устройство и вместо вопроса *"Что оно может сделать для меня?"* спросить *"Что я могу сделать с ним?"*. Именно такой образ мышления ведет к тому, что обучение будет продолжаться всю жизнь.

В конечном итоге я написал эту книгу для того, чтобы помочь разработчикам программного обеспечения сгладить кривую обучения. Я считаю, что язык Java как нельзя лучше подходит для этого.

Эта книга построена таким образом, чтобы постепенно знакомить вас с различными аспектами написания кода на Java, причем каждая глава основывается на ранее пройденных уроках.

**Глава 1 "Знакомство с Java".** В этой главе содержится простое ознакомление с Java. В ней также рассказывается о настройке среды разработки, включая установку таких инструментов, как менеджер зависимостей и IDE. В этой главе также рассматривается объектно-ориентированное программирование (ООП) и представлены новые возможности Java 21.

**Глава 2 "Фундаментальные структуры программирования".** В этой главе рассматриваются некоторые основные строительные блоки синтаксиса Java. Она начинается с обязательной программы "hello world" и переходит к чтению вводимых данных, обработке ошибок и управлению логическим потоком программы. После изучения основ мы переходим к работе с файлами и специфическим для Java способам создания методов и конструкторов.

**Глава 3 "Строки, символы и регулярные выражения".** В этой главе рассматриваются различные способы работы с текстовыми данными и их обработки. Начав с простого примера с использованием символов ASCII, мы перейдем к рассмотрению некоторых более продвинутых методов, входящих в класс `String`. В завершение главы мы познакомимся с регулярными выражениями и покажем, как использовать их в практических примерах.

**Глава 4 "Массивы, коллекции и записи".** Эта глава знакомит читателя с различными структурами, которые могут использоваться для хранения данных в памяти. Уделяя внимание различным концепциям и случаям использования массивов, списков, множеств и словарей, в этой главе также рассматриваются записи и вводятся упорядоченные коллекции (новинка в Java 21).

**Глава 5 "Арифметические операции".** В этой главе вы узнаете, как компьютеры выполняют арифметические действия на примитивных уровнях, включая разницу между операциями с целыми числами и плавающей точкой. В ней также рассмотрена детерминированная сущность арифметики для небольшого введения в юнит-тестирование.

**Глава 6 "Общие структуры данных".** В этой главе читатель познакомится с процессом использования Java для создания таких структур данных, как стеки, очереди и различные виды связанных списков. В ней также рассматривается построение двоичного дерева и выполнение простого поиска данных.

**Глава 7 "Работа с базами данных".** В этой главе рассказывается о том, как создавать Java-приложения, хранящие данные в базах данных. В ней также рассказывается о базах данных PostgreSQL и Apache Cassandra®, демонстрируется построение простых моделей данных и выполнение распространенных команд CQL и SQL.

**Глава 8 "Веб-приложения".** В этой главе рассматривается создание веб-сервисов restful и full-stack веб-приложений на Java. В ней также представлены фреймворки Spring и Vaadin, и показано, как использовать их для создания полнофункциональных веб-приложений.

**Глава 9 "Графика в Java".** В этой главе на примерах рассматриваются вопросы отображения графики и анимации. В ней также рассматривается создание классической аркадной игры на Java.

**Глава 10 "Завершающий проект на Java".** Эта глава является кульминацией многих тем, рассмотренных в предыдущих главах, и показывает, как использовать их для создания приложения о фильмах. Векторный поиск становится последней новой темой, предоставляя читателям простой способ создания сервиса рекомендаций фильмов.

## Пакет кодов и цветные изображения

Перейдите по ссылке, чтобы скачать пакет кода и цветные изображения книги:

**https://rebrand.ly/nkskce0**

Пакет кода для книги также размещен на GitHub по адресу:

**https://github.com/bpbpublications/Code-with-Java-21**

В случае обновления кода, он будет обновляться в существующем репозитории GitHub.

Пакеты кода из нашего богатого каталога книг и видео доступны по адресу:

**https://github.com/bpbpublications**

Ознакомьтесь с ними!

# Глава 1

# Знакомство с Java

## Введение

Добро пожаловать в "Программируем на Java"! Независимо от того, новичок вы или опытный программист, эта книга поможет вам понять и эффективно использовать один из самых распространенных языков программирования в мире. Помимо фундаментальных аспектов Java, мы обсудим новые возможности, представленные в версии 21, и покажем, как правильно их использовать.

Java 21 является релизом Long-Term Support (LTS) с поддержкой до 2031 года. Эта книга призвана не только помочь вам в обучении, но и стать удобным справочником на длительный срок. Она содержит код, который поможет вам разобраться с каждым примером и стать успешным программистом.

> **Примечание.** Термины "программист", "разработчик программного обеспечения" и "кодер" часто используются как взаимозаменяемые.

## Структура

В этой главе мы обсудим следующие темы.

- Преимущества создания приложений на Java.
- Различные компоненты среды разработки Java.
- Общие инструменты, используемые для дополнения процесса разработки на Java.
- Принципы объектно-ориентированного программирования.
- Новые возможности Java 21.

## Цели

Цель этой книги — вдохновить вас на создание следующего поколения Java-приложений. В этой главе мы рассмотрим язык Java на высоком уровне, стремясь

предоставить достаточно подробную информацию для начала работы. К концу главы мы поймем, что отличает Java от других языков и как использовать его для написания мощных приложений.

# Почему стоит программировать на Java?

Java распространена повсеместно; она работает на миллиардах устройств по всему миру. Ее также используют предприятия из списка Fortune 500 для создания сервисов и приложений, которые помогают им зарабатывать миллиарды долларов каждый год. Нет необходимости говорить о том, что спрос на Java-разработчиков высок и, скорее всего, сохранится еще долгое время.

Существует несколько типов машин, способных работать с Java, включая (но не ограничиваясь) следующие:

- персональные компьютеры (ПК) для домашнего и рабочего использования;
- мобильные устройства;
- игровые консоли;
- встраиваемые устройства.

Такие основные свойства Java, как независимость от платформы, универсальность и безопасность, сделали его одним из самых популярных языков программирования в мире. Кроме того, при работе с ним легко получить помощь, поскольку учебные материалы по Java можно быстро найти на YouTube, LinkedIn и многих других сайтах.

Независимо от того, хотите ли вы научиться программировать в качестве хобби или получить навык, способный привести к успешной карьере, знание Java — это отличный опыт, которым нужно обладать.

# Настройка среды

Прежде чем приступить к написанию программ на Java, необходимо убедиться, что наша среда правильно создана и настроена. Вот что нам понадобится для успешной работы:

- компьютер под управлением Windows, Linux или MacOS;
- среда выполнения Java (Java Runtime Environment, JRE);
- интегрированная среда разработки (Integrated Development Environment, IDE);
- менеджер зависимостей Java;
- платформа контроля кода.

Хотя в этой книге сделаны некоторые поправки для только начинающих изучать Java, она предназначена и для тех, кто имеет, по крайней мере, средний уровень

общего знания программирования. И хотя в книге представлен обзор конфигурирования среды разработки, исчерпывающее описание всех возможных конфигураций выходит за рамки этой книги. Предполагается, что читатель установит и настроит те необходимые инструменты, которые ему наиболее знакомы.

## Операционная система

Одно из главных преимуществ Java заключается в том, что она легко переносима. Это означает, что один и тот же Java-код может работать под Windows, Linux или MacOS без каких-либо изменений. Аналогично, не имеет значения, на платформе какой операционной системы (ОС) написан Java-код. Как программисту, вам важно хорошо знать свою ОС и понимать ее нюансы и отличия от других ОС, когда это необходимо.

Например, важно помнить, что Windows не заботится о заглавных символах в именах файлов, в то время как для Linux и MacOS это важно. Windows также отличается от Linux и MacOS окончаниями строк в файлах. Все это может создать проблемы при создании работающих с файлами приложений и другими аспектами на уровне ОС.

## Java Runtime Environment

Еще одна часть среды разработки, которая необходима для работы с Java, — это *среда выполнения Java* (Java Runtime Environment, JRE). Этот пакет предоставляет все доступные библиотеки, необходимые для выполнения вашего Java-кода. В данной книге рассматривается Java 21 — версия Java, которая должна быть установлена для правильной работы примеров из этой книги.

## Java Development Kit

Важно также помнить, что доступны для загрузки как JRE, так и *Java Development Kit (JDK)*. В то время как JRE предоставляет полную среду для запуска Java-программ, комплект разработчика JDK содержит как JRE, так и дополнительные инструменты для разработчиков, позволяющие создавать и настраивать Java-программы. Поскольку нам понадобятся дополнительные средства разработки, JDK необходим для выполнения представленных в этой книге примеров.

## Поставщики и редакции JDK

Существует несколько компаний-разработчиков JDK, в том числе Microsoft, Oracle и IBM. Их сборки JDK обычно предназначены для корпоративного использования, и большинство из них требуют платной лицензии или контракта.

Многие производители также выпускают различные редакции в зависимости от целей использования и базовых инфраструктур.

- **Micro Edition.** Уменьшенная сборка JDK, предназначенная для встраиваемых систем и других устройств с небольшим количеством вычислительных ресурсов.
- **Standard Edition.** Сборка среднего уровня, ориентированная на машины разработчиков и аппаратное обеспечение класса рабочей станции.
- **Enterprise Edition.** Полноценная сборка, предназначенная для предприятий и высокопроизводительных систем.

В этой книге мы будем использовать *OpenJDK*, бесплатную версию Java Development Kit — Standard Edition с открытым исходным кодом. Последние версии OpenJDK (версия 21) для различных операционных систем и архитектур можно найти на сайте **https://jdk.java.net/21/**.

# Установка

Вы можете пропустить этот шаг, если ваша IDE поставляется с JDK. В противном случае OpenJDK загружается в виде сжатого файла; обычно это tar- или ZIP-файл. Расположение файла, в которое его нужно распаковать, зависит от операционной системы. Тем не менее, его нужно поместить в место, доступное в среде.

> **Примечание.** Для установки JDK вам, скорее всего, понадобятся права администратора или суперпользователя[1].

Вы можете запустить эту команду, чтобы проверить установку JDK или узнать версию, которая у вас установлена:

```
java -version
```

Если JRE или JDK уже установлены, вы увидите результат, похожий на этот:

```
openjdk version "21-ea" 2023-09-19
OpenJDK Runtime Environment (build 21-ea+16-1326)
OpenJDK 64-Bit Server VM (build 21-ea+16-1326, mixed mode, sharing)
```

Так как в этой книге рассматривается Java 21, основная версия, указанная в списке, должна быть 21.

## Windows

Стандартное расположение JDK — в каталоге Program Files. Установите переменную окружения `JAVA_HOME` для этого пути. Кроме того, вам может понадобиться добавить ее в переменную окружения `PATH`.

[1] Специальная учетная запись пользователя, используемая для администрирования системы. — *Прим. ред.*

### MacOS

Аналогичный подход можно использовать и на Mac. После распаковки tar-файла добавьте его местоположение в переменную окружения PATH (в файле .bashrc).

Кроме того, для OpenJDK существует формула[2] Homebrew, которая берет на себя все заботы по установке и настройке переменных окружения. Ее можно запустить из терминала следующим образом:

```
brew install openjdk@21
```

### Linux

Аналогичным образом, в Linux файл tar может быть распакован и указан в переменной PATH в файле .bashrc. Кроме того, поставляемые менеджеры пакетов для Linux также могут получить доступ к необходимым репозиториям OpenJDK. Конкретная используемая команда зависит от разновидности Linux.

Если вы работаете на производной Red Hat Linux (например, Fedora, CentOS), OpenJDK может быть установлен с помощью менеджера пакетов yum:

```
sudo yum install java-21-openjdk
```

Кроме того, для тех, кто работает на производном от Debian дистрибутиве Linux (например, Ubuntu, Cinnamon), OpenJDK может быть установлен с помощью менеджера пакетов apt:

```
sudo apt install openjdk-21-jdk
```

Важно отметить, что если у вас установлено несколько JDK/JRE, вам может потребоваться изменить версию по умолчанию. Это можно сделать, обновив системные альтернативы:

```
sudo update-alternatives --config java
```

## Управление версиями

У некоторых разработчиков на рабочих станциях может быть установлено несколько JRE/JDK. Настоятельно рекомендуется использовать менеджер окружения Java. Например, пользователи MacOS и Linux могут установить такой инструмент, как *jEnv*, зайдя на сайт: **https://www.jenv.be/**.

Существует также jEnv для Windows, доступный в следующем репозитории GitHub: **https://github.com/FelixSelter/JEnv-for-Windows**.

---

[2] Инструкция по установке конкретного программного приложения. С ее помощью можно установить различные версии Java на MacOS, например, последнюю или конкретную версию OpenJDK. — *Прим. ред.*

# Интегрированная среда разработки

Прежде чем вы сможете писать код на любом языке (включая Java), вам понадобится специальный инструмент. Как минимум, необходим текстовый редактор вроде Notepad, Sublime или Vim. Однако большинство разработчиков предпочитают использовать интегрированную среду разработки (IDE).

IDE — это больше, чем просто редактор кода. IDE предоставляет программисту доступ к инструментам, призванным облегчить написание кода. Обычно она позволяет легко и быстро собирать и компилировать код, взаимодействовать с системой контроля версий, выбирать другой JDK, устанавливать определенные переменные окружения или библиотеки. Вот краткий список популярных IDE:

- Eclipse;
- IntelliJ IDEA;
- NetBeans;
- VS Code.

Большинство разработчиков очень трепетно относятся к используемой ими IDE. Примеры, приведенные в этой книге, безусловно, будут работать в любой IDE, поэтому вы можете использовать ту, которая вам больше нравится.

> **Примечание.** Как уже упоминалось, некоторые IDE (например, Eclipse) поставляются в комплекте с JDK. Отсутствие необходимости возиться с дополнительной установкой — привлекательный вариант для многих разработчиков.

# Управление зависимостями в Java

Управление зависимостями библиотек может быть сложным в любом языке. В экосистеме Java есть инструменты, которые помогают в этом процессе. Два наиболее популярных инструмента управления зависимостями — *Gradle* и *Maven*.

Примеры в этой книге и в соответствующих репозиториях GitHub были созданы с помощью Maven. Вы можете выбрать и свободно использовать любой из них.

# Система контроля версий

*Git* — самый распространенный в мире инструмент контроля версий, он будет использоваться и в этой книге. Все примеры из этой книги можно найти в репозитории GitHub.

Читателям, которые хотят в полной мере использовать все доступные ресурсы для обучения, рекомендуется создать учетную запись на сайте **https://github.com**. Кроме того, рекомендуется установить Git локально для доступа к различным командам.

**Примечание.** Некоторые IDE имеют плагин или другую интеграцию с Git, что избавляет от необходимости устанавливать его отдельно.

## Установка Git

Если вы работаете на Mac или Windows PC, на GitHub есть пакеты автоматической установки, доступные по адресу **https://github.com/git-guides/install-git**.

Пользователи Mac также могут установить Git через Homebrew, выполнив следующие действия:

```
brew install git
```

Пользователи Linux могут установить Git с помощью менеджера пакетов своего дистрибутива. Если вы работаете на Linux, производном от Red Hat, OpenJDK можно установить с помощью менеджера пакетов `dnf`:

```
sudo dnf install git-all
```

Или с помощью менеджера пакетов `yum`:

```
sudo yum install git-all
```

Кроме того, для тех, кто работает на Linux, производном от Debian (например, Ubuntu, Mint), OpenJDK может быть установлен с помощью менеджера пакетов `apt`:

```
sudo apt-get install git-all
```

Выполните следующую команду, чтобы проверить установку Git:

```
git --version
```

Если Git установлен, результат должен выглядеть примерно так:

```
git version 2.32.0
```

## Объектно-ориентированное программирование

Мы не можем говорить о Java, не обсудив предварительно *объектно-ориентированное программирование* (*ООП*). В этом разделе мы познакомимся с четырьмя основными принципами ООП. Кроме того, рассмотрим некоторые преимущества и недостатки ООП, а также то, как эти принципы будут помогать нам в процессе изучения глав.

По сути, ООП — это парадигма, в которой проектирование программного обеспечения определяется данными и тем, как они классифицируются в виде объектов. В отличие от этого, программирование в не использующих ООП языках обычно основывается на функциях и логике (*Gillis, Lewis 2021*).

Основным строительным блоком ООП является класс. Класс — это, по сути, шаблон для объекта, который мы хотим создать и использовать в дальнейшем. В Java каждый класс обычно находится в собственном файле.

Классы обычно содержат методы и свойства. Методы — это отдельные блоки кода, которые обычно предназначены для выполнения определенной функции. Свойства — это переменные, которые можно читать и изменять только путем вызова специальных методов класса.

Ниже приведены четыре основных принципа ООП:

- инкапсуляция;
- наследование;
- абстракция;
- полиморфизм.

Давайте вкратце рассмотрим каждый из них.

## Инкапсуляция

В Java есть понятие области видимости. Все переменные имеют одну из трех классификаций областей видимости.

- **Private.** Переменная, которая может быть изменена только кодом внутри своего класса.
- **Protected.** Переменная, которая может быть изменена кодом внутри собственного пакета.
- **Public.** Переменная, которая может быть изменена кодом из любого места.

  **Примечание.** Если переменная объявлена без одной из классификаций областей видимости, Java предполагает, что она имеет область видимости protected, и это означает, что доступ к ней разрешен только из ее текущего пакета.

Идея *инкапсуляции* заключается в том, что данные и методы объекта содержатся в едином блоке. Свойства объекта — это переменные с закрытой областью видимости private, которые не могут быть напрямую изменены или прочитаны извне объекта. Доступ к ним осуществляется с помощью специально разработанных общедоступных методов, известных как геттеры и сеттеры. *Геттер* используется, когда другой метод хочет прочитать или получить значение свойства. Аналогично, *сеттер* вызывается, когда другой метод хочет изменить или установить значение свойства.

С точки зрения программиста, это позволяет нам полностью контролировать доступ к свойствам объекта. Такой подход может быть полезен при поиске и отладке неисправностей. Если мы хотим узнать, как или где изменяется свойство, нужно просто посмотреть на его метод-сеттер и поискать его в предполагаемых классах.

По сути, инкапсуляция — это подход к разработке, который накладывает ограничения на доступ, чтобы обеспечить безопасное обращение к свойствам объекта.

## Наследование

Принцип *наследования* позволяет классам быть производными от других. Иногда это называют отношениями "родитель | ребенок", когда *дочерний класс* (он же производный класс) наследует методы и свойства от *родительского класса* (он же базовый или суперкласс).

Например, интернет-магазин (иногда его называют e-tailer) хочет продавать товары онлайн. Эти товары могут существенно отличаться друг от друга, например, это могут быть фильмы, книги, закуски или велосипеды. Свойства каждого из этих товаров будут разными, но для целей продажи в интернете они должны иметь некоторые общие характеристики, такие как название, категория и цена.

Для этого мы можем создать базовый класс `Product`, содержащий свойства `name`, `category` и `price` (а также соответствующие методы getter/setter). Пример класса `Product` (с публичными методами доступа к свойствам `name`, `category` и `price`) показан здесь:

```
class Product {
    private String name;
    private String category;
    private BigDecimal price;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getCategory() {
        return this.category;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    public BigDecimal getPrice() {
        return this.price;
    }
```

```
    public void setPrice(BigDecimal price) {
        this.price = price;
    }
}
```

Каждый тип продукта может наследовать класс `Product` (с помощью ключевого слова `extends`). Здесь мы покажем пример для фильмов:

```
public class Movie extends Product {
    private String title;
    private int lengthInMinutes;

    public String getTitle() {
        return this.title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public int getLengthInMinutes() {
        return this.category;
    }

    public void setLengthInMinutes(int category) {
        this.category = category;
    }
}
```

Поскольку класс `Movie` наследуется от базового класса `Product`, все объекты класса `Movie` будут иметь свойства обоих классов. Поскольку в нашем интернет-магазине представлено несколько различных типов товаров, это избавит нас от необходимости встраивать базовые свойства в каждый отдельный класс.

## Абстракция

*Абстракция* — это идея скрыть детали реализации, раскрыв только основные методы класса. Реализация абстракции в Java осуществляется с помощью интерфейсов и абстрактных классов. Они определяют методы, которые должны быть реализованы наследуемым подклассом, но при этом скрывают детали реализации.

Давайте вернемся к нашему примеру с интернет-магазином, где мы ранее говорили о создании базового класса `Product`. Есть дополнительные преимущества в том, что

все наши классы типов товаров наследуются от базового или абстрактного класса. Одно из них заключается в том, что многие различные типы товаров могут рассматриваться как объекты `Product`, поскольку они наследуются от класса `Product`.

Неважно, является ли некий товар на самом деле объектом класса `Movie` или `Book`; он все равно будет обладать свойствами класса `Product`, чтобы его можно было продавать онлайн. Самое приятное в этом то, что мы можем передать класс `Book` другой команде разработчиков. Они могли бы просто сосредоточиться на использовании класса `Book` для выполнения специфических для книг действий, без необходимости разбираться в том, как их книги добавляются в корзину покупателя; этим должен заниматься класс `Product`.

Ниже приведен пример, показывающий, как абстракция помогает добавить объект `Movie` в корзину покупателя:

```
private User user;

public void movieShopping() {
    Movie movie = getMovieByTitle("The Empire Strikes Back");
    addToCart(movie, 1);
}

public void addToCart(Product product, int qty) {
    cart.add(user.getId, product, qty);
}
```

Как видите, объект типа `Movie` может быть передан в метод `addToCart()`, поскольку он является наследником класса `Product`. Это сделано для того, чтобы показать на простом примере преимущества абстракции данных и то, как ее можно использовать.

Теперь предположим, что наш интернет-магазин является частью крупной розничной компании, в которой работают сотни разработчиков. В предыдущем примере мы использовали абстракцию данных для разделения обязанностей между различными командами разработчиков, которые работают с разными типами товаров.

Если мы посмотрим в код метода `addToCart()`, то увидим, что он просто вызывает метод `add()` объекта `cart`. Давайте посмотрим, что делает метод `add()`:

```
public Cart() {
    CartDAL cartDAL;

    public void add(UUID userId, Product product, int qty) {
        CartLineItem line = new CartLineItem();
        line.userId = userId;
        line.productId = product.getId;
        line.qty = qty;

        cartDAL.save(userId, product, qty);
    }
}
```

В этом случае разработчикам нашего сайта нужно беспокоиться только о вызове метода `add()` объекта `cart`. Им не нужно беспокоиться о том, что делает этот метод. Однако разработчики команды Cart Team создают и поддерживают службу Cart Service. Внутри службы `Cart Service` метод `add()` определен таким образом, что принимает свои параметры, создает объект `CartLineItem`, устанавливает его свойства, а затем сохраняет его в слое доступа к данным (Data Access Layer, DAL). Это пример абстрагирования метода, поскольку команда Cart Team абстрагировала детали сохранения данных в корзине от команды Website Team.

## Полиморфизм

Понятие *полиморфизм* происходит от греческого выражения "одна вещь — много форм". По сути, это то, что позволяет одним методам и объектам приобретать характеристики других. В Java существует два вида полиморфизма: динамический и статический.

### Статический полиморфизм

*Статический полиморфизм* — это концепция времени компиляции, часто встречающаяся при *перегрузке метода*. Теперь вы можете задать простой вопрос: "Что значит перегрузить метод? И зачем нам это нужно?".

Это одна из тех возможностей, которые демонстрируют универсальность Java. Предположим, что мы создаем математическую библиотеку. В рамках этой библиотеки мы хотим иметь возможность складывать два целых числа, поэтому мы создаем метод, который выглядит следующим образом:

```
public int add (int num1, int num2) {
    return num1 + num2;
}
```

Достаточно просто, верно?

А что, если два числа, которые мы хотим сложить, являются `BigDecimals`? В этом случае метод `add` не будет работать из-за несовпадения типов. Решение — перегрузить метод `add`, написав другой метод `add`, который работает с `BigDecimals`:

```
public BigDecimal add (BigDecimal num1, BigDecimal num2) {
    return num1.add(num2);
}
```

> **Примечание.** Числа типа `BigDecimal` не работают со стандартным оператором `plus` (+), поэтому нам нужно использовать их метод `add`. Мы подробно рассмотрим это в *главе 5 "Арифметические операции"*.

Если понадобится, мы можем создать еще один. Может быть, наш пользователь хочет иметь возможность складывать два числа типа `double`?

Мы можем решить эту задачу, перегрузив `add` еще раз:

```
public double add (double num1, double num2) {
    return num1 + num2;
}
```

Следуя концепции полиморфизма, перегрузка позволяет нам предоставлять пользователям простые методы `add`, не слишком заботясь о том, с какими числовыми типами они работают. Они просто вызывают метод `add` с двумя подходящими типами, а наш класс делает все остальное.

## Динамический полиморфизм

Полиморфизм проявляется в *динамическом контексте* во время выполнения программы при переопределении наследованного метода. Проще говоря, *переопределение* — это написание нового метода с тем же именем, что и у метода из класса, от которого наследуемся, чтобы новый метод имел приоритет. Рассмотрим пример.

В качестве базового класса будет использоваться автомобиль. У всех автомобилей есть двери, и все двери требуют средств, с помощью которых их можно открыть. Поэтому наш базовый класс `Car` будет иметь метод для открытия дверей. Поскольку все объекты `car` наследуют базовый класс `Car`, все они получают метод `openDoor` по умолчанию:

```
public void openDoor() {
    door.open("outward");
}
```

Теперь предположим, что мы создаем класс `McLarenP1`. Поскольку двери McLaren P1 открываются вверх, а не наружу (как у большинства автомобилей), нам понадобится другой метод `openDoor`. Поэтому мы переопределим метод `openDoor` базового класса `Car` и напишем свой собственный, локально в классе `McLarenP1`:

```
public void openDoor() {
    door.open("upward");
}
```

Может быть, создадим класс `DeLorean`? В этом случае нам также нужно будет переопределить `openDoor`, но немного по-другому:

```
public void openDoor() {
    door.open("gull-wing");
}
```

А что если мы создадим класс `FordFusion`? В этом случае метод `openDoor`, используемый по умолчанию, подойдет как нельзя лучше.

Суть в том, что следование полиморфизму в Java обеспечивает возможность как перегружать, так и переопределять методы. Эти возможности позволяют *одной вещи принимать много форм*.

## Преимущества ООП

Мы уже обсуждали некоторые преимущества объектно-ориентированного программирования, но давайте укажем их.

- **Повторное использование (переиспользование) кода.** Часто используемые методы не нужно переписывать в каждом классе, в котором они необходимы. Даже классы можно повторно использовать в различных приложениях.
- **Модульность.** Принципы ООП поощряют модульность, позволяя разбивать сложные проблемы на более мелкие задачи.
- **Сотрудничество.** Разные команды могут работать над одним и тем же или смежными проектами совместно, поскольку различные модули и классы часто могут быть созданы независимо друг от друга.

ООП-подход к программированию имеет явные преимущества перед традиционным подходом, основанным на логике или функциях. Мы будем применять этот подход на практике в следующих главах.

# Что нового в Java 21?

За последнее время в Java появилось несколько невероятных возможностей. В частности, в Java 21 появились три новые возможности, которые мы рассмотрим:

- виртуальные потоки;
- упорядоченные коллекции;
- строковые шаблоны.

## Виртуальные потоки

Вероятно, самая обсуждаемая новая функция Java 21 — это *виртуальные потоки*. Виртуальная многопоточность, реализованная в Java 19 в качестве предварительной функции, представляет собой значительный сдвиг в том, как обрабатывается параллелизм в Java. Разработчики теперь работают с виртуальными потоками в своем коде вместо обычных потоков операционной системы.

Виртуальный поток взаимодействует с виртуальной машиной Java (JVM), которая может назначить виртуальный поток своему собственному процессу или в конечном итоге разделить процесс с другим виртуальным потоком. Таким образом, виртуальные потоки представляют собой (*Tyson 2022*) абстрактный слой для создания потоков, предоставляя JVM возможность управлять доступными ресурсами на уровне ОС.

Например, раньше класс Java-приложения может реализовать интерфейс `Runnable` для одновременного запуска определенных частей приложения.

Этот класс также может иметь публичный метод для запуска потока:

```
public class MyApplication implements Runnable {
    private Thread appThread;

    public void startAppThread() {
        appThread = new Thread(this);
        appThread.start();
    }
// Далее идет программный код
```

С виртуальными потоками тот же класс и метод будут выглядеть следующим образом:

```
public class MyApplication implements Runnable {
    public void startAppThread() {
        Thread.startVirtualThread(this);
    }
}
```

Это предусмотрено специально и позволяет легко модифицировать существующее Java-приложение для использования преимуществ виртуальных потоков. Применение виртуальных потоков также приводит к повышению производительности кода за счет значительного сокращения создания отдельных процессов на уровне ОС.

Важно отметить, что существуют некоторые ограничения (*Tyson 2022*), касающиеся эффективного использования виртуальных потоков:

- для управления количеством одновременных потоков следует использовать семафор, а не пул потоков. JVM управляет пулом потоков;
- все виртуальные потоки считаются потоками-демонами, что означает, что вызывающее приложение не может быть закрыто, пока они запущены;
- приоритет виртуальных потоков не может быть изменен.

В последующих главах мы будем использовать виртуальные потоки.

## Упорядоченные коллекции

Работа с типами коллекций Java (списками, множествами и словарями) стала проще. Коллекции Java теперь имеют то, что называется (*Parlog N. 2023*[3]) *порядком встречи* (encounter order). Это стало возможным благодаря появлению интерфейса `SequencedCollection`. По сути, теперь коллекции будут отслеживать порядок, в котором были добавлены их элементы. Это позволяет реализовать эти новые методы на всех списках и некоторых типах коллекций (при условии, что элементы коллекции имеют тип `E`):

- `addFirst(element)`
- `addLast(element)`

[3] Здесь и далее подобные ссылки развернуто представлены в приложении I — *Прим. ред.*

- `<E> getFirst()`
- `<E> getLast()`
- `<E> removeFirst()`
- `<E> removeLast()`
- `SequencedCollection<E> reversed()`

Коллекции `Map` также имеют аналогичные новые методы (предполагаем, что `map` имеет записи с ключом `K` и значением `V`):

- `V putFirst(K, V)`
- `V putLast(K, V)`
- `Entry<K, V> firstEntry()`
- `Entry<K, V> lastEntry()`
- `Entry<K, V> pollFirstEntry()`
- `Entry<K, V> pollLastEntry()`
- `SequencedMap<K, V> reversed()`
- `SequencedSet<K> sequencedKeySet()`
- `SequencedCollection<V> sequencedValues()`
- `SequencedSet<Entry<K, V>> sequencedEntrySet()`

Более подробно мы рассмотрим упорядоченные коллекции в *главе 4 "Массивы, коллекции и записи"*.

# Строковые шаблоны

Созданные с целью решения проблемы сложной конкатенации строк, *строковые шаблоны* уже какое-то время существуют в других языках. По сути, строковые шаблоны позволяют вводить переменные и выражения в предварительно созданные строки, что значительно упрощает составление строк во время выполнения программы.

Рассмотрим ситуацию, когда мы обрабатываем пользовательский ввод, например, когда пользователь входит в банковский или платежный аккаунт. Если бы мы хотели поприветствовать его и отобразить его текущий баланс, код выглядел бы примерно так:

```
private String welcomeUser (User user) {
    String returnVal = "Hello " + user.getFirstName() + ",
    your balance is $" + user.getBalance();
    return returnVal;
}
```

Или с помощью класса `StringBuilder`:

```
private String welcomeUser (User user) {
    String returnVal = new StringBuilder()
```

```
        .append("Hello ")
        .append(user.getFirstName())
        .append(", your balance is $")
        .append(user.getBalance())
        .toString();
    return returnVal;
}
```

Но гораздо проще использовать строковые шаблоны:

```
private String welcomeUser (User user) {
    String returnVal = STR."Hello \{user.getFirstName()},
        your balance is $\{user.getBalance()}";
    return returnVal;
}
```

Это значительно упрощает подход к обработке данных и построению строк во время выполнения программы. Более подробно мы рассмотрим строковые шаблоны в *главе 3 "Строки, символы и регулярные выражения"*.

# Заключение

Одним словом, с Java 21 есть чему порадоваться. В следующих главах опытные Java-программисты найдут более простые способы взаимодействия с мощными приложениями и их создания. А те, кто только начинает изучать Java, приготовьтесь сделать первые шаги в большой мир!

# Важно помнить

- Убедитесь, что вы загрузили JDK, а не JRE.
- Рекомендуется создать аккаунт на GitHub, если у вас его еще нет.
- Четыре основных принципа ООП:
  - инкапсуляция;
  - наследование;
  - абстракция;
  - полиморфизм.
- Новые возможности Java 21:
  - виртуальные потоки;
  - упорядоченные коллекции;
  - строковые шаблоны.

Глава 2

# Фундаментальные структуры программирования

## Введение

Теперь, когда мы поговорили о Java и Java 21, рассмотрим несколько простых примеров. В этой главе мы сосредоточимся на работе с основами программирования, такими как использование переменных, создание методов класса, работа с файлами и управление потоком наших программ.

Также мы рассмотрим некоторые специфические для Java аспекты работы с объектами, классами и конструкторами. Кроме того, мы будем указывать на некоторые принципы ООП, когда они будут встречаться.

Кроме того, обратим внимание на моменты, которые помогут закрепить лучшие практики написания кода на Java. В этой книге мы будем использовать стиль Кернигана и Ричи. Он также известен как "Единственный Правильный Скобочный Стиль" (One True Brace Style) или 1TBS.

## Структура

В этой главе мы обсудим следующие темы.

- Начало работы.
- Maven.
- Переменные и форматирование.
- Чтение входных данных.
- Обработка ошибок.
- Управление потоком с помощью условных операторов и циклов.
- Файлы.
- Методы и конструкторы.

# Цели

Цели этой главы заключаются в следующем:

- помочь вам научиться писать простой код на Java и запускать его в IDE;
- создать прочный фундамент знаний по основному синтаксису Java;
- понять, как Java работает с базовой операционной системой.

# Начало работы

Давайте создадим новый пустой проект. Шаги будут немного отличаться в зависимости от вашей IDE и выбора менеджера зависимостей. В этой книге мы рассмотрим примеры использования менеджера зависимостей Maven.

Если его еще нет, настройте вашу IDE, добавив Java 21 в качестве доступной JRE. В Eclipse IDE это можно сделать с помощью следующих меню и диалоговых окон: **Window | Preferences | Java | Installed JREs**.

# Maven

В вашей IDE создайте новый проект Maven. Убедитесь, что в файле pom.xml заданы следующие идентификатор группы и артефакт:

```
<groupId>com.codewithjava21</groupId>
<artifactId>chapterexercises</artifactId>
```

Также добавьте следующие свойства, чтобы использовалась соответствующая версия Java:

```
<properties>
    <java.version>21</java.version>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
</properties>
```

> **Примечание.** В зависимости от версии JDK, используемой IDE по умолчанию, нам может потребоваться установить Java 21 в качестве системной библиотеки JRE в свойствах сборки нашего нового проекта.

## HelloWorld

Давайте создадим в нашем проекте новый Java-класс со следующими свойствами:

- `Name: HelloWorld`
- `Package: chapter2`
- `Чекбокс для public static void main(String[] args): Установлен`

**Примечание.** Почти в каждой книге по программированию присутствует обязательная программа "hello world". Ее смысл в том, чтобы показать новичкам основы построения и выполнения программ.

В зависимости от используемой IDE, новый класс уже будет содержать небольшое количество кода. Например, те, кто использует IDE Eclipse, должны увидеть что-то вроде этого:

```
package chapter2;
public class HelloWorld {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}
```

Перечислим несколько моментов, на которые стоит обратить внимание.

- Первое, что мы видим здесь, — это определение нашего пакета. *Пакет* (package) — это просто контейнер для Java-классов, связанных друг с другом.
- Между определениями пакета и класса мы будем импортировать дополнительные библиотеки. Этот класс простой, поэтому оператор `import` нам сейчас не понадобится.
- Наше определение класса устанавливает общедоступный класс `HelloWorld`. Весь последующий код должен быть заключен в фигурные скобки {}.
- Далее мы определяем наш метод `main`, который состоит из нескольких частей:
  - как и класс, он является общедоступным (`public`). Это означает, что его можно вызывать как внутри класса, так и за его пределами;
  - он объявлен как статический (`static`), что означает, что его можно вызывать извне класса, не требуя инстанцирования класса как объекта;
  - он не возвращает значения, поэтому тип возврата метода — `void`;
  - имя метода — `main`;
  - в качестве аргумента метод принимает массив строк. Это сделано специально и позволяет разработчику быстро принимать любые передаваемые параметры;
  - внутри метода `main` есть комментарий. Он начинается с сообщения `TODO`, чтобы напомнить нам, что мы должны *сделать* что-то позже. Любая строка в Java, начинающаяся с двух прямых косых черт `//`, является комментарием.

**Примечание.** Мы также можем заключить несколько строк в "блочный комментарий", используя прямую косую черту и звездочку `/*` в начале, а затем закрывая звездочкой и косой чертой `*/`.

Давайте начнем с удаления комментария и замены его строкой кода, которая что-то делает. Мы хотим вывести на экран приветствие `Hello world!`, и это можно сделать с помощью простой строки кода:

```
System.out.print("Hello world!");
```

Это вызывает метод `print` из библиотеки `System.out` в Java. В результате текст, заключенный в кавычки в списке параметров метода, выводится на терминал или консоль.

Теперь метод `main` должен выглядеть следующим образом:

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

В вашей IDE должна быть кнопка или другое средство для запуска программы. Выполнение этой программы приводит к выводу следующего результата:

```
Hello world!
```

> **Примечание.** Если программа не запускается, посмотрите на сообщения об ошибках, выдаваемые в консоли. Сообщения об ошибках носят описательный характер и обычно указывают на природу и место возникновения проблемы.

## Анонимные классы *main*

В Java 21 также появилась новая предварительная функция, известная как *анонимные классы main*, позволяющая облегчить выполнение данного упражнения для новичков в Java. Это означает, что наш метод `main` может выглядеть следующим образом:

```
public main() {
    System.out.print("Hello world!");
}
```

Это гораздо проще для понимания новичков!

Если вы хотите включить предварительные функции в вашем JDK, есть несколько способов (например, *Baeldung 2022)* сделать это. Хотя по умолчанию они отключены, их можно активировать в настройках проекта или Java-компилятора вашей IDE. Пользователи Maven также могут включить предварительные функции JDK с помощью плагина компилятора Maven в файле pom.xml:

```
<plugins>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
            <source>21</source>
            <target>21</target>
            <compilerArgs>
                --enable-preview
            </compilerArgs>
        </configuration>
    </plugin>
</plugins>
```

Вы также можете использовать флаг `enable-preview` при запуске Java из командной строки:

```
java --enable-preview HelloWorld.class
```

> **Примечание.** Предварительные функции часто не являются полноценными и могут давать непредсказуемые результаты. Их следует включать только опытным пользователям.

# Переменные и форматирование

Теперь давайте рассмотрим *переменные*. Переменные похожи на закладки; они указывают на конкретные данные, хранящиеся в оперативной памяти компьютера (RAM). Мы можем получить доступ к данным, хранящимся в переменной Java, при соблюдении следующих двух условий:

- мы знаем имя переменной;
- область видимости переменной позволяет нам получить к ней доступ.

Как уже говорилось в *главе 1 "Знакомство с Java"*, все переменные имеют одну из трех областей видимости: private, protected и public. Следуя правилам, определенным для каждой области видимости, мы можем получить доступ к данным переменной. Более подробно об областях видимости мы поговорим позже.

Все переменные имеют тип. Тип переменной сообщает Java (и программисту), какой вид данных хранится в ней. Сейчас мы остановимся на типе `String`. `String` ("строка") позволяет хранить текстовые данные в формате ASCII и UTF. Подробнее об этом мы поговорим в *главе 3 "Строки, символы и регулярные выражения"*. Главное, что нужно запомнить, — это то, что строка позволяет нам работать с текстовыми данными.

> **Примечание.** ASCII означает American Standard Code for Information Interchange (Американский стандартный код для обмена информацией), являвшийся стандартом кодировки символов для первых компьютеров. Впоследствии он был включен в гораздо более крупный и расширяемый стандарт UTF, который расшифровывается как Uniform Transformation Format. UTF был расширен, чтобы закодировать более 1 миллиона символов из множества письменных языков. Таблица преобразования символов ASCII/UTF представлена в *приложении 2 "Таблица преобразования UTF"*.

Важно понимать, что, хотя в Java существует несколько типов переменных, они всегда являются либо ссылочными, либо примитивными типами. *Примитивные типы* хранятся в памяти в виде своих буквенных значений, в то время как ссылочные типы являются по сути закладками на некое место в памяти. *Ссылочные типы* обычно представляют собой объекты, но могут быть и другими статическими классами. Список примитивных типов в Java приведен в табл. 2.1.

*Таблица 2.1. Список примитивных типов Java*

| Название | Описание |
|---|---|
| `boolean` | Однобитный тип данных, который имеет значение либо 'true', либо 'false'. |
| `Byte` | 8-битный тип, используемый для работы с двоичными данными |
| `Char` | 16-битный тип данных, используемый для хранения одного символа ASCII/UTF |
| `double` | 64-битный тип данных с плавающей точкой |
| `Float` | 32-битный тип данных с плавающей точкой |
| `Int` | 32-битный целочисленный тип |
| `Long` | 64-битный целочисленный тип |
| `Short` | 16-битный целочисленный тип |

Каждый примитивный тип также имеет свой собственный ссылочный тип, который иногда называют *классом-оберткой*. Эти классы-обертки предоставляют дополнительную функциональность, которая может быть полезна при преобразовании данных между типами. Существуют и дополнительные ссылочные типы, например тип `String`, который мы собираемся использовать.

Давайте сделаем дополнение к нашему классу `HelloWorld`. Добавим в наш класс новую переменную типа `String` под названием `firstName`, которая будет содержать наше собственное имя:

```
public class HelloWorld {
private static String firstName = "Aaron";
    public static void main(String[] args) {
```

> **Примечание.** Мы должны определить нашу переменную как статическую, потому что наш класс является статическим классом. Это означает, что методы нашего класса могут быть запущены без инстанцирования класса в качестве объекта.

Также добавим новый оператор `printf`:

```
System.out.print("Hello world!");
System.out.printf("Welcome to the world of Java, %s!", firstName);
```

Выполнение этого кода должно привести к следующему результату:

```
Hello world!Welcome to the world of Java, Aaron!
```

В чем же разница между `print` и `printf`? Оператор `print` просто печатает текстовую строку, переданную в него в качестве параметра. Но оператор `printf` позволяет нам добавлять правила форматирования для настройки вывода данных из переменных. Мы использовали правило форматирования `%s`, позволяющее выводить данные из переменных `String`.

Другой вопрос: "Почему наши текстовые строки были выведены в одну строку?". Ответ заключается в том, что мы не указали нашей программе выводить эти две строки на разных строках. Это можно сделать с помощью одного простого изменения: использовать оператор `println` вместо `print`:

```
System.out.println("Hello world!");
System.out.printf("Welcome to the world of Java, %s!", firstName);
```

Теперь, если мы запустим нашу программу, то увидим, что текст выводится на двух разных строках, как показано ниже:

```
Hello world!
Welcome to the world of Java, Aaron!
```

Давайте добавим в наш класс `HelloWorld` еще одну переменную, чтобы отслеживать возраст. Определим возраст как статическую `static` целочисленную переменную прямо под переменной `firstName`:

```
private static String firstName = "Aaron";
private static int age = 47;
```

Мы также хотим добавить возраст в выходные данные. Давайте сделаем это с помощью еще одного оператора `printf`. На этот раз мы используем спецификатор формата `%d`, чтобы вывести числовые целочисленные данные:

```
System.out.println("Hello world!");
System.out.printf("Welcome to the world of Java, %s!", firstName);
System.out.printf("age = %d. It's never too late to learn Java!", age);
```

Сохраним и снова запустим программу, которая должна показать следующий результат:

```
Hello world!
Welcome to the world of Java, Aaron!age = 47. It's never too late to learn Java!
```

Вывод двух последних операторов `printf` снова оказался в одной строке. Мы не можем использовать `println`, поскольку он не принимает спецификаторы формата или переменные.

Однако эту проблему можно решить, добавив в строку специальный символ новой строки. Символ новой строки — это два символа, обратный слеш `\` и буква `n`. Он выглядит как `\n`, и это то, что нам нужно, чтобы вывести текст на разных строках.

Давайте добавим его в конец текстовых строк для обоих операторов `printf`:

```
System.out.printf("Welcome to the world of Java, %s!\n", firstName);
System.out.printf("age = %d. It's never too late to learn Java!\n", age);
```

Помните наш предыдущий разговор о различиях в окончаниях строк в разных операционных системах? Учитывая это, пользователям ОС Windows, возможно, придется использовать `\r\n`, а не просто `\n`.

Выполнение кода приводит к следующему результату:

```
Hello world!
Welcome to the world of Java, Aaron!
age = 47. It's never too late to learn Java!
```

Вы можете спросить: "Почему так происходит?".

Клавиатуры современных компьютеров произошли от пишущих машинок, которые использовались в XX веке. Когда вы заканчивали печатать строку на пишущей машинке, вам нужно было сделать две вещи:

- переместить каретку в исходное положение в начале строки;
- переместить подачу бумаги на следующую строку.

Когда компьютеры начали взаимодействовать с большими работающими в центрах обработки данных принтерами, были созданы специальные управляющие коды для реализации возврата каретки и перевода строки (Carriage Returns and Line Feeds, CRLF). Эти управляющие коды до сих пор присутствуют в современных компьютерных языках и наборах кодировок символов. В некоторых языках их просто называют CRLF.

Другим способом решения этой проблемы было бы добавление специального разделителя строк из библиотеки `System`:

```
System.out.printf("Welcome to the world of Java, %s!" +
System.lineSeparator(), firstName);
System.out.printf("age = %d. It's never too late to learn Java!" +
System.lineSeparator(), age);
```

Важно отметить, что Windows и MacOS/Linux по-разному обрабатывают окончания строк. Именно поэтому мы указали, что пользователям Windows, возможно, придется использовать `\r\n`, в то время как другие могут обойтись `\n`. Однако преимущество использования `System.lineSeparator()` или `println` в том, что они автоматически подстраиваются под эти тонкие различия в окончаниях строк на уровне операционных систем.

# Чтение входных данных

Давайте создадим новый класс Java с именем `ReadingInput`. Он должен находиться внутри пакета `chapter2` и иметь общедоступный (`public`) метод `main`. Прежде чем добавить в класс какой-либо код, давайте импортируем библиотеку `Scanner`. Это можно сделать с помощью оператора `import` с именем библиотеки. Обязательно поместите этот и все остальные операторы `import` между определением пакета и определением класса:

```
package chapter2;
import java.util.Scanner;
public class ReadingInput {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}
```

Мы будем использовать библиотеку `Scanner` для считывания числа, введенного пользователем. Затем мы преобразуем этот числовой код в соответствующий ему символ ASCII/UTF. Внутри метода `main` удалим комментарий, выведем несколько быстрых инструкций и инициализируем наш объект `Scanner`:

```
public static void main(String[] args) {
    System.out.print("Введите число от 31 до 256: ");
    Scanner inputScanner = new Scanner(System.in);
```

По сути, мы создаем объект как новый экземпляр класса `Scanner`. Мы называем объект `inputScanner`, а затем используем ключевое слово `new` для его инициализации, вызывая конструктор `Scanner`.

*Конструктор* — это особый вид методов, которые вызываются для инстанцирования нового объекта класса.

Метод(ы) конструктора содержат код, который выполняется во время создания объекта, чтобы быть уверенным, что он инициализирован правильно. Мы вызвали конструктор Scanner, передав в качестве параметра ссылку Java на стандартный вход (`Standard Input`, `STDIN`) `System.in`.

Далее мы будем использовать объект `inputScanner` для получения информации о вводе с клавиатуры:

```
String inputStr = inputScanner.nextLine();
inputScanner.close();
```

Создадим новую переменную `String` с именем `inputStr` для получения данных с клавиатуры. Поскольку `Scanner` работает с потоками ввода, мы вызовем метод `close`, поскольку он больше нам не понадобится:

```
int number = Integer.parseInt(inputStr);
System.out.printf("Символ для ASCII-кода %d - это %c", number, (char) number);
```

Далее мы преобразуем `inputStr` в целочисленную переменную с именем `number` с помощью метода `parseInt` из класса `Integer`. Наконец, выведем наши результаты, включая исходный числовой код (`number`) и ASCII/UTF-символ для этого числа, путем приведения `number` к символьному типу (`char`).

Запуск этой программы и ввод с клавиатуры числа 68 приводит к такому результату:

```
Введите число от 31 до 255: 68
Символ для ASCII-кода 68 - это D
```

# Обработка ошибок

Пока что наша программа для преобразования чисел в соответствующие им символы ASCII/UTF работает отлично. Но что произойдет, если ввести не ожидаемые ею входные данные?

Запустите программу снова, но вместо числа введите букву a:

```
Введите число от 0 до 255: 68
Exception in thread "main" java.lang.NumberFormatException: For input string: "a"
```

```
at java.base/java.lang.NumberFormatException.forInputString
(NumberFormatException.java:67)
at java.base/java.lang.Integer.parseInt(Integer.java:665)
at java.base/java.lang.Integer.parseInt(Integer.java:781)
at chapter2.ReadingInput.main(ReadingInput.java:14)
```

В этом случае наши данные привели к тому, что программа выбросила исключение (а именно `NumberFormatException`). Это произошло потому, что метод `parseInt` класса `Integer` ожидает получить число из входных данных. Чтобы избежать этого, необходимо выполнить валидацию ввода. Давайте попробуем окружить наши операторы `parseInt` и `printf` оператором `try/catch`.

Таким образом, если `parseInt` не сработает, то будет выброшено исключение `NumberFormatException`, которое мы видели ранее. Но мы перехватим это исключение и предоставим пользователю изящное сообщение о неудаче:

```
try {
    int number = Integer.parseInt(inputStr);
    System.out.printf("Символ для ASCII-кода %d - это %c", number, (char) number);
} catch (NumberFormatException ex) {
    System.out.println("Извините, можно вводить только цифры.");
}
```

Теперь, если мы повторно запустим нашу программу и введем в качестве входных данных букву 'a', мы увидим следующее:

```
Введите число от 31 до 256: a
Извините, можно вводить только цифры.
```

Обратите внимание, что `NumberFormatException` — это Java-класс, который наследуется от базового класса `Exception`. Это означает, что приведенный код также является корректным:

```
} catch (Exception ex) {
System.out.println("Извините, можно вводить только цифры.");
}
```

Однако всегда лучше отлавливать конкретные исключения. Мы можем захотеть поразному обрабатывать различные типы сбоев, и в этом случае можем неправильно интерпретировать и, таким образом, нечаянно скрыть истинную природу ошибки. Если мы собираемся перехватывать базовый тип `Exception`, нам, вероятно, следует показать его фактическое сообщение об ошибке. К счастью, это можно сделать и при этом перехватить `NumberFormatException`:

```
} catch (NumberFormatException ex) {
    System.out.println("Извините, можно вводить только цифры.");
} catch (Exception ex) {
    System.out.println(ex.getMessage());
}
```

Такая возможность последовательно объединять разделы `catch` позволяет нам реализовать очень динамичный уровень обработки ошибок.

**Примечание.** При перехвате нескольких исключений помните, что они обрабатываются в указанном порядке. Поэтому базовый тип исключения всегда должен быть последним из "пойманных".

## Операторы *if*

Что, если мы введем число меньше 31?

```
Введите число от 31 до 255: 30
Символ для ASCII-кода 68 - это
```

В зависимости от набора символов, установленного на вашем компьютере, вы можете не увидеть ничего, а можете увидеть какой-нибудь другой странно выглядящий символ. Помните, упоминалось, что первоначально наборы символов включали специальные коды для принтеров и что эти управляющие символы до сих пор присутствуют в современных наборах кодировки символов? Так вот, именно к этой категории и относятся числа ниже 32. Даже ввод числа 32 ничего не покажет, поскольку 32 — это код для пробела на клавиатуре.

Необходимо ограничить ввод цифр, чтобы они были больше или равны 32. Это необходимо потому, что все, что ниже этого числа, не будет отображать полезный символ. Этого можно добиться с помощью условия `if/else`:

```
if (number > 31) {
    System.out.printf("Символ для ASCII-кода %d - это %c", number, (char) number);
} else {
    System.out.println("Извините, разрешены только числа от 32 и выше.");
}
```

А как насчет ограничения на максимальный ввод? Ранее упоминалось, что на вход принимаются только числа от 31 до 256. Давайте добавим еще одно условие в оператор `if`:

```
if (number > 31 && number < 256) {
    System.out.printf("Символ для ASCII-кода %d - это %c", number, (char) number);
} else {
    System.out.println("Извините, разрешены только числа от 32 до 255.");
}
```

В данном случае мы добавили в оператор `if` оператор and, представленный двойным амперсандом `&&`. Таким образом, прежде чем преобразовать нашу числовую переменную в символ ASCII/UTF, мы проверяем, что она больше `31` *и* меньше `256`.

Именно здесь можно увидеть, что оператор `if/else` представляет собой булеву логическую структуру. По сути, мы проверяем, что все в предложении `if` равно *true* ("истина"), прежде чем запускать код под ним. Если нет, мы запускаем код в блоке `else`.

**Примечание.** В некоторых языках оператор `if` именуется оператором `if/then`. Это связано с тем, что в таких языках используется ключевое слово 'then', указывающее на конец условия и начало кода для выполнения. Поскольку в Java нет ключевого слова 'then', мы будем называть его просто 'оператор `if`'.

Однако символы больше 255 полезны и для других письменных языков, поэтому давайте закомментируем проверку `if` и будем использовать наш оригинал:

```
//if (number > 31 && number < 256) {
if (number > 31) {
    System.out.printf("Символ для ASCII-кода %d – это %c", number, (char) number);
} else {
    System.out.println(("Извините, разрешены только числа от 32 и выше.");
}
```

## Операторы *switch/case*

Еще один способ проверить соответствие — использовать оператор `switch/case`. Операторы `switch/case` полезны, когда необходимо обработать несколько вариантов входных данных, и создание условия `if` для всех них было бы громоздким.

Давайте создадим новый класс Java с именем `RandomCase`. Он должен находиться внутри пакета `chapter2` и иметь public-метод[1] `main`. Сначала импортируем библиотеку Java `Random`. Начнем с того, что создадим новый объект с именем `random` как новый экземпляр класса `Random`:

```
package chapter2;
import java.util.Random;

public class RandomCase {
public static void main(String[] args) {
    Random random = new Random();
    }
}
```

> **Примечание.** Существует общепринятая практика именования всех классов с использованием "верблюжьего регистра" (camel case), когда в именах нет пробелов, а для первой буквы каждого слова используется заглавный символ. Методы и переменные также называются с использованием верблюжьего регистра, но первая буква имени всегда строчная.

Под определением нашего объекта `random` давайте воспользуемся методом `nextInt`, чтобы сгенерировать случайное число от одного до пяти:

```
int rndNumber = random.nextInt(5) + 1;
```

Вызов метода `nextInt` с передачей числа `5` в качестве параметра сгенерирует случайное целое значение от `0` до `4`. Мы добавляем `+ 1` в конце, чтобы довести результат до минимального значения `1` и максимального `5`.

---

[1] Метод, к которому можно обращаться из любого места программы. Имеет самую высокую степень открытости. — *Прим. ред.*

Далее мы проверим значение rndNumber с помощью оператора switch/case и примем соответствующее решение:

```
switch(rndNumber) {
case 1:
    System.out.println("Один");
    break;
case 2:
    System.out.println("Два");
    break;
```

Как видите, мы начинаем с оператора switch и передаем ему работу с переменной rndNumber. Внутри фигурных скобок используется оператор case, чтобы проверить rndNumber на наличие определенного значения. Затем идет блок кода, который будет выполняться для каждого случая.

Когда код будет выполнен, он завершается оператором break. Оператор break важен, потому что сообщает Java, что мы закончили работу с оператором switch/case. Если оператор break отсутствует, компилятор Java проверяет оставшиеся случаи, пока в одном из них не появится оператор break или пока не закончится оператор switch/case.

Иногда такое поведение полезно. Но для нашей цели, например, после того, как мы установили соответствие в операторе case для числа 2 и вывели Два на терминал, нам не нужно проверять остальные случаи.

Давайте добавим остальные случаи:

```
case 3:
    System.out.println("Три");
    break;
case 4:
    System.out.println("Четыре");
    break;
case 5:
    System.out.println("Пять");
    break;
default:
    System.out.println("Здесь произошло что-то странное...");
}
```

Когда завершается проверка всех случаев, остается вариант по умолчанию default. Думайте о нем как о блоке для всех прочих значений оператора switch/case. Если значение проходит через оператор switch/case, и не совпадает ни с одним условием, выполняется код внутри блока по умолчанию.

> **Примечание.** Блок по умолчанию default не должен содержать оператора break, так как после него нет дополнительных блоков.

Если выполнить предыдущий код, то на терминал будет выведено слово, обозначающее одно из чисел.

## Циклы

Что необходимо сделать, чтобы программа RandomCase выполнялась несколько раз? Это можно сделать с помощью цикла. *Циклы* — это отличный способ проделать следующее:

- повторять код определенное количество раз;
- повторять код до тех пор, пока не наступит другое условие;
- итерировать большую коллекцию данных или элементов.

### Циклы *for*

Допустим, нужно, чтобы наша программа RandomCase выполнялась 10 раз. Это можно сделать, заключив код в цикл for:

```
for (int counter = 0; counter < 10; counter++) {
    int rndNumber = random.nextInt(5) + 1;

    switch(rndNumber) {
    case 1:
        System.out.println("Один");
        break;
    case 2:
        System.out.println("Два");
        break;
    case 3:
        System.out.println("Три");
        break;
    case 4:
        System.out.println("Четыре");
        break;
    case 5:
        System.out.println("Пять");
        break;
    default:
        System.out.println("Здесь произошло что-то странное...");
    }
}
```

Давайте посмотрим на оператор for:

```
for (int counter = 0; counter < 10; counter++)
```

Оператор for работает с тремя параметрами:

- индекс (переменная conter);
- условие;
- изменение индекса/счетчика.

По сути, мы устанавливаем счетчик `counter` в ноль и выполняем код, увеличивая `counter`. Цикл завершится, как только `counter` станет больше или равен числу `10`. Таким образом, код в цикле будет запущен 10 раз (от 0 до 9). Выполнение программы должно дать результаты, похожие на следующие:

```
Три
Четыре
Три
Один
Пять
Три
Три
Два
Четыре
Один
```

Запустите эту программу несколько раз. При каждом выполнении она должна выдавать разные результаты.

## Циклы *while*

Еще один тип цикла — цикл `while`. Циклы `while` предназначены для выполнения до тех пор, пока не будет выполнено заданное условие. Допустим, мы хотим, чтобы наш цикл выполнялся до тех пор, пока не будет случайно сгенерировано число четыре (4). Для этого перед циклом нужно создать новую булеву переменную `fourFound`. Затем с помощью оператора `while` запустим цикл до тех пор, пока `fourFound` не станет истиной:

```
boolean fourFound = false;

//for (int counter = 0; counter < 10; counter++) {
while (!fourFound) {
    int rndNumber = random.nextInt(5) + 1;
```

**Примечание.** В этом примере мы просто закомментировали оператор `for` и заменили его на оператор `while`. Таким образом, можно оставить код цикла в прежнем виде.

Единственное изменение, которое нужно внести, — это учесть случай (`case`), когда будет найдено число четыре:

```
case 4:
    System.out.println("Четыре");
    fourFound = true;
    break;
```

Когда мы установим значение `fourFound` в `true` в промежутке между оператором `case` и оператором `break`, все должно работать. Если забыть об этом шаге, то про-

грамма будет работать в *бесконечном цикле*. Это означает, что код цикла будет выполняться безостановочно, пока программу не остановят принудительно. Кроме того, если установить значение fourFound в true *после* оператора break, то программа никогда не дойдет до этой строки кода, что также приведет к запуску бесконечного цикла.

### Циклы do

Еще один тип циклов известен как цикл do или иногда как цикл do/while. Чтобы преобразовать наш код в цикл do, необходимо внести следующие изменения:

```
Random random = new Random();
boolean fourFound = false;

//for (int counter = 0; counter < 10; counter++) {
// while (!fourFound)
do {
    int rndNumber = random.nextInt(5) + 1;

    switch(rndNumber) {
    case 1:
        System.out.println("Один");
        break;
    case 2:
        System.out.println("Два");
        break;
    case 3:
        System.out.println("Три");
        break;
    case 4:
        System.out.println("Четыре");
        fourFound = true;
        break;
    case 5:
        System.out.println("Пять");
        break;
    default:
        System.out.println("Здесь произошло что-то странное...");
    }
} while (!fourFound);
```

Смысл в том, что в цикле do код цикла будет выполнен хотя бы один раз. В цикле while над телом цикла может быть код, устанавливающий условие оператора while в true, что означает, что код никогда не выполнится. А вот цикл do, напротив, выполнится хотя бы один раз. Это можно проверить, изменив значение параметра fourFound на true. Давайте попробуем сделать так:

```
boolean fourFound = true;
```

При использовании цикла do наш код цикла выполняется один раз, печатает сгенерированное значение на терминале и завершается. Однако запуск кода с уже установленным значением `fourFound` в `true` с помощью цикла `while` ничего не выводит.

# Файлы

Давайте создадим новый Java-класс в нашем проекте. Назовем его `SimpleFileWorker`, убедимся, что он является частью пакета `chapter2`, и что он создан с методом `public static void main`.

## Запись в файл

Добавьте импорт классов `FileWriter` и `IOException` из библиотеки `java.io`. Убедитесь, что они перечислены между определениями пакетов и классов:

```
package chapter2;
import java.io.FileWriter;
import java.io.IOException;
public class SimpleFileWorker {
```

Давайте создадим новый объект `FileWriter` с именем `writer`, передав в качестве аргумента строку `gamesCatalog.txt`. Благодаря этому будет открыт для записи файл gamesCatalog.txt. Если он не существует, он будет создан. Мы также окружим определение нашего объекта `writer` командой `try/catch`, отлавливающей `IOException`:

```
try {
    FileWriter writer = new FileWriter("gamesCatalog.txt");
} catch (IOException writerEx) {
    System.out.println("Во время записи произошла ошибка:");
    writerEx.printStackTrace();
}
```

Теперь давайте запишем в файл две строки текста. Для этого можно использовать метод `write` объекта `FileWriter`. Добавьте эти строки после создания объекта `writer`:

```
// заголовок
writer.write("название, компания, год");

// данные
writer.write("Pitfall!, Activision, 1982");
```

При работе с файлами важно закрывать все открытые дескрипторы файлов. Давайте убедимся, что мы это сделали:

```
writer.close();
```

Эта программа не имеет терминального вывода, поэтому также добавим короткое сообщение (после `try/catch`), чтобы оповестить пользователей о завершении работы программы:

```
System.out.println("Запись завершена!");
```

Запустите программу. Как только вы увидите сообщение "Запись завершена!", откройте сессию в терминале и перейдите в каталог этого проекта. Начните с вывода списка файлов в каталоге.

```
Windows:
dir

Linux/MacOS:
ls -al
```

Найдите файл с именем gamesCatalog.txt. Теперь отобразим содержимое этого файла:

```
Windows:
type gamesCatalog.txt

Linux/MacOS:
cat gamesCatalog.txt
```

Результат должен выглядеть примерно так:

```
имя, компания, годPitfall!, Activision, 198
```

Мы забыли добавить правильный символ окончания строки (из нашего предыдущего обсуждения CRLF) к строкам в файле. Это можно исправить, скорректировав операторы write:

```
// заголовок
writer.write("название, компания, год\r\n");

// данные
writer.write("Pitfall!, Activision, 1982\r\n");
```

**Примечание.** Пользователям Linux и Macintosh нужно добавить только символ \n.

Теперь метод main нашего класса SimpleFileWorker должен выглядеть следующим образом:

```
public static void main(String[] args) {
    try {
        FileWriter writer = new FileWriter("gamesCatalog.txt");

        // заголовок
        writer.write("название, компания, год\n");

        // data
        writer.write("Pitfall!, Activision, 1982\n");

        // close file writer
        writer.close();
```

```
    } catch (IOException writerExc) {
        System.out.println("Во время записи произошла ошибка:");
        e.printStackTrace();
    }
    System.out.println("Запись завершена!");
}
```

Теперь повторно запустите программу и выведите содержимое файла. Вывод должен выглядеть следующим образом:

```
название, компания, год
Pitfall!, Activision, 1982
```

Итак, чего же мы добились? По сути, мы сделали следующее:

- открыли новый файловый обработчик к файлу gamesCatalog.txt и создали этот файл;
- записали две строки в файл;
- закрыли файл.

Для небольших объемов данных это должно работать просто отлично.

Однако если бы мы работали с большими объемами данных и высокой скоростью записи, то пришлось бы внести некоторые коррективы. Класс `FileWriter`, например, не осуществляет никакого кэширования или буферизации, чтобы помочь компенсировать высокую скорость записи. Он полагается на операционную систему для управления дисковым кэшем и другими ресурсами.

Поэтому для продвинутых приложений имеет смысл использовать класс `BufferedWriter`. Класс `BufferedWriter` выполняет дополнительное управление кэшем и уменьшение обратного давления в куче[2] Java. К счастью, создать объект этого класса очень просто, поскольку в качестве параметра он принимает объект `FileWriter`. Но сначала давайте добавим его в секцию `import` нашего класса `SimpleFileWorker`:

```
import java.io.BufferedWriter;
```

Теперь давайте изменим имя нашего объекта `writer` на `fileWriter`. Инстанцируйте новый объект `BufferedWriter`, передав в качестве параметра объект `FileWriter`:

```
FileWriter fileWriter = new FileWriter("gamesCatalog.txt");
BufferedWriter writer = new BufferedWriter(fileWriter);
```

В качестве примера, если мы хотим сократить этот процесс, то можем сделать все это в одной строке, как показано ниже:

```
BufferedWriter writer = new BufferedWriter(new FileWriter("gamesCatalog.txt"));
```

Добавим в файл еще несколько строк к `data` помимо `Pitfall`:

```
writer.write("Pitfall, Activision, 1982\r\n");
writer.write("Crackpots, Activision, 1983\r\n");
```

[2] Куча в Java — это динамически распределяемая область оперативной памяти, создаваемая при старте JVM. Используется для JRE классов и размещения объектов. — *Прим. ред.*

```
writer.write("Yars' Revenge, Atari, 1981\r\n");
writer.write("Warlords, Atari, 1981\r\n");
writer.write("Defender, Atari, 1981\r\n");
writer.write("Adventure, Atari, 1980\r\n");
```

Важно отметить, что и в классах `FileWriter` и `BufferedWriter` имеется метод `flush`. Он полезен, когда файл открыт как для чтения, так и для записи, и вы хотите убедиться, что все содержимое файлового кэша операционной системы сброшено на диск (перед попыткой его чтения).

Для наших целей нам не нужно вызывать эту операцию напрямую. Операция `flush` также вызывается из метода `close`. Поэтому, если не забыть вызвать метод `close` для объекта `writer`, наши данные будут сохранены на диске.

После запуска программы отображение содержимого файла gamesCatalog.txt должно привести к следующим результатам:

```
название, компания, год
Pitfall!, Activision, 1982
Crackpots, Activision, 1983
Yars' Revenge, Atari, 1981
Warlords, Atari, 1981
Defender, Atari, 1981
Adventure, Atari, 1980
```

## Чтение из файла

Теперь давайте попробуем считать данные из файла. Сначала добавим две новые записи в раздел `import`. Для примеров, показанных ниже, нам понадобятся классы `FileReader` и `BufferedReader`:

```
import java.io.BufferedReader;
import java.io.FileReader;
```

Теперь можно было бы использовать тот же подход, что и при записи в файлы. Но знайте, что хотя класс `FileReader` прекрасно подходит для чтения небольших файлов, вместо него лучше использовать `BufferedReader`. Причины использования класса `BufferedWriter` (вместо `FileWriter`) справедливы и для использования класса `BufferedReader` для чтения файлов.

Давайте создадим новый объект `BufferedReader` с именем `reader` после того, как выведем сообщение `"Запись завершена!"`. Также добавим пустой `println`, чтобы обеспечить дополнительный межстрочный интервал в выводе. Кроме того, убедитесь, что новый объект `reader` инстанцируется внутри собственного блока `try/catch`:

```
System.out.println("Запись завершена!");
System.out.println();

try {
    BufferedReader reader = new BufferedReader(new FileReader("gamesCatalog.txt"));
```

```
} catch (IOException readerEx) {
    System.out.println("Во время записи произошла ошибка:");
    readerEx.printStackTrace();
}
```

Далее, внутри try/catch, мы инициализируем строковую переменную gameLine результатом метода readLine. Затем построим цикл while с условием выполнения кода внутри него, пока gameLine не равна null. В коде осуществим вывод строки из файла, а затем снова вызовем метод readLine:

```
// прочитаем первую строку
String gameLine = reader.readLine();

while (gameLine != null) {
    System.out.println(gameLine);
    // прочитаем следующую строку
    gameLine = reader.readLine();
}
```

По мере выполнения цикла while наш код продолжает вызывать метод readLine. Когда в файле больше не останется строк для чтения, метод readLine возвращает null. Когда gameLine принимает значение null, условие while становится равным false, и цикл завершается.

Проще говоря, мы написали небольшую программу для записи заголовка и строк в файл, а также для считывания этих строк из него.

## Чтение данных строки из файла

Также можно поручить программе считывать только определенные данные из файла. Например, давайте прочитаем названия игр, которые были выпущены в 1981 году. Чтобы это произошло, необходимо выполнить несколько основных шагов:

- прочитать все строки из файла;
- разобрать каждую строку на следующие столбцы:
  - название
  - компания
  - год
- если год равен 1981, вывести строку из файла.

Вне цикла код должен оставаться практически таким же. Но внутри цикла начнем с того, что выделим значение каждого столбца из gameLine. Это можно сделать с помощью метода split, который может быть вызван для любой строки. Метод split возвращает строковый массив, поэтому создадим массив с именем gameColumns и инициализируем его значением, полученным после разделения gameLine запятой:

```
String[] gameColumns = gameLine.split(",");
```

О таких вещах, как массивы и строковые методы, мы поговорим в последующих главах книги. Пока же достаточно сказать, что мы разделили gameLine на три столбца. Помните, что наш файл разделен запятыми: в каждой строке файла есть две запятые, разделяющие каждую строку на три значения.

Поскольку Java работает с системой счисления, начинающейся с нуля, три значения в массиве имеют индексы 0, 1 и 2. Таким образом, столбец "год" находится под индексом 2. Для него можно задать собственную переменную:

```
String strYear = gameColumns[2];
```

Со значениями обычно проще работать в их родных типах данных, поэтому давайте преобразуем год в целое число. Для этого можно воспользоваться методом parseInt класса-обертки Integer:

```
int year = Integer.parseInt(strYear);
```

Для большей эффективности кода можно объединить обе предыдущие строки в одну:

```
int year = Integer.parseInt(gameColumns[2]);
```

Теперь давайте создадим проверку if для вывода только тех строк из файла, в которых год year указан как 1981:

```
if (year == 1981) {
    System.out.println(gameLine);
}
```

Давайте попробуем запустить то, что у нас уже есть. К сожалению, возникнет исключение:

```
Запись завершена!
Exception in thread "main" java.lang.NumberFormatException: For input string: " год"
    at
java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.
java:67)
    at java.base/java.lang.Integer.parseInt(Integer.java:647)
    at java.base/java.lang.Integer.parseInt(Integer.java:777)
    at chapter2.SimpleFileWorker.main(SimpleFileWorker.java:49)
```

Хотя исключение здесь одно, в коде имеются две проблемы. Если посмотреть на нижнюю часть сообщения об исключении, то можно увидеть, что проблема возникла в методе main в строке 49. В строке 49 должна быть вот эта строка кода:

```
int year = Integer.parseInt(gameColumns[2]);
```

Двумя строками выше имеется явное указание на метод parseInt().

Если посмотреть на первую строку в сообщении об исключении, то можно увидеть текст, который привел к ошибке метода parseInt():

```
NumberFormatException: For input string: " год"
```

В конечном итоге эта ошибка была вызвана строкой заголовка. Строка заголовка — это первая строка в файле, и выглядит она следующим образом:

```
название, компания, год
```

Поскольку строка с именем столбца "год" не может быть преобразована в целое число, возникает исключение NumberFormatException. Чтобы избежать этого, нам нужно правильно обработать строку заголовка. За пределами цикла while создадим новую булеву переменную headerRead и инициализируем ее значением false. Таким образом, в начале цикла мы будем знать, что заголовок еще не прочитан:

```
boolean headerRead = false;
```

Далее давайте заключим наш код со split(), parseInt() и логикой if внутри еще одного if. Оператор if должен проверить, равен ли headerRead true. Если значение headerRead равно false, необходимо лишь установить его в true. Этого будет достаточно, чтобы код пропустил заголовок файла:

```
if (headerRead) {
    String[] gameColumns = gameLine.split(",");
    int year = Integer.parseInt(gameColumns[2]);
    if (year == 1981) {
        System.out.println(gameLine);
    }
} else {
    headerRead = true;
}
```

Но если запустить код сейчас, то обнаружим *вторую* проблему:

```
NumberFormatException: For input string: " 1982"
```

Посмотрите на пробел перед цифрами. Метод parseInt() не может преобразовать строку 1982 в целое число, пока в строке есть ведущий пробел. К счастью, есть простой метод, который может справиться с этой проблемой, — метод trim().

Метод trim() удаляет из строки лишние пробелы, и его можно вызвать для строки год. Лишние пробелы могут вызвать проблемы, поэтому выполнение trim() для файлов и пользовательского ввода обычно является хорошей идеей. Теперь обрабатывающая строка должна выглядеть следующим образом:

```
int year = Integer.parseInt(gameColumns[2].trim());
```

> **Примечание.** Как и метод split(), метод trim() может быть вызван для всех строковых данных в Java.

Теперь полный код части программы, связанной с чтением файлов, должен выглядеть следующим образом:

```
try {
    BufferedReader reader = new BufferedReader(new FileReader("gamesCatalog.txt"));

    // прочитаем первую строку
    String gameLine = reader.readLine();
    boolean headerRead = false;

    while (gameLine != null) {
        if (headerRead) {
```

```
            String[] gameColumns = gameLine.split(",");
            int year = Integer.parseInt(gameColumns[2].trim());
            if (year == 1981) {
                System.out.println(gameLine);
            }
        } else {
            headerRead = true;
        }
        // прочитаем следующую строку
        gameLine = reader.readLine();
    }
} catch (IOException readerEx) {
    System.out.println("Во время записи произошла ошибка:");
    readerEx.printStackTrace();
}
```

Запуск этой программы должен привести к следующему результату:

```
Запись завершена!

Yars' Revenge, Atari, 1981
Warlords, Atari, 1981
Defender, Atari, 1981
```

# Методы и конструкторы

Давайте рассмотрим методы и конструкторы. *Методы*, также известные как функции или подпрограммы, — это просто небольшие блоки написанного с определенной целью кода. Если мы обнаруживаем, что код в нашем методе `main` становится длинным, обычно хорошей идеей является перенос части кода в другой метод или два. Это особенно актуально, если небольшой фрагмент кода вызывается несколько раз.

*Конструкторы* — это особые виды методов, вызываемые при создании объекта. Классы объектов имеют один или несколько конструкторов, в зависимости от количества различных способов создания объекта.

Чтобы проиллюстрировать эти понятия, давайте создадим несколько новых классов в пакете `chapter2`.

- `MetricUnitConverter`. Со статическим не возвращающим данные методом `main`;
- `MeasurementValue`. Используем этот класс для хранения данных измерений;
- `InvalidUOMException`. Расширение класса `RuntimeException`.

## Пример программы *MetricUnitConverter*

Создадим программу для получения измерений в метрических или имперских единицах и преобразования данных между ними.

Мы будем использовать следующие *единицы измерения* (units of measure, UOM):

- имперские единицы:
  - дюймы (`inches`)
  - футы (`ft`)
  - мили (`mi`)
- метрические единицы:
  - сантиметры (`см`)
  - метры (`m`)
  - километры (`км`)

### Класс *InvalidUOMException*

Начнем с класса `InvalidUOMException`, являющегося пользовательским классом исключений для решения конкретного типа проблем. Если пользователь передает UOM, которого нет в этом списке (см. выше), необходимо выбросить наше пользовательское исключение. Чтобы реализовать эту функциональность, можно создать простой класс `InvalidUOMException` следующим образом:

```
package chapter2;
public class InvalidUOMException extends RuntimeException {
    public InvalidUOMException() {
        super("Не удалось определить UOM этой записи.");
    }
}
```

Класс `InvalidUOMException` является наследником `RuntimeException` (который сам наследуется от базового класса `Exception`). Это позволит нам работать с ним, используя стандартные средства обработки исключений Java. У него есть простой конструктор без аргументов, который использует оператор `super` для вызова конструктора класса-родителя и инициализации объекта с нашим пользовательским сообщением об ошибке.

> **Примечание.** Вы можете получить предупреждение о том, что пользовательский класс исключений не объявляет static final serialVersionUID. Чтобы обойти это, можно указать вашей IDE сгенерировать его. В противном случае это предупреждение можно проигнорировать, и JVM сгенерирует его во время выполнения.

### *MeasurementValue* POJO

Теперь давайте перейдем к нашему классу `MeasurementValue`. Мы будем использовать этот класс для создания *POJO-объекта* Java (Plain Old Java Object — "простой объект Java", "старый добрый объект Java") для отслеживания входных данных, UOM и преобразованных значений.

Сначала определим наш класс и свойства объекта:

```
package chapter2;
public class MeasurementValue {
    private double inputValue;
    private String inputUOM;
    // metric
    private double centimeters;
    private double meters;
    private double kilometers;
    // imperial
    private double inches;
    private double feet;
    private double miles;
```

Чтобы быть уверенными в том, что сможем обрабатывать большие числа, будем использовать тип double. Теперь давайте создадим несколько констант для облегчения преобразования между различными UOM:

```
// константы преобразования
private final int CENTIMETERS_PER_METER = 100;
private final int METERS_PER_KILOMETER = 1000;
private final int INCHES_PER_FOOT = 12;
private final int FEET_PER_MILE = 5280;
private final double METERS_PER_FOOT = 0.3048d;
```

Наши *константы* (обозначаемые ключевым словом final) — это целые числа, которые могут быть определены как integer. Java будет неявно преобразовывать значения, полученные в результате арифметических операций между double и integer, в double.

Последняя определенная константа поможет преобразовать метры в футы. Поскольку METERS_PER_FOOT — это десятичное значение, нам нужно поставить символ d в конце, чтобы оно было преобразовано в double. Таким образом, константа METERS_PER_FOOT приобретает значение 0,3048d.

> **Примечание.** Считается, что в стиле Java лучше всего определять имена постоянных переменных-констант, используя "*змеиный регистр*" (snake case), в котором слова разделены символами подчеркивания ("_"). Кроме того, имена переменных-констант должны быть написаны заглавными буквами.

Для двух последних переменных необходимо определить класс MeasurementValue, который позволит правильно отформатировать вывод различных значений. Объект dFormat является экземпляром класса DecimalFormat, и мы вызовем его конструктор, чтобы указать, что необходимо ограничить наши значения double двумя точками десятичной точности.

Также будет создан объект mKmFormat класса DecimalFormat. Разница между уровнем миль и километров намного больше, чем у других, более мелких UOM.

Поэтому необходимо обеспечить большую точность для этих UOM:

```
private static final DecimalFormat dFormat = new DecimalFormat("0.00");
private static final DecimalFormat mKmFormat = new DecimalFormat("0.0000");
```

Определив свойства объекта, форматирователи и константы, перейдем к конструкторам класса `MeasurementValue`. Создадим два конструктора. Один конструктор будет принимать строку, состоящую из числового значения и значения UOM, разделенных пробелом. Другой будет принимать параметры для числового значения (в виде `double`) и UOM по отдельности:

```
public MeasurementValue(String valueStr) throws Exception {
    parseInput(valueStr);
    runConversions();
}

public MeasurementValue(double value, String uom) throws Exception {
    inputValue = value;
    inputUOM = uom;
    runConversions();
}
```

Вспомните из *главы 1 "Знакомство с Java"*, что перегрузка конструкторов — это форма статического полиморфизма. В реальном мире мы бы сделали так для удобства разработчиков, использующих наш класс. У них уже могут быть разобраны и разделены значение `value` и UOM. Или же у них может быть необработанный пользовательский ввод. В любом случае, такие конструкторы помогут им выбрать, как построить приложение. Но для нашей цели в этой книге мы будем использовать конструктор, который принимает пользовательский ввод, хранящийся в строке `valueStr`.

Оба определения наших конструкторов включают ключевое слово `throws`, чтобы передать все возникшие исключения вызывающему методу или классу. Это делается потому, что нам нужно сосредоточиться на обработке исключений в нашем методе `main`, к которому вскоре перейдем.

В этих конструкторах мы указали два private-метода[3], которые еще не существуют: `parseInput()` и `runConversions()`. Сейчас мы их создадим. Поскольку наш конструктор зависит от метода `parseInput()`, создадим его первым:

```
private void parseInput(String input) {
    String[] params = input.trim().split(" ");
    inputValue = Double.parseDouble(params[0]);
    inputUOM = params[1];
}
```

Метод `parseInput()` может быть небольшим, но он делает довольно много. У него область доступа private, поскольку мы не хотим, чтобы он вызывался извне класса

[3] Метод, который доступен только внутри того класса, где он присутствует. — *Прим. ред.*

MeasurementValue. Его тип возврата — void, так как мы можем устанавливать свойства объекта напрямую (поэтому возвращать ничего не нужно), и он принимает на входе строковый параметр с простым именем input.

Сначала метод выполняет обрезку trim входной строки, а затем разбивает ее на строковый массив с именем params, предполагая, что значения разделены одним пробелом. Затем он берет первый элемент в массиве params (params[0]), преобразует его в числовой тип double и сохраняет результат в свойстве объекта inputValue. Затем он сохраняет второй элемент массива params (params[1]) в свойстве объекта inputUOM.

Далее создадим метод runConversions(). Идея этого метода заключается в том, что он проверяет UOM, а затем выполняет необходимые преобразования, чтобы мы знали, какое у нас значение в сантиметрах, дюймах, метрах, футах, километрах и милях. Поскольку у нас есть несколько значений UOM, для которых нужно запустить логику, лучшим решением станет использование оператора switch/case.

Определение метода будет выглядеть следующим образом:

```
private void runConversions() throws Exception {
```

Как и метод parseInput(), метод runConversions() будет private, поскольку не нужно, чтобы он вызывался извне класса MeasurementValue. Аналогично, у него не будет возвращаемого типа, поскольку этот метод будет изменять свойства объекта напрямую. В определении также присутствует оператор throws, поскольку все исключения будут передаваться *наверх*.

Затем определим оператор switch/case и код, выполняющийся в зависимости от значения inputUOM:

```
switch (inputUOM) {
case "in":
    inches = inputValue;
    feet = inches / INCHES_PER_FOOT;
    miles = feet / FEET_PER_MILE;
    convertImperialToMetric();
    break;
```

Мы создадим код с case для каждого из шести типов UOM, которые могут подаваться на вход. В предыдущем фрагменте представлен код для UOM в дюймах, помеченный аббревиатурой in. По сути, если мы получаем значение в дюймах, необходимо установить соответствующее свойство объекта (в данном случае inches) равным числу, хранящемуся в свойстве inputValue.

Теперь, когда установлено, что значение выражено в дюймах, можно использовать определенные нами константы для вычисления значения в футах и милях. Аналогичным образом можно сохранить эти данные в свойствах объекта feet и miles. Затем мы вызываем метод convertImperialToMetric() (который создадим в ближайшее время), чтобы вычислить значение измерения в сантиметрах, метрах и километрах. Наконец, завершим код case для дюймов оператором break, поскольку нет необходимости проверять другие UOM.

После создания аналогичных case для футов ("ft") и миль ("mi") мы создадим метрические case, начав с case для сантиметров:

```
case "cm":
    centimeters = inputValue;
    meters = centimeters / CENTIMETERS_PER_METER;
    kilometers = meters / METERS_PER_KILOMETER;
    convertMetricToImperial();
    break;
```

Как и в предыдущем случае с дюймами (inches), в случае с сантиметрами (centimeters) код запускается, если inputUOM имеет строковое значение cm. Получив значение в сантиметрах, можно быстро использовать константы для вычисления метров и километров. Затем вызовем метод convertMetricToImperial() (который создадим в ближайшее время) для преобразования значения измерения в дюймы, футы и мили. Аналогичным образом завершаем работу с помощью оператора break, чтобы не тратить лишнее время на обработку условий, которые, как мы знаем, не будут выполнены.

Аналогичные случаи мы создадим для метров ("m") и километров ("km"). Наконец, создадим обработку по умолчанию (default):

```
default:
     throw new InvalidUOMException();
}
```

Обработка по умолчанию (default) срабатывает, если значение строки inputUOM не соответствует ни одному из шести предопределенных UOM. Единственное, что делает этот обработчик, — выбрасывает наше пользовательское исключение InvalidUOMException. Очевидно, что если поток программы дошел до этого места, то что-то не так, поэтому мы создаем новый объект нашего пользовательского исключения и "выбрасываем" (throw) его.

Полный код нашего метода runConversions() выглядит следующим образом:

```
private void runConversions() throws Exception {
    switch (inputUOM) {
    case "in":
        inches = inputValue;
        feet = inches / INCHES_PER_FOOT;
        miles = feet / FEET_PER_MILE;
        convertImperialToMetric();
        break;
    case "ft":
        feet = inputValue;
        inches = feet * INCHES_PER_FOOT;
        miles = feet / FEET_PER_MILE;
        convertImperialToMetric();
        break;
```

```
    case "mi":
        miles = inputValue;
        feet = miles * FEET_PER_MILE;
        inches = feet * INCHES_PER_FOOT;
        convertImperialToMetric();
        break;
    case "cm":
        centimeters = inputValue;
        meters = centimeters / CENTIMETERS_PER_METER;
        kilometers = meters / METERS_PER_KILOMETER;
        convertMetricToImperial();
        break;
    case "m":
        meters = inputValue;
        centimeters = meters * CENTIMETERS_PER_METER;
        kilometers = meters / METERS_PER_KILOMETER;
        convertMetricToImperial();
        break;
    case "km":
        kilometers = inputValue;
        meters = kilometers * METERS_PER_KILOMETER;
        centimeters = meters * CENTIMETERS_PER_METER;
        convertMetricToImperial();
        break;
    default:
        throw new InvalidUOMException();
    }
}
```

Двигаясь дальше, давайте рассмотрим метод `convertImperialToMetric()`. Идея этого метода заключается в том, что у нас уже есть значение для свойства `feet`. Свойства `feet` и `meters` были выбраны в качестве основы для преобразования `Metric/Imperial`, поскольку они находятся в промежутке между другими измерениями. Если бы мы выбрали сантиметры и дюймы, точность определения километров и миль была бы искажена. Аналогично, использование километров и миль исказило бы точность для сантиметров и дюймов.

Поэтому идея метода `convertImperialToMetric()` заключается в том, что сначала мы конвертируем футы в метры. Затем, используя метры, мы вычисляем сантиметры и километры:

```
private void convertImperialToMetric() {
    // используем  feet/meters для преобразования
    meters = feet * METERS_PER_FOOT;
    centimeters = meters * CENTIMETERS_PER_METER;
    kilometers = meters / METERS_PER_KILOMETER;
}
```

Мы используем тот же подход для метода convertMetricToImperial(). Сначала вычисляется свойство feet, исходя из значения, хранящегося в свойстве meters. Как только мы получим значение футов, используем наши константы для вычисления дюймов и миль:

```
private void convertMetricToImperial() {
    // используем feet/meters для преобразования
    feet = meters / METERS_PER_FOOT;
    inches = feet * INCHES_PER_FOOT;
    miles = feet / FEET_PER_MILE;
}
```

Далее мы создадим общедоступные public-методы для класса MeasurementValue. Но нам нужен только метод toString(). Сейчас все объекты наследуют метод toString() от базового класса Object. Но мы хотим быть уверены, что вывод свойства будет правильно отформатирован, поэтому мы *перегрузим* этот метод, написав свой собственный:

```
public String toString() {
    dFormat.setRoundingMode(RoundingMode.UP);
    mKmFormat.setRoundingMode(RoundingMode.UP);

    StringBuilder values = new StringBuilder();
    values.append("\ninches = ");
    values.append(dFormat.format(inches));
    values.append("\nfeet = ");
    values.append(dFormat.format(feet));
    values.append("\nmiles = ");
    values.append(mKmFormat.format(miles));
    values.append("\ncentimeters = ");
    values.append(dFormat.format(centimeters));
    values.append("\nmeters = ");
    values.append(dFormat.format(meters));
    values.append("\nkilometers = ");
    values.append(mKmFormat.format(kilometers));

    return values.toString();
}
```

Как показано, мы начинаем с установки режима округления для наших объектов DecimalFormat в UP ("вверх"). Затем мы инициализируем объект StringBuilder значениями имен. Мы создаем StringBuilder с текстом пояснений для наших свойств и форматированных свойств.

**Примечание.** Мы начинаем каждую строку с символа новой строки '\n', чтобы значения наших свойств были легко читаемы.

### Замечание об инкапсуляции

Обычно POJO-классы имеют публичные методы для доступа к свойствам объекта, называемые *геттерами* и *сеттерами*. Создание геттеров и сеттеров — это хороший способ контролировать или разрешать доступ к данным внутри POJO-классов. Однако входные данные в наш класс поступают только через конструкторы, а выходные доступны только через метод `toString()`. Поэтому, в духе правильной инкапсуляции нашей программы, мы не должны позволять читать или записывать какие-либо свойства наших отдельных объектов.

### Класс *MetricUnitConverter (main)*

Написав вспомогательные классы, давайте создадим класс `MetricUnitConverter` и соединим его с другими. Определение этого класса будет выглядеть следующим образом:

```
package chapter2;

import java.util.Scanner;

public class MetricUnitConverter {
```

Необходимо убедиться, что у него также есть метод `main`. Для начала выведем несколько кратких инструкций, чтобы помочь нашим пользователям вводить данные в правильном формате:

```
public static void main(String[] args) {
    // инструкции
    System.out.println("Эта программа конвертирует единицы измерения (ЕИ)
    между  метрическими и имперскими единицами.");
    System.out.println("Допустимые ЕИ: (in, ft, mi, cm, m, km");
    System.out.println("Примеры: (14 ft, 5 km)");
    System.out.print("Введите число с его ЕИ: ");
```

Далее необходимо создать объект `inputScanner`, получить входные данные из `STDIN` и закрыть объект:

```
// получение входных данных
Scanner inputScanner = new Scanner(System.in);
String inputStr = inputScanner.nextLine();
inputScanner.close();
```

После того как получены данные от пользователя, нам необходимо проверить, содержится ли в них пробел. Это связано с тем, что принимаемый нами ввод — это числовое значение и UOM, разделенные пробелом. Если пробел отсутствует, продолжать работу не следует:

```
// проверка наличия пробела
int spacePos = inputStr.indexOf(" ");
```

Как только убедимся, что во входной строке присутствует пробел, можно продолжать. Первое, что необходимо сделать, это окружить нашу логику `try/catch`. Вспомните, что класс `MeasurementValue` содержал несколько операторов `throws`. Оператор `try/catch` должен будет проверять эти исключения.

Внутри оператора `try/catch` создадим новый объект класса `MeasurementValue` с именем `measurement` и вызовем его первый конструктор с нашим `inputStr`. Затем можно вывести результаты всех измерений UOM с помощью метода `toString()` объекта (который мы перегрузили):

```
if (spacePos > 0) {
    try {
        MeasurementValue measurement = new MeasurementValue(inputStr);
        System.out.print(measurement.toString());
    } catch (Exception ex) {
        System.out.println(ex.getMessage());
}
```

Далее построим блок `else`, чтобы отловить случаи, когда `inputStr` не содержит пробела:

```
} else {
    System.out.println("Проверьте ваши входные данные; "
    +
    "в них должен быть пробел между значением и ЕИ.");
}
```

Полностью код класса `MetricUnitConverter` должен выглядеть следующим образом:

```
package chapter2;

import java.util.Scanner;

public class MetricUnitConverter {

public static void main(String[] args) {
    // инструкции
    System.out.println("Эта программа конвертирует единицы измерения (ЕИ)
    между метрическими и имперскими единицами.");
    System.out.println("Допустимые ЕИ: (in, ft, mi, cm, m, km)");
    System.out.println("Примеры: (14 ft, 5 km)");
    System.out.print("Введите число с его ЕИ: ");

    // получение входных данных
    Scanner inputScanner = new Scanner(System.in);
    String inputStr = inputScanner.nextLine();
    inputScanner.close();

    // проверка наличия пробела
    int spacePos = inputStr.indexOf(" ");
```

```
    if (spacePos > 0) {
        try {
            MeasurementValue measurement = new MeasurementValue(inputStr);
            System.out.print(measurement.toString());
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    } else {
        System.out.println("Проверьте ваши входные данные; "
        +
        "в них должен быть пробел между значением и ЕИ.");
        }
    }
}
```

Запуск этого кода и подача входных данных `8 m` должны дать такой результат:

```
Эта программа конвертирует единицы измерения (ЕИ) между метрическими
и имперскими единицами.
Допустимые ЕИ: (in, ft, mi, cm, m, km)
Примеры: (14 ft, 5 km)
Введите число с его ЕИ: 8m

inches = 314.97
feet = 26.25
miles = 0.0050
centimeters = 800.00
meters = 8.00
kilometers = 0.0080
```

Вы можете попробовать другие входные данные и данные, которые могут вызвать исключения (например, ввести `8m` без пробела).

# Заключение

В этой главе мы рассмотрели основные операторы и структуры Java. Обсудили печать вывода на терминал и чтение ввода с клавиатуры, а также использование циклов для выполнения простых задач. Мы также поработали с файлами и познакомились с более сложными темами, такими как разбор строк и обработка исключений.

Рассмотренные в этой главе темы будут часто упоминаться и использоваться в последующих главах. Эти фундаментальные темы актуальны для создания всех видов приложений, от простых взаимодействий до сложнейших интеграций.

В следующей главе более подробно рассматриваются символьные и строковые типы данных. Мы также изучим некоторые методы, входящие в состав класса String, и даже используем несколько простых регулярных выражений.

# Важно помнить

- Классы именуются с использованием верблюжьего регистра (camel case), начиная с заглавной буквы.
- Переменные называются в верблюжьем регистре, начиная со строчной буквы.
- Константы называются в змеином регистре (snake case), со всех заглавных букв.
- При работе с `File` или `Scanner` (или любым другим потоковым `stream` объектом) не забывайте закрывать его по завершении работы.
- Исключения могут быть обработаны сразу или выброшены `throw` вверх, но они должны быть обработаны до того, как попадут к пользователю.
- Циклы `For` хороши для обработки заранее определенного числа итераций.
- Циклы `While` подходят для обработки неизвестного количества итераций.
- Код внутри циклов `Do/While` будет выполнен хотя бы один раз, а код в цикле `While` — не обязательно.

Глава 3

# Строки, символы и регулярные выражения

## Введение

Эта глава посвящена работе с текстовыми данными. Мы познакомимся со строками и символами и поймем, как использовать их с помощью методов разбора и равенства. Также рассмотрим регулярные выражения и увидим, насколько мощными они могут быть для выполнения операций сравнения строк и сопоставления шаблонов.

## Структура

В этой главе мы обсудим следующие темы.

- Символы.
- Строки.
- Равенство строк.
- Регулярные выражения.

## Цели

Поскольку мы все еще продолжаем развивать фундаментальные знания о Java, цель этой главы — помочь понять, как работать с символьными и строковыми данными. Мы изучим некоторые методы класса `String` и применим их на практике. А также узнаем о регулярных выражениях и о том, когда их следует использовать.

## Символы

*Символ* — это тип данных, предназначенный для хранения одного алфавитно-цифрового символа. Однако "под капотом" символ хранится как целочисленное

значение. Когда символ отображается или считывается в Java, соответствующее ему целое число преобразуется в символ с помощью набора кодировок JVM (также известного как charset). По умолчанию для Java (начиная с Java 18) используется кодировка UTF-8.

До сих пор мы работали в основном со строками. Строки — это, по сути, массивы символов. Некоторые реализации строковых библиотек в других языках создают и обрабатывают строки как массивы символов.

Давайте создадим новый класс Java. Назовем этот класс `FunWithCharacterCodes` и убедимся, что он находится в пакете `chapter3`. Не забудьте также создать в этом классе метод `main()`. Внутри метода `main()` определим три строки и три символа для заглавных букв `A`, `B` и `C`. Затем просто выведем их на печать вместе с кодировкой по умолчанию (`charset`):

```
import java.nio.charset.Charset;

public class FunWithCharacterCodes {

    public static void main(String[] args) {
        String strA = "A";
        String strB = "B";
        String strC = "C";

        char upperA = 'A';
        char upperB = 'B';
        char upperC = 'C';

        System.out.println(strA);
        System.out.println(strB);
        System.out.println(strC);

        System.out.println(upperA);
        System.out.println(upperB);
        System.out.println(upperC);

        System.out.println(Charset.defaultCharset());
    }
}
```

Выполнение приведенного выше кода должно привести к таким результатам:

```
A
B
C
A
B
C
UTF-8
```

Теперь подправим две строки кода. Давайте дважды выведем значение для буквы A. Сначала скорректируем строку, которая печатает строку strA:

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

Далее также изменим строку, которая печатает строку upperA:

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

Кажется, что это одно и то же. Однако при выполнении этого кода получается следующий результат:

```
AA
B
C
130
B
C
UTF-8
```

Возможно, представленный вывод — не то, что мы ожидали, но давайте подробно рассмотрим код, чтобы понять, что произошло и почему был показан именно такой вывод.

Мы уже использовали оператор + (плюс) для строк, поэтому конкатенированный результат AA должен быть ожидаемым. Применение оператора + к двум символам A дало числовое значение 130.

Это произошло потому, что Java разобрала оператор +, а это значит, что при отсутствии перегрузки (как это делает класс String) он ожидает работы с числовыми операндами. Java не знала, что делать с двумя примитивными символами, поэтому она использовала двоичное числовое преобразование.

*Двоичное числовое продвижение* — это процесс изменения небольшого числового типа в более крупный. Обычно это происходит при выполнении операции над двумя числовыми типами, которые не совпадают. В этом процессе Java работает с символами, преобразуя их в целые числа. Код символа ASCII/UTF для заглавной буквы A равен 65. Таким образом, *сложение* символов (A + A) равносильно сложению *65* и *65*, или *130*.

Это можно использовать при чтении и проверке вводимых символов. Давайте воспользуемся классом Scanner, чтобы считать один символ с клавиатуры и проверить его диапазон в алфавите:

```
//  считываем символ с клавиатуры
Scanner inputScanner = new Scanner(System.in);
System.out.println("\nВведите один символ.");
String inputStr = inputScanner.nextLine();
```

Далее создадим новую символьную переменную inputChar и установим ее значение с помощью метода string charAt(), чтобы взять первый символ из inputStr:

```
char inputChar = inputStr.charAt(0);
```

Теперь закроем inputScanner и проверим, находится ли inputChar между буквами A и K, L и Q или R и Z:

```
inputScanner.close();

if (inputChar >= 'A' && inputChar < 'L') {
    System.out.println("Символ находится между A and K.");
} else if (inputChar >= 'L' && inputChar < 'R') {
    System.out.println("Символ находится между L and Q.");
} else {
    System.out.println("Символ должен находиться между R and Z.");
}

System.out.println(inputChar + " = " + (int)inputChar);
```

В завершение напечатаем введенный символ и его UTF-код. Запуск этой программы должен привести к следующему результату:

```
AA
B
C
130
B
C
UTF-8
Введите один символ.
F
Символ находится между A и K.
F = 70
```

Все работает хорошо, пока мы не попробуем ввести строчный символ:

```
Введите один символ.
a
Символ находится между R и Z.
a = 97
```

Строчная буква a не находится между R и Z. Это ошибка в нашей проверке if. Да, у строчных символов UTF-индекс больше, чем у прописных. Давайте расширим наш оператор if, чтобы учесть это:

```
if (inputChar >= 'A' && inputChar < 'L') {
    System.out.println("Символ находится между A и K.");
} else if (inputChar >= 'L' && inputChar < 'R') {
    System.out.println("Символ находится между L и Q.");
} else if (inputChar >= 'R' && inputChar < 'Z') {
    System.out.println("Символ должен находиться между R и Z.");
```

```
} else if (inputChar >= 'a' && inputChar < 'l') {
    System.out.println("Символ находится между a и k.");
} else if (inputChar >= 'l' && inputChar < 'r') {
    System.out.println("Символ находится между l и q.");
} else {
    System.out.println("Символ должен находиться между r and z.");
}
```

Теперь при вводе символа в нижнем регистре будут получены правильные результаты:

```
Введите один символ.
a
Символ находится между a и k.
a = 97
```

## ASCII-арт

Некоторые коды символов ASCII/UTF указывают на линии и фигуры, которые можно использовать для создания простых изображений. Это называется *ASCII-арт*, и он уже несколько десятилетий используется для придания программам особого шарма.

Чтобы использовать его, создадим новый класс Java с именем `CharacterArt` внутри пакета `chapter3`; убедитесь, что у него есть метод `main()`. Укажите эти три `import` между определениями пакета и класса:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
```

Наша программа будет считывать UTF-коды из файла с именем commandKeyCodes.txt.

Программа предполагает, что этот файл находится в каталоге data/ нашего проекта. Файл представляет собой текстовый файл с разделенными запятыми данными, и его можно легко просмотреть из терминальной сессии в каталоге нашего проекта:

```
Windows:
type data\commandKeyCodes.txt

Linux/MacOS:
cat data/commandKeyCodes.txt
```

Содержимое файла выглядит следующим образом:

```
9484,9472,9472,9488,9484,9472,9472,9488
9474,9484,9488,9474,9474,9484,9488,9474
9474,9492,9524,9524,9508,9500,9496,9474
9492,9472,9516,9516,9508,9500,9472,9496
9484,9472,9508,9500,9524,9524,9472,9488
```

```
9474,9484,9508,9500,9516,9516,9488,9474
9474,9492,9496,9474,9474,9492,9496,9474
9492,9472,9472,9496,9492,9472,9472,9496
```

Чтобы обработать его, давайте создадим `BufferedReader` (для файла) внутри `try/catch` и прочитаем первую строку:

```
public static void main(String[] args) {
    try {
        BufferedReader reader = new BufferedReader(new
FileReader("data/commandKeyCodes.txt"));

        // Считываем первую строку
        String dataLine = reader.readLine();
```

Как и раньше, напишем цикл `while` для обработки строк в файле путем *разбиения* их запятой. Числовые коды в этом файле не содержат пробелов, поэтому в этот раз нам не нужно запускать `trim()`. Мы разделим строку `dataLine` на массив с именем `data`:

```
while (dataLine != null) {
    String[] data = dataLine.split(",");
```

Внутри цикла `while` мы используем цикл `for` для перебора элементов в массиве `data`. Затем мы используем класс-обертку `Integer` для преобразования значения в целое число с именем `number`. Наконец, выведем `number` на экран, преобразовав его в символ:

```
for (String strNumber : data) {
    int number = Integer.parseInt(strNumber);

    System.out.print((char)number);
}
```

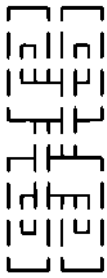
Поскольку работа с текущей строкой закончена, необходимо выполнить `println()`, чтобы перейти к следующей строке. Также прочитаем следующую строку из файла в `dataLine`, что запустит логику цикла `while` заново:

```
    System.out.println();

    dataLine = reader.readLine();
}
```

После этого остается закрыть `reader` и обработать исключение:

```
        reader.close();

    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

Запуск этой программы должен привести к следующему результату:



Мы нарисовали символ, напоминающий командную клавишу Macintosh. Учитывая то, что мы проделали это с помощью нескольких строк кода и файла размером восемь на восемь с разделенными запятыми числами, легко понять, что возможности печати ограничены только нашим воображением.

> **Примечание.** Таблица преобразования UTF представлена в *приложении 2 "Таблица преобразования UTF"*.

# Строки

Мы уже работали со строками в предыдущих главах. Но теперь рассмотрим более подробно мощные функции, которые можно использовать для работы со строками.

Создайте новый Java-класс с именем `WorkingWithStrings` внутри пакета `chapter3`. Для этого класса также потребуется метод `main()`. Внутри метода `main()` определите строку с именем `email`.

```
String email = "victoria.ploetz@largecorp.com";
```

Как видим, адреса электронной почты сотрудников Large Corp построены по следующей схеме:

- имя;
- точка;
- фамилия;
- знак "at" ("@");
- `largecorp` в нижнем регистре;
- точка, а затем суффикс `com`.

Начнем с разделения email на более мелкие составляющие. Для этого сначала нужно определить расположение двух символов:

- знак `@`;
- первая точка.

## indexOf

Когда известно расположение символов в строке электронной почты, это позволяет использовать методы работы со строками для их разбора. Для этого используется метод `indexOf()`:

```
// получение позиций в строке
int dotPos = email.indexOf('.');
int atPos = email.indexOf('@');
```

Таким образом, метод `indexOf()` возвращает целочисленное значение, указывающее на позицию запрашиваемого символа. Возможный диапазон возвращаемых значений — от нуля до конца строки. Если запрашиваемый символ или строка не найдены, возвращается значение `-1`. Помните, что в Java используются нулевые индексы и нумерация, а значит, позиции символов в строке электронного письма выглядят так, как показано в табл. 3.1.

***Таблица 3.1.** Позиции символьных индексов строки email*

| v | i | c | t | o | r | i | a | . | p | l | o | e | t | z | @ | L | a | r | g | e | c | o | r | p | . | c | o | m |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 |

В сопоставлении, показанном в табл. 3.1, хранящееся в `atPos` значение должно быть равно 15, а значение, хранящееся в `dotPos`, должно быть равно 8.

> **Примечание.** По умолчанию функция `indexOf()` возвращает индекс первого вхождения переданного символа или строки.

## substring

Теперь с помощью метода `substring()` выделим имя сотрудника и сохраним его в переменной `firstName`:

```
String firstName = email.substring(0, dotPos);
```

Мы указываем подстроку с начальным индексом и точкой остановки после последнего символа, который нам нужен. Аналогичным образом можно создать переменные для `lastName` и `company`:

```
String lastName = email.substring(dotPos + 1, atPos);
String company = email.substring(atPos + 1, dot2Pos);
```

Добавим операторы для вывода строковых значений:

```
System.out.println("Имя: " + firstName);
System.out.println("Фамилия: " + lastName);
System.out.println("Компания: " + company);
```

Выполнение кода должно дать следующие результаты:

```
Имя: victoria
Фамилия: ploetz
Компания: largecompany
```

## *toUpperCase*

Имя нашего пользователя записано не в правильном регистре (то есть, не с заглавной буквы в начале каждого имени). Это можно исправить, написав новый метод `properCase`:

```
private static String properCase(String name) {
    char firstLetter = Character.toUpperCase(name.charAt(0));

return firstLetter + name.substring(1);
}
```

Наш метод `properCase()` принимает строковую переменную с именем `name` и начинает с определения символа `firstLetter` (первой буквы в строке `name`). Затем выполняется метод `toUpperCase()` для этой буквы, "заставляя" ее быть в верхнем регистре. Затем мы возвращаем значение `firstLetter`, конкатенированное с подстрокой `name` от позиции 1 до конца строки `name`.

Когда метод `substring()` вызывается с единственным числовым параметром, он, по сути, работает с начальной позицией для разделения строки на части, но без конечной. Поэтому он вернет всё оставшееся значение строки, начиная с этого индекса.

> **Примечание.** Метод `toUpperCase()` является частью классов `String` и `Character`. Кроме того, выполнение `toUpperCase()` для строки, полностью или частично состоящей из символов верхнего регистра, не окажет никакого влияния на символы верхнего регистра, в то время как все символы нижнего регистра будут преобразованы в верхний.

Теперь мы можем вызывать метод `properCase()` перед операторами `System.out.println`:

```
firstName = properCase(firstName);
lastName = properCase(lastName);
```

Повторный запуск нашего кода должен дать результаты, аналогичные этим:

```
Имя: Victoria
Фамилия: Ploetz
Компания: largecompany
```

## *toLowerCase*

Поскольку существует метод `toUpperCase()`, то есть и метод `toLowerCase()` в классе `String`. Этот метод может быть полезен при обработке пользовательских полей ввода, таких как адреса электронной почты. Адреса электронной почты не чувствительны к регистру и должны храниться и обрабатываться в нижнем регистре.

Однако иногда мы можем столкнуться с пользователем, который вводит их в верхнем регистре или даже в смешанном регистре:

```
String messedUpEmail = "bObJoNeS@BIGGERCOMPANY.com";
System.out.println("messedUpEmail.toLower() = " +
    messedUpEmail.toLowerCase());
```

Таким образом, можно легко применить метод `toLowerCase()` для отображения строки в нужном регистре.

Если добавить этот код в завершение метода `main()` и запустить его, то результат будет выглядеть следующим образом:

```
Имя: Victoria
Фамилия: Ploetz
Компания: largecompany

messedUpEmail.toLower() = bobjones@biggercompany.com
```

Как было показано, строковые методы, такие как `charAt()`, `toUpperCase()`, `toLowerCase()`, `indexOf()` и `substring()`, могут быть очень полезны при разборе потоков входящих данных.

# Сравнение строк

Равенство символов и других примитивных типов не вызывает затруднений. Однако равенство строк может оказаться более сложным.

После оператора `println` добавим проверку `if`, чтобы узнать, является ли имя нашего пользователя `Victoria`:

```
if (firstName == "Victoria") {
    System.out.println("Ваше имя Victoria!");
} else {
    System.out.println("Извините, Ваше имя НЕ Victoria.");
}
```

Выполнение этого кода дает следующий результат:

```
Имя: Victoria
Фамилия: Ploetz
Компания: largecompany

messedUpEmail.toLower() = bobjones@biggercompany.com

Извините, Ваше имя НЕ Victoria.
```

Переменная `firstName` определенно равна строковому значению `Victoria`, так в чем же может быть дело? Проблема в том, что, поскольку строки не являются примитивными типами, строковые переменные хранят только указатель на место в памяти. Здесь мы сравниваем идентификатор ссылки, указывающий на строковое значение. В данном случае это не даст желаемого результата.

К счастью, у строк есть свой собственный оператор равенства, известный как метод `equals()`:

```
if (firstName.equals("Victoria")) {
    System.out.println("Ваше имя Victoria!");
} else {
    System.out.println("Извините, Ваше имя НЕ Victoria.");
}
```

Теперь, когда запустим код, то получим ожидаемые результаты:

```
Имя: Victoria
Фамилия: Ploetz
Компания: largecompany

messedUpEmail.toLower() = bobjones@biggercompany.com

Ваше имя Victoria!
```

Как можно увидеть, оператор равенства строк `equals` — это то, что нужно использовать при проверке, имеют ли строки одинаковое значение.

## Сравнение строковых суффиксов

Нам может понадобиться проверить, равен ли конец строки определенному значению. К счастью, в поставляемом в Java классе `String` есть метод, позволяющий удовлетворить и это требование.

Начнем с создания нового метода с именем `isBusinessEmail`. Этот метод будет возвращать булево значение `boolean` и принимать в качестве параметра строку с адресом электронной почты. Идея этого метода заключается в том, чтобы проверить окончание строки адреса электронной почты на соответствие некоторым известным окончаниям, которые указывают на то, что:

- пользователь является студентом;
- пользователь указал личный адрес электронной почты.

Это распространенный фильтр, используемый в веб-формах, когда компании хотят сосредоточить свои усилия на людях, которые могут стать хорошими потенциальными клиентами для продаж:

```
private static boolean isBusinessEmail(String email) {
    boolean validEmail = true;

    if (email.endsWith("@gmail.com")) {
        validEmail = false;
    } else if (email.endsWith(".edu")) {
        validEmail = false;
    }

    return validEmail;
}
```

Логика этого метода заключается в том, что сначала мы инициализируем булево значение `validEmail` значением `true`. Затем проверяем, является ли указанный адрес электронной почты адресом Gmail (заканчивающимся на @gmail.com), и устанавливаем значение `validEmail` равным `false`. Это указывает на то, что пользователь указал *личный* адрес электронной почты, в то время как нам нужен его *рабочий* адрес. После этого выполняется еще одна проверка `if`, чтобы узнать, заканчивается ли адрес электронной почты строкой `.edu`, что указывает на то, что пользователь, скорее всего, является студентом университета. В этом случае значение `validEmail` также будет равно `false`. Наконец, мы возвращаем значение `validEmail` вызывающему методу.

Чтобы проверить это в действии (в конце нашего метода `main()`), можно определить новый адрес электронной почты с именем `email2`, который будет иметь одно из этих недопустимых окончаний:

```
String email2 = "khadiya8821@mnsu.edu";
```

Затем можно создать две if-проверки для наших адресов электронной почты:

```
if (isBusinessEmail(email)) {
    System.out.println(email + " – соответствует!");
} else {
    System.out.println(email + " – не соответствует!");
}
if (isBusinessEmail(email2)) {
    System.out.println(email2 + " – соответствует!");
} else {
    System.out.println(email2 + " – не соответствует!");
}
```

Если мы запустим наш полный (на данный момент) код, результат будет выглядеть примерно так:

```
Имя: Victoria
Фамилия: Ploetz
Компания: largecompany
messedUpEmail.toLower() = bobjones@biggercompany.com

Ваше имя Victoria!

victoria.ploetz@largecompany.com – соответствует!
khadiya8821@mnsu.edu – не соответствует!
```

Как видно, электронная почта Хадии (Khadiya) не соответствует (заканчивается на `.edu`), в то время как электронная почта Виктории (Victoria) отвечает нашим требованиям к адресу деловой электронной почты.

## Сравнение префиксов строк

Можно также проверить начало строкового значения. Допустим, необходимо выполнить специальную обработку пользователей, чьи фамилии начинаются с `Pl`. Для этого мы можем использовать метод `startsWith()` класса `String` в Java.

Это возможно реализовать с помощью простой проверки `if` в конце кода в нашем методе `main()`:

```
if (lastName.toUpperCase().startsWith("PL")) {
    System.out.println("У " + firstName + " фамилия "
        + lastName + " определенно начинается с Pl.");
}
```

Как видно, мы просто проверяем, есть ли у строки `lastName` префикс `PL`, и выводим на экран сообщение, если есть.

> **Примечание.** Помните, что строки Java чувствительны к регистру. Поэтому при проверке равенства для полученной из пользовательского ввода строки лучше всего принудительно установить верхний или нижний регистр. Таким образом, не придется учитывать несколько типов регистров (верхний или нижний) для каждого символа в строке.

Теперь, когда запустим наш код, последняя строка вывода будет выглядеть следующим образом:

```
У Victoria фамилия Ploetz определенно начинается с Pl.
```

## *contains*

Еще одним полезным методом класса `String` в Java является метод `contains()`. С его помощью можно проверить, существует ли меньшая строка внутри большей. Хорошим вариантом использования этого метода может стать телефонный номер. Предположим, что необходимо применить специальную обработку к телефонным номерам с кодом центрального офиса 188. В целях тестирования можно задать пару телефонных номеров (в виде строк) в конце метода `main()` и проверить:

```
String phoneNumber = "444-867-5309";
String phoneNumber2 = "444-188-2300";
if (phoneNumber.contains("-188-")) {
    System.out.println(phoneNumber + " содержит необходимое число 188!");
}

if (phoneNumber2.contains("-188-")) {
    System.out.println(phoneNumber2 + " содержит необходимое число 188!");
}
```

Если мы выполним код, то результат будет выглядеть следующим образом:

```
444-188-2300 содержит необходимое число 188!
```

Это происходит потому, что только `phoneNumber2` содержит строку, соответствующую `"-188-"`, а это значит, что проверка `if` для другой переменной (`phoneNumber`) ничего не дает. В этом случае важно включить дефис (`-`) вокруг `188`, чтобы наши `contains()` не совпадали с другими вхождениями `188`.

# Регулярные выражения

До сих пор мы показывали простые способы проверки префикса или суффикса строки на наличие определенного значения. Но что, если наши потребности более сложны? Регулярные выражения — это мощный инструмент, позволяющий быстро и эффективно проверять строки.

> **Примечание.** Термин *регулярное выражение* часто сокращают до regex или regexes (во множественном числе).

Чтобы использовать регулярные выражения (*Baeldung 2022b*), мы импортируем два класса стандартной библиотеки Java: `Pattern` и `Matcher`. Давайте добавим их в наш класс `WorkingWithStrings`:

```
package chapter3;

import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class WorkingWithStrings {
```

Чтобы применить регулярное выражение к строке, сначала определим шаблон, а затем попытаемся сопоставить строку с этим шаблоном. Продолжая наш пример с `contains` и номером телефона, можно построить шаблон (в конце метода `main()`) следующим образом:

```
Pattern phone188Pattern =
Pattern.compile("[0-9]{3}\\-188\\-[0-9]{4}");
```

Давайте разберем его по частям:

- `[0-9]{3}`

  `[0-9]` в скобках означает, что мы ищем одну цифру от нуля до девяти. Сразу после него стоит `{3}`, указывающая на то, что необходимо выполнить предыдущее совпадение три раза. По сути, это означает, что мы ищем строку, начинающуюся с любых трех цифр;

- `\\-`

  Означает, что нужно искать символ дефис после первых трех цифр. Поскольку символ тире имеет особое значение в синтаксисе регулярных выражений, необходимо экранировать его двумя обратными слешами `\\`;

- `188`

  Здесь используется строка 188, так как необходимо искать телефонные номера с кодом центрального офиса 188;

- `\\-`

  И снова необходим символ дефис;

- `[0-9]{4}`

  Аналогично первому шаблону в данном регулярном выражении, эта часть ищет совпадение четырех последовательных цифр.

Определив шаблон, давайте создадим два объекта класса `Matcher`:

```
Matcher phone188Matcher = phone188Pattern.matcher(phoneNumber);
Matcher phone188Matcher2 = phone188Pattern.matcher(phoneNumber2);
```

По сути, нам нужен объект `Matcher` для каждой строки, которую мы пытаемся сопоставить. Теперь можно строить наши if-проверки с помощью метода `find()` для каждого объекта `Matcher`:

```
if (phone188Matcher.find()) {
    System.out.println(phoneNumber + " содержит необходимое число 188! (regex)");}

if (phone188Matcher2.find()) {
    System.out.println(phoneNumber2 + " содержит необходимое число 188! (regex)");
}
```

Запуск кода должен показать это в конце вывода:

```
444-188-2300 содержит необходимое число! (regex)
```

По сути, мы выполнили ту же операцию, что и ранее с помощью метода `contains()`, но с более тонкой настройкой. Преимущество использования регулярного выражения в том, что можно более абстрактно подходить к вопросу о том, что именно необходимо сопоставить. Например, методы `contains()`, `endsWith()` и `startsWith()` требуют явных строк для сопоставления. Но регулярные выражения позволяют сказать, например, что *"я хочу убедиться в том, что мы сопоставляем три числа"*, так что это определенно обеспечивает бóльшую гибкость. Можно использовать регулярное выражение для проверки правильности ввода телефонных номеров:

```
Pattern validPhonePattern = Pattern.compile("[0-9]{3}\\-[0-9]{3}\\-[0-9]{4}");
Matcher phoneMatcher = validPhonePattern.matcher(phoneNumber);
Matcher phoneMatcher2 = validPhonePattern.matcher(phoneNumber2);

if (phoneMatcher.find()) {
    System.out.println(phoneNumber + " является корректным телефонным номером (regex)");
}

if (phoneMatcher2.find()) {
    System.out.println(phoneNumber2 + " является корректным телефонным номером! (regex)");
}
```

В результате выполнения этого кода в конце нашего вывода должно получиться следующее:

```
444-867-5309 является корректным телефонным номером! (regex)
444-188-2300 является корректным телефонным номером! (regex)
```

Регулярные выражения также дают большую гибкость при работе со строками. Допустим, некоторые наши пользователи решили войти в систему или использовать электронную почту с псевдонимом вместо своего имени. Могут ли регулярные выражения помочь нам в этом?

Давайте начнем с определения некоторых строк псевдонимов в конце метода `main()`. Будем использовать по два для Виктории и нового пользователя по имени Роберт (некоторые люди по имени Роберт предпочитают, чтобы их называли Роб или Боб):

```
String nickname = "Toria";
String nickname2 = "Vicky";
String robert = "Robert";
String nickname3 = "Rob";
String nickname4 = "Bob";
```

Начнем с проверки `Robert`. Этот шаблон не представляет сложности:

```
System.out.println("\nШаблон для Bob:");
Pattern bobPattern = Pattern.compile("[B|R]ob");
```

Этот шаблон очень прост. Необходимо найти соответствие со строками `Bob` или `Rob`. Символ вертикальная черта '|' является регулярным выражением или оператором. Поэтому начало шаблона `[B|R]` проверяет, является ли первый символ в строке заглавной буквой `B` или заглавной буквой `R`. Следующие два символа `ob` — это литералы, и мы ищем соответствие по второму и третьему символам, которые соответствуют строчной букве `o`, а затем строчной букве `b`.

Поскольку мы собираемся выполнить операцию regex несколько раз, давайте создадим метод, который возьмет на себя всю *тяжесть* этой работы. Назовем этот метод `matchName`, и он будет принимать объект класса `Pattern` и имя в виде строки:

```
private static void matchName(Pattern pattern, String name) {
    Matcher matcher = pattern.matcher(name);
    if (matcher.find()) {
        System.out.println("Совпадение найдено! Добро пожаловать, " + name + "!");
} else {
        System.out.println("Извините," + name + ", совпадений не найдено.");
    }
}
```

Начнем метод с того, что возьмем переданный объект шаблона `pattern` и инстанцируем объект класса `Matcher`, используя имя `name` в качестве параметра. Определив `matcher`, мы вызовем метод `find()` в проверке `if` и выведем соответствующее сообщение.

Теперь можно вызвать этот метод для всех шести имен, которые мы определили:

```
matchName(bobPattern, nickname);
matchName(bobPattern, nickname2);
matchName(bobPattern, nickname3);
matchName(bobPattern, nickname4);
matchName(bobPattern, robert);
matchName(bobPattern, firstName);
```

Если запустить наш код, то в конце вывода увидим следующее:

```
Шаблон для Bob:
Извините, Toria, совпадений не найдено.
Извините, Vicky, совпадений не найдено.
Совпадение найдено! Добро пожаловать, Rob!
Совпадение найдено! Добро пожаловать, Bob!
Совпадение найдено! Добро пожаловать, Robert!
Извините, Victoria, совпадений не найдено.
```

Как видно из результатов, `Rob`, `Bob` и `Robert` совпали! `Victoria` и ее никнеймы не были сопоставлены, что правильно.

Как же нам сопоставить никнеймы Виктории? Начнем с создания экземпляра объекта `Pattern`, чтобы определить, как они должны быть сопоставлены:

```
System.out.println("\nШаблон для Victoria:");
Pattern victoriaPattern = Pattern.compile("[Vic|][[T|t]oria]");
```

Этот шаблон состоит из двух отдельных частей.

- `[Vic|]`

  Эта часть ищет совпадение либо со строкой, начинающейся либо с `Vic`, либо ни с чего. Обратите внимание, что вертикальная черта `|` (оператор "или") находится после `Vic`, и после нее ничего нет. По сути, если строка начинается с `Vic`, то у нас есть совпадение. Если нет, то за *начало* строки принимается следующая часть.

- `[[T|t]oria]`

  Эта часть ищет совпадения со строками `Toria` или `toria`. В последнем случае предполагается, что это суффикс (конец) строки.

Теперь вызовем метод `matchName()`, передав ему шаблон `victoriaPattern` для тех же шести имен:

```
matchName(victoriaPattern, nickname);
matchName(victoriaPattern, nickname2);
matchName(victoriaPattern, nickname3);
matchName(victoriaPattern, nickname4);
matchName(victoriaPattern, robert);
matchName(victoriaPattern, firstName);
```

В результате выполнения кода в конце вывода должно получиться следующее:

```
Шаблон для Victoria:
Совпадение найдено! Добро пожаловать, Toria!
Совпадение найдено! Добро пожаловать, Vicky!
```

```
Извините, Rob, совпадений не найдено.
Извините, Bob, совпадений не найдено.
Извините, Robert, совпадений не найдено.
Совпадение найдено! Добро пожаловать, Victoria!
```

Все варианты `Bob` не совпали, `Toria` совпала из-за второй части шаблона, `Vicky` совпала из-за определения шаблона `[Vic|]`, а `Victoria` совпала, потому что совпали обе части нашего определения шаблона.

Как видно, регулярные выражения могут быстро усложняться, но это отличный способ проверки правил ввода с помощью мощных, но абстрактных инструментов.

# Заключение

В этой главе мы обсудили различные способы работы с символьными и строковыми данными. Сначала мы обсудили символьные типы данных и посмотрели, как они работают с конкатенацией и кодами ASCII/UTF. Мы рассмотрели различные методы класса `String`, предоставляющего множество инструментов для управления и разбора текстовых данных. Наконец, мы поработали над равенством строк, обратив внимание на то, чем оно отличается от проверки равенства примитивных типов. Часть нашей работы с равенством строк привела нас к рассмотрению регулярных выражений и способов использования их в качестве мощных инструментов.

В следующей главе мы продолжим работать со строковыми данными, изучая такие структуры, как массивы и коллекции. Мы также рассмотрим записи Java.

# Важно помнить

- Набором символов по умолчанию для Java (начиная с Java 18) является UTF-8.
- Класс-обертка `Character` имеет множество полезных методов, помогающих нам разбирать и работать с символьными данными.
- Существуют разнообразные ASCII/UTF-символы, которые можно использовать для придания изюминки вашим программам.
- Найти первое вхождение символа в строке можно с помощью метода `indexOf()`.
- Нумерация символов в строках Java начинается с нуля (первый индекс равен нулю, а не единице).
- Меньшая часть строки может быть получена с помощью метода `substring()`.
- Сложные регулярные выражения дают нам чрезвычайно высокий уровень гибкости и настраиваемости при выполнении операций равенства строк.

Глава 4

# Массивы, коллекции и записи

## Введение

В этой главе мы рассмотрим различные способы хранения и группировки данных. Ранее мы уже обсудили некоторые из основных тем этой главы. Теперь мы углубимся в изучение таких понятий, как массивы, коллекции и записи.

Также мы обсудим упорядоченные коллекции — новую возможность в Java 21. Упорядочивание позволит нам отслеживать порядок добавления элементов в коллекцию и значительно обогатит работу со списками, множествами и словарями.

## Структура

В этой главе мы рассмотрим следующие темы.

- Массивы.
- Множества.
- Списки.
- Словари.
- Записи.

## Цели

В этой главе мы подробно рассмотрим способы, с помощью которых Java позволяет группировать данные и работать с ними в памяти. Мы сосредоточимся на следующем:

- создание массивов из типов данных, с которыми мы уже знакомы;
- понимание различных моделей поведения и возможностей множеств, списков и словарей;

- узнаем, как хранить данные в различных типах коллекций и извлекать их из них;
- изучение и понимание типа записи в Java.

# Массивы

*Массивы* — это простой способ хранения нескольких индексированных значений одного и того же типа данных. Мы уже несколько раз использовали массивы в предыдущих главах. При работе с файлами и чтении строк ввода массивы определялись следующим образом:

```
String[] gameColumns = gameLine.split(",");
```

Тогда использование созданных методом `split()` массивов помогло разделить строки файла на колонки. Кроме того, каждый раз, когда мы начинаем новую программу, мы определяем массив:

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

Все верно, параметр `args`, который определяется в каждом методе `main()`, — это массив строк. До сих пор мы ничего не делали с этим массивом. Однако это простой способ принимать и организовывать входные данные в наших программах.

Создайте новый Java-класс с именем `JavaArguments` в пакете `chapter4` и убедитесь, что в нем есть метод `static void main`:

```
package chapter4;
public class JavaArguments {
    public static void main(String[] args) {
        System.out.println("Вы предоставили " + args.length
        + " аргумент(а/ов):");
        for (int index = 0; index < args.length; index++) {
            System.out.printf("Аргумент #%d:",index);
            System.out.println(args[index]);
        }
    }
}
```

Суть идеи проста. Мы выводим количество аргументов (`args.length`) и итерируем по массиву `args`. Затем выводим каждый аргумент вместе с его числовым индексом. Если запустить эту программу без указания аргументов, она выдаст следующий результат:

```
Вы предоставили 0 аргумент(а/ов):
```

Итак, как же передать аргументы в Java-программу? Есть два способа сделать это. Первый — из командной строки. Если открыть терминальную сессию, то можно передать аргументы Java-программе из каталога проекта. Нам понадобится подка-

талог target/classes. Находясь в этом каталоге, можно использовать основной исполняемый файл Java для запуска нашего скомпилированного класса с тем количеством аргументов, которое мы решили добавить:

```
cd target/classes

pwd

/Users/aaronploetz/Documents/workspace/CodeWithJava21/target/classes

java chapter4.JavaArguments The cake is a lie!Вы предоставили 5 аргумент(а/ов):
Аргумент #0:The
Аргумент #1:cake
Аргумент #2:is
Аргумент #3:a
Аргумент #4:lie!
```

**Примечание.** В зависимости от того, как ваша IDE создает файлы классов в целевом каталоге, может потребоваться компиляция файла JavaArguments.java с помощью команды `javac`.

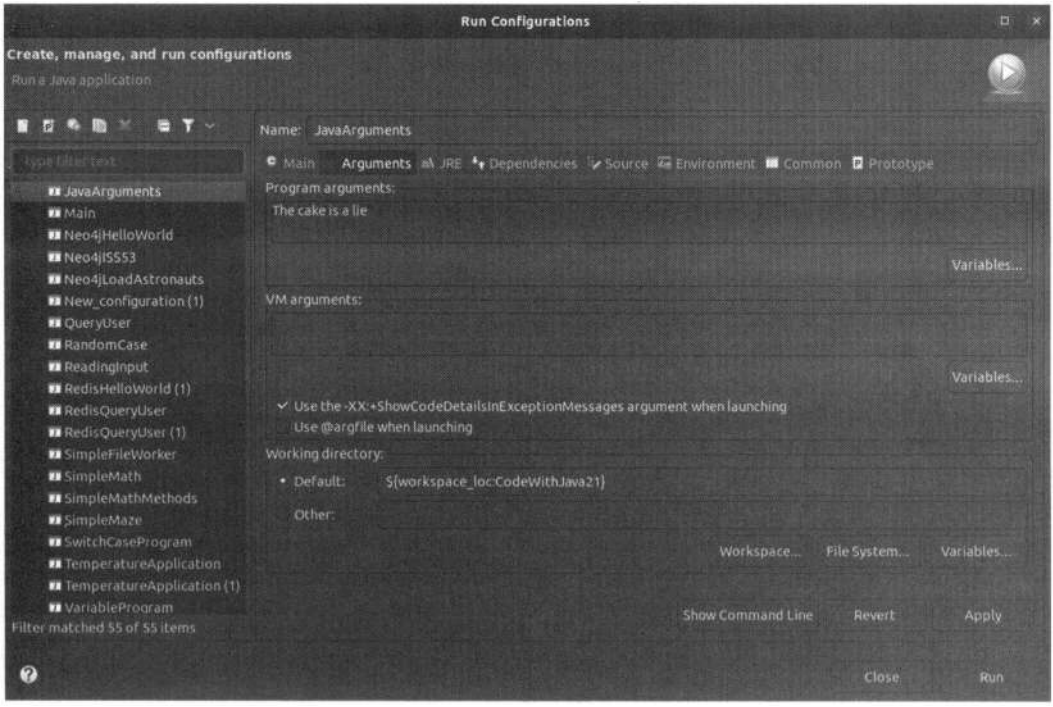



**Рис. 4.1.** Вкладка **Arguments** окна **Run Configurations** в Eclipse IDE

Второй способ передачи аргументов — из вашей IDE. Eclipse, например, позволяет использовать **Run Configurations**. Эта функция дает возможность сохранять и повторно использовать аргументы времени выполнения и переменные окружения. В **Eclipse** эту функцию можно найти в свойствах проекта, как показано на рис. 4.1.

В качестве аргументов программы введем простое предложение: `The cake is a lie!`. Если запустить программу с этой строкой в качестве аргументов, то получим следующий результат:

```
Вы предоставили 5 аргумент(а/ов):
Аргумент #0:The
Аргумент #1:cake
Аргумент #2:is
Аргумент #3:a
Аргумент #4:lie!
```

Массив аргументов может быть полезным инструментом для передачи ограниченного количества параметров в Java-программы.

При желании можно просмотреть массив `args` в обратном порядке. Для этого достаточно просто изменить цикл `for`:

```
for (int index = args.length - 1; index > -1; index--) {
```

Запуск кода должен привести к следующему результату:

```
Вы предоставили 5 аргумент(а/ов):
Аргумент #4:lie
Аргумент #3:a
Аргумент #2:is
Аргумент 1#1:cake
Аргумент #0:The
```

## Многомерные массивы

Массивы также могут иметь несколько измерений в своих индексах. Чтобы продемонстрировать это, давайте создадим новый класс Java. Назовем его `CoordinateArrays` и убедимся, что он находится в пакете `chapter4` и содержит метод `main()`.

Начнем с того, что определим в классе двумерный private-массив для хранения местоположения героев на игровой сетке:

```
public class CoordinateArrays {
    private static String [][] gameGrid = new String[5][5];
    public static void main(String[] args) {
```

Этот оператор определяет новый массив `gameGrid`, содержащий максимум 5x5 (25) значений. Определив массив, можно добавить положение наших героев:

```
gameGrid[3][2] = "Byorki - Ranger";
gameGrid[3][3] = "K'lar - Fighter";
```

```
gameGrid[4][3] = "Tyrenni - Wizard";
gameGrid[1][3] = "Athena - Rogue";
gameGrid[0][1] = "Jarrod - Cleric";
```

Теперь в `gameGrid` хранятся данные о местоположении наших героев, как показано на рис. 4.2.

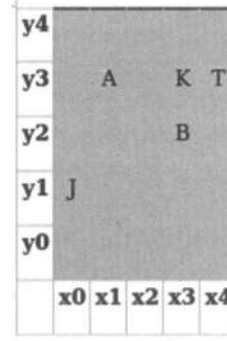



**Рис. 4.2.** Сетка с координатами X, Y для наших героев, обозначенных по первой букве в их имени

Можно отобразить положение конкретного героя, указав его координаты в сетке:

```
System.out.println(gameGrid[4][3]);
```

Если запустить этот код, то увидим следующий результат:

```
Tyrenni - Wizard
```

> **Примечание.** Java выделяет память исходя из максимально возможного размера определяемой переменной. С многомерными массивами нужно быть осторожными, чтобы не задать их слишком большими, иначе они будут занимать большую часть (или всю) доступной памяти в Java.

Давайте напишем новый private static метод для вывода сетки:

```
private static void printGrid() {
    System.out.println();
```

Начнем с печати пустой строки, чтобы сохранить нужный интервал между выводимыми данными. Затем используем пару вложенных циклов `for` для итерации по обоим измерениям нашего массива `gameGrid`. Проблема в том, что сетка (как показано на рис. 4.2) работает от нуля до четырех, увеличиваясь слева направо и снизу вверх, а наш стандартный вывод работает слева направо и сверху вниз. Поэтому нам придется выполнять итерации по оси Y в *обратном направлении*:

```
for (int indexY = 4; indexY > -1; indexY--) {
    // вывод индекса по оси Y
    System.out.print(indexY);
    for (int indexX = 0; indexX < 5; indexX++) {
```

Далее необходимо проверить, является ли текущий элемент `gameGrid` нулевым или пустым. Если да, то это незанятое пространство, и мы напечатаем квадратный кон-

тур. Мы выполним эту проверку, потому что программы обычно работают быстрее, если проверять условия, которые с наибольшей вероятностью будут истинными:

```
        if ((gameGrid[indexX][indexY] == null)
|| (gameGrid[indexX][indexY].equals(""))) {
            // вывести символ квадрата
            System.out.print((char)9634);
```

Далее построим условие `else`, чтобы вывести значение для текущей записи `gameGrid`:

```
            // вывод первой буквы имени персонажа
            System.out.print(
gameGrid[indexX][indexY].charAt(0));
            }
        }
        System.out.println();
    }
    // вывод индекса по оси X
    System.out.println(" 01234");
}
```

Мы завершаем внутренний цикл с помощью `println()`, чтобы перейти к следующей строке. Наконец, мы завершаем внешний цикл, печатая числа для оси X в виде индекса в нижней части сетки. Теперь вернитесь к нашему методу `main()` и добавьте вызов нового метода `printGrid()` в конце.

Запуск этого кода должен привести к следующему результату:

```
Tyrenni - Wizard

4□□□T□
3□□BK□
2□□□□□
1□□□A□
0□J□□□
 01234
```

Также можно довольно легко перемещать одного из наших героев. Создайте новый private static метод `movePlayer` и укажите для него следующие три параметра:

- `char direction;`
- `int currentX;`
- `int currentY.`

В методе `movePlayer` мы начнем с инициализации двух новых переменных значениями координат `currentX` и `currentY`:

```
private static void movePlayer(char direction,
int currentX, int currentY) {

    int newX = currentX;
    int newY = currentY;
```

Далее выполним оператор switch/case для переменной direction и соответствующим образом скорректируем наши новые координаты:

```
switch (Character.toUpperCase(direction)) {
case 'N':
    // север
    newY++;
    break;
case 'S':
    // юг
    newY--;
    break;
case 'W':
    // запад
    newX--;
    break;
default:
    // восток
    newX++;
}
```

Наш герой может двигаться на север и юг, увеличивая и уменьшая (соответственно) значение оси newY. На восток (по умолчанию) и на запад можно перемещаться, увеличивая и уменьшая (соответственно) значение оси newX. Для фактического перемещения будем сохранять значение, хранящееся в координатах currentX и currentY:

```
String hero = gameGrid[currentX][currentY];
// движение игрока
gameGrid[newX][newY] = hero;
// стираем старую координату
gameGrid[currentX][currentY] = "";
```

Затем установим для значения hero его новое местоположение на сетке и завершим процесс установкой для текущего местоположения значения пустой строки. Наконец, выведем строку, описывающую действия по перемещению:

```
System.out.println("Переместился" + hero + " с ("
+
currentX + "," + currentY
+
") на " + "(" + newX + "," + newY + ")");
```

Вернувшись в метод main(), добавим вызов movePlayer(), задав ему направление N (север) и текущее местоположение Афины (Athena):

```
// перемещаем Athena с 1,3 на 1,4
movePlayer('N', 1, 3);
printGrid();
```

Запуск кода должен вывести следующие результаты:

```
Tyrenni - Wizard
4□□□□□
3□A□KT
2□□□B□
1J□□□□
0□□□□□
 01234
Переместился Athena — Rogue с (1,3) на (1,4)

4□A□□□
3□□□KT
2□□□B□
1J□□□□
0□□□□□
 01234
```

Если рассмотреть сетку внизу, то можно увидеть, что Athena (представленная буквой A) переместилась на север (вверх) с оси Y, равной 3, на ось Y, равную 4.

# Коллекции и словари

Java предоставляет и другие структуры данных, способных хранить данные одного типа, но обладающих гораздо большей функциональностью и универсальностью. Эти структуры известны как коллекции.

Существует три основных типа структур коллекций Java, которые мы рассмотрим:

- множества;
- списки;
- словари.

Как и все в Java, все это является объектом. Это означает, что всё наследуется от базового класса `Object`. Кроме того, множества и списки также наследуются от базового класса `Collection`. Хотя словари и не наследуются от `Collection`, они все равно являются частью фреймворка Java `Collections` и имеют схожие методы и способы работы с данными внутри них.

Для наших примеров с коллекциями мы создадим новый класс Java с именем `WorkingWithCollections`. Он должен находиться внутри пакета `chapter4`, иметь метод `main()` и содержать указанные импорты:

```
package chapter4;
import java.util.Collections;
```

```
public class WorkingWithCollections {

    public static void main(String[] args) {

    }
}
```

Прежде чем мы пойдем дальше, давайте создадим новый статический метод для вывода содержимого коллекции, с которой будем работать. Добавьте новый метод с именем `printCollection` в конец нашего нового класса. Он должен быть типа `void` (ничего не возвращает) и принимать в качестве единственного параметра объект типа `Collection`:

```
private static void printCollection(Collection collection) {

    for (Object element : collection) {
        System.out.printf("%s, ", element.toString());
    }

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

**Примечание.** Технически можно обойтись без вызова метода `toString()` в элементе. Методы `print()`, `printf()` и `println()` неявно преобразуют выходной параметр в строку.

Этот метод должен быть способен отобразить содержимое любого множества или списка, поскольку все они наследуются от базового класса `Collection`.

## Множества

Простейшим типом коллекции является *множество* (иногда их называют наборами). Множество — это уникальная коллекция из нескольких элементов данных. Существуют различные типы множеств с некоторыми тонкими различиями, которые нам предстоит рассмотреть.

Давайте начнем с создания нового массива строк и инициализации его именами наших героев:

```
String[] heroes = {"Byorki", "K'lar", "Tyrenni", "Athena", "Jarrod"};
```

В последующих примерах мы должны использовать такие импорты:

```
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;
import java.util.TreeSet;
```

### HashSet

Далее создадим новый HashSet и загрузим в него данные из массива heroes (с помощью метода Collections addAll()):

```
Set<String> heroSet = new HashSet<>();
Collections.addAll(heroSet, heroes);
printCollection(heroSet);
```

Сначала объявляем heroSet типом Set<String>. Этот код сообщает Java, что heroSet будет представлять собой *множество строковых значений*. Однако Set — это абстрактный класс. Нам все равно нужно инициализировать его с помощью конкретного класса, например HashSet.

В HashSet мы задали пустое множество угловых скобок (<>). Таким образом, он наследует тот же класс элементов (String), который мы определили для абстрактного класса Set. Стоит отметить, что в старых версиях Java для внутреннего класса элемента также должна была быть указана реализация. Рассмотрим этот пример:

```
Set<String> heroSet = new HashSet<String>();
```

К счастью, так больше делать не нужно, в чем вы убедитесь на примерах реализаций коллекций в этой главе и во всей книге.

Инициализировав heroSet как пустой HashSet, мы использовали метод addAll() из класса Collections, чтобы заполнить heroSet значениями из массива heroes. Поскольку heroes определен как строковой массив (что соответствует классу элементов), метод загружает heroSet значениями из heroes. Выполнение кода должно привести к следующему результату:

```
Byorki, K'lar, Tyrenni, Athena, Jarrod,
```

Давайте добавим элемент в heroSet:

```
heroSet.add("Byorki");
```

Byorki уже есть, но мы добавим ее снова. Теперь давайте запустим код и проверим результат:

```
Byorki, K'lar, Tyrenni, Athena, Jarrod,
```

Именно так! Вывод будет таким же. Значения внутри множеств должны быть уникальными, поэтому наш вызов метода add() терпит неудачу. Давайте добавим другое имя:

```
heroSet.add("Rik");
```

Выполнение кода дает такой результат:

```
Byorki, K'lar, Rik, Tyrenni, Athena, Jarrod,
```

Помните, что HashSet не поддерживает порядка данных.

### *LinkedHashSet*

Давайте изменим определение `heroSet`. Мы закомментируем его и переопределим его реализацию, чтобы использовать `LinkedHashSet`:

```
// Set<String> heroSet = new HashSet<>();
Set<String> heroSet = new LinkedHashSet<>();
```

Также необходимо импортировать `LinkedHashSet`:

```
import java.util.LinkedHashSet;
```

Если не вносить никаких других изменений, выполнение кода приведет к следующему результату:

```
Byorki, K'lar, Tyrenni, Athena, Jarrod, Rik,
```

Обратите внимание, что `Rik` переместился в конец множества. Это произошло потому, что `LinkedHashSet` сохраняет порядок, в котором были добавлены элементы. Поскольку `Rik` был добавлен последним, он находится в конце множества.

### *TreeSet*

Изменим определение `heroSet`, на этот раз превратив его реализацию в `TreeSet`.

```
// Set<String> heroSet = new HashSet<>();
// Set<String> heroSet = new LinkedHashSet<>();
Set<String> heroSet = new TreeSet<>();
```

Не забудьте также выполнить `import TreeSet`:

```
import java.util.TreeSet;
```

И снова, без каких-либо других изменений, запуск нашего кода дает такие результаты:

```
Athena, Byorki, Jarrod, K'lar, Rik, Tyrenni,
```

Значения `heroSet` теперь располагаются в алфавитно-цифровом порядке. `TreeSet` реализует бинарную сортировку своих элементов. Порядок, в котором они были добавлены, не сохраняется, но иногда нам нужно, чтобы коллекция была отсортирована в алфавитно-цифровом порядке.

Удаление элемента из множества — простая задача:

```
heroSet.remove("Rik");
```

Добавив `remove()` для `Rik` и запустив код, мы получим следующий результат:

```
Athena, Byorki, Jarrod, K'lar, Tyrenni,
```

## Списки

Давайте перейдем к работе со списками. *Списки* — это простая коллекция элементов, упорядоченных по индексу. Они функционируют так же, как массивы, и так же

допускают дублирование записей. Подобно множествам, существует несколько различных реализаций списков, которые мы рассмотрим в этой статье.

Следующие импорты необходимы в работе с примерами:

```
import java.util.ArrayList;
import java.util.List;
import java.util.LinkedList;
```

Различные типы имеют некоторые тонкие различия, которые мы рассмотрим. Давайте начнем с создания нового массива строк и инициализации его именами наших героев:

```
String[] heroes = {"Byorki", "K'lar", "Tyrenni", "Athena", "Jarrod"};
```

### *ArrayList*

Давайте определим новый список строк под названием `monsterList` с типом реализации `ArrayList`:

```
List<String> monsterList = new ArrayList<>();
```

Определившись, добавим в него несколько элементов:

```
monsterList.add("Kobald");
monsterList.add("Skeleton");
monsterList.add("Zombie");
monsterList.add("Rats");
monsterList.add("Skeleton");
```

После этого вызовем `printCollection()`, чтобы вывести список в консоль:

```
printCollection(monsterList);
```

Выполнение кода добавит эту строку в наш вывод:

```
Kobald, Skeleton, Zombie, Rats, Skeleton,
```

Обратите внимание, что список монстров расположен в том порядке, в котором мы вводили элементы, а `Skeleton` присутствует в нем дважды. Теперь воспользуемся методом `sort()` из класса `Collection`, чтобы упорядочить `monsterList`:

```
Collections.sort(monsterList);
```

Запустив код, можно увидеть следующее содержимое `monsterList`:

```
Kobald, Rats, Skeleton, Skeleton, Zombie,
```

Как видите, элементы в `monsterList` теперь отсортированы по алфавитно-цифровому порядку. Это работает, потому что `List` определен для хранения строковых значений, а класс `String` в Java реализует интерфейс сравнения для работы со строками.

Удаление элемента из списка идентично тому, как элемент удаляется из множества. Выполним `remove()` для `Skeleton`:

```
monsterList.remove("Skeleton");
```

В результате выполнения кода содержимое `monsterList` будет выглядеть следующим образом:

```
Kobald, Rats, Skeleton, Zombie,
```

Как видите, первое вхождение `Skeleton` было удалено из списка. Однако второе вхождение `Skeleton` осталось. Это важный аспект, который следует учитывать при выполнении функции `remove()` для списка. В то время как первое вхождение удаляется, дубликаты именованного элемента не удаляются.

Метод `remove()` также работает по числовому индексу. Если бы требовалось удалить элемент `Skeleton` по индексу, это можно было бы сделать следующим образом:

```
monsterList.remove(2);
```

Также можно получить доступ к отдельным элементам в `ArrayList`, используя числовой индекс элемента. В конце нашего кода выведем второй элемент списка (индекс 1) с помощью метода `get()`:

```
System.out.println(monsterList.get(1));
```

Запустив код, вы получите следующий результат для `monsterList` и элемента в позиции 1 списка `monsterList`:

```
Kobald, Rats, Skeleton, Zombie,
Rats
```

> **Примечание.** Помните, что Java работает с нулевыми числовыми индексами. Первый элемент в списке имеет индекс 0, второй — индекс 1, третий — индекс 2 и так далее.

### *Vector*

Также можно определить `monsterList` с помощью специального класса `Vector`. Мы не будем приводить примеры для него, поскольку его поведение аналогично поведению `ArrayList`. Вот как будет выглядеть инстанцирование:

```
List<String> monsterList = new Vector<>();
```

В функциональном поведении `ArrayList` и `Vector` нет никакой разницы. Преимуществом создания списка типа `Vector` будет то, что `Vector`, как известно, является потокобезопасным. В результате при использовании `Vector` мы можем понести небольшой ущерб производительности. Поэтому, если приложение не использует потоки, `ArrayList` будет лучшим вариантом.

#### Безопасность потоков

По сути, состояние объекта может стать поврежденным или непоследовательным, если он используется многими потоками процессов одновременно. Безопасные для потоков объекты и типы используют такие методы, как *синхронизация*, чтобы обеспечить доступ к ним только одного потока за раз.

### LinkedList

LinkedList в Java — это реализация связного списка (традиционной структуры данных). По сути, элементы в связном списке сортируются по их позиции вставки относительно других элементов. Элементы всегда добавляются в конец связного списка.

Уникальность связного списка заключается в том, что элементы хранят ограниченные знания о других элементах списка. В нашем случае класс LinkedList в Java известен как *двусвязный список*. Он считается двусвязным, потому что каждый элемент имеет указатель на следующий и предыдущий элементы списка.

Мы создадим LinkedList для хранения данных о городах, которые встречаются нашим героям в их путешествиях. Чтобы получить полную функциональность доступных методов, нам нужно будет указать LinkedList в качестве базового и конкретного классов:

```
LinkedList<String> cityList = new LinkedList<>();
cityList.add("Elddim");
cityList.add("Crystwind");
cityList.add("Fallraen");
cityList.add("Meren");
cityList.add("Lang");

printCollection(cityList);
```

Запуск кода показывает эти значения для cityList:

```
Elddim, Crystwind, Fallraen, Meren, Lang,
```

Как наглядно видно на рис. 4.3, они действительно перечислены в том порядке, в котором были добавлены.

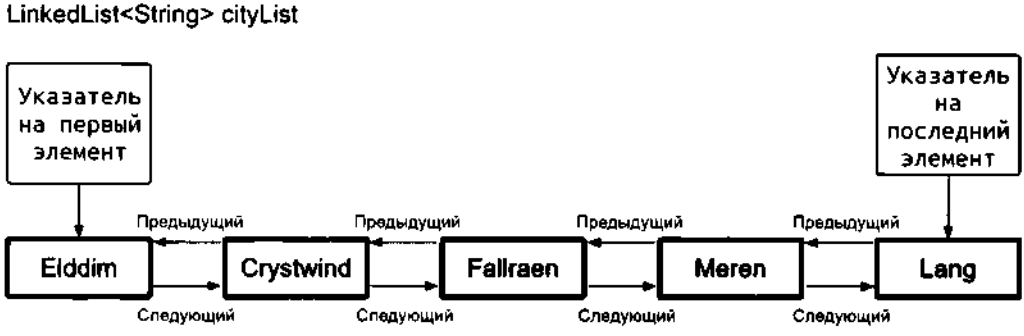



**Рис. 4.3.** Визуальное представление связного списка cityList

Как и в ArrayList, к элементам LinkedList можно обращаться по числовому индексу:

```
System.out.println(cityList.get(3));
```

Выполнение этого кода добавляет следующую строку в наш вывод:

```
Meren
```

Как уже было показано на примере ArrayList, также можно выполнять удаление по индексу или по элементу:

```
cityList.remove("Meren");
printCollection(cityList);
```

Разница в том, что за кулисами должны были произойти следующие шаги:

- элемент Meren найден;
- элемент Meren удален;
- элемент Lang установил свой указатель предыдущего элемента на Fallraen;
- элемент Fallraen установил свой указатель следующего элемента на Lang.

Запуск кода теперь возвращает следующее:

```
Elddim, Crystwind, Fallraen, Meren, Lang,
Meren
Elddim, Crystwind, Fallraen, Lang,
```

LinkedList также предоставляет другие уникальные и полезные методы. Если бы нам нужно было заглянуть в начало списка и вывести располагающийся там элемент, можно было бы использовать этот вариант:

```
System.out.println(cityList.peek());
printCollection(cityList);
```

Этот код добавляет следующие строки в вывод:

```
Elddim
Elddim, Crystwind, Fallraen, Lang,
```

То же самое мы можем сделать с последним элементом:

```
System.out.println(cityList.peekLast());
printCollection(cityList);
```

Это дает следующий результат:

```
Lang
Elddim, Crystwind, Fallraen, Lang,
```

Кроме того, мы можем "вытащить" элемент из списка при помощи poll:

```
System.out.println(cityList.poll());
printCollection(cityList);
```

Последние две строки вывода выглядят следующим образом:

```
Elddim
Crystwind, Fallraen, Lang,
```

Существует также метод pollLast(), демонстрирующий аналогичное поведение для последнего элемента в списке. Разница между методами peek() и poll() заключается в следующем:

- peek() возвращает значение первого элемента в списке;
- poll() возвращает значение первого элемента в списке и удаляет этот элемент из списка.

Мы обсудим метод pollLast() и его аналог pollFirst() в разделе, посвященном упорядоченным коллекциям.

> **Примечание.** Помимо того, что каждый элемент имеет указатели на предыдущий и следующий элементы, LinkedList содержит указатели на первый и последний элементы в списке.

Важно понимать, когда использовать ArrayList, а когда LinkedList. Все сводится к количеству элементов и типу операции. Технически, существует понятие, известное как *нотация Big O*. По сути, это способ представления операционной сложности операции или алгоритма. Добавление элемента в ArrayList — это операция O(n). Время, необходимое для выполнения операции (O), пропорционально количеству (n) элементов в списке. Это связано с тем, что при добавлении в ArrayList необходимо выполнить поиск по индексам, прежде чем понять, куда вставить новый элемент.

С другой стороны, добавление элемента в LinkedList — это (если использовать нотацию Big O) операция O(1). Это означает, что время, необходимое для выполнения операции (O), неизменно, независимо от количества элементов в списке. Это происходит потому, что по умолчанию все новые элементы в LinkedList попадают в конец. Java не нужно тратить дополнительное время на то, чтобы понять, куда их поместить.

В результате, если мы планируем использовать большие списки и больше добавлять в список, чем читать из него, то имеет смысл использовать LinkedList. С другой стороны, если мы собираемся тратить большую часть времени на чтение элементов списка (и не добавлять их слишком много), то ArrayList или Vector подойдут как нельзя лучше.

Более подробно мы обсудим связные списки в одной из следующих глав.

## Словари

Одним из наиболее уникальных типов коллекций являются *словари*[1]. Словари полезны для хранения небольших объемов данных, основанных на ключах/значениях. Некоторые базы данных, такие как Apache Cassandra, используют словари для обеспечения небольших уровней денормализации (хранения дополнительных свойств в строке данных) во избежание необходимости дополнительного запроса.

Прежде чем мы начнем работать со словарями, создадим небольшой статический метод для вывода содержимого нашего словаря в консоль. Нельзя использовать метод printCollection(), так как класс Map не наследуется от базового класса Collection.

[1] Иногда их еще называют "карты" или "мапы". — *Прим. перев.*

Добавьте новый метод с именем `printMap` в конец нашего класса. Он должен иметь тип `void` (без возвращаемого типа) и принимать в качестве единственного параметра объект типа `Map`:

```
private static void printMap(Map map) {
    System.out.println();
    Set<Object> keys = map.keySet();

    for (Object key : keys) {
        System.out.printf("%s: %s\n", key, map.get(key));
    }
}
```

Как видно, метод `printMap()` работает иначе, чем метод `printCollection()`. Для начала инициализируем множество (`Set`) `keys` значением, возвращаемым из метода `keySet()` объекта `map`. Множество `keys` имеет тип элементов `Object`, поскольку мы не знаем тип ключа словаря. Поэтому используем базовый класс `Object`, чтобы он мог принимать любые значения. Затем в цикле `for` перебираем все ключи с помощью метода `get()` объекта `map` и в методе `printf()` выводим связанное с этим ключом значение.

Также создадим метод `printKeys`, который будет выводить только ключи словаря. Поскольку у нас будет несколько примеров, где вывод ключей и значений может быть очень подробным, это даст нам преимущество в краткости.

Наш метод `printKeys()` имеет некоторое сходство с методом `printMap()`, как показано ниже. Добавьте следующий метод в конец нашего класса `WorkingWithCollections`:

```
private static void printKeys(Map map) {
    System.out.println();
    Set<Object> keys = map.keySet();

    for (Object key : keys) {
        System.out.printf("%s, ", key);
    }

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

В Java существует несколько типов реализаций `Map`. Наиболее часто используемой является `HashMap`, с которой и начнем.

Следующие импорты позволят нам выполнить приведенные ниже примеры:

```
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.TreeMap
```

### HashMap

В конце метода main() создадим новый словарь с именем spellbook. Он должен иметь тип ключа String и тип значения String. Затем используем метод put() словаря для добавления пар ключ/значение в spellbook и закончим вызовом метода printMap():

```
Map<String,String> spellbook = new HashMap<>();

spellbook.put("Огненный шар",
    "Огненный шар, наносящий по 8 единиц урона за уровень магии.");
spellbook.put("Исцеляющее прикосновение",
    "Прикосновение к раненому игроку восстанавливает по 5 очков
    за  уровень персонажа.");
spellbook.put("Молния",
    "Поток молний, наносящий по 10 единиц урона за уровень магии.");
spellbook.put("Создать воду",
    "Создает по 10 литров воды за уровень магии.");
spellbook.put("Превращение",
    "Превращает обычные предметы в золото.");

printMap(spellbook);
```

Выполнение нашего кода должно дать новый результат в итоге:

```
Огненный шар: Огненный шар, наносящий по 8 единиц урона за уровень магии.
Превращение: Превращает обычные предметы в золото.
Создать воду: Создает по 10 литров воды за уровень магии.
Исцеляющее прикосновение: Прикосновение к раненому игроку восстанавливает
по 5 очков за  уровень персонажа.
Молния: Поток молний, наносящий по 10 единиц урона за уровень магии.
Огненный шар, Превращение, Создать воду, Исцеляющее прикосновение, Молния,
```

Если проанализировать вывод, то можно увидеть, что пары ключ/значение в spellbook отображаются не в том порядке, в котором они были добавлены (помещены).

HashMap также предлагает дополнительные конструкторы/параметры, что может быть полезно в определенных случаях:

```
HashMap<>(initialSize, customLoadFactor)
```

Дополнительные параметры конструктора описаны ниже:

- `initialSize`

  Это целое число, представляющее количество элементов, которые LinkedHashMap будет содержать вначале;

- `customLoadFactor`

  Это плавающее число, указывающее, какой определенный процент объема от начального размера должен вызвать увеличение использования памяти LinkedHashMap. По умолчанию это 75%, указывается как .75f.

Эти дополнительные параметры можно использовать для тонкой настройки использования памяти и, в конечном счете, производительности HashMap. Операции изменения размера могут быть затратными, поэтому, если максимальный размер словаря известен заранее, его указание может дать выигрыш в производительности.

> **Примечание.** Хотя HashMap может быть создан только с параметром начального размера initialSize, он не может быть создан только с пользовательским коэффициентом загрузки customLoadFactor. Начальный размер всегда должен присутствовать при указании пользовательского коэффициента загрузки.

### *Hashtable*

Подобно взаимосвязи Vector и ArrayList, реализация Hashtable является потокобезопасной альтернативой HashMap. Если нам требуется функциональность HashMap для поддержки взаимодействия с несколькими потоками, то Hashtable — это то, что нужно:

```
Map<String,String> spellbook = new Hashtable<>();
```

## *LinkedHashMap*

Далее мы изменим тип spellbook на LinkedHashMap. LinkedHashMap похож на LinkedList тем, что каждая запись имеет указатели на следующий и предыдущий элементы:

```
Map<String,String> spellbook = new LinkedHashMap<>();
```

Выполнение нашего кода покажет, что ключи словаря расположены в следующем порядке:

```
Огненный шар, Исцеляющее прикосновение, Молния, Создать воду, Превращение,
```

Как видно, теперь они расположены в том порядке, в котором были добавлены.

Подобно HashMap, LinkedHashMap также предлагает дополнительные конструкторы/параметры, что может быть полезно для определенных требований. LinkedHashMap также имеет еще один параметр, позволяющий запускать переупорядочивание при доступе.

```
LinkedHashMap<>(initialSize, customLoadFactor, accessOrderEnabled)
```

Эти дополнительные параметры конструктора объясняются следующим образом:

- `initialsize`

  Это целое число, представляющее количество элементов, которые LinkedHashMap будет содержать вначале;

- `customLoadFactor`

  Это плавающее число, указывающее, какой определенный процент объема от начального размера должен вызвать увеличение использования памяти LinkedHashMap. По умолчанию это 75%, указывается как `.75f`;

- `accessOrderEnabled`

  Это булево значение, указывающее, должен ли доступ к элементу (get) вызывать операцию переупорядочивания. В результате пара ключ/значение, к которой обращались в последний раз, окажется внизу списка.

Это можно продемонстрировать по-другому, переопределив наш экземпляр `LinkedHashMap`:

```
Map<String,String> spellbook = new LinkedHashMap<>(5,.8f,true);
```

Теперь закомментируйте вызов `printMap()` и выведите значение ключа `Молния`:

```
//printMap(spellbook);
System.out.print(spellbook.get("Молния"));
printKeys(spellbook);
```

Запуск этого кода показывает новый порядок ключей:

```
Огненный шар, Исцеляющее прикосновение, Создать воду, Превращение, Молния,
```

Как видно, ключ для `Молния` теперь отображается в конце, так как к нему обращались последний раз.

### *ConcurrentModificationException*

В предыдущем примере мы закомментировали вызов метода `printMap()`. Если не закомментировать эту строку, наш код завершится с ошибкой `ConcurrentModificationException`.

Помните, что `for-each` в `printMap()` сначала получает ключи, а затем выполняет `get()` для каждого ключа. Передавая значение `true` для включения порядка доступа, вызов `get()` эффективно изменяет словарь, постоянно помещая наиболее часто используемую пару ключ/значение в конец словаря. Java распознает такую ситуацию и выбрасывает исключение `ConcurrentModificationException`, чтобы предотвратить бесконечный цикл.

## *TreeMap*

Теперь изменим нашу реализацию `spellbook`, чтобы использовать `TreeMap`:

```
Map<String,String> spellbook = new TreeMap<>();
```

Запуск кода теперь показывает такой список ключей в конце:

```
Исцеляющее прикосновение, Молния, Огненный шар,  Превращение, Создать воду,
```

Как видно, словарь отсортирован по ключам в буквенно-цифровом порядке. Это происходит потому, что `TreeMap` за кулисами реализует (*JavaTPoint 2021*) двоичный поиск (красно-черное дерево) для ускорения работы.

> **Примечание.** В отличие от предыдущих реализаций словаря, `TreeMap` не позволяет добавить пару ключ/значение с ключом `null`.

## Упорядоченные коллекции

Новой возможностью Java 21 являются (*Parlog N. 2023*) *упорядоченные коллекции*. По сути, в них раскрывается то, что известно как порядок встреч. Списки теперь также наследуются от нового класса, известного как `SequencedCollection`, предоставляющего множество новых методов для упрощения работы с первой и последней записью списка. Это позволяет создавать новые модели поведения, аналогичные тем, что возможны в связных списках:

- `getFirst()`. Возвращает первый элемент в коллекции;
- `getLast()`. Возвращает последний элемент в коллекции;
- `addFirst(element)`. Добавляет новый элемент в начало коллекции;
- `addLast(element)`. Добавляет новый элемент в конец коллекции;
- `reversed()`. Возвращает представление списка в обратном порядке.

> **Примечание.** Методы `addFirst()` и `addLast()` будут выбрасывать исключения, если коллекция не модифицируема или если добавление нарушит текущий порядок отсортированной коллекции.

Хотя множества не получили такого же уровня улучшений, у них появился доступ к методу `reversed()`. Вызов метода `reversed()` для множества вернет объект `SequencedSet`, содержащий представление множества в обратном порядке.

> **Примечание.** Это представление для множеств и списков отражает состояние базовой коллекции, и любые изменения в ней также будут учтены.

Словари также выиграли от наследования нового класса в Java 21, и этот класс называется `SequencedMap`. Этот класс расширяет следующие методы для всех реализаций словарей:

- `putFirst()`. Добавляет новую пару ключ/значение в начало словаря;
- `putLast()`. Добавляет новую пару ключ/значение в конец словаря;
- `firstEntry()`. Возвращает первую пару ключ/значение в словаре;
- `lastEntry()`. Возвращает последнюю пару ключ/значение в словаре;
- `pollFirstEntry()`. Удаляет и возвращает пару ключ/значение из начала словаря;
- `pollLastEntry()`. Удаляет и возвращает пару ключ/значение из конца словаря;
- `sequencedKeySet()`. Позволяет получить упорядоченное множество ключей `sequencedSet`;
- `sequencedValues()`. Возвращает упорядоченный список значений карты с помощью интерфейса `SequencedMap`;
- `sequencedEntrySet()`. Возвращает упорядоченное множество `sequencedSet` элементов карты.

Эти методы очень полезны для разработчиков, поскольку они облегчают процесс обучения работе с коллекциями в Java.

# Записи

Концепция записей была введена в Java 14. По сути, *записи* — это неизменяемые объекты Plain Old Java Objects (POJO). Одно из критических замечаний в адрес POJO заключается в том, что при их создании возникает много шаблонного кода, поскольку всем POJO необходимы такие вещи, как конструкторы, закрытые свойства и пары геттеров/сеттеров.

> **Примечание.** В этом смысле термин "шаблонный код" относится к стандартному коду, который вводится в каждый объект, и добавление которого считается утомительным.

Мы создадим запись для отслеживания объектов комнат, которые наши герои будут исследовать. Запись `Room`, безусловно, можно назвать неизменяемой (immutable), поскольку обычно комнаты в приключенческих играх определяются заранее и не меняются.

Начнем с определения записи `Room`:

```
record Room(String name, String description, List<String> exits) {
}
```

Внутри скобок мы можем определить любой дополнительный код, который нам нужен для завершения записи `Room`. Одним из общих требований для приключенческих игр является знание возможных выходов из комнаты. Поэтому мы создадим публичный метод `getExits()`. Также мы реализуем проверку `if-else` для форматирования вывода в зависимости от того, сколько выходов присутствует в списке:

```
public String getExits() {
    StringBuilder exitDesc = new StringBuilder();

    if (exits.isEmpty()) {
        exitDesc.append("Здесь нет очевидных выходов.");
    } else if (exits.size() == 1) {
        exitDesc.append("Здесь есть выход на ");
        exitDesc.append(exits.get(0));
    } else if (exits.size() == 2) {
        exitDesc.append(exits.get(0));
        exitDesc.append(" и ");
        exitDesc.append(exits.get(1));
    } else {
        exitDesc.append("Здесь есть выходы на ");
        boolean first = true;

        for (String exit : exits) {
            if (!first) {
                exitDesc.append(", ");
```

```
            } else {
                first = false;
            }
            exitDesc.append(exit);
        }
    }

    exitDesc.append(".");
    return exitDesc.toString();
}
```

Идея заключается в том, что сначала проверяется, есть ли в списке `exits` (это `ArrayList`) записи, для чего вызывается метод `isEmpty()` и выдается соответствующий ответ. Аналогично, существуют соответствующие способы указать ответ, если есть только один или два выхода, так что мы рассмотрим и это. Наконец, переберем все выходы и поставим перед ними запятые (за исключением первого выхода).

Определив запись `Room`, создадим список `cabinExits` (это `ArrayList`) и создадим экземпляр `Room` с именем `lakeCabin`:

```
List<String> cabinExits = new ArrayList();
cabinExits.add("Юг");
cabinExits.add("Запад");
Room lakeCabin = new Room("Хижина у озера",
"Вы стоите снаружи хижины"
+" у озера, на юге и востоке видна вода. На юге"
+" располагается красная пристань.", cabinExits);
```

Наконец, выведем описание `lakeCabin` и вызовем метод `getExits()`:

```
System.out.println(lakeCabin.Description());
System.out.println(lakeCabin.getExits());
```

Выполнение нашего кода должно дать результат, похожий на этот:

```
Вы стоите снаружи хижины у озера, на юге и востоке видна вода. На юге
располагается красная пристань.
Нет выходов на юг и запад.
```

Как видно, записи — это быстрый способ определения небольших неизменяемых POJO без необходимости использования множества стандартных шаблонных методов.

## Построение простого примера

Допустим, мы хотим создать небольшую симуляцию ролевой игры (RPG). Для работы с этим примером нам потребуется создать три класса:

- `RPGSimulation`. Наш основной класс;

- `Player`. Базовый класс игрока, который содержит большинство общих данных для объектов игрока;
- `Hero`. Класс нашего героя, который будет наследоваться от класса Player.

### Класс *RPGSimulation*

Давайте начнем с создания нового статического класса в пакете `chapter4`, с методом `main()`, и назовем его `RPGSimulation`:

```
package chapter4;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Random;
import java.util.TreeMap;

public class RPGSimulation {

    record Monster (String name, int attack, int maxDamage,
    int defense) {
        static Random random = new Random();
        static int hitPoints = 2;
        static boolean alive = true;

        public int rollAttack() {
            return random.nextInt(attack) + 1;
        }

        public int rollDamage() {
            return random.nextInt(maxDamage) + 1;
        }

        public boolean isAlive() {
            return alive;
        }
    }

    public static void main(String[] args) {
        Random randomNumber = new Random();
        int monsterCount = randomNumber.nextInt(4) + 1;
```

После определения нашего класса мы создадим запись `Monster` вне метода `main()`. Это позволит вызывать запись о монстре из других закрытых методов.

В записи `Monster` будут определены свойства для имени, атаки, максимального урона и защиты (`name`, `attack`, `maxDamage`, `defense`). Также по умолчанию будет установлено значение `hitPoints`, равное 2, а булевой переменной `alive` ("живой") по умол-

чанию будет присвоено значение `true`. После этого создадим два public-метода `rollAttack` и `rollDamage`, которые будут выдавать случайные значения на основе значений, содержащихся в `attack` и `maxDamage`. Завершим методом с именем `isAlive` для отображения булевого свойства `alive`.

Внутри метода `main()` мы создадим генератор случайных чисел с именем `randomNumber` и начнем с генерации случайного количества монстров, которые будут присутствовать в RPG-игре. Сгенерированное значение будет случайным числом от 1 до 4 и будет храниться в целочисленной переменной `monsterCount`.

Далее создадим новый список `ArrayList` с именем `monsters`, в котором будут храниться значения для нескольких записей типа `Monster`:

```
List<Monster> monsters = new ArrayList<>();

for (int monsterIdx = 0; monsterIdx < monsterCount; monsterIdx++) {
    int typeIdx = randomNumber.nextInt(4);

    switch (typeIdx) {
    case 0:
        monsters.add(new Monster("Kobald", 2, 8, 1));
        break;
    case 1:
        monsters.add(new Monster("Skeleton", 2, 8, 2));
        break;
    case 2:
        monsters.add(new Monster("Zombie", 1, 6, 2));
        break;
    default:
        monsters.add(new Monster("Rats", 1, 4, 1));
    }
}
```

В приведенном выше коде монстры перебираются с помощью цикла `for-each`. Внутри цикла случайным образом выбирается число от 0 до 3 для типа генерируемого монстра. Затем с помощью `switch-case` вызывается конструктор `Monster` с соответствующими значениями и добавляется имя монстра в список `monsters`.

Далее воспользуемся повторно кодом из класса `WorkingWithCollections` для `spellbook`, поскольку у нас есть один герой, который будет ее использовать:

```
Map<String,String> spellbook = new TreeMap<>();
spellbook.put("Огненный шар",
    "Огненный шар, наносящий по 8 единиц урона за уровень магии.");
spellbook.put("Исцеляющее прикосновение",
    "Прикосновение к раненому игроку восстанавливает
    по 5 очков за  уровень персонажа.");
spellbook.put("Молния",
    "Поток молний, наносящий по 10 единиц урона за уровень магии.");
```

```
spellbook.put("Создать воду",
    "Создает по 10 литров воды за уровень магии.");
spellbook.put("Превращение",
    "Превращает обычные предметы в золото.");
```

### Класс *Player*

Далее мы создадим два новых класса в пакете `chapter4` (без метода `main`), назвав их `Player` и `Hero`.

Начнем с класса `Player`:

```
package chapter4;

import java.util.Random;

public class Player {
    private Random random = new Random();
    private String name;
    private int attack;
    private int maxDamage;
    private int defense;
    private int hitPoints;
    private boolean alive;
}
```

Как видно, мы начинаем с определения наших закрытых (private) свойств. Затем создадим конструктор:

```
public Player(String name, int attack, int maxDamage, int defense) {
    this.name = name;
    this.attack = attack;
    this.maxDamage = maxDamage;
    this.defense = defense;
    this.alive = true;
}
```

В конструкторе `Player` мы сохраним переданные значения свойств `name`, `attack`, `maxDamage` и `defense`. Значение параметра `alive` будет равно `true`.

Далее нам понадобятся общедоступные (public) методы для генерации значений на основе значений `attack` и `maxDamage`:

```
public int rollAttack() {
    return random.nextInt(attack) + 1;
}

public int rollDamage() {
    return random.nextInt(maxDamage) + 1;
}
```

Далее создадим геттеры и сеттеры. Помните, в начале главы мы обсуждали записи Java и упоминали об утомительном шаблонном коде? Вот что под этим подразумевалось:

```
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public int getAttack() {
    return attack;
}

public void setAttack(int attack) {
    this.attack = attack;
}

public int getMaxDamage() {
    return maxDamage;
}

public void setMaxDamage(int maxDamage) {
    this.maxDamage = maxDamage;
}

public int getDefense() {
    return defense;
}

public void setDefense(int defense) {
    this.defense = defense;
}

public int getHitPoints() {
    return this.hitPoints;
}

protected void setHitPoints(int hitPoints) {
    this.hitPoints = hitPoints;
}

public boolean isAlive() {
    return this.alive;
}
```

К счастью, в большинстве IDE предусмотрена быстрая генерация геттеров и сеттеров.

**Примечание.** IDE Eclipse может автоматически генерировать геттеры и сеттеры. Просто щелкните правой кнопкой мыши в середине файла класса, выберите Source и затем Generate Getters and Setters.

Наконец, нам понадобится public-метод, который будет уменьшать hitPoints на определенное количество единиц урона:

```
public void decrementHitPoints(int damage) {
    this.hitPoints = this.hitPoints - damage;
}
```

## Класс *Hero*

Далее создадим класс Hero. Класс Hero будет расширять (extend) класс Player, чтобы наследоваться от него как от базового класса:

```
package chapter4;
import java.util.Map;

public class Hero extends Player {
    private Map<String,String> spellbook;
```

Мы также определим spellbook как private-свойство. В классе Hero определим два конструктора:

```
    public Hero(String name, int attack, int maxDamage, int defense) {
        super(name, attack, maxDamage, defense);
        super.setHitPoints(5);
    }
```

Первый конструктор принимает параметры для name, attack, maxDamage и defense. Затем он использует ключевое слово super, чтобы передать эти параметры базовому классу, в данном случае Player. Он также инициализирует свойство hitPoints значением 5 для всех героев с использованием соответствующего метода-сеттера.

Второй конструктор аналогичен, за исключением того, что он принимает параметр для spellbook:

```
    public Hero(String name, int attack, int maxDamage, int defense,
        Map<String,String> spellbook) {
        this(name, attack, maxDamage, defense);
        this.spellbook = spellbook;
    }
```

Как было показано ранее, второй конструктор использует ключевое слово this для вызова первого конструктора с первыми четырьмя переданными параметрами. Затем он инициализирует свойство spellbook с помощью словаря spellbook, переданного в качестве последнего параметра.

Поскольку все свойства наследуются от класса `Player`, нам нужно создать геттеры и сеттеры только для свойства `spellbook`:

```
    public Map<String, String> getSpellbook() {
        return spellbook;
    }

    public void setSpellbook(Map<String, String> spellbook) {
        this.spellbook = spellbook;
    }
}
```

## Продолжение работы с классом *RPGSimulation*

После создания классов `Player` и `Hero` можно вернуться к классу `RPGSimulation`. В конце метода `main()` (после кода `spellbook`) мы определим наших героев:

```
Hero byorki = new Hero("Byorki", 8, 5, 5);
Hero klar = new Hero("K'lar", 10, 12, 3);
Hero tyrenni = new Hero("Tyrenni", 6, 2, 6, spellbook);
```

Теперь, когда наши три героя инстанцированы как объекты, добавим их в новый список `ArrayList` с именем `heroes`:

```
List<Hero> heroes = new ArrayList<>();
heroes.add(byorki);
heroes.add(klar);
heroes.add(tyrenni);
```

Теперь добавим новый метод в конец класса `RPGSimulation`. Это будет статический метод `generatePlayerOrder`, возвращающий `ArrayList` со значениями типа `String` и принимающий в качестве параметров список `Hero` и список `Monster`. Назначение этого метода заключается в том, чтобы взять имена наших героев и монстров и случайным образом объединить их в один список.

Для этого мы определим `playerCount` как общее количество элементов в обоих списках. Список `returnValue` должен быть того же размера, что и `playerCount`, после того как все игроки будут назначены в основной список. По сути, мы будем случайным образом решать, что выбрать из списка `heroList` или `monsterList`, а затем случайным образом выбирать допустимый индекс из этого списка:

```
private static List<Object> generatePlayerOrder(List<Hero> heroList,
List<Monster> monsterList) {
    List<Hero> tempHeroList = new ArrayList<Hero>(List.copyOf(heroList));
    List<Monster> tempMonsterList =
        new ArrayList<Monster>(List.copyOf(monsterList));
    List<Object> returnValue = new ArrayList<>();
    Random random = new Random();
    int playerCount = heroList.size() + monsterList.size();
```

```
    while (returnValue.size() < playerCount) {
        if (random.nextBoolean()) {
            if (!tempHeroList.isEmpty()) {
                int heroIndex = random.nextInt(tempHeroList.size());
                returnValue.add(tempHeroList.get(heroIndex));
                tempHeroList.remove(heroIndex);
            }
        } else {
            if (!tempMonsterList.isEmpty()) {
                int monsterIndex = random.nextInt(tempMonsterList.size());
                returnValue.add(tempMonsterList.get(monsterIndex));
                tempMonsterList.remove(monsterIndex);
            }
        }
    }
    return returnValue;
}
```

Первым делом мы создадим два новых списка `ArrayList` с именами `tempHeroList` и `tempMonsterList`. Это позволит удалять объекты, чтобы отслеживать, кто был выбран, не затрагивая основные списки `heroList` и `monsterList`. Затем мы определяем несколько переменных, которые помогут нам на этом пути, включая `ArrayList` с именем `returnValue`, представляющий собой список (`List`) элементов типа `Object`, так что мы можем добавлять в него объекты любого класса.

Затем мы создаем цикл `while`, который будет выполняться до тех пор, пока размер нашего списка `returnValue` будет меньше общего количества элементов, присутствующих в `heroList` и `monsterList`, вместе взятых (`playerCount`). Внутри цикла `while` мы переключаемся между извлечением случайных элементов (используя два случайно сгенерированных значения) из `tempHeroList` и `tempMonsterList`. Таким образом, мы получаем список всех игроков в игре (героев и монстров), расположенных в как можно более случайном порядке.

> **Примечание.** Метод `nextBoolean()` класса `Random` делает именно то, о чем говорит его название, — случайным образом возвращает либо истинное, либо ложное значение.

Сформировав список `returnValue`, мы вернем его вызывающему методу. Вернувшись в метод `main()`, создадим новый список (`List`) с элементами типа `Object` и установим его равным значению, возвращаемому методом `generatePlayerOrder()`:

```
List<Object> playerOrder = generatePlayerOrder(heroes, monsters);
```

Затем выполним итерации через `playerOrder`, чтобы смоделировать раунд действий в приключенческой ролевой игре. Первое, что нужно сделать, — это проверить, является ли текущий объект игрока героем или монстром. Это можно сделать с помощью ключевого слова `instanceof`. Также нужно убедиться, что наш герой жив,

прежде чем позволить ему действовать, и это можно сделать путем вызова метода `isAlive()`, унаследованного от класса `Player`:

```
for (Object player : playerOrder) {
    System.out.println();

    if (player instanceof Hero hero) {
        if (hero.isAlive()) {
            String name = ((Hero) player).getName();
            int monsterIndex = randomNumber.nextInt(monsters.size());
            Monster targetMonster = monsters.get(monsterIndex);

            System.out.println(name
                + " атакует " + targetMonster.name());

            int attack = hero.rollAttack();
            if (attack >= targetMonster.defense) {
                int damage = hero.rollDamage();
                System.out.println(name
                + " атакует на " + attack + " и наносит "

                + targetMonster.name() + " "
                + damage + " единиц(ы) урона");
                targetMonster.decrementHitPoints(damage);

                if (!targetMonster.isAlive()) {
                    System.out.println(targetMonster.name()
                    + " погиб!");
                }
            } else {
                System.out.println(name + " атакует на "
                + attack + " и не попадает в "
                + targetMonster.name());
            }
        }
```

Внутри вложенных проверок `if` мы подготовимся к основной логике действия, определив переменные для имени героя, случайного индекса монстра и объекта самого монстра (`monster`). Действие начинается с сохранения результата выполнения героем атаки (`attack`). Если значение `attack` больше или равно защите (`defence`) монстра-цели (`targetMonster`), то засчитывается попадание, и в консоль выводятся соответствующие сообщения. Также будет выведено соответствующее сообщение, если метод `isAlive()` монстра-цели вернет значение `false` после выполнения действия.

Следующей за логикой действий героя идет проверка `else` (для корневого `if (player instanceof Hero)` в предыдущем коде), если текущий игрок имеет тип

Monster. Логика аналогична той, что мы проделали с объектами Hero, за исключением того, что она выполняет вызовы свойств и методов записи Monster и выбирает случайный индекс героя hero:

```
} else if (player instanceof Monster monster) {
    if (monster.isAlive()) {
        String name = monster.name();
        int heroIndex = randomNumber.nextInt(heroes.size());
        Hero targetHero = heroes.get(heroIndex);
        String heroName = targetHero.getName();

        System.out.println(monster.name()
        + " атакует " + heroName);

        int attack = monster.rollAttack();

        if (attack >= targetHero.getDefense()) {
            int damage = monster.rollDamage();
            System.out.println(name
            + " атакует на " + attack + " и наносит " + heroName
            + " " + damage + " единиц(ы) урона.");

        targetHero.decrementHitPoints(damage);

        if (!targetHero.isAlive()) {
            System.out.println(heroName + " погиб!");
            }
            } else {
            System.out.println(name
            + " атакует на " + attack
            + " и не попадает в " + heroName);
        }
    }
}
```

Выполнение этого кода создает список героев, формирует случайный список монстров и выполняет один раунд их действий в ролевой игре. Учитывая случайность компонентов, результат будет варьироваться, но выглядеть он должен примерно так:

```
Rats атакует Tyrenni
Rats атакует на 5 и не попадает в Tyrenni

Byorki атакует Skeleton
Byorki атакует на 7 и наносит Skeleton 4 единиц(ы) урона.
Skeleton погиб!
```

```
Tyrenni атакует Kobald
Tyrenni атакует на 3 и наносит Kobald 2 единиц(ы) урона.
Kobald погиб!

K'lar атакует Kobald
K'lar атакует на 6 и наносит Kobald 10 единиц(ы) урона.
Kobald погиб!
```

В этом разделе мы использовали многое из рассмотренного ранее в этой главе. Мы также узнали несколько новых приемов и ключевых слов:

- вызов конструктора из другого конструктора того же класса с помощью `this()`;
- построение списков базовых классов, которые могут содержать различные типы объектов;
- проверка типа объекта с помощью `instanceof`;
- генерация случайного булева числа с помощью метода `nextBoolean()` класса `Random`;
- сравнение определения и реализации записей и POJO.

# Заключение

В этой главе мы подробно рассмотрели массивы, коллекции и записи. А также подробно рассмотрели различные типы реализаций коллекций, их использование, преимущества и недостатки. Кроме того, мы обсудили упорядоченные коллекции и некоторые из новых методов и возможностей, предоставляемых Java 21. Наконец, мы использовали полученные знания, написав быструю симуляцию RPG, чтобы попрактиковаться в использовании коллекций и других понятий.

В следующей главе обсудим, как Java обрабатывает арифметические операции. Мы рассмотрим операции с целыми числами (которые использовались в этой главе) и операции над типами с плавающей точкой.

# Важно помнить

- Массивы — это простые структуры, которые позволяют хранить несколько значений в одной переменной.
- Массивы индексируются целыми числами, и их максимальный размер должен быть определен при инициализации.
- Значения массивов могут индексироваться по нескольким измерениям. В наших примерах мы продемонстрировали двумерные массивы, но, конечно, возможны и другие варианты.

- Множества содержат только уникальные элементы, в то время как списки допускают дубликаты.
- `HashSet` сохраняет порядок добавляемых элементов, в то время как `TreeSet` располагает элементы в алфавитно-цифровом порядке.
- Метод `sort()` из класса `Collections` — хороший способ сортировки элементов в коллекции, если поддерживается или определен компаратор типа элемента.
- При работе с элементами `List` с помощью потоков используйте `Vector` в качестве реализации. `ArrayList` не поддерживает безопасность потоков.
- `LinkedList` следует использовать при большом количестве элементов и частом добавлении.
- Словарь хранит небольшие объемы данных в парах ключ/значение и не наследуется от базового класса `Collection`.
- `HashMap` не упорядочен, `LinkedHashMap` упорядочен по порядку добавления элементов, а `TreeMap` упорядочен по ключам.
- Последовательные коллекции позволяют разработчикам легко получить доступ к *порядку встреч* базовой коллекции, включая новые методы доступа к первому и последнему элементам.
- Записи — это отличный способ определить небольшие неизменяемые объекты без необходимости создавать отдельный POJO с большим количеством объемного шаблонного кода.

# Глава 5
# Арифметические операции

*Математика — это язык, на котором Бог написал Вселенную.*
*— Галилео Галилей*

## Введение

В этой главе мы разберемся, как выполнять арифметические операции в Java. Это позволит нам обсудить некоторые нюансы обработки математических данных компьютерами. Мы также рассмотрим, как создавать простые модульные тесты, поскольку детерминированная природа арифметики является хорошей основой для обучения основам модульного тестирования.

Хотя некоторым математика может показаться пугающей, мы, разработчики программного обеспечения, должны принять ее. Понимание того, как компьютеры выполняют арифметические операции, имеет центральное значение для изучения разработки программного обеспечения и, как полагал Галилей, Вселенной.

## Структура

В этой главе мы рассмотрим следующие темы.

- Целочисленная арифметика.
- Арифметика чисел с плавающей точкой.

## Цели

Основная цель этой главы — сформировать базовое понимание того, как использовать арифметические операции в коде Java. Мы также воспользуемся возможностью познакомиться с модульным тестированием. В связи с этим в данной главе поставлены следующие задачи:

- узнать, как выполняется целочисленная арифметика в Java;

- выяснить, как выполняется арифметика чисел с плавающей точкой в Java;
- понять, как компьютеры выполняют арифметические действия "под капотом";
- узнать немного о модульном тестировании и о том, как использовать его для обеспечения согласованного поведения методов.

# Целочисленная арифметика

Для начала рассмотрим простую арифметику с целыми числами. Создайте новый класс Java с именем `MathExamples` в новом пакете с именем `chapter5`. Убедитесь, что у этого класса есть метод `main()`:

```
package chapter5;

public class mathExamples {
    public static void main() {

    }
}
```

Внутри метода `main()` определим две целочисленные переменные:

```
int intNumA = 5;
int intNumB = 3;
```

## Сложение

Теперь давайте сложим их вместе и выведем результат:

```
System.out.println(intNumA + " + " + intNumB + " = "
    +intNumA + intNumB);
```

Если запустить этот код, то результат должен быть следующим:

```
5 + 3 = 53
```

Но это не то, чего мы ожидали. Ведь 5 плюс 3 равно 8, а не 53.

Помните, что числовые переменные в операторе `println` неявно преобразуются в строки. В конечном счете, нам это и нужно. Но сначала мы хотим, чтобы числа складывались. Для этого мы заключим `intNumA` и `intNumB` в дополнительное множество круглых скобок:

```
System.out.println(intNumA + " + " + intNumB + " = "
    + (intNumA + intNumB));
```

Теперь, когда запускаем код, порядок операций меняется, и скобки обрабатываются перед преобразованием в строку:

```
5 + 3 = 8
```

Давайте посмотрим, что происходит "под капотом".

И `intNumA`, и `intNumB` являются целыми числами. Поскольку целые числа являются примитивным типом, их значения хранятся в системе base-2 (также известной как двоичная), как показано здесь:

```
intNumA = 5 = 1 0 1
      place = 4 2 1

intNumB = 3 = 0 1 1
      place = 4 2 1
```

Десятичное число 5 в двоичной системе счисления[1] представлено как 101, потому что на месте четверки стоит 1, а на месте единицы — еще одна 1. Аналогично, десятичное число 3 превращается в 011, потому что в нем есть 1 на месте двойки и 1 на месте единицы. По сути, это означает, что мы складываем 101 и 011:

```
  101
 +011
 ----
 1000

answer = 1 0 0 0 = 8
 place = 8 4 2 1
```

Двоичное сложение выполняется следующим образом:

1. На месте единицы мы складываем 1 и 1, что в двоичном исчислении равно 10. Таким образом, записываем 0.
2. На место двойки переносим 1 и прибавляем 1, что в двоичном исчислении равно 10. Снова записываем 0.
3. На место четверки переносим 1 и прибавляем 1, что равно 10. И снова записываем 0.
4. На месте восьмерки переносим 1 и прибавляем 0. Теперь записываем 1, и у нас есть полный ответ.

1 на месте восьмерки и 0 на местах четверки, двойки и единицы дают нам десятичное значение 8 для ответа.

Теперь, поскольку мы планируем выполнить несколько таких операций, вместо этого создадим public-метод `add`:

```
public static int add(int intNum1, int intNum2) {
    return intNum1 + intNum2;
}
```

---

[1] Позиционная система счисления с основанием 2. В двоичной системе счисления числа записываются с помощью двух символов (0 и 1). Используется практически во всех современных компьютерах и прочих вычислительных электронных устройствах. — *Прим. ред.*

Теперь давайте изменим оператор println, приведенный выше:

```
System.out.println(intNumA + " + " + intNumB + " = "
    +add(intNumA,intNumB));
```

Выполнение этого кода должно привести к такому же результату, как и выше. Обратите внимание, что поскольку intNumA и intNumB вычисляются в методе add(), дополнительные скобки внутри метода println() не требуются.

## Тестирование *add()* с помощью JUnit

Создание методов для арифметических операций дает нам возможность естественным образом перейти к модульному тестированию. *Модульное тестирование (юнит-тестирование)* — это систематическая форма тестирования, предполагающая создание небольших тестов для отдельных частей приложения. Обычно это означает, что разработчики программного обеспечения создают тест для каждого метода класса.

Модульные тесты хорошо работают с детерминированными операциями. В разработке программного обеспечения детерминированная операция — это операция, которая (при наличии множества входных данных) будет возвращать один и тот же предсказуемый результат при каждом выполнении. Поскольку арифметические операции детерминированы по своей природе, они должны стать отличными примерами простых модульных тестов.

В нашем случае воспользуемся библиотекой тестирования JUnit, чтобы создать модульный тест для метода add().

Создайте новый **JUnit Test Case**. Если этот пункт недоступен в меню **File | New**, выберите **Other**. В следующем диалоговом окне прокрутите вниз до **JUnit** и выберите **JUnit Test Case**, как показано на рис. 5.1.

В следующем диалоговом окне нужно указать пакет, имя и тестируемый класс. Добавьте следующие записи (рис. 5.2):

- **Package:** chapter5
- **Name:** MathExamplesTests
- **Class under test:** MathExamples

Появится еще одно диалоговое окно, но на этом этапе можно нажать кнопку Finish. В результате будет создан файл класса MathExamplesTests.

> **Примечание.** Возможно, потребуется выполнить подтверждение, чтобы разрешить добавление **JUnit5** в путь сборки.

Вспомните *главу 2 "Фундаментальные структуры программирования"*, где мы создали наш файл Maven pom.xml. Необходимо изменить этот файл, чтобы включить в него одно новое свойство и одну новую зависимость.

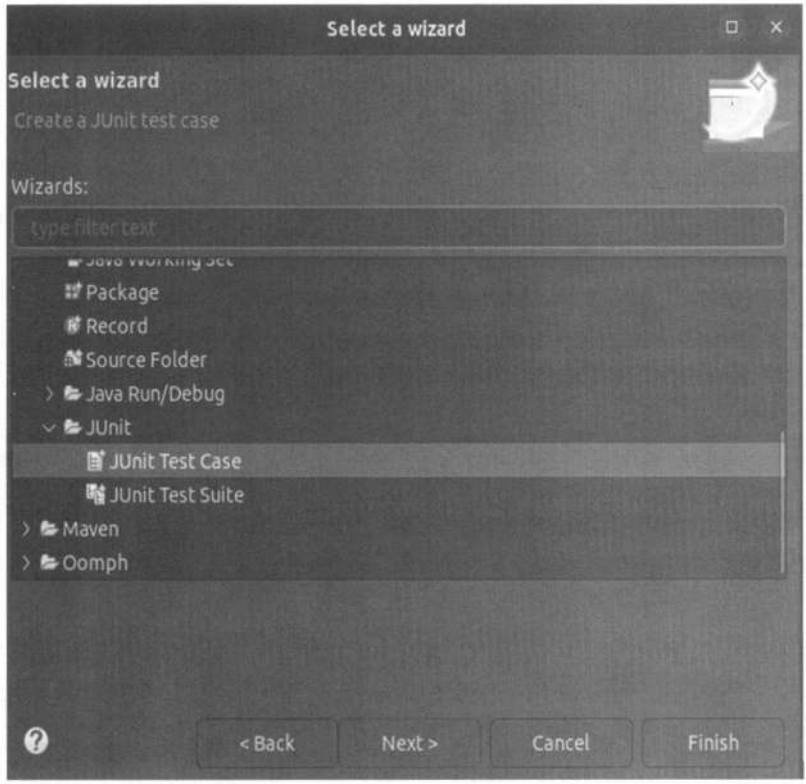



**Рис. 5.1.** Создание нового **JUnit Test Case** в Eclipse IDE

Откройте файл pom.xml и добавьте текущую версию для `junit.jupiter.version` в раздел свойств `properties`:

```
<properties>
    <java.version>21</java.version>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
    <junit.jupiter.version>5.9.3</junit.jupiter.version>
</properties>
```

Далее прокрутите в конец файла pom.xml и добавьте новую зависимость. Поскольку это первая зависимость, которую мы добавляем, необходимо создать новый раздел для зависимостей `dependencies`. Нам нужно добавить зависимость `JUnit`, указав следующие дополнительные свойства:

- **Group ID:** `org.junit.jupiter`
- **Artifact ID:** `junit-jupiter-api`
- **Version:** `${junit.jupiter.version}`
- **Scope:** `test`

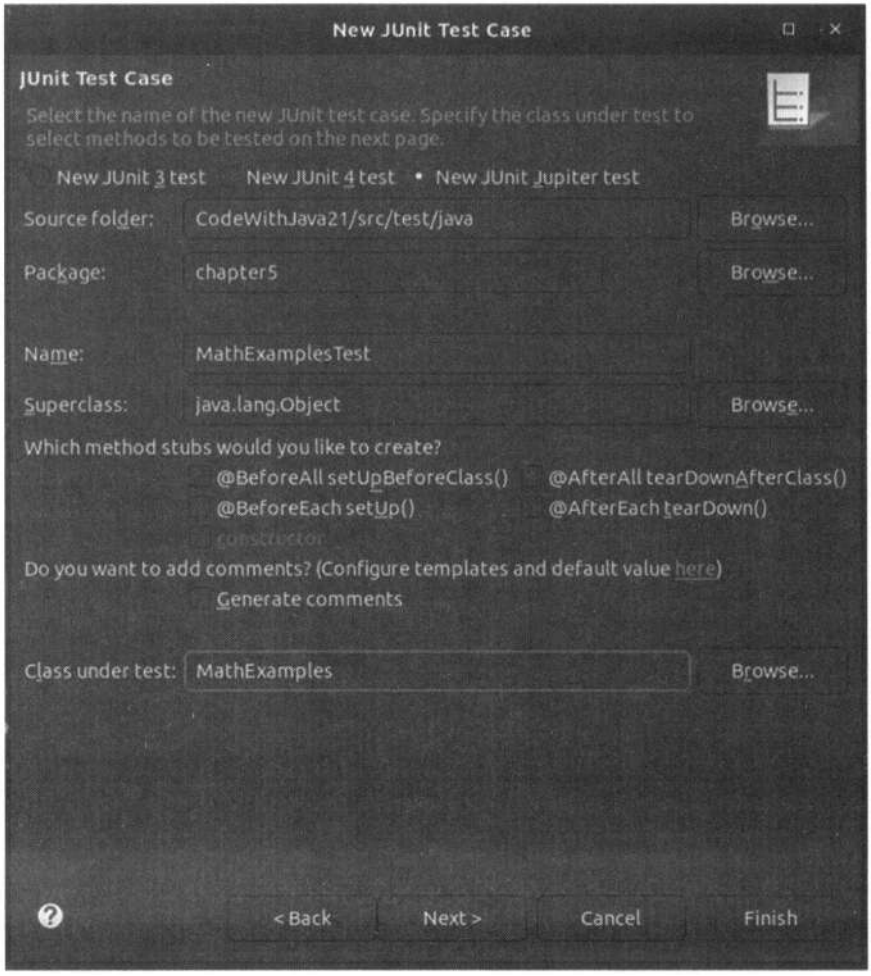



**Рис. 5.2.** Создание нового **JUnit Test Case** в Eclipse IDE

Код секции зависимостей в pom.xml должен выглядеть следующим образом:

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>${junit.jupiter.version}</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

После размещения этого кода можно вернуться к классу `MathExamplesTests`:

```
package chapter5;
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;
```

```
class MathExamplesTests {

}
```

Важными моментами предыдущего кода (помимо определения класса) являются импорты. У нас есть `import` для класса `org.junit.jupiter.api.Test` и `static import` для `org.junit.jupiter.api.Assertions.assertEquals`. Ключевое слово `static` позволяет нам включить метод `assertEquals()` из статического класса `Assertions`.

Давайте создадим новый метод с именем `testAdd` внутри класса:

```
@Test
void testAdd() {
    assertEquals(8,MathExamples.add(5,3));
}
```

Проще говоря, аннотация `@Test` идентифицирует этот метод как тест JUnit. Внутри метода `testAdd()` мы просто вызываем метод `assertEquals()`, чтобы указать, что значение `8` должно быть равно результату вызова нашего метода `add()` с параметрами `5` и `3`.

Чтобы проверить метод `testAdd()`, можно кликнуть правой кнопкой мыши на классе `MathExamplesTests` в проводнике проекта и выбрать **Run As**, а затем **JUnit Test**. Если все сделано правильно, наши JUnit-тесты должны сообщить об успешном выполнении (1/1), без каких-либо ошибок или сбоев.

Тесты JUnit выполняются в процессе Maven до того, как код будет запущен. Если тест завершится неудачно, наш код не будет запущен. Это становится полезным в больших производственных средах. Если в метод вносятся изменения, модульный тест может гарантировать, что метод будет выдавать те же результаты.

> **Примечание.** Иногда модульные тесты пишутся раньше, чем логика в тестируемом классе. Это называется Test Driven Development (TDD), и это помогает гарантировать, что методы написаны для достижения определенных, конкретных требований.

## Вычитание

Возвращаясь к нашему классу `MathExamples`, напишем метод для вычитания:

```
public static int subtract(int intNum1, int intNum2) {
    return intNum1 - intNum2;
}
```

В нашем классе `MathExamplesTests` можно написать короткий модульный тест для метода `subtract()` с именем `testSubtract`:

```
@Test
void testSubtract() {
    assertEquals(2,MathExamples.subtract(5,3));
}
```

Вернитесь в наш метод main() и добавьте новый оператор println:

```
System.out.println(intNumA + " - " + intNumB + " = "
    + subtract(intNumA,intNumB));
```

Теперь при выполнении кода должен получиться такой результат:

```
5 + 3 = 8
5 - 3 = 2
```

## Умножение

Хотя два предыдущих примера были довольно простыми, в Java используются разные символы для умножения и деления (как и в большинстве языков программирования). Начнем с нового метода умножения. Как показано ниже, вместо буквы x в Java для операции умножения используется звездочка (*):

```
public static int multiply(int intNum1, int intNum2) {
    return intNum1 * intNum2;
}
```

В нашем классе MathExamplesTests можно написать короткий модульный тест для метода multiply() под названием testMultiply:

```
@Test
void testMultiply() {
    assertEquals(15,MathExamples.multiply(5,3));
}
```

Вернувшись в метод main(), добавим оператор println для метода умножения:

```
System.out.println(intNumA + " x " + intNumB + " = "
    + multiply(intNumA,intNumB));
```

Выполнение кода должно привести к следующему результату:

```
5 + 3 = 8
5 - 3 = 2
5 x 3 = 15
```

## Деление

Деление также отличается, поскольку вместо символа (÷) в Java для операций деления используется прямая косая черта /. Это более логично, поскольку символ (÷) отсутствует на большинстве клавиатур. Создадим простой метод и для этого:

```
public static int divide(int intDividend, int intDivisor) {
    return intDividend / intDivisor;
}
```

В нашем классе MathExamplesTests можно написать короткий модульный тест для метода divide().

В методе `testDivide()` будем утверждать, что 8, деленное на 2, равно 4:

```
@Test
void testDivide() {
    assertEquals(4,MathExamples.divide(8,2));
}
```

Вернувшись в метод `main()`, добавим оператор `println` и для метода деления:

```
System.out.println(intNumA + " " + (char)247 + " " + intNumB + " = "
    +divide(intNumA,intNumB));
```

Запуск нашего кода должен привести к такому результату:

```
5 + 3 = 8
5 - 3 = 2
5 x 3 = 15
5 ÷ 3 = 1
```

Подождите, что случилось? Почему 5 ÷ 3 дает значение 1?

Многие читатели, вероятно, ожидали увидеть 1,6667 или 1⅔. Помните, что целочисленная арифметика работает только с целыми числами. Так как делитель 3 входит в делимое 5 только 1 раз, то правильным ответом здесь будет частное 1. Посмотрите на рис. 5.3.

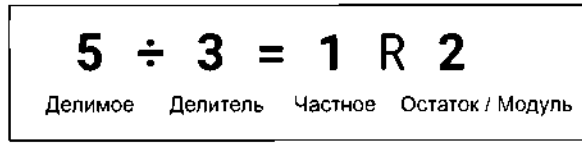



**Рис. 5.3.** Краткое напоминание о составляющих операции деления

## Модуль

Частым вопросом при делении целых чисел является: "Что случилось с остатком?". Java абсолютно точно вычисляла весь ответ: и частное, и остаток. Но при делении целых чисел оператор (/) возвращает только частное (часть ответа, состоящую из целых чисел). Остаток (или модуль), то есть часть ответа, которая "пропала", все еще есть, но мы его не видим. Чтобы выполнить целочисленное деление и вернуть только остаток, нам нужно выполнить операцию с модулем. Чтобы вернуть модуль в Java, нам нужно использовать оператор %.

В нашем классе `MathExamples` создадим метод, который будет возвращать модуль делимого и делителя:

```
public static int modulo(int intDividend, int intDivisor) {
    return intDividend % intDivisor;
}
```

В нашем классе MathExamplesTests можно также создать модульный тест для метода modulo():

```
@Test
void testModulo() {
    assertEquals(2,MathExamples.modulo(5,3));
}
```

Аналогичным образом добавим оператор println в наш метод main():

```
System.out.println(intNumA + " mod " + intNumB + " = "
    + modulo(intNumA,intNumB));
```

Выполнение кода должно привести к такому результату:

```
5 + 3 = 8
5 - 3 = 2
5 x 3 = 15
5 ÷ 3 = 1
5 mod 3 = 2
```

## Возведение в степень

Еще одна распространенная арифметическая операция, о которой необходимо знать разработчикам программного обеспечения, — это оценка числа по экспоненте. Иногда это называют возведением числа в степень. К сожалению, в Java нет собственного оператора для функций экспоненты, поэтому нам придется воспользоваться методом pow() из библиотеки Math. Начнем с написания метода exponent() в нашем классе MathExamples, чтобы взять основание base и возвести его в степень power:

```
public static int exponent(int base, int power) {
    return (int) Math.pow(base, power);
}
```

> **Примечание.** Метод Math.pow() принимает параметры типа double и возвращает значение типа double. Хотя метод неявно преобразует свои параметры из целых чисел в дробные double, нам необходимо привести возвращаемое значение к целому числу.

Можно создать модульный тест для метода exponent(), используя те же числа, о которых говорилось выше. Это позволит проверить операцию, когда 5 в степени 3 равно значению 125:

```
@Test
void testExponent() {
    assertEquals(125,MathExamples.exponent(5, 3));
}
```

Теперь можно добавить оператор println в конец метода main() в классе MathExamples:

```
System.out.println(intNumA + " в степени " + intNumB + " = "
    + exponent(intNumA,intNumB));
```

Выполнение кода приведет к следующему результату:

```
5 + 3 = 8
5 - 3 = 2
5 x 3 = 15
5 ÷ 3 = 1
5 mod 3 = 2
5 в степени 3 = 125
```

# Арифметика чисел с плавающей точкой

Теперь, когда мы хорошо разобрались в целочисленной арифметике, давайте перейдем к арифметике чисел с плавающей точкой. Числа с плавающей точкой (например, float[2] и double[3]) отличаются тем, что в них необходимо учитывать основное число и его десятичные значения. Кроме того, при работе с ними существует фиксированное количество точек точности, поэтому они также известны как типы с *фиксированной точностью*.

Прежде чем мы продолжим, давайте определим два термина: точность и прецизионность. Точность — это мера погрешности, а прецизионность — степень различия. При работе с арифметическими операциями нам нужны точность и прецизионность. Однако ограничения двоичной системы счисления (и, в конечном счете, базового компьютерного оборудования) в некоторых обстоятельствах вынуждают нас отказываться от одного в пользу другого.

Мы обнаружим, что это приводит к некоторым нюансам при округлении чисел, которые не очень хорошо укладываются в дроби чисел base-2 (двоичных). Это приводит к оценкам в некоторых точках точности, поэтому многие разработчики описывают арифметику чисел с плавающей точкой как (*Darcy 2021*) приближение к реальной арифметике.

Давайте начнем с определения двух новых переменных типа `double` в нашем классе `MathExamples`:

```
double dblNumC = 5.2d;
double dblNumD = 3.1d;
```

## Сложение

Теперь мы перегрузим наш метод `add()`, чтобы он принимал и возвращал числа типа `double`:

```
public static double add(double dblNum1, double dblNum2) {
    return dblNum1 + dblNum2;
}
```

[2] 32-битное число с плавающей запятой. — *Прим. ред.*

[3] 64-битное число с плавающей запятой. — *Прим. ред.*

**Примечание.** Хорошей практикой написания кода считается размещение перегруженных методов рядом друг с другом внутри класса. Так другим разработчикам будет проще понять, что происходит, и они смогут ясно увидеть, что метод был перегружен.

Давайте добавим оператор `println` в конец нашего метода `main()`:

```
System.out.println(dblNumC + " + " + dblNumD + " = "
    +add(dblNumC,dblNumD));
```

Выполнение кода должно дать такие результаты в конце предыдущего вывода:

```
5.2 + 3.1 = 8.3
```

## Особенности работы с арифметикой чисел с плавающей точкой

Теперь давайте попробуем сделать что-то немного другое. Определим еще две переменные типа `double` со значениями `0.1` и `0.2` и добавим оператор `println`, чтобы показать результат работы метода `add()`:

```
double dblNumE = 0.1d;
double dblNumF = 0.2d;

System.out.println(dblNumE + " + " + dblNumF + " = "
    + add(dblNumE,dblNumF));
```

Теперь давайте запустим код:

```
3.1 + 5.2 = 8.3
0.1 + 0.2 = 0.30000000000000004
```

Такой результат, вероятно, не ожидался.

Помните, мы упоминали, что арифметика с плавающей точкой — это приближение. Проблема в том, что десятичное значение `0.1` (также представляемое как дробь 1/10) является повторяющейся дробью при выражении в двоичном виде. Подобно тому, как десятичная дробь ⅓ представляет собой не завершающееся представление[4] (0,3333), 1/10 может быть кратко и точно выражена как `0.1`. С другой стороны, двоичное представление `0.1` является незавершающимся представлением и выглядит следующим образом:

```
Decimal: 0.1 = Binary: 0.0001100110011001100110011001100110011
```

Это явление происходит благодаря *факторизации целых чисел* каждой системы счисления. Простые числа — это числа, которые могут делиться только на единицу и на себя. Таким образом, простые факторы числа[5] — это простые числа, на которые оно может быть равномерно разделено.

---

[4] Не завершающееся десятичное представление (дробь) означает, что данное число будет содержать бесконечное количество цифр справа от десятичной точки. Существует два типа: повторяющиеся или не повторяющиеся. — *Прим. ред.*

[5] См. Википедия: **https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F**. — *Прим. перев.*

В десятичной системе (*Wiffin 2017*) простые факторы для 10 — 2 и 5. Поэтому такие дроби, как ½, ¼, ⅕, ⅛ и 1/10, могут быть представлены точно. Это означает, что оставшиеся дроби ⅓, ⅙, 1/7 и 1/9 являются незавершающимися, так как в их знаменателях (3, 6, 7, 9) в качестве наибольшего простого фактора используется 3 или 7.

Однако в двоичной системе единственным простым фактором 2 является 2. Поэтому дроби ½, ¼ и ⅛ могут быть точно представлены. Но другие распространенные дроби, такие как 0,1 (1/10) или 0,2 (⅕), являются незавершающимися.

К сожалению, это означает, что преобразование между десятичной и двоичной системами счисления влечет за собой различные степени приближения. Как разработчики программного обеспечения, мы должны быть уверены, что учитываем это.

Представление десятичной дроби `0.1` в формате `float` в Java выглядит следующим образом:

```
0.1f = 0.100000001490116119384765625
```

Однако, поскольку мы используем число типа `double`, в нашем примере представление `0.1` десятичной дроби в Java выглядит следующим образом:

```
0.1d = 0.1000000000000000055511151231257827021181583404541015625
```

Это указывает на еще один нюанс примитивных типов, который заключается в том, что `0.1f` и `0.1d` технически не равны. На самом деле, мы можем проверить это, добавив эту строку кода в конец нашего метода `main()`:

```
System.out.println((0.1f == 0.1d));
```

Запуск кода должен дать следующий результат:

```
3.1 + 5.2 = 8.3
0.1 + 0.2 = 0.30000000000000004
false
```

Как показано, использование оператора `println` для отображения того, равны ли `0.1f` и `0.1d`, дает булевый ответ `false`. Мы также можем увидеть эту разницу, если вернемся к нашему определению `dblNumE` и `dblNumF` и настроим их, как показано здесь:

```
double dblNumE = 0.1f;
double dblNumF = 0.2f;
```

Теперь запуск кода дает такой результат:

```
3.1 + 5.2 = 8.3
0.10000000149011612 + 0.20000000298023224 = 0.30000000447034836
false
```

Дело в том, что 64-битный `double` способен обеспечить гораздо большую точность и прецизионность, чем 32-битный `float`. Фактически, `double` обеспечивает точность в 54 бита, в то время как `float` — только в 24 бита. Использование типа с большей точностью помогает нам получить более точный результат.

**Примечание.** Более подробную информацию об этой проблеме можно найти на сайте **https://0.30000000000000004.com/**.

### Модульное тестирование

В предыдущем разделе, где мы перегрузили наш метод `add()` для работы с типами с плавающей точкой, мы не создали в классе `MathExamplesTests` соответствующий юнит-тест. Это не было недосмотром. Следует с осторожностью относиться к модульным тестам, построенным на результатах с плавающей точкой, поскольку, как мы увидели, их нельзя считать детерминированными.

### *BigDecimal*

Один из способов обойти эту проблему — использовать класс `BigDecimal`. Этот класс полезен при выполнении операций с валютой, так как он детерминирован и дает точный результат. `BigDecimal` хранит значения base-10 нативно, поэтому нам не грозят приближения с плавающей точкой, которые происходят при преобразовании десятичных чисел в двоичные.

Сначала мы импортируем классы `BigDecimal` и `RoundingMode`:

```
import java.math.BigDecimal;
import java.math.RoundingMode;
```

Далее мы еще раз перегрузим наш метод `add()`:

```
public static BigDecimal add(BigDecimal bdNum1, BigDecimal bdNum2) {
    return bdNum1.add(bdNum2);
}
```

Вернувшись в метод `main()`, мы переопределим `0.1` и `0.2` как `BigDecimals`, создадим объект класса `RoundingMode` и выведем результат на экран:

```
RoundingMode rmHalfUp = RoundingMode.HALF_UP;
BigDecimal bdNumE = new BigDecimal(0.1).setScale(1, rmHalfUp);
BigDecimal bdNumF = new BigDecimal(0.2).setScale(1, rmHalfUp);

System.out.println(bdNumE + " + " + bdNumF + " = "
+add(bdNumE,bdNumF));
```

Запуск кода даст следующий результат:

```
5.2 + 3.1 = 8.3
0.10000000149011612 + 0.20000000298023224 = 0.30000000447034836
false
0.1 + 0.2 = 0.3
```

Отлично! Это то, что мы ожидали увидеть при попытке сложить `0.1` и `0.2`.

Класс `RoundingMode` предлагает несколько опций для управления поведением округления чисел `BigDecimal`. Эти опции (*IBM 2022*) подробно описаны в табл. 5.1.

***Таблица 5.1.*** *Краткое описание опций, доступных в классе Java* `RoundingMode`

| Опция | Описание |
|---|---|
| `CEILING` | Округление в направлении положительной бесконечности |
| `DOWN` | Округление в направлении нуля |
| `FLOOR` | Округление в направлении отрицательной бесконечности |
| `HALF_DOWN` | Округление до ближайшего соседа; если оба соседа находятся на одинаковом расстоянии, округляет вниз |
| `HALF_EVEN` | Округление до ближайшего соседа; если оба соседа находятся на одинаковом расстоянии, округляет в сторону четного соседа |
| `HALF_UP` | Округление до ближайшего соседа; если оба соседа находятся на одинаковом расстоянии, округляется в сторону увеличения; это поведение `BigDecimal` по умолчанию |
| `UNNECESSARY` | Нет округления |
| `UP` | Округление в сторону от нуля |

## Вычитание

Вернемся к методам арифметики чисел с плавающей точкой. В конце класса `MathExamples` добавим новый метод для вычитания двух чисел типа `double` и получения ответа в виде числа типа `double`. Как и в случае с методом `add()`, этот метод должен перегружать существующий метод `subtract()`:

```
public static double subtract(double dblNum1, double dblNum2) {
    return dblNum1 - dblNum2;
}
```

Точно так же для его вызова добавим оператор `println` в наш метод `main()`:

```
System.out.println(dblNumC + " - " + dblNumD + " = "
    + subtract(dblNumC,dblNumD));
```

Запуск кода должен дать следующий результат:

```
5.2 + 3.1 = 8.3
0.1 + 0.2 = 0.30000000000000004
false
5.2 - 3.1 = 2.1
```

## Умножение

Аналогично тому, как было в случае с операцией вычитания, теперь перегрузим наш метод multiply():

```
public static double multiply(double dblNum1, double dblNum2) {
    return dblNum1 * dblNum2;
}
```

Кроме того, добавим оператор println в конец метода main(), чтобы вызвать и вывести результат:

```
System.out.println(dblNumC + " x " + dblNumD + " = "
    +multiply(dblNumC,dblNumD));
```

Запуск кода должен дать следующий результат:

```
5.2 + 3.1 = 8.3
0.1 + 0.2 = 0.30000000000000004
false
5.2 - 3.1 = 2.1
5.2 x 3.1 = 16.12
```

## Деление

Двигаемся дальше и также перегрузим наш метод divide() для работы с числами с плавающей точкой:

```
public static double divide(double dblNum1, double dblNum2) {
    return dblNum1 / dblNum2;
}
```

Также добавим оператор println в конец метода main():

```
System.out.println(dblNumC + " " + (char)247 + " " + dblNumD + " = "
    +divide(dblNumC,dblNumD));
```

Выполнение кода даст следующий результат:

```
5.2 + 3.1 = 8.3
0.1 + 0.2 = 0.30000000000000004
false
5.2 - 3.1 = 2.1
5.2 x 3.1 = 16.12
5.2 ÷ 3.1 = 1.6774193548387097
```

### Форматирование значений с плавающей точкой

В данном случае мы видим остаток от операции деления, выраженный в десятичном представлении. Однако часто требуется показать более краткое представление числа с плавающей точкой. Мы можем сделать это, ограничив отображаемые точки точности.

### printf

Допустим, необходимо показать только первые три знака десятичной точности остатка. Есть несколько способов добиться этого. Первый — использовать оператор `printf` вместо оператора `println`:

```
System.out.printf("%1.3f " + (char)247 + " %1.3f = %1.3f \n",
   dblNumC, dblNumD, divide(dblNumC,dblNumD));
```

С помощью оператора `printf` мы, по сути, указываем маркеры (известные как *правила форматирования*), определяющие наши числовые значения. Поскольку это типы с плавающей точкой, мы обозначаем их символами `%f`, при этом желаемая точность слева и справа от десятичной точки находится между символом `%` и символом `f`.

Запуск нашего кода теперь должен дать немного другой результат:

```
5.200 ÷ 3.100 = 1.677
```

Можно провести дополнительную настройку, регулируя точность во входных данных, поскольку мы знаем, что там нужен только один десятичный знак:

```
System.out.printf("%1.1f " + (char)247 + " %1.1f = %1.3f \n",
    dblNumC, dblNumD, divide(dblNumC,dblNumD));
```

Теперь вывод выглядит чище:

```
5.2 ÷ 3.1 = 1.677
```

> **Примечание.** Помните, что оператор `printf` не включает символ конца строки, поэтому нужно позаботиться о том, чтобы включить его самостоятельно.

Оператор `printf` поддерживает следующие правила форматирования, перечисленные в табл. 5.2.

***Таблица 5.2.*** *Доступные правила форматирования для использования с оператором* `printf`

| Правило | Описание |
|---|---|
| `%b` | Вывод булевой переменной |
| `%B` | Вывод булевой переменной, переведенной в верхний регистр |
| `%c` | Вывод символьной переменной |
| `%C` | Вывод символьной переменной, переведенной в верхний регистр |
| `%d` | Вывод десятичной целочисленной переменной |
| `%1.1f` | Вывод десятичной переменной с плавающей точкой; можно также указать количество цифр справа и слева от десятичной точки |
| `%n` | Вывод новой строки |
| `%1s` | Вывод строковой переменной, в которой также можно указать минимальную длину |
| `%t` | Вывод переменной даты/времени |

### DecimalFormat

Иногда использование оператора printf не является приемлемым вариантом. Вместо него можно использовать класс DecimalFormat. Сначала импортируем класс DecimalFormat:

```
import java.text.DecimalFormat;
```

Затем можно создать новый объект DecimalFormat с нужной строкой формата и использовать его для ограничения отображения переменных с плавающей точкой:

```
DecimalFormat dFormat = new DecimalFormat("#,###.###");
System.out.println(dblNumC + " " + (char)247 + " " + dblNumD + " = "
    + dFormat.format(divide(dblNumC,dblNumD)));
```

Теперь, когда выполним код, dblNumC и dblNumD должны быть отформатированы так же, как и при использовании оператора printf:

```
5.2 + 3.1 = 8.3
0.10000000149011612 + 0.20000000298023224 = 0.30000000447034836
false
0.1 + 0.2 = 0.3
5.2 - 3.1 = 2.1
5.2 x 3.1 = 16.12
5.2 ÷ 3.1 = 1.677
5.2 ÷ 3.1 = 1.677
```

## Возведение в степень

Далее перегрузим наш метод exponent(), чтобы написать код возведения в степень чисел с плавающей точкой. Это будет проще, чем в случае с целыми числами, поскольку метод Math.pow() естественным образом возвращает тип double:

```
public static double exponent(double base, double power) {
    return Math.pow(base, power);
}
```

Также добавим оператор println в конец метода main() и используем наш объект dFormat:

```
System.out.println(dblNumC + " в степени " + dblNumD + " = "
    + dFormat.format(exponent(dblNumC,dblNumD)));
```

Запуск кода должен дать следующий результат:

```
5.2 + 3.1 = 8.3
0.10000000149011612 + 0.20000000298023224 = 0.30000000447034836
false
0.1 + 0.2 = 0.3
5.2 - 3.1 = 2.1
```

```
5.2 x 3.1 = 16.12
5.2 + 3.1 = 1.677
5.2 + 3.1 = 1.677
5.2 в степени 3.1 = 165.81
```

## Квадратный корень

Мы не рассматривали квадратные корни в разделе целочисленных операций, так как вычисление квадратного корня обычно требует использования типов с плавающей запятой. Давайте создадим новый метод в классе `MathExamples` для определения квадратных корней. В Java нет собственного оператора для определения квадратных корней, поэтому воспользуемся методом `Math.sqrt()`:

```
public static double squareRoot(double number) {
    return Math.sqrt(number);
}
```

Можно добавить `println` в наш метод `main()`, чтобы отобразить результат:

```
System.out.println("Квадратный корень числа " + dblNumC + " = "
    +dFormat.format(squareRoot(dblNumC)));
```

Запуск кода должен дать следующий результат:

```
5.2 + 3.1 = 8.3
0.10000000149011612 + 0.20000000298023224 = 0.30000000447034836
false
0.1 + 0.2 = 0.3
5.2 - 3.1 = 2.1
5.2 x 3.1 = 16.12
5.2 ÷ 3.1 = 1.677
5.2 ÷ 3.1 = 1.677
5.2 в степени 3.1 = 165.81
Квадратный корень числа 5.2 = 2.28
```

## Кубический корень

Мы не будем приводить здесь соответствующий пример (поскольку код похож на тот, в котором вычислялся квадратный корень), но библиотека `Math` в Java также содержит метод кубического корня. Мы можем использовать эту библиотеку, чтобы легко написать свой собственный метод:

```
public static double cubeRoot(double number) {
    return Math.cbrt(number);
}
```

### Модуль числа

Иногда полезно показать значение без влияния его знакового бита. В таких случаях неважно, положительная или отрицательная переменная, поэтому выполняется операция получения абсолютного значения, чтобы убрать знак (по сути, заставляя число быть положительным). Для этого пригодится метод `Math.abs()`:

```
public static double absoluteVal(double number) {
    return Math.abs(number);
}
```

Чтобы продемонстрировать его работу, вызовем наш метод `absoluteVal()` дважды, по одному разу для положительного и отрицательного числа:

```
System.out.println("Абсолютное значение числа " + dblNumC + " = "
    + dFormat.format(absoluteVal(dblNumC)));

double dblNumG = -9f;

System.out.println("Абсолютное значение числа " + dblNumG + " = "
    + dFormat.format(absoluteVal(dblNumG)));
```

Теперь, когда мы выполним этот код, в выводе появятся следующие две строки:

```
Абсолютное значение числа 5.2 = 5.2
Абсолютное значение числа -9.0 = 9
```

## Заключение

В этой главе мы рассмотрели целочисленную арифметику и арифметику чисел с плавающей точкой. Было продемонстрировано, как можно использовать библиотеку `Math` в Java для некоторых более сложных арифметических операций. Мы также рассмотрели различные способы форматирования значений с плавающей точкой для соответствующих типов вывода.

Мы рассмотрели модульное тестирование и узнали, как использовать его, чтобы показать, что наши арифметические методы дают ожидаемые результаты. Важно отметить, что модульное тестирование — это отдельная дисциплина. О модульном тестировании и разработке с использованием тестов написаны целые книги, а мы уделили этому лишь часть главы. Настоятельно рекомендуем изучить модульное тестирование самостоятельно, поскольку информации о нем имеется гораздо больше, чем в той небольшая части, которую мы рассмотрели здесь.

Были рассмотрены некоторые особенности целочисленной арифметики и арифметики чисел с плавающей точкой. Для нас, разработчиков, очень важно понимать, как ведут себя арифметические операции в определенных обстоятельствах. Многие разработчики, работая над сложными распределенными системами, были озадачены тем, как целочисленное деление используется для вычисления таких вещей, как

кворум репликации. Понимание материала этой главы является основополагающим не только для программирования на Java в частности, но и для более широкого мира компьютеров в общем.

В следующей главе мы обсудим такие структуры данных, как стеки, очереди и связные списки. Мы также рассмотрим двоичные деревья и покажем пример их использования.

## Важно помнить

- При выполнении операций в строке или операторе `print` не забудьте заключить их в дополнительное множество круглых скобок, чтобы избежать преждевременного приведения к `String`.
- Примитивные типы работают и хранятся в двоичной системе счисления ("base-2").
- Библиотека `JUnit` позволяет создавать модульные (юнит) тесты для обеспечения согласованного поведения отдельных методов.
- Операции умножения и деления используют `*` и `/` вместо x и ÷, соответственно.
- Остаток от операции деления целого числа можно получить с помощью оператора модуля `%`.
- В Java нет собственных операторов для некоторых более сложных арифметических операций. Для этих операций можно обратиться к библиотеке `Math` в Java, содержащей такие дополнительные методы, как `pow()`, `sqrt()`, `abs()` и многие другие.
- Арифметика чисел с плавающей точкой в значительной степени является приближенной и должна использоваться с осторожностью.
- Если требуются точные цифры (например, при вычислении валюты), вместо `float` или `double` следует использовать тип `BigDecimal`.
- На форматы вывода плавающей точки можно влиять с помощью оператора `printf` или класса `DecimalFormat`.

Глава 6

# Общие структуры данных

## Введение

В этой главе мы рассмотрим, как создавать и использовать некоторые общие фундаментальные структуры для управления и работы с данными в Java. Сначала мы поговорим о стеках и очередях и рассмотрим их построение. Затем обсудим различные типы связных списков. Наконец, рассмотрим двоичные деревья и продемонстрируем их использование на простом примере.

## Структура

В этой главе мы обсудим структуры данных, которые обычно используются при разработке программного обеспечения и в низкоуровневых вычислениях.

- Стеки.
- Очереди.
- Связные списки.
- Двоичные деревья.

## Цели

Основная цель этой главы — сформировать понимание общих базовых структур данных. Мы рассмотрим каждую структуру данных, построим ее, а затем разберем пример. Нашими задачами являются:

- понять поведение стеков и очередей;
- узнать, как используются стеки и очереди;
- разобраться, что такое связные списки и двоичные деревья и как их обходить.

# Стеки

Начнем с создания *стека* — фундаментальной структуры, используемой в различных областях вычислительной техники. Он характеризуется поведением по принципу LIFO (Last In, First Out — "последний вошел, первый вышел"). Элементы могут быть добавлены только в верхнюю часть стека в операции, известной как push. Если в стек добавляется или заталкивается ("запушивается") другой элемент, то самый последний элемент перемещается вниз, и новый элемент оказывается на вершине стека (рис. 6.1).

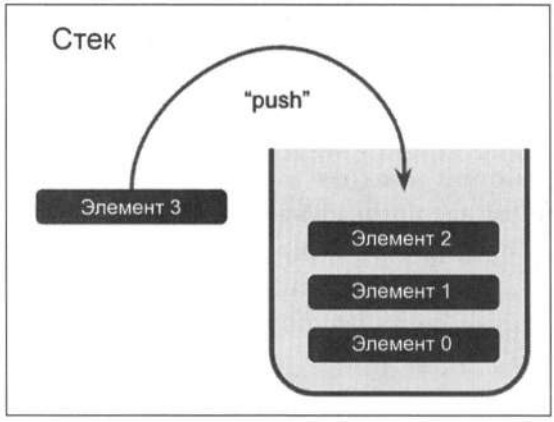



**Рис. 6.1.** Демонстрация операции push для добавления элемента в стек

Независимо от того, где могут находиться элементы в стеке, доступ можно получить только к самому верхнему элементу. Если выполняется операция pop, самый верхний элемент удаляется из вершины стека и возвращается обратно (рис. 6.2).

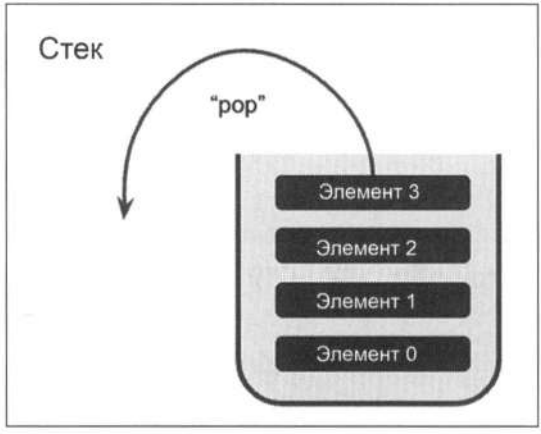



**Рис. 6.2.** Демонстрация операции pop по удалению элемента из стека

В нашей IDE создадим новый класс с именем `Item` и убедимся, что он находится в новом пакете с именем `chapter6`. У этого класса не должно быть метода `main`:

```
package chapter6;

public class Item {
```

Наш класс `Item` должен иметь закрытое (private) свойство с именем `name` и типом `String`, и конструктор без аргументов:

```
    private String name;
    public Item() {
}
```

> **Примечание.** `Item` также можно определить как `Record`. Однако в данном случае определение элемента как POJO дает нам дополнительную гибкость, необходимую для использования его со стеками, очередями и связными списками.

Также создадим одноаргументный конструктор, который принимает в качестве параметра переменную `name` типа `String` и устанавливает ее равной свойству `name`. Классу `Item` также понадобится пара геттер/сеттер для свойства `name`:

```
    public Item(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Далее создадим новый класс в пакете `chapter6` с названием `Stack`. Как и класс `Item`, он не должен иметь метода `main`. Обязательно определите массив типа `Item` с именем `items`, а также целочисленные свойства именами `maxCount` и `stackCount`:

```
package chapter6;

public class Stack {
    private Item[] items;
    private int maxCount;
    private int stackCount = 0;
```

Мы хотим, чтобы у класса `Stack` было два конструктора. Один из них принимает целое число для ожидаемого количества элементов.

Второй не принимает никаких параметров, а просто вызывает другой конструктор с целочисленным значением 10:

```
public Stack() {
    this(10);
}

public Stack(int numItems) {
    maxCount = numItems;
    items = new Item[maxCount];
}
```

Далее нам понадобится метод private void с именем resizeArray. Если мы увеличим наш стек сверх максимального размера массива items, то понадобится способ изменить его размер. По умолчанию будем каждый раз увеличивать его на 5.

Изменение размера массива невозможно в Java. Поэтому просто создадим новый массив и скопируем в него текущий массив элементов:

```
private void resizeItemsArray() {
    maxCount = maxCount + 5;
    Item[] newArray = new Item[maxCount];

        for (int counter = 0; counter < stackCount; counter++) {
            newArray[counter] = items[counter];
    }
    items = newArray;
}
```

В приведенном коде мы делаем следующее:

- создаем новый массив (newArray), в котором будет на пять элементов больше, чем текущее значение maxCount;
- копируем элементы из массива items в newArray;
- перезаписываем items значением newArray.

**Примечание.** Сейчас самое время использовать метод Collections.copy(), но массивы не наследуются от базового класса Collection.

Далее напишем public-метод, который будет обрабатывать операцию push для добавления новых элементов в стек. Сначала необходимо проверить, не превышает ли новый размер массива текущий maxCount, а затем вызвать наш метод resizeItemsArray(), если превышает. После этого увеличим stackCount на единицу, а затем добавим новый элемент item в позицию с индексом [stackCount минус один] (единица вычитается из-за того, что индексация начинается с нуля):

```
public void push(Item item) {
    if (stackCount + 1 >= maxCount) {
        resizeItemsArray();
    }
```

```
    stackCount++;
    items[stackCount - 1] = item;
}
```

В стеках также присутствует операция peek, "открывающая окно" для верхнего элемента в стеке. Это позволяет просматривать верхний элемент, не удаляя его:

```
public Item peek() {
    Item returnVal = items[stackCount - 1];
    return returnVal;
}
```

Давайте напишем public-метод для обработки операции pop. Метод pop() довольно прост. Мы определяем новый объект Item в качестве возвращаемого значения и инстанцируем его результатом нашего метода peek(), который должен быть верхним элементом. Затем уменьшаем значение stackCount и возвращаем извлеченный элемент:

```
public Item pop() {
    Item returnVal = peek();
    stackCount--;
    return returnVal;
}
```

> **Примечание.** Мы не удаляем элемент из списка. Мы просто уменьшаем stackCount (количество элементов в стеке), чтобы этот элемент оказался вне диапазона для наших операций, основанных на стеке. Теперь можно удалить элемент, установив на его месте значение null или пустую строку. Мы даже можем уменьшить или сократить размер itemsArray, если хотим эффективно использовать память.

Однако выполнение этих операций сопряжено с определенными затратами, которые измеряются как в производительности, так и в увеличении сложности кода. Проще уменьшить рабочий диапазон элементов и игнорировать все, что находится за его пределами.

Чтобы завершить класс Stack, нужно просто добавить еще два метода:

- геттер для stackCount;
- перегруженный метод toString(), чтобы можно было контролировать форматирование по умолчанию.

Метод getStackCount() используется для возврата целочисленной private-переменной[1] stackCount. Важно отметить, что мы не собираемся создавать сеттер для

[1] Переменные, которые видны и доступны только в пределах класса, которому они принадлежат. — *Прим. ред.*

stackCount, так как он используется внутри класса Stack, и вызывающий метод не должен иметь возможности напрямую устанавливать это значение:

```
public int getStackCount() {
    return stackCount;
}
```

Для перегруженного метода toString() мы выполняем итерацию по массиву items и выводим в одной строке и элемент, и его индекс. Поскольку в стеке первый элемент (index == 0) находится *внизу*, будем перебирать элементы items в обратном порядке:

```
    public String toString() {
        StringBuilder returnVal = new StringBuilder("\n");

            for (int counter = stackCount - 1; counter >= 0; counter--) {
                returnVal.append(counter);
                returnVal.append(" - ");
                returnVal.append(items[counter].getName());
                returnVal.append("\n");
            }

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

Подготовив два вспомогательных класса, создадим новый класс DataStructuresExamples внутри пакета chapter6. Убедитесь, что у этого класса есть метод main(). Первое, что мы сделаем в нашем методе main(), — это создадим новый объект Stack с именем stack:

```
package chapter6;

public class DataStructuresExamples {

    public static void main(String[] args) {
        Stack stack = new Stack();
```

Чтобы показать, как работает наш стек, можно создать несколько новых объектов класса Item на примере создания стека для управления списком воспроизведения фильмов:

```
Item martian = new Item("The Martian");
Item patriotGames = new Item("Patriot Games");
Item bladerunner = new Item("Blade Runner");
Item bladerunner2049 = new Item("Blade Runner 2049");
Item apollo13 = new Item("Apollo 13");
Item firstMan = new Item("First Man");
```

```
Item empireStrikesBack = new Item("The Empire Strikes Back");
Item rogueOne = new Item("Rogue One");
Item alexander = new Item("Alexander");
Item starwars = new Item("Star Wars");
Item runningMan = new Item("Running Man");
```

Создав элементы фильма, выведем на экран сообщение о том, что мы работаем со стеком, а затем поместим при помощи push шесть фильмов (по отдельности) в этот стек:

```
System.out.println("Пример стека:");

stack.push(firstMan);
stack.push(apollo13);
stack.push(rogueOne);
stack.push(empireStrikesBack);
stack.push(bladeRunner2049);
stack.push(bladeRunner);

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

Запуск кода приведет к следующему результату:

```
Пример стека:

5 - Blade Runner
4 - Blade Runner 2049
3 - The Empire Strikes Back
2 - Rogue One
1 - Apollo 13
0 - First Man
```

Если результат совпадает с тем, что приведено, значит, пока все хорошо. Теперь поэкспериментируем с операциями push и println и посмотрим на результат. К примеру, вытащим (pop) из стека один фильм, поместим (push) еще несколько и, возможно, вытащим/поместим еще один:

```
System.out.println(stack.pop().getName() + " был извлечен из стека.");

stack.push(patriotGames);
stack.push(martian);
stack.push(alexander);
stack.push(runningMan);

System.out.println(stack.pop().getName() + " был извлечен из стека.");

stack.push(starwars);
```

Теперь выполнение кода должно дать следующий результат:

```
Blade Runner был извлечен из стека.
Running Man был извлечен из стека.

8 - Star Wars
7 - Alexander
6 - The Martian
5 - Patriot Games
4 - Blade Runner 2049
3 - The Empire Strikes Back
2 - Rogue One
1 - Apollo 13
0 - First Man
```

Как было показано, мы работали с объектом стека, "затолкав" в него несколько фильмов и вытащив из него два, а затем отобразив результаты.

Вспомните начало этой главы, когда мы говорили о том, что стеки используются в самых разных областях вычислительной техники. Так вот, в Java есть стековые структуры, с которыми мы постоянно работаем и не замечаем их, потому что они абстрагированы от нас.

Рассмотрим только что написанную программу. При выполнении нашего класса `DataStructureExamples` метод `main()` выталкивается в стек для хранения методов и их данных, что обеспечивает выполнение методов до конца по принципу LIFO. Например, по мере работы метода `main()` его выполнение часто откладывается до завершения вновь вызванного метода, как показано на рис. 6.3.

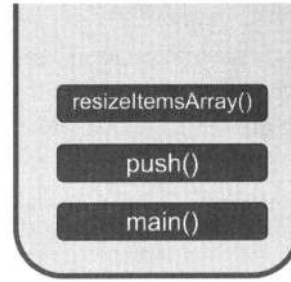



**Рис. 6.3.** Визуализация того, как JVM использует стек для управления порядком выполнения методов

Как видим, стеки — это полезные структуры данных, широко распространенные в вычислительной технике.

# Очереди

Далее мы перейдем к построению очереди. В отличие от стека, *очередь* более универсальна. Новые элементы добавляются в конец очереди, а самые старые элементы обрабатываются первыми в порядке FIFO (First In, First Out — "первый вошел, первый вышел).

Мы можем найти множество примеров использования очередей в вычислительной технике. Брокеры потоковых сообщений, такие как Apache Kafka и Apache Pulsar, хранят данные в структурах, называемых топиками, которые, по сути, являются распределенными очередями. Вы когда-нибудь интересовались, какая песня будет следующей в музыкальном потоковом сервисе? За это также отвечает очередь. Отправляете задание на принтер на работе? Возможно, придется столкнуться с тем, что ваши коллеги делают то же самое, и задание попадает в очередь на печать.

Давайте создадим новый класс с именем `Queue` внутри пакета `chapter6`. Как и в случае с классом `Stack`, наш класс `Queue` не должен иметь метода `main`. Аналогично, добавим private-свойства для массива `items`, а также целых чисел `maxCount` и `queueCount`:

```
package chapter6;

public class Queue {
    private Item[] items;
    private int maxCount;
    private int queueCount = 0;
```

Такой же подход применим и к конструкторам. По сути, можно вызвать конструктор без аргументов, чтобы создать массив `items` с местом для 10 элементов по умолчанию. Этот конструктор вызовет другой наш конструктор, который принимает число элементов в качестве единственного аргумента и использует его для инициализации `maxCount` и `items`:

```
public Queue() {
    this(10);
}

public Queue(int numItems) {
    maxCount = numItems;
    items = new Item[maxCount];
}
```

Также воспользуемся private-методом `resizeItemsArray()` из класса `Stack`, изменив его код для использования `queueCount`:

```
private void resizeItemsArray() {
    // увеличиваем максимальное количество элементов на 5
    maxCount = maxCount + 5;
```

```
    // создание массива элементов большего размера
    Item[] newArray = new Item[maxCount];

    // копирование массива текущих элементов в новый
    for (int counter = 0; counter < queueCount; counter++) {
        newArray[counter] = items[counter];
    }

    // переопределение текущего массива элементов на новый массив
    items = newArray;
}
```

В очередях обычно есть методы, позволяющие просматривать первые и последние элементы в этой очереди. Для этого мы создадим два геттера:

```
public Item getFront() {
    if (queueCount > 0) {
        return items[0];
    } else {
        return null;
    }
}

public Item getBack() {
    if (queueCount > 0) {
        return items[queueCount - 1];
    } else {
        return null;
    }
}
```

Оба метода вернут правильный элемент только в том случае, если массив `items` не пуст. Если массив пуст, возвращается значение `null`.

> **Примечание.** Будьте осторожны при возврате `null` из метода. В этом случае вызывающий метод должен быть очень внимателен к тому, как обрабатываются возвращаемые значения, иначе может возникнуть исключение `NullPointerException`.

Далее создадим метод `enqueue()`. Он похож на метод `push()` из класса `Stack`, в котором мы сначала проверяли новый размер массива `items` и при необходимости осуществляли увеличение. В любом случае мы увеличим `queueCount` и добавим новый элемент в конец очереди:

```
public void enqueue(Item item) {
    if (queueCount + 1 >= maxCount) {
        resizeItemsArray();
    }
```

```
    queueCount++;
    items[queueCount - 1] = item;
}
```

Удаление элемента с помощью метода `dequeue()` — немного более сложная задача. Для начала нужно убедиться, что очередь не пуста. Если она пуста, вызов метода `dequeue()` должен вернуть `null`.

После того как мы получили множество начальных элементов в качестве возвращаемого значения, нужно переместить каждый элемент вниз на одну индексную позицию. После этого можно уменьшить значение `queueCount` и вернуть первый элемент:

```
public Item dequeue() {
    if (queueCount == 0) {
        return null;
    } else {
        Item returnVal = getFront();

        // перемещаем все другие элементы вниз
        for (int counter = 1; counter < queueCount; counter++) {
            items[counter - 1] = items[counter];
        }

        queueCount--;
        return returnVal;
    }
}
```

Можно завершить работу над классом `Queue`, создав общедоступный метод-геттер для `queueCount` и перегрузив метод `toString()` (по аналогии с классом `Stack`):

```
public int getQueueCount() {
    return queueCount;
}

public String toString() {
    StringBuilder returnVal = new StringBuilder("\n");

    for (int counter = 0; counter < queueCount; counter++) {
        returnVal.append(counter);
        returnVal.append(" - ");
        returnVal.append(items[counter].getName());
        returnVal.append("\n");
    }

    return returnVal.toString();
}
```

Далее вернемся к нашему классу DataStructuresExamples и добавим следующий код в конец метода main():

```
System.out.println("Пример очереди:");

Queue queue = new Queue();

queue.enqueue(starwars);
queue.enqueue(bladeRunner);
queue.enqueue(empireStrikesBack);
queue.enqueue(patriotGames);
queue.enqueue(bladeRunner2049);

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

Если запустим наш код, то увидим предыдущий результат из примера со стеком и этот результат:

```
Пример очереди:


0 - Star Wars
1 - Blade Runner
2 - The Empire Strikes Back
3 - Patriot Games
4 - Blade Runner 2049
```

Судя по этому результату, наша очередь действительно индексирует элементы по мере их добавления, причем начальный элемент (Star Wars) находится в позиции с индексом 0, а конечный (Blade Runner 2049) — в позиции с индексом 4.

Давайте внесем еще одну поправку в метод main() для примера Queue, вызвав метод dequeue() дважды, а затем еще раз распечатав содержимое очереди:

```
System.out.println(queue.dequeue().getName() + " удален.");
System.out.println(queue.dequeue().getName() + " удален.");

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

Выполнение нашего кода должно привести к таким результатам для примера Queue:

```
Пример очереди:


0 - Star Wars
1 - Blade Runner
2 - The Empire Strikes Back
3 - Patriot Games
4 - Blade Runner 2049
```

```
Star Wars удален.
Blade Runner удален.

0 - The Empire Strikes Back
1 - Patriot Games
2 - Blade Runner 2049
```

Обратите внимание, что после того, как первые два элемента были удалены из списка, индексы элементов были дважды уменьшены (по одному разу для каждого удаления).

## Связные списки

Давайте перейдем к связным спискам. Мы обсуждали связные списки еще в *главе 4 "Массивы, коллекции и записи"*. Мы создадим свой собственный класс связного списка, чтобы убедиться, что понимаем, какие концепции здесь работают.

Сначала нам нужно внести некоторые изменения в наш класс `Item`. Мы создадим двусвязный список. Как уже говорилось в *главе 4*, это список, в котором элементы связаны со следующим и предыдущим элементами. Для реализации этого нашему классу `Item` тоже нужны эти ссылки. В классе `Item` добавим еще два свойства класса:

```
private Item prevItem;
private Item nextItem;
```

Мы также добавим геттер и сеттер для работы с этими свойствами:

```
public Item getPrevItem() {
    return this.prevItem;
}

public void setPrevItem(Item item) {
    this.prevItem = item;
}

public Item getNextItem() {
    return this.nextItem;
}

public void setNextItem(Item item) {
    this.nextItem = item;
}
```

Теперь может показаться, что в класс `Item` добавлено слишком много всего, особенно если учесть, что наши реализации `Stack` и `Queue` не будут использовать такой функционал. Но если свойства `prevItem` и `nextItem` не используются, то они будут

равны null, а ссылки на нулевой указатель в Java занимают ничтожное количество памяти в куче JVM.

**Примечание.** Добавление этих свойств делает `Item` самореферентным[2] классом.

Далее создадим новый класс в пакете `chapter6` с именем `LinkedList`. У этого класса не должно быть метода `main`. Мы создадим три свойства класса для обработки ссылок на первый и последний элементы, а также целое число для подсчета количества элементов в списке:

```
package chapter6;

public class LinkedList {
    private Item firstItem;
    private Item lastItem;
    private int listCount = 0;
```

Обратите внимание, что на этот раз мы не определяли массив. Наш класс `LinkedList` требует определения только первого и последнего элементов. Свойство `firstItem` будет содержать ссылку на второй элемент, а тот — на третий, и так далее, пока мы не доберемся до последнего элемента.

Наш класс `LinkedList` будет иметь следующие два конструктора:

- конструктор без аргументов, не содержащий никакого кода;
- конструктор с одним аргументом, который инстанцирует список с одним элементом.

Давайте создадим конструкторы:

```
public LinkedList() {
}

public LinkedList(Item item) {
    addItem(item);
}
```

**Примечание.** Несмотря на то, что наш безаргументный конструктор не содержит кода, его наличие все равно необходимо, чтобы класс мог быть инстанцирован (как объект) без параметров.

Далее мы будем работать над добавлением элементов в список. Пока что наши элементы будут добавляться только в *начало* списка. Чтобы реализовать наш новый метод `add()`, необходимо иметь способ обработки добавления одного элемента в пустой список, поскольку мы будем выполнять действия, которые понадобятся только в этом случае.

---

[2] Самореферентный класс в Java — это универсальный тип, который ссылается на себя. — *Прим. ред.*

Поэтому для этого мы напишем специальный private-метод `setWithOneItem`:

```
private void setWithOneItem(Item newItem) {
    firstItem = newItem;
    lastItem = newItem;
}
```

От метода `setWithOneItem()` не требуется многого, но это единственный случай, когда нам нужно установить и `firstItem`, и `lastItem` на *один и тот же* элемент.

Далее создадим новый метод с именем `addItem`. Он будет принимать элемент с именем `newItem` в качестве единственного параметра, проверять, является ли список пустым, и при необходимости вызывать метод `setWithOneItem()`. Если `listCount` не равен нулю, то для добавления нового элемента необходимо выполнить три действия:

- установить текущий `firstItem` в качестве следующего элемента для `newItem`;
- установить предыдущий элемент исходного `firstItem` в качестве `newItem`;
- установить `newItem` в качестве нового первого элемента в списке.

И не забудем увеличить `listCount`, независимо от того, какая логика будет применена:

```
public void addItem(Item newItem) {
    if (listCount == 0) {setWithOneItem(newItem);
} else {newItem.setNextItem(firstItem);firstItem.setPrevItem(newItem);firstItem =
newItem;
}
listCount++;
}
```

Также мы хотим иметь возможность находить элемент по имени. В этом методе мы пройдемся по списку, начиная с `firstItem`. Итерация происходит путем вызова метода `getNextItem()` каждого элемента. Если найден элемент, соответствующий нашей строке `name`, то замыкаем цикл и возвращаем этот элемент. Если мы дошли до последнего элемента (который имеет значение `null` для своего `nextItem`), мы возвращаем `null`:

```
public Item findItemByName(String name) {

    Item currentItem = firstItem;
        while (currentItem != null) {
            if (currentItem.getName().equals(name)) {
                // нашли!
                return currentItem;
            }currentItem = currentItem.getNextItem();
        }
    return null;
}
```

Аналогичным образом необходимо иметь возможность *удалять* элемент по имени. Можно воспользоваться написанным нами методом `findItemByName()`, чтобы найти элемент по имени. Удаление элемента из списка — это простая задача "*осиротить*" элемент, переуказав его `nextItem` и `previousItem` друг на друга:

```
public boolean removeItemByName(String name) {

    Item itemFound = findItemByName(name);

    if (itemFound != null) {
        Item previous = itemFound.getPrevItem();
        Item next = itemFound.getNextItem();

        previous.setNextItem(next);
        next.setPrevItem(previous);

        listCount--;
        return true;
    }

    return false;
}
```

Нам также понадобятся методы-геттеры для наших свойств. Все три свойства управляются изнутри, поэтому нам не нужно создавать для них методы-сеттеры:

```
public Item getFirstItem() {
    return firstItem;
}

public Item getLastItem() {
    return lastItem;
}

public int getListCount() {
    return this.listCount;
}
```

Наконец, нашему классу `LinkedList` необходим перегруженный метод `toString()`, позволяющий применять определенное форматирование для имен элементов:

```
public String toString() {

    StringBuilder returnVal = new StringBuilder("\n");

    Item item = firstItem;

    while (item != null) {
        returnVal.append(item.getName());
```

```
        returnVal.append("\n");
        item = item.getNextItem();
    }

return returnVal.toString();

}
```

Возвращаясь к классу `DataStructureExamples`, теперь можно реализовать наш `LinkedList`. Чтобы быть последовательными с нашим предыдущим примером из *главы 4 "Массивы, коллекции и записи"*, мы будем в качестве элементов использовать те же элементы данных (вымышленные названия городов). Это приведет к созданию логической структуры, подобной той, что показана на рис. 6.4.

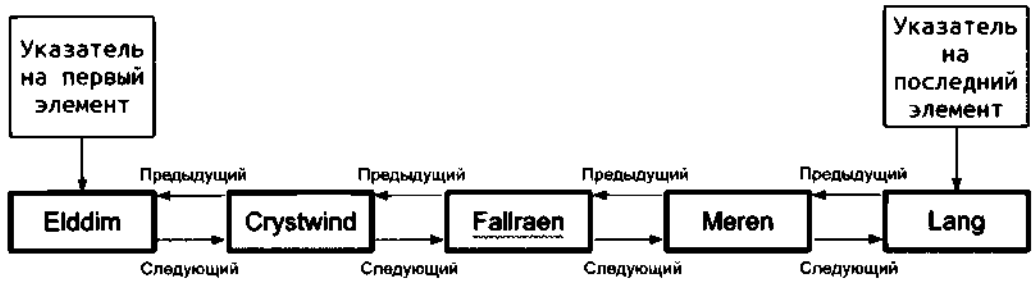



**Рис. 6.4.** Визуальное представление нашего связного списка, содержащего те же названия городов, которые использовались в *главе 4*

Для этого создадим элементы (города), добавим их в `LinkedList` и отобразим его содержимое:

```
System.out.println("Пример связного списка:");

LinkedList linkedList = new LinkedList();
Item elddim = new Item("Elddim");
Item crystwind = new Item("Crystwind");
Item fallraen = new Item("Fallraen");
Item meren = new Item("Meren");
Item lang = new Item("Lang");
Item hiroth = new Item("Hiroth");

linkedList.addItem(lang);
linkedList.addItem(meren);
linkedList.addItem(fallraen);
linkedList.addItem(crystwind);
linkedList.addItem(elddim);

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

Если запустить этот код, то увидим следующий результат:

```
Пример связного списка:

Elddim
Crystwind
Fallraen
Meren
Lang
```

Теперь давайте проверим, присутствуют ли в списке два города. Это можно сделать, добавив данный код в конец метода `main()` класса `DataStructuresExamples`:

```
System.out.println("Содержит ли список " + crystwind.getName() + "?");

if (linkedList.findItemByName(crystwind.getName()) != null) {
    System.out.println("Да!");
} else {
    System.out.println("Нет, не обнаружен.");
}

System.out.println("Содержит ли список " + hiroth.getName() + "?");

if (linkedList.findItemByName(hiroth.getName()) != null) {
    System.out.println("Да!");
} else {
    System.out.println("Нет, не обнаружен.");
}
```

Запуск этого кода должен привести к следующему результату:

```
Пример связного списка:

Elddim
Crystwind
Fallraen
Meren
Lang

Содержит ли список Crystwind?
Да!
Содержит ли список Hiroth?
Нет, не обнаружен.
```

Теперь давайте в последний раз подкорректируем `DataStructuresExamples`, удалив город `Meren`:

```
System.out.println("Теперь удалим" + meren.getName());
linkedList.removeItemByName(meren.getName());

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

Выполним код и посмотрим, что получилось:

```
Пример связного списка:

Elddim
Crystwind
Fallraen
Meren
Lang

Содержит ли список Crystwind?
Да!
Содержит ли список Hiroth?
Нет, не обнаружен.
Теперь удалим Meren

Elddim
Crystwind
Fallraen
Lang
```

Как видно, мы успешно удалили один из городов.

Важно понять, что связному списку не нужен (или не требуется) индекс, чтобы быстро найти нужный элемент. Все, что есть у нашего класса `LinkedList`, — это объекты, указывающие на первый и последний элемент списка. Остальная часть списка хранится в отношениях между следующим и предыдущим свойствами каждого элемента.

Обратите внимание, что мы создали двусвязный список, что означает, что элементы нашего связного списка знают как о *следующем*, так и о *предыдущем* элементе. Это также означает, что мы можем работать с нашим списком, начиная с начала или с конца. Например, если необходимо добавить элементы в конец списка, можно поступить следующим образом:

```
public void addItemAtBack(Item newItem) {

    if (listCount == 0) {
        setWithOneItem(newItem);
    } else {
```

```
        newItem.setPrevItem(lastItem);
        lastItem.setNextItem(newItem);
        lastItem = newItem;
    }
}
```

Преимущества связного списка становятся очевидными, когда мы понимаем, что он не ограничивается массивом или коллекцией. Это также означает, что элементы связного списка не нужно хранить в памяти (в куче). Когда мы пополняем список, нет необходимости изменять его размер. Кроме того, в данном случае мы добавляем только в начало списка, что означает (в сочетании с отсутствием необходимости изменять размер), что операции добавления `add` выполняются очень быстро.

Связные списки широко используются в вычислительной технике. Хорошим примером является веб-браузер, который использует связный список за кнопками "назад" и "вперед", чтобы пользователи могли быстро переходить от одной веб-страницы к другой. Музыкальные плейлисты — это еще один тип связного списка. Можно задать *цикл* для музыкального плейлиста, в этом случае последний элемент будет *следующим* за первым, создавая круговой связный список.

# Двоичные деревья

Еще одной распространенной структурой данных является *двоичное (бинарное) дерево*. Двоичные деревья являются коллекциями данных, представляющие узлы, которые связаны между собой ветвями. Первый узел, добавляемый в двоичное дерево, называется корнем. Корневой узел, как и все узлы, может соединяться с двумя другими узлами как слева, так и справа.

Если к узлу нужно добавить новый узел, проверяется значение нового узла. Если значение нового узла меньше значения текущего узла, он соединяется с ним слева. Если значение узла больше значения текущего узла, он соединяется с ним справа. Таким образом, двоичное дерево может быстро отсортировать множество данных.

Начнем с класса `Node`. Создайте новый класс с именем `Node` внутри пакета `chapter6`. У этого класса не должно быть метода `main`. Создадим private-свойства для целочисленного значения и еще два объекта `Node` с именами `leftNode` и `rightNode`:

```
package chapter6;

public class Node {

    private int value;
    private Node leftNode;
    private Node rightNode;
```

**Примечание.** Определение класса `Node` содержит два объекта класса `Node`: `leftNode` и `rightNode`. Это известно как самореферентный класс. Таким образом, один узел будет поддерживать ссылки на другие узлы в дереве.

Наш класс Node будет иметь один конструктор, который будет принимать один целочисленный параметр. Это целое число будет использоваться для инициализации свойства value узла:

```
public Node (int number) {
    this.value = number;
}
```

Далее узел должен знать правила добавления нового узла. Поэтому нам нужно создать метод insert(). Этот метод будет принимать в качестве параметра целое число с именем newNumber. Если newNumber меньше значения текущего узла, то по нашей логике мы перемещаемся вниз по левой стороне узла. Если newNumber больше значения текущего узла, то по логике перемещаемся вниз по правой стороне узла:

```
public void insert(int newNumber) {
    if (newNumber < this.value) {
        if (leftNode == null) {
            leftNode = new Node(newNumber);
        } else {
            leftNode.insert(newNumber);
        }
    } else if (newNumber > this.value) {
    if (rightNode == null) {
            rightNode = new Node(newNumber);
        } else {
            rightNode.insert(newNumber);
        }
    }
}
```

Вне зависимости от того, с какой стороны мы двигаемся вниз (справа или слева), если узел этой стороны равен null (не имеет значения), мы достигли "листа" (концевой вершины). Это означает, что можно сделать узел этой стороны новым узлом (инициализированным с помощью newNumber). Однако если у этого узла есть значение, то мы вызываем метод insert() для этого узла, и процесс повторяется до тех пор, пока не будет найден узел-лист и не будет вставлен новый узел.

> **Примечание.** Рассматривая предыдущий код, обратите внимание, что у нас нет условия if для проверки, равен ли newNumber значению текущего узла. Узел двоичного дерева в его текущем состоянии не принимает дубликаты. Если дубликат будет добавлен, он будет просто проигнорирован.

Можно завершить работу над классом Node, создав три *геттера* для наших свойств:

```
public Node getLeftNode() {
    return this.leftNode;
}
```

```
public Node getRightNode() {
    return this.rightNode;
}

public int getValue() {
    return this.value;
}
```

Далее создадим класс `Tree`. Убедитесь, что этот класс называется `Tree`, не имеет метода `main` и находится внутри пакета `chapter6`. У него должно быть только одно свойство — объект типа `Node` с именем `root`:

```
package chapter6;

public class Tree {
    private Node root;
```

Наш класс `Tree` будет иметь два конструктора:

- конструктор без аргумента;
- конструктор с одним аргументом, принимающий целое число и инициализирующий `root` как новый `Node` целочисленным значением.

Давайте теперь создадим конструкторы:

```
public Tree() {
}

public Tree(int number) {
    root = new Node(number);
}
```

Класс `Tree` также будет иметь метод `insert()`, который установит `root` узлом, если он равен `null`, или вызовет метод `insert()` объекта `root`, если у него есть значение:

```
public void insert(int number) {
if (root == null) {
        root = new Node(number);
    } else {
        root.insert(number);
    }
}
```

Теперь давайте рассмотрим обход дерева. Мы создадим два метода: private-метод `traverse` и public-метод `traverseFromRoot`. Метод `traverseFromRoot()` будет вызывать private-метод `traverse()`, передавая в качестве параметра узел `root`. Метод `traverse()` проверяет, не является ли переданный ему узел пустым. Если да, то выполнение возвращается без каких-либо дальнейших действий. Если у узла есть зна-

чение, то метод `traverse()` рекурсивно вызывает себя на левом и правом узлах, пока не достигнет узла-листа:

```
public void traverseFromRoot() {
    traverse(root);
}

private void traverse(Node node) {

    if (node == null) {
        return;
    }

    traverse(node.getLeftNode());
    System.out.println(node.getValue());
    traverse(node.getRightNode());
}
```

Теперь давайте вернемся в наш класс `DataStructuresExamples` и добавим следующие строки в нижнюю часть метода `main()`:

```
System.out.println("Пример дерева:");

Tree tree = new Tree(47);

tree.insert(48);
tree.insert(20);
tree.insert(15);
tree.insert(26);
tree.insert(18);

tree.traverseFromRoot();
```

По сути, мы создаем новое дерево `Tree` и инстанцируем его так, чтобы оно содержало один узел с целым значением 47. Затем мы вызываем метод `insert()` дерева с пятью новыми целыми числами без определенного порядка. Наконец, мы вызываем метод дерева `traverseFromRoot()`, чтобы отобразить его содержимое.

Выполнение этого кода должно привести к следующему результату:

```
Пример дерева:
15
18
20
26
47
48
```

Построив метод traverse() таким образом, чтобы в первую очередь выбирались левосторонние узлы, мы, по сути, написали операцию бинарной сортировки. Визуальное представление нашего бинарного дерева представлено на рис. 6.5.

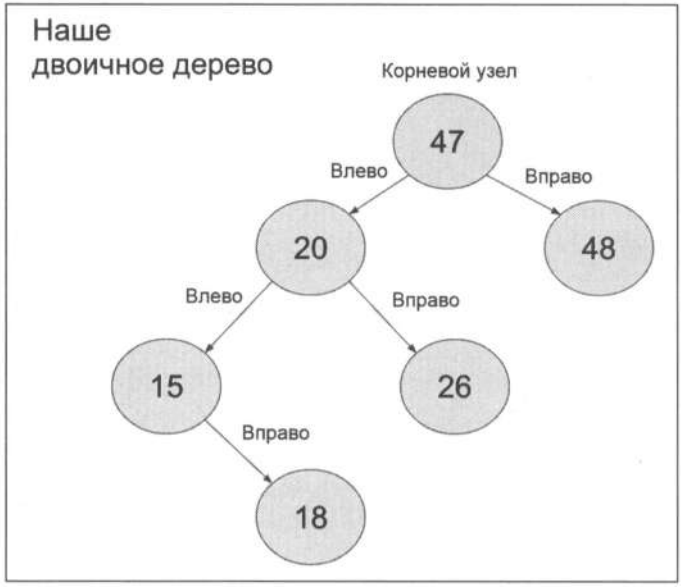



**Рис. 6.5.** Числа, добавленные в нашем примере двоичного дерева, с указанием 47 в качестве корневого узла

Если бы требовалось получить хранящиеся в дереве значения в обратном порядке, можно было бы создать два новых метода в классе Tree. Единственное различие заключается в том, что сначала нам пришлось бы пройти по правой стороне:

```
public void reverseFromRoot() {
    reverse(root);
}

private void reverse(Node node) {
    if (node == null) {
        return;
    }

    reverse(node.getRightNode());
    System.out.println(node.getValue());
    reverse(node.getLeftNode());
}
```

Затем можно добавить две строки кода в конец метода main() класса DataStructureExamples:

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

Запуск кода должен привести к следующему результату:

```
Пример дерева:
15
18
20
26
47
48
48
47
26
20
18
15
```

На этих примерах можно наглядно убедиться, что деревья — это мощные структуры данных, позволяющие быстро хранить и извлекать упорядоченную информацию. Двоичные деревья также широко используются в работе файловых систем и хранилищах электронной почты.

# Заключение

В этой главе мы рассмотрели несколько распространенных структур данных, таких как стеки, очереди, связные списки и двоичные деревья. С помощью рассмотренных в этой главе примеров вы увидели, что эти структуры являются фундаментальными строительными блоками для методологий обработки и хранения данных.

Стоит отметить, что многие компании используют эти структуры и концепции на технических собеседованиях и при решении задач по программированию для потенциальных новых сотрудников. Поэтому понимание концепций, изложенных в этой главе, не только сделает вас лучше как программистов, но и поможет найти работу.

# Важно помнить

- Стеки — это полезные структуры для отслеживания объектов, которые должны быть обработаны в порядке вложенности, в порядке "последний вошел — первый вышел" (Last In, First Out — LIFO). Вызовы методов в стеке программы — хороший пример этого.
- Очереди предназначены для хранения данных, которые должны обрабатываться в порядке поступления (First In, First Out — FIFO).

- Связные списки имеют более быстрое время добавления и меньшие ограничения памяти по сравнению с другими структурами данных.
- Элементы связного списка знают только о *следующем* элементе в списке и могут быть просмотрены только в одну сторону.
- Элементы двусвязного списка знают как о *следующем*, так и о *предыдущем* элементе списка и могут быть просмотрены как вперед, так и назад.
- Узлы двоичного дерева можно быстро перемещать на основе их значений, которые хранятся в связанных узлах слева или справа.

Глава 7

# Работа с базами данных

## Введение

В этой главе мы обсудим, как создавать Java-приложения, использующие базы данных. Сначала будет проведено краткое знакомство с базами данных, включая историю и теорию компьютерных наук, лежащую в их основе. Затем мы поговорим о том, как подключаться к базам данных. Наконец, мы узнаем, как работать с двумя разными типами баз данных, на примере выполнения простых операций создания, чтения, обновления и удаления (Create, Read, Update, Delete — CRUD).

## Структура

Мир баз данных и разработки баз данных довольно велик. Мы будем подходить к этой теме с точки зрения разработчика, обсуждая способы решения задач по созданию Java-приложений с большим объемом данных.

Для этого в данной главе будут рассмотрены следующие темы.

- Введение в базы данных.
- Теорема CAP.
- PostgreSQL.
- Apache Cassandra.
- Выбор правильной базы данных.

## Цели

Эта глава посвящена фундаментальным знаниям о том, как работать с базами данных и думать о них как Java-разработчик. Для этого мы поставим перед собой следующие задачи:

- понять основные различия между реляционной базой данных и базой данных NoSQL;

- узнать, как подключаться к базам данных без жесткого указания учетных данных доступа;
- понять преимущества использования подготовленных операторов;
- научиться выполнять основные запросы и операции;
- понять, что таблицы для реляционной базы данных должны быть смоделированы иначе, чем те же таблицы в базе данных NoSQL.

# Введение в базы данных

Базы данных — это программные приложения, которые обслуживают, организуют и сохраняют данные на диске. База данных предназначена для работы с операционной системой, что позволяет обеспечить наилучшую производительность при чтении и записи. Таким образом, приложение может подключаться к базе данных для быстрого поиска или хранения данных, не разбираясь в тонкостях работы с локальной файловой системой.

Существует несколько типов баз данных и множество видов баз данных. Понимание целей и требований создаваемого приложения очень важно для выбора правильной базы данных. Также полезно знать, как появилась существующая линейка баз данных.

## Краткая история баз данных

Базы данных появились в конце 1960-х годов, когда стала очевидной необходимость в хранении данных. До этого разработчики просто записывали данные в файлы, которые управлялись вызывающим приложением и сильно различались по формату. В это время возникли два основных типа *систем управления базами данных (СУБД)*: иерархические и сетевые базы данных.

Однако такой подход к хранению данных был сопряжен со множеством проблем. Было сложно поддерживать целостность и согласованность часто используемых точек данных. Отношения между данными также были проблематичны, особенно в иерархической модели.

В 1970 году доктор Эдгар Ф. Кодд (Edgar F. Codd), в то время консультант IBM, написал работу[1] под названием "A Relational Model for Large, Shared Data Banks". Эта работа считается изобретением реляционной системы управления базами данных (РСУБД). Вскоре после этого (*Oracle 2004*) IBM выпустила язык структурированных английских запросов (Structured English Query Language, SEQUEL) для работы с созданной Коддом моделью. Позже этот язык был переименован в язык структурированных запросов (Structured Query Language, SQL).

[1] Опубликована в журнале "Communications of The ACM". — *Прим. ред.*

В 1979 году компания Relational Software, Inc. выпустила свой флагманский продукт *Oracle V2*, первую коммерчески доступную СУБД. Появились и другие продукты для реляционных баз данных, а в начале 1990-х годов свое место на предприятиях заняли такие базы данных, как Sybase, FoxBase и Microsoft SQL Server.

До начала 2000-х годов реляционные базы данных хорошо справлялись с поставленными перед ними задачами. С внезапным развитием интернета вскоре стало очевидно, что базы данных, созданные для работы в пределах одной машины, с трудом справляются с требованиями масштабных интернет-приложений.

Именно в это время исследователи вдохновились недавно вышедшей работой[2] доктора Эрика Брюера (Eric Brewer) и доктора Армандо Фокса (Armando Fox) под названием "Harvest, Yield, and Scalable Tolerant Systems". В этой работе Брюер и Фокс постулировали (*Brewer, Fox 1999*), что все распределенные системы пытаются достичь трех свойств: согласованности, доступности и устойчивости к сетевым разделениям. Однако далее они заявили, что реально можно достичь только двух из этих трех свойств и что архитекторы распределенных систем должны рассмотреть возможность отказа от одного из них для достижения своих целей. Эта идея стала известна как *теорема CAP*. Более того, идеи, изложенные Брюером и Фоксом в этой статье, считаются началом распространения распределенных или нереляционных баз данных.

В конце 2000-х — начале 2010-х годов появилось несколько новых продуктов для распределенных баз данных. Эти базы данных использовали иной подход к решению проблем, чем традиционные РСУБД. Когда РСУБД требовалось больше ресурсов, единственным способом было увеличить их в пределах одной системы, добавив больше дисков, оперативной памяти или более быстрый процессор. Этот процесс известен как *вертикальное масштабирование*. Однако вертикальное масштабирование имеет свои пределы, поскольку одна система может обслуживать или поддерживать только определенное количество оперативной памяти, дисков или даже процессоров без замены материнской платы.

Новые базы данных стали известны как базы данных *Not Only SQL*, *NoSQL* (Не только SQL). Базы данных NoSQL воплотили в себе идею архитектуры для обеспечения устойчивости к разделениям (а затем либо высокой доступности, либо высокой согласованности), позволяя одной базе данных существовать на нескольких физических машинах. Когда базе данных не хватало ресурсов, база данных NoSQL могла быстро использовать дополнительные ресурсы другой вновь добавленной машины. Этот процесс известен как *горизонтальное масштабирование*. Он стал еще более быстрым с появлением облачных сервисов в 2010-х годах, что позволило добавлять еще один экземпляр машины к кластеру баз данных NoSQL всего несколькими щелчками мыши.

Сегодня широко используются как базы данных NoSQL, так и базы данных РСУБД. Крупные предприятия, как правило, используют несколько продуктов баз данных,

[2] Опубликована в сборнике Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS 99). — *Прим. ред.*

что позволяет разработчикам приложений выбирать подходящий инструмент для решения конкретной задачи.

# Теорема CAP

В предыдущем разделе мы кратко упомянули теорему CAP. Напомним, что она гласит, что любая сетевая система с общими данными может иметь только два из трех желаемых свойств. Данная теорема является ключевой для понимания того, как устроены современные распределенные системы.

Для начала давайте определим каждое из свойств CAP, чтобы четко представлять себе операционные цели определенных баз данных.

## Согласованность

Базы данных NoSQL (и даже первичные/вторичные отказоустойчивые СУБД) обычно хранят несколько копий всех точек данных. Согласованность — это мера способности базы данных:

- проводить обновления и синхронизировать все копии каждой реплики данных;
- предотвращать чтение пользователем старых или неактуальных значений.

Базы данных, которые могут выполнять эти две задачи, считаются *сильно согласованными*.

## Доступность

Всем известно, что производительность кода имеет значение, и программное обеспечение для баз данных не является исключением. Все запросы к базе данных в той или иной степени страдают от задержек, особенно когда несколько реплик данных хранятся на разных физических или логических машинах. Доступность измеряет способность базы данных (*Brewer, Fox 1999*) всегда читать и возвращать данные по запросу. По сути, если мы выполняем запрос и данные существуют, мы должны получить ответ на этот запрос. Базы данных, которые могут это делать, считаются *высокодоступными*.

## Устойчивость к разделениям

Аппаратные сбои могут случаться и случаются. Рассмотрим гипотетического облачного провайдера с центрами обработки данных и вычислительными регионами, расположенными по всему миру. Облачные провайдеры такого масштаба обычно имеют десятки тысяч серверов, работающих для поддержки операций клиентов. Когда организация имеет такой масштаб аппаратного обеспечения, практически

никогда не бывает такого момента, когда 100% всего доступного оборудования работает надежно.

Базы данных, работающие в таких средах, должны учитывать небольшое количество локальных аппаратных сбоев. Базы данных, которые могут успешно обслуживать результаты во время аппаратного сбоя, считаются *устойчивыми к разделению*.

## Обозначения CAP

Давайте предложим визуализацию этой темы. Предположим, что мы поместим каждое из свойств CAP (Consistency, Availability и Partition Tolerance) на углы треугольника (рис. 7.1). В этом случае каждая сторона треугольника представляет собой текущую парадигму для рассматриваемой распределенной системы или базы данных. Это происходит потому, что в любой момент времени можно поддерживать только два из этих свойств.

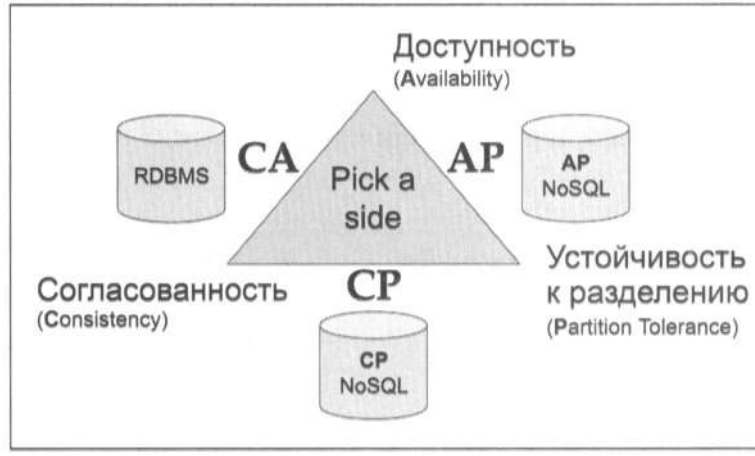



**Рис. 7.1. Визуализация теоремы CAP**

Подобный взгляд на базы данных приводит к появлению следующих операционных парадигм, иногда называемых обозначениями CAP:

- системы AP поддерживают высокую доступность и устойчивость к разделениям. Такие базы данных могут предоставлять результаты с низкой задержкой, распределенные по нескольким машинам (некоторые из них могут находиться в режиме отказа), но согласованными они должны быть в последнюю очередь. Это означает, что запрос может возвращать результаты из обновленных данных, которые могут не соответствовать последнему состоянию. Тем не менее в большинстве баз данных AP задержка между обновлением реплик обычно составляет двузначные миллисекунды;
- системы CP поддерживают устойчивость к разделениям и сильную согласованность. Этот тип распределенных систем обслуживает свои данные таким образом, чтобы предотвратить возврат несовместимых данных реплик. Часто это

достигается с помощью семафоров или других механизмов блокировки. Распределенные системы CP часто поддерживают аппаратный отказ и разделение данных для достижения ограниченной доступности и горизонтального масштабирования;

- системы CA поддерживают как сильную согласованность, так и высокую доступность. Большинство баз данных CA — это СУБД, которые изначально были разработаны для развертывания на отдельных машинах, хотя сейчас большинство СУБД имеют опции для распределенного развертывания.

# PostgreSQL

Теперь, когда мы познакомились с базами данных и некоторыми идеями, на которых они основаны, давайте начнем работать с базой данных.

Мы будем практиковаться на PostgreSQL, широко используемой СУБД.

PostgreSQL, также известная как Postgres, — это реляционная база данных, которая является активно развивающимся проектом с открытым исходным кодом с конца 1980-х годов. Postgres работает на всех основных операционных системах, включая Windows, MacOS, Linux и другие. Текущая основная версия PostgreSQL — 17.2, а версия 18 на момент написания книги находится в бета-версии.

> **Примечание.** Подробнее о проекте базы данных PostgreSQL можно узнать на сайте **https://www.postgresql.org/about/**.

## ElephantSQL

Поскольку основное внимание в этой книге уделяется разработке на Java, а не созданию инфраструктуры баз данных, мы не будем рассматривать процесс загрузки, установки, настройки и запуска Postgres локально. Мы будем использовать облачную базу данных Postgres как сервис (Database as a Service, DBaaS) под названием ElephantSQL. У ElephantSQL есть бесплатный тариф, который должен предоставить нам достаточно ресурсов для выполнения упражнений из этой книги.

Перейдите на сайт **https://www.elephantsql.com/** и создайте учетную запись. В процессе регистрации ElephantSQL попросит создать команду. Название команды можно выбрать любое. На панели инструментов новой команды найдите зеленую кнопку **Create New Instance** в правом верхнем углу (рис. 7.2).

Как и название команды, имя базы данных не имеет технического значения для нашей работы, поэтому его можно задать по своему усмотрению. Обязательно выберите тарифный план **Tiny Turtle**, чтобы не требовалась информация об оплате. Поле **Tags** является необязательным и может быть полезным для команд, использующих несколько баз данных. Нажмите зеленую кнопку **Select Region**, чтобы продолжить.

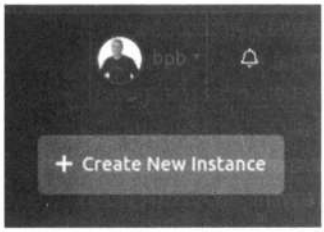



**Рис. 7.2.** Кнопка **Create New Instance** на панели ElephantSQL. В верхней части указано название команды **bpb**

На следующем экране просмотрите доступные облачные регионы и выберите географически близкий регион. Когда все будет готово, нажмите зеленую кнопку **Review**. На последнем экране просмотрите экземпляр базы данных, который будет создан, и нажмите кнопку **Create Instance**, когда будете готовы (рис. 7.3).

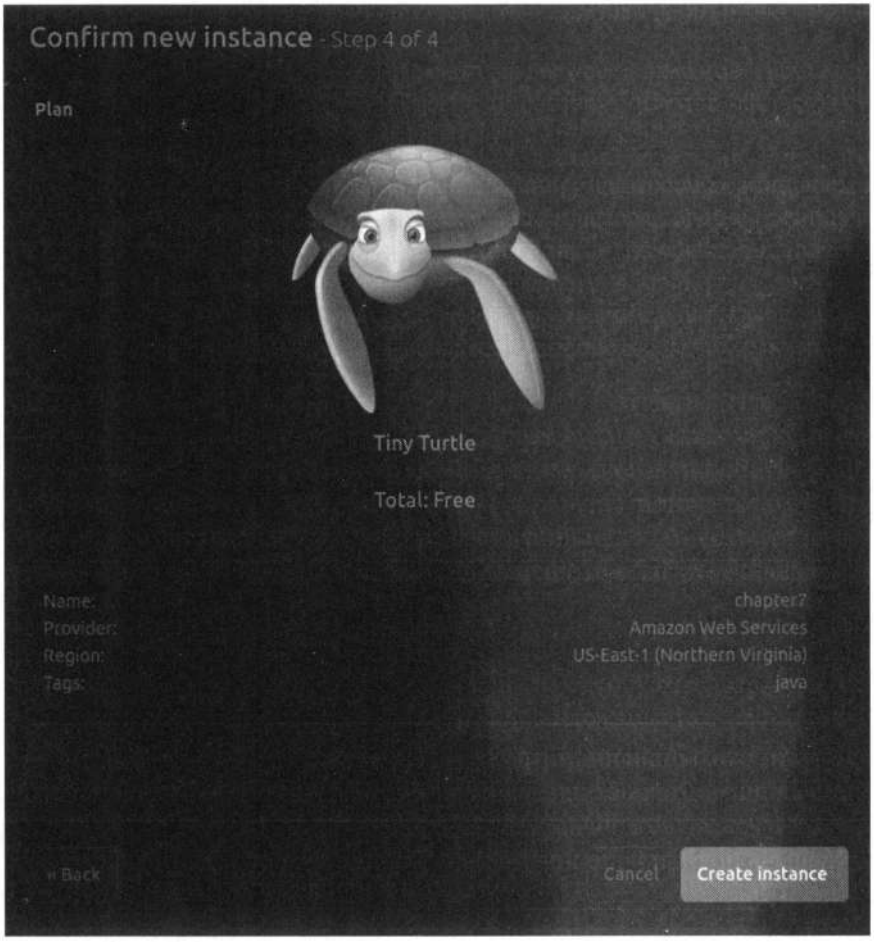



**Рис. 7.3.** Обзор базы данных ElephantSQL, которая будет создана в облаке

**Примечание.** Обязательно запишите имя пользователя и пароль для новой базы данных и сохраните их где-нибудь.

## Схема

Прежде чем начать использовать наш новый экземпляр базы данных, нужно создать *схему базы данных* (schema). Схема базы данных — это множество ограничений, которые указывают базе данных, как организовывать и хранить наши данные. Чтобы определить схему, нам нужно использовать подмножество SQL, известное как *язык определения данных* (Data Definition Language, DDL).

В большинстве баз данных существует понятие таблицы, которая является базовой организационной структурой для хранения данных. Таблицы состоят из строк и столбцов. *Строка* — это небольшая группа данных, которые хранятся и часто запрашиваются совместно. Строки данных в таблице обычно связаны друг с другом. *Столбцы* представляют собой различные свойства или поля, которыми может обладать строка.

Например, таблица с названием product catalog будет содержать строки с различными товарами. Каждый товар будет иметь столбец для таких свойств, как название, описание, цвет и размер.

Мы создадим пять таблиц. Каждая из них будет предназначена для хранения различных данных об астронавтах, которые летали в рамках проекта NASA (Национального управления по аэронавтике и исследованию космического пространства) под названием Gemini (1961–1966 гг.). В наших таблицах будут храниться данные о самих астронавтах, о том, в каких университетах они учились, в какой группе NASA состояли, и о миссиях, в рамках которых летали.

На панели **ElephantSQL** в левой части выберите опцию **BROWSER**. Это вызовет экран **SQL Browser**, где мы будем создавать таблицы, запускать запросы и просматривать данные.

Начнем с таблицы `university`. Введите в текстовое поле SQL Query следующее:

```
CREATE TABLE university (
    id INT PRIMARY KEY,
    name VARCHAR(100)
);
```

Не забудьте нажать синюю кнопку **Execute** справа от текстовой ячейки.

Приведенный выше DDL SQL предписывает Postgres создать новую таблицу со следующими двумя столбцами:

- `id` — целое значение, представляющее собой уникальный идентификатор для таблицы;
- `name` — строка переменной длины, содержащая не более 100 символов и представляющая собой название университета.

Столбец `id` также является нашим *первичным ключом* (primary key). Первичный ключ таблицы — это ограничитель, представляющий собой столбец (или столбцы) и уникально идентифицирующий каждую запись или строку в таблице. При разработке первичного ключа можно использовать два подхода.

- *Естественный ключ* (natural key) — это первичный ключ, который уже является частью таблицы и заведомо уникален.
- *Суррогатный ключ* (surrogate key) — это столбец, который специально создан для того, чтобы быть первичным ключом, и практически не добавляет ценности к реальным хранимым данным. Часто этот столбец представляет собой число.

Столбец `id` в таблице `university` является суррогатным ключом.

> **Примечание.** В PostgreSQL таблицы не обязаны иметь первичный ключ, но он настоятельно рекомендуется.

Далее мы создадим аналогичную таблицу для групп астронавтов NASA с именем `nasa_group`. Удалите предыдущую команду из текстового блока `SQL Query` и замените ее следующей:

```
CREATE TABLE nasa_group (
    id INT PRIMARY KEY,
    year INT
);
```

> **Примечание.** Не забывайте нажимать кнопку **Execute** после ввода каждого оператора `CREATE TABLE` в браузере запросов ElephantSQL.

Теперь давайте создадим таблицу для хранения данных по каждой из миссий астронавтов:

```
CREATE TABLE missions (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    start_date TIMESTAMP,
    end_date TIMESTAMP
);
```

В таблице `missions` используется новый тип данных, с которым мы раньше не сталкивались, — `timestamp` (временная метка). Временная метка — это, по сути, комбинация даты и точного времени. Поскольку у NASA есть точные данные о начале и конце каждой миссии, в наших столбцах `start_date` и `end_date` будет использоваться тип `timestamp` для точного представления времени, о котором идет речь.

Теперь создадим таблицу `astronauts` для хранения данных о самих астронавтах:

```
CREATE TABLE astronauts (
    name VARCHAR(100) PRIMARY KEY,
    nasa_group_id INT,
    dob DATE,
    birthplace VARCHAR(50),
```

```
    university_id INT,
    FOREIGN KEY(nasa_group_id) REFERENCES nasa_group(id),
    FOREIGN KEY(university_id) REFERENCES university(id)
);
```

Давайте пройдемся по структуре таблицы `astronauts`.

- `Name`

  Полное имя астронавта имеет максимум 100 символов и является основным ключом таблицы. Это естественный ключ, поскольку столбец имени является необходимой частью данных таблицы.

- `nasa_group_id`

  Целое число, представляющее группу NASA, в которую был набран астронавт.

- `dob`

  Дата рождения астронавта (date of birth, DOB). Здесь используется тип даты `date`, так как нам не нужен компонент времени `time`.

- `birthplace` (место рождения)

  50-символьная строка с указанием места рождения астронавта.

- `university_id`

  Целое число, представляющее собой суррогатный ключ в таблице университетов `university`, предназначенный для указания последнего университета, в котором учился астронавт.

- `FOREIGN KEY`

  В эту таблицу были добавлены ограничения, заставляющие столбцы `nasa_group_id` и `university_id` ссылаться на другие таблицы (`nasa_group` и `university` соответственно). Если строка, записанная в таблицу `astronauts`, не содержит валидного идентификатора в этих таблицах, будет выдана ошибка.

Ограничения внешнего ключа в таблице `astronauts` помогают нам поддерживать реляционную целостность. Поскольку данные о каждом астронавте распределены по всем этим таблицам, они связаны между собой.

Так как многие астронавты учились в одних и тех же университетах, данные об университете хранятся в его таблице. Это помогает устранить избыточность данных и повысить их целостность. Ведь при хранении данных о каждом астронавте существует большая вероятность неправильного написания названия университета. Размещение университетов в своей таблице значительно снижает вероятность ошибки. Аналогичным образом, если исправить написание названия университета один раз, оно будет исправлено для всех астронавтов, которые на него ссылаются.

НАСА набирает астронавтов в группы. Многие астронавты, которые летали вместе в рамках проекта Gemini, были набраны вместе. В одной группе несколько астронавтов. Поэтому таблица `nasa_group` имеет отношение *"многие-к-одному"* с таблицей `astronauts`. Аналогично, таблица `university` также имеет отношения *"многие-к-одному"* с таблицей `astronauts`.

С миссиями дело обстоит немного иначе. Один астронавт может совершить несколько (много) полетов. Однако в капсуле Gemini находилось два астронавта. В более поздних миссиях "Аполлон" участвовало три астронавта. Поэтому таблица `missions` имеет отношение *"многие-ко-многим"* с таблицей `astronauts`.

Отношения "многие-ко-многим" очень сложны. Один из приемов реляционного моделирования данных — создание промежуточной таблицы между двумя таблицами. *Таблица-мост* (bridge table) имеет только два столбца, и оба они являются частью первичного ключа. Ее задача — абстрагировать отношения "многие-ко-многим", спрятав под ними два отношения "многие-к-одному". Наша последняя таблица — это таблица-мост:

```
CREATE TABLE astronaut_missions (
    astronaut_name VARCHAR(100),
    mission_id INT,
    PRIMARY KEY (astronaut_name, mission_id)
);
```

В итоге в этой таблице будут храниться уникальные пары `astronaut_name` и `mission_id`. Подробнее о таблице-мосте мы поговорим, когда будем делать к ней запрос.

## Нормализация

Процесс создания схемы базы данных с таблицами на основе их сущностей данных известен как нормализация. Схему базы данных можно оценить по уровню нормализации, который называется *нормальной формой*. Вот наиболее распространенные уровни нормализации.

- Первая нормальная форма (1NF). Устранение избыточных данных.
- Вторая нормальная форма (2NF). Схема в 1NF, плюс устранены все частичные зависимости. Это означает, что все столбцы в каждой таблице зависят от своих первичных ключей.
- Третья нормальная форма (3NF). Схема находится в 2NF, а кроме того были устранены все транзитивные зависимости. Это означает, что ни одно из значений столбцов в любой таблице не зависит от столбцов, не являющихся ключами.

Наша схема близка к третьей нормальной форме. Чтобы полностью привести ее в 3NF, необходимо создать суррогатный ключ `astronaut_id` в таблице `astronauts` и использовать его в качестве единственного столбца — первичного ключа вместо столбца `name`. Сейчас, если бы мы захотели обновить имена астронавтов, нам пришлось бы обновлять обе таблицы — `astronauts` и `astronaut_missions`. Однако наша схема имеет бóльшую ценность для обучения в том виде, в котором она есть, с примером естественного ключа и обсуждением того, как ее можно улучшить.

Хотя нормализация схемы к 3NF остается общепринятой практикой, она не всегда соблюдается. Администраторы реляционных систем управления базами данных

(database administrators, DBA) со временем поняли, что устранение операций JOIN, дублирование или денормализация частей схемы может привести к увеличению производительности базы данных.

Нормализация и базовая теория реляционных баз данных были разработаны в те времена, когда дисковое пространство стоило очень дорого, поэтому она также рассматривалась как оптимизация для экономии средств. При построении схемы важно помнить об этом, хотя относительная стоимость дискового пространства сегодня не так высока, как в прошлые годы.

## Загрузка данных

Теперь нам предстоит загрузить данные в наши таблицы. Для этого воспользуемся оператором INSERT из языка манипулирования данными (Data Manipulation Language, DML). Для начала давайте поместим три строки в таблицу nasa_group:

```
INSERT INTO nasa_group (id, year) VALUES (1, 1959);
INSERT INTO nasa_group (id, year) VALUES (2, 1962);
INSERT INTO nasa_group (id, year) VALUES (3, 1963);
```

Как показано выше, в операторе INSERT используется ключевое слово INTO для указания имени таблицы. Затем перечисляются столбцы, для которых необходимо указать значения, после чего следует ключевое слово VALUES с параметризованным списком значений для каждой строки. Можно сразу ввести все три оператора INSERT в текстовый блок SQL Browser (они должны быть разделены точками с запятой), а затем нажать кнопку **Execute**.

То же самое можно проделать и с таблицей university:

```
INSERT INTO university (id, name) VALUES (1,'US Military Academy');
INSERT INTO university (id, name) VALUES (2,'Purdue University');
INSERT INTO university (id, name) VALUES (3,'Princeton University');
INSERT INTO university (id, name) VALUES (4,'Air Force Institute of Technology');
INSERT INTO university (id, name) VALUES (5,'University of Washington');
INSERT INTO university (id, name) VALUES (6,'US Naval Academy');
INSERT INTO university (id, name) VALUES (7,'University of Michigan');
INSERT INTO university (id, name) VALUES (8,'Georgia Institute of Technology');
```

То же самое можно проделать для загрузки данных в таблицу missions. Приведем операторы INSERT для первых трех миссий Gemini:

```
INSERT INTO missions (id, name, start_date, end_date)
VALUES (1, 'Gemini 3', '1965-03-23 14:24:00', '1965-03-23 17:16:31');
INSERT INTO missions (id, name, start_date, end_date)
VALUES (2, 'Gemini 4', '1965-06-03 15:15:59','1965-06-07 17:12:11');
INSERT INTO missions (id, name, start_date, end_date)
VALUES (3, 'Gemini 5', '1965-08-21 13:59:59', '1965-08-29 12:55:13');
```

**Примечание.** Совершенно нормально, если операторы SQL занимают несколько строк, при условии, что они правильно завершаются точкой с запятой.

Проделайте то же самое с данными по оставшимся миссиям Gemini (а также с дополнительными данными по другим проектам NASA, включая Mercury, Apollo и прочие):

```
INSERT INTO astronauts (name, dob, birthplace, nasa_group_id, university_id)
VALUES ('Buzz Aldrin','1930-01-20','Montclair, NJ', 3, 1);
INSERT INTO astronauts (name, dob, birthplace, nasa_group_id, university_id)
VALUES ('Neil Armstrong','1930-08-05','Wapakoneta, OH', 2, 2);
INSERT INTO astronauts (name, dob, birthplace, nasa_group_id, university_id)
VALUES ('Frank Borman','1928-03-14','Gary, IN', 2, 1);
INSERT INTO astronaut_missions(astronaut_name, mission_id)
VALUES ('Buzz Aldrin',10);
INSERT INTO astronaut_missions(astronaut_name, mission_id)
VALUES ('Buzz Aldrin',14);
INSERT INTO astronaut_missions(astronaut_name, mission_id)
VALUES ('Neil Armstrong',6);
INSERT INTO astronaut_missions(astronaut_name, mission_id)
VALUES ('Neil Armstrong',14);
```

## Запрос данных

Если необходимо просмотреть наши данные, то можно воспользоваться оператором SQL `SELECT`:

```
SELECT * FROM university;
```

В операторе `SELECT` можно указать список имен отдельных столбцов, разделенных запятыми, которые мы хотим просмотреть. Или же можно указать звездочку (как показано выше). Ключевое слово `FROM` используется вместе с `SELECT` для указания базы данных и таблицы, к которой выполняется запрос. В нашем случае базой данных по умолчанию является `public`, поэтому мы укажем имя таблицы `university`. Этот запрос должен вернуть следующие данные:

```
id name
1 US Military Academy
2 Purdue University
3 Princeton University
4 Air Force Institute of Technology
5 University of Washington
6 US Naval Academy
7 University of Michigan
8 Georgia Institute of Technology
```

SQL также позволяет ограничить количество строк, возвращаемых в результате. Это очень полезно, если таблица содержит много строк.

Запрос с предложением LIMIT, равным 5, должен дать следующий результат:

```
SELECT * FROM astronauts LIMIT 5;
```

| name | nasa_group_id | dob | birthplace | university_id |
|---|---|---|---|---|
| Buzz Aldrin | 3 | 1930-01-20 | Montclair, NJ | 1 |
| Neil Armstrong | 2 | 1930-08-05 | Wapakoneta, OH | 2 |
| Frank Borman | 2 | 1928-03-14 | Gary, IN | 1 |
| Gene Cernan | 3 | 1934-03-14 | Chicago, IL | 2 |
| Michael Collins | 3 | 1930-10-31 | Rome, Ita | 1 |

В нашем результате мы видим столбцы с названиями nasa_group_id и university_id. Эти названия столбцов могут что-то значить для нас, но большинство людей не хотят искать дополнительные данные. Чтобы получить данные из таблиц nasa_group и university, мы можем выполнить JOIN:

```
SELECT a.name, a.nasa_group_id AS group, a.dob, a.birthplace, u.name as university
FROM astronauts a
INNER JOIN university u ON u.id = a.university_id
LIMIT 5;
```

| name | group | dob | birthplace | university |
|---|---|---|---|---|
| Buzz Aldrin | 3 | 1930-01-20 | Montclair, NJ | US Military Academy |
| Neil Armstrong | 2 | 1930-08-05 | Wapakoneta, OH | Purdue University |
| Frank Borman | 2 | 1928-03-14 | Gary, IN | US Military Academy |
| Gene Cernan | 3 | 1934-03-14 | Chicago, IL | Purdue University |
| Michael Collins | 3 | 1930-10-31 | Rome, Italy | US Military Academy |

При выполнении операции JOIN с SELECT всегда полезно явно называть возвращаемые столбцы (а не использовать звездочку), поскольку это может привести к путанице с аналогично названными столбцами. Как показано выше, таблицы, указанные в операторах FROM и JOIN, можно обозначить псевдонимом (здесь a — астронавт, u — университет).

Также полезно использовать условие WHERE для фильтрации возвращаемых данных. С помощью этого запроса можно вернуть всех астронавтов Gemini, которые учились в US Naval Academy:

```
SELECT a.name, g.year, a.dob, a.birthplace, u.name as university
FROM astronauts a
INNER JOIN university u ON u.id = a.university_id
INNER JOIN nasa_group g ON g.id = a.nasa_group_id
WHERE u.name = 'US Naval Academy';
```

| name | year | dob | birthplace | university |
|---|---|---|---|---|
| Jim Lovell | 1962 | 1928-03-25 | Cleveland, OH | US Naval Academy |
| Wally Schirra | 1959 | 1923-03-12 | Hackensack, NJ | US Naval Academy |
| Tom Staffor | 1962 | 1930-09-17 | Weatherford, OK | US Naval Academy |

В этом запросе мы также добавили еще один JOIN, чтобы получить столбец nasa_group year вместо id.

> **Примечание.** Выполнение запроса SELECT без условия WHERE может вызвать серьезные проблемы с производительностью в больших реляционных базах данных. Всегда полезно использовать WHERE или LIMIT, чтобы ограничить объем возвращаемых данных.

## Доступ из Java

После всего этого можно вернуться к нашей среде разработки Java и поработать над доступом к этим данным из Java-кода.

pom.xml

Откройте файл pom.xml нашего проекта и добавьте следующий код в раздел зависимостей dependencies:

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.6.0</version>
</dependency>
```

Это позволит Maven получить доступ к драйверу PostgreSQL Java Database Connectivity (JDBC). Мы будем использовать его для подключения к нашей базе данных.

### Класс *PostgresConn*

Мы также создадим специальный класс для обработки подключений к нашей базе данных PostgreSQL. Создайте новый класс Java. Убедитесь, что он находится в пакете с именем chapter7 и именем PostgresConn. Этому классу понадобятся три оператора import из библиотеки java.sql, как показано ниже:

```
package chapter7;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class PostgresConn {
    private Connection conn;
```

Внутри определения нашего класса мы также определим новый объект Connection с именем conn.

Сначала создадим новый private-метод connectToPostgres. Он должен принимать три строковых параметра — url, username и password:

```
private void connectToPostgres(String url, String username, String password) {
    try {
        // подключение к базе данных
        conn = DriverManager.getConnection(url, username, password);
    } catch (SQLException e) {
        System.out.println(e.getMessage());
    }
}
```

Внутри нашего метода connectToPostgres() мы устанавливаем объект conn результатом выполнения метода DriverManager.getConnection(). Поскольку метод getConnection() передает нам SQLException для обработки, обернем его внутри оператора try/catch и выведем содержимое сообщения об исключении.

Над методом connectToPostgres() добавим наш конструктор. Мы создадим единственный конструктор, который просто принимает строковые параметры url, username и password, а затем вызывает метод connectToPostgres():

```
public PostgresConn(String url, String username, String password) {
    connectToPostgres(url, username, password);
}
```

Нам также понадобится public-метод для закрытия соединения с базой данных. Всегда полезно правильно закрыть соединение с базой данных, когда завершена работа с ней. Многие серверы баз данных имеют ограничение на количество активных соединений, которыми можно управлять одновременно. Закрытие соединения со стороны приложения позволяет вернуть его в пул соединений базы данных. Метод закрытия соединения довольно прост:

```
public void closePostgresConnection() {
    try {
        conn.close();
    } catch (SQLException e) {
        System.out.println(e.getMessage());
    }
}
```

По сути, мы вызываем метод close() для нашего объекта conn и отлавливаем возможный SQLException:

Наконец, необходимо создать public-геттер для нашего объекта conn:

```
public Connection getConn() {
    return this.conn;
}
```

### Класс *AstronautPostgresDAL*

Создайте еще один новый класс Java. Назовите его `AstronautPostgresDAL` и убедитесь, что он находится внутри пакета `chapter7`. Этот класс будет выступать в качестве *слоя доступа к данным* (data access layer, DAL), позволяя нам абстрагировать код доступа к базе данных от основного программного кода.

> **Примечание.** Идея создания DAL для работы с кодом доступа к базе данных является стандартной практикой. Это не только помогает абстрагировать слой базы данных от бизнес-логики, но и облегчает изменение основной базы данных, если это потребуется в какой-то момент.

Классу `AstronautPostgresDAL` понадобятся операторы `import` для объявления коллекций `List` и `ArrayList`. Также нам понадобится доступ к библиотекам драйвера Postgres для подготовленных операторов, набора результатов, SQL-операторов и возможного исключения SQL:

```
package chapter7;

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import java.util.ArrayList;
import java.util.List;

public class AstronautPostgresDAL {
    private static PostgresConn postgres;

    protected record AstronautMission(
        String missionName, String startDate,
        String endDate, String astronautName) {};
```

Кроме того, мы определим локальный private-объект для нашего соединения с Postgres (ссылающийся на класс, который мы написали) и локальную protected-запись для наших данных `AstronautMission`.

Классу `AstronautPostgresDAL` понадобится простой конструктор. По сути, необходимо убедиться, что наш объект `postgres` (который управляет нашим соединением с базой данных) инициализирован должным образом. Наш конструктор будет принимать строковые параметры `url` базы данных, имя пользователя `username` и пароль `password`. Затем он будет использовать эти параметры для инициализации нашего объекта соединения `PostgresConn`:

```
public AstronautPostgresDAL (String url, String username,
        String password) {
    postgres = new PostgresConn(url, username, password);
}
```

Первая задача — получить полный список астронавтов Gemini. Для этого напишем метод `getGeminiRoster`, который будет возвращать список `List` типа `String`. Начнем с создания версии метода без параметров, которая перегрузит его, вызвав стандартный запрос SQL с ограничением LIMIT, равным `20`:

```
public List<String> getGeminiRoster() {
    return getGeminiRoster(20);
}
```

Затем мы создадим метод, который будет иметь то же имя и возвращаемый тип, но в качестве параметра будет принимать целое число с именем `limit`. Внутри метода определим возвращаемое значение как новый `ArrayList`. Затем мы сформируем простой SQL-запрос для получения столбца `name` из таблицы `astronauts` с конкатенированным значением `limit` в конце. Этот запрос должен вернуть всех астронавтов Gemini из базы данных.

> **Примечание.** Будьте осторожны, создавая таким образом SQL-запросы в коде. Подобные действия с типами `String` могут привести к вредоносной атаке, известной как SQL Injection Attack, когда злоумышленник вводит `SQL DELETE` (или другой вредоносный код) в веб-форму в надежде стереть или запросить конфиденциальные данные. Всегда лучше встраивать переменные в запрос с помощью подготовленного оператора. Однако для целого числа в `LIMIT` наш риск здесь невелик.

```
public List<String> getGeminiRoster(int limit) {
    List<String> returnVal = new ArrayList<>();

    String astronautSQL = "SELECT name FROM astronauts LIMIT "
        + limit;
```

> **Примечание.** Всегда полезно использовать `WHERE` или `LIMIT`, чтобы ограничить количество возвращаемых данных. При том небольшом объеме данных, который у нас есть, в этом нет необходимости. Однако в интересах формирования хороших привычек мы добавим `LIMIT` в наш запрос.

Далее, внутри оператора `try`, мы определим наш объект `Statement` и назовем его `pgStatement`, вызвав метод `createStatement()` объекта соединения Postgres. Затем мы создадим новый объект `ResultSet` с именем `geminiAstronauts` и установим его равным результату метода `excuteQuery()`, передав ему наш `astronautSQL`:

```
    try {
        Statement pgStatement = postgres.getConn().createStatement();
        ResultSet geminiAstronauts =
            pgStatement.executeQuery(astronautSQL);

        while (geminiAstronauts.next()) {
            returnVal.add(geminiAstronauts.getString("name"));
        }
```

```
    } catch (SQLException e) {
        System.out.println(e.getMessage());
    }
    return returnVal;
}
```

Выполнив запрос и сохранив результаты в объекте `geminiAstronauts`, можно просмотреть их в цикле `while`, который будет выполняться до тех пор, пока метод `next()` нашего набора результатов является истинным (`true`). Внутри цикла `while` мы получим значение столбца имени астронавта для каждой строки и добавим его в наш список `returnVal`.

Вы, наверное, заметили, что для большинства наших методов базы данных Postgres требуется `try/catch`. Это связано с тем, что они передают нам `SQLException` для обработки. Как только мы закрываем блок `try/catch`, выводя сообщение об исключении, можно вернуть список `returnVal` вызывающему методу.

### Класс *GeminiAstronautsRDBMS*

Далее создайте новый Java-класс и назовите его `GeminiAstronautsRDBMS`. Убедитесь, что он находится в пакете `chapter7` и имеет метод `main`. Нашему классу `GeminiAstronautsRDBMS` понадобятся импорты `Set`, `HashSet` и `Random` из библиотеки `java.util`:

```
package chapter7;

import java.util.HashSet;
import java.util.Set;
import java.util.Random;

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

Внутри нашего метода `main` мы обратимся к переменным окружения для получения учетных данных базы данных. Это необходимо, потому что учетные данные доступа к базе данных никогда не должны находиться внутри кода. Поэтому установим набор наших учетных данных в переменных окружения операционной системы (позже) и обратимся к ним, вызвав метод `System.getenv()`:

```
String url = System.getenv("POSTGRES_URL");
String username = System.getenv("POSTGRES_USER");
String password = System.getenv("POSTGRES_PASSWORD");

AstronautPostgresDAL astronautDAL =
    new AstronautPostgresDAL(url, username, password);

System.out.println("Астронавты проекта Gemini:");
```

Затем можно использовать вновь созданные переменные для инстанциирования (создания объекта) нашего слоя доступа к данным `AstronautPostgresDAL`. Далее мы вызовем метод `getGeminiRoster()` на нашем DAL, выполним итерацию по результатам с помощью цикла `for` и выведем на экран имена отдельных астронавтов:

```
List<String> geminiAstronauts = astronautDAL.getGeminiRoster();

for (String astronaut : geminiAstronauts) {
    System.out.println(astronaut);
}

System.out.println();
```

Попробуйте запустить этот код. Он должен завершиться неуспешно с примерно таким сообщением:

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"java.sql.Connection.createStatement()" because the return value of
"chapter7.PostgresConn.getConn()" is null
    at chapter7.GeminiAstronautsRDBMS.main(GeminiAstronautsRDBMS.java:29)
```

Это исключение происходит потому, что мы не определили учетные данные нашей базы данных в среде.

В Eclipse IDE запуск нашего класса `GeminiAstronautsRDBMS` должен был создать новую конфигурацию запуска. Просмотреть доступные конфигурации запуска можно, выбрав **Run Configurations** из меню **Run** в Eclipse. Это должно выглядеть примерно так, как показано на рис. 7.4.

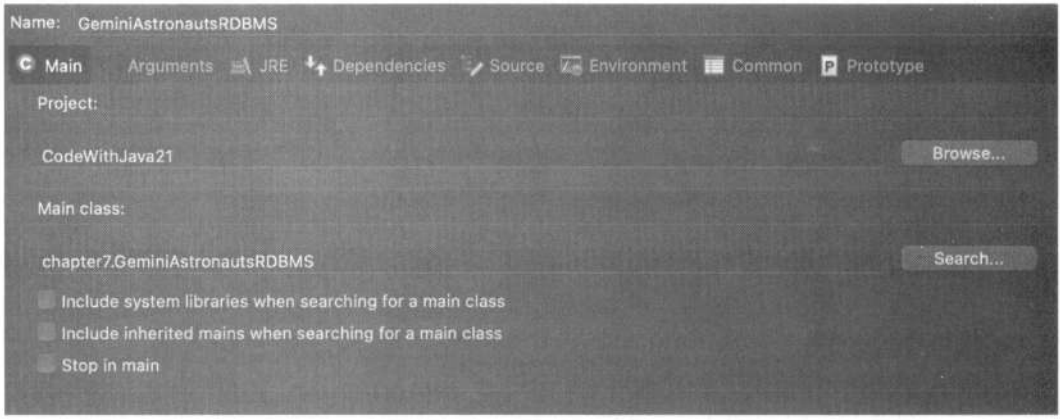



**Рис. 7.4.** Вкладка **Main** конфигурации запуска `GeminiAstronautsRDBMS`

Если конфигурации запуска не существует, то ее нужно создать. Далее перейдите на вкладку **Environment**. Здесь добавим три переменные окружения. Информация, необходимая для определения этих переменных, находится на странице **Details** в ElephantSQL для нашего экземпляра базы данных, как показано на рис. 7.5.

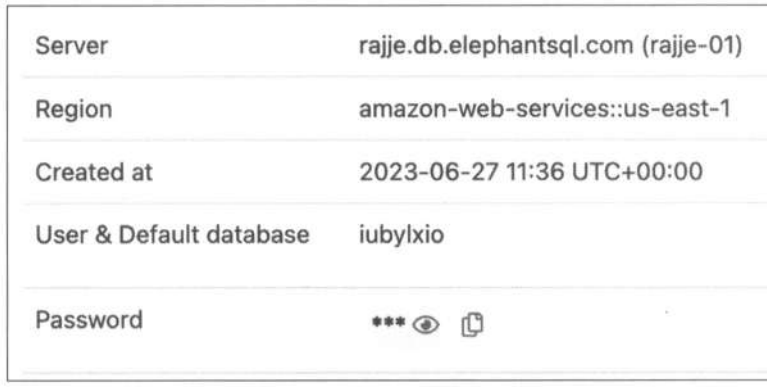



**Рис. 7.5.** Информация о нашей базе данных на странице **Details** в ElephantSQL

Каждый отдельный экземпляр базы данных будет иметь свой собственный уникальный сервер, имя пользователя и пароль. Следующие переменные, которые мы будем создавать, основаны на показанном на рис. 7.5 примере, поэтому не забудьте внести соответствующие изменения:

```
POSTGRES_URL: jdbc:postgresql://rajje.db.elephantsql.com/iubylxio
POSTGRES_USER: iublxio
POSTGRES_PASSWORD: QYKQfZNgDFShS_EY8lcpAh1yZoi6nbA0
```

Хотя **Password** скрыт *звездочками* [•••] (см. рис. 7.5), его можно отобразить, щелкнув на значке *глаза* [👁], и скопировать, щелкнув на значке *копирования* [📋].

Если бы мы запускали наш Java-код из командной строки, то пришлось бы задать эти переменные на уровне ОС. Реализация зависит от базовой ОС.

**Linux/MacOS**

```
export POSTGRES_PASSWORD=QYKQfZNgDFShS_EY8lcpAh1yZoi6nbA0
export
POSTGRES_URL=jdbc:postgresql://rajje.db.elephantsql.com/iubylxio
export POSTGRES_USER=iubylxio
```

**Windows**

```
set POSTGRES_PASSWORD="QYKQfZNgDFShS_EY8lcpAh1yZoi6nbA0"
set POSTGRES_URL="jdbc:postgresql://rajje.db.elephantsql.com/iubylxio"
set POSTGRES_USER="iubylxio"
```

> **Примечание.** Демонстрация определения переменных окружения в командной строке приведена в качестве примера. Если мы определим их в конфигурации выполнения в нашей IDE, нам не нужно будет этого делать.

С этими переменными, определенными в конфигурации выполнения, запуск приведенного выше кода должен привести к следующему результату:

```
Астронавты проекта Gemini:
Buzz Aldrin
```

```
Neil Armstrong
Frank Borman
Gene Cernan
Michael Collins
Pete Conrad
Gordon Cooper
Richard Gordon
Gus Grissom
Jim Lovell
Jim McDivitt
Wally Schirra
Dave Scott
Tom Stafford
Ed White
John Young
```

Как видите, мы успешно выполнили запрос по 16 астронавтам проекта Gemini.

Далее приведем результаты более сложного запроса. Мы напишем код для возврата случайных данных по трем миссиям Gemini вместе с астронавтами, которые в них участвовали.

## Пересмотр класса *AstronautPostgresDAL*

Прежде чем продолжить, давайте вернемся к нашему классу AstronautPostgresDAL. Здесь мы добавим новый public-метод типа List (нашей записи AstronautMission) и назовем его getMissionAstronauts. Метод должен принимать единственный параметр типа String с именем missionName. Начнем мы этот метод с определения возвращаемого значения, которое представляет собой список AstronautMission с именем returnVal:

```
public List<AstronautMission> getMissionAstronauts(String missionName) {
    List<AstronautMission> returnVal = new ArrayList<>();
```

Теперь давайте построим запрос. Как уже обсуждалось ранее в этой главе, наши таблицы астронавтов и миссий имеют отношения "многие-ко-многим". Чтобы должным образом обработать эти отношения, мы создали промежуточную таблицу astronaut_missions. Чтобы запросить данные об астронавтах и их миссиях, нам нужно выполнить JOIN на обеих таблицах astronaut_missions и missions. Взгляните на следующий код:

```
String missionSQL = "SELECT m.name AS missionname, m.start_date, "
    + "m.end_date, a.name "
    + "FROM astronauts a "
    + "INNER JOIN astronaut_missions am ON am.astronaut_name = a.name "
    + "INNER JOIN missions m ON m.id = am.mission_id "
    + "WHERE m.name = ?;";
```

Мы установим результат выполнения этого запроса в качестве значения строковой переменной с именем `missionSQL`. Обратите внимание на условие `WHERE` в приведенном выше запросе. Оно будет фильтровать данные на основе названия миссии, представленного вопросительным знаком (`?`). Чтобы динамически отправлять этому запросу название новой миссии во время выполнения, мы используем *подготовленный оператор*.

> **Примечание.** Столбцы с одинаковыми именами в разных таблицах, возвращаемые в одном и том же наборе результатов, могут сделать объект `ResultSet` запутанным. Уникальность можно обеспечить с помощью ключевого слова `AS`, чтобы создать псевдоним для этого столбца внутри набора результатов.

Подготовленные операторы — это объекты запросов к базе данных, которые позволяют нам выполнять один и тот же запрос несколько раз с разными параметрами. Подготовленные операторы работают лучше, поскольку их SQL нужно разобрать только один раз, а выполнять можно много раз.

Далее мы создадим новый подготовленный оператор с именем `missionStatement`. Установим его равным результату выполнения метода соединения `prepareStatement()` и передадим в качестве параметра наш запрос `missionSQL`:

```
try {
    PreparedStatement missionStatement =
        postgres.getConn().prepareStatement(missionSQL);
```

Подготовив оператор, воспользуемся методом `setString()` нашего `missionStatement`, чтобы привязать к нему значение названия нашей миссии (в порядковой позиции `1`).

> **Примечание.** Порядковые позиции переменных для подготовленных операторов начинаются с единиц, а не с нулей, как в остальной части Java. Поэтому порядковая позиция `1` представляет собой первый вопросительный знак, найденный в подготовленном операторе.

Выполним наш запрос, вызвав метод `executeQuery()` объекта `missionStatement`, и установим его результат в новый объект `ResultSet` с именем `missionAstronauts`

```
missionStatement.setString(1, missionName);
ResultSet missionAstronauts = missionStatement.executeQuery();
```

На данном этапе мы выполнили связанный с `missionName` запрос и сохранили результаты в наборе результатов `missionAstronauts`. Теперь можно просмотреть набор результатов с помощью цикла `while`, создать запись `AstronautMission` с именем миссии, диапазоном дат и астронавтом, и добавить эту запись в список `returnVal`. Взгляните на следующий код:

```
while (missionAstronauts.next()) {
    AstronautMission astronautMission = new AstronautMission(
        missionAstronauts.getString("missionname"),
        missionAstronauts.getString("start_date"),
        missionAstronauts.getString("end_date"),
```

```
            missionAstronauts.getString("name"));
        returnVal.add(astronautMission);
    }
```

После этого можно завершить выполнение try/catch, отловив при этом SQLException. Наконец, можно вернуть returnVal и завершить работу метода. Взгляните на следующий код:

```
        } catch (SQLException e) {
            e.printStackTrace();
        }

        return returnVal;
    }
```

## Пересмотр класса *GeminiAstronautsRDBMS*

Вернувшись в класс GeminiAstronautsRDBMS, в методе main() после последнего блока кода создадим множество Set целых чисел, а также объект Random:

```
Set<Integer> randomMissions = new HashSet<>();
Random random = new Random();
// генерация трех случайных чисел
while (randomMissions.size() < 3) {
    int missionNumber = random.nextInt(10) + 3;
    randomMissions.add(missionNumber);
}
```

Определив объекты randomMissions и random, начнем выполнять цикл while до тех пор, пока множество randomMissions будет содержать менее трех элементов. Внутри цикла while сгенерируем случайное число от 3 до 12 (включительно), представляющее пилотируемые полеты Gemini в космос. Затем добавим сгенерированное число в множество randomMissions.

> **Примечание.** Помните, что значения во множествах Set уникальны. Поэтому нам не нужно беспокоиться о том, что одно и то же число может быть сгенерировано дважды, поскольку любые дубликаты при добавлении в randomMissions будут проигнорированы.

Теперь с помощью цикла for переберем все числа, сгенерированные во множестве randomMissions. Затем создадим строку с именем Gemini (с пробелом в конце) и присоединим к ней номер миссии (missionNum). Затем можно вызвать метод getMissionAstronauts(), который мы написали выше, передав mission с помощью стандартного метода toString():

```
for (Integer missionNum : randomMissions) {

    StringBuilder mission = new StringBuilder("Gemini ");
    mission.append(missionNum.toString());
```

```
    List<AstronautMission> missionAstronauts =
        astronautDAL.getMissionAstronauts(mission.toString());

    for (AstronautMission astronautMission : missionAstronauts) {
        System.out.print(astronautMission.missionName() + " ");
        System.out.print(astronautMission.startDate() + " -> ");
        System.out.print(astronautMission.endDate() + " - ");
        System.out.println(astronautMission.astronautName());
    }

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

Можно вложить еще один цикл `for` внутрь существующего цикла, чтобы просмотреть список `missionAstronauts`. Внутри него нужно вывести `missionName`, `startDate`, `endDate` и `astronautName` для каждого астронавта в конкретной миссии Gemini.

Выполнение этого кода должно по-прежнему показывать полный список астронавтов Gemini, а в конце выдать нечто похожее на это:

```
Gemini 3 1965-03-23 14:24:00 -> 1965-03-23 17:16:31 - Gus Grissom
Gemini 3 1965-03-23 14:24:00 -> 1965-03-23 17:16:31 - John Young

Gemini 4 1965-06-03 15:15:59 -> 1965-06-07 17:12:11 - Jim McDivitt
Gemini 4 1965-06-03 15:15:59 -> 1965-06-07 17:12:11 - Ed White

Gemini 9 1966-06-03 13:39:33 -> 1966-06-06 14:00:23 - Gene Cernan
Gemini 9 1966-06-03 13:39:33 -> 1966-06-06 14:00:23 - Tom Stafford
```

Как видите, мы успешно создали схему РСУБД и набор данных, используя базу данных PostgreSQL, и выполнили запрос с помощью Java.

# Apache Cassandra

После обсуждения реляционных баз данных и PostgreSQL в частности мы перейдем к несколько иному взгляду на управление данными. Мы обсудим Apache Cassandra, NoSQL-базу данных, используемую многими крупными предприятиями, работающими с большими данными.

Изначально Cassandra была разработана в Facebook[3] и выпущена в 2008 году в виде открытого исходного кода. После недолгого пребывания в качестве *инкубаторско-*

[3] Экстремистская организация, деятельность которой запрещена на территории РФ. — *Прим. ред.*

*го[4] проекта* в Apache Software Foundation в 2010 году ей был присвоен статус проекта высшего уровня. С тех пор Apache Cassandra широко используется на предприятиях различных отраслей, включая провайдеров потокового видео, сервисы совместных поездов, финансовые учреждения и розничную торговлю. Сегодня одни из самых крупных внедрений Apache Cassandra находятся в таких компаниях, как Apple, Netflix и Uber.

Изучение Apache Cassandra — естественный следующий шаг для книги о программировании на Java, поскольку Cassandra написана на Java. Аналогичным образом, большая часть экосистемы инструментов Cassandra также основана на Java. Знание Java обеспечивает дополнительный уровень понимания при устранении проблем в мире Cassandra.

Текущая версия Apache Cassandra 5.0.

> **Примечание.** Подробнее о проекте базы данных Apache Cassandra можно узнать на сайте **https://cassandra.apache.org/**.

## Astra DB

Как уже говорилось ранее, основное внимание в этой книге уделяется разработке на Java, а не созданию инфраструктуры распределенных баз данных. Мы будем использовать облачную СУБД Cassandra DBaaS под названием Astra DB. Astra DB была разработана компанией DataStax, которая предоставляет корпоративные версии Cassandra с момента ее создания. Astra DB также имеет бесплатный уровень, который должен предоставить нам достаточно ресурсов для выполнения упражнений в этой книге.

Перейдите на сайт **https://astra.datastax.com/** и создайте учетную запись. После входа в систему должна появиться пользовательская панель **Astra**, как показано на рис. 7.6. В левой нижней части экрана нажмите на кнопку **Create a Database**.

> **Примечание.** Новая база данных, созданная в Astra DB, по умолчанию будет иметь коэффициент репликации, равный трем. Это означает, что каждый фрагмент данных будет храниться три раза, и каждая реплика данных будет находиться на отдельном экземпляре машины.

В появившемся окне (рис. 7.7) мы зададим имя нашей базы данных, а также пространство ключей. *Пространство ключей* — это логическая структура в Cassandra, которая помогает отделить наши таблицы от других. Это очень полезно в большом многопользовательском кластере баз данных. Имя базы данных не имеет значения,

---

[4] Apache Incubator — шлюз для проектов с открытым исходным кодом, которые должны стать полноценными проектами Apache Software Foundation. Проект «Инкубатор» был создан в октябре 2002 года для обеспечения доступа к Apache Software Foundation для проектов и баз кода, желающих стать частью усилий Фонда. Все пожертвования кода от внешних организаций и существующих внешних проектов, желающих перейти на Apache, должны поступать через инкубатор. — *Прим. ред.*

но для упражнений в этой книге мы должны сохранить имя пространства ключей как **astronaut_data**.

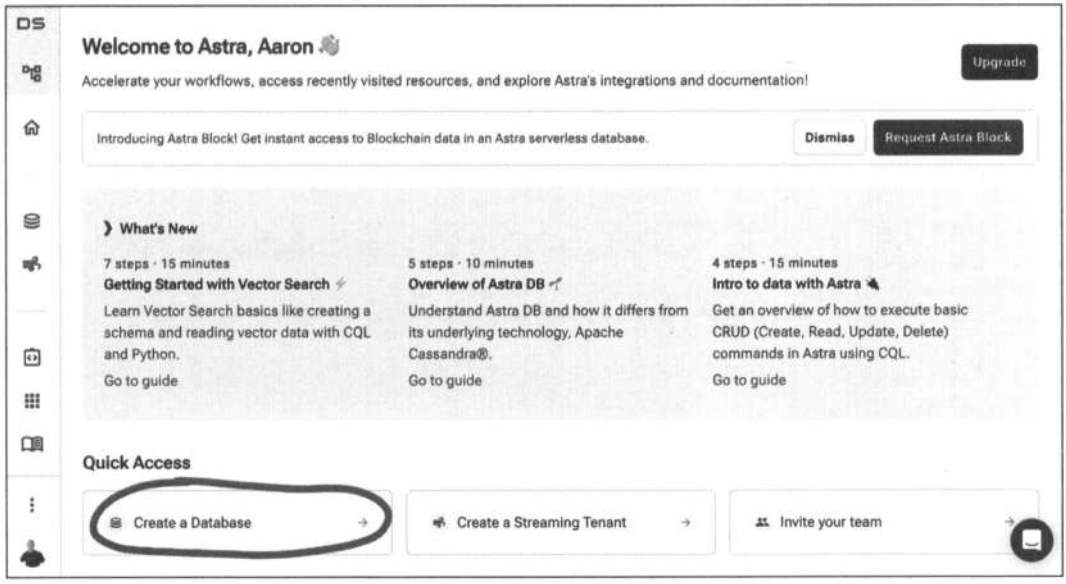



**Рис. 7.6.** Пользовательская панель **Astra** после успешного входа в систему. Кнопка **Create a Database** (обведено)

Чтобы убедиться, что мы можем использовать бесплатный уровень Astra DB, выберите в качестве провайдера **Google Cloud Platform (GCP)**. Не все регионы будут доступны на бесплатном уровне. В раскрывающемся списке **Region** выберите ближайший регион из тех, что выделены жирным шрифтом. Например, тем, кто живет в Европе, следует выбрать регион **St. Ghislain, Belgium** (Сен-Гислен, Бельгия). На рис. 7.7 выбран регион AWS, расположенный в Северной Вирджинии, США (us-east1).

На следующем экране, когда база данных станет активной, должны появиться опции, показанные на рис. 7.8. Нажмите кнопку **Generate Token**, чтобы получить учетные данные для доступа к только что созданной базе данных. На экране есть опции для быстрого сохранения и/или копирования сгенерированных учетных данных. Обязательно сделайте то или другое (или и то, и другое).

Также не забудьте нажать кнопку **Get Bundle**. Astra DB хранит почти все, что нужно для подключения, в предварительно сгенерированных защищенных zip-файлах. Сюда входят такие вещи, как имена хостов, номера портов и сертификаты безопасности транспортного уровня (Transport Layer Security, TLS). Так обеспечивается безопасность данных, передаваемых по сети между приложением и базой данных. Не забудьте переместить защищенный пакет в другое место, но запомните его, так как позже нам понадобится его каталог.

Создав базу данных, перейдем к созданию схемы.

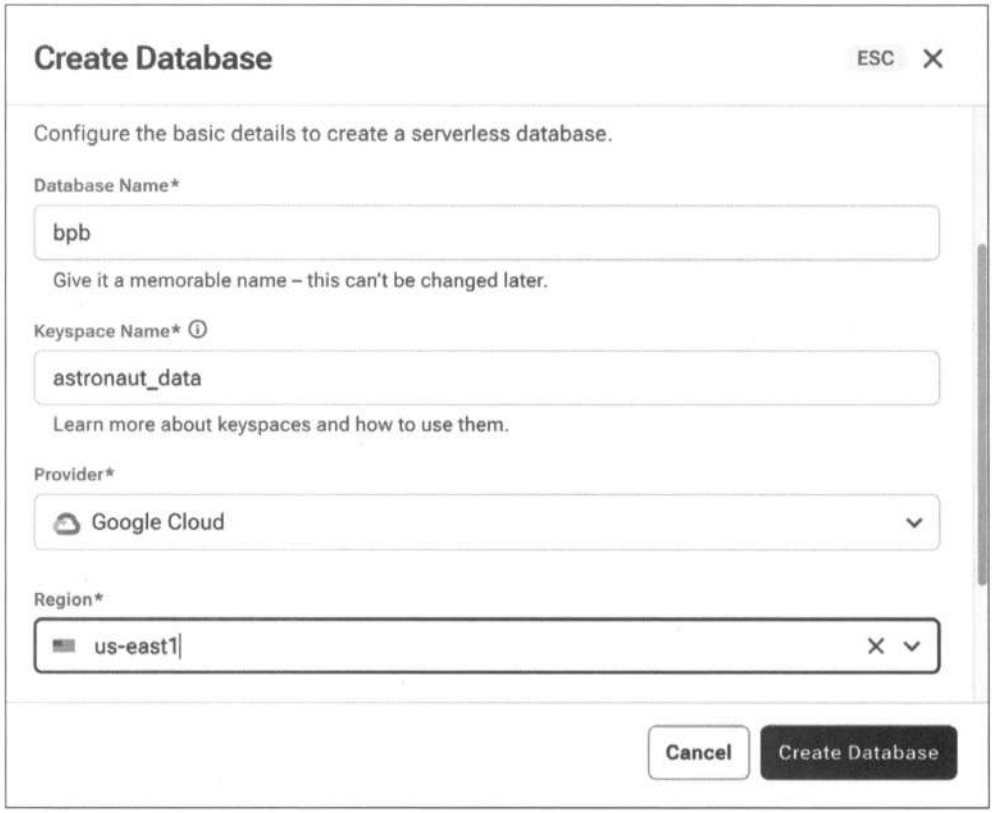



**Рис. 7.7.** Окно **Create Database**, показывающее, как настроить базу данных

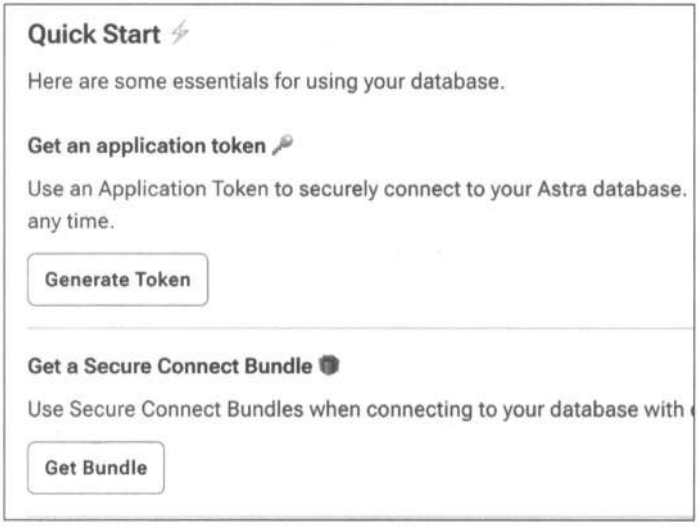



**Рис. 7.8.** Опции генерации токена и защищенного пакета для новой базы данных. Обе эти задачи должны быть выполнены

## Схема

Как и в случае с Postgres, создадим таблицы для хранения данных в Cassandra. Однако в Cassandra мы создадим таблицы для поддержки наших перспективных запросов. Например, нам понадобятся таблицы для поддержки следующих запросов:

- запрос астронавтов по имени;
- запрос астронавтов по группе набора NASA;
- запрос астронавтов по университету, в котором они учились;
- поиск астронавтов по миссиям, в которых они участвовали.

Исходя из наших требований, мы создадим четыре таблицы. Каждая из этих таблиц будет иметь определение первичного ключа, созданное для поддержки условия `WHERE` нашего запроса.

Чтобы построить схему таблицы, сначала нам нужно воспользоваться утилитой CQL shell (`cqlsh`). В Astra DB можно получить доступ к `cqlsh`, сначала щелкнув на сведения о нашей базе данных (рис. 7.9), а затем перейдя на вкладку **CQL Console**. Оказавшись в консоли **CQL**, необходимо указать **Cassandra**, что мы хотим получить доступ к пространству ключей `astronaut_data`. Это можно сделать с помощью команды `use`:

```
use astronaut_data;
```

Теперь можно запускать операторы `CREATE TABLE` (см. рис. 7.9).

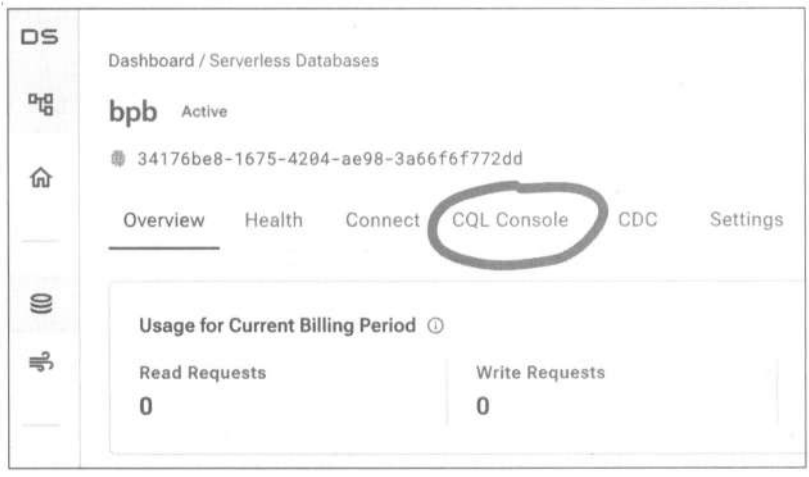



**Рис. 7.9.** Страница сведений о базе данных **bpb** с вкладкой **CQL Console** (обведено)

Первой будет таблица `astronauts`, и у нее будет один первичный ключ — `name`:

```
CREATE TABLE astronauts (
    name TEXT PRIMARY KEY,
    dob DATE,
    birthplace TEXT,
```

```
    nasa_group_id INT,
    nasa_group_year INT,
    university_name TEXT
);
```

Apache Cassandra использует собственный SQL-подобный язык запросов, известный как Cassandra Query Language (CQL). Несмотря на сходство с SQL во многих аспектах, в CQL отсутствуют многие функциональные возможности SQL. Большинство из них обусловлено архитектурными отличиями Cassandra от реляционных баз данных. Другие различия связаны с повторным использованием определенных ключевых слов SQL для обеспечения различной функциональности. Эти различия будут освещаться по мере необходимости.

> **Примечание.** Как правило, важно не делать предположений о том, как аналогичные команды будут работать в разных системах, базах данных и т.п. Как и в случае с любой новой системой, обязательно прочитайте соответствующую документацию по команде или структуре, прежде чем использовать ее в производственной среде.

При рассмотрении приведенного выше кода CQL для определения таблицы `astronauts` можно заметить некоторые различия в типах данных. Вместо `VARCHAR` с заданной длиной в Cassandra используется тип `VARCHAR`, не требующий длины. Тип `VARCHAR` в Cassandra также является псевдонимом типа данных `TEXT`, и оба при извлечении преобразуются в строки Java.

Первичные ключи в Casandra также ведут себя иначе, чем их аналоги в РСУБД. Во-первых, Cassandra требует наличия первичного ключа во всех таблицах. Это связано с тем, что первичный ключ помогает определить, на каких узлах базы данных (экземплярах) должны храниться данные. Без первичного ключа Cassandra не будет знать, где хранить запрашиваемые для записи данные, поэтому он необходим.

Во-вторых, все первичные ключи в Cassandra уникальны. Это может привести к нежелательным последствиям, когда несколько операций записи в один и тот же первичный ключ просто перезапишут друг друга.

Следующая таблица будет поддерживать наш запрос на поиск астронавтов по номеру их группы набора NASA:

```
CREATE TABLE astronauts_by_group (
    nasa_group_id INT,
    nasa_group_year INT STATIC,
    astronaut_name TEXT,
    dob DATE,
    birthplace TEXT,
    university_name TEXT,
    PRIMARY KEY (nasa_group_id, astronaut_name)
) WITH CLUSTERING ORDER BY (astronaut_name ASC);
```

У этой таблицы есть несколько новых особенностей. Во-первых, мы указали *составной первичный ключ* — это означает, что он состоит из нескольких столбцов. Первая часть первичного ключа — это ключ раздела. Этот ключ хэшируется в токен, и все запросы к нему отправляются на узел или узлы, отвечающие за диапазон токенов, в который он попадает.

Вторая часть первичного ключа известна как *ключ кластеризации*. Ключ кластеризации таблицы определяет порядок сортировки на диске внутри раздела. Дополнительная часть ключа кластеризации указывается в условии `WITH`, поскольку порядок кластеризации задается вместе с направлением сортировки (по возрастанию). Если ключ кластеризации указан в определении первичного ключа без выражения `CLUSTERING ORDER BY`, то направление сортировки по умолчанию будет по возрастанию (`ASC`).

> **Примечание.** Поскольку по умолчанию используется восходящий порядок, все определения таблиц, требующие разбиения данных по убыванию, должны содержать предложение `CLUSTERING ORDER BY` с указанием имени столбца и направления сортировки по убыванию (`DESC`).

Наконец, столбец `nasa_group_year` обозначен как столбец `STATIC`. Это ключевое слово указывает Cassandra хранить эти данные на уровне разделов. В противном случае каждая строка в разделе будет хранить свое собственное значение для этого столбца, которое будет одинаковым. В данном случае все астронавты из группы набора 2 будут храниться в одном разделе и иметь одно и то же значение `nasa_group_year` равное 1962. Поэтому нет необходимости хранить одно и то же значение в каждой строке астронавта.

Следующая таблица будет поддерживать запросы к астронавтам по университетам, в которых они учились:

```
CREATE TABLE astronauts_by_university (
    university_name TEXT,
    astronaut_name TEXT,
    nasa_group_id INT,
    nasa_group_year INT,
    dob DATE,
    birthplace TEXT,
    PRIMARY KEY (university_name, astronaut_name)
);
```

Как уже говорилось ранее, определение первичного ключа указывает на то, что мы будем запрашивать по `university_name` и обеспечивать сортировку по возрастанию по `astronaut_name`. Как было показано, использование естественных ключей в Cassandra является стандартной практикой. Это связано с тем, что определение первичного ключа устанавливает, как можно запрашивать таблицу, и большинство пользователей будут лучше понимать `university_name`, чем суррогатный `university_id`. Подход с суррогатным ключом лучше работает в мире РСУБД, потому что суррo-

гатные ключи обычно абстрагируются за кулисами (через JOIN), и пользователям обычно не нужно понимать их или знать о них.

Таблица `astronauts_by_mission` будет построена аналогичным образом:

```
CREATE TABLE astronauts_by_mission (
    mission_name TEXT,
    astronaut_name TEXT,
    mission_start_date TIMESTAMP STATIC,
    mission_end_date TIMESTAMP STATIC,
    university_name TEXT,
    nasa_group_id INT,
    nasa_group_year INT,
    dob DATE,
    birthplace TEXT,
    PRIMARY KEY (mission_name, astronaut_name)
);
```

Основное отличие заключается в том, что дата начала миссии (`mission_start_date`) и дата окончания миссии (`mission_end_date`) определены как `STATIC`. Таким образом, детали миссии хранятся на уровне разделов вместе с `mission_name`.

### Денормализация

В последнем разделе о проектировании схем РСУБД упоминалась идея денормализации и то, как отклонение от нее в правильных сценариях может привести к повышению производительности. В Apache Cassandra и многих других NoSQL-базах данных подход к моделированию данных с помощью денормализации является стандартным. Это связано с тем, что он помогает моделям данных лучше работать в парадигме распределенных баз данных.

Как и в Apache Cassandra, так и во многих других NoSQL-базах данных, идея заключается в том, чтобы построить таблицы так, чтобы они соответствовали нашим запросам. Таким образом, практика денормализации становится стандартным подходом, а не исключением из правил.

## Загрузка данных

Далее мы загрузим данные в наши таблицы. Как и в случае с загрузкой данных в РСУБД, будем использовать оператор `INSERT` языка CQL для записи данных в наши таблицы.

Сначала запишем данные в нашу главную таблицу. Следующий CQL добавит четыре строки в таблицу `astronauts`:

```
INSERT INTO astronauts (name, dob, birthplace, nasa_group_id, nasa_group_year,
university_name)
VALUES ('Buzz Aldrin','1930-01-20','Montclair, NJ', 3, 1963, 'US Military Academy');
```

```
INSERT INTO astronauts (name, dob, birthplace, nasa_group_id, nasa_group_year,
university_name)
VALUES ('Neil Armstrong','1930-08-05','Wapakoneta, OH', 2, 1962, 'Purdue University');
INSERT INTO astronauts (name, dob, birthplace, nasa_group_id, nasa_group_year,
university_name)
VALUES ('Frank Borman','1928-03-14','Gary, IN', 2, 1962, 'US Military Academy');
INSERT INTO astronauts (name, dob, birthplace, nasa_group_id, nasa_group_year,
university_name)
VALUES ('Gene Cernan','1934-03-14','Chicago, IL', 3, 1963, 'Purdue University');
```

Аналогичным образом можно загрузить несколько строк данных в таблицу `astronauts_by_group`:

```
INSERT INTO astronauts_by_group (astronaut_name, dob, birthplace, nasa_group_id,
nasa_group_year, university_name)
VALUES ('Buzz Aldrin','1930-01-20','Montclair, NJ', 3, 1963, 'US Military Academy');
INSERT INTO astronauts_by_group (astronaut_name, dob, birthplace, nasa_group_id,
nasa_group_year, university_name)
VALUES ('Neil Armstrong','1930-08-05','Wapakoneta, OH', 2, 1962, 'Purdue University');
INSERT INTO astronauts_by_group (astronaut_name, dob, birthplace, nasa_group_id,
nasa_group_year, university_name)
VALUES ('Frank Borman','1928-03-14','Gary, IN', 2, 1962, 'US Military Academy');
INSERT INTO astronauts_by_group (astronaut_name, dob, birthplace, nasa_group_id,
nasa_group_year, university_name)
VALUES ('Gene Cernan','1934-03-14','Chicago, IL', 3, 1963, 'Purdue University');
```

Загрузка данных в две оставшиеся таблицы (`astronauts_by_university` и `astronauts_by_mission`) практически идентична.

## Запрос данных

Теперь, когда данные загружены, можно выполнить несколько простых запросов.

> **Примечание.** Прежде чем выполнять запрос к таблицам астронавтов, убедитесь, что используется ключевое пространство `astronaut_data`, как мы делали выше. Текущее пространство ключей должно быть видно в командной строке `cqlsh`.

Как и в SQL, в CQL можно запрашивать данные с помощью оператора `SELECT`. Давайте выполним быстрый запрос, чтобы проверить первые пять строк в таблице `astronauts`:

```
SELECT name, birthplace, dob, university_name
FROM astronauts LIMIT 5;
```

```
name          | birthplace          | dob         | university_name
--------------+---------------------+-------------+---------------------------
Tom Stafford  | Weatherford, OK     | 1930-09-17  | US Naval Academy
Gus Grissom   | Mitchell, IN        | 1926-04-03  | Purdue University
```

```
Gene Cernan    | Chicago, IL         | 1934-03-14   | Purdue University
Frank Borman   | Gary, IN            | 1928-03-14   | US Military Academy
Buzz Aldrin    | Montclair, NJ       | 1930-01-20   | US Military Academy

(5 rows)
```

Похоже, мы успешно загрузили данные в таблицу astronauts. Теперь давайте попробуем выполнить запрос к таблице другим способом. Вместо того, чтобы выбирать все столбцы, мы выберем только name, а затем вызовем функцию CQL token() для этого же столбца name:

```
SELECT name, token(name)
FROM astronauts LIMIT 10;

Name           | system.token(name)
-----------------+---------------------
Tom Stafford   | -8607991424493416553
Gus Grissom    | -8176206291262860019
Gene Cernan    | -7356385580624371974
Frank Borman   | -6869292563996231224
Buzz Aldrin    | -6706969656915434294
John Young     | -6312120392268580847
Jim Lovell     | -1248849440530769199
Ed White       |  977776247299278927
Pete Conrad    |  1176349381245339423
Jim McDivitt   |  2355184077141388896

(10 rows)
```

Первичным ключом таблицы astronauts является столбец name. Добавив в запрос функцию token() и передав в качестве параметра name, можно увидеть токены, использующиеся для определения размещения в кластере базы данных.

Cassandra использует хэш-алгоритм Murmur3, который хэширует значения ключей разбиения на одинаковые токены в разных системах. Выполнение приведенного выше запроса должно дать одинаковые результаты от вызова функции token() для каждой строки.

Теперь давайте повторим запрос, аналогичный тому, что мы выполняли в Postgres. Давайте получим данные об астронавтах и их университетах. Несколько астронавтов Gemini учились в US Military Academy (Военной академии США), поэтому начнем с нее:

```
SELECT astronaut_name, dob, birthplace, university_name as university
FROM astronauts_by_university
WHERE university_name = 'US Military Academy';
```

```
astronaut_name      | dob         | birthplace          | university
--------------------+-------------+---------------------+-----------------
Buzz Aldrin         | 1930-01-20  | Montclair, NJ       | US Military Academy
Dave Scott          | 1932-06-06  | San Antonio, TX     | US Military Academy
Ed White            | 1930-11-14  | San Antonio, TX     | US Military Academy
Frank Borman        | 1928-03-14  | Gary, IN            | US Military Academy
Michael Collins     | 1930-10-31  | Rome, Italy         | US Military Academy

(5 rows)
```

Как видите, в списке есть пять астронавтов, которые учились в Военной академии США. Если внимательно изучить строки, можно увидеть, что они упорядочены по столбцу `astronaut_name`.

Аналогичным образом можно запросить данные о трех астронавтах Gemini, которые учились в Военно-морской академии США:

```
SELECT astronaut_name, dob, birthplace, university_name as university
FROM astronauts_by_university
WHERE university_name = 'US Naval Academy';
```

```
astronaut_name      | dob         | birthplace          | university
--------------------+-------------+---------------------+----------------
Jim Lovell          | 1928-03-25  | Cleveland, OH       | US Naval Academy
Tom Stafford        | 1930-09-17  | Weatherford, OK     | US Naval Academy
Wally Schirra       | 1923-03-12  | Hackensack, NJ      | US Naval Academy

(3 rows)
```

> **Примечание.** Как и в случае с Postgres, в Cassandra не следует выполнять запросы без условий `WHERE` или `LIMIT`. Подобные запросы в Cassandra могут перегрузить узлы кластера и привести к таймаутам запросов или даже сбоям в работе узлов.

## Доступ из Java

Загрузив данные, давайте вернемся в нашу среду разработки Java.

### pom.xml

Чтобы получить доступ к Cassandra из Java, сначала необходимо установить Java-драйвер **DataStax** с открытым исходным кодом для Apache Cassandra. Откройте файл pom.xml нашего проекта и добавьте следующую зависимость (dependency):

```
<dependency>
    <groupId>com.datastax.oss</groupId>
```

```
    <artifactId>java-driver-core</artifactId>
    <version>4.16.0</version>
</dependency>
```

Это позволит Maven получить доступ к Java-драйверу для Cassandra.

### Класс *CassandraConn*

Далее создадим новый Java-класс с именем `CassandraConn` внутри пакета `chapter7`. У него не будет метода `main`, но необходимо определить импорт для использования `List` и поиска файла на диске:

```
package chapter7;

import java.nio.file.Paths;
import java.util.List;

import com.datastax.oss.driver.api.core.CqlSession;

public class CassandraConn {
    private CqlSession cqlSession;
```

Также создадим private-объект `CqlSession` с именем `cqlSession` для управления соединением с Cassandra. Затем создадим конструктор, который будет принимать четыре параметра типа `String`:

- имя пользователя (`username`);
- пароль (`password`);
- `secureBundleFileLocation` для хранения имени файла и его расположения в каталоге;
- переменная `keypace`, содержащая имя ключевого пространства, используемого по умолчанию (если не указано иное).

Как и в случае с классом-коннектором PostgreSQL, обернем нашу попытку подключения в `try/catch`. Java-драйвер Cassandra имеет "строителя" (`builder`) для своего объекта `CqlSession`, позволяющего определить и выполнить наше соединение одной-единственной многострочной командой:

```
public CassandraConn(String username, String pwd,
    String secureBundleLocation, String keyspace) {

    try {
        cqlSession = CqlSession.builder()
            .withCloudSecureConnectBundle
                (Paths.get(secureBundleLocation))
            .withAuthCredentials(username, pwd)
            .withKeyspace(keyspace)
            .build();
```

```
            System.out.println("[OK] Успех!");
            System.out.printf("[OK] Добро пожаловать в Astra DB! "
        "Подключено к ключевому пространству %s\n",
            cqlSession.getKeyspace().get());
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
}
```

Обычно строитель/билдер `CQLSession` принимает параметры для контекста безопасности SSL/TLS и список IP-адресов. В Astra DB обо всем этом позаботится наш защищенный пакет, поэтому необходимо задать расположение с помощью метода `withCloudSecureConnectBundle()`. Мы можем задать учетные данные и ключевое пространство по умолчанию `keyspace` с помощью аналогичных методов `withAuthCredentials()` и `withKeyspace()`, соответственно. Затем можно запустить процесс подключения, вызвав метод `build()`. После успешного подключения мы можем вывести приветственное сообщение с именем пространства ключей по умолчанию, к которому мы подключились, используя метод `getKeyspace()` класса `CqlSession`.

Нашей основной программе понадобится доступ к соединению, поэтому давайте быстро создадим для нее метод-геттер:

```
public CqlSession getCqlSession() {
    return cqlSession;
}
```

Также создадим метод `finalize()` для нашего класса. В Java метод `finalize()` автоматически вызывается, когда объект определяется как вышедший из области видимости и забирается сборщиком мусора JVM перед его уничтожением. В методе `finalize()` мы обязательно вызовем метод `close()` для нашего соединения:

```
protected void finalize() {
    cqlSession.close();
    System.out.println("[shutdown_driver] Закрываем соединение");
    System.out.println();
}
```

## Класс *AstronautCassandraDAL*

Как и в случае с Postgres, мы будем использовать DAL в качестве уровня абстракции между нашим основным кодом и кодом доступа к базе данных. Создайте новый Java-класс с именем `AstronautCassandraDAL` и убедитесь, что он находится в пакете `chapter7`. У этого нового класса не должно быть метода `main`.

Новому классу потребуются импорты List, ArrayList и time-тип Instant. Также потребуются четыре объекта из драйвера Cassandra — BoundStatement, PreparedStatement, ResultSet и Row:

```
package chapter7;

import java.time.Instant;
import java.util.ArrayList;
import java.util.List;

import com.datastax.oss.driver.api.core.cql.BoundStatement;
import com.datastax.oss.driver.api.core.cql.PreparedStatement;
import com.datastax.oss.driver.api.core.cql.ResultSet;
import com.datastax.oss.driver.api.core.cql.Row;

public class AstronautCassandraDAL {

    private CassandraConn cassandra;

    protected record AstronautMission(
        String missionName, Instant startDate,
        Instant endDate, String astronautName) {};
```

Как показано выше, мы также определяем private-переменную для нашего класса CassandraConn, а также запись AstronautMission. Эта запись AstronautMission будет немного отличаться от той, которую мы создали для Postgres, поскольку в ней используется тип Instant для дат начала и окончания миссии.

Теперь давайте создадим конструктор. Наш конструктор будет принимать строковые параметры для имени пользователя username и пароля password, а также местоположение на диске файла пакета безопасного подключения и имя пространства ключей keyspace. Эти параметры будут использоваться для инициализации объекта соединения CassandraConn:

```
public AstronautCassandraDAL(String username, String password,
        String bundleLoc, String keyspace) {

    cassandra = new CassandraConn(username, password, bundleLoc,
        keyspace);
}
```

Закончив с конструктором, перейдем к созданию методов getGeminiRoster(), аналогичных тем, что ранее создавались для PostgresDAL. Мы снова используем тот же подход, перегружая методы, чтобы они могли работать как с заданным ограничением, так и с версией без параметров.

Таким образом, первая версия метода `getGeminiRoster()` без параметров просто вызывает другую с ограничением по умолчанию `20`:

```
public List<String> getGeminiRoster() {
    return getGeminiRoster(20);
}
```

Другой метод принимает целое число с именем `limit`. Он начинается с определения возвращаемого значения в виде списка `String` с именем `returnVal`. Затем мы определяем запрос, чтобы извлечь столбец `name` из таблицы `astronauts`, указывая при этом `LIMIT`, который был передан в метод в качестве параметра:

```
public List<String> getGeminiRoster(int limit) {
    List<String> returnVal = new ArrayList<>();

    String astronautCQL = "SELECT name FROM astronauts LIMIT "
        + limit;

    try {
        ResultSet astronauts =
            cassandra.getCqlSession().execute(astronautCQL);

        for (Row astronaut: astronauts) {
            returnVal.add(astronaut.getString("name"));
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }

    return returnVal;
}
```

Затем мы создаем `try/catch` вокруг нашего вызова доступа к базе данных. Мы инициализируем новый объект `ResultSet` с именем `astronauts` для результата выполненного запроса. Наконец, мы пройдем по каждой строке CQL в наборе результатов и добавим значение столбца `name` в список `returnVal`. Как только поймаем (`catch`) возможное исключение, сможем вернуть наш `returnVal` вызывающему методу.

## Класс *GeminiAstronautsNoSQL*

Далее мы создадим новый Java-класс. Он должен называться `GeminiAstronautsNoSQL`, находиться внутри пакета `chapter7` и иметь метод `main`. Классу `GeminiAstronautsNoSQL` понадобятся импорты `Set`, `HashSet`, `List` и генератора случайных чисел `Random` из библиотеки `java.util`.

Также мы импортируем запись AstronautMission из класса AstronautCassandraDAL:

```
package chapter7;

import java.util.HashSet;
import java.util.List;
import java.util.Random;
import java.util.Set;

import chapter7.AstronautCassandraDAL.AstronautMission;

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

После этого давайте определим локальные переменные для наших переменных окружения и используем их для инстанцирования класса AstronautCassandraDAL в качестве объекта с именем astronautDAL:

```
String bundleLoc = System.getenv("ASTRA_DB_BUNDLE");
String username = System.getenv("ASTRA_DB_USER");
String password = System.getenv("ASTRA_DB_PASSWORD");
String keyspace = System.getenv("ASTRA_DB_KEYSPACE");

AstronautCassandraDAL astronautDAL =
       new AstronautCassandraDAL(username, password, bundleLoc, keyspace);
```

Далее определим новый список типа String с именем geminiAstronauts и установим его равным результату вызова метода getGeminiRoster(). Затем пройдем по каждому имени, возвращенному в списке geminiAstronauts, и выведем его на экран:

```
System.out.println("Астронавты проекта Gemini:");

List<String> geminiAstronauts = astronautDAL.getGeminiRoster();

for (String astronaut : geminiAstronauts) {
    System.out.println(astronaut);
}

System.out.println();
```

Выполнение этого кода должно дать примерно такой результат:

```
null
Project Gemini Astronauts:
Cannot invoke "com.datastax.oss.driver.api.core.CqlSession.execute(String)"
because the return value of "chapter7.CassandraConn.getCqlSession()" is null


Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"com.datastax.oss.driver.api.core.CqlSession.prepare(String)" because
the return value of "chapter7.CassandraConn.getCqlSession()" is null
```

```
at chapter7.AstronautCassandraDAL.getMissionAstronauts
(AstronautCassandraDAL.java:54)
at chapter7.GeminiAstronautsNoSQL.main(GeminiAstronautsNoSQL.java:45)
```

Это исключение происходит потому, что мы не определили переменные окружения в конфигурации запуска нашей IDE. Для этого класса необходимы четыре переменные, и их значения довольно легко найти. Во-первых, посмотрим на файл учетных данных Astra DB, который мы загрузили при генерации нового токена во время создания базы данных. По умолчанию этот файл должен находиться в каталоге `Downloads`:

```
cat ~/Downloads/bpb-token.json

{
    "clientId": "JuPplXTOxsdfOeZRUZKIX",
    "secret": "iZLo0nus_cS_9vPTejpPH2+MDXdZh1IfGq0Z6Tn3jmqkNKzT-
EZj,ACj8olji_DZH8YQZayjLGZX.PCHSALwfCMH.Lpfs89d7doS1rWAoqEkf1cFxwyqt.I-nLiZf",
    "token": "AstraCS:JuPplXTOxLBxZgmOeZRUZKIX:2f1da723f14cdfa03605d3"
}
```

Обратите внимание, что каждый файл `*token.json`, который будет сгенерирован и загружен, является уникальным. Файл, показанный выше, — пример. Но то, что нам нужно, — это json-свойство `"token"` в нижней части. Запишите это свойство, так как оно потребуется нам в скором времени.

Четыре необходимые нам переменные окружения показаны здесь, вместе с примерами значений:

```
ASTRA_DB_BUNDLE: /Users/aaronploetz/Downloads/secure-connect-bpb.zip
ASTRA_DB_KEYSPACE: astronaut_data
ASTRA_DB_PASSWORD: AstraCS:JuPplXTOxLBxZgmOeZRUZKIX:2f1da723f14cdfa03605d3
ASTRA_DB_USER: token
```

Ниже приведены рекомендации по их установке.

- `ASTRA_DB_BUNDLE`

  Расположение каталога и имя файла zip-пакета безопасного соединения, который был загружен с сайта **astra.datastax.com** ранее. Зависит от операционной системы, имени пользователя и настроек браузера по умолчанию.

- `ASTRA_DB_KEYSPACE`

  Как уже говорилось выше, это значение должно быть установлено равным `astronaut_data` или тому, как был назван `KEYSPACE`.

- `ASTRA_DB_PASSWORD`

  Значение `"token"` из приведенного выше файла `*token.json`.

- `ASTRA_DB_USER`

  Здесь всегда должно быть слово `"token"`.

Если мы устанавливаем эти переменные внутри **Run Configurations** в нашей Eclipse IDE, то они должны быть определены так, как показано на рис. 7.10.

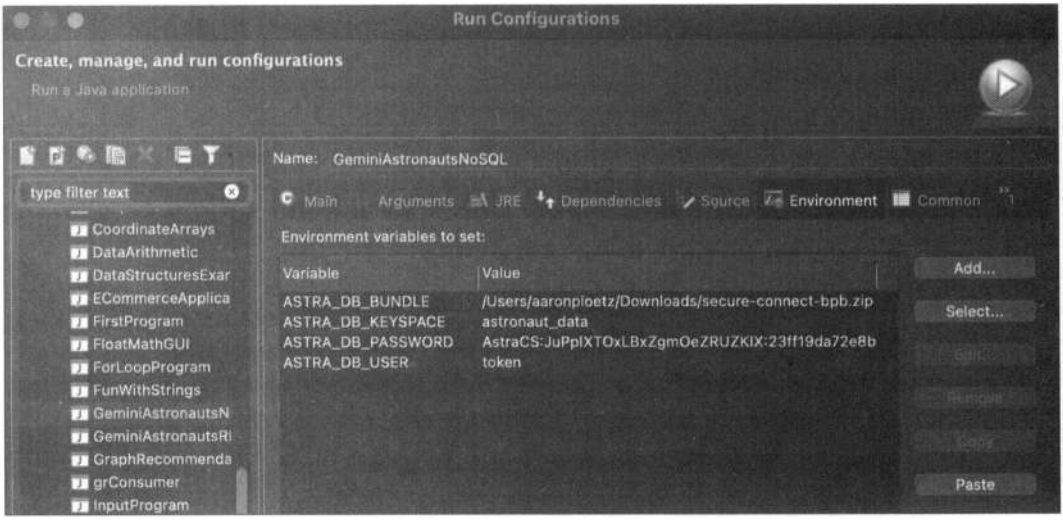



**Рис. 7.10.** Конфигурация запуска для класса `GeminiAstronautsNoSQL` с переключением на вкладку **Environment**

Теперь, запустив наш класс `GeminiAstronautsNoSQL`, мы должны получить следующий результат:

```
[OK] Успех!
[OK] Добро пожаловать в Astra DB!
Подключено к ключевому пространству astronaut_data
Project Gemini Astronauts:
Tom Stafford
Gus Grissom
Gene Cernan
Frank Borman
Buzz Aldrin
John Young
Jim Lovell
Ed White
Pete Conrad
Jim McDivitt
Neil Armstrong
Richard Gordon
Michael Collins
Gordon Cooper
Dave Scott
Wally Schirra
```

После этого можно переходить к следующему запросу, который покажет астронавтов, участвовавших в трех случайных миссиях Gemini.

## Пересмотр класса *AstronautCassandraDAL*

Для начала вернемся к нашему классу `AstronautCassandraDAL` и создадим новый метод `getMissionAstronauts`. Он будет возвращать значение списка записей `AstronautMission` и принимать название миссии (`missionName`) в качестве строкового параметра.

Начнем с определения возвращаемого значения (`returnVal`), CQL-запроса (`missionCQL`), подготовленного оператора (`missionStatement`), связанного оператора (`boundCQLMission`) и набора результатов (`missionAstronauts`):

```
public List<AstronautMission> getMissionAstronauts(String missionName) {
    List<AstronautMission> returnVal = new ArrayList<>();

    String missionCQL = "SELECT mission_name, mission_start_date, "
                + "mission_end_date, astronaut_name "
                + "FROM astronauts_by_mission "
                + "WHERE mission_name = ?;";
    PreparedStatement missionStatement =
            cassandra.getCqlSession().prepare(missionCQL);

BoundStatement boundCQLMission =
        missionStatement.bind(missionName);
ResultSet missionAstronauts =
        cassandra.getCqlSession().execute(boundCQLMission);
```

Затем завершим метод, построив цикл `for` для перебора строк в наборе результатов. Внутри цикла создается новая запись `AstronautMission` и добавляется в `returnVal`. Когда цикл завершается, возвращаются следующие результаты:

```
for (Row amRow : missionAstronauts) {
    AstronautMission astronautMission = new AstronautMission(
        amRow.getString("mission_name"),
        amRow.getInstant("mission_start_date"),
        amRow.getInstant("mission_end_date"),
        amRow.getString("astronaut_name"));

    returnVal.add(astronautMission);
}

return returnVal;
```

## Пересмотр класса *GeminiAstronautsNoSQL*

Вернемся к классу `GeminiAstronautsNoSQL` — наш код для поиска астронавтов по миссиям будет выглядеть очень похоже на то, что было в классе `GeminiAstronautsRDBMS`.

Мы создадим множество целых чисел Set, а также объект Random, чтобы сгенерировать три миссии Gemini:

```
Set<Integer> randomMissions = new HashSet<>();
Random random = new Random();
// генерация 3 случайных чисел
while (randomMissions.size() < 3) {
    int missionNumber = random.nextInt(10) + 3;
    randomMissions.add(missionNumber);
}
```

Определив объекты randomMissions и random, выполним цикл while, если множество randomMissions содержит менее трех элементов. Внутри цикла while мы сгенерируем случайное число от 3 до 12 (включительно), представляющее пилотируемые полеты Gemini в космос. Затем мы добавим сгенерированное число во множество randomMissions.

Теперь с помощью цикла for переберем все числа, сгенерированные во множестве randomMissions. Затем создадим строку с именем Gemini (с пробелом в конце) и присоединим к ней номер миссии (missionNum). Затем вызовем метод getMissionAstronauts() на нашем DAL, передав mission с помощью стандартного метода toString():

```
for (Integer missionNum : randomMissions) {

    StringBuilder mission = new StringBuilder("Gemini ");
    mission.append(missionNum.toString());
    List<AstronautMission> missionAstronauts =
        astronautDAL.getMissionAstronauts(mission.toString());
```

Наконец, построим вложенный цикл for (внутри цикла, показанного выше), который будет перебирать записи AstronautMission и выводить их содержимое:

```
    for (AstronautMission astronautMission : missionAstronauts) {
        System.out.print(astronautMission.missionName() + " ");
        System.out.print(astronautMission.startDate() + " -> ");
        System.out.print(astronautMission.endDate() + " - ");
        System.out.println(astronautMission.astronautName());
    }

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

Запуск нашего кода должен дать результат (после полного списка астронавтов Gemini), похожий на этот:

```
Gemini 3 1965-03-23T14:24:00Z -> 1965-03-23T17:16:31Z - Gus Grissom
Gemini 3 1965-03-23T14:24:00Z -> 1965-03-23T17:16:31Z - John Young
Gemini 7 1965-12-04T19:30:03Z -> 1965-12-18T14:05:04Z - Frank Borman
```

```
Gemini 7 1965-12-04T19:30:03Z -> 1965-12-18T14:05:04Z - Jim Lovell
Gemini 10 1966-07-18T22:20:26Z -> 1966-07-21T21:07:05Z - John Young
Gemini 10 1966-07-18T22:20:26Z -> 1966-07-21T21:07:05Z - Michael Collins
[shutdown_driver] Закрываем соединение
```

Как видно, мы успешно разработали схему распределенной базы данных и набор данных, используя базу данных Apache Cassandra, и выполнили запросы к ней с помощью Java.

# Выбор подходящей базы данных

Эта тема вызывает много споров и, конечно, имеет множество ответов. Главное — понять требования создаваемого приложения. Фактически, можно взять любой проект приложения и обсудить его требования в терминах теоремы CAP (из предыдущей части этой главы), и в результате прийти к базе данных, поддерживающей желаемое поведение.

Проще говоря, реляционную базу данных следует использовать, если требуется динамическая модель запросов и производительность не вызывает опасений. Приложения, которые работают с данными в парадигме онлайновой аналитической обработки (online analytical processing, OLAP), больше выиграют от использования РСУБД. С другой стороны, если требуется производительность в большом географическом регионе, то, вероятно, лучше использовать базу данных NoSQL. В конечном итоге все зависит от данных и того, как их нужно обслуживать.

# Заключение

В этой главе мы кратко обсудили историю баз данных и лежащую в их основе теорию компьютерных знаний. Затем мы познакомились с различными типами баз данных и подробно рассмотрели, как использовать две из них. Более подробную информацию о работе с PostgreSQL и Apache Cassandra можно найти на сайтах соответствующих проектов, а также в приложениях к этой книге.

Один момент, который следует прояснить в этой главе, заключается в том, что хорошие платформы DBaaS будут предлагать бесплатный уровень с достаточным количеством ресурсов для соответствующего тестирования и определения возможности долгосрочного использования. В современном мире разработчики не хотят задумываться о технологиях хранения данных, лежащих в основе их приложений. Использование облачной СУБД избавляет разработчиков от необходимости запускать и поддерживать инфраструктурные компоненты.

Однако разработчикам важно понимать, как строить модели данных и встраивать уровни абстракции данных в свои приложения. В этой главе объясняется концепция написания кода DAL, а также то, как слабая связь между приложением и его

слоем данных делает изменение основной базы данных гораздо более простой задачей. Рассмотренные в этой главе концепции могут стать мощными инструментами для Java-разработчиков.

В следующей главе мы обсудим создание веб-приложений на Java. Расскажем о веб-сервисах RESTful и пользовательских интерфейсах, а также о том, как использовать их для создания полноценного приложения. Мы будем использовать полученные в этой главе знания о базах данных, а также изучим, как создавать веб-сервисы и простые фронт-энды веб-страниц.

## Важно помнить

- Реляционные базы данных были разработаны для эффективного хранения данных.
- Базы данных NoSQL были разработаны для эффективного обслуживания данных.
- Базы данных с открытым исходным кодом (например, PostgreSQL и Apache Cassandra) широко используются во многих отраслях.
- Одним из способов внедрения учетных данных в приложение является использование переменных окружения.
- Учетные данные базы данных не должны быть жестко прописаны в приложении.
- Запрос `SELECT` всегда должен содержать оператор `WHERE` или `LIMIT`.
- Подготовленные операторы — отличный способ повысить производительность запросов, которые часто выполняются. Это связано с тем, что базе данных не нужно разбирать их при каждом использовании.
- При использовании подготовленных операторов с циклом подготовьте запрос вне цикла и выполните его внутри цикла.
- Подготовленные операторы также могут снизить риск атаки SQL-инъекции.
- DAL — это важная концепция, которую необходимо понимать при создании приложений с большим объемом данных.
- Объекты подключения к базе данных должны быть созданы один раз и использоваться повторно. Не рекомендуется открывать новое соединение для каждой операции.
- Каждая база данных была разработана для решения определенной проблемы. Обычно это проблема, которую не могли хорошо решить ее предшественники.

Глава 8

# Веб-приложения

## Введение

В этой главе мы поговорим о веб-архитектуре и создании веб-приложений. За последние пару десятилетий в этой области произошли значительные изменения. Раньше каждое приложение поставлялось со своим *пользовательским интерфейсом (UI)*, точками доступа к данным и способом отображения информации.

Сегодня большинство приложений запускается через веб-браузер, включая многие инструменты разработчика. Такие приложения, как платформы управления инфраструктурой, системы контроля версий, текстовые редакторы, электронные таблицы и инструменты для создания презентаций, могут работать в веб-браузере. Существуют даже среды разработки, которые выполняются через браузер, например *GitPod*. В предыдущей главе мы развертывали базы данных и работали с ними в командных оболочках, причем все это делалось через браузер.

## Структура

В этой главе мы обсудим вызовы RESTful. Создадим небольшие веб-сервисы, затем построим графический интерфейс, и в итоге разработаем полноценное приложение. Чтобы начать этот путь, рассмотрим следующие темы.

- Операции RESTful.
- Веб-сервисы с Spring Boot.
- Создание пользовательских интерфейсов.

## Цели

Цели обучения в этой главе — сформировать базовое понимание того, как создавать веб-приложения на Java. К концу этой главы мы:

- узнаем, что такое веб-сервис и как его создавать;

- поймем свойства конечных точек RESTful веб-сервисов;
- узнаем, как создавать веб-сервисы на Java с помощью Spring Boot;
- научимся создавать веб-приложения для работы с большими объемами данных с помощью Spring Data;
- узнаем, как создавать графические интерфейсы с помощью фреймворка Vaadin.

# Операции Restful

В веб-разработке понятие *Representational State Transfer (REST)* описывает конкретные операции и способы работы с данными в веб-приложении. Раньше аббревиатура REST писалась полностью прописными буквами, но теперь она известна просто как Rest.

Сервис считается Restful, если он соответствует определенным стандартам или принципам. Идея заключается в том, что Restful-сервисы должны быть:

- *легкими* (потреблять как можно меньше ресурсов);
- *целевыми* (выполнять как можно меньше функций (предпочтительно одну));
- *масштабируемыми* (спроектированы для получения выгоды от параллельной работы нескольких своих копий);
- *нестационарными* (хотя они могут передавать или работать с постоянными данными, ничто в сервисе не требует постоянства);
- *независимыми* (не зависеть от других сервисов для успешной работы).

При создании Restful-сервисов существуют определенные типы базовых операций *протокола передачи гипертекста* (Hypertext Transfer Protocol, HTTP), которые они могут выполнять:

- **GET**. Наиболее распространенная операция Restful, используемая для получения данных. Параметры указываются в URI запроса, и возвращается ответ.
- **POST**. Используется для создания новой записи с состоянием. Параметры могут быть указаны в URI или в теле запроса, в итоге возвращается ответ.
- **PUT**. Функционирует аналогично POST, но предназначен для обновления существующих данных.
- **DELETE**. Как следует из названия, предназначен для удаления существующих данных по параметрам в URI.

  **Примечание.** Существуют и другие операции Restful, но мы рассматриваем только эти четыре.

Все веб-запросы и ответы имеют заголовок и тело. Restful-сервисам обычно нужны данные из одного или другого (или из обоих). Об этом мы поговорим подробнее.

## URI Restful

Стоит отметить, что Restful-операции имеют свою собственную структуру URI. Как правило, это название версии и сервиса, за которым следует комбинация существительных во множественном числе. Например:

**https://www.bigboxco.com/storeapi/v1/stores/{номер_магазина}/details**

В данном случае вымышленная компания Big Box Company предоставляет сервис, известный как storeapi. Это первая версия данного сервиса, о чем свидетельствует символ v1. Конечная точка (эндпойнт) stores принимает номер магазина, а затем предлагает дополнительные конечные точки, включая details.

Версионирование (version) URI конечных точек сервисов является общепринятой хорошей практикой. Таким образом, если нам понадобится внести изменения в одну или несколько конечных точек сервиса, наши пользователи смогут использовать старую версию до тех пор, пока не будут готовы к переходу. Представим, например, что мы сделали код для изменения сервиса stores для storeapi компании Big Box Company. Лучше всего реализовать это изменение как новый сервис, обновив версию до v2.

> **Примечание.** К сожалению, на различных веб-форумах ведется много дискуссий о том, как *правильно* создавать и называть *настоящие* Restful-сервисы. Многое в этом вопросе субъективно, поэтому опытные веб-разработчики могут захотеть построить свои конечные точки сервисов иначе, чем показано в этой книге. Это не повлияет на базовую функциональность сервиса. Кроме того, читателям рекомендуется не вступать в подобные догматические дискуссии в Интернете, поскольку они обычно не приводят к значимому решению.

## Простые операции

Один из способов быстро потренироваться в потреблении (использовании) веб-сервиса Restful — это использование публичного API, предоставляемого Национальной метеорологической службой США (National Weather Service, NWS). Мы можем использовать этот сервис для получения данных о погоде с любой из приведенных здесь метеостанций: **https://forecast.weather.gov/stations.php?foo=2**.

Например, давайте воспользуемся станцией, определенной для международного аэропорта Миннеаполис/Сент-Пол, с идентификатором станции KMSP.

Введите в браузере следующий адрес:

**https://api.weather.gov/stations/kmsp/observations/latest**

Выполнив этот запрос, мы получим некоторое количество исходных данных. Если мы найдем на странице текст `temperature`, то увидим такой блок данных:

```
"temperature": {
    "unitCode": "wmoUnit:degC",
    "value": 21.100000000000001,
    "qualityControl": "V"
},
```

Эти данные представлены в формате JavaScript Object Notation (JSON). В JSON каждый блок данных заключен в пару фигурных скобок {}. Эти блоки также могут быть встроены или вложены друг в друга. Например, если мы перейдем от температуры temperature, то увидим, что она заключена внутри блока свойств properties:

```
    "properties": {
        "@id": "https://api.weather.gov/stations/KMSP/observations/2023-
        07-09T14:53:00+00:00",
        "@type": "wx:ObservationStation",
        "elevation": {
            "unitCode": "wmoUnit:m",
            "value": 255
    },
    "station": "https://api.weather.gov/stations/KMSP",
    "timestamp": "2023-07-09T14:53:00+00:00",
    "rawMessage": "KMSP 091453Z 23011G19KT 10SM SCT100 BKN110 21/09 A2993
    RMK AO2 SLP130 T02110094 58002",
    "textDescription": "Mostly Cloudy",
    "icon": "https://api.weather.gov/icons/land/day/bkn?size=medium",
    "presentWeather": [],
    "temperature": {
        "unitCode": "wmoUnit:degC",
        "value": 21.100000000000001,
        "qualityControl": "V"
    },
```

Мы также можем запускать Restful веб-вызовы из терминала или интерфейса командной строки с помощью приложения cURL. cURL — это инструмент командной строки для работы с данными из Интернета. Созданный в 1996 году как небольшой проект шведским разработчиком (и лауреатом премии Polhem) Даниэлем Стенбергом (Daniel Stenberg), cURL сейчас широко используется в качестве основного инструмента для выполнения веб-запросов на системном уровне. Он стал довольно распространенным, поскольку cURL поставляется с Linux, MacOS, Windows 11 и более поздними версиями Windows 10. Его можно встретить даже во многих других типах встраиваемых систем — например, многие бытовые приборы и автомобили, выпускаемые сегодня, также работают с cURL нативно.

> **Примечание.** Для получения дополнительной информации о cURL посетите сайт **https://www.curl.se**.

Чтобы проверить установку cURL, можно выполнить команду `curl --version` в терминале:

```
curl --version
```

```
curl 7.81.0 (x86_64-pc-linux-gnu) libcurl/7.81.0 OpenSSL/3.0.2 zlib/1.2.11
brotli/1.0.9 zstd/1.4.8 libidn2/2.3.2 libpsl/0.21.0 (+libidn2/2.3.2)
libssh/0.9.6/openssl/zlib nghttp2/1.43.0 librtmp/2.3 OpenLDAP/2.5.14
```

```
Release-Date: 2022-01-05
Protocols: dict file ftp ftps gopher gophers http https imap imaps ldap ldaps
mqtt pop3 pop3s rtmp rtsp scp sftp smb smbs smtp smtps telnet tftp
Features: alt-svc AsynchDNS brotli GSS-API HSTS HTTP2 HTTPS-proxy IDN IPv6
Kerberos Largefile libz NTLM NTLM_WB PSL SPNEGO SSL TLS-SRP UnixSockets zstd
```

Чтобы выполнить тот же запрос (для получения данных о метеостанции) из `curl`, сделаем следующее:

```
curl -X GET https://api.weather.gov/stations/kmsp/observations/latest
-H "Content-Type: application/json"
-H "Accept: application/json"
```

Сначала используется опция `-X`, чтобы указать, что выполняется `GET`-запрос. Затем указан URI, за которым следуют параметры заголовка с помощью опции `-H` дважды.

В данном случае указывается `Content-Type application/json`, что означает, что мы говорим cURL, что тело запроса будет в формате JSON. У этого запроса нет тела запроса, поэтому оно нам не нужно. Далее указываем свойство `Accept` в заголовке, также со значением `application/json`. Это наш способ сообщить веб-серверу, что мы хотим получить данные в формате JSON.

> **Примечание.** Поскольку свойства `Accept` указывают веб-серверу, как мы хотим, чтобы наши данные были отформатированы в ответе, они обычно не используются в заголовках ответов.

Прежде чем двигаться дальше, давайте выполним еще один GET-запрос cURL. Если прошло несколько дней с тех пор, как мы работали над предыдущей главой, наш облачный экземпляр Astra DB, возможно, перешел в *спящий* режим. Нам необходимо активировать его, поскольку он потребуется для некоторых сервисов, которые мы будем создавать по ходу работы над этой главой. К счастью, операции Astra DB открыты для Restful API, и его можно пробудить, запросив конечную точку `keyspaces` нашей базы данных:

```
curl -X GET https://34176be8-1675-4204-ae98-3a66f6f772dd-us-
east1.apps.astra.datastax.com/api/rest/v2/schemas/keyspaces/
-H "X-Cassandra-Token:AstraCS:JuPplXTOxLBxZgmOeZRUZKIX:2f1da723f14cdfa03605d3"
-H "Content-Type: application/json"
-H "Accept: application/json"

{"message":"Resuming your database, please try again shortly."}
```

URI нашей базы данных состоит из идентификатора базы данных и региона. Мы также передаем в заголовке запроса значения `Content-Type`, `Accept` и `X-Cassandra-Token`. Обратите внимание, что каждая конечная точка и токен будут разными для каждого экземпляра Astra DB.

**Примечание.** Будьте осторожны и не передавайте конфиденциальные данные в незащищенном (http) веб-запросе. Все веб-сайты и сервисы должны быть защищены сертификатом TLS.

# Веб-сервисы с помощью Spring Boot

Теперь перейдем к созданию веб-сервисов на Java. Веб-сервисы — это, по сути, методы, которые открываются и запускаются в Интернете без графического интерфейса, что делает их "безголовыми" (headless). Один веб-сервис обычно выполняет одну функцию — запись или получение данных. Группу веб-сервисов можно назвать *слоем сервисов*.

Как правило, создание и запуск слоя веб-сервисов включает в себя установку локального веб-сервера и развертывание на нем кода. Однако мы пойдем простым путем и воспользуемся Spring Boot.

*Spring Boot* — это фреймворк для веб-сервисов, который позволяет быстро создавать и развертывать автономные веб-приложения на Java для тестирования или производства. Spring Boot позволяет разработчику встроить веб-сервер вместе со скомпилированными двоичными файлами Java, что значительно упрощает развертывание приложения.

В Spring присутствует онлайн-инструмент для быстрой сборки и настройки стартовых файлов проекта. Достаточно нажать несколько кнопок, и все готово к работе. Чтобы перейти к **spring initializr**, просто откройте сайт: **https://start.spring.io/**. Мы увидим экран, изображенный на рис. 8.1 и 8.2.

Для нашего приложения Weather Application мы сконфигурируем его, как показано на рис. 8.1, используя следующие свойства:

- **Project: Maven**
- **Language: Java**
- **Spring Boot version: 3.1.1**
- **Group:** `com.codewithjava21`
- **Artifact and name:** `weatherapp`
- **Description:** `Consumes and displays weather data`
- **Package name:** `com.codewithjava21.weatherapp`
- **Packaging: Jar**
- **Java version: 21**

На этом закончим, давайте посмотрим на правую часть экрана, показанную на рис. 8.2. Мы добавим три зависимости. Для этого достаточно нажать кнопку **ADD DEPENDENCIES**, а затем найти каждую из следующих зависимостей по отдельности:

- **Spring Web**

- **Vaadin**
- **Spring Data for Apache Cassandra**

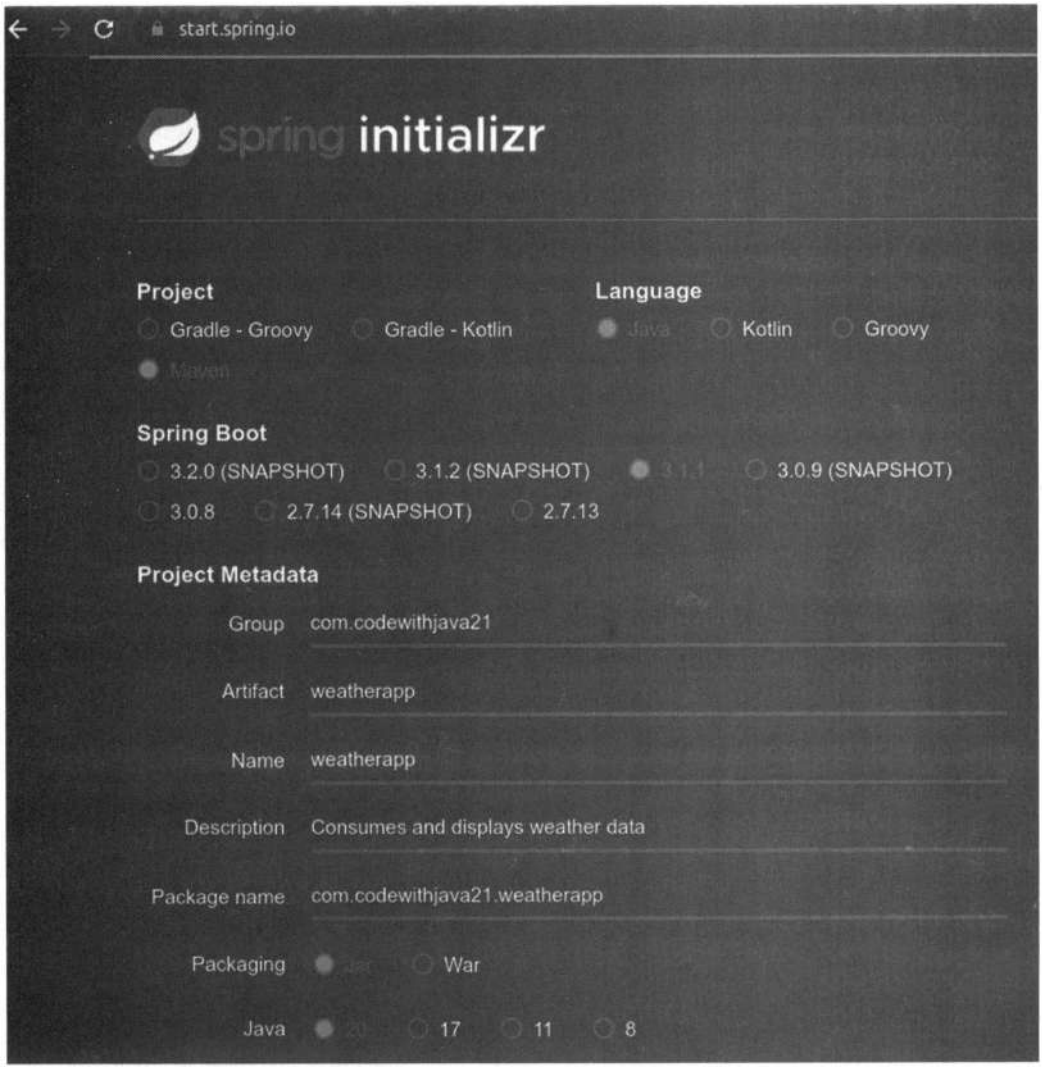



**Рис. 8.1.** Левая часть **spring initializr**, позволяющая задать управление зависимостями и другие параметры конфигурации проекта

Рассмотрев обе стороны экрана **spring initializr**, можно нажать кнопку **Generate**. Это приведет к тому, что веб-браузер скачает файл с именем weatherapp.zip.

Чтобы импортировать этот файл в нашу IDE, сначала переместим ZIP-файл в каталог рабочего пространства и распакуем его. Затем в Eclipse выберем меню **File**, а затем опцию **Import**.

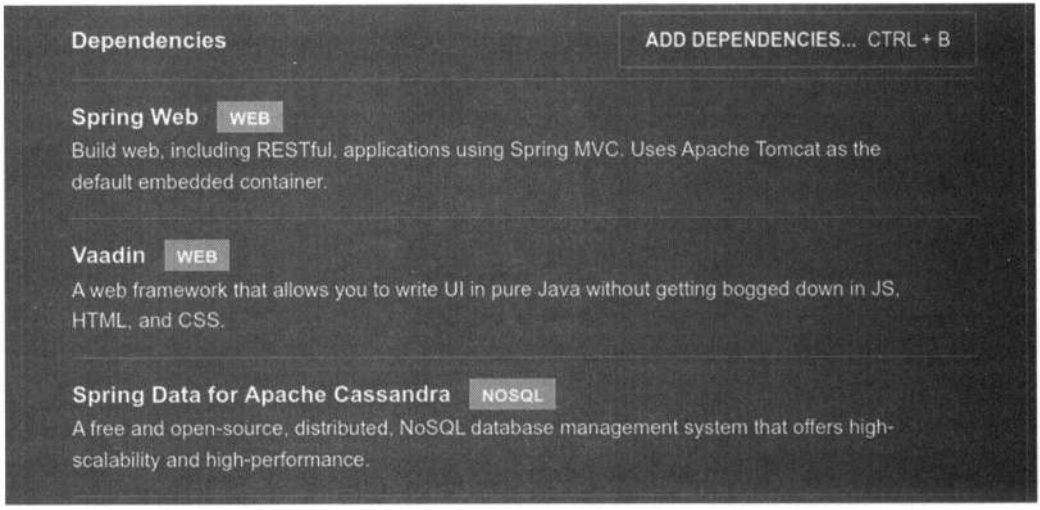



**Рис. 8.2.** Правая часть **spring initializr**.
Здесь добавляются начальные зависимости проекта

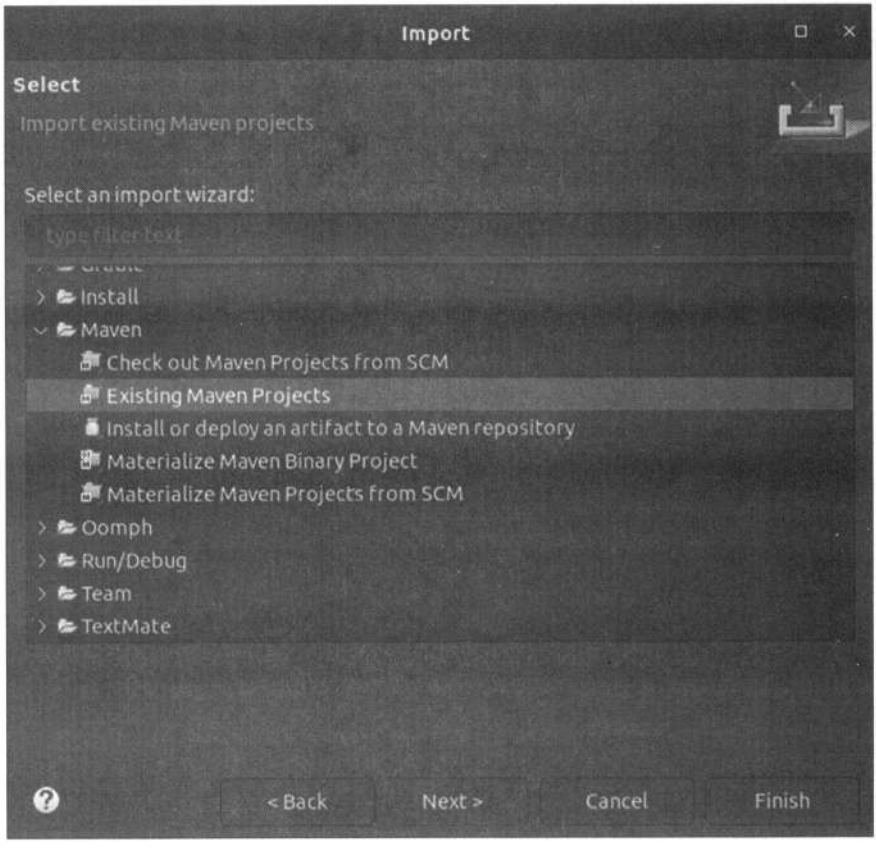



**Рис. 8.3.** Опция диалогового окна **Import**, указывающая, что наш zip-файл будет существующим проектом Maven

В следующем диалоговом окне укажем, что мы импортируем существующие проекты Maven **Existing Maven Projects** (как показано на рис. 8.3), и нажмем кнопку **Next**. Далее перейдем в каталог, в котором находится новая папка **weatherapp**, и щелкнем в ней, убедившись, что файл pom.xml присутствует. После этого нажмите кнопку **Finish**, чтобы завершить процесс.

> **Примечание.** Если мы находимся в директории, содержащей файл pom.xml, IDE должна найти проект для импорта.

Теперь мы импортировали новый проект в IDE. После этого мы должны увидеть **weatherapp** в проводнике проектов в левой части IDE (рис. 8.3).

### pom.xml

Посмотрите на файл pom.xml, который был сгенерирован **Spring Boot Initializr**. При создании приложения это определенно помогает ускорить начальные циклы разработки, быстро доводя программистов до той стадии, когда они могут начать писать код.

Тем не менее внесем некоторые коррективы. Во-первых, если в **Spring Boot Initializr** не была доступна опция для Java 21, нужно это исправить. Найдите раздел свойств **properties** и убедитесь, что свойство `java.version` установлено соответствующим образом:

```
<properties>
    <java.version>21</java.version>
```

Далее нужно закомментировать зависимости dependency Spring Data Cassandra и Vaadin. Они не понадобятся сразу, а при неправильной настройке иногда могут вызвать проблемы. Обязательно используйте символы комментариев в стиле HTML/XML (`<!-- -->`). Теперь раздел зависимостей dependencies должен выглядеть следующим образом:

```
<dependencies>
<!-- <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-cassandra</artifactId>
    </dependency> -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
<!-- <dependency>
        <groupId>com.vaadin</groupId>
        <artifactId>vaadin-spring-boot-starter</artifactId>
    </dependency> -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
```

```
        <scope>test</scope>
    </dependency>
</dependencies>
```

## MVC

Многие программные продукты построены на основе широко известных шаблонов проектирования. По сути, эти шаблоны представляют собой стандартные способы создания архитектуры программных компонентов. Мы будем использовать наиболее распространенный паттерн проектирования веб-приложений, известный как паттерн "*Модель, представление, контроллер*" (Model, View, Controller — MVC).

В MVC компоненты веб-приложения разбиваются на три части.

- **Модель**. Структура данных и то, как они попадают в базовое хранилище данных.
- **Представление**. Пользовательский интерфейс, который управляет взаимодействием с пользователем и тем, как данные отображаются для него.
- **Контроллер**. Средний слой, который абстрагирует модель от представления. Он обрабатывает запросы данных из пользовательского интерфейса и иногда может выступать в качестве отдельного поставщика конечных точек обслуживания.

Мы построим наше погодное приложение с использованием паттерна MVC, и начнем с контроллера.

### Контроллер погодного приложения

Создайте новый Java-класс с именем `WeatherAppController` внутри пакета `com.codewithjava21.weatherapp`. Этот класс не должен иметь метода `main`. Нам понадобятся импорты из Spring Framework для следующих классов: `ResponseEntity`, `GetMapping`, `RequestMapping` и `RestController`:

```
package com.codewithjava21.weatherapp;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RequestMapping("/weather")
@RestController
public class WeatherAppController {
```

Классы `RequestMapping` и `RestController` являются аннотациями. С помощью этих аннотаций Spring Boot знает, как быстро подключить наше приложение к сервису. Мы просто должны использовать их для указания Spring Boot, что представляет собой каждый компонент. Наш класс определяет Restful-контроллер, поэтому нам нужна аннотация `@RestController`. Базовое имя сервиса для этого контроллера —

weather, поэтому нам нужно указать его с помощью аннотации @RequestMapping, как показано выше.

## Конечная точка сервиса Hello World

Теперь мы создадим простую конечную точку сервиса нашего погодного приложения. Создадим метод с именем getHello. Он будет снабжен аннотацией @GetMapping, указывающей имя конечной точки сервиса helloworld. Метод будет иметь возвращаемый тип ResponseEntity<String>:

```
@GetMapping("/helloworld")
public ResponseEntity<String> getHello() {
    return ResponseEntity.ok("Hello world!\n");
}
```

Сам метод довольно прост. Мы вернем результат метода ResponseEntity.ok() с сообщением Hello world! в качестве единственного параметра. Если все работает, метод должен вернуть HTTP-код ответа 200 (ok).

**Примечание.** Краткий справочник по кодам ответов HTTP можно найти в *приложении 4*.

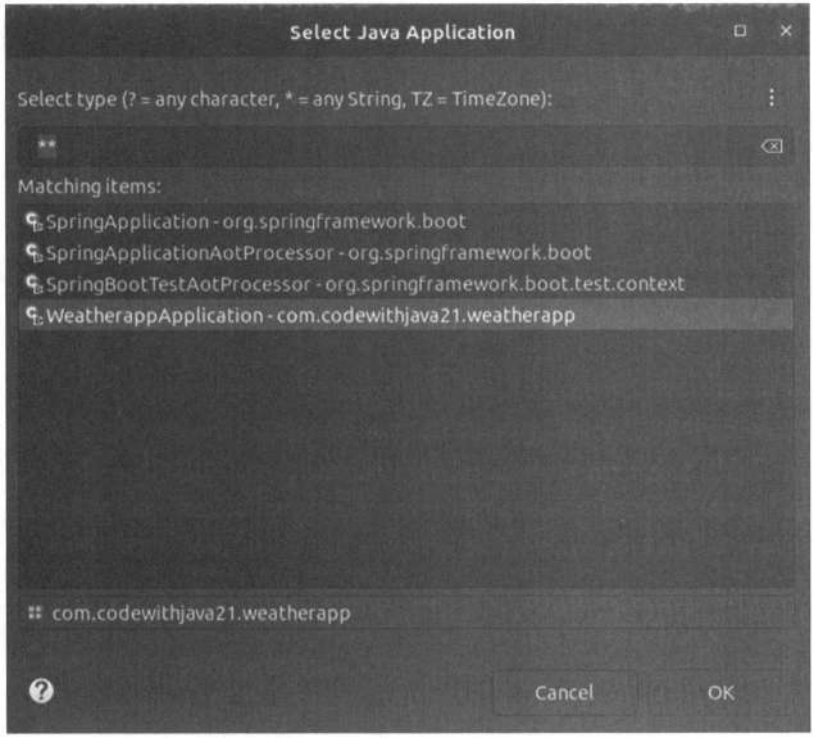



**Рис. 8.4.** Убедитесь, что выбрали из списка приложение WeatherApplication

Теперь можно выполнить Maven Install через IDE, а затем запустить наше приложение. При первой попытке запуска нам может быть предложено выбрать наше приложение (рис. 8.4).

При запуске нашего приложения из IDE в консольном окне должен появиться журнал сообщений, похожий на этот:

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
:: Spring Boot ::                (v3.1.1)
2023-07-13T07:00:57.366-05:00 INFO 8481 --- [ main]
c.c.weatherapp.WeatherappApplication : Starting WeatherappApplication
using Java 21-ea with PID 8481
```

Далее будет несколько дополнительных строк. Последние две очень важны:

```
2023-07-13T07:00:59.444-05:00 INFO 8481 --- [ main]
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s):
8080 (http) with context path ''
2023-07-13T07:00:59.457-05:00 INFO 8481 --- [ main]
c.c.weatherapp.WeatherappApplication : Started WeatherappApplication
in 2.592 seconds (process running for 3.124)
```

Последняя строка показывает, что наше приложение действительно запущено. Предпоследняя строка показывает, что встроенный Apache Tomcat® запущен на порту 8080 по протоколу (http).

Запустив это приложение, давайте перейдем в терминал, чтобы протестировать наш сервис Hello World с помощью cURL. Обязательно помните следующее:

- мы выполняем операцию Get;
- в URI указывается http;
- используйте локальный адрес, за которым следует двоеточие и порт (127.0.0.1:8080);
- укажите имя службы weather и конечную точку службы helloworld, разделенные косой чертой (/weather/helloworld).

```
curl -X GET http://127.0.0.1:8080/weather/helloworld

Hello world!
```

Теперь у нас есть простая рабочая конечная точка веб-сервиса, которую можно вызывать из командной строки или через веб-браузер. Теперь перейдем к более сложному примеру.

## Модель погодного приложения

Теперь создадим сервисы для нашего погодного приложения. Прежде чем мы сможем добавить новые конечные точки сервисов для него, необходимо создать модель данных.

Spring предоставляет нам набор инструментов для управления нашей моделью, основанный на используемой базе данных. Этот набор инструментов известен как **Spring Data**. Для нашего проекта мы будем использовать подмножество **Spring Data**, известное как **Spring Data Cassandra**.

### Определение нового пространства ключей

Поскольку мы возобновили работу нашей базы данных Astra DB, давайте перейдем на панель **Astra Dashboard** и нажмем кнопку **Add Keyspace**, как показано на рис. 8.5.

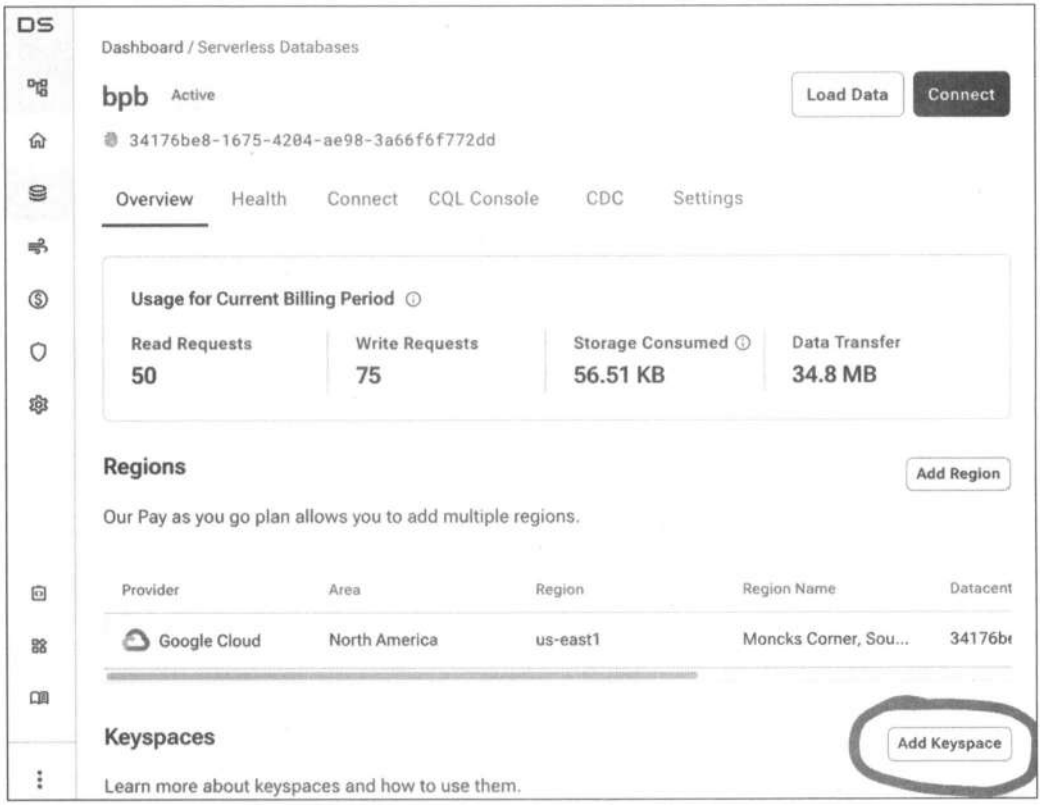



**Рис. 8.5.** Панель **Astra Dashboard** с кнопкой **Add Keyspace** (обведено) в правом нижнем углу

Назовем это новое пространство ключей weatherapp. Далее перейдем на вкладку **CQL Console** и используем (use) наше новое пространство ключей:

```
use weatherapp;
```

## Определение новой таблицы

Чтобы хранить данные о погоде в нашей базе данных, сначала нужно создать новую таблицу. Новая таблица будет называться weather_by_station_by_month, а ее CQL-определение будет выглядеть следующим образом:

```
CREATE TABLE weather_by_station_by_month (
    station_id TEXT,
    month_bucket INT,
    reading_timestamp TIMESTAMP,
    reading_icon TEXT,
    station_coordinates_lat FLOAT,
    station_coordinates_lon FLOAT,
    temperature_c FLOAT,
    wind_direction_deg INT,
    wind_speed_kmh FLOAT,
    wind_gust_kmh FLOAT,
    visibility_m INT,
    precipitation_last_hour FLOAT,
    cloud_cover MAP<INT,TEXT>,
    PRIMARY KEY ((station_id,month_bucket),reading_timestamp)
) WITH CLUSTERING ORDER BY (reading_timestamp DESC);
```

Это должен быть простой способ организации и хранения данных, получаемых от NWS.

Идея определения нашего первичного ключа состоит в том, чтобы создать составной ключ раздела из station_id и month_bucket. Сначала мы будем хранить только релевантные данные о станциях. Но при желании мы можем хранить данные о нескольких станциях. Хотя Cassandra отлично справится с размещением наших данных в распределенной базе данных, если бы мы разделили данные на основе station_id, то в итоге получили бы горячие разделы.

*Горячие разделы* (hot partitions) — это разделы, из которых читают или в которые пишут гораздо чаще, чем в другие. Это может привести к неравномерному распределению данных и шаблонов доступа. Поэтому в качестве ключа раздела нужно использовать не только station_id.

Основанные на времени данные иногда называют данными *временных рядов*. Другая проблема с использованием только station_id в качестве ключа раздела заключается в том, что он будет расти без ограничений. Со временем, когда мы продолжим записывать данные для каждой новой временной метки timestamp, раздел станет слишком большим. Поэтому нам нужно создать компонент времени, который

будет добавляться к нашему ключу раздела, чтобы гарантировать, что наш раздел будет расти только до конечной величины.

Этот прием моделирования данных известен как "корзина/бакет" (bucketing). Для нашего случая нам подойдет бакет времени, основанный на месяце. Если мы будем хранить запись в разделе для каждой станции по часам, то никогда не превысим 744 строки в разделе (24 часа в сутки, максимум 31 день в месяце). Поэтому каждая строка будет хранить `month_bucket`, который будет представлять собой комбинированное значение года и числового месяца (например: 202307 для июля 2023 года).

Внутри каждого раздела нам нужно что-то, что позволит однозначно идентифицировать каждый ряд. Свойство `timestamp` подходит под это требование. Однако `timestamp` — не самое удачное название для столбца базы данных, поскольку во многих базах данных (в том числе и в Cassandra) есть тип данных `timestamp`. Поэтому мы назовем его `reading_timestamp`, подразумевая, что это временная метка текущего показания погоды.

## Генерация нового маркера

К сожалению, наш маркер доступа к базе данных, который был автоматически сгенерирован Astra в предыдущей главе, не обладает достаточными привилегиями для данного приложения. Поэтому нам нужно сгенерировать новый. На панели управления Astra перейдите к базам данных. Выберите базу данных **bpb** (рис. 8.6) и найдите значок меню с тремя точками в крайнем правом углу. Щелкните на нем, чтобы увидеть опцию **Generate a Token**.

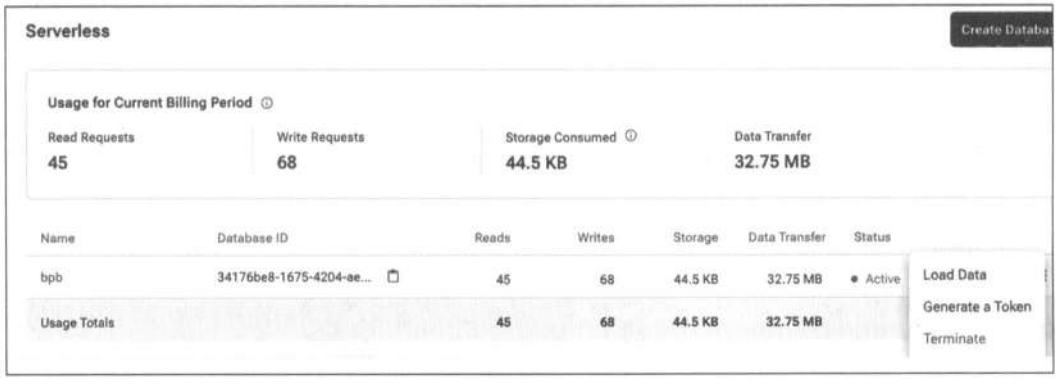



**Рис. 8.6.** Информация базы данных **bpb**, показывающая, как сгенерировать новый токен доступа

Проекты, использующие фреймворк Spring Data, способны воссоздать схему базы данных, если она не существует. Для работы им потребуются привилегии на уровне создания таблиц и других схем. Поэтому нужно создать новый токен для роли администратора базы данных Database Administrator.

## Установка свойств приложения и переменных окружения

После того как получен новый токен, понадобится установить переменные окружения, как мы делали это в предыдущей главе. Нашему приложению потребуются следующие переменные окружения:

- ASTRA_DB_KEYSPACE
- ASTRA_DB_APP_TOKEN
- ASTRA_DB_ID
- ASTRA_DB_REGION

> **Примечание.** Помните, что в среде Eclipse IDE можно задать переменные как часть конфигурации выполнения. Подробнее об этом читайте в предыдущей главе.

Внутри проекта можно увидеть файл с именем application.properties, расположенный в каталоге src/main/resources. Начните с переименования этого файла в application.yml, чтобы мы могли использовать формат Yet Another Markup Language (YAML) для конфигурации нашего приложения. Файл application.yml должен выглядеть следующим образом:

```
server:
  port: 8080
  error:
    include-stacktrace: always
  spring:
    application:
      name: WeatherApp
    profiles:
      active: default
    data:
      cassandra:
        schema-action: NONE

  astra:
    api:
      application-token: ${ASTRA_DB_APP_TOKEN}
      database-id: ${ASTRA_DB_ID}
      database-region: ${ASTRA_DB_REGION}
      cross-region-failback: false
    cql:
      enabled: true
      download-scb:
        enabled: true
      driver-config:
        basic:
          session-keyspace: ${ASTRA_DB_KEYSPACE}
          request:
            timeout: 8s
```

```
      consistency: LOCAL_QUORUM
      page-size: 5000
    advanced:
      connection:
        init-query-timeout: 10s
        set-keyspace-timeout: 10s
      control-connection:
        timeout: 10s
```

**Примечание.** Переменные окружения указываются в разделе `astra`, так как фреймворк Spring может подтягивать их из ОС.

## Изменение pom.xml

На данном этапе пришло время пересмотреть файл pom.xml и убрать из него зависимость `dependency` для Spring Data Cassandra. Мы также должны добавить еще одну зависимость `dependency` для Astra Spring Boot Starter:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-cassandra</artifactId>
</dependency>
<dependency>
    <groupId>com.datastax.astra</groupId>
    <artifactId>astra-spring-boot-3x-starter</artifactId>
    <version>0.6.4</version>
</dependency>
```

## Класс *WeatherPrimaryKey*

Создайте новый класс с именем `WeatherPrimaryKey` внутри пакета `com.codewithjava21.weatherapp`. У него не должно быть метода `main`.

```
package com.codewithjava21.weatherapp;

import java.time.Instant;

import org.springframework.data.cassandra.core.cql.PrimaryKeyType;
import org.springframework.data.cassandra.core.mapping.PrimaryKeyClass;
import org.springframework.data.cassandra.core.mapping.PrimaryKeyColumn;

@PrimaryKeyClass
public class WeatherPrimaryKey {
```

Наш класс будет содержать четыре импорта. Мы хотим импортировать класс `Instant` из библиотеки времени (`time`) Java. Затем из библиотеки Spring Data

Cassandra импортируем классы PrimaryKeyType, PrimaryKeyClass и PrimaryKeyColumn. Мы будем использовать PrimaryKeyClass в качестве аннотации, которая предшествует определению нашего класса (как показано в коде выше).

Далее определим три private-свойства нашего класса WeatherPrimaryKey:

```
@PrimaryKeyColumn(name = "station_id",
                  ordinal = 0,
                  type = PrimaryKeyType.PARTITIONED)
private String stationId;

@PrimaryKeyColumn(name = "month_bucket",
                  ordinal = 1,
                  type = PrimaryKeyType.PARTITIONED)
private int monthBucket;

@PrimaryKeyColumn(name = "reading_timestamp",
                  ordinal = 2,
                  type = PrimaryKeyType.CLUSTERED)
private Instant timestamp;
```

Эти свойства будут использоваться для свойств station_id, month_bucket и timestamp. Обратите внимание, что мы предваряем определения этих свойств аннотацией @PrimaryKeyColumn. Эта аннотация поможет Java-репозиторию лучше понять, как в Cassandra определяются столбцы первичных ключей. Убедитесь, что:

- свойства name соответствуют именам столбцов из таблицы;
- свойство ordinal соответствует их порядку в определении первичного ключа;
- тип компонента первичного ключа совпадает (разделенный или кластеризованный).

Далее создадим простой конструктор с тремя аргументами, который поможет инстанцировать новые объекты WeatherPrimaryKey:

```
public WeatherPrimaryKey(String stationId, int monthBucket,
       Instant timestamp) {

    this.stationId = stationId;
    this.monthBucket = monthBucket;
    this.timestamp = timestamp;
}
```

Наконец, завершающее, что нужно нашему классу, — это методы getter и setter для каждого свойства:

```
public String getStationId() {
    return stationId;
}
```

```
public void setStationId(String stationId) {
    this.stationId = stationId;
}

public int getMonthBucket() {
    return monthBucket;
}

public void setMonthBucket(int monthBucket) {
    this.monthBucket = monthBucket;
}

public Instant getTimestamp() {
    return timestamp;
}

public void setTimestamp(Instant timestamp) {
    this.timestamp = timestamp;
}
```

### Класс *WeatherEntity*

Закончив с этим классом, давайте создадим новый класс `WeatherEntity` и убедимся, что он является частью пакета `com.codewithjava21.weatherapp`. В нем должно быть четыре импорта, включая тип `Map`, а также классы `Column`, `PrimaryKey` и `Table` из библиотеки Spring Data Cassandra:

```
package com.codewithjava21.weatherapp;

import java.util.Map;

import org.springframework.data.cassandra.core.mapping.Column;
import org.springframework.data.cassandra.core.mapping.PrimaryKey;
import org.springframework.data.cassandra.core.mapping.Table;

@Table("weather_by_station_by_month")
public class WeatherEntity {
```

В Spring Data наши классы сущностей напрямую сопоставляются с таблицами базы данных. Поэтому мы будем использовать аннотацию `@Table` (с именем нашей таблицы), чтобы указать, к какой таблице сопоставляется этот класс. Создадим свойства для нашего первичного ключа (используя класс `WeatherPrimaryKey`) и каждого из наших отдельных столбцов. Столбцы будут отмечены аннотацией `@column`, которая укажет на их соответствие имени столбца в Cassandra:

```
@PrimaryKey
private WeatherPrimaryKey primaryKey;
```

```
@Column("reading_icon")
private String readingIcon;

@Column("station_coordinates_lat")
private float stationCoordinatesLatitude;

@Column("station_coordinates_lon")
private float stationCoordinatesLongitude;

@Column("temperature_c")
private float temperatureCelsius;

@Column("wind_direction_deg")
private int windDirectionDegrees;

@Column("wind_speed_kmh")
private float windSpeedKMH;

@Column("wind_gust_kmh")
private float windGustKMH;

@Column("visibility_m")
private int visibilityM;

@Column("precipitation_last_hour")
private float precipitationLastHour;

@Column("cloud_cover")
private Map<Integer,String> cloudCover;
```

Также необходимо создать геттеры и сеттеры для каждого из свойств, включая первичный ключ (`primary key`).

> **Примечание.** В целях краткости мы не будем приводить определения геттеров и сеттеров для этого класса, поскольку их довольно много. К этому моменту мы уже знаем, как создавать геттеры и сеттеры. За дополнительной информацией обращайтесь к сопутствующему книге GitHub-репозиторию.

## Класс *WeatherReading*

Далее необходимо создать новый класс `WeatherReading` внутри пакета `com.codewithjava21.weatherapp`. Это простой `POJO`-класс, который будет использоваться для возврата данных о погоде в вызывающие сервисы:

```
package com.codewithjava21.weatherapp;

import java.time.Instant;
```

```
import java.util.Map;

public class WeatherReading {
    private String stationId;
    private int monthBucket;
    private Instant timestamp;
    private String readingIcon;
    private float stationCoordinatesLatitude;
    private float stationCoordinatesLongitude;
    private float temperatureCelsius;
    private int windDirectionDegrees;
    private float windSpeedKMH;
    private float windGustKMH;
    private int visibilityM;
    private float precipitationLastHour;
    private Map<Integer,String> cloudCover;
```

Как и в случае с классом `WeatherEntity`, здесь не приводятся геттеры и сеттеры.

### Интерфейс *WeatherAppRepository*

Далее создадим новый интерфейс `WeatherAppRepository` внутри пакета `com.codewithjava21.weatherapp`. Для создания интерфейса потребуется два импорта из фреймворка Spring Data, включая классы `CassandraRepository` и `Repository`:

```
package com.codewithjava21.weatherapp;
import org.springframework.data.cassandra.repository.CassandraRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface WeatherAppRepository extends
        CassandraRepository<WeatherEntity,WeatherPrimaryKey> {
    @Query("SELECT * FROM weather_by_station_by_month WHERE station_id=?0
    AND month_bucket=?1 LIMIT 1")
    List<WeatherEntity> findByStationIdAndMonthBucket(String stationId,
    int monthBucket);
}
```

Сначала пометим наш интерфейс аннотацией `@Repository`. Затем расширим интерфейс, чтобы использовать класс `CassandraRepository`, передавая классы `WeatherEntity` и `WeatherPrimaryKey` в качестве аргументов типа.

Мы также определим пользовательский запрос `SELECT`. Spring Data по умолчанию предоставляет поиск по полному первичному ключу. Однако мы хотим запрашивать только две части нашего первичного ключа и ограничивать результаты с помощью `LIMIT`. Поэтому мы определим метод запроса с именем `findByStationIdAndMonthBucket`, передадим `stationId` и `monthBucket` в качестве параметров и определим точный запрос с помощью аннотации `@Query`.

## Построение JSON-объектов ответа

Для создания следующих частей приложения вернемся к нашему классу `WeatherAppController` и его вспомогательным классам. Сначала нам нужно создать Java-класс, который поможет нам сериализовать вывод погодного JSON-ответа NWS в POJO. Если вспомнить вывод из конечной точки сервиса **https://api.weather.gov/stations/kmsp/observations/latest**, то становится очевидным, что понадобится несколько различных типов объектных классов, вложенных друг в друга.

### Класс *Measurement*

Если рассмотреть возвращаемый сервисом JSON, то многие измерения в разделе свойств имеют стандартный формат: `unitCode`, `value` и `qualityControl`. Свойство `qualityControl` мы пропустим, так как оно не является полезным (для нас). Но нам понадобится класс `Measurement` со свойствами для `unitCode` и `value`.

Создайте новый класс с именем `Measurement` внутри пакета `com.codewithjava21.weatherapp`. Наш класс будет иметь только private-свойства для `unitCode` и `value`, а также геттеры и сеттеры для каждого из них:

```
package com.codewithjava21.weatherapp;

public class Measurement {

        private String unitCode;
        private float value;

        public String getUnitCode() {
                return unitCode;
        }

        public void setUnitCode(String unitCode) {
                this.unitCode = unitCode;
        }

        public float getValue() {
                return value;
        }

        public void setValue(float value) {
                this.value = value;
        }
}
```

### Класс *CloudLayer*

Одной из самых уникальных структур данных в разделе свойств является раздел `CloudLayer`. Раздел `CloudLayer` похож на класс `Measurement`, но имеет имя `base`. На

основе этого свойства и свойства amount мы можем создать класс, использующий наш класс Measurement.

Создайте новый класс с именем CloudLayer внутри пакета com.codewithjava21.weatherapp. У него будет два приватных свойства: объект Measurement с именем base и свойство String с именем amount:

```
package com.codewithjava21.weatherapp;

public class CloudLayer {

        private Measurement base;
        private String amount;

        public Measurement getBase() {
                return base;
        }

        public void setBase(Measurement base) {
                this.base = base;
        }

        public String getAmount() {
                return amount;
        }

        public void setAmount(String amount) {
                this.amount = amount;
        }
}
```

## Класс *Properties*

Теперь необходимо создать класс Properties. Создайте новый класс с именем Properties внутри пакета com.codewithjava21.weatherapp. Ему потребуется единственный импорт Instant из библиотеки времени (time) Java. Мы создадим в нем несколько private-свойств, соответствующих секции properties JSON из вызова вебсервиса, включающего массив CloudLayer:

```
package com.codewithjava21.weatherapp;

import java.time.Instant;

public class Properties {
    private String station;
    private Instant timestamp;
    private String icon;
    private Measurement temperature;
```

```
    private Measurement windDirection;
    private Measurement windSpeed;
    private Measurement windGust;
    private Measurement visibility;
    private Measurement precipitationLastHour;
    private CloudLayer[] cloudLayers;
```

Как и в случае с классом `WeatherEntity`, здесь не будут приведены определения геттеров и сеттеров этого класса. Но они должны быть созданы.

### Класс *Geometry*

Один из разделов JSON ответа NWS назывался `Geometry`, и в нем содержалась информация о метеостанции. В нем был тип `String` с именем `type`, а также массив с плавающей точкой для хранения координат широты и долготы станции. Создадим новый класс с именем `Geometry` внутри пакета `com.codewithjava21.weatherapp` с соответствующими свойствами:

```
package com.codewithjava21.weatherapp;

public class Geometry {
    private String type;
    private float[] coordinates;

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public float[] getCoordinates() {
        return coordinates;
    }

    public void setCoordinates(float[] coordinates) {
        this.coordinates = coordinates;
    }
}
```

### Класс *LatestWeather*

Наконец, у нас есть все необходимые компоненты для создания основного, корневого класса объекта для JSON ответа. Создайте новый класс `LatestWeather` внутри пакета `com.codewithjava21.weatherapp`. Он будет иметь три private-свойства для

идентификатора станции (id), расположения станции (geometry) и свойств (properties):

```
package com.codewithjava21.weatherapp;

public class LatestWeather {
    private String id;
    private Geometry geometry;
    private Properties properties;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public Geometry getGeometry() {
        return geometry;
    }

    public void setGeometry(Geometry geometryCoordinates) {
        this.geometry = geometryCoordinates;
    }

    public Properties getProperties() {
        return properties;
    }

    public void setProperties(Properties properties) {
        this.properties = properties;
    }
}
```

## Пересмотр контроллера погодного приложения

А теперь можно вернуться в наш класс `WeatherApplicationController`. Сначала создадим три метода. Первый метод будет иметь модификатор доступа `protected`, называться `getBucket` и возвращать тип `int`. Этот метод сгенерирует наш бакет месяцев на основе временной метки, которую мы получаем из NWS.

Наш метод, по сути, определит новый объект `ZonedDateTime` (требуется новый импорт из `java.time`) из `String timestamp`, которую мы считываем. Затем мы извлечем из него год `year` и месяц `month`, построим из них строку и вернем ее в виде целого числа:

```
protected int getBucket(Instant timestamp) {
ZonedDateTime date = ZonedDateTime.parse(timestamp.toString());
```

```
// разберем дату на год и месяц, чтобы создать бакет месяца
Integer year = date.getYear();
Integer month = date.getMonthValue();
StringBuilder bucket = new StringBuilder(year.toString());

if (month < 10) {
    bucket.append("0");
}
bucket.append(month);

return Integer.parseInt(bucket.toString());
}
```

Одна из возможных ошибок, которая может здесь произойти, заключается в том, что при построении целого числа 202307 (для июля 2023 года) оно может отображаться как 20237. Нам нужно убедиться, что в нем присутствует ведущий ноль. В конце концов, именно так мы представляем наши даты. Поэтому у нас есть оператор if, проверяющий, что если месяц меньше 10, то к нему добавляется (append) ноль.

> **Примечание.** Несмотря на то, что мы хотим защитить наши методы-аксессоры (геттеры) от доступа извне, нам нужно будет вызывать метод getBucket() из нашего пользовательского интерфейса. Поэтому мы присвоим ему статус protected, чтобы он был доступен из нашего пакета.

Следующий метод должен будет связать наш объект LatestWeather с классом WeatherEntity для базы данных. Это будет private-метод с именем mapLatestWeatherToWeatherEntity, принимающий параметры погодного объекта типа LatestWeather и строки stationId. Метод начнет работу с определения объекта returnVal класса WeatherEntity, сериализации timestamp и создания бакета:

```
private WeatherEntity mapLatestWeatherToWeatherEntity(
        LatestWeather weather, String stationId) {

    WeatherEntity returnVal = new WeatherEntity();

    // используем timestamp из ответа для определения даты
    Instant timestamp = weather.getProperties().getTimestamp();
    int bucket = getBucket(timestamp);

    // генерация первичного ключа
    WeatherPrimaryKey key = new WeatherPrimaryKey(stationId,
           bucket, timestamp);

    returnVal.setPrimaryKey(key);
    returnVal.setReadingIcon(weather.getProperties().getIcon());
    returnVal.setStationCoordinatesLatitude(
              weather.getGeometry().getCoordinates()[0]);
```

```
    returnVal.setStationCoordinatesLongitude(
              weather.getGeometry().getCoordinates()[1]);
    returnVal.setTemperatureCelsius(
              weather.getProperties().getTemperature().getValue());
    returnVal.setWindDirectionDegrees((int)
              weather.getProperties().getWindDirection().getValue());
    returnVal.setWindGustKMH(
              weather.getProperties().getWindGust().getValue());
    returnVal.setPrecipitationLastHour(
              weather.getProperties().getPrecipitationLastHour()
              .getValue());
```

Мы инстанцируем `WeatherPrimaryKey` в качестве ключа и установим его в `returnVal` вместе со всеми остальными свойствами. Наконец, пройдем по слоям облаков и соберем их в словарь, установим `cloudMap` в `returnVal` и вернем его:

```
    // обработка слоев облаков
    CloudLayer[] clouds = weather.getProperties().getCloudLayers();
    Map<Integer,String> cloudMap = new HashMap<>();

    for (CloudLayer layer : clouds) {
        // измерения возвращаются как float, но нам нужны int
        cloudMap.put((int)layer.getBase().getValue(),
        layer.getAmount());
    }

returnVal.setCloudCover(cloudMap);

return returnVal;
}
```

Для последнего метода создадим private-метод `mapWeatherEntityToWeatherReading`. В то время как наш предыдущий метод переводит необработанный ответ от конечной точки NWS в сущность Spring Data, этот метод переводит класс сущности в POJO. Таким образом, можно возвращать объекты класса `WeatherReading` и предотвратить необходимость для вызывающего приложения или пользователя знать что-либо о нашем классе `WeatherPrimaryKey`:

```
private WeatherReading mapWeatherEntityToWeatherReading(
        WeatherEntity entity) {

    WeatherReading returnVal = new WeatherReading();

    returnVal.setStationId(entity.getPrimaryKey().getStationId());
    returnVal.setMonthBucket(entity.getPrimaryKey().getMonthBucket());
    returnVal.setStationCoordinatesLatitude(
            entity.getStationCoordinatesLatitude());
```

```
    returnVal.setStationCoordinatesLongitude(
            entity.getStationCoordinatesLongitude());
    returnVal.setTimestamp(entity.getPrimaryKey().getTimestamp());
    returnVal.setTemperatureCelsius(entity.getTemperatureCelsius());
    returnVal.setWindSpeedKMH(entity.getWindSpeedKMH());
    returnVal.setWindDirectionDegrees(
            entity.getWindDirectionDegrees());
    returnVal.setWindGustKMH(entity.getWindGustKMH());
    returnVal.setReadingIcon(entity.getReadingIcon());
    returnVal.setVisibilityM(entity.getVisibilityM());
    returnVal.setPrecipitationLastHour(
            entity.getPrecipitationLastHour());
    returnVal.setCloudCover(entity.getCloudCover());

    return returnVal;
}
```

Метод `mapWeatherEntityToWeatherReading()` прост, поскольку все, что ему нужно, — это выполнить сопоставление свойств двух классов объектов один к одному.

Когда все методы и классы объектов созданы и собраны, можно приступать к написанию метода конечной точки сервиса:

```
@PutMapping("/latest/station/{stationid}")
public ResponseEntity<WeatherReading> putLatestData(
    @PathVariable(value="stationid") String stationId) {

    LatestWeather response = restTemplate.getForObject(
            "https://api.weather.gov/stations/" + stationId
            + "/observations/latest",
            LatestWeather.class);

    // Сопоставить последние показания с WeatherEntity
    WeatherEntity weatherEntity =
            mapLatestWeatherToWeatherEntity(response, stationId);

    // сохранить показания погоды
    weatherRepo.save(weatherEntity);

    WeatherReading currentReading = mapWeatherEntityToWeatherReading(
             weatherEntity);

    return ResponseEntity.ok(currentReading);
}
```

Теперь выполним `mvn clean install` в командной строке или из IDE.

**Примечание.** Во время выполнения `mvn clean install` может возникнуть проблема с автогенерируемым тестом в проекте. Поскольку мы его не используем, смело удаляйте файл:
src/test/java/com/codewithjava21/weatherapp/WeatherappApplicationTests.java.

Чтобы запустить наше приложение, необходимо нажать кнопку **Run** в IDE (при условии, что конфигурация запуска определена). Его также можно запустить из командной строки с помощью Maven:

```
mvn spring-boot:run
```

Как только сервис запустится, его можно протестировать с помощью конечной точки `helloworld`, как это делалось ранее. Или можно запустить операцию `PUT` с помощью `curl`:

```
curl -X PUT http://127.0.0.1:8080/weather/latest/station/kmsp
```

```
{"primaryKey":{"stationId":"kmsp","monthBucket":202307,"timestamp":
"2023-07-17T22:53:00Z"},"readingIcon":"https://api.weather.gov/icons/land/day/
bkn?size=medium","stationCoordinatesLatitude":-
93.22,"stationCoordinatesLongitude":44.88,
"temperatureCelsius":23.3,"windDirectionDegrees":330,"windSpeedKMH":0.0,
"windGustKMH":0.0,"visibilityM":0,"precipitationLastHour":0.0,
"cloudCover":{"1520":"FEW","2130":"8KN"}}
```

При вызове этой конечной точки службы данные о погоде:

- создаются на основе последних данных, полученных от NWS для станции `kmsp`;
- сохраняются в нашей базе данных Astra.

Если перейти на вкладку **CQL session** в Astra, можно получить запрос к строке, которая только что была записана:

```
token@cqlsh> use weatherapp ;
token@cqlsh:weatherapp> SELECT station_id, reading_timestamp, temperature_c FROM
weather_by_station_by_month ;
```

```
station_id     | reading_timestamp                  | temperature_c
---------------+------------------------------------+-----------------
       kmsp    | 2023-07-17 22:53:00.000000+0000    | 23.3
```

Также напишем конечную точку для получения последней записи о погоде по идентификатору станции (`StationId`) и бакету месяца (`MonthBucket`). Наш метод будет называться `getLatestData` и обслуживаться на конечной точке сервиса `/latest/station{stationid}/month/{month}`, как указано в аннотации `@GetMapping`. Метод будет просто вызывать пользовательский метод запроса в нашем `WeatherRepository`, проверять наличие null и возвращать результат:

```
@GetMapping("/latest/station/{stationid}/month/{month}")
public ResponseEntity<WeatherReading> getLatestData(
```

```
        @PathVariable(value="stationid") String stationId,
        @PathVariable(value="month") int monthBucket) {

    WeatherEntity recentWeather =
            weatherRepo.findByStationIdAndMonthBucket(
                stationId, monthBucket);

    WeatherReading currentReading = mapWeatherEntityToWeatherReading(
            recentWeather);

    if (currentReading != null) {
        return ResponseEntity.ok(currentReading);
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

После запуска нашего сервиса можно вызвать эту конечную точку с помощью `curl` с операцией `GET`:

```
curl -X GET http://127.0.0.1:8080/weather/latest/station/kmsp/month/202307
```

```
{"primaryKey":{"stationId":"kmsp","monthBucket":202307,"timestamp":"2023-07-
18T00:53:00Z"},"readingIcon":"https://api.weather.gov/icons/land/night/
sct?size=medium","stationCoordinatesLatitude":-
93.22,"stationCoordinatesLongitude":44.88,"temperatureCelsius":21.7,
"windDirectionDegrees":310,"windSpeedKMH":0.0,"windGustKMH":0.0,
"visibilityM":0,"precipitationLastHour":0.0,"cloudCover":
{"1520":"FEW","2130":"SCT"}}
```

Теперь, когда завершающая версия погодного сервиса работает, можно переходить к созданию простого фронт-энда.

## Создание пользовательских веб-интерфейсов

Проектирование и создание пользовательских веб-интерфейсов может быть сложной задачей. Конечный результат часто представляет собой комбинацию множества языков и фреймворков, включая JavaScript, React.js и CSS. К счастью, фреймворк Vaadin позволяет Java-разработчикам быстро создавать простые и функциональные пользовательские интерфейсы.

### Пересмотр pom.xml

На этом этапе пришло время вернуться к файлу pom.xml и раскомментировать зависимость `dependency` для `vaadin-spring-boot-starter`:

```
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-spring-boot-starter</artifactId>
</dependency>
```

## Вид погодного приложения

Теперь займемся пользовательским интерфейсом для нашего погодного приложения. Для этого необходимо создать новый класс `WeatherMainView` внутри пакета `com.codewithjava21.weatherapp`. Наш класс будет иметь несколько импортов, в том числе десять из фреймворка Vaadin:

```
package com.codewithjava21.weatherapp;

import com.vaadin.flow.component.button.Button;
import com.vaadin.flow.component.button.ButtonVariant;
import com.vaadin.flow.component.Component;
import com.vaadin.flow.component.html.Image;
import com.vaadin.flow.component.orderedlayout.HorizontalLayout;
import com.vaadin.flow.component.orderedlayout.VerticalLayout;
import com.vaadin.flow.component.radiobutton.RadioButtonGroup;
import com.vaadin.flow.component.textfield.TextField;
import com.vaadin.flow.component.grid.Grid;
import com.vaadin.flow.router.Route;
import java.time.Instant;
import java.util.ArrayList;
import java.util.List;
import org.springframework.http.ResponseEntity;

@Route("")
public class WeatherMainView extends VerticalLayout {
```

После определения импортов необходимо добавить аннотацию `@Route` с пустой строкой в качестве параметра. Она сообщает нашему веб-серверу, что определенная этим классом страница находится в относительном корне веб-адреса. Когда приложение будет запущено, нужно просто перейти по адресу **http://127.0.0.1:8080/**, и мы увидим нашу страницу.

Наш класс также должен наследоваться от класса `VerticalLayout` из Vaadin. Поэтому его нужно добавить в конец определения нашего класса с помощью ключевого слова `extends`.

Далее, имеется несколько private-свойств, которые необходимо определить в нашем классе. Большинство свойств — это компоненты Vaadin, которые будут встроены в пользовательский интерфейс. Текстовые поля, радиокнопки, изображения и сетки данных, безусловно, помогут дополнить визуальную функциональность нашего приложения:

```
private Image iconImage = new Image();
private TextField stationId = new TextField("Statio ID");
private TextField month = new TextField("Year/Month");
private TextField dateTime = new TextField("Date/Time");
private TextField temperature = new TextField("Temperature");
private TextField windSpeed = new TextField("Wind Speed");
```

```
private TextField windDirection = new TextField("Wind Direction");
private TextField visibility = new TextField("Visibility");
private TextField precipitationLastHour = new TextField(
        "Precipitation 1 hour");
private RadioButtonGroup<String> unitSelector =
        new RadioButtonGroup<>();
private Grid<Cloud> cloudGrid = new Grid<>(Cloud.class);

private WeatherAppController controller;
private record Cloud(int elevation, String desc) {
}
```

Также определим private-свойство для нашего `WeatherAppController`, поскольку потребуется вызывать его сервисы. Кроме того, создадим запись с именем `Cloud`, чтобы было проще работать с данными облачного слоя.

Далее создадим конструктор нашего класса. Его единственным параметром будет `WeatherAppRepository`, поскольку он нужен для объявления экземпляра контроллера. Также установим значения по умолчанию для станции `kmsp` и сгенерируем `monthBucket` для текущей даты. После этого можно настроить `cloudGrid` (не забудьте задать соответствующие размеры для столбцов и сетки). Мы также можем напрямую указать наши свойства `elevation` и `desc` из записи `Cloud`:

```
public WeatherMainView(WeatherAppRepository repo) {
    controller = new WeatherAppController(repo);

    // устанавливаем значения по умолчанию
    Integer monthBucket = controller.getBucket(Instant.now());
    month.setValue(monthBucket.toString());
    stationId.setValue("kmsp");

    // конфигурируем сетку
    cloudGrid.addColumn(Cloud::elevation)
        .setWidth("100px")
        .setHeader("Elevation");
    cloudGrid.addColumn(Cloud::desc)
        .setWidth("150px")
        .setHeader("Description");
    cloudGrid.setWidth("250px");
    cloudGrid.setHeight("250px");

    // составляем макет
    add(buildControls());
    add(buildStationDataView());
    add(buildTempPrecipView());
    add(buildCloudVisibilityView());
}
```

В конце конструктора выполняется сборка макета страницы. Поскольку наш класс по умолчанию наследуется от класса `VerticalLayout` из `Vaadin`, любые вызовы метода `add()` будут располагать возвращаемые компоненты вертикально. Это означает, что любое горизонтальное проектирование должно быть выполнено в методах `buildControls()`, `buildStationDataView()`, `buildTempPrecipView()` и `buildCloudVisibilityView()`, которые рассмотрим далее.

### Методы горизонтального построения

Сначала создадим private-метод `buildControls`. У него не будет никаких параметров, и он будет возвращать тип класса `Component`. В каждом из наших методов построения мы начнем с создания нового объекта `HorizontalLayout` с именем `layout`. Мы будем использовать объект `layout` и возвращать его с нашими горизонтально расположенными компонентами:

```
private Component buildControls() {
    HorizontalLayout layout = new HorizontalLayout();

    Button queryButton = new Button("Refresh");
    queryButton.addThemeVariants(ButtonVariant.LUMO_PRIMARY);
    layout.add(queryButton, buildUnitRadio());

    queryButton.addClickListener(click -> {
        refreshData();
    });

    return layout;
}
```

В этом конкретном методе мы создаем элементы управления для нашего пользовательского интерфейса. К ним относится кнопка **Refresh**, которая будет вызывать наш метод `getLatestData()`/конечную точку на нашем контроллере (и, в конечном счете, выполнять запрос к нашей базе данных). В дополнение к назначению одной из тем кнопок Vaadin, мы также создадим слушателя (listener). Метод `addClickListener()`, по сути, говорит Vaadin, что каждый раз, когда кнопка нажимается, она должна вызывать метод `refreshData()` (который создадим позже). Метод `add()` макета — это место, где мы добавляем нашу кнопку `queryButton` в макет, а также результат нашего метода `buildUnitRadio()` (который создадим позже).

Метод `buildUnitRadio` создает селектор/радиокнопку[1]. Эта радиокнопка позволит переключать отображение между метрическими (`Metric`) и (`Imperial`) имперскими единицами измерения. По сути, мы обозначим ее как `Units` и определим ее элементы как `Celsius/Metric` и `Fahrenheit/Imperial`.

---

[1] Радиокнопка (от англ. *radio button*), или переключатель, — элемент интерфейса, который позволяет пользователю выбрать одну опцию из предопределённого набора. — *Прим. ред.*

Также мы установим начальное значение по умолчанию как Cesius/Metric:

```
private Component buildUnitRadio() {
    HorizontalLayout layout = new HorizontalLayout();

    unitSelector.setLabel("Units");
    unitSelector.setItems("Celsius/Metric", "Fahrenheit/Imperial");
    unitSelector.setValue("Celsius/Metric");

    unitSelector.addValueChangeListener(click -> {
        refreshData();
    });

    layout.add(unitSelector);

    return layout;
}
```

У кнопки unitSelector также будет определен слушатель. Каждый раз, когда она будет нажата (независимо от конечного значения), будет вызываться метод refreshData().

Далее создадим метод buildStationView. Этот метод соберет данные для нашей метеостанции, а также текущее изображение иконки и компоненты времени. Он относительно прост, по сравнению с большинством других. По сути, необходимо добавить наши private-компоненты stationId, month, iconImage и dateTime в HorizontalLayout и вернуть его в качестве результата:

```
private Component buildStationDataView() {
    HorizontalLayout layout = new HorizontalLayout();

    layout.add(stationId, month, iconImage, dateTime);

    return layout;
}
```

Далее можно реализовать метод buildTempPrecipView. Этот метод создает визуальные элементы для данных о текущей температуре, ветре и осадках:

```
private Component buildTempPrecipView() {
    HorizontalLayout layout = new HorizontalLayout();

    layout.add(temperature, precipitationLastHour,
        windSpeed, windDirection);

    return layout;
}
```

Завершающий build-метод[2] — это метод `buildCloudLayerView`. Он просто добавляет `cloudGrid` и текстовое поле `visibility` в `HorizontalLayout`:

```
private Component buildCloudVisibilityView() {
    HorizontalLayout layout = new HorizontalLayout();

    layout.add(cloudGrid, visibility);

    return layout;
}
```

Прежде чем мы продолжим, нам понадобится несколько private-методов для вычисления преобразований между метрическими и имперскими единицами:

```
private float computeFahrenheit(float celsius) {
    return (celsius * 9 / 5) + 32;
}

private float computeMiles(float kilometers) {
    return (kilometers * 1.609F);
}

private int computeFeet(int meters) {
    return (int)(meters * 3.281F);
}

private float computeInches(float millimeters) {
    return millimeters / 25.4F;
}
```

Также потребуется метод для преобразования направления ветра из градусов в строку. Это позволит присвоить текстовому полю `windDirection` направление по компасу, основанное на значении в диапазоне 360 градусов:

```
private String convertWindDirection(Integer degrees) {
    StringBuilder returnVal = new StringBuilder();

    if ((degrees > 338 && degrees <= 360) ||
            degrees >= 0 && degrees < 23) {
        returnVal.append("North");
    } else if (degrees > 22 && degrees < 68) {
        returnVal.append("Northeast");
    } else if (degrees > 67 && degrees < 113) {
        returnVal.append("East");
    } else if (degrees > 112 && degrees < 158) {
        returnVal.append("Southeast");
```

[2] Используется в шаблоне Builder для генерации неизменного объекта по завершению всех необходимых параметров. — *Прим. ред.*

```
    } else if (degrees > 157 && degrees < 203) {
        returnVal.append("South");
    } else if (degrees > 202 && degrees < 248) {
        returnVal.append("Southwest");
    } else if (degrees > 247 && degrees < 293) {
        returnVal.append("West");
    } else {
        returnVal.append("Northwest");
    }

    return returnVal.toString();
}
```

Наконец, можно создать метод `refreshData`. Этот метод будет выполнять вызов нашего сервисного слоя, связывать данные ответа с пользовательским интерфейсом и реализовывать любую необходимую нам дополнительную логику. Сначала вызовем метод `getLatestData()` на нашем контроллере. Затем установим данные для объекта класса `WeatherReading` с именем `latestWeather`, а затем присвоим большинство его свойств локальным переменным (так с ними проще работать):

```
private void refreshData() {
    ResponseEntity<WeatherReading> latest = controller.getLatestData(
        stationId.getValue(), Integer.parseInt(month.getValue()));
    WeatherReading latestWeather = latest.getBody();

    Instant time = latestWeather.getTimestamp();
    Float temp = latestWeather.getTemperatureCelsius();
    Float windSpd = latestWeather.getWindSpeedKMH();
    String iconURL = latestWeather.getReadingIcon();
    Integer windDir = latestWeather.getWindDirectionDegrees();
    Integer visib = latestWeather.getVisibilityM();
    Float precip = latestWeather.getPrecipitationLastHour();
```

Далее проверим значение нашей радиокнопки единиц измерения. Если оно не равно `Celsius/Metric`, то переведем температуру, скорость ветра, видимость и осадки в имперские единицы. После этого установим значения для большинства компонентов пользовательского интерфейса:

```
if (!unitSelector.getValue().equals("Celsius/Metric")) {
    temp = computeFahrenheit(temp);
    windSpd = computeMiles(windSpd);
    visib = computeFeet(visib);
    precip = computeInches(precip);
}

temperature.setValue(temp.toString());
windSpeed.setValue(windSpd.toString());
```

```
precipitationLastHour.setValue(precip.toString());
dateTime.setValue(time.toString());
iconImage.setSrc(iconURL);
```

Далее мы убедимся, что скорость ветра и видимость больше нуля. Если погодные данные свидетельствуют, что видимость равна нулю, мы отобразим текст `Unlimited`. Кроме того, скорость ветра нужно показывать только в том случае, если она больше нуля. Нет необходимости преобразовывать градусы в направление по компасу, если ветер равен нулю:

```
if (visib > 0) {
    visibility.setValue(visib.toString());
} else {
    visibility.setValue("Unlimited");
}

if (windSpd > 0) {
    windDirection.setValue(convertWindDirection(windDir));
    windDirection.setVisible(true);
} else {
    windDirection.setVisible(false);
}
```

Наконец, обработаем облачные слои. Пройдемся по ним и (после проверки необходимости преобразования в имперские единицы) создадим список с использованием записи `Cloud`. Полученный список облаков затем легко установить для заполнения сетки:

```
    List<Cloud> clouds = new ArrayList<>();

    for (int key : latestWeather.getCloudCover().keySet()) {

        String description = latestWeather.getCloudCover().get(key);
        if (!unitSelector.getValue().equals("Celsius/Metric")) {
            key = computeFeet(key);
        }

        Cloud cloud = new Cloud(key,description);
        clouds.add(cloud);
    }

    cloudGrid.setItems(clouds);
}
```

### Запуск погодного приложения

Когда все готово, можно запускать наше приложение. Нажмите кнопку **Run** в IDE или выполните команду `mvn spring-boot:run` из командной строки.

Когда приложение будет запущено, перейдите в браузере по адресу **http://127.0.0.1:8080/**. После этого должна появиться наша веб-страница, большинство данных на которой будут пусты. Нажмите на кнопку **Refresh** и посмотрите на результат, показанный на рис. 8.7.

Если при нажатии кнопки **Refresh** ничего не происходит, проверьте таблицу, чтобы убедиться, что в ней есть данные. Возможно, потребуется выполнить операцию PUT (из командной строки), чтобы получить в таблицу строку с последними данными о погоде:

```
curl -X PUT http://127.0.0.1:8080/weather/latest/station/kmsp
```

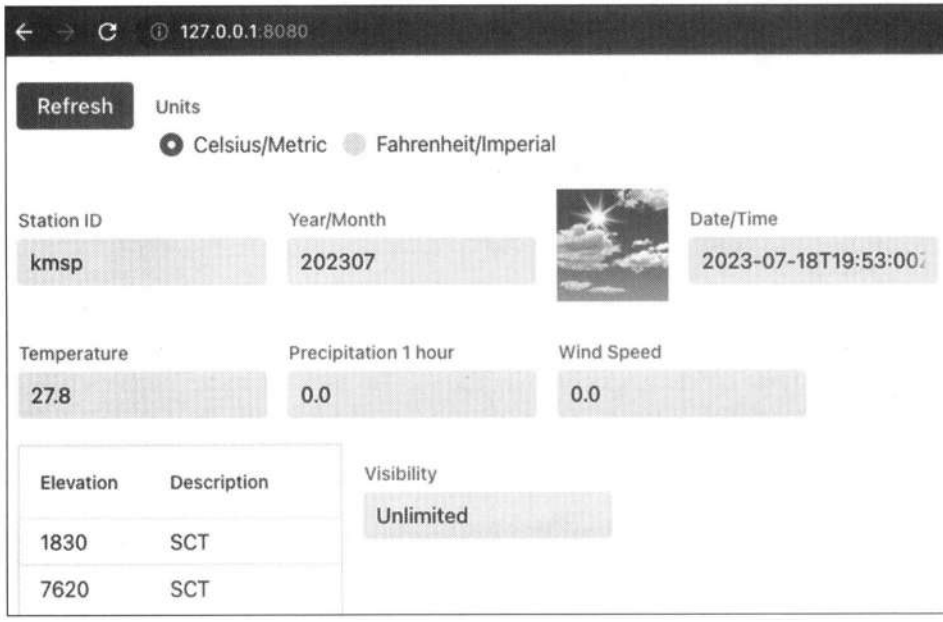



**Рис. 8.7.** Наше погодное приложение работает!

Смысл этого раздела заключался в том, чтобы показать, как построить простой внешний пользовательский интерфейс для приложения с использованием только Java. Vaadin отлично справился с нашими требованиями по отображению данных о погоде.

# Заключение

В этой главе мы обсудили различные аспекты создания веб-приложений. Мы использовали три фреймворка — Spring Boot, Spring Data и Vaadin — чтобы облегчить эту задачу. В итоге мы смогли создать функционирующее погодное веб-приложение. Мы также применили на практике идеи свободного взаимодействия. Вместо того чтобы создавать гигантское, взаимозависимое, монолитное приложе-

ние, мы следовали паттерну проектирования MVC, благодаря которому функциональные части нашего приложения были отделены друг от друга.

Важно понимать, что внешнее веб-представление должно заботиться только о взаимодействии с данными и их отображении для пользователя. Модель данных или DAL должны заботиться только о хранении и получении данных. Эти два слоя не должны ничего знать друг о друге. Именно так можно создавать приложения, компоненты которых не зависят друг от друга. Независимость этих уровней приложений также позволяет независимо масштабировать их базовую инфраструктуру.

В следующей главе мы рассмотрим работу с графикой в Java. Изучим подходы к рисованию простых фигур, таких как линии, квадраты и круги. А также продемонстрируем методы анимирования фигур.

## Важно помнить

- Restful-сервисы — это легкие, целевые, масштабируемые и слабосвязанные сервисы.
- Обычными Restful-операциями HTTP являются `GET`, `POST`, `PUT` и `DELETE`.
- Запросы и ответы Restful имеют отдельные компоненты, такие как тело и заголовок.
- Разработка приложения по паттерну MVC — это отличный способ отделить модель, представление и контроллер друг от друга.
- Многие базы данных имеют различные уровни разрешений для разных функций. Помните, что Spring Data обычно требуется разрешение на изменение схемы базы данных.
- В Apache Cassandra моделирование разделов с помощью компонента времени или бакета — отличная стратегия, чтобы не дать им вырасти слишком большими.
- Зачастую хорошей идеей является создание простого старого Java-объекта (Plain Old Java Object, POJO) вместе с классом сущности SpringData. Таким образом, POJO можно возвращать вызывающим сервисам и пользователям, не раскрывая базовую модель данных.
- Классы Vaadin обычно наследуются от класса `VerticalLayout` и собирают основные компоненты в несколько классов `HorizontalLayout`.
- Некоторые компоненты Vaadin могут быть сконфигурированы со слушателями, которые будут выполнять определенные функции при обнаружении действий пользователя (клик мыши или нажатие клавиши).

# Глава 9
# Графика в Java

## Введение

В этой главе мы узнаем, как разрабатывать графические программы на Java. Создание графики — это одновременно одна из самых увлекательных и сложных областей программирования. Многие люди изучают графику, потому что хотят заниматься видеоиграми. Отметим, что в ней много логики и математики, что делает ее отличным учебным пособием!

Здесь мы сможем проявить творческий подход и повеселиться во время обучения. Начнем с простых линий и фигур и перейдем к анимации. В конце главы мы создадим клон классической игры Atari конца 1970-х годов.

## Структура

В этой главе мы рассмотрим следующие темы.

- Простая графика с помощью AWT и Swing.
- Анимация.
- Java Breakout.

## Цели

В этой главе мы сформируем начальный уровень знаний об использовании простых графических техник. Для достижения этого мы поставили перед собой следующие задачи:

- научиться рисовать простые графические элементы;
- понять, как анимировать видимые объекты;
- узнать, как управлять скоростью и динамикой работы приложения с помощью виртуального потока;

- научиться строить логику для обработки столкновений фигур и пользовательского ввода.

# Простая графика с помощью AWT и Swing

Для реализации графики в Java мы будем использовать классы из библиотек Abstract Window Toolkit (AWT) и Swing. Библиотека AWT существует уже давно; библиотека Swing улучшает ее и предлагает некоторые дополнительные функции.

> **Примечание.** Библиотека JavaFX была создана для улучшения возможностей AWT и Swing. Однако читателям будет полезно познакомиться с AWT и Swing, поскольку многие предприятия сегодня все еще используют код этих библиотек.

## Класс SimpleDraw

Давайте создадим новый класс Java с именем `SimpleDraw`. Убедитесь, что он находится в пакете с именем `chapter9` и что у него есть метод `main`. Этот класс будет нуждаться в единственном импорте — классе `JFrame` из Swing:

```
package chapter9;

import javax.swing.JFrame;

public class SimpleDraw {

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

Внутри метода `main()` создадим новый объект `JFrame()` с именем `frame`. Затем зададим начальные свойства объекта `frame`:

```
JFrame frame = new JFrame();

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setTitle("Simple Drawings");
```

Эти свойства довольно просты. Нам нужно, чтобы приложение останавливалось при закрытии окна, а заголовку мы хотим присвоить имя `Simple Drawings`.

## Класс *MyPanel*

Далее создадим новый класс `MyPanel` внутри пакета `chapter9`. У него не должно быть метода `main`. В классе будет несколько импортов, включая класс `JPanel` из Swing, класс `ImageIO` из библиотеки ImageIO, а также классы `BufferedImage`, `Color`, `Dimension`, `Graphics` и `Graphics2D` из AWT.

Наш класс также должен наследоваться от класса `JPanel` и начинать свою работу с генерации `serialVersionUID`:

```
package chapter9;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;

import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;

import javax.swing.JPanel;

public class MyPanel extends JPanel {

    private static final long serialVersionUID =
            5433149762760327082L;

    private BufferedImage logo;
```

Также необходима private-переменная класса с именем `logo` типа `BufferedImage`. Она будет использоваться для рисования PNG-изображения на нашей панели.

Далее необходимо создать конструктор для класса `MyPanel`. Этот конструктор будет выполнять некоторые необходимые задачи для создания нашей `JPanel`. Сюда входит установка размера окна и цвета фона, а также разрешение окну принимать модальный фокус (быть выбранным):

```
public MyPanel() {
    this.setPreferredSize(new Dimension(800, 600));
    this.setBackground(Color.black);
    this.setFocusable(true);

    logo = loadImage("data/bpb.png", 100, 100);
}
```

> **Примечание.** Чтобы метод `loadImage()` работал, файл изображения bpb.png должен находиться в каталоге data/ основного проекта. Этот файл можно загрузить из доступных для этой книги онлайн-материалов,.

В этой программе мы рассмотрим пример, который отображает предварительно созданное PNG-изображение. Для этого необходимо создать private-метод `loadImage` (на него есть ссылка в коде выше). Метод `loadImage()` должен принимать

путь (String) к каталогу с файлом, а также параметры ширины width и высоты height для масштабирования scale изображения image:

```
private BufferedImage loadImage(String imagePath, int width,
        int height) {
    BufferedImage scaledImage = null;

    try {
        // загрузка изображения
        File imgFile = new File(imagePath);
        BufferedImage image = ImageIO.read(imgFile);
        // масштабирование изображения
        scaledImage = new BufferedImage(width, height,
                image.getType());
        Graphics2D g2 = scaledImage.createGraphics();
        g2.drawImage(image, 0, 0, width, height, null);
        g2.dispose();
    } catch (IOException ex) {
        ex.printStackTrace();
    }

    return scaledImage;
}
```

**Примечание.** Масштабирование изображения важно, потому что мы хотим убедиться, что изображение поместится на нашей панели вместе с другими графическими элементами.

Метод loadImage() использует try/catch для обработки случая, когда файл изображения не может быть найден. Изображение считывается с диска в объект BufferedImage с именем image. Затем оно масштабируется путем установки нового BufferedImage с именем scaledImage, используемого для создания локального объекта Graphics2D с именем g2. Затем вызывается метод drawImage() объекта g2 для отрисовки изображения на scaledImage. Затем возвращается scaledImage, чтобы он сохранился в переменной logo и мог быть легко нарисован на нашей панели позже.

Теперь, когда конструктор и загрузчик изображений готовы, можно создать public-метод paintComponent с единственным параметром в виде объекта Graphics. Метод paintComponent() наследуется, поэтому мы переопределяем его здесь. Начнем с вызова того же метода в нашем суперклассе. Это позволит убедиться, что заданные в конструкторе параметры выполняются:

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D)g;
```

Также преобразуем наш объект Graphics в объект Graphics2D под названием g2. Библиотека g2 содержит большинство инструментов рисования, которые мы будем использовать.

Давайте начнем с отрисовки одной линии голубого цвета:

```
g2.setColor(Color.CYAN);
g2.drawLine(100, 100, 700, 500);
```

Метод `setColor()` позволяет создать новый цвет или использовать любой из двенадцати предопределенных цветов, приведенных в табл. 9.1.

**Таблица 9.1.** Список предопределенных цветов в классе AWT Color

| Цвет | Цвет | Цвет |
|---|---|---|
| Black | Green | Pink |
| Blue | Light Gray | Red |
| Cyan | Magenta | White |
| Dark Gray | Orange | Yellow |

Затем можно нарисовать линию, которая будет проведена от координат `100,100` до `700,500`. Важно отметить, что координаты панели начинаются с `0,0` в левом верхнем углу окна. Поэтому линия будет начинаться на `100` пикселей вправо и на `100` пикселей вниз. Далее она будет продолжаться в направлении вправо вниз, пока не достигнет `700` пикселей вправо и `500` пикселей вниз, как показано на рис. 9.1.

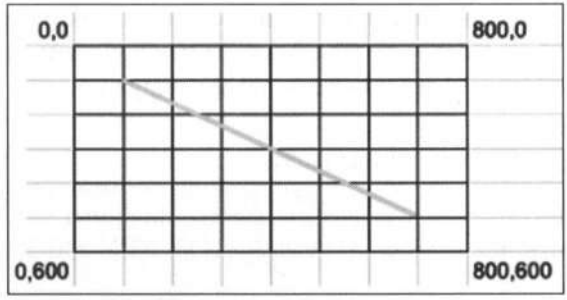



**Рис. 9.1.** Визуальное изображение координат `frame` и нашей линии голубого цвета

Теперь давайте вернемся к классу `SimpleDraw`. Добавьте следующий код в конец метода `main()`:

```
MyPanel panel = new MyPanel();
frame.add(panel);
frame.pack();
frame.setVisible(true);
```

Здесь выполняется создание объекта класса `MyPanel` с именем `panel`. Затем добавляем `panel` в объект `frame`. Кроме того, вызовем метод `pack()` для `frame`, так как это поможет убедиться, что все содержимое внутри `frame` имеет нужный размер. Наконец, дадим `frame` команду быть видимым.

В этот момент, если запустить код, должно появиться окно размером 800×600 пикселей с черным фоном. Голубая линия должна начинаться в левом верхнем углу и идти к правому нижнему углу. Программу можно остановить, просто закрыв окно.

Давайте вернемся к классу `MyPanel`. После рисования голубой линии нарисуем пурпурную линию, которая будет параллельна ей:

```
g2.setColor(Color.MAGENTA);
g2.drawLine(100, 200, 700, 600);
```

> **Примечание.** Отображаемый элемент может поддерживать операции рисования с координатами, превышающими его максимальный размер. Однако, несмотря на то, что результаты выходят за пределы элемента и не доступны для просмотра, они все равно занимают память.

Кроме того, можно рисовать квадраты и прямоугольники с помощью метода `drawRect()`. Параметрами являются координата x, координата y, а затем желаемые ширина и высота фигуры. Давайте нарисуем зеленый квадрат размером 100×100 пикселей:

```
g2.setColor(Color.GREEN);
g2.drawRect(600, 100, 100, 100);
```

Этот метод должен нарисовать квадрат зеленого цвета. Если бы потребовалось заполнить этот квадрат, можно было бы использовать вызов метода `fillRect()`:

```
g2.fillRect(600, 100, 100, 100);
```

Также можно нарисовать квадрат или прямоугольник с закругленными углами. Параметры метода `fillRoundRect()` те же самые, за исключением двух новых параметров, указывающих ширину `width` и высоту `height` окружности:

```
g2.setColor(Color.BLUE);
g2.fillRoundRect(400, 100, 100, 100, 50, 50);
```

> **Примечание.** Если установить значения высоты, ширины, высоты окружности и ширины окружности для метода `fillRoundRect()` в одно и то же значение, то будет нарисован круг!

Как указано в примечании выше, метод `fillRoundRect()` может нарисовать окружность. Однако гораздо проще рисовать окружности с помощью методов `drawOval()` и `fillOval()`. Оба метода принимают четыре параметра: координату x (`coordX`), координату y (`coordY`), ширину `width` и высоту `height`. Чтобы нарисовать круг (а не овал), ширина и высота должны быть установлены равными требуемому диаметру круга в пикселях. Сейчас мы нарисуем фиолетовый круг диаметром `100` пикселей в координатах `100,400`:

```
g2.setColor(new Color(128,0,192));
g2.drawOval(100, 400, 100, 100);
```

Чтобы нарисовать заполненный круг, можно заменить метод `drawOval()` на метод `fillOval()`:

```
g2.fillOval(100, 400, 100, 100);
```

Обратите внимание, что фиолетовый не входит в число предопределенных цветов, поэтому нам пришлось создать его, инстанцировав новый объект класса `Color`. У него есть несколько различных конструкторов, но мы использовали конструктор с тремя целочисленными параметрами для значений от `0` до `255` для красного, зеленого и синего цветов соответственно. Для нашего оттенка фиолетового мы использовали следующие значения:

- красный: `128`;
- зеленый: `0`;
- синий: `192`.

> **Примечание.** Если в проекте требуется не один из двенадцати предопределенных цветов, то при генерации нового цвета RGB со значениями от 0 до 255 для красного, зеленого и синего можно получить более 16,5 миллиона возможных цветов.

Список распространенных цветов и их RGB-кодов можно найти в приложениях.

Далее рассмотрим изображение, которое мы загрузили и масштабировали ранее, и с его помощью продемонстрируем, как нарисовать изображение в формате PNG или JPG на нашей панели:

```
g2.drawImage(logo, 250, 400, null);
```

Метод `drawImage()` класса `Graphics2D` принимает в качестве параметров объект `BufferedImage`, координату X (`coordX`), координату Y (`coordY`) и `ImageObserver`. Мы не будем использовать объект `ImageObserver`, поэтому просто установим его в `null`.

В результате выполнения кода должно получиться нечто похожее на то, что показано на рис. 9.2.

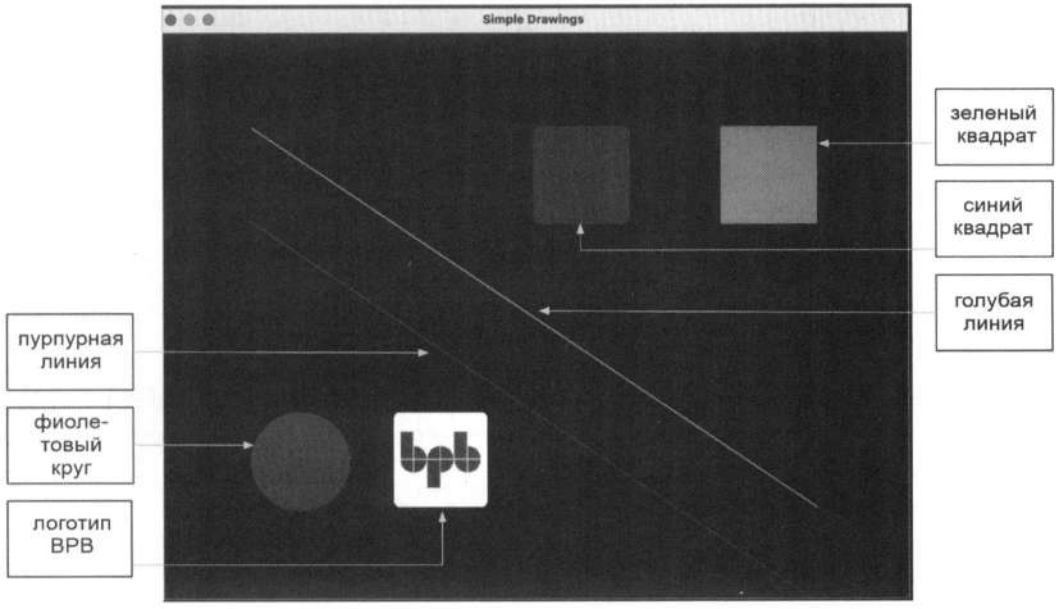



**Рис. 9.2.** Изображение, сгенерированное нашей программой SimpleDraw

# Анимация

Теперь, когда понятны основы рисования простых фигур на `JPanel`, научимся их анимировать. Здесь мы создадим простую анимацию планет нашей Солнечной системы, вращающихся вокруг Солнца.

## Класс *Planet*

Для начала создадим новый POJO-класс с именем `Planet`. Этот класс должен находиться внутри пакета `chapter9` и не должен иметь метода `main`. Классу `Planet` потребуется импортировать класс `Color` из библиотеки AWT, и он будет иметь следующие private-свойства:

- `int coordX`
- `int coordY`
- `int degree`
- `int diameter`
- `int distance`
- `int speed`
- `Color color`

Создадим класс `Planet`:

```
package chapter9;

import java.awt.Color;

public class Planet {

    private int coordX;
    private int coordY;
    private int degree;
    private int diameter;
    private int distance;
    private int speed;

    private Color color;
```

Далее создадим конструктор `Planet`. Он должен принимать четыре параметра: `Color` из AWT и три целых числа для диаметра `diameter`, орбиты `orbit` и скорости `speed` соответственно:

```
public Planet(Color color, int diameter, int orbit, int speed) {

    this.color = color;
    this.diameter = diameter;
    this.speed = speed;
```

```
    this.distance = orbit * 100;
    this.degree = 0;
}
```

Мы будем использовать `color`, `diameter` и `speed` для установки одноименных свойств класса. Все планеты будут начинаться с `0` градусов (из `360` для полной орбиты). Параметр `orbit` будет целым числом, представляющим собой последовательный индекс траектории орбиты вокруг Солнца, с возможными значениями от `1` до `4`. Чем ниже орбита, тем ближе планета к Солнцу. Мы будем использовать `orbit` для вычисления расстояния `distance` до Солнца, путем умножения ее на 100 пикселей.

Нам понадобится новый private-метод для вычисления следующих координат X,Y планеты на основе ее текущего градуса на круговой орбите. Мы назовем этот метод `computeNewXY`. Для начала преобразуем градусы в радианы, умножив свойство `degree` на Pi/180. Затем можно легко вычислить новые координаты X,Y, умножив косинус (для X) или синус (для Y) наших радианов на свойство `distance`:

```
private void computeNewXY() {
    double radians = degree * Math.PI / 180;
    coordX = (int) (distance * Math.cos(radians));
    coordY = (int) (distance * Math.sin(radians));
}
```

Обратите внимание, что координаты X,Y вычисляются относительно другой позиции, которая будет рассмотрена в методе `computeNewXY()`. Далее, нам нужен private-метод для вычисления радиуса планеты. Это легко сделать, поскольку у нас есть диаметр планеты:

```
private int getRadius() {
    return diameter / 2;
}
```

Нашим первым public-методом будет метод `update`. Этот метод не принимает никаких параметров и будет вызываться для вычисления следующего перемещения планеты по ее орбите:

```
public void update() {
    degree -= speed;

    if (degree < 0) {
        degree += 360;
    }

    computeNewXY();
}
```

Фактически, планета будет двигаться против часовой стрелки. Следовательно, мы будем вычитать свойство `speed` из текущего значения `degree`. Метод `update()` гаран-

тирует, что свойство degree всегда будет иметь корректное значение от 0 до 359. После вычисления нового градуса он вызывает метод computeNewXY(), который мы разработали выше.

Наконец, у нас есть геттеры для наших свойств. Нет необходимости создавать методы-сеттеры, поскольку не нужно изменять какие-либо свойства за пределами класса. Единственные геттеры, на которые стоит обратить внимание, — это методы getCoordX() и getCoordY(). Это связано с тем, что мы скорректировали наши координаты X,Y, чтобы учесть, что точка отрисовки отличается от центра. Поэтому нужно вычесть радиус из X и Y, чтобы правильно отцентрировать орбиту:

```
public int getCoordX() {
    return coordX - getRadius();
}

public int getCoordY() {
    return coordY - getRadius();
}
```

Для остальных свойств геттеры будут просто возвращать значение свойства, и мы завершаем работу с классом Planet.

## Класс *SolarSystem*

Создайте новый Java-класс с именем SolarSystem. Он должен находиться внутри пакета chapter9, и у него не должно быть метода main. Класс SolarSystem будет иметь семь импортов, включая JPanel из библиотеки Swing, List и ArrayList из библиотеки Util, а затем Color, Dimension, Graphics и Graphics2D из библиотеки AWT. Класс SolarSystem также будет наследоваться от класса JPanel и реализовывать интерфейс Runnable:

```
package chapter9;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;

import java.util.ArrayList;
import java.util.List;

import javax.swing.JPanel;

public class SolarSystem extends JPanel implements Runnable {
```

Реализация класса Runnable позволит нашим объектам класса SolarSystem работать как потоки. Потоки могут быть очень полезны в играх или графических приложе-

ниях, так как они дают разработчику больше контроля над поведением приложения, включая установку скорости обновления панели.

После генерации `serialVersionUID` мы определим несколько свойств. Во-первых, мы ограничим анимацию 60 кадрами в секунду (`frames per second`). Переменная называется `fPS` и инстанцируется как постоянное (с помощью ключевого слова `final`) целое число со значением 60. Мы также определим свойства для размера и середины панели, а также `List<Planet>` и `panelThread` типа `Thread`:

```
private static final long serialVersionUID = -6923126786235441890L;

private final int fPS = 60; // frames per second

private int panelWidth;
private int panelHeight;
private int middleWidth;
private int middleHeight;

private List<Planet> planetList;

private Thread panelThread;
```

> **Примечание.** Как платформенные, так и виртуальные потоки управляются классом `Thread`.

Теперь создадим два конструктора для класса `SolarSystem`. Один конструктор будет принимать ширину `width` и высоту `height` (как целые числа) для указания размера панели. Другой конструктор будет неаргументированным, он просто вызывает другой:

```
public SolarSystem() {
    this(1024,1024);
}

public SolarSystem(int width, int height) {
    panelWidth = width;
    panelHeight = height;
    middleWidth = panelWidth / 2;
    middleHeight = panelHeight / 2;

    panelThread = Thread.ofVirtual()
        .name("solarSystemThread")
        .unstarted(this);

    this.setPreferredSize(new Dimension(panelWidth, panelHeight));
    this.setBackground(Color.black);
    this.setFocusable(true);
    this.planetList = new ArrayList<>();

    // Меркурий
    planetList.add(new Planet(Color.DARK_GRAY, 20, 1, 5));
```

```
    // Венера
    planetList.add(new Planet(Color.GRAY, 48, 2, 4));
    // Земля
    planetList.add(new Planet(Color.BLUE, 50, 3, 3));
    // Марс
    planetList.add(new Planet(Color.RED, 25, 4, 2));
}
```

Наш главный конструктор начинает с установки свойств для размеров панели, включая середину. Затем он инстанцирует `panelThread` как виртуальный поток с именем `solarSystemThread` и устанавливает использование класса как `unstarted`. Далее устанавливаются размеры панели и цвет фона, а также модальный фокус окна. Наконец, он инстанцирует `planetList` и добавляет в него новые `Planets` (Меркурий, Венеру, Землю и Марс).

Далее определим метод `run`. Этот метод наследуется от класса `Thread`, и его необходимо переопределить. Метод `run()` довольно прост. Мы проверим, существует ли `panelThread` (с помощью метода `isAlive()`), а затем вызовем наши методы `update()` и `repaint()`. Наконец, вызовем метод `sleep()` из класса `Thread`, чтобы приостановить работу приложения на короткое время (одна шестидесятая секунды).

```
@Override
public void run() {
    while (panelThread.isAlive()) {

        update();
        repaint();

        try {
            Thread.sleep(1000/fPS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Далее создадим private-метод `update()`. Метод `update()` перебирает объекты `Planet` в `planetList` и вызывает их public-методы `update()`. При этом перед перерисовкой на панели пересчитывается положение каждой планеты:

```
private void update() {
    for (Planet planet : planetList) {
        planet.update();
    }
}
```

Далее создадим новый public-метод с именем `paintComponent`. Этот метод вызывается, когда происходит вызов метода `repaint()` из `AWT Component` (см. `run()` выше).

Аналогично тому, что мы делали в классе `MyPanel`, этот метод выполняет фактическое рисование и раскрашивание. Вызвав тот же метод в нашем суперклассе и инстанцировав объект `Graphics2D`, начнем с рисования Солнца:

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D)g;

    // отрисовка Солнца
    int diameter = 48;
    int radius = diameter / 2;

    g2.setColor(Color.YELLOW);
    g2.fillOval(middleWidth - radius, middleHeight - radius,
            diameter, diameter);

    // отрисовка планет
    for (Planet planet : planetList) {
        g2.setColor(planet.getColor());
        g2.fillOval(middleWidth + planet.getCoordX(),
        middleHeight + planet.getCoordY(),
        planet.getDiameter(), planet.getDiameter());
}

    g2.dispose();
}
```

После рисования Солнца мы переходим по списку `planetList` и используем данные о каждом объекте планеты, чтобы нарисовать его в новом положении. В завершение мы избавляемся от объекта `g2` класса `Graphics2D`.

Прежде чем перейти к нашему основному классу, необходимо создать последний метод. Метод `start()` позволит запустить `panelThread`, тем самым инициируя цикл, построенный в методе `run()` выше:

```
public void start() {
    panelThread.start();
}
```

Теперь код класса `SolarSystem` завершен, и можно переходить к классу `DrawPlanets`.

## Класс *DrawPlanets*

Чтобы собрать все это воедино, создадим новый Java-класс с именем `DrawPlanets`. Этот класс будет находиться в пакете `chapter9` и должен иметь метод `main`. Класс `DrawPlanets` будет иметь единственный импорт `JFrame` из библиотеки Swing:

```
package chapter9;

import javax.swing.JFrame;

public class DrawPlanets {
```

Наш метод main создает объект JFrame с именем frame. Как и в случае с SimpleDrawClass, установим свойства frame, чтобы он правильно закрывался и имел имя "Planet Orbits":

```
public static void main(String[] args) {
    JFrame frame = new JFrame();

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setTitle("Planet Orbits");

    SolarSystem panel = new SolarSystem();
    frame.add(panel);
    frame.pack();
    frame.setVisible(true);

    panel.start();
}
```

Затем создадим объект SolarSystem с именем panel, добавим panel в frame, вызовем метод pack() и обеспечим видимость. Наконец, вызовем метод start() панели, чтобы запустить виртуальный поток.

Запуск класса DrawPlanets должен привести к появлению окна с планетами, вращающимися вокруг Солнца (рис. 9.3).

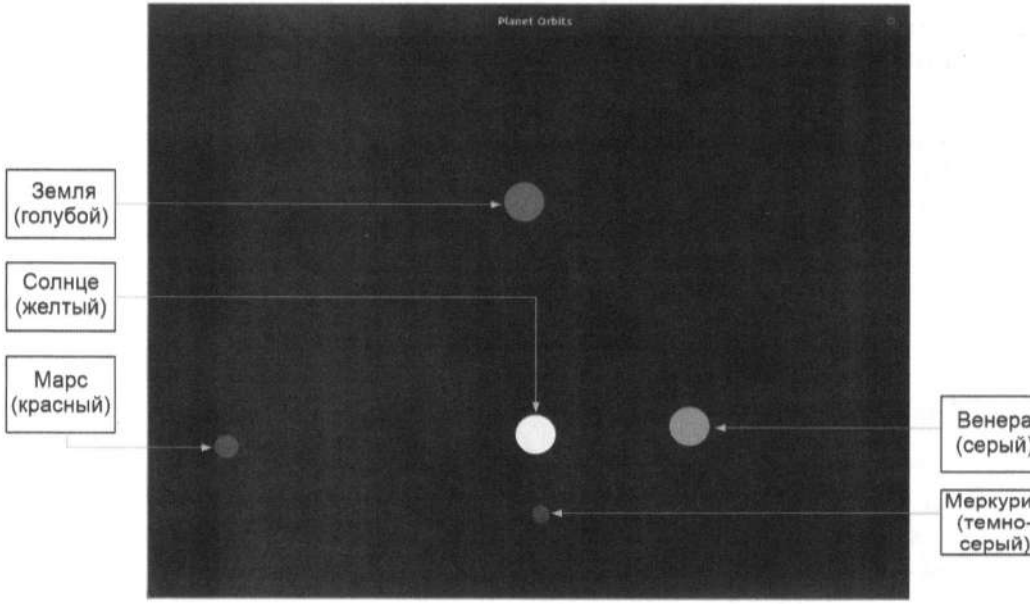



**Рис. 9.3.** Работа класса DrawPlanets, в котором показаны Земля, Марс, Венера и Меркурий, вращающиеся вокруг Солнца

# Java Breakout

Период конца 1970-х — начала 1980-х годов был, безусловно, интересным временем для индустрии видеоигр. Сначала появились аркадные игры, а вскоре после этого в домах появились первые игровые приставки. Основой для бума видеоигр того времени послужили классические игры, такие как Breakout от Atari.

> **История Breakout**
>
> Впервые Breakout была создана компанией Atari в 1976 году как аркадная игра. Проект был передан молодому игровому дизайнеру по имени Стив Джобс. Джобс описал игру своему другу, Стиву Возняку, и поручил ему разработку игры. Они закончили работу над игрой всего за четыре дня (*Hanson 2015*), и, по легенде, пара не спала во время работы над проектом. Несколько месяцев спустя Стив Джобс и Стив Возняк стали соучредителями компании, которую мы все знаем как Apple Inc.

Мы создадим клон Breakout под названием Java Breakout. Java Breakout — это игра для одного игрока, в которой он управляет бруском, чтобы направить мяч в массив разноцветных кирпичей. Цель игры заключается в том, чтобы все кирпичи оказались выбиты.

## pom.xml

Создадим новый простой проект Maven. Проект должен обладать следующими свойствами (`properties`):

- **Group ID:** `com.codewithjava21`
- **Artifact ID:** `javabreakout`
- **Name:** `JavaBreakout`

После создания проекта отредактируйте полученный файл pom.xml и добавьте в него следующие `properties`:

```
<properties>
    <java.version>21</java.version>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
</properties>
```

> **Примечание.** Секция `<properties>` — это элемент первого уровня, находящийся непосредственно внутри элемента `<project>`.

После создания проекта можно приступать к созданию классов приложения.

## Класс *Ball*

Нам нужен класс, который будет управлять поведением мяча. Для этого создадим новый Java-класс с именем `Ball` внутри пакета `javabreakout`. У этого класса не должно быть метода `main`. Класс `Ball` должен импортировать класс `Random` и устанавливать несколько свойств объекта:

```
package javabreakout;

import java.util.Random;

public class Ball {
    private final int maxSpeed = 8;
    private final int oneThird = maxSpeed / 3;

    private boolean movingUp;
    private boolean movingLeft;
    private int ballSizeOffset;
    private int ballX;
    private int ballY;
    private int ballSize;
    private int hSpeed;
    private int vSpeed;
```

Нам понадобятся целочисленные свойства `integer` для хранения размера, координат и скорости мяча. Также важно знать, в каком направлении движется мячик, поэтому создадим два булевых свойства `movingUp` и `movingLeft`. Нам также понадобится постоянное целое число `maxSpeed`, которое установим равным 8. Аналогично, нам понадобится постоянное целое число для одной трети `maxSpeed`, которое вычисляется как `maxSpeed`, деленное на значение 3.

> **Примечание.** Идея уменьшения скорости на одну треть была реализована до того, как было определено оптимальное значение `maxSpeed`, равное 8. Технически, в этом сценарии переменная `oneThird` будет фактически содержать значение, равное одной четверти от `maxSpeed`.

Конструктор класса `Ball` будет принимать параметры для размера, ширины панели, высоты кирпичей, а также высоты пространства над кирпичами. Последние три параметра помогут определить начальную позицию мяча. Его начальное положение должно быть в центре экрана и ниже кирпичей. Мы позаботимся об этих вычислениях в конструкторе:

```
public Ball(int size, int panelWidth, int brickHeightx8,
        int brickBuffer) {
    ballSize = size;
    ballSizeOffset = (size / 2) + 1;

    ballX = panelWidth / 2;
    ballY = brickHeightx8 + brickBuffer + 10;
```

```
    // начальные скорость и угол (45 градусов)
    hSpeed = maxSpeed;
    vSpeed = maxSpeed;

    // начальное направление
    Random leftRightDirection = new Random();
    movingLeft = leftRightDirection.nextBoolean();
    movingUp = false;
}
```

Аналогичным образом установим значение начального угла для мяча равным 45 градусам. Этим параметром можно управлять путем установления значений горизонтальной и вертикальной скорости мяча, равных `maxSpeed`. Кроме того, определим случайным образом, куда изначально будет двигаться мяч — влево или вправо.

Далее создадим метод `update()`. Этот метод будет перемещать мяч с учетом его скорости. Кроме того, метод будет добавлять или вычитать скорость в зависимости от направления движения мяча:

```
public void update() {
    if (movingLeft) {
        ballX -= hSpeed;
    } else {
        ballX += hSpeed;
    }

    if (movingUp) {
        ballY -= hSpeed;
    } else {
        ballY += vSpeed;
    }
}
```

Угол движения мяча может меняться, и у нас два метода управления им. По сути, мы будем прибавлять или вычитать вычисленное значение свойства `oneThird`, чтобы увеличить или уменьшить угол:

```
public void increaseAngle() {
    if (vSpeed - oneThird > 1) {
    // не хотим, чтобы угол стал слишком низким (горизонтально)
    hSpeed += oneThird;
    vSpeed -= oneThird;
    }
}

public void decreaseAngle() {
    if (hSpeed - oneThird > 1) {
```

```
    // не хотим, чтобы угол стал слишком низким (вертикально)
    hSpeed -= oneThird;
    vSpeed += oneThird;
    }
}
```

Когда мяч ударяется о потолок, брусок или кирпич, необходимо изменить его вертикальное направление. Для этого создадим небольшой метод:

```
public void flipVerticalDirection() {
    if (movingUp) {
        movingUp = false;
    } else {
        movingUp = true;
    }
}
```

Чтобы завершить работу над классом `Ball`, создадим геттеры и сеттеры для следующих свойств:

- `movingUp`
- `movingLeft`
- `ballX`
- `ballY`

Также следует создать геттеры для `ballSize` и `ballSizeOffset` (эти два свойства не нуждаются в сеттерах).

## Класс *Brick*

Создайте новый Java-класс с именем `Brick` внутри пакета `javabreakout`. Этот класс будет иметь один импорт — класс `Color` из библиотеки AWT. В нем будут свойства для сохранения координат и цвета текущего кирпича, а также того, сломан он или нет:

```
package javabreakout;

import java.awt.Color;

public class Brick {
    private int brickX;
    private int brickY;
    private int brickMaxX;
    private int brickMaxY;
    private Color color;
    private boolean broken;
```

Наш конструктор будет принимать параметры для координат, размера и цвета:

```
public Brick(int brickX, int brickY, int width, int height,
        Color color) {
    this.color = color;
    this.brickX = brickX;
    this.brickY = brickY;
    this.brickMaxX = brickX + width;
    this.brickMaxY = brickY + height;
    this.broken = false;
}
```

Нам понадобятся геттеры для всех наших свойств, но сеттеры нужны только для свойств `color` и `broken`. Это связано с тем, что кирпичи не двигаются, но когда они будут сломаны, свойство `broken` будет установлено в `true`, а цвет `color` будет установлен в черный.

## Класс *Paddle*

Нам также понадобится класс для определения и управления бруском. Создайте новый Java-класс с именем `Paddle` внутри пакета `javabreakout`:

```
package javabreakout;

public class Paddle {
    private int paddleX;
    private int paddleY;
    private int paddleHeight;
    private int paddleWidth;
    private int paddleSpeed;
```

Классу `Paddle` нужны свойства для местоположения, размера и скорости бруска. Все они задаются в конструкторе:

```
public Paddle(int paddleX, int paddleY, int paddleWidth,
        int paddleHeight, int paddleSpeed) {
    this.paddleX = paddleX;
    this.paddleY = paddleY;
    this.paddleHeight = paddleHeight;
    this.paddleWidth = paddleWidth;
    this.paddleSpeed = paddleSpeed;
}
```

Брусок может двигаться только влево или вправо. Создадим два метода, учитывающих это:

```
public void moveLeft() {
    paddleX -= paddleSpeed;
}
```

```
public void moveRight() {
    paddleX += paddleSpeed;
}
```

Чтобы завершить этот класс, просто добавим геттеры для всех свойств.

## Класс *KeyHandler*

В приложении Breakout брусок управляется с клавиатуры. Поэтому необходимо разработать класс, обрабатывающий нажатия определенных клавиш. Создайте новый Java-класс с именем `KeyHandler` внутри пакета `javabreakout`. Класс `KeyHandler` имеет два импорта: класс `KeyEvent` и интерфейс `KeyListener` из библиотеки AWT. Наш класс `KeyHandler` также будет реализовывать интерфейс `KeyListener`:

```
package javabreakout;

import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

public class KeyHandler implements KeyListener {
    private BreakoutPanel panel;

    private boolean leftPressed = false;
    private boolean rightPressed = false;
```

Наш класс также имеет свойства для определения, были ли нажаты *левая* или *правая* клавиши. Кроме того, имеется свойство для класса `BreakoutPanel`, который пока не создан.

Важно отметить, что класс `KeyHandler` будет инстанцирован в объект и также задан в качестве свойства в классе `BreakoutPanel`. В Java это известно как *кольцевая ссылка*. В целом, кольцевые ссылки имеют тенденцию приводить к сложным проблемам с зависимостями и тупиковым ситуациям, которые трудно разрешить. Но использование кольцевой ссылки избавляет нас от необходимости:

- писать дополнительную сложную логику для запуска нового мяча;
- передавать дополнительное свойство вместе с каждым событием нажатия клавиши.

> **Примечание.** Общепринятой рекомендацией в отношении кольцевых ссылок является отказ от их использования. Обычно они создают больше проблем, чем приносят пользы. Однако в нужной ситуации они могут быть полезны.

Наш конструктор требует, чтобы `BreakoutPanel` передал ссылку на себя в качестве единственного параметра:

```
public KeyHandler(BreakoutPanel breakoutPanel) {
    this.panel = breakoutPanel;
}
```

Реализация интерфейса `KeyListener` требует переопределения трех public-методов: `keyPressed`, `keyReleased` и `keyTyped`.

Мы не будем использовать метод `keyTyped()`, поэтому он останется пустым:

```
@Override
public void keyTyped(KeyEvent e) {

}
```

Метод `KeyPressed()` позволяет определить, перемещает ли игрок брусок влево или вправо, а также пытается ли он запустить новый мяч. Метод принимает объект `KeyEvent`, с помощью которого устанавливается локальная целочисленная переменная `code`. Затем с помощью этого кода можно определить, какая клавиша была нажата:

```
@Override
public void keyPressed(KeyEvent keyPress) {

    int code = keyPress.getKeyCode();

    if (code == KeyEvent.VK_A) {
        leftPressed = true;
    }
    if (code == KeyEvent.VK_D) {
        rightPressed = true;
    }
    if (panel.getBallIsDead()) {
        if (code == KeyEvent.VK_ENTER) {
            panel.releaseBall();
        }
    }
}
```

Для перемещения влево и вправо игроки будут использовать клавиши <A> и <D> соответственно. Эти клавиши были выбраны из стандартной раскладки WASD, обычно используемой в играх. В проверках `if` устанавливается истинным свойство `leftPressed` или свойство `rightPressed`.

В этом методе также появляется кольцевая ссылка на `BreakoutPanel`. Клавиша <Enter> используется для запуска нового мяча, но она работает только в том случае, если в данный момент в игре нет другого мяча. Проверка `if` для этого использует результат метода `getBallIsDead()` класса `BreakoutPanel`. В противном случае нажатие клавиши <Enter> игнорируется. Метод запуска нового мяча `releaseBall()` также находится в `BreakoutPanel` и вызывается в этом экземпляре.

Последний метод, который необходимо написать, — это метод `keyReleased()`. Как и метод `keyPressed()`, он принимает событие `KeyEvent`, чтобы определить, была ли нажатая ранее клавиша отпущена.

Если да, то мы просто устанавливаем булевы свойства `leftPressed` или `rightPressed` в `false`:

```
@Override
public void keyReleased(KeyEvent keyRelease) {
    int code = keyRelease.getKeyCode();

    if (code == KeyEvent.VK_A) {
        leftPressed = false;
    }
    if (code == KeyEvent.VK_D) {
        rightPressed = false;
    }
}
```

Наконец, классу `KeyHandler` нужны геттеры для `leftPressed` и `rightPressed`. Также необходимо создать два геттера для этого класса:

```
public boolean isLeftPressed() {
    return leftPressed;
}

public boolean isRightPressed() {
    return rightPressed;
}
```

## Класс *BreakoutPanel*

Класс `BreakoutPanel` — это то место, где творится волшебство в этом приложении. Создайте новый Java-класс с именем `BreakoutPanel` внутри пакета `javabreakout`. Этот класс будет наследоваться от класса `JPanel` и реализовывать интерфейс `Runnable`. Класс `BreakoutPanel` потребует импорта из следующих библиотек:

- **AWT:** `Color`, `Dimension`, `Font`, `Graphics`, `Graphics2D`
- **Util:** `ArrayList`, `List`
- **Swing:** `JPanel`

Давайте создадим класс `BreakoutPanel`:

```
package javabreakout;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Graphics2D;

import java.util.ArrayList;
import java.util.List;
```

```
import javax.swing.JPanel;

public class BreakoutPanel extends JPanel implements Runnable {

    private static final long serialVersionUID =
        -7279076888542180135L;

    private final int fPS = 60; // кадров в секунду
    private final int brickWidth = 64;
    private final int brickHeight = 32;
    private final int brickBuffer = 128;
    private final int ballSize = 11;
```

Нашему классу потребуется сгенерированный `serialVersionUID`. Также определим пять констант с помощью ключевого слова `final` и установим их значения. Эти константы предназначены для основных параметров нашей игры, таких как количество кадров в секунду, размер кирпичей и мяча, а также количество пикселей над кирпичами.

В классе `BreakoutPanel` также есть четыре простых свойства. Два булевых значения для отслеживания того, является ли мяч "пропавшим" (`BallIsDead`) и играбельным (`BallIsPlayable`) (эти значения не исключают друг друга). Также у нас будут свойства для хранения высоты `height` и ширины `width` самой `JPanel`, а также еще нескольких вещей:

```
private boolean ballIsDead;
private boolean ballIsPlayable;

private int panelHeight;
private int panelWidth;
private int score = 0;
private int ballsRemaining = 3;
```

Нам также нужны свойства класса для отслеживания списка кирпичей и доступных цветов (в том порядке, в котором они должны использоваться). На панели будет отображаться ограниченное количество текста, поэтому для этого создадим объект `Font` с именем `arial40`:

```
private Font arial40 = new Font("Arial", Font.PLAIN, 40);

private List<Brick> bricks;
private List<Color> colorList;
```

Кроме того, необходимы свойства для объектов классов `Ball`, `KeyHandler` и `Paddle`. Как и в случае с классом `SolarSystem` (из предыдущего раздела), понадобится свойство для потока:

```
private Ball ball;
private KeyHandler keyHandler;
```

```
private Paddle paddle;
private Thread panelThread;
```

Для конструкторов мы используем тот же подход, что и в классе SolarSystem, и создадим два конструктора. Один принимает аргументы для размера панели, а другой, безаргументный конструктор, просто вызывает первый:

```
public BreakoutPanel() {
    this(1024,1024);
}

public BreakoutPanel(int width, int height) {
    panelWidth = width;
    panelHeight = height;

    this.setPreferredSize(new Dimension(panelWidth, panelHeight));
    this.setBackground(Color.black);
    this.setFocusable(true);

    keyHandler = new KeyHandler(this);
    this.addKeyListener(keyHandler);

    bricks = generateBricks();
    paddle = new Paddle((panelWidth / 2) - 64,
            panelHeight - 200, 128, 16, 16);
    ballIsDead = true;
    ballIsPlayable = false;
    panelThread = Thread.ofVirtual()
            .name("Breakout")
            .unstarted(this);
}
```

Наш основной конструктор установит значения высоты height и ширины width панели, а также цвет фона (черный) и позволит панели получить модальный фокус на рабочем столе. Далее создадим объект класса KeyHandler, назовем его keyHandler и передадим его (this) в качестве единственного параметра (ссылки на текущий класс). Затем сгенерируем кирпичи и создадим новый объект класса Paddle с именем paddle. Мы также установим начальные состояния для булевых свойств ballIsDead и ballIsPlayable. Наконец, конструктор создаст panelThread как новый виртуальный поток в незапущенном unstarted состоянии.

Прежде чем мы сможем сгенерировать кирпичи, нам понадобится список цветов. Как правило, кирпичи в Breakout строятся в разных цветах для каждого ряда. Поскольку у нас будет восемь рядов кирпичей, создадим метод для составления списка из восьми цветов и хранения их в свойстве colorList:

```
private void generateColors() {
    colorList = new ArrayList<>();
```

```
    colorList.add(Color.RED);
    colorList.add(Color.MAGENTA);
    colorList.add(Color.PINK);
    colorList.add(Color.GRAY);
    colorList.add(Color.YELLOW);
    colorList.add(Color.CYAN);
    colorList.add(Color.GREEN);
    colorList.add(Color.BLUE);
}
```

Далее создадим метод `generateBricks()`. После инициализации возвращаемого значения и номеров строк/столбцов кирпичей мы будем рисовать кирпичи сверху вниз и слева направо. Вложенные циклы `while` обеспечат построение столбца из 8 кирпичей 16 раз. Каждый раз мы создаем новый объект `Brick`, сохраняем его параметры и добавляем его в список:

```
private List<Brick> generateBricks() {

    generateColors();
    List<Brick> returnVal = new ArrayList<>();
    int brickRow = 0;
    int brickCol = 0;

    while (brickCol < 16) {
        while (brickRow < 8) {
            Brick newBrick = new Brick(brickCol * brickWidth,
                (brickRow * brickHeight) + brickBuffer,
                brickCol + brickWidth, brickRow + brickHeight,
                colorList.get(brickRow));
            returnVal.add(newBrick);
            brickRow++;
        }
        brickRow = 0;
        brickCol++;
    }
return returnVal;
}
```

Как только завершено построение столбца из 8 кирпичей, мы устанавливаем значение `brickRow` (счетчик) снова в ноль и увеличиваем `brickCol`. Цикл завершается, когда построен шестнадцатый столбец. Обратите внимание, что значение `brickRow` используется для установки цвета кирпичей.

При расчете высоты кирпича также используется значение `brickBuffer`, равное 128. Это связано с тем, что между верхней частью панели и верхним рядом кирпичей должна быть открытая область размером 128 пикселей.

Поскольку игра Breakout будет работать как виртуальный поток, переопределим метод `run()`. По сути, игра будет работать до тех пор, пока метод потока панели `isAlive()` будет возвращать значение `true`. Внутри цикла вызовем метод `update()`, а также метод `repaint()` панели, а затем заставим поток заснуть (сделать паузу) на 16⅔ секунды:

```
@Override
public void run() {
    while (panelThread.isAlive()) {
        update();
        repaint();

        // вычисление пауз на основе количества кадров в секунду
        try {
            Thread.sleep(1000 / fPS);
            } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

У этого кода есть одна возможная проблема, связанная с количеством кадров, обрабатываемых в секунду. На данный момент код запускается, а затем приостанавливается на короткое время. Время паузы равно одной секунде, деленной на значение `fPS`, которое равно `60`. Предполагается, что мы не учитываем процессорное время, необходимое для выполнения методов `update()` и `repaint()`.

Во время разработки этого кода время, необходимое для выполнения этих двух методов, было измерено и составило менее одной миллисекунды при работе на процессоре AMD A8-5500 2012 года. Таким образом, хотя время вычислений, необходимое для выполнения `update()` и `repaint()`, не равно нулю, оно является незначительным, учитывая современную архитектуру процессоров. Однако для достижения оптимальной эффективности разработчик, конечно, может переделать этот метод, чтобы записывать время до запуска `update()` и после запуска `repaint()` и вычитать разницу из значения, используемого в методе `Thread.sleep()`.

Далее мы разработаем метод `update()`. В нем учитываются новые положения бруска и мяча. К счастью, кирпичи не двигаются, поэтому нет необходимости что-либо пересчитывать для них. Для бруска также нужно проверить с помощью `keyHandler`, была ли нажата левая или правая клавиша, и если да, то скорректировать его движение с помощью методов `moveLeft()` и `moveRight()`. Кроме того, мы удостоверимся, что брусок не выходит за пределы видимой части экрана:

```
private void update() {
    // брусок
    if (keyHandler.isLeftPressed() || keyHandler.isRightPressed()) {
        if (keyHandler.isLeftPressed()) {
            if (paddle.getPaddleX() - paddle.getPaddleSpeed() > 0) {
```

```
                paddle.moveLeft();
            }
        } else {
            if (paddle.getPaddleX() + paddle.getPaddleSpeed()
                    < panelWidth) {
                paddle.moveRight();
            }
        }
    }

    // мяч
    if (!ballIsDead) {
        checkCollision();

        if (ballIsPlayable) {
            // Метод checkCollision может отрисовать мяч неиграбельным
            ball.update();
        }
    }
}
```

Для мяча необходимо обновлять информацию только в том случае, если он существует и может двигаться. Мы также вызовем метод `checkCollision()`, чтобы узнать, касается ли мяч в данный момент кирпича, стены или бруска.

Метод `checkCollision()` довольно сложен и написан с учетом различных возможностей. Во-первых, инстанцируются переменные, которые помогают узнать координаты мяча и бруска, а также размеры бруска.

Первый `if` проверяет, не вышла ли координата Y мяча за пределы нижней границы видимого игрового пространства на панели. Помните, что точка `0,0` находится в левом верхнем углу, поэтому проверка значения `panelHeight` происходит в нижней части экрана. Если она вышла за нижнюю границу, мы объявляем мяч "ушедшим" и неиграбельным, после чего уничтожаем его и уменьшаем значение оставшихся мячей `ballsRemaining`:

```
private void checkCollision() {
    int ballX = ball.getBallX();
    int ballY = ball.getBallY();
    int paddleX = paddle.getPaddleX();
    int paddleY = paddle.getPaddleY();
    int paddleWidth = paddle.getPaddleWidth();
    int paddleHeight = paddle.getPaddleHeight();

    if (ballY > panelHeight) {
        // нижняя граница
        ballIsDead = true;
        ballIsPlayable = false;
```

```
        // уничтожаем мяч
        ball = null;
        ballsRemaining--;
```

Далее нужно проверить, движется ли мяч вниз и коснулся ли он бруска. Если это так, то изменим вертикальное направление мяча с помощью метода `setMovingUp()`. Мы также проверим, нажимает ли игрок клавишу направления в самый момент столкновения, и соответствующим образом изменим угол наклона мяча:

```
    } else if (ballY >= paddleY && !ball.isMovingUp()) {
        // брусок
        // нижняя ось Y и проверка оси X
        if (ballY < paddleY + paddleHeight &&
                    ballX >= paddleX &&
                    ballX <= paddleX + paddleWidth) {

            ball.setMovingUp(true);

            // проверка необходимости изменения угла мяча
            if (keyHandler.isLeftPressed()) {
                if (ball.isMovingLeft()) {
                    ball.increaseAngle();
                } else {
                    ball.decreaseAngle();
                }
            } else if (keyHandler.isRightPressed()) {
                if (ball.isMovingLeft()) {
                    ball.decreaseAngle();
                } else {
                    ball.increaseAngle();
                }
            }
        }
```

В следующих двух проверках `if` мы смотрим, не коснулся ли мяч какой-нибудь боковой стенки. Если да, то меняем горизонтальное направление, вызывая метод `setMovingLeft()` мяча:

```
    } else if (ballX <= 0 && ball.isMovingLeft()) {
        // левая стенка
        ball.setMovingLeft(false);
    } else if (ballX >= panelWidth && !ball.isMovingLeft()) {
        // правая стенка
        ball.setMovingLeft(true);
```

Здесь проверяется, не столкнулся ли мяч с каким-либо из кирпичей. Проверка `if` выясняет, равны ли координаты мяча верхним, нижним, левым и правым координа-

там всей области кирпичей или находятся между ними. Если это так, то мы перебираем все кирпичи, чтобы узнать, какой именно целый кирпич столкнулся с мячом:

```
    } else if (ballY <= (brickHeight * 8) + brickBuffer + brickHeight
            && ballY > brickBuffer) {
        // кирпичи
        for (Brick brick : bricks) {
            if (!brick.isBroken()) {
            // проверка на столкновение, если кирпич не сломан
                int brickX = brick.getBrickX();
                int brickY = brick.getBrickY();
                int brickMaxX = brick.getBrickMaxX();
                int brickMaxY = brick.getBrickMaxY();

                if (ballX >= brickX && ballX <= brickMaxX
                        && ballY >= brickY && ballY <= brickMaxY) {
                    // кирпич ломается
                    brick.setBroken(true);
                    brick.setColor(Color.BLACK);
                    score++;

                    // изменяем направление мяча
                    // после столкновения с кирпичом
                    ball.flipVerticalDirection();
                }
            }
        }
```

Если столкновение произошло с неповрежденным кирпичом, вызываем метод `setBroken()` для этого кирпича со значением `true`, устанавливаем для него черный цвет с помощью метода `setColor()` и увеличиваем счет `score`. Таким образом, выполняется мягкое удаление сломанных кирпичей. Они все еще остаются на месте, но не вызывают столкновений и не видны (потому что совпадают с цветом фона).

Здесь проверяется, достиг ли мяч самой верхней части экрана. Если да, необходимо, чтобы он отскочил вниз, и поэтому вызываем метод `setMovingUp()` мяча для изменения его вертикального направления. Если программный поток проходит эту проверку без вызова какой-либо дополнительной логики, значит, мяч ни с чем не столкнулся. Давайте рассмотрим следующий код:

```
    } else if (ball.getBallY() <= 1) {
    // верхняя стенка
        ball.setMovingUp(false);
    }
    // иначе, без столкновений
}
```

Последний большой метод, который необходимо создать, — это переопределенный метод `paintComponent()`, вызываемый при вызове метода `repaint()`. Сначала нужно

вызвать тот же метод в нашем суперклассе JPanel и создать новый локальный объект Graphics2D с именем g2. Затем перебираем все кирпичи и рисуем каждый из них:

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D)g;

    // кирпичи
    for (Brick brick : bricks) {
        int brickX = brick.getBrickX();
        int brickY = brick.getBrickY();
        g2.setColor(brick.getColor());
        g2.fillRect(brickX, brickY, brickWidth, brickHeight);
    }
```

После того, как кирпичи отображены, можно перейти к бруску:

```
    // брусок
    g2.setColor(Color.WHITE);
    g2.fillRect(paddle.getPaddleX(), paddle.getPaddleY(),
    paddle.getPaddleWidth(), paddle.getPaddleHeight());
```

Далее нарисуем мяч. Мяч — это очень маленький квадрат, и мы используем переменную centerOffset, чтобы убедиться, что рисуем мяч из его центра, а не из левого верхнего угла. Это снижает сложность обнаружения столкновений. Давайте посмотрим на следующий код:

```
    // мяч
    if (ball != null) {
        // СЕРЕБРЯНЫЙ
        g2.setColor(new Color(192,192,192));
        int centerOffset = ball.getBallSizeOffset();
        g2.fillRect(ball.getBallX() - centerOffset,
        ball.getBallY() - centerOffset, ball.getBallSize(),
        ball.getBallSize());
}
```

Наконец, мы отображаем текущие значения очков score и оставшихся мячей ballsRemaining в верхней части экрана. Технически текст отображается в игровой области, но он не вызывает столкновений и находится слишком высоко, чтобы мешать игроку:

```
    // количество набранных очков и оставшихся мячей
    StringBuilder scoreBuilder = new StringBuilder("Score: ");
    scoreBuilder.append(score);
```

```
    StringBuilder currentBallBuilder =
            new StringBuilder("Current Ball: ");
    currentBallBuilder.append(ballsRemaining);

    g2.setColor(Color.white);
    g2.setFont(arial40);
    g2.drawString(scoreBuilder.toString(), 50, 50);
    g2.drawString(currentBallBuilder.toString(), 700, 50);
    g2.dispose();
}
```

Завершающие методы, которые нам нужно создать, совсем небольшие. Для запуска игры необходим метод, который вызывает метод `start()` на `panelThread`:

```
public void start() {
    panelThread.start();
}
```

Нам нужен метод для создания нового мяча и ввода его в игру. Метод `releaseBall()` выполнит это, если значение `ballsRemaining` больше нуля:

```
public void releaseBall() {
    if (ballsRemaining > 0) {
        ball = new Ball(ballSize, panelWidth, brickHeight * 8,
                brickBuffer);
        ballIsDead = false;
        ballIsPlayable = true;
    }
}
```

Наконец, необходим геттер для свойства `ballIsDead`:

```
public boolean getBallIsDead() {
    return this.ballIsDead;
}
```

## Класс *BreakoutGame*

После всего этого можно переходить к последнему классу. Создайте новый Java-класс с именем `BreakoutGame` и убедитесь, что он находится внутри пакета `javabreakout`. Этот класс будет иметь один импорт (`JFrame` из Swing) и должен иметь метод `main`:

```
package javabreakout;

import javax.swing.JFrame;

public class BreakoutGame {
```

```
    public static void main(String[] args) {
        JFrame frame = new JFrame();

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setTitle("Java Breakout");

        BreakoutPanel panel = new BreakoutPanel();
        frame.add(panel);
        frame.pack();
        frame.setVisible(true);

        panel.start();
    }
}
```

Метод `main()` игры Breakout создает новый объект `JFrame` с именем `frame`. Затем мы настраиваем для `frame` желаемое поведение при закрытии, даем `frame` название-заголовок `"Java Breakout"`, и указываем, что `frame` должен быть видимым.

> **Примечание.** На компьютерах Mac слишком ранний вызов метода `setVisible()` класса `JFrame` может привести к тому, что `KeyListener` не будет работать должным образом. Поэтому рекомендуется вызывать `setVisible(true)` в качестве последнего действия, выполняемого над `JFrame`.

Затем создается новый объект класса `BreakoutPanel` с именем `panel`. Наконец, мы добавляем `panel` к `frame`, заполняем `frame`, устанавливаем его видимость и вызываем метод `start()` на `panel`, чтобы начать игру.

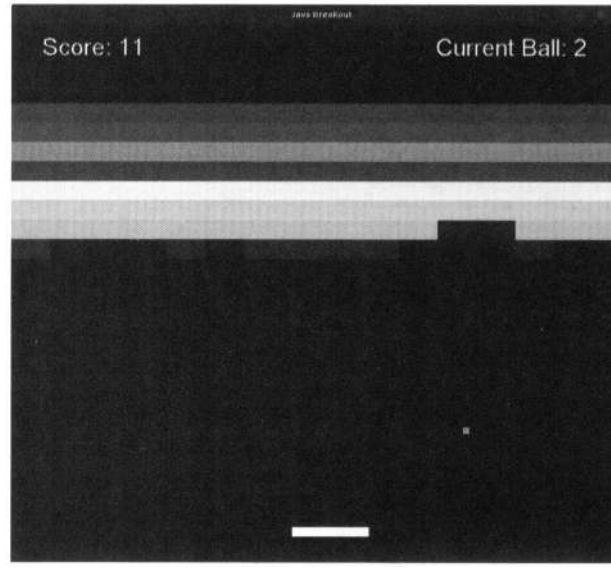



**Рис. 9.4.** Завершенная и запущенная игра Java Breakout

Выполнение `BreakoutGame` в IDE должно запустить нашу игру и дать нам возможность поиграть в нее. Попробуйте перемещать брусок вперед-назад с помощью клавиш <A> и <D> и нажмите клавишу <Enter>, когда будете готовы играть! На экране должно отобразиться нечто похожее на рис. 9.4.

Чтобы поиграть в Java Breakout вне IDE, откройте терминал и перейдите в каталог проекта. Поскольку мы собрали Java Breakout, используя Maven в качестве менеджера зависимостей, сборку можно вызвать следующей командой:

```
mvn clean install
```

После этого JAR-файл jar-with-dependencies должен оказаться в каталоге target/. Затем игру можно запустить, выполнив команду:

```
java -jar target/javabreakout-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

# Заключение

В этой главе мы узнали, как создавать графические программы на Java. Мы начали с простых фигур, а затем перешли к несложной анимации. Изучив, как создавать визуальные элементы, мы создали практически полноценную аркадную игру! Мы продолжим практиковать полученные знания в следующей главе, в которой создадим полноценное веб-приложение на Java.

Java Breakout была разработана для того, чтобы помочь в изучении уроков этой главы. Однако это не полная игра. Ее можно доработать, чтобы включить в нее дополнительные функции, такие как:

- меню;
- экран окончания игры (когда игрок использует свой последний мяч);
- экран победы, когда игрок разбивает все кирпичи;
- звуки;
- возможность перезапуска игры;
- повышающиеся уровни сложности (больше кирпичей, мяч движется быстрее и т. д.);
- соответствующая физика при разбивании кирпича сбоку.

В следующей главе мы создадим наш финальный проект. Этот проект будет представлять собой веб-приложение с базой данных, в котором будут использованы все полученные нами в предыдущих главах знания.

# Важно помнить

- Программирование графики с помощью библиотек AWT и Swing в Java поддерживает рисование на панели `JPanel`, добавленной в `JFrame`.

- Координаты в `JPanel` начинаются с координат `0,0` (X, Y) в левом верхнем углу. Увеличение координат X приводит к движению вправо, а увеличение координат Y — к движению вниз.
- Методы AWT `Graphics2D` должны иметь установленный цвет перед выполнением.
- В библиотеке AWT `Graphics2D` есть двенадцать предопределенных цветов.
- Метод AWT `draw` нарисует контур указанной фигуры тем цветом, который был задан ранее.
- Метод AWT `fill` нарисует контур указанной фигуры и зальет его цветом, который был задан ранее.
- Если класс должен выполняться как поток, он должен наследоваться от класса `Runnable`, а метод `run()` должен быть переопределен.
- Класс `Thread` подходит для реализации платформенных потоков и виртуальных потоков.
- Циклические ссылки могут быть полезны в подходящем контексте, но лучше их избегать.
- Обязательно загружайте изображения и шрифты заранее (в конструкторе), чтобы их можно было быстро вызвать во время выполнения.

# Глава 10

# Завершающий Java-проект

## Введение

Добро пожаловать в заключительную главу! Здесь мы будем работать над финальным проектом, в котором будет использована бóльшая часть навыков и знаний, приобретенных в предыдущих главах.

## Структура

В этой главе при работе над проектом рассматриваются следующие темы.

- Введение в киноприложение.
- База данных.
- Загрузчик данных.
- Запрос данных.
- Создание проекта киноприложения.
- Модель.
- Контроллер.
- Запрос сервиса.
- Вид.

## Цели

Основная цель этой главы — применить на практике уроки, изученные в предыдущих главах. В ней также будет показано, как использовать знания из этих уроков для создания приложения. Цели обучения следующие:

- построить слой данных для нашего приложения на полезных, реальных примерах;
- понять некоторые компромиссы при проектировании баз данных и приложений;

- узнать, как использовать тип `Optional` для предотвращения ошибок с нулевым указателем;
- узнать, как точно настраивать визуальные элементы в Vaadin.

## Знакомство с приложением для работы с фильмами

Мы создадим приложение, которое будет каталогизировать фильмы и хранить информацию о них в базе данных. Наше приложение будет веб-ориентированным (пока что) и будет хранить данные о различных фильмах. Данные о фильмах в базе данных можно будет искать по идентификатору, полному названию или векторному запросу (векторный поиск будет представлен позже в этой главе).

## Архитектура

Подобно тому, как это делалось в *главе 8 "Веб-приложения"*, мы будем использовать шаблон проектирования Model View Controller (MVC). Если бы это было реальное приложение для реальной компании, то в будущем нас могли бы попросить создать мобильное приложение. Поэтому имеет смысл максимально разделить фронт-энд, сервисный уровень и уровень доступа к данным (Data Access Layer, DAL).

Учитывая эти требования, создание веб-приложения с помощью Spring Boot кажется правильным подходом. С помощью Spring Boot можно легко открыть конечные точки сервисов нашего контроллера для использования другими приложениями. Также можно добавить библиотеки для использования базы данных по нашему выбору с помощью Spring Data. И мы также знаем, что можем использовать библиотеки Vaadin для создания внешнего пользовательского интерфейса.

## База данных

Для проектирования слоя данных давайте начнем с обсуждения того, что должно делать наше приложение, чтобы быть функциональным. Если вы уже пользовались таким сайтом или приложением, как Internet Movie Database (**https://imdb.com**) или The Movie Database (**https://www.themoviedb.org/**), то у вас должно быть четкое понимание того, что должно делать наше приложение:

- находить фильм;
- добавлять/обновлять изображение для фильма;
- предлагать похожие фильмы на основе выбранного фильма.

Исходя из этих функциональных требований, наша модель должна поддерживать следующие запросы:

- запрос данных о фильме по id и названию;
- хранение изображений для фильма;
- запрос на поиск похожих фильмов.

## Выбор базы данных

Такие сайты, как IMDB.com, ежемесячно посещают более 200 миллионов уникальных посетителей (*IMDB 2023*). Гипотетически, нашему новому сайту потребуется некоторое время, чтобы достичь такого трафика. Учитывая это, предположим, что в бэкенде необходимо использовать распределенную базу данных. Кроме того, база должна быть способна к масштабированию и росту, так как наши потребности в данных и трафике, скорее всего, будут расти со временем. Поэтому в качестве базы данных для сайта мы будем использовать Apache Cassandra.

## Создание новой векторной базы данных

В мире векторных баз данных *векторы* — это числовые представления (часто массив чисел с плавающей запятой) данных в многомерном пространстве. Векторные базы данных — это хранилища данных, поддерживающие поиск по данным, представленным в векторном виде. Сами векторные вложения часто создаются с помощью генеративного процесса искусственного интеллекта (ИИ), известного как *трансформация*, в результате чего данные, выраженные в виде текста, изображения или звука, преобразуются в массив чисел.

Векторный поиск обычно строится на алгоритмах сходства на основе косинусов, таких как K-Nearest Neighbor (KNN) или Approximate Nearest Neighbor (ANN). Для поддержки приведенного выше запроса на поиск похожих фильмов нам потребуется база данных, способная поддерживать запрос с помощью ANN. Это позволит нам выдавать релевантные предложения о том, какие фильмы пользователям стоит посмотреть в следующий раз.

После изучения предыдущих глав у нас все еще должна быть бесплатная учетная запись Astra DB, поэтому мы будем использовать ее для базы данных Cassandra. Для начала необходимо создать новую базу данных. В браузере перейдите на сайт **https://astra.datastax.com**, авторизуйтесь и создайте новую базу данных Vector.

> **Примечание.** Для этого приложения следует создать новую базу данных Vector (а не Serverless). Бессерверная база данных, которую мы создали на Astra, не поддерживает векторы и не может быть использована для этого приложения.

Необходимо настроить новую базу данных со следующими параметрами:

- **Имя базы данных** (Database name): `bpbMovies`
- **Имя пространства ключей** (Keyspace name): `movieapp`

- **Провайдер** (Provider): `Google Cloud`
- **Регион** (Region): (выберите ближайший доступный регион)

Хотя токен, который использовался в *Главе 7 "Работа с базами данных"*, все еще должен работать, нужно скачать новый пакет безопасного подключения нашей новой базы данных. Скопировать пакет можно в любое место на диске (не обязательно в каталог Downloads).

> **Примечание.** Если требуется новый токен, вернитесь к *главе 7 "Работа с базами данных"*, чтобы получить инструкции по генерации нового токена для Astra DB.

## Проектирование таблиц

Для реализации предложенных нами шаблонов запросов нам понадобятся две таблицы, созданные внутри пространства ключей `movieapp`. Первой будет таблица с именем `movies`, с единственным первичным ключом `movie_id`. Эта таблица будет содержать все данные о каждом фильме, которые будут храниться в базе данных. Последним столбцом базы данных будет наше векторное вложение, которое будет храниться как семимерный float.

Чтобы создать новые таблицы, перейдите в панель Astra DB, выберите базу данных **bpbMovies**, а затем вкладку **CQL Console**. Для начала используем пространство ключей:

```
use movieapp;
```

Затем можно задать определение таблицы:

```
CREATE TABLE movies (
    movie_id INT PRIMARY KEY,
    imdb_id TEXT,
    original_language TEXT,
    genres MAP<INT,TEXT>,
    website TEXT,
    title TEXT,
    description TEXT,
    release_date DATE,
    year INT,
    budget BIGINT,
    revenue BIGINT,
    runtime float,
    movie_vector vector<float,7>
);
```

Кроме того, необходимо создать индекс для этой таблицы. Вторичные индексы гораздо более распространены (и полезны) в мире реляционных баз данных. Но они не часто используются в базах данных NoSQL (например, Apache Cassandra), поскольку распределенные индексы становятся проблематичными в больших мас-

штабах. Однако нам необходимо иметь такой индекс, чтобы сделать возможным выполнение векторного поискового запроса. Он не будет таким проблематичным, как обычный запрос на основе индекса, поскольку мы ограничим результаты (и, соответственно, потребление ресурсов):

```
CREATE CUSTOM INDEX ON movieapp.movies (movie_vector) USING 'StorageAttachedIndex'
```

Наша вторая таблица будет называться `movies_by_title` и будет поддерживать запрос фильмов по их точному названию. Поскольку `title` является первичным ключом, `movie_id` — единственный неключевой столбец, присутствующий в таблице:

```
CREATE TABLE movies_by_title (
    title TEXT PRIMARY KEY,
    movie_id INT
);
```

Эта техника моделирования известна как *ручной индекс*[1] в Cassandra. По сути, при запросе по названию мы сделаем два запроса. Первый будет запрашивать таблицу `movies_by_title` по `title`. Второй запрос будет выполняться к таблице `movies` с идентификатором `movie_id`, полученным из первого запроса. Эта техника позволяет не использовать вторичный индекс.

Хотя подход с ручным индексом может показаться нелогичным, он будет работать быстрее и эффективнее, чем если бы использовался вторичный индекс. Как упоминалось ранее, запрос с вторичным индексом, построенным по столбцу `title`, потребует большого количества вычислительных и сетевых ресурсов, как показано на рис. 10.1.

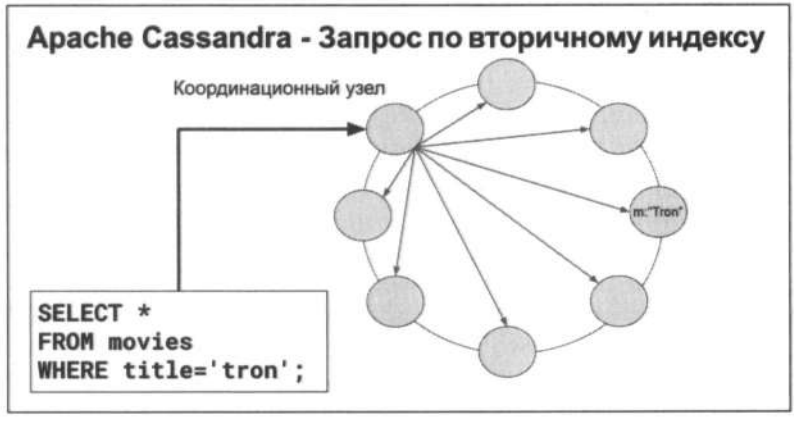



**Рис. 10.1.** Визуальное представление сетевого пути запроса по вторичному индексу в Cassandra

Предположим, что мы пытаемся найти фильм `"tron"` с помощью вторичного индекса по столбцу `title`. Поскольку `title` не является ключом раздела, узел (узлы) кла-

[1] Индекс, созданный вручную пользователем по определенному столбцу. Если данные для этого столбца уже существуют, Cassandra создает индексы данных во время выполнения команды. — Прим. ред.

стера, ответственные за его данные, не могут быть определены путем выполнения хеширования значения в Murmur3. Поэтому приложение (драйвер) выбирает узел в качестве координатора запросов.

Координационный узел будет опрашивать все узлы в кластере в попытке найти совпадение по индексированному значению для столбца title. Затем координатор соберет ответы со всех узлов, сформирует набор результатов и вернет его приложению. Это очень много для одного узла. Эта проблема усугубляется по мере роста кластера и объема данных.

Однако если создать отдельную таблицу для поддержки запросов по названию, то такой запрос может быть выполнен за гораздо меньшее количество сетевых переходов. По сути, запрос по ключам разделов позволяет приложению получать данные непосредственно с отвечающих за них узлов, как показано на рис. 10.2.

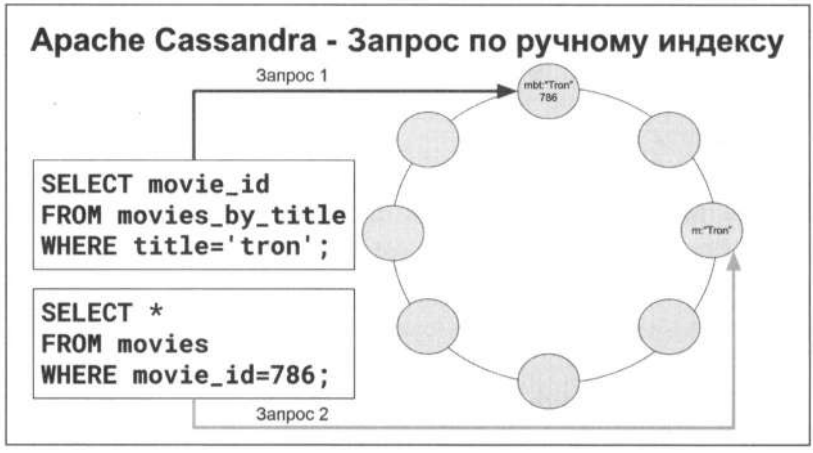



**Рис. 10.2.** Запрос по ручному индексу в Cassandra, иллюстрирующий, что два запроса по ключу раздела более эффективны, чем один запрос по вторичному индексу

**Примечание.** Отправка запросов непосредственно на узлы, отвечающие за запрашиваемые данные, требует от приложения подключения к Apache Cassandra с использованием политики балансировки нагрузки с учетом маркеров. Astra DB спроектирована таким образом, чтобы использовать оптимизированный подход координационного узла, но все равно будет потреблять меньше ресурсов, чем аналогичный запрос, выполняемый на вторичном индексе.

# Загрузчик данных

Для нашего набора данных будет использована сокращенная версия файла movies_metadata.csv с сайта Kaggle. Kaggle — это сайт, на котором ученые, инженеры и другие энтузиасты в области информационных технологий обмениваются и работают над различными общедоступными наборами данных.

Наша версия файла будет содержать всего около 1000 строк. Этого должно быть достаточно для получения разнообразных наборов данных без ожидания загрузки данных.

> **Примечание.** Полная версия файла movies_metadata.csv содержит более 45000 фильмов и находится здесь по адресу: **https://www.kaggle.com/datasets/rounakbanik/the-movies-dataset?select=movies_metadata.csv**.

## pom.xml

Чтобы загрузить набор данных в базу данных, необходимо создать небольшое приложение. Можно просто взять CSV-файл и использовать загрузчик данных на панели Astra DB. Однако требуется скорректировать данные, чтобы правильно загрузить их в наши таблицы. Поэтому создадим небольшое пакетное приложение для загрузки данных. В среде разработки создадим новый проект Maven со следующими свойствами:

- **Group ID**: `com.codewithjava21`
- **Artifact ID**: `movieapp`
- **Name**: `moviedataloader`

После создания проекта отредактируйте полученный файл pom.xml и добавьте в него следующий раздел свойств `properties`:

```
<properties>
    <java.version>21</java.version>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
    <cassandra.driver.version>4.17.0</cassandra.driver.version>
</properties>
```

Затем добавьте следующий раздел зависимостей `dependencies`:

```
<dependencies>
    <dependency>
        <groupId>com.datastax.oss</groupId>
        <artifactId>java-driver-core</artifactId>
        <version>${cassandra.driver.version}</version>
    </dependency>
</dependencies>
```

## Класс *CassandraConnection*

Поскольку наш загрузчик данных не будет использовать Spring Data Cassandra, понадобится создать свой класс для управления соединениями с Cassandra.

Создайте новый класс с именем CassandraConnection внутри пакета com.codewithjava21.movieapp.cassandraconnect. В нем будут импортированы классы InetSocketAddress, Paths и Lists из Java, а также класс CqlSession из библиотеки DataStax Driver:

```
package com.codewithjava21.movieapp.cassandraconnect;

import java.net.InetSocketAddress;
import java.nio.file.Paths;
import java.util.List;
import com.datastax.oss.driver.api.core.CqlSession;
```

В определении класса зададим свойство CqlSession, а первый конструктор примет параметры, необходимые для подключения к кластеру Apache Cassandra:

```
public class CassandraConnection {

    private CqlSession cqlSession;

    public CassandraConnection(String username, String pwd,
            List<InetSocketAddress> endpointList, String keyspace,
            String datacenter) {
        try {
            cqlSession = CqlSession.builder()
                .addContactPoints(endpointList)
                .withAuthCredentials(username, pwd)
                .withKeyspace(keyspace)
                .withLocalDatacenter(datacenter)
                .build();

            System.out.println("[OK] Success");
            System.out.printf("[OK] Welcome to Apache Cassandra! Connected
            to Keyspace %s\n", cqlSession.getKeyspace().get());
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
```

Внутри конструктора свойство cqlSession инициализируется с помощью метода builder() класса CqlSession. Используя with-методы[2] builder(), мы определим учетные данные, пространство ключей и локальный датацентр, а затем вызовем метод build(). Это должно быть сделано внутри блока try/catch, чтобы можно было поймать исключение и вывести на экран сообщение о нем.

[2] Метод используется для изменения определенных полей объекта (например, можно изменить год, месяц или день). — *Прим. ред.*

Далее создадим второй конструктор. Он будет принимать параметры, необходимые для подключения к кластеру Astra DB:

```
public CassandraConnection(String username, String pwd,
    String secureBundleLocation, String keyspace) {
    try {
        cqlSession = CqlSession.builder()
            .withCloudSecureConnectBundle(
                    Paths.get(secureBundleLocation))
            .withAuthCredentials(username, pwd)
            .withKeyspace(keyspace)
            .build();

        System.out.println("[OK] Success");
        System.out.printf("[OK] Welcome to Astra DB! Connected to
        Keyspace %s\n", cqlSession.getKeyspace().get());
    } catch (Exception ex) {
        System.out.println(ex.getMessage());
    }
}
```

За исключением использования защищенного пакета вместо конечных точек и имени локального датацентра, этот конструктор будет во многом похож на первый.

Далее потребуется `public`-геттер, чтобы открыть свойство `cqlSession`:

```
public CqlSession getCqlSession() {
    return cqlSession;
}
```

Наконец, добавим метод `finalize` (вызываемый при уничтожении объекта), чтобы объявить о завершении работы и убедиться, что сессия закрыта:

```
protected void finalize() {
    System.out.println("[shutdown_driver] Closing connection");
    System.out.println();
    cqlSession.close();
}
```

## Класс *AstraConnection*

Далее создадим класс, который будет абстрагировать класс `CassandraConnection`, чтобы мы могли создать конструктор для подключения к Astra DB. Создадим новый класс с именем `AstraConnection` внутри пакета `com.codewithjava21.movieapp.cassandraconnect`. Он будет наследоваться от класса `CassandraConnection` и включать импорт класса `CqlSession` из библиотеки DataStax Driver:

```
package com.codewithjava21.movieapp.cassandraconnect;

import com.datastax.oss.driver.api.core.CqlSession;
```

```
public class AstraConnection extends CassandraConnection {
    static final String ASTRA_ZIP_FILE =
            System.getenv("ASTRA_DB_SECURE_BUNDLE_PATH");
    static final String ASTRA_PASSWORD =
            System.getenv("ASTRA_DB_APP_TOKEN");
    static final String ASTRA_KEYSPACE =
            System.getenv("ASTRA_DB_KEYSPACE");
```

Наш класс также определит три статических (`static`), постоянных (`final`) строковых (`String`) свойства для пакета безопасного соединения, пароля и пространства ключей. Эти свойства будут автоматически извлекаться из соответствующих переменных окружения ОС при инстанцировании класса.

У нас будет один конструктор, и он будет вызывать конструктор `CassandraConnection`, созданный нами для подключения к Astra DB. Поскольку мы используем токен Astra DB в качестве пароля, можно жестко прописать имя пользователя в токене, чтобы избежать путаницы:

```
public AstraConnection() {
    super("token", ASTRA_PASSWORD, ASTRA_ZIP_FILE,
            ASTRA_KEYSPACE);
}
```

Аналогично тому, как это было сделано с классом `CassandraConnection`, мы откроем объект `CqlSession`, а также предоставим метод `finalize()`. Оба эти метода будут вызывать свои аналоги (через `super`) в классе `CassandraConnection`:

```
public CqlSession getCqlSession() {
    return super.getCqlSession();
}

public void finalize() {
    super.finalize();
}
```

## Класс *Movie*

Создайте новый класс с именем `Movie` внутри пакета `com.codewithjava21.movieapp.batchloader`. В нем будут импортированы классы `LocalDate` и `Map` из Java, а также класс `CqlVector` из библиотеки DataStax Driver:

```
package com.codewithjava21.movieapp.batchloader;

import java.time.LocalDate;
import java.util.Map;
import com.datastax.oss.driver.api.core.data.CqlVector;
```

Класс `Movie` должен иметь свойства, соответствующие нашей таблице фильмов `movies`:

```
public class Movie {
    private Integer movieId;
    private String imdbId;
    private String title;
    private String description;
    private Float runtime;
    private String originalLanguage;
    private Map<Integer,String> genres;
    private String website;
    private LocalDate releaseDate;
    private Long budget;
    private Long revenue;
    private Integer year;
    private CqlVector<Float> vector;
```

Нам также понадобятся public-геттеры и сеттеры для каждого свойства.

> **Примечание.** Мы используем классы-обертки для свойств примитивных типов, таких как Float, Integer и Long. Это упростит работу, когда будем вводить эти свойства в наше представление позже.

## Класс *MovieDataLoader*

Создайте новый Java-класс `MovieDataLoader` внутри пакета `com.codewithjava21.movieapp.batchloader`. Убедитесь, что у него есть метод `main`. Этот класс также потребует несколько импортов, включая четыре из библиотеки DataStax Driver и десять из стандартных библиотек Java:

```
package com.codewithjava21.movieapp.batchloader;

import com.codewithjava21.movieapp.cassandraconnect.AstraConnection;

import com.datastax.oss.driver.api.core.CqlSession;
import com.datastax.oss.driver.api.core.cql.BoundStatement;
import com.datastax.oss.driver.api.core.cql.PreparedStatement;
import com.datastax.oss.driver.api.core.data.CqlVector;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
```

```
import java.util.Map;
import java.util.Set;

public class MovieDataLoader {
```

Наш класс MovieDataLoader будет иметь семь private-свойств, четыре из которых предназначены для создания подготавливаемых операторов записи в базу данных:

```
private static CqlSession session;
private static PreparedStatement INSERTStatement;
private static PreparedStatement INSERTByTitleStatement;

private final static String strCQLINSERT = "INSERT INTO movies "
"(movie_id,imdb_id,original_language,genres,"
"Website,title,description,release_date,year,budget,
"revenue,runtime,movie_vector) "
"VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?)";
private final static String strCQLINSERTByTitle =
"INSERT INTO movies_by_title (title, movie_id)"
"VALUES (?,?)";

private static Map<String,Integer> genreIDMainMap = new HashMap<>();
private static Map<Integer,Integer> collectionIDMainMap =
        new HashMap<>();
```

Наш метод main() начнется с подключения к кластеру Cassandra на Astra DB и подготовки двух операторов CQLINSERT:

```
public static void main(String[] args) {
    AstraConnection conn = new AstraConnection();
    session = conn.getCqlSession();

    INSERTStatement = session.prepare(strCQLINSERT);
    INSERTByTitleStatement = session.prepare(strCQLINSERTByTitle);
```

Далее начнем выполнение блока try/catch, откроем файл данных, прочитаем первую строку и создадим переменную, определяющую, был ли прочитан заголовок файла. Кроме того, инициализируем переменную для подсчета количества обработанных фильмов:

```
    try {
        BufferedReader reader = new BufferedReader(
        new FileReader("data/movies_metadata.csv"));
        String movieLine = reader.readLine();
        boolean headerRead = false;
        int movieCount = 0;
```

Далее создадим цикл while, который будет обрабатывать поля в каждой строке до тех пор, пока не будет прочитан заголовок. Поскольку данные в файлах с разделен-

ными запятыми значениями (comma separated values, CSV), очевидно, разделены запятыми, мы можем разделить поля соответствующим образом. Однако некоторые описания и прочий текст могут содержать запятые внутри двойных кавычек. Чтобы решить эту проблему, необходимо использовать регулярное выражение:

```
while (movieLine != null) {
    if (headerRead) {
        String[] movieColumns = movieLine
            .split(",(?=(?:[^\"]*\"[^\"]*\")*[^\"]*$)", -1);
```

По сути, регулярное выражение позволяет разделять по запятой (*Baeldung 2023*), используя положительный просмотр вперед. Убедитесь, что запятая не заключена в пару двойных кавычек или что перед ней находится четное количество двойных кавычек.

Далее создадим новый объект Movie с именем movie и присвоим ему свойства из String-массива movieColumns:

```
Movie movie = new Movie();
String collections = movieColumns[1];
movie.setBudget(Long.parseLong(movieColumns[2]));
String genres = movieColumns[3];
movie.setWebsite(movieColumns[4]);
movie.setMovieId(Integer.parseInt(movieColumns[5]));
movie.setImdbId(movieColumns[6]);
movie.setOriginalLanguage(movieColumns[7]);
movie.setDescription(movieColumns[9]);
float popularity = Float.parseFloat(movieColumns[10]);
```

Во время тестирования было замечено, что свойства даты выхода и продолжительности фильма часто оказывались пустыми, поэтому проверим их и примем соответствующие меры:

```
if (!movieColumns[14].isEmpty()) {
    movie.setReleaseDate(
            LocalDate.parse(movieColumns[14]));
    movie.setYear(movie.getReleaseDate().getYear());
}

movie.setRevenue(Long.parseLong(movieColumns[15]));

if (!movieColumns[16].isEmpty()) {
     movie.setRuntime(
             Float.parseFloat(movieColumns[16]));
} else {
    movie.setRuntime(0F);
}
```

Помимо установки свойства title, необходимо установить еще несколько локальных переменных, которые помогут вычислить векторное вложение:

```
                movie.setTitle(movieColumns[20]);
                float voteAverage = Float.parseFloat(
                        movieColumns[22]);
                int voteCount = Integer.parseInt(movieColumns[23]);
                int collectionId = getCollectionId(collections);

                // обрабатываем жанры
                Map<Integer,String> genreMap = buildGenreMap(genres);
                movie.setGenres(genreMap);
                Integer[] genre = getGenreIds(movie
                        .getGenres().keySet());
```

Наконец, вызовем метод generateVector() для создания векторного вложения и зададим свойство vector для объекта movie. После этого запишем данные в Cassandra, выполним проверку на наличие булевого значения headerRead, а затем прочитаем следующую строку в файле, прежде чем продолжить цикл while:

```
                CqlVector<Float> vector = CqlVector.newInstance(
                    generateVector(collectionId, genre, popularity,
                    voteAverage, voteCount));

                movie.setVector(vector);
                System.out.println(movie.getTitle());
                writeToCassandra(movie);
                movieCount++;
            } else {
                headerRead = true;
            }
            // читаем следующую строку
            movieLine = reader.readLine();
        }
```

После завершения цикла закрываем чтение файла, выводим на экран общее количество фильмов (movieCount), перехватываем (catch) исключение и закрываем метод:

```
        reader.close();
        System.out.printf("%d movies written\n", movieCount);
    } catch (IOException readerEx) {
        System.out.println("Error occurred while reading:");
        readerEx.printStackTrace();
    }
}
```

Теперь можно перейти к private-методам для конкретных функций. Начнем с создания нового private-метода с именем writeToCassandra.

Он будет принимать объект Movie и обеспечивать запись его данных в наши две таблицы:

```
private static void writeToCassandra(Movie movie) {
    // запись данных о фильме
    BoundStatement movieInsert = INSERTStatement.bind(
        movie.getMovieId(), movie.getImdbId(),
        movie.getOriginalLanguage(), movie.getGenres(),
        movie.getWebsite(), movie.getTitle(),
        movie.getDescription(), movie.getReleaseDate(),
        movie.getYear(), movie.getBudget(),
        movie.getRevenue(), movie.getRuntime(),movie.getVector());
    session.execute(movieInsert);

    // запись в to movies_by_title
    BoundStatement movieByTitleInsert = INSERTByTitleStatement.bind(
            movie.getTitle().toLowerCase(), movie.getMovieId());
    session.execute(movieByTitleInsert);
}
```

Обратите внимание, что для таблицы movies_by_title название фильма принудительно переводится в нижний регистр. Поскольку Cassandra требует точного соответствия регистру при запросе, принудительное использование нижнего регистра помогает устранить проблемы с фильмами, у которых в названии смешанный регистр. Мы также переведем пользовательские запросы в нижний регистр, тем самым повысив шансы на совпадение за счет исключения влияния регистра в запросе.

Далее потребуется создать private-метод, принимающий строку и разбирающий ее, чтобы вернуть идентификатор коллекции (collectionId). Идентификаторы коллекции — это способ, с помощью которого фильмы с одним или несколькими продолжениями группируются вместе. Поскольку фильмы с одинаковым идентификатором коллекции связаны друг с другом, будем использовать его для параметров векторного вложения. Мы назовем этот метод getCollectionId, и он будет принимать строку и возвращать целое число. Строка, которую мы ему передадим, содержит список пар ключ/значение. По сути, мы разобьем строковой параметр collection запятой, пройдем по массиву, чтобы найти ключ id, и вернем его значение в виде целого числа.

Однако также требуется нормализовать эти идентификаторы коллекций, поскольку они будут использоваться в качестве параметров в нашем векторном поиске. Если между некоторыми идентификаторами будет слишком большой разброс, результаты рекомендаций могут оказаться искаженными. Поэтому сохраним исходный collectionId в свойстве класса collectionIDMainMap и присвоим каждой коллекции номер, который будет автоматически увеличиваться:

```
private static int getCollectionId(String collections) {
    int collectionId = 0;
    boolean idFound = false;
    String[] collArray = collections.split(",");
```

```
    for (String collection : collArray) {
        String[] kv = collection.split(":");

        if (kv[0].contains("'id'")) {
            idFound = true;
            int originalCollectionId = Integer.parseInt(kv[1].trim());

            if (collectionIDMainMap
                    .containsKey(originalCollectionId)) {
                collectionId = collectionIDMainMap
                        .get(originalCollectionId);
            } else {
                collectionId = collectionIDMainMap.size() + 1;
                collectionIDMainMap.put(originalCollectionId,
                        collectionId);
            }
            break;
        }
    }

    if (!idFound) {
        collectionId = 999;
    }

return collectionId;
}
```

Если мы не найдем `collectionId` (фильм не является частью коллекции), то присвоим ему числовое значение `999`. Это поможет избежать ситуации, когда непринадлежность к коллекции имеет тот же эффект, что и принадлежность к большой коллекции. В векторном выражении `999` должно быть достаточно далеко от допустимых идентификаторов коллекций, чтобы не влиять на результаты.

Далее нам понадобится метод для построения карты жанров наших фильмов. Все фильмы помечаются такими категориями, как боевик, научная фантастика, драма и так далее. Однако количество жанров, к которым относится каждый фильм, может варьироваться в широких пределах. Некоторые из них могут быть частью четырех или пяти, а некоторые — только одного или двух. Поэтому карта `map` — это подходящая структура данных для их хранения.

Мы создадим новый `private`-метод, возвращающий карту `Map` типов `Integer` и `String`, под названием `buildGenreMap`. Он будет анализировать список пар `id/name`, разделенных запятыми. Мы будем игнорировать ID (аналогично тому, как мы делали выше с ID коллекций) и вместо этого создадим нормализованный ID жанра.

В строковом параметре жанра genre также может быть несколько лишних символов в названии, поэтому выполним несколько методов replace() для их удаления:

```
private static Map<Integer,String> buildGenreMap(String genres) {
    Map<Integer,String> returnVal = new HashMap<>();
    String[] genreArray = genres.split(",");

    Integer id = 0;
    String name = "";

    for (String genre : genreArray) {
        String[] genreKV = genre.split(":");

        if (genreKV[0].contains("'name'")) {
            name = genreKV[1]
                    .replaceAll("'","")
                    .replaceAll("\"","")
                    .replaceAll("}","")
                    .replaceAll("]","");

            // принадлежит ли name карте genreIDMainMap?
            if (genreIDMainMap.containsKey(name)) {
                id = genreIDMainMap.get(name);
            } else {
                id = genreIDMainMap.size() + 1;
                genreIDMainMap.put(name, id);
            }
            returnVal.put(id, name.trim());
        }
    }

    return returnVal;
}
```

При построении вектора поиска мы также будем использовать первые три жанра для каждого фильма. Учитывая разброс жанров для каждого фильма, три — это хорошее медианное значение. Теперь можно было бы просто получить идентификаторы из карты, которую мы сгенерировали выше, в виде Set и выполнить toArray(), но это может привести к тому, что мы получим меньше трех значений. Поэтому напишем метод, который будет делать это за нас, инициализируя массив идентификаторов жанров тремя нулями и обновляя их по мере необходимости:

```
private static int[] getGenreIds(Set<Integer> genreIds) {
    int[] genre = {0, 0, 0};
    int counter = 0;
```

```
    for (Integer id : genreIds) {
        if (counter >= genre.length) {
            break;
        }

        genre[counter] = id;
        counter++;
    }

return genre;
}
```

Наконец, понадобится метод для генерации нашего вектора. Как уже говорилось, столбец `movie_vector` — это семимерный `float`, который создается в виде списка с использованием типа-обертки `Float`. По сути, каждый фильм будет иметь вектор, состоящий из следующих свойств по порядку:

- идентификатор коллекции;
- жанр 1;
- жанр 2;
- жанр 3;
- популярность;
- средний рейтинг голосов;
- общее количество голосов за рейтинг выше среднего.

Чтобы правильно обработать каждое из этих свойств в векторное вложение, используем следующий код:

```
private static List<Float> generateVector(Integer collectionId,
        Integer[] genre, float popularity, float voteAverage,
        Integer voteCount) {
    // movie_vector <float,7>
    List<Float> returnVal = new ArrayList<>();
    returnVal.add(Float.parseFloat(collectionId.toString()));
    returnVal.add(Float.parseFloat(genre[0].toString()));
    returnVal.add(Float.parseFloat(genre[1].toString()));
    returnVal.add(Float.parseFloat(genre[2].toString()));
    returnVal.add(popularity);
    returnVal.add(voteAverage);
    returnVal.add(Float.parseFloat(voteCount.toString()));

    return returnVal;
}
```

Перед запуском загрузчика данных добавьте переменные окружения в конфигурацию запуска.

Наш загрузчик данных требует установки следующих переменных окружения:

- ASTRA_DB_SECURE_BUNDLE_PATH
- ASTRA_DB_APP_TOKEN
- ASTRA_KEYSPACE

Теперь, когда код готов, а переменные окружения установлены, можно запускать загрузчик данных. Если все работает, на экране можно будет увидеть названия фильмов, появляющиеся по мере их загрузки из файла. После завершения работы мы должны увидеть, что было загружено 1005 фильмов:

```
The Empire Strikes Back
Return of the Jedi
Back to the Future
Aliens
Alien
1005 movies written
```

# Запрос данных

Теперь, когда данные загружены, можно проверить их, протестировав некоторые из запросов, которые будут использоваться в приложении. В браузере вернитесь на приборную панель Astra DB, выберите базу данных **bpbMovies**, а затем вкладку **CQL Console**.

Первое, что нужно сделать, — это использовать наше пространство ключей:

```
use movieapp;
```

Теперь можно попробовать выполнить запрос по названию (`title`). Не забывайте передавать названия фильмов только в нижнем регистре. Поскольку в нашей базе данных всего 1005 фильмов, у нас нет полного набора данных из Kaggle. Поэтому выбирайте один из тех, что отображаются при запуске загрузчика:

```
SELECT * FROM movies_by_title
WHERE 'star wars';

title          | movie_id
---------------+-----------
star wars      | 11
(1 rows)
```

В результате выполнения этого запроса получен идентификатор фильма, с которым можно работать.

Итак, теперь давайте выполним запрос к таблице фильмов:

```
SELECT movie_id, title, release_date FROM movies
WHERE movie_id=11;

movie_id      | title       | release_date
--------------+-------------+--------------
          11 | Star Wars   | 1977-05-25
(1 rows)
```

Поскольку в этой таблице много столбцов, мы предусмотрительно выбрали несколько из них, чтобы сохранить место. Но теперь давайте повторим этот запрос с другим столбцом в конце:

```
SELECT movie_id, title, movie_vector FROM movies
WHERE movie_id=11;

movie_id      | title       | movie_vector
--------------+-------------+-------------------------------------
          11 | Star Wars   | [37, 4, 8, 13, 42.1497, 8.1, 6778]
(1 rows)
```

Теперь, когда у нас есть столбец `movie_vector`, можно попробовать выполнить векторный поиск. Синтаксис запроса теперь будет немного другим. В данном случае не будет использоваться условие `WHERE`. Вместо этого используем `ORDER BY` с условием `ANN OF` языка CQL:

```
SELECT title FROM movies
ORDER BY movie_vector ANN OF [37, 4, 8, 13, 42.1497, 8.1, 6778]
LIMIT 6;

title                       | movie_vector
----------------------------+-----------------------------------
Star Wars                   | [37, 4, 8, 13, 42.1497, 8.1, 6778]
The Empire Strikes Back     | [37, 4, 8, 13, 19.47096, 8.2, 5998]
Return of the Jedi          | [37, 4, 8, 13, 14.58609, 7.9, 4763]
The Lion King               | [49, 1, 3, 7, 21.60576, 8, 5520]
Pocahontas                  | [10, 1, 3, 4, 13.28007, 6.7, 1509]
 Batman                     | [18, 5, 8, 0, 19.10673, 7, 2145]
(6 rows)
```

Как показано выше, векторный поиск на основе вектора `movie_vector`, созданного для названия `Star Wars`, вернул похожие фильмы. Первые два результата (после самого фильма) — это два других фильма из цикла `Star Wars`, о чем свидетельствует значение 37 в позиции `collectionID` вектора.

Считаются ли три других фильма похожими или нет — вопрос субъективный. Результаты, безусловно, улучшились бы при наличии большего количества данных, и, как было указано выше, мы используем лишь часть исходного файла `movies_metadata.csv`.

Если бы мы хотели немного разобраться, в каком порядке возвращаются фильмы, можно было бы добавить в запрос функцию CQL `similarity_cosine()`. В качестве параметров она принимает векторный столбец и вектор:

```
SELECT title, similarity_cosine(movie_vector,
[37, 4, 8, 13, 42.1497, 8.1, 6778]) as similarity
FROM movies
ORDER BY movie_vector ANN OF [37, 4, 8, 13, 42.1497, 8.1, 6778]
LIMIT 6;
```

```
title                 | similarity | movie_vector
----------------------+-----------------------------------------------------
Star Wars             |          1 | [37, 4, 8, 13, 42.1497, 8.1, 6778]
The Empire Strikes Back |   0.999998 | [37, 4, 8, 13, 19.47096, 8.2, 5998]
Return of the Jedi    |   0.999996 | [37, 4, 8, 13, 14.58609, 7.9, 4763]
The Lion King         |   0.999995 | [49, 1, 3, 7, 21.60576, 8, 5520]
Pocahontas            |   0.999995 | [10, 1, 3, 4, 13.28007, 6.7, 1509]
Batman                |            | 0.999992 | [18, 5, 8, 0, 19.10673, 7, 2145]
(6 rows)
```

> **Примечание.** Обычно векторный поиск используется с векторными вложениями, созданными на основе результатов LLM (large language model, большая языковая модель) с помощью Word2Vec или другого метода обработки естественного языка (natural language processing, NLP). Хотя относительные значения идентификаторов коллекций не имеют смысла по отношению друг к другу, векторный поиск — это быстрый и простой способ предоставления рекомендаций пользователям.

# Создание проекта киноприложения

Получив данные, можно переходить к созданию приложения о фильмах. Поскольку мы будем использовать Spring Boot и Spring Data для облегчения работы, откройте браузер и перейдите в Spring Initializr (рис. 10.3) по адресу:

**https://https://start.spring.io/**

Убедитесь, что выбраны следующие опции:

- **Project**: Maven
- **Spring Boot**: `3.1.2`
- **Group**: `com.codewithjava21`

- **Artifact**: `movieapp`
- **Name**: `movieapp`
- **Description**: `Movie application for the Code with Java 21 book`
- **Package name**: `com.codewithjava21.movieapp`
- **Packaging**: `Jar`
- **Java**: `21` (или более высокая)
- **Dependencies**: Spring Web, Spring Data for Apache Cassandra, Vaadin

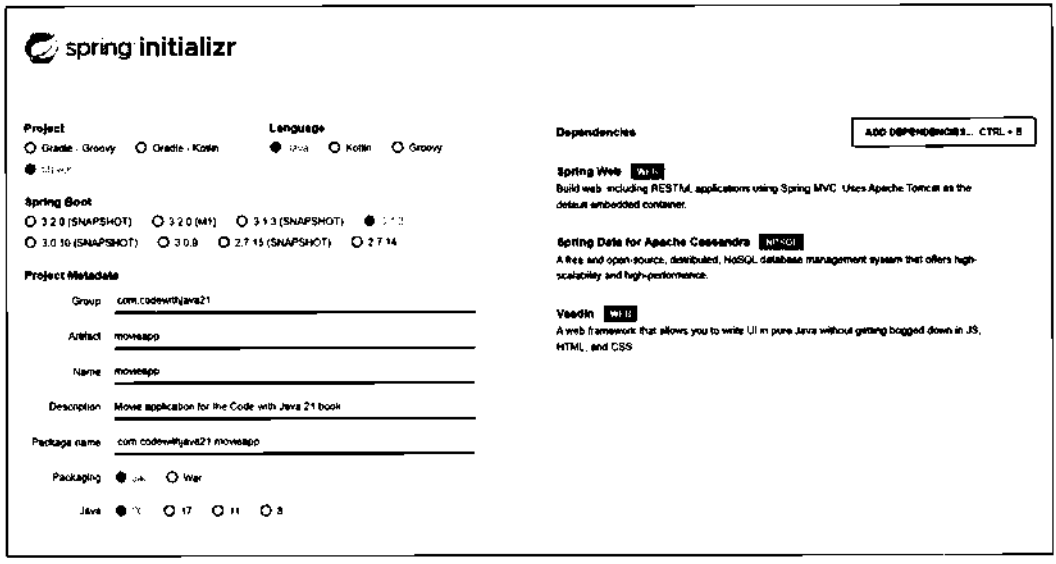



**Рис. 10.3.** Spring Initializr показывает необходимые параметры для киноприложения

После этого нажмите на кнопку **Generate**, чтобы начать загрузку. Затем переместим загруженный zip-файл в каталог рабочей области нашей IDE и распакуем его. Далее перейдем в IDE и импортируем его как новый Maven-проект.

## Каталог изображений

Для этого приложения потребуется создать каталог (папку) для хранения изображений фильмов. Внутри папки проекта (на том же уровне, что и файл pom.xml) обязательно создайте новый каталог с именем images.

## pom.xml

После импорта проекта нужно внести некоторые изменения в файл pom.xml. Прежде всего, если необходимо выбрать не Java 21 (на рис. 10.3 показана Java 20), а что-то другое, следует внести это изменение в раздел свойств `properties`.

Кроме того, нужно добавить свойство для версии драйвера Cassandra, чтобы использовалась версия 4.17.0:

```
<properties>
    <java.version>21</java.version>
    <cassandra.driver.version>4.17.0</cassandra.driver.version>
    <vaadin.version>24.1.2</vaadin.version>
</properties>
```

Далее потребуется добавить зависимости для Astra Spring Boot Starter и для драйвера Cassandra. Astra Spring Boot Starter проверит правильность подключения переменных окружения Astra. Зависимость от драйвера Cassandra позволит нам переопределить версию драйвера Cassandra от Spring Data, что даст возможность использовать функциональность Vector Search:

```
<dependency>
    <groupId>com.datastax.astra</groupId>
    <artifactId>astra-spring-boot-3x-starter</artifactId>
    <version>0.6.4</version>
</dependency>
<dependency>
    <groupId>com.datastax.oss</groupId>
    <artifactId>java-driver-core</artifactId>
    <version>${cassandra.driver.oss.version}</version>
</dependency>
```

## application.yml

Spring Boot должен автоматически сгенерировать файл application.properties. Переименуем этот файл в application.yml и отредактируем его так, чтобы он содержал следующие данные:

```
server:
  port: 8080
  error:
    include-stacktrace: always

spring:
  application:
    name: MovieApp
  profiles:
    active: default
  data:
    cassandra:
      schema-action: NONE
astra:
  api:
```

```
    application-token: ${ASTRA_DB_APP_TOKEN}
    database-id: ${ASTRA_DB_ID}
    database-region: ${ASTRA_DB_REGION}
    cross-region-failback: false
  cql:
    enabled: true
    download-scb:
      enabled: true
  driver-config:
    basic:
      session-keyspace: ${ASTRA_DB_KEYSPACE}
      request:
        timeout: 8s
        consistency: LOCAL_QUORUM
        page-size: 5000
    advanced:
      connection:
        init-query-timeout: 10s
        set-keyspace-timeout: 10s
      control-connection:
        timeout: 10s
```

# Модель

После подготовки предварительных элементов проекта наконец можно перейти к самому приложению. Мы начнем с модели, поскольку она является фундаментом, на котором строится приложение.

## Класс *Movie*

Начнем с класса `Movie`. Создайте новый Java-класс `Movie` с пакетом `com.codewithjava21.movieapp.service`. Классу потребуется шесть импортов, включая класс `CqlVector` из библиотеки DataStax, классы `Column`, `PrimaryKey` и `Table` из Spring Data, а также классы `Local/Date` и `Map` из Java:

```
package com.codewithjava21.movieapp.service;

import java.time.LocalDate;
import java.util.Map;

import org.springframework.data.cassandra.core.mapping.Column;
import org.springframework.data.cassandra.core.mapping.PrimaryKey;
import org.springframework.data.cassandra.core.mapping.Table;

import com.datastax.oss.driver.api.core.data.CqlVector;
```

Определение класса потребует аннотации `@Table` из Spring Data, указывающей имя таблицы фильмов `movies`:

```
@Table("movies")
public class Movie
```

Затем можно создать свойства для каждого столбца таблицы `movies`. Поскольку свойство `movieId` является первичным ключом таблицы, воспользуемся аннотацией `@PrimaryKey` из Spring Data:

```
    @PrimaryKey("movie_id")
    private Integer movieId;
```

Далее можно определить остальные свойства:

```
    private String title;
    private String description;
    private float runtime;
    private String image;
    private Map<Integer,String> genres;
    private String website;
    private Long budget;
    private Long revenue;
    private Integer year;
```

Однако некоторые имена свойств не будут совпадать с именами столбцов в таблице. В решении этой проблемы поможет аннотация `@Column` из Spring Data:

```
    @Column("imdb_id")
    private String imdbId;
    @Column("original_language")
    private String originalLanguage;
    @Column("release_date")
    private LocalDate releaseDate;
    @Column("movie_vector")
    private CqlVector<Float> vector;
```

Для всех этих свойств должны быть созданы соответствующие геттеры и сеттеры. Здесь они не приводятся для краткости.

## Интерфейс *MovieRepository*

Создайте новый Java-интерфейс внутри пакета `com.codewithjava21.movieapp.service` с названием `MovieRepository`. Он будет наследоваться от класса `CassandraRepository` из библиотеки Spring Data. `CassandraRepository` должен быть типизирован классами `Movie` и `Integer`, которые соответствуют типу первичного ключа. Определение интерфейса также должно быть дополнено аннотацией `@Repository` из Spring Data:

```
package com.codewithjava21.movieapp.service;
import org.springframework.data.cassandra.repository.CassandraRepository;
import org.springframework.stereotype.Repository;
```

```
@Repository
public interface MovieByRepository extends
        CassandraRepository<Movie,Integer> {

    @Query("SELECT * FROM movies ORDER BY movie_vector ANN OF ?0 LIMIT 6")
    List<Movie> findMoviesByVector(CqlVector<Float> vector);
}
```

Внутри интерфейса определяется пользовательский запрос с помощью аннотации `@Query`. Здесь будет задан векторный поисковый запрос. Параметры обозначаются вопросительным знаком и порядковым номером. Поскольку наш запрос принимает один параметр, мы будет использовать `?0` для привязки первого параметра. Ниже аннотации мы присвоим запросу имя `findMoviesByVector` и обязуем его возвращать список типа `Movie`. В качестве единственного параметра он принимает `CqlVector` типа `Float`.

## Класс MovieByTitle

Создайте новый Java-класс внутри пакета `com.codewithjava21.movieapp.service` с именем `MovieByTitle`. Класс `MovieByTitle` будет использовать те же импорты Spring Data, что и класс `Movie`. Свойства должны как можно точнее соответствовать двум столбцам в таблице `movies_by_title`, с использованием аннотаций Spring Data при необходимости:

```
package com.codewithjava21.movieapp.service;

import org.springframework.data.cassandra.core.mapping.Column;
import org.springframework.data.cassandra.core.mapping.PrimaryKey;
import org.springframework.data.cassandra.core.mapping.Table;

@Table("movies_by_title")
public class MovieByTitle {

    @PrimaryKey("title")
    private String title;

    @Column("movie_id")
    private int movieId;

    public int getMovieId() {
        return movieId;
    }

    public void setMovieId(int movieId) {
        this.movieId = movieId;
    }
```

```
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

### Интерфейс *MovieByTitleRepository*

Создайте новый Java-интерфейс внутри пакета `com.codewithjava21.movieapp.service` с названием `MovieByTitleRepository`. Он будет наследоваться от класса `CassandraRepository` из библиотеки Spring Data. `CassandraRepository` должен быть типизирован классом `MovieByTitle` и `String`, который соответствует типу первичного ключа. В определении интерфейса также должна быть использована аннотация `@Repository` из Spring Data:

```
package com.codewithjava21.movieapp.service;

import org.springframework.data.cassandra.repository.CassandraRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface MovieByTitleRepository extends
CassandraRepository<MovieByTitle,String> {

}
```

## Контроллер

Для нашего контроллера потребуется всего один класс. Этот класс будет называться `MovieAppController`.

### Класс *MovieAppController*

Создайте новый Java-класс внутри пакета `com.codewithjava21.movieapp.service` с именем `MovieAppController`. Класс `MovieAppController` потребует импорта классов `ResponseEntity`, `GetMapping`, `PathVariable` и `RestController` из Spring. Он также потребует импорта стандартных Java-классов `ArrayList`, `List` и `Optional`:

```
package com.codewithjava21.movieapp.service;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
```

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RequestMapping("/movieapp")
@RestController
public class MovieAppController {
    private MovieRepository movieRepo;
    private MovieByTitleRepository movieTitleRepo;
```

Классу `MovieAppController` понадобятся аннотации `@RequestMapping` и `@RestController`. Аннотация `@RequestMapping` также должна указывать относительный URL для сервиса — укажем его как `/movieapp`. Классу также потребуются два private-свойства; по одному для каждого интерфейса `Repository`.

У класса будет один конструктор, принимающий параметры для каждого из репозиториев и устанавливающий private-свойства:

```
public MovieAppController(MovieRepository movieRepo,
        MovieByTitleRepository movieTitleRepo) {
    this.movieRepo = movieRepo;
    this.movieTitleRepo = movieTitleRepo;
}
```

Наш контроллер будет открывать три конечные точки, по одной для каждого требуемого запроса. Начнем с самого простого метода для конечной точки, который будет возвращать один фильм по его ID (внутри класса `ResponseEntity`). Для этого потребуется аннотация `@GetMapping`, чтобы указать дополнительную составляющую относительного URL для ID фильма:

```
@GetMapping("/movies/{id}")
public ResponseEntity<Optional<Movie>>
        getMovieByMovieId(@PathVariable(value="id") int movieId) {

    Optional<Movie> returnVal = movieRepo.findById(movieId);

    if (returnVal.isPresent()) {
        return ResponseEntity.ok(returnVal);
    }

    return ResponseEntity.ok(Optional.ofNullable(null));
}
```

Метод использует тип `Optional`, полезный при работе с запросами данных, которые могут возвращать значения `null`. По сути, если вызов метода `findById()` объекта `movieRepo` возвращает `null`, мы можем обработать его соответствующим образом.

Далее создадим метод конечной точки, который будет возвращать один фильм по его названию. Этот метод будет работать на той же основе, что и предыдущий, за исключением того, что нужно выполнить два запроса, чтобы получить фильм. Кроме того, у нас может быть третий запрос, если мы во второй раз запросим таблицу `movies_by_title`, добавив к названию "the". Благодаря этому и использованию нижнего регистра в названиях, мы, надеюсь, достаточно ослабили критерии поиска:

```
@GetMapping("/movies/{title}")
public ResponseEntity<Optional<Movie>> getMovieByTitle (@PathVariable(value="title")
String movieTitle) {

    Optional<Movie> returnVal = Optional.ofNullable(new Movie());
    Optional<MovieByTitle> movieByTitle =
            movieTitleRepo.findById(movieTitle.toLowerCase());
    if (movieByTitle.isEmpty()) {
        // Попробуем еще раз с "the" в начале
        movieByTitle = movieTitleRepo.findById("the "
                + movieTitle.toLowerCase());
    }

    if (movieByTitle.isPresent()) {
        int movieId = movieByTitle.get().getMovieId();
        returnVal = movieRepo.findById(movieId);

        return ResponseEntity.ok(returnVal);
    }

    return ResponseEntity.ok(Optional.ofNullable(null));
}
```

Последний метод контроллера будет предназначен для конечной точки рекомендаций `recommendations`. Он будет использовать `@GetMapping` для определения URL конечной точки, принимающей ID фильма в качестве единственного параметра. Этот метод выполняет два запроса, первый возвращает фильм по его ID, что позволяет получить доступ к вектору фильма. Второй запрос — это собственно векторный поиск, использующий возвращенный вектор в качестве параметра:

```
@GetMapping("/recommendations/movie/{id}")
public ResponseEntity<List<Movie>> getMovieRecommendationsById(
        @PathVariable(value="id") int movieId) {

    List<Movie> returnVal = new ArrayList<>();
    Optional<Movie> origMovie = movieRepo.findById(movieId);

    // Получаем список фильмов по вектору исходного фильма
    returnVal = movieRepo.findMoviesByVector(
            origMovie.get().getVector());
```

```
    // Первым элементом в списке будет исходный фильм, поэтому удаляем его
    returnVal.remove(0);
    return ResponseEntity.ok(returnVal);
}
```

Если вспомнить CQL-запросы, которые выполнялись ранее в этой главе, исходный фильм всегда является первой записью в наборе результатов. Поэтому можно смело удалять этот фильм из нашего списка `List`, и не возвращать его.

# Запрос к сервису

После этого можно запускать наше приложение Spring Boot и запрашивать конечные точки сервиса с помощью команды `curl` (из терминальной сессии) или веб-браузера.

## Фильмы по ID

Чтобы получить данные о фильме по ID, выполните следующую команду `curl`:

```
curl -XGET http://localhost:8080/movieapp/movies/id/1891
```

```
{"movieId":1891,"title":"The Empire Strikes Back","description":"\
"The epic saga continues as Luke Skywalker, in hopes of defeating the evil
Galactic Empire, learns the ways of the Jedi from aging master Yoda. But Darth
Vader is more determined than ever to capture Luke. Meanwhile, rebel leader
Princess Leia, cocky Han Solo, Chewbacca, and droids C-3PO and R2-D2 are thrown
into various stages of capture, betrayal and despair.\"","runtime":124.0,
"image":null,"genres":{"4":"Adventure","8":"Action","13":"Science Fiction"},
"website":"http://www.starwars.com/films/star-wars-episode-v-the-empire-
strikesback","budget":18000000,"revenue":538400000,"year":1980,"imdbId":
"tt0080684","originalLanguage":"en","releaseDate":"1980-05-17","vector":
{"empty":false}}
```

## Фильмы по названию

Чтобы получить данные о фильме по названию, выполните следующую команду `curl`:

```
curl -XGET http://localhost:8080/movieapp/movies/title/aliens
```

```
{"movieId":679,"title":"Aliens","description":"\"When Ripley's lifepod is found
by a salvage crew over 50 years later, she finds that terra-formers are on the
very planet they found the alien species. When the company sends a family of
colonists out to investigate her story, all contact is lost with the planet and
colonists. They enlist Ripley and the colonial marines to return and search for
answers.\"","runtime":137.0,"image":null,"genres":{"8":"Action","10":
"Thriller","11":"Horror","13":"Science Fiction"},"website":"","budget":
18500000,"revenue":183316455,"year":1986,"imdbId":"tt0090605",
"originalLanguage":"en","releaseDate":"1986-07-18","vector":{"empty":false}}
```

### Рекомендации фильмов

Чтобы получить данные о рекомендациях для фильма по идентификатору, выполните следующую команду `curl`:

```
curl -XGET http://localhost:8080/movieapp/recommendations/movie/id/1891

[{"movieId":1892,"title":"Return of the Jedi",...
{"movieId":8587,"title":"The Lion King",...
{"movieId":862,"title":"Toy Story",...
{"movieId":10530,"title":"Pocahontas",...
```

# Представление

Как и контроллер, уровень представления нашего приложения будет состоять из одного класса. Класс `MovieApMainView` будет реализовывать многие классы и компоненты `Vaadin`, и выступать в качестве основного интерфейса для наших пользователей.

### Класс *MovieAppMainView*

Завершающая часть нашего приложения — это представление. Создайте новый Java-класс `MovieAppMainView` в пакете `com.codewithjava21.movieapp`. Он должен наследоваться от класса `VerticalLayout` из библиотеки Vaadin. Этот класс потребует много импортов, поэтому не будем перечислять их здесь. Их можно найти в ресурсах кода к этой книге.

> **Примечание.** IDE может рекомендовать или потребовать определить новый `serialVersionID`. Это нормально, и его можно сгенерировать без проблем.

Для определения класса потребуется аннотация `@Route` из библиотеки Vaadin. Для разных классов `Vaadin` (веб-страниц) могут быть определены разные маршруты. Но для наших целей пустой URL означает размещение нашего приложения в корне:

```
@Route("")
public class MovieAppMainView extends VerticalLayout {
    private TextField queryField = new TextField();
    private RadioButtonGroup<String> queryBy =
            new RadioButtonGroup<>();
    private Button queryButton;
    private Button upButton;
```

Мы также определим ряд компонентов и свойств `Vaadin` как private в самом классе. Это необходимо, потому что динамическая природа приложения потребует от нас доступа к компонентам `Vaadin`, которые уже были размещены как видимые на стра-

нице, и их изменения. Мы определим компоненты запроса (Field, RadioButton и queryButton), а также кнопку загрузки.

Далее определим шесть свойств изображения. Первое из них называется image и будет использоваться в качестве основного изображения для каждого фильма. Остальные изображения предназначены для рекомендаций и пронумерованы с первого по пятое:

```
private Image image = new Image();
private Image recImage1 = new Image();
private Image recImage2 = new Image();
private Image recImage3 = new Image();
private Image recImage4 = new Image();
private Image recImage5 = new Image();
```

Воспользуемся компонентом Span из библиотеки Vaadin, чтобы создать теги или значки для каждого из жанров фильма:

```
private Span genre1 = new Span();
private Span genre2 = new Span();
private Span genre3 = new Span();
```

Кроме того, потребуется несколько компонентов TextField:

```
private TextField movieId = new TextField("ID");
private TextField releaseDate = new TextField("release date");
private TextField website = new TextField("website");
private TextField imdbWebsite = new TextField("IMDB website");
private TextField imdb = new TextField("IMDB");
private TextField language = new TextField("original language");
private TextField budget = new TextField("budget");
private TextField revenue = new TextField("revenue");
private TextField voteRating = new TextField("rating");
private TextField votes = new TextField("total votes");
```

Компоненты Paragraph будут использоваться для отображения фрагментов данных фильма, для которых не нужны текстовая область или рамка:

```
private Paragraph description = new Paragraph();
private Paragraph title = new Paragraph();
private Paragraph recommendation1 = new Paragraph();
private Paragraph recommendation2 = new Paragraph();
private Paragraph recommendation3 = new Paragraph();
private Paragraph recommendation4 = new Paragraph();
private Paragraph recommendation5 = new Paragraph();
private Paragraph year = new Paragraph();
```

Есть несколько дополнительных компонентов и свойств, которые необходимо определить. Нам понадобится локаль для форматирования сумм доходов и бюджета. Шаблон регулярного выражения будет использоваться для определения того, является ли значение числом. Нам понадобятся буфер памяти, ресурс потока и компонент `Upload`, позволяющий загружать изображения для каждого фильма:

```
    private Locale enUS = Locale.US;
    private Pattern numericPattern =
            Pattern.compile("-?\\d+(\\.\\d+)?");
    private MemoryBuffer buffer;
    private Upload upload;
    private StreamResource noImgFileStream;
    private String noImageFile = "images/noImage.png";
    private Map<Integer,String> mapGenres = new HashMap<>();
    private MovieAppController controller;
```

Кроме того, определим свойства для локальной реализации карты жанров фильма, а также для упрощения доступа к контроллеру.

Конструктор будет принимать параметры для каждого хранилища, чтобы можно было определить наш контроллер. Эти параметры будут подставляться Spring Boot во время выполнения. Мы также будем использовать метод `add()` из Vaadin Layout для вертикального построения отдельных частей нашей страницы. Мы не создали методы, указанные в вызовах `add()`, поэтому сделаем это следующим шагом:

```
public MovieAppMainView(MovieRepository mRepo,
        MovieByTitleRepository mtRepo) {

    controller = new MovieAppController(mRepo, mtRepo);

    add(buildQueryBar());
    add(buildTitle());
    add(buildImageUpdateControls());
    add(buildImageData(), description);
    add(buildGenreData());
    add(buildMovieMetaData());
    add(buildFinancialData());
    add(buildRatingData());
    add(buildWebsiteData());
    add(new Paragraph("You may also enjoy these similar titles:"));
    add(buildRecommendations());
}
```

Сначала создадим `QueryBar`. Это будет основной инструмент, с помощью которого пользователи будут находить фильмы. Наши методы `build` начнутся с определения объекта `layout` класса `HorizontalLayout`. В этом методе мы создадим кнопку

queryButton, добавим значок лупы в поле queryField, а затем вызовем дополнительный метод build для создания радиокнопки:

```
private Component buildQueryBar() {
    HorizontalLayout layout = new HorizontalLayout();

    queryButton = new Button("Query");

    queryButton.addThemeVariants(ButtonVariant.LUMO_PRIMARY);
    Icon search = new Icon(VaadinIcon.SEARCH);
    queryField.setPrefixComponent(search);

    layout.add(queryField, queryButton, buildQueryRadio());

    queryButton.addClickListener(click -> {
        refreshData();
    });

    return layout;
}
```

Кроме того, добавим слушателя для кнопки queryButton. Он будет вызывать метод refreshData при нажатии на кнопку.

Метод buildQueryRadio() создаст радиокнопку — она позволит пользователю выбрать, выполнять ли запрос по ID или по названию. По умолчанию установим значение ID. Взгляните на следующий код:

```
private Component buildQueryRadio() {
    HorizontalLayout layout = new HorizontalLayout();

    queryBy.setLabel("Query by:");
    queryBy.setItems("ID", "Title");
    queryBy.setValue("ID");

    layout.add(queryBy);

    return layout;
}
```

Метод buildTitle() очень прост. Мы используем его для настройки параметров каскадной таблицы стилей (Cascading Style Sheet, CSS) для текста заголовка фильма. В данном случае мы хотим, чтобы заголовок был выделен жирным шрифтом и имел очень большой размер:

```
private Component buildTitle() {
    HorizontalLayout layout = new HorizontalLayout();
```

```
    title.getStyle()
        .set("font-weight", "bold")
        .set("font-size", "x-large");

    layout.add(title, year);

    return layout;
}
```

Метод `buildGenreData()` инициализирует компоненты жанров (`Span`). Мы настраиваем тему `Span` быть значком и устанавливаем, что каждый жанр изначально невидим. Таким образом, если фильм имеет только один или два жанра, нам не придется отображать пустой значок:

```
private Component buildGenreData() {
    HorizontalLayout layout = new HorizontalLayout();

    genre1.getElement().getThemeList().add("badge");
    genre1.setVisible(false);

    genre2.getElement().getThemeList().add("badge");
    genre2.setVisible(false);

    genre3.getElement().getThemeList().add("badge");
    genre3.setVisible(false);

    layout.add(genre1, genre2, genre3);

    return layout;
}
```

Одна из проблем, с которой можно столкнуться, заключается в том, что данные о фильмах находятся в свободном доступе и являются общественным достоянием. Однако изображения, связанные с этими фильмами, являются интеллектуальной собственностью киностудий, поэтому мы не можем их предоставить. Но что мы можем сделать, так это создать механизм, позволяющий пользователям самим загружать изображения для каждого фильма. Метод `buildImageUpdateControls()` обеспечит эту функциональность.

Сначала создадим кнопку загрузки, затем определим буфер памяти для загрузки файлов изображений и используем его для инстанцирования компонента `Upload`. Можно добавить кнопку загрузки `upload` в компонент и настроить его так, чтобы он принимал только изображения в формате `JPG` или `PNG`. В компонент `Upload` также добавим слушателя, который будет срабатывать при успешной загрузке файла изображения (либо с помощью выбора файла, либо перетаскиванием):

```
private Component buildImageUpdateControls() {
    HorizontalLayout layout = new HorizontalLayout();
```

```
    upButton = new Button("Upload JPG or PNG");
    upButton.addThemeVariants(ButtonVariant.LUMO_PRIMARY);
    buffer = new MemoryBuffer();
    upload = new Upload(buffer);
    upload.setUploadButton(upButton);
    upload.setAcceptedFileTypes("image/png", "image/jpg",
            "image/jpeg");
    upload.addSucceededListener(event -> {
        // генерация имени файла
        int movieID = Integer.parseInt(movieId.getValue());
        StringBuilder filename = new StringBuilder("images/");
        String mimeType = event.getMIMEType();

        filename.append("movie_");
        filename.append(movieId.getValue());

        if (mimeType.equals("image/jpg") ||
                mimeType.equals("image/jpeg")) {
            filename.append(".jpg");
        } else {
            // мы принимаем только jpeg или png, поэтому обрабатываем png
            filename.append(".png");
        }

        InputStream inStream = buffer.getInputStream();
        try {
            // получаем файл из памяти
            byte[] byteBuffer = new byte[inStream.available()];
            inStream.read(byteBuffer);

            // запись на диск
            File destination = new File(filename.toString());
            Files.write(byteBuffer, destination);
            inStream.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

        // отображаем новое изображение
        getImage(movieID);
        upload.clearFileList();
    });

    layout.add(upload);

    return layout;
}
```

Слушатель сначала сгенерирует имя файла `filename` для только что загруженного изображения и добавит соответствующее расширение (`.jpg` или `.png`) в зависимости от типа изображения. После этого извлекаются данные изображения из входного потока буфера и записываются на диск. Последней задачей, которую должен выполнить слушатель, является вывод изображения на экран и очистка имени файла `filename` из компонента загрузки `upload`.

Далее напишем метод для создания основных данных изображения. Метод `buildImageData()` установит изображение по умолчанию в случае, когда изображения нет. Мы также настроим высоту изображения с помощью CSS на 300 пикселей:

```
private Component buildImageData() {
    HorizontalLayout layout = new HorizontalLayout();

    try {
        FileInputStream fileStream = new FileInputStream(
                new File(noImageFile));
        noImgFileStream = new StreamResource("image",() -> {
            return fileStream;
        });
        image.setSrc(noImgFileStream);
    } catch (Exception e) {
        e.printStackTrace();
    }

    getImage(-1);
    image.setHeight("300px");
    layout.add(image);

    return layout;
}
```

Далее напишем метод `getImage()`, который абстрагирует детали загрузки изображения. Это позволит использовать его как для основного изображения, так и для рекомендуемых изображений фильмов:

```
private void getImage(int movieID) {
    StreamResource src = getImageStream(movieID);
    image.setSrc(src);
}
```

Метод `getImageStream()` будет осуществлять загрузку запрошенного изображения с локального диска. Он сформирует имя файла `filename` и сначала попытается загрузить изображение в формате `jpg`. Если это не удастся, то будет сделана попытка загрузить изображение в формате `png`. Если файл изображения корректный и изображение может быть загружено, то возвращается его файловый поток:

```
private StreamResource getImageStream(int movieID) {
    StringBuilder filename = new StringBuilder("images/");
```

```
    if (movieID >= 0) {
        filename.append("movie_");
        filename.append(movieID);
        filename.append(".jpg");
        // сначала пробуем jpg

        if (!new File(filename.toString()).exists()) {
            // затем пробуем png
            filename = new StringBuilder("images/");
            filename.append("movie_");
            filename.append(movieID);
            filename.append(".png");
        }

    try {
            FileInputStream imgFileStream = new FileInputStream(
                    new File(filename.toString()));
            StreamResource src = new StreamResource("image",() -> {
                return imgFileStream;
            });
            return src;
        } catch (FileNotFoundException ex) {
            // файл не найден;
            // установлено значение No Image для файлового потока
            return noImgFileStream;
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    return noImgFileStream;
}
```

Далее создадим метод, собирающий большую часть оставшихся некатегоризированных данных. К ним относятся `movieId`, идентификатор фильма на сайте **imdb.com** и язык оригинала фильма. Поскольку эти данные присваиваются компонентам `TextField`, они будут установлены как `ReadOnly`:

```
private Component buildMovieMetaData() {
    HorizontalLayout layout = new HorizontalLayout();

    movieId.setReadOnly(true);
    imdb.setReadOnly(true);
    language.setReadOnly(true);

    layout.add(movieId, imdb, language);

    return layout;
}
```

После этого соберем финансовые данные о фильме. К ним относятся бюджет фильма `budget`, доход `revenue` и дата выхода `releaseDate`. Мы также хотим убедиться, что соответствующие текстовые поля `TextFields` установлены в режим `ReadOnly`:

```
private Component buildFinancialData() {
    HorizontalLayout layout = new HorizontalLayout();

    releaseDate.setReadOnly(true);
    budget.setReadOnly(true);
    revenue.setReadOnly(true);

    layout.add(releaseDate, budget, revenue);

    return layout;
}
```

Далее создадим метод, собирающий данные о рейтинге фильма. Здесь всего две точки данных: средняя оценка по шкале от 1 до 10 и общее количество голосов. Мы также используем метод `buildRatingData()` для добавления золотой звезды (значка) в текстовое поле `voteRating`:

```
private Component buildRatingData() {
    HorizontalLayout layout = new HorizontalLayout();

    voteRating.setReadOnly(true);
    votes.setReadOnly(true);

    Icon star = new Icon(VaadinIcon.STAR);
    star.setColor("gold");
    voteRating.setPrefixComponent(star);

    layout.add(voteRating, votes);

    return layout;
}
```

Следующий метод будет получать данные с внешних источников. Мы напишем метод `buildWebstieData()` для сбора и соответствующего отображения данных об официальном сайте фильма и записи о фильме на **imdb.com**:

```
private Component buildWebsiteData() {
    HorizontalLayout layout = new HorizontalLayout();

    website.setReadOnly(true);
    website.setWidth("400px");
    imdbWebsite.setReadOnly(true);
    imdbWebsite.setWidth("400px");
```

```
    layout.add(website, imdbWebsite);

    return layout;
}
```

Следующий метод — довольно сложный. Мы создадим области отображения для пяти рекомендаций, которые появляются для каждого фильма. Каждая рекомендация будет иметь свой собственный вертикальный макет высотой 100 пикселей. У нее также будет слушатель, позволяющий пользователю щелкнуть по рекомендации и отобразить фильм. Кроме того, размер текста рекомендации будет установлен на совсем маленький (extra-small) с помощью CSS:

```
private Component buildRecommendations() {
    HorizontalLayout layout = new HorizontalLayout();

    VerticalLayout vLayout1 = new VerticalLayout();
    recImage1.setHeight("100px");
    recImage1.addClickListener( event -> {
        String[] movieText = recommendation1.getText().split(" - ");
        queryField.setValue(movieText[0]);
        queryBy.setValue("ID");
        refreshData();
    });
    recommendation1.getStyle()
        .set("font-size", "x-small");
    vLayout1.add(recImage1,recommendation1);

    VerticalLayout vLayout2 = new VerticalLayout();
    recImage2.setHeight("100px");
    recImage2.addClickListener( event -> {
        String[] movieText = recommendation2.getText().split(" - ");
        queryField.setValue(movieText[0]);
        queryBy.setValue("ID");
        refreshData();
    });
    recommendation2.getStyle()
        .set("font-size", "x-small");
    vLayout2.add(recImage2,recommendation2);
```

Приведенные выше блоки кода предназначены для создания отображений для первых двух рекомендаций. К сожалению, поскольку они уже добавлены в основной макет в виде компонентов, мы можем работать только с одной из них за раз. Следующий код повторяет логику для трех оставшихся рекомендаций:

```
    VerticalLayout vLayout3 = new VerticalLayout();
    recImage3.setHeight("100px");
    recImage3.addClickListener( event -> {
        String[] movieText = recommendation3.getText().split(" - ");
        queryField.setValue(movieText[0]);
```

```
        queryBy.setValue("ID");
        refreshData();
    });
    recommendation3.getStyle()
        .set("font-size", "x-small");
    vLayout3.add(recImage3,recommendation3);

    VerticalLayout vLayout4 = new VerticalLayout();
    recImage4.setHeight("100px");
    recImage4.addClickListener( event -> {
    String[] movieText = recommendation4.getText().split(" - ");
        queryField.setValue(movieText[0]);
        queryBy.setValue("ID");
        refreshData();
    });
    recommendation4.getStyle()
        .set("font-size", "x-small");
    vLayout4.add(recImage4,recommendation4);

    VerticalLayout vLayout5 = new VerticalLayout();
    recImage5.setHeight("100px");
    recImage5.addClickListener( event -> {
        String[] movieText = recommendation5.getText().split(" - ");
        queryField.setValue(movieText[0]);
        queryBy.setValue("ID");
        refreshData();
    });
    recommendation5.getStyle()
        .set("font-size", "x-small");
    vLayout5.add(recImage5,recommendation5);

    layout.add(vLayout1,vLayout2,vLayout3,vLayout4,vLayout5);

    return layout;
}
```

При отображении данных о доходах и бюджете фильма необходимо соответствующим образом отформатировать их. Метод `formatMoney()` справится с этой задачей, используя свойство `enUS`, которое было определено нами ранее для локали[3] English-US:

```
private String formatMoney(Long money) {
    NumberFormat numFormat = NumberFormat.getCurrencyInstance(enUS);
    return numFormat.format(Double.parseDouble(money.toString()));
}
```

[3] Локаль — набор параметров, определяющий локализованные настройки пользовательского интерфейса. — *Прим. ред.*

Кроме того, нам понадобится метод для проверки того, является ли строка числом. Это будет сделано с помощью регулярного выражения (numericPattern), которое мы определили ранее. Взгляните на следующий код:

```
private boolean isNumeric(String value) {
    if (value == null) {
        return false;
    }

    return numericPattern.matcher(value).matches();
}
```

В методе refreshData() много всего, но он также управляет логикой отображения информации для каждого фильма. Поскольку этот метод в значительной степени зависит от взаимодействия пользователя с панелью запроса, начнем с определения того, запрашивает ли он фильм по ID или по названию. Исходя из этого, определим, как получить фильм (по ID или по названию) и извлечь информацию о нем из базы данных:

```
private void refreshData() {
    Optional<Movie> optionalMovie;

    if (queryBy.getValue().equals("ID")) {
        if (isNumeric(queryField.getValue())) {
            optionalMovie = controller.getMovieByMovieId(
                    Integer.parseInt(queryField.getValue()))
                .getBody();
        } else {
            optionalMovie = Optional.ofNullable(null);
        }
    } else {
    // имя
    optionalMovie = controller.getMovieByTitle(
            queryField.getValue()).getBody();
    }
```

Когда у нас есть фильм, можно начать присваивать данные свойствам класса:

```
    if (optionalMovie != null && optionalMovie.isPresent()) {

        Movie movie = optionalMovie.get();

        String strTitle = movie.getTitle();
        String strDescription = movie.getDescription();
        LocalDate ldReleaseDate = movie.getReleaseDate();
        Integer intYear = movie.getYear();
        mapGenres = movie.getGenres();
        String strWebsiteUrl = movie.getWebsite();
        String strImdbId = movie.getImdbId();
        String strLanguage = movie.getOriginalLanguage();
```

```
title.setText(strTitle);
description.setText(strDescription);

if (movie.getMovieId() != null) {
    movieId.setValue(movie.getMovieId().toString());
    // когда есть ID, можно получить изображение
    getImage(movie.getMovieId());
}

if (movie.getReleaseDate() != null) {
    releaseDate.setValue(ldReleaseDate.toString());
}

if (intYear != null) {
    year.setText(intYear.toString());
}
```

Здесь будем обрабатывать значки жанров. Начнем с того, что установим их все невидимыми. Затем пройдем по карте MapGenres, устанавливая текст жанра для значков по порядку:

```
// обрабатываем значки жанров
int genreCounter = 0;
genre1.setVisible(false);
genre2.setVisible(false);
genre3.setVisible(false);

for (String genre : mapGenres.values()) {

    switch (genreCounter) {
    case 0:
        genre1.setText(genre);;
        genre1.setVisible(true);
        break;
    case 1:
        genre2.setText(genre);
        genre2.setVisible(true);
        break;
    case 2:
        genre3.setText(genre);
        genre3.setVisible(true);
        break;
    default:
        break;
    }

    genreCounter++;
}
```

Далее зададим данные веб-сайта и языка, а с ними бюджет budget и доход revenue. budget и revenue также вызывают метод formatMoney(). Мы также установим данные о количестве голосов из соответствующих источников в векторе фильма:

```
website.setValue(strWebsiteUrl);
imdbWebsite.setValue("https://www.imdb.com/title/"
        +strImdbId);
imdb.setValue(strImdbId);
language.setValue(strLanguage);

if (movie.getBudget() != null) {
    budget.setValue(formatMoney(movie.getBudget()));
}

if (movie.getRevenue() != null) {
    revenue.setValue(formatMoney(movie.getRevenue()));
}

if (movie.getVector() != null) {
    voteRating.setValue(movie.getVector().get(5).toString());
    votes.setValue(movie.getVector().get(6).toString());
}
```

Далее подготовим данные для каждой из рекомендаций фильма:

```
if (movieId.getValue().length() > 0) {
    Integer movieID = Integer.parseInt(movieId.getValue());

    List<Movie> recommendedMovies =
            controller.getMovieRecommendationsById(movieID)
        .getBody();

    int movieCounter = 0;
    for (Movie recMovie : recommendedMovies) {

        StreamResource resource =
                getImageStream(recMovie.getMovieId());
        StringBuilder titleText = new StringBuilder(
                recMovie.getMovieId().toString());
        titleText.append(" - ");
        titleText.append(recMovie.getTitle());

        switch (movieCounter) {
        case 0:
            recImage1.setSrc(resource);
            recommendation1.setText(titleText.toString());
            break;
        case 1:
            recImage2.setSrc(resource);
            recommendation2.setText(titleText.toString());
            break;
```

```
            case 2:
                recImage3.setSrc(resource);
                recommendation3.setText(titleText.toString());
                break;
            case 3:
                recImage4.setSrc(resource);
                recommendation4.setText(titleText.toString());
                break;
            default:
                recImage5.setSrc(resource);
                recommendation5.setText(titleText.toString());
            }

            movieCounter++;

                if (movieCounter > 4) {
                    break;
                }
            }
        }
    } else {
        Notification.show("No movie found for those query parameters."
                ,5000, Position.TOP_CENTER);
    }
}
```

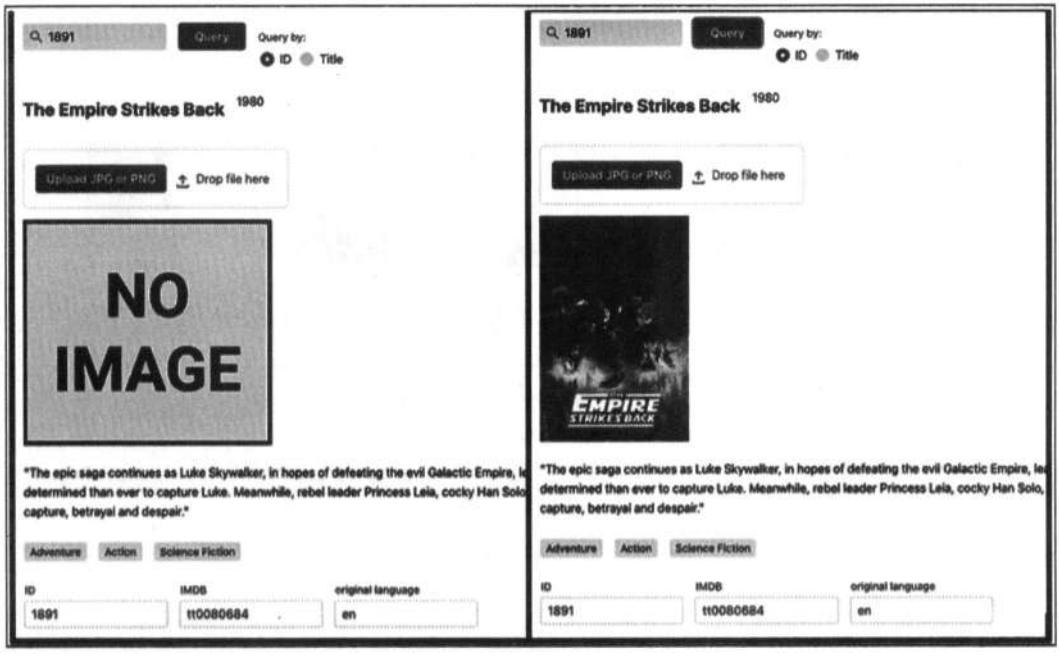



**Рис. 10.4.** Скриншот для фильма "The Empire Strikes Back" до и после того, как загруженное изображение было перенесено в компонент загрузки

Наконец, в случае, если фильм не может быть найден по введенному пользователем запросу, с помощью класса `Notification` из Vaadin выведем сообщение на пять секунд в верхней части страницы.

Теперь, когда представление завершено, можно запустить наше приложение. Перейдите в браузер по адресу **http://127.0.0.1:8080/**. Приложение должно быть похоже на то, что изображено на рис. 10.4.

Перейдя на страницу конкретного фильма, можно прокрутить страницу вниз и увидеть больше данных о нем (рис. 10.5).

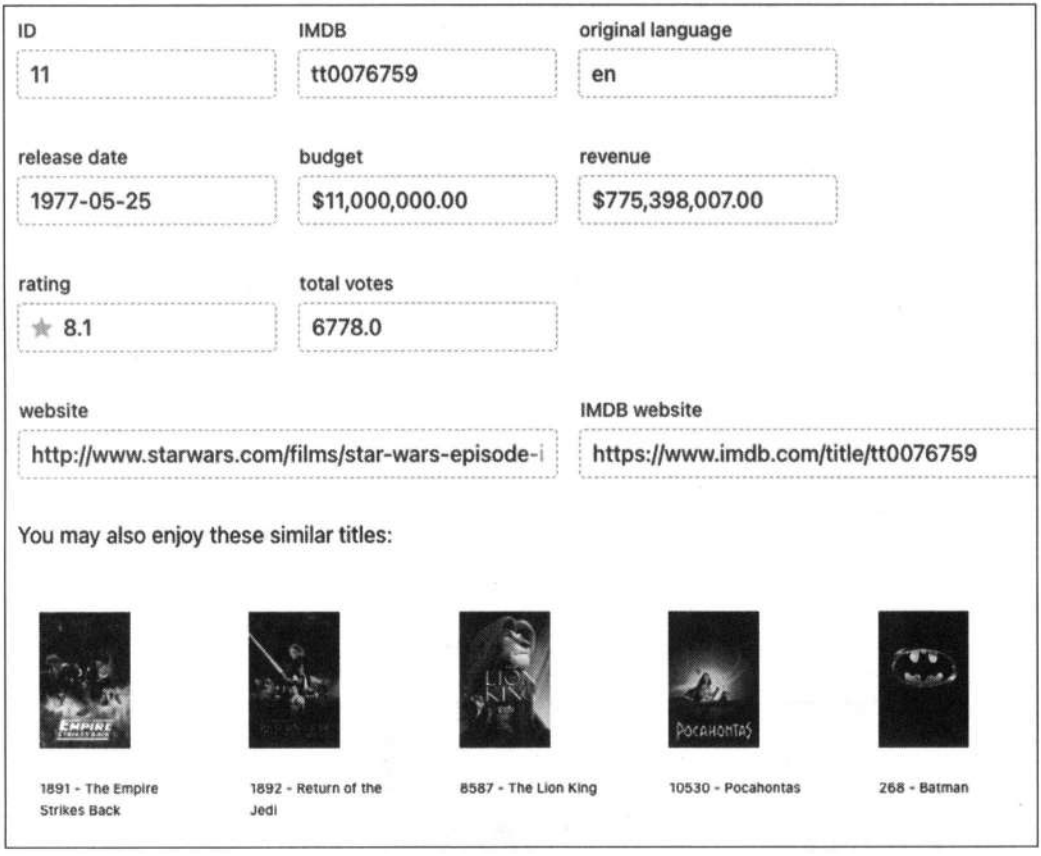



**Рис. 10.5.** Вторая страница данных по фильму "Star Wars", внизу показаны рекомендации

# Заключение

В этой главе мы применили многое из того, что узнали в предыдущих главах, и создали довольно сложное приложение. Мы также познакомились с типом `Optional` и векторным поиском (`vector`).

Одной из самых сложных частей этого проекта было создание специального приложения для обработки данных. К сожалению, это суровая реалия области обработки данных. Простого импорта данных или перетаскивания CSV-файла в базу данных редко бывает достаточно. Часто данные должны быть подвергнуты различным уровням предварительной обработки, прежде чем их можно будет считать пригодными для использования. Даже файл movies_metadata.csv, загруженный с сайта Kaggle "как есть", оказался непригодным для работы, поскольку несколько записей содержали лишние разрывы строк в описании, что сбивало процесс загрузки данных.

Несмотря на свою функциональность, это приложение не является полным. Разработчики могут сделать многое для расширения его функциональности. Например, можно написать функцию, позволяющую пользователю редактировать данные фильма или даже вводить новый фильм с помощью пользовательского интерфейса. Также можно позволить пользователям оценивать каждый фильм, а затем пересчитывать данные рейтинга и векторное вложение. Кроме того, можно использовать LLM для создания векторного вложения на основе названий фильмов, а затем использовать метод сходства для векторного поиска на основе этих данных. Возможности безграничны, как звезды на небе.

## Важно помнить

- Базы данных Cassandra, созданные с помощью Astra, требуют специального экземпляра с поддержкой `vector`.
- Запросы к вторичным индексам в распределенной базе данных предназначены для удобства, а не для повышения производительности.
- Векторный поисковый запрос с `ANN` на CQL не обязательно должен использовать условие `WHERE`, но требует наличия условий `ORDER BY` и `LIMIT`.
- Тип `Optional` в Java — отличный способ предотвратить `NullPointerExceptions` при запросе к базе данных.
- `Vaadin` использует метод `getStyle()` для раскрытия базового CSS своих компонентов. Это позволяет выполнять точную визуальную настройку и конфигурацию.

# Приложения

**ПРИЛОЖЕНИЕ 1.** Ссылки

**ПРИЛОЖЕНИЕ 2.** Таблица преобразования UTF

**ПРИЛОЖЕНИЕ 3.** Справочник команд баз данных

**ПРИЛОЖЕНИЕ 4.** Общие коды ответов HTTP

**ПРИЛОЖЕНИЕ 5.** Основные цветовые коды

**ПРИЛОЖЕНИЕ 6.** Сбор мусора

# Приложение 1

# Ссылки


Baeldung (2022). Java Preview Features. Размещен на сайте (04.12.2023 г.): **https://www.baeldung.com/java-preview-features**.

Baeldung (2022b). A Guide to Java Regular Expressions API. Размещен на сайте (06.05.2023 г.): **https://www.baeldung.com/regular-expressions-java**.

Baeldung (2023). Ignoring Commas in Quotes When Splitting a Comma-separated String. Размещен на сайте (18.08.2023 г.): **https://www.baeldung.com/java-split-string-commas**.

Brewer E., Fox A. (1999). Harvest, Yield, and Scalable Tolerant Systems. Proceedings of the Seventh Workshop on Hot Topics in Operating Systems, Rio Rico, AZ, USA. Pages 174–178.

Darcy J. (2021). Floating-Point Arithmetic: What Every Java Programmer Should Know! Retrieved via the @Java YouTube-канал на сайте (02.06.2023 г.): **https://www.youtube.com/watch?v=ajaHQ9S4uTA**.

Gillis A., Lewis S. (2021). Object Oriented Programming (OOP). TechTarget. Размещен на сайте (01.04.2023 г.): **https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP**.

Hanson B. (2015). How Steve Wozniak's Breakout Defined Apple's Future. Game Informer. Размещен на сайте (30.09.2023 г.): **https://www.gameinformer.com/b/features/archive/2015/10/09/how-steve-wozniak-s-breakout-defined-apple-s-future.aspx**.

IBM (2022). Big Decimal (BigDecimal) Support for Real Numbers. IBM Sterling B2B Integrator documentation. Размещен на сайте (03.06.2023 г.): **https://www.ibm.com/docs/en/b2b-integrator/5.2?topic=fdif-big-decimal-bigdecimal-support-real-numbers-2**.

IMDB (2023). IMDB.com: Overview | LinkedIn. Размещен на сайте (06.08.2023 г.): **https://www.linkedin.com/company/imdb-com/**.

JavaTPoint (2021). Red-black tree in Data Structure. JavaTPoint. Размещен на сайте (21.05.2023 г.): **https://www.javatpoint.com/red-black-tree**.

Oracle (2004). Introduction to Oracle SQL – History of SQL. Oracle Database SQL Reference. Oracle Corporation. Размещен на сайте (19.06.2023 г.): **https://docs.oracle.com/cd/B13789_01/server.101/b10759/intro.htm**.

Parlog N. (2023). New (Sequenced) Collections in Java 21 – Inside Java Newscast #45. Размещен на сайте (08.04.2023 г.): **https://nipafx.dev/inside-java-newscast-45/**.

Tyson M. (2022). Intro to virtual threads: A new approach to Java concurrency. InfoWorld. Размещен на сайте (07.04.2023 г.): **https://www.infoworld.com/article/3678148/intro-to-virtual-threads-a-new-approach-to-java-concurrency.html**.

Wiffin E. (2017). Floating Point Math. Размещен на сайте (03.06.2023 г.): **https://0.30000000000000004.com/**.

# Приложение 2

# Таблица преобразования UTF

Приведем таблицу преобразования UTF:

| Dec | Oct | Hex | UTF | Binary | Dec | Oct | Hex | UTF | Binary |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | NULL | 00000000 | 128 | 200 | 80 | | 10000000 |
| 1 | 1 | 1 | SOH | 00000001 | 129 | 201 | 81 | | 10000001 |
| 2 | 2 | 2 | STX | 00000010 | 130 | 202 | 82 | | 10000010 |
| 3 | 3 | 3 | ETX | 00000011 | 131 | 203 | 83 | | 10000011 |
| 4 | 4 | 4 | EOT | 00000100 | 132 | 204 | 84 | | 10000100 |
| 5 | 5 | 5 | ENQ | 00000101 | 133 | 205 | 85 | | 10000101 |
| 6 | 6 | 6 | ACK | 00000110 | 134 | 206 | 86 | | 10000110 |
| 7 | 7 | 7 | BELL | 00000111 | 135 | 207 | 87 | | 10000111 |
| 8 | 10 | 8 | BSP | 00001000 | 136 | 210 | 88 | | 10001000 |
| 9 | 11 | 9 | TAB | 00001001 | 137 | 211 | 89 | | 10001001 |
| 10 | 12 | A | LF | 00001010 | 138 | 212 | 8A | | 10001010 |
| 11 | 13 | B | VTAB | 00001011 | 139 | 213 | 8B | | 10001011 |
| 12 | 14 | C | FF | 00001100 | 140 | 214 | 8C | | 10001100 |
| 13 | 15 | D | CR | 00001101 | 141 | 215 | 8D | | 10001101 |
| 14 | 16 | E | | 00001110 | 142 | 216 | 8E | | 10001110 |
| 15 | 17 | F | | 00001111 | 143 | 217 | 8F | | 10001111 |
| 16 | 20 | 10 | | 00010000 | 144 | 220 | 90 | | 10010000 |
| 17 | 21 | 11 | | 00010001 | 145 | 221 | 91 | | 10010001 |
| 18 | 22 | 12 | | 00010010 | 146 | 222 | 92 | | 10010010 |
| 19 | 23 | 13 | | 00010011 | 147 | 223 | 93 | | 10010011 |
| 20 | 24 | 14 | | 00010100 | 148 | 224 | 94 | | 10010100 |
| 21 | 25 | 15 | | 00010101 | 149 | 225 | 95 | | 10010101 |
| 22 | 26 | 16 | | 00010110 | 150 | 226 | 96 | | 10010110 |

| Dec | Oct | Hex | UTF | Binary | Dec | Oct | Hex | UTF | Binary |
|---|---|---|---|---|---|---|---|---|---|
| 23 | 27 | 17 | | 00010111 | 151 | 227 | 97 | | 10010111 |
| 24 | 30 | 18 | | 00011000 | 152 | 230 | 98 | | 10011000 |
| 25 | 31 | 19 | | 00011001 | 153 | 231 | 99 | | 10011001 |
| 26 | 32 | 1A | | 00011010 | 154 | 232 | 9A | | 10011010 |
| 27 | 33 | 1B | | 00011011 | 155 | 233 | 9B | | 10011011 |
| 28 | 34 | 1C | | 00011100 | 156 | 234 | 9C | | 10011100 |
| 29 | 35 | 1D | | 00011101 | 157 | 235 | 9D | | 10011101 |
| 30 | 36 | 1E | | 00011110 | 158 | 236 | 9E | | 10011110 |
| 31 | 37 | 1F | | 00011111 | 159 | 237 | 9F | | 10011111 |
| 32 | 40 | 20 | SPACE | 00100000 | 160 | 240 | A0 | | 10100000 |
| 33 | 41 | 21 | ! | 00100001 | 161 | 241 | A1 | ¡ | 10100001 |
| 34 | 42 | 22 | " | 00100010 | 162 | 242 | A2 | ¢ | 10100010 |
| 35 | 43 | 23 | # | 00100011 | 163 | 243 | A3 | £ | 10100011 |
| 36 | 44 | 24 | $ | 00100100 | 164 | 244 | A4 | ¤ | 10100100 |
| 37 | 45 | 25 | % | 00100101 | 165 | 245 | A5 | ¥ | 10100101 |
| 38 | 46 | 26 | & | 00100110 | 166 | 246 | A6 | ¦ | 10100110 |
| 39 | 47 | 27 | ' | 00100111 | 167 | 247 | A7 | § | 10100111 |
| 40 | 50 | 28 | ( | 00101000 | 168 | 250 | A8 | ¨ | 10101000 |
| 41 | 51 | 29 | ) | 00101001 | 169 | 251 | A9 | © | 10101001 |
| 42 | 52 | 2A | * | 00101010 | 170 | 252 | AA | ª | 10101010 |
| 43 | 53 | 2B | + | 00101011 | 171 | 253 | AB | « | 10101011 |
| 44 | 54 | 2C | , | 00101100 | 172 | 254 | AC | ¬ | 10101100 |
| 45 | 55 | 2D | - | 00101101 | 173 | 255 | AD | | 10101101 |
| 46 | 56 | 2E | . | 00101110 | 174 | 256 | AE | ® | 10101110 |
| 47 | 57 | 2F | / | 00101111 | 175 | 257 | AF | ¯ | 10101111 |
| 48 | 60 | 30 | 0 | 00110000 | 176 | 260 | B0 | ° | 10110000 |
| 49 | 61 | 31 | 1 | 00110001 | 177 | 261 | B1 | ± | 10110001 |
| 50 | 62 | 32 | 2 | 00110010 | 178 | 262 | B2 | ² | 10110010 |
| 51 | 63 | 33 | 3 | 00110011 | 179 | 263 | B3 | ³ | 10110011 |
| 52 | 64 | 34 | 4 | 00110100 | 180 | 264 | B4 | ´ | 10110100 |
| 53 | 65 | 35 | 5 | 00110101 | 181 | 265 | B5 | µ | 10110101 |
| 54 | 66 | 36 | 6 | 00110110 | 182 | 266 | B6 | ¶ | 10110110 |
| 55 | 67 | 37 | 7 | 00110111 | 183 | 267 | B7 | · | 10110111 |
| 56 | 70 | 38 | 8 | 00111000 | 184 | 270 | B8 | ¸ | 10111000 |

| Dec | Oct | Hex | UTF | Binary | Dec | Oct | Hex | UTF | Binary |
|---|---|---|---|---|---|---|---|---|---|
| 57 | 71 | 39 | 9 | 00111001 | 185 | 271 | B9 | ¹ | 10111001 |
| 58 | 72 | 3A | : | 00111010 | 186 | 272 | BA | º | 10111010 |
| 59 | 73 | 3B | ; | 00111011 | 187 | 273 | BB | " | 10111011 |
| 60 | 74 | 3C | < | 00111100 | 188 | 274 | BC | ¼ | 10111100 |
| 61 | 75 | 3D | = | 00111101 | 189 | 275 | BD | ½ | 10111101 |
| 62 | 76 | 3E | > | 00111110 | 190 | 276 | BE | ¾ | 10111110 |
| 63 | 77 | 3F | ? | 00111111 | 191 | 277 | BF | ¿ | 10111111 |
| 64 | 100 | 40 | @ | 01000000 | 192 | 300 | C0 | À | 11000000 |
| 65 | 101 | 41 | A | 01000001 | 193 | 301 | C1 | Á | 11000001 |
| 66 | 102 | 42 | B | 01000010 | 194 | 302 | C2 | Â | 11000010 |
| 67 | 103 | 43 | C | 01000011 | 195 | 303 | C3 | Ã | 11000011 |
| 68 | 104 | 44 | D | 01000100 | 196 | 304 | C4 | Ä | 11000100 |
| 69 | 105 | 45 | E | 01000101 | 197 | 305 | C5 | Å | 11000101 |
| 70 | 106 | 46 | F | 01000110 | 198 | 306 | C6 | Æ | 11000110 |
| 71 | 107 | 47 | G | 01000111 | 199 | 307 | C7 | Ç | 11000111 |
| 72 | 110 | 48 | H | 01001000 | 200 | 310 | C8 | È | 11001000 |
| 73 | 111 | 49 | I | 01001001 | 201 | 311 | C9 | É | 11001001 |
| 74 | 112 | 4A | J | 01001010 | 202 | 312 | CA | Ê | 11001010 |
| 75 | 113 | 4B | K | 01001011 | 203 | 313 | CB | Ë | 11001011 |
| 76 | 114 | 4C | L | 01001100 | 204 | 314 | CC | Ì | 11001100 |
| 77 | 115 | 4D | M | 01001101 | 205 | 315 | CD | Í | 11001101 |
| 78 | 116 | 4E | N | 01001110 | 206 | 316 | CE | Î | 11001110 |
| 79 | 117 | 4F | O | 01001111 | 207 | 317 | CF | Ï | 11001111 |
| 80 | 120 | 50 | P | 01010000 | 208 | 320 | D0 | Ð | 11010000 |
| 81 | 121 | 51 | Q | 01010001 | 209 | 321 | D1 | Ñ | 11010001 |
| 82 | 122 | 52 | R | 01010010 | 210 | 322 | D2 | Ò | 11010010 |
| 83 | 123 | 53 | S | 01010011 | 211 | 323 | D3 | Ó | 11010011 |
| 84 | 124 | 54 | T | 01010100 | 212 | 324 | D4 | Ô | 11010100 |
| 85 | 125 | 55 | U | 01010101 | 213 | 325 | D5 | Õ | 11010101 |
| 86 | 126 | 56 | V | 01010110 | 214 | 326 | D6 | Ö | 11010110 |
| 87 | 127 | 57 | W | 01010111 | 215 | 327 | D7 | × | 11010111 |
| 88 | 130 | 58 | X | 01011000 | 216 | 330 | D8 | Ø | 11011000 |
| 89 | 131 | 59 | Y | 01011001 | 217 | 331 | D9 | Ù | 11011001 |
| 90 | 132 | 5A | Z | 01011010 | 218 | 332 | DA | Ú | 11011010 |

| Dec | Oct | Hex | UTF | Binary | Dec | Oct | Hex | UTF | Binary |
|---|---|---|---|---|---|---|---|---|---|
| 91 | 133 | 5B | [ | 01011011 | 219 | 333 | DB | Û | 11011011 |
| 92 | 134 | 5C | \ | 01011100 | 220 | 334 | DC | Ü | 11011100 |
| 93 | 135 | 5D | ] | 01011101 | 221 | 335 | DD | Ý | 11011101 |
| 94 | 136 | 5E | ^ | 01011110 | 222 | 336 | DE | Þ | 11011110 |
| 95 | 137 | 5F | _ | 01011111 | 223 | 337 | DF | ß | 11011111 |
| 96 | 140 | 60 | ` | 01100000 | 224 | 340 | E0 | à | 11100000 |
| 97 | 141 | 61 | a | 01100001 | 225 | 341 | E1 | á | 11100001 |
| 98 | 142 | 62 | b | 01100010 | 226 | 342 | E2 | â | 11100010 |
| 99 | 143 | 63 | c | 01100011 | 227 | 343 | E3 | ã | 11100011 |
| 100 | 144 | 64 | d | 01100100 | 228 | 344 | E4 | ä | 11100100 |
| 101 | 145 | 65 | e | 01100101 | 229 | 345 | E5 | å | 11100101 |
| 102 | 146 | 66 | f | 01100110 | 230 | 346 | E6 | æ | 11100110 |
| 103 | 147 | 67 | g | 01100111 | 231 | 347 | E7 | ç | 11100111 |
| 104 | 150 | 68 | h | 01101000 | 232 | 350 | E8 | è | 11101000 |
| 105 | 151 | 69 | i | 01101001 | 233 | 351 | E9 | é | 11101001 |
| 106 | 152 | 6A | j | 01101010 | 234 | 352 | EA | ê | 11101010 |
| 107 | 153 | 6B | k | 01101011 | 235 | 353 | EB | ë | 11101011 |
| 108 | 154 | 6C | l | 01101100 | 236 | 354 | EC | ì | 11101100 |
| 109 | 155 | 6D | m | 01101101 | 237 | 355 | ED | í | 11101101 |
| 110 | 156 | 6E | n | 01101110 | 238 | 356 | EE | î | 11101110 |
| 111 | 157 | 6F | o | 01101111 | 239 | 357 | EF | ï | 11101111 |
| 112 | 160 | 70 | p | 01110000 | 240 | 360 | F0 | ð | 11110000 |
| 113 | 161 | 71 | q | 01110001 | 241 | 361 | F1 | ñ | 11110001 |
| 114 | 162 | 72 | r | 01110010 | 242 | 362 | F2 | ò | 11110010 |
| 115 | 163 | 73 | s | 01110011 | 243 | 363 | F3 | ó | 11110011 |
| 116 | 164 | 74 | t | 01110100 | 244 | 364 | F4 | ô | 11110100 |
| 117 | 165 | 75 | u | 01110101 | 245 | 365 | F5 | õ | 11110101 |
| 118 | 166 | 76 | v | 01110110 | 246 | 366 | F6 | ö | 11110110 |
| 119 | 167 | 77 | w | 01110111 | 247 | 367 | F7 | ÷ | 11110111 |
| 120 | 170 | 78 | x | 01111000 | 248 | 370 | F8 | ø | 11111000 |
| 121 | 171 | 79 | y | 01111001 | 249 | 371 | F9 | ù | 11111001 |
| 122 | 172 | 7A | z | 01111010 | 250 | 372 | FA | ú | 11111010 |
| 123 | 173 | 7B | { | 01111011 | 251 | 373 | FB | û | 11111011 |
| 124 | 174 | 7C | \| | 01111100 | 252 | 374 | FC | ü | 11111100 |

| Dec | Oct | Hex | UTF | Binary | Dec | Oct | Hex | UTF | Binary |
|---|---|---|---|---|---|---|---|---|---|
| 125 | 175 | 7D | } | 01111101 | 253 | 375 | FD | ý | 11111101 |
| 126 | 176 | 7E | ~ | 01111110 | 254 | 376 | FE | þ | 11111110 |
| 127 | 177 | 7F | | 01111111 | 255 | 377 | FF | ÿ | 11111111 |

Перейдем к кодам ASCII Art, используемым в *главе 3 "Строки, символы и регулярные выражения"*. Для краткости удалено восьмеричное преобразование:

| Dec | Hex | UTF | Binary | Dec | Hex | UTF | Binary |
|---|---|---|---|---|---|---|---|
| 9472 | 2500 | ─ | 10010100000000 | 9600 | 2580 | ▀ | 10010110000000 |
| 9473 | 2501 | ━ | 10010100000001 | 9601 | 2581 | ▁ | 10010110000001 |
| 9474 | 2502 | │ | 10010100000010 | 9602 | 2582 | ▂ | 10010110000010 |
| 9475 | 2503 | ┃ | 10010100000011 | 9603 | 2583 | ▃ | 10010110000011 |
| 9476 | 2504 | ┄ | 10010100000100 | 9604 | 2584 | ▄ | 10010110000100 |
| 9477 | 2505 | ┅ | 10010100000101 | 9605 | 2585 | ▅ | 10010110000101 |
| 9478 | 2506 | ┆ | 10010100000110 | 9606 | 2586 | ▆ | 10010110000110 |
| 9479 | 2507 | ┇ | 10010100000111 | 9607 | 2587 | ▇ | 10010110000111 |
| 9480 | 2508 | ┈ | 10010100001000 | 9608 | 2588 | █ | 10010110001000 |
| 9481 | 2509 | ┉ | 10010100001001 | 9609 | 2589 | ▉ | 10010110001001 |
| 9482 | 250A | ┊ | 10010100001010 | 9610 | 258A | ▊ | 10010110001010 |
| 9483 | 250B | ┋ | 10010100001011 | 9611 | 258B | ▋ | 10010110001011 |
| 9484 | 250C | ┌ | 10010100001100 | 9612 | 258C | ▌ | 10010110001100 |
| 9485 | 250D | ┍ | 10010100001101 | 9613 | 258D | ▍ | 10010110001101 |
| 9486 | 250E | ┎ | 10010100001110 | 9614 | 258E | ▎ | 10010110001110 |
| 9487 | 250F | ┏ | 10010100001111 | 9615 | 258F | ▏ | 10010110001111 |
| 9488 | 2510 | ┐ | 10010100010000 | 9616 | 2590 | ▐ | 10010110010000 |
| 9489 | 2511 | ┑ | 10010100010001 | 9617 | 2591 | ░ | 10010110010001 |
| 9490 | 2512 | ┒ | 10010100010010 | 9618 | 2592 | ▒ | 10010110010010 |
| 9491 | 2513 | ┓ | 10010100010011 | 9619 | 2593 | ▓ | 10010110010011 |
| 9492 | 2514 | └ | 10010100010100 | 9620 | 2594 | ▔ | 10010110010100 |
| 9493 | 2515 | ┕ | 10010100010101 | 9621 | 2595 | ▕ | 10010110010101 |

| Dec | Hex | UTF | Binary | Dec | Hex | UTF | Binary |
|---|---|---|---|---|---|---|---|
| 9494 | 2516 | ┖ | 10010100010110 | 9622 | 2596 | ▖ | 10010110010110 |
| 9495 | 2517 | ┗ | 10010100010111 | 9623 | 2597 | ▗ | 10010110010111 |
| 9496 | 2518 | ┘ | 10010100011000 | 9624 | 2598 | ▘ | 10010110011000 |
| 9497 | 2519 | ┙ | 10010100011001 | 9625 | 2599 | ▙ | 10010110011001 |
| 9498 | 251A | ┚ | 10010100011010 | 9626 | 259A | ▚ | 10010110011010 |
| 9499 | 251B | ┛ | 10010100011011 | 9627 | 259B | ▛ | 10010110011011 |
| 9500 | 251C | ├ | 10010100011100 | 9628 | 259C | ▜ | 10010110011100 |
| 9501 | 251D | ┝ | 10010100011101 | 9629 | 259D | ▝ | 10010110011101 |
| 9502 | 251E | ┞ | 10010100011110 | 9630 | 259E | ▞ | 10010110011110 |
| 9503 | 251F | ┟ | 10010100011111 | 9631 | 259F | ▟ | 10010110011111 |
| 9504 | 2520 | ┠ | 10010100100000 | 9632 | 25A0 | ■ | 10010110100000 |
| 9505 | 2521 | ┡ | 10010100100001 | 9633 | 25A1 | □ | 10010110100001 |
| 9506 | 2522 | ┢ | 10010100100010 | 9634 | 25A2 | ▢ | 10010110100010 |
| 9507 | 2523 | ┣ | 10010100100011 | 9635 | 25A3 | ▣ | 10010110100011 |
| 9508 | 2524 | ┤ | 10010100100100 | 9636 | 25A4 | ▤ | 10010110100100 |
| 9509 | 2525 | ┥ | 10010100100101 | 9637 | 25A5 | ▥ | 10010110100101 |
| 9510 | 2526 | ┦ | 10010100100110 | 9638 | 25A6 | ▦ | 10010110100110 |
| 9511 | 2527 | ┧ | 10010100100111 | 9639 | 25A7 | ▧ | 10010110100111 |
| 9512 | 2528 | ┨ | 10010100101000 | 9640 | 25A8 | ▨ | 10010110101000 |
| 9513 | 2529 | ┩ | 10010100101001 | 9641 | 25A9 | ▩ | 10010110101001 |
| 9514 | 252A | ┪ | 10010100101010 | 9642 | 25AA | ▪ | 10010110101010 |
| 9515 | 252B | ┫ | 10010100101011 | 9643 | 25AB | ▫ | 10010110101011 |
| 9516 | 252C | ┬ | 10010100101100 | 9644 | 25AC | ▬ | 10010110101100 |
| 9517 | 252D | ┭ | 10010100101101 | 9645 | 25AD | ▭ | 10010110101101 |
| 9518 | 252E | ┮ | 10010100101110 | 9646 | 25AE | ▮ | 10010110101110 |
| 9519 | 252F | ┯ | 10010100101111 | 9647 | 25AF | ▯ | 10010110101111 |
| 9520 | 2530 | ┰ | 10010100110000 | 9648 | 25B0 | ▰ | 10010110110000 |

| Dec | Hex | UTF | Binary | Dec | Hex | UTF | Binary |
|---|---|---|---|---|---|---|---|
| 9521 | 2531 | ┤ | 10010100110001 | 9649 | 25B1 | ▱ | 10010110110001 |
| 9522 | 2532 | ┤ | 10010100110010 | 9650 | 25B2 | ▲ | 10010110110010 |
| 9523 | 2533 | ┤ | 10010100110011 | 9651 | 25B3 | △ | 10010110110011 |
| 9524 | 2534 | ┴ | 10010100110100 | 9652 | 25B4 | ▴ | 10010110110100 |
| 9525 | 2535 | ┴ | 10010100110101 | 9653 | 25B5 | ▵ | 10010110110101 |
| 9526 | 2536 | ┴ | 10010100110110 | 9654 | 25B6 | ▶ | 10010110110110 |
| 9527 | 2537 | ┴ | 10010100110111 | 9655 | 25B7 | ▷ | 10010110110111 |
| 9528 | 2538 | ┴ | 10010100111000 | 9656 | 25B8 | ▸ | 10010110111000 |
| 9529 | 2539 | ┴ | 10010100111001 | 9657 | 25B9 | ▹ | 10010110111001 |
| 9530 | 253A | ┴ | 10010100111010 | 9658 | 25BA | ► | 10010110111010 |
| 9531 | 253B | ┴ | 10010100111011 | 9659 | 25BB | ▻ | 10010110111011 |
| 9532 | 253C | ┼ | 10010100111100 | 9660 | 25BC | ▼ | 10010110111100 |
| 9533 | 253D | ┼ | 10010100111101 | 9661 | 25BD | ▽ | 10010110111101 |
| 9534 | 253E | ┼ | 10010100111110 | 9662 | 25BE | ▾ | 10010110111110 |
| 9535 | 253F | ┼ | 10010100111111 | 9663 | 25BF | ▾ | 10010110111111 |
| 9536 | 2540 | ┼ | 10010101000000 | 9664 | 25C0 | ◀ | 10010111000000 |
| 9537 | 2541 | ┼ | 10010101000001 | 9665 | 25C1 | ◁ | 10010111000001 |
| 9538 | 2542 | ┼ | 10010101000010 | 9666 | 25C2 | ◂ | 10010111000010 |
| 9539 | 2543 | ┼ | 10010101000011 | 9667 | 25C3 | ◂ | 10010111000011 |
| 9540 | 2544 | ┼ | 10010101000100 | 9668 | 25C4 | ◀ | 10010111000100 |
| 9541 | 2545 | ┼ | 10010101000101 | 9669 | 25C5 | ◅ | 10010111000101 |
| 9542 | 2546 | ┼ | 10010101000110 | 9670 | 25C6 | ◆ | 10010111000110 |
| 9543 | 2547 | ┼ | 10010101000111 | 9671 | 25C7 | ◇ | 10010111000111 |
| 9544 | 2548 | ┼ | 10010101001000 | 9672 | 25C8 | ◈ | 10010111001000 |
| 9545 | 2549 | ┼ | 10010101001001 | 9673 | 25C9 | ◉ | 10010111001001 |
| 9546 | 254A | ┼ | 10010101001010 | 9674 | 25CA | ◊ | 10010111001010 |
| 9547 | 254B | ┼ | 10010101001011 | 9675 | 25CB | ○ | 10010111001011 |

| Dec | Hex | UTF | Binary | Dec | Hex | UTF | Binary |
|---|---|---|---|---|---|---|---|
| 9548 | 254C | ┌ | 10010101001100 | 9676 | 25CC | ◌ | 10010111001100 |
| 9549 | 254D | ┍ | 10010101001101 | 9677 | 25CD | ◍ | 10010111001101 |
| 9550 | 254E | ┎ | 10010101001110 | 9678 | 25CE | ◎ | 10010111001110 |
| 9551 | 254F | ┏ | 10010101001111 | 9679 | 25CF | ● | 10010111001111 |
| 9552 | 2550 | ═ | 10010101010000 | 9680 | 25D0 | ◐ | 10010111010000 |
| 9553 | 2551 | ‖ | 10010101010001 | 9681 | 25D1 | ◑ | 10010111010001 |
| 9554 | 2552 | ╒ | 10010101010010 | 9682 | 25D2 | ◒ | 10010111010010 |
| 9555 | 2553 | ╓ | 10010101010011 | 9683 | 25D3 | ◓ | 10010111010011 |
| 9556 | 2554 | ╔ | 10010101010100 | 9684 | 25D4 | ◔ | 10010111010100 |
| 9557 | 2555 | ╕ | 10010101010101 | 9685 | 25D5 | ◕ | 10010111010101 |
| 9558 | 2556 | ╖ | 10010101010110 | 9686 | 25D6 | ◖ | 10010111010110 |
| 9559 | 2557 | ╗ | 10010101010111 | 9687 | 25D7 | ◗ | 10010111010111 |
| 9560 | 2558 | ╘ | 10010101011000 | 9688 | 25D8 | ◘ | 10010111011000 |
| 9561 | 2559 | ╙ | 10010101011001 | 9689 | 25D9 | ◙ | 10010111011001 |
| 9562 | 255A | ╚ | 10010101011010 | 9690 | 25DA | ◚ | 10010111011010 |
| 9563 | 255B | ╛ | 10010101011011 | 9691 | 25DB | ◛ | 10010111011011 |
| 9564 | 255C | ╜ | 10010101011100 | 9692 | 25DC | ◜ | 10010111011100 |
| 9565 | 255D | ╝ | 10010101011101 | 9693 | 25DD | ◝ | 10010111011101 |
| 9566 | 255E | ╞ | 10010101011110 | 9694 | 25DE | ◞ | 10010111011110 |
| 9567 | 255F | ╟ | 10010101011111 | 9695 | 25DF | ◟ | 10010111011111 |
| 9568 | 2560 | ╠ | 10010101100000 | 9696 | 25E0 | ◠ | 10010111100000 |
| 9569 | 2561 | ╡ | 10010101100001 | 9697 | 25E1 | ◡ | 10010111100001 |
| 9570 | 2562 | ╢ | 10010101100010 | 9698 | 25E2 | ◢ | 10010111100010 |
| 9571 | 2563 | ╣ | 10010101100011 | 9699 | 25E3 | ◣ | 10010111100011 |
| 9572 | 2564 | ╤ | 10010101100100 | 9700 | 25E4 | ◤ | 10010111100100 |
| 9573 | 2565 | ╥ | 10010101100101 | 9701 | 25E5 | ◥ | 10010111100101 |
| 9574 | 2566 | ╦ | 10010101100110 | 9702 | 25E6 | ◦ | 10010111100110 |

| Dec | Hex | UTF | Binary | Dec | Hex | UTF | Binary |
|---|---|---|---|---|---|---|---|
| 9575 | 2567 | ┧ | 10010101100111 | 9703 | 25E7 | ◧ | 10010111100111 |
| 9576 | 2568 | ┨ | 10010101101000 | 9704 | 25E8 | ◨ | 10010111101000 |
| 9577 | 2569 | ┩ | 10010101101001 | 9705 | 25E9 | ◩ | 10010111101001 |
| 9578 | 256A | ┪ | 10010101101010 | 9706 | 25EA | ◪ | 10010111101010 |
| 9579 | 256B | ┫ | 10010101101011 | 9707 | 25EB | ◫ | 10010111101011 |
| 9580 | 256C | ╬ | 10010101101100 | 9708 | 25EC | ◬ | 10010111101100 |
| 9581 | 256D | ╭ | 10010101101101 | 9709 | 25ED | ◭ | 10010111101101 |
| 9582 | 256E | ╮ | 10010101101110 | 9710 | 25EE | ◮ | 10010111101110 |
| 9583 | 256F | ╯ | 10010101101111 | 9711 | 25EF | ◯ | 10010111101111 |
| 9584 | 2570 | ╰ | 10010101110000 | 9712 | 25F0 | ◰ | 10010111110000 |
| 9585 | 2571 | ╱ | 10010101110001 | 9713 | 25F1 | ◱ | 10010111110001 |
| 9586 | 2572 | ╲ | 10010101110010 | 9714 | 25F2 | ◲ | 10010111110010 |
| 9587 | 2573 | ╳ | 10010101110011 | 9715 | 25F3 | ◳ | 10010111110011 |
| 9588 | 2574 | ╴ | 10010101110100 | 9716 | 25F4 | ◴ | 10010111110100 |
| 9589 | 2575 | ╵ | 10010101110101 | 9717 | 25F5 | ◵ | 10010111110101 |
| 9590 | 2576 | ╶ | 10010101110110 | 9718 | 25F6 | ◶ | 10010111110110 |
| 9591 | 2577 | ╷ | 10010101110111 | 9719 | 25F7 | ◷ | 10010111110111 |
| 9592 | 2578 | ╸ | 10010101111000 | 9720 | 25F8 | ◸ | 10010111111000 |
| 9593 | 2579 | ╹ | 10010101111001 | 9721 | 25F9 | ◹ | 10010111111001 |
| 9594 | 257A | ╺ | 10010101111010 | 9722 | 25FA | ◺ | 10010111111010 |
| 9595 | 257B | ╻ | 10010101111011 | 9723 | 25FB | ▢ | 10010111111011 |
| 9596 | 257C | ╼ | 10010101111100 | 9724 | 25FC | ■ | 10010111111100 |
| 9597 | 257D | ╽ | 10010101111101 | 9725 | 25FD | □ | 10010111111101 |
| 9598 | 257E | ╾ | 10010101111110 | 9726 | 25FE | ▪ | 10010111111110 |
| 9599 | 257F | ╿ | 10010101111111 | 9727 | 25FF | ◿ | 10010111111111 |

# Приложение 3

# Справочник команд баз данных

В этом приложении дается краткое описание распространенных команд баз данных, используемых в SQL, CQL и их разновидностях. Соблюдение официального стандартного синтаксиса SQL зависит от базы данных, поэтому ознакомьтесь с соответствующей документацией производителя.

**Примечание.** Необязательный синтаксис обозначен ниже в скобках [].

## *SELECT*

Оператор SELECT используется для возврата данных из таблицы базы данных. В нем может быть указан список столбцов, которые необходимо вернуть из таблицы (задается оператором FROM). Возвращаемые строки определяются условием WHERE.

Синтаксис:

```
SELECT * | [column1_name [AS alias],column2_name,columnN_name]
FROM [database_name|keyspace_name.]table_name
[[INNER|LEFT|RIGHT] JOIN join_table_name ON column_name[operator]value]
[WHERE column_name[operator]value [[AND|OR] column_name[operator]value]]
[LIMIT N];
```

Условие WHERE может содержать несколько условий фильтрации, основанных на реляционной алгебре. Условия представляют собой ряд операторов равенства, включая равенство (=), неравенство (!= или <>), а также больше или меньше (<, >). Условия разделяются логическими операторами AND и OR.

**Примечание.** Только в некоторых NoSQL синтаксис SELECT позволяет использовать оператор OR, который часто инициирует сканирование множества разделов. Однако все реляционные базы данных позволяют использовать OR.

Важно отметить, что производительность запросов SELECT в значительной степени зависит от содержимого условия WHERE. Несвязанные запросы (SELECT без условия WHERE) приводят к полному сканированию таблицы для возврата всех строк.

## INSERT

Оператор INSERT позволяет записать новые данные в таблицу. В этом операторе должны быть перечислены значения столбцов, в которые необходимо записать данные, после чего следует оператор VALUES с соответствующими значениями столбцов, заключенными в круглые скобки.

Синтаксис:

```
INSERT INTO [database_name|keyspace_name.]table_name (column1[,column2,columnN])
VALUES (value1[,value2,valueN]);
```

Реляционные базы данных, скорее всего, выдадут ошибку, если пользователь попытается вставить строку, которая уже существует. Базы данных NoSQL (например, Cassandra), скорее всего, допустят эту операцию, перезаписав все неключевые значения строки.

## UPDATE

Оператор UPDATE предназначен для корректировки существующих значений столбцов. Для него требуется оператор SET, указывающий, какие пары столбец/значение должны быть установлены. Также в операторе UPDATE имеется условие WHERE, указывающее базе данных, какие строки должны быть обновлены.

Синтаксис:

```
UPDATE [database_name|keyspace_name.]table_name
SET column1=value1[,column2=value2,columnN=valueN]
[WHERE column_name[operator]value [[AND|OR] column_name_N[operator]value_N]];
```

Реляционные базы данных, скорее всего, выдадут ошибку, если пользователь попытается обновить несуществующую строку. Базы данных NoSQL (например, Cassandra), скорее всего, разрешат эту операцию, фактически добавляя новую строку в таблицу.

Следует внимательно отнестись к условию WHERE. В CQL оно должно содержать ключ раздела и может содержать дополнительные столбцы ключа кластеризации. Однако в SQL оно необязательно. Это означает, что UPDATE без условия WHERE применит записи в операторе SET *к каждой строке в таблице*.

## DELETE

Оператор DELETE предназначен для удаления существующих строк, которые соответствуют критериям в условии WHERE.

Синтаксис:

```
DELETE [column_name]
FROM [database_name|keyspace_name.]table_name
[WHERE column_name[operator]value [[AND|OR] column_name_N[operator]value_N]];
```

**Примечание.** Выполнение DELETE без условия WHERE приведет к удалению данных для каждого указанного столбца в таблице. Если столбцы не указаны, то будут удалены все строки таблицы, поэтому используйте этот оператор с осторожностью.

## CREATE TABLE

С помощью CREATE TABLE создается новая таблица, в которой будут храниться строки данных.

Синтаксис:

```
CREATE TABLE [IF NOT EXISTS] [database_name|keyspace_name.]table_name (
column1_name data_type [PRIMARY KEY],
[column2_name data_type,
columnN_name data_type,]
[PRIMARY KEY(key1,key2)]
);
```

Некоторые базы данных допускают отсутствие заданного первичного ключа на момент создания, другие — нет.

## CREATE INDEX

Эта команда создает индекс для указанной таблицы и комбинации столбцов.

Синтаксис:

```
CREATE INDEX [index_name] ON [table]([column]);
```

Результаты и требования, предъявляемые к дополнительным *вторичным* индексам, зависят от типа базы данных. Для большинства таблиц реляционных баз данных столбцы, поддерживающие фильтрацию в условии WHERE, должны содержать индекс. Обычно добавление индекса повышает производительность запросов.

Однако для баз данных NoSQL вторичные индексы, как правило, не способствуют повышению производительности запросов. Базы данных NoSQL (например, Apache Cassandra) спроектированы таким образом, чтобы распределять данные особым образом. Вторичные индексы, хотя и удобны, но нарушают первоначальную схему передачи данных. Поэтому вторичные индексы в базах данных NoSQL следует создавать редко и использовать нечасто.

Одним из общих эффектов индексов для всех типов баз данных является то, что они увеличивают ресурсы, необходимые для выполнения операции записи. Это происходит потому, что при записи необходимо обновить не только данные в таблице, но и все связанные с ней индексы.

# Приложение 4

# Общие коды ответов HTTP

В этом приложении приведен список распространенных кодов ответа HTTP. Существуют и другие коды HTTP, однако следующая таблица полезна для расшифровки распространенных кодов ответов при устранении неполадок.

**Таблица П4.1.** Перечень распространенных кодов ответов HTTP

| Код ответа HTTP | Описание |
|---|---|
| 200 | **Ok** |
| 301 | **Redirect** (перенаправление) |
| 302 | **Temporary redirect** (временное перенаправление) |
| 304 | **Not modified** (не изменено) |
| 400 | **Bad request** (плохой запрос) |
| 401 | **Unauthorized** (неавторизованный) |
| 403 | **Forbidden** (запрещено) |
| 404 | **Resource not found** (ресурс не найден) |
| 405 | **Method not allowed** (метод не разрешен) |
| 500 | **Server error** (ошибка сервера) |
| 502 | **Bad gateway** (проблема со шлюзом) |
| 503 | **Service unavailable** (сервис недоступен) |

# Приложение 5

# Основные цветовые коды

В этом приложении представлен краткий список часто используемых цветов с указанием их красного, зеленого, синего (red, green, blue — RGB) и шестнадцатеричного кодов. Эти коды являются стандартными для большинства веб- и графических интерфейсов.

**Таблица П5.1.** Перечень распространенных цветов и их кодов

| Цвет | Коды | | | |
|---|---|---|---|---|
| | Red | Green | Blue | Hex |
| Черный | 0 | 0 | 0 | 000000 |
| Темно-синий | 0 | 0 | 139 | 00008B |
| Синий | 0 | 0 | 255 | 0000FF |
| Темно-зеленый | 0 | 128 | 0 | 008000 |
| Зеленый | 0 | 255 | 0 | 00FF00 |
| Голубой | 0 | 255 | 255 | 00FFFF |
| Бордовый | 128 | 0 | 0 | 800000 |
| Фиолетовый | 128 | 0 | 128 | 800080 |
| Темно-фиолетовый | 128 | 0 | 192 | 8000C0 |
| Серый | 128 | 128 | 128 | 808080 |
| Коричневый | 165 | 42 | 42 | A52A2A |
| Серебристый | 192 | 192 | 192 | C0C0C0 |
| Красный | 255 | 0 | 0 | FF0000 |
| Пурпурный | 255 | 0 | 255 | FF00FF |
| Оранжевый | 255 | 165 | 0 | FFA500 |
| Желтый | 255 | 255 | 0 | FFFF00 |
| Белый | 255 | 255 | 255 | FFFFFF |

# Приложение 6

# Сбор мусора

Сбор мусора — необходимая функция JVM. В то время как языки более низкого уровня, такие как C, требуют от разработчика управлять использованием памяти, Java использует сборку мусора для удаления из памяти объектов, выходящих за пределы области видимости. В разных версиях Java по умолчанию используются разные методы для достижения этой цели. В табл. П6.1 приведен краткий список различных сборщиков мусора Java.

**Таблица П6.1.** Список сборщиков мусора Java и их типичных параметров

| Имя | Версии Java | Соответствующие настройки | Примечания |
|---|---|---|---|
| Concurrent Mark and Sweep (CMS) | 5-8 | `-XX:+UseConcMarkSweepGC` | Для правильной настройки CMS требуется множество различных параметров. Более подробную информацию можно найти в документации OpenJDK |
| Garbage First Collector (G1GC) | 7- | `-XX:+UseG1GC`<br>`-XX:MaxGCPauseMillis` | Сборщик мусора по умолчанию начиная с Java 9 |
| Z Garbage Collector (ZGC) | 15- | `-XX:+UseZGC` | |
| Shenandoah Garbage Collector | 17- | `-XX:+UseShenandoahGC` | |

**Примечание.** Версии, указанные для каждого сборщика мусора, представляют собой версии, в которых сборщик считается готовым к использованию и не находится в статусе устаревшего.

Рекомендуется задавать параметры, касающиеся размера кучи мусора Java, а не полагаться на вычисления по умолчанию. Некоторые сборщики (например, G1GC) рекомендуют задавать только подмножество этих параметров (например, `-Xmn` не следует использовать с G1GC). Параметры размера JVM приведены в табл. П6.2.

**Таблица П6.2.** Список параметров размера кучи в Java

| Параметр | Описание |
|---|---|
| `-Xmx` | Максимальный размер кучи памяти JVM |
| `-Xms` | Начальный размер кучи памяти JVM |
| `-Xmn` | Размер пространства нового поколения в куче памяти JVM |

Дополнительную информацию о параметрах кучи и сборки мусора для Java можно найти в документации Oracle по Java Hotspot VM Options: **https://www.oracle.com/java/technologies/javase/vmoptions-jsp.html**.

# Предметный указатель

## N



## O



## P



## R



## S



## T



## U



## V



## W

## А



## Б



## В



## Г



## Д



## Е



## З



## И



## К

## М



## Н



## О



## П



## Р

## С



## Т



## У



## Ф



## Х



## Ц



## Ч



## Ш



## Я