Функциональное программирование

Ричард Уорбэртон

Лямбда-выражения в Java 8

ДМК ПРЕСС
издательство

Ричард Уорбэртон

Лямбда-выражения в Java 8

Richard Warburton

Java 8 Lambdas

Functional Programming for the Masses

O'REILLY®

Ричард Уорбэртон

Лямбда-выражения в Java 8

Функциональное программирование – в массы

Москва, 2014

УДК 004.432.42Java 8
ББК 32.973.26-018.1
У62

Уорбэртон Р.
У62 Лямбда-выражения в Java 8. Функциональное программирование – в массы / пер. с анг. А. А. Слинкина. – М.: ДМК Пресс, 2014. – 192 с.: ил.

ISBN 978-5-97060-919-6

Если вы имеете опыт работы с Java SE, то из этой книги узнаете об изменениях в версии Java 8, обусловленных появлением в языке лямбда-выражений. Вашему вниманию будут представлены примеры кода, упражнения и увлекательные объяснения того, как можно использовать эти анонимные функции, чтобы сделать код проще и чище, и как библиотеки помогают в решении прикладных задач.

Лямбда-выражения – относительно простое изменение в языке Java; в первой части книги показано, как правильно ими пользоваться. В последующих главах демонстрируется, как лямбда-выражения позволяют повысить производительность программы за счет распараллеливания, писать более простой конкурентный код и точнее моделировать предметную область, в том числе создавать более качественные предметно-ориентированные языки.

Издание предназначено для программистов разной квалификации, как правило уже работающих с Java, но не имеющих опыта функционального программирования.

УДК 004.432.42Java 8
ББК 32.973.26-018.1

ISBN 978-1-449-37077-0 (анг.)

ISBN 978-5-97060-919-6 (рус.)

Содержание

Об авторе

Ричард Уорбэртон – технолог-эмпирик, увлекающийся решением сложных технических задач, требующих глубокого понимания предмета. Профессионально занимался проблемами статического анализа, верификацией части компилятора и разработкой усовершенствованной автоматизированной технологии обнаружения ошибок. Позже заинтересовался методами анализа данных для высокопроизводительных вычислений. Является руководителем лондонского сообщества пользователей Java и членом комитета JCP, организует процесс подачи запросов на улучшение для Java 8 в части лямбда-выражений и механизмов работы с датой и временем. Ричард также часто выступает на конференциях, в том числе JavaOne, DevoxxUK и JAX London. Получил степень доктора философии по информатике в Варвикском университете, где занимался теоретическими вопросами построения компиляторов.

Предисловие

В течение многих лет функциональное программирование считалось уделом небольшой кучки специалистов, неизменно провозглашавших его превосходство, но не способных убедить массы в мудрости своего подхода. И эту книгу я написал прежде всего для того, чтобы оспорить идею о том, будто функциональному стилю присущи какое-то особое превосходство и убежденность в том, что он доступен лишь немногим избранным.

Последние два года я убеждал разработчиков, входящих в лондонское сообщество пользователей Java, попробовать те или иные аспекты Java 8. Как оказалось, многим членам нашего сообщества очень нравятся предоставленные им новые идиомы и библиотеки. Возможно, их несколько смущают терминология и элитарность технологии, но преимущества, которые несет с собой толика несложного функционального программирования, никого не оставляют равнодушными. Все согласны, что гораздо проще читать код манипуляции объектами и коллекциями, написанный с использованием нового Streams API, – например, для выделения музыкальных альбомов, выпущенных в Великобритании, из списка `List` всех альбомов.

Из опыта проведения таких мероприятий я вынес важный урок – все зависит от примеров. Человек учится, решая простые примеры и осознавая стоящие за ними закономерности. Я также понял, что терминология легко может оттолкнуть учащегося, поэтому всегда стараюсь объяснять трудные идеи простыми словами.

Для многих механизмы функционального программирования, включенные в Java 8, представляются невероятно ограниченными: ни тебе монад[1], ни отложенных вычислений на уровне языка, ни дополнительной поддержки неизменяемости. С точки зрения программиста-прагматика, это прекрасно; нам нужна возможность выражать абстракции на уровне библиотек, чтобы можно быть писать простой и чистый код, решающий конкретную задачу. Даже лучше, если кто-то уже написал за нас эти библиотеки, чтобы мы могли сосредоточиться на своей повседневной работе.

---

[1] Больше это слово в тексте ни разу не встретится.

Зачем мне читать эту книгу?

В этой книге мы рассмотрим следующие вопросы.

- Как писать простой, чистый и понятный читателю код, особенно в части работы с коллекциями.
- Как с помощью параллелизма повысить производительность.
- Как более точно моделировать предметную область и создавать более качественные предметно-ориентированные языки.
- Как писать более простой и безошибочный параллельный код.
- Как тестировать и отлаживать лямбда-выражения.

Повышение продуктивности разработчика – не единственная причина добавления лямбда-выражений в Java; действуют еще и глубинные течения в нашей индустрии.

Кому стоит прочитать эту книгу?

Эта книга предназначена разработчикам, пишущим на Java, знакомым с основами Java SE и желающим идти в ногу со значительными изменениями, появившимися в Java 8.

Если вам интересно узнать о том, что такое лямбда-выражения и как они могут повысить ваш профессионализм, читайте дальше! Не предполагается никаких предварительных знаний о лямбда-выражениях или еще каких-то новшествах в базовых библиотеках; все необходимые сведения будут изложены по ходу дела.

Конечно, мне хотелось бы, чтобы каждый разработчик приобрел эту книгу, но, по совести говоря, она нужна не всем. Если вы вообще не знаете языка Java, то эта книга не для вас. С другой стороны, хотя тема лямбда-выражений в Java рассматривается очень подробно, я ничего не рассказываю о том, как они используются в других языках.

Не ожидайте введения в такие аспекты Java SE, как коллекции, анонимные внутренние классы или механизм обработки событий в Swing. Предполагается, что все это вы уже знаете.

Как читать эту книгу

Эта книга построена на примерах: вслед за знакомством с новой концепцией сразу идет код. Иногда в коде может встретиться что-то такое, с чем вы не совсем знакомы. Не пугайтесь – объяснение последует очень скоро, чаще всего в следующем же абзаце.

У такого подхода есть еще и то достоинство, что он позволяет по ходу дела экспериментировать с новыми идеями. Более того, в конце многих глав имеются дополнительные примеры для самостоятельной работы. Я настоятельно рекомендую выполнять эти упражнения – ката. Навык мастера ставит, и – как известно любому программисту-прагматику – очень легко впасть в заблуждение, думая, что понимаешь какой-то код, тогда как на самом деле упустил из виду важную деталь.

Поскольку смысл лямбда-выражений заключается в том, чтобы абстрагировать сложность, убрав ее в библиотеки, то я остановлюсь на нескольких приятных нововведениях в общих библиотеках. В главах 2–6 рассматриваются изменения в самом языке и усовершенствованные библиотеки, входящие в состав JDK 8.

Последние три главы касаются практических применений функционального программирования. В главе 7 я расскажу о нескольких приемах, упрощающих тестирование и отладку кода. В главе 8 объясняется, как применить к лямбда-выражениям общепринятые принципы правильного проектирования программного обеспечения. Затем, в главе 9, мы поговорим о параллелизме и о том, как с помощью лямбда-выражений писать понятный параллельный код, пригодный для сопровождения. Там, где это уместно, я буду знакомить вас со сторонними библиотеками.

Первые четыре главы, наверное, стоит рассматривать как вводный материал – вещи, которые должен знать всякий, кто хочет правильно использовать Java 8. Последующие главы сложнее, зато они научат вас полноценно и уверенно применять лямбда-выражения в собственных проектах. По всей книге рассыпаны упражнения, решения к ним имеются на сайте в GitHub. Если вы не будете пренебрегать упражнениями, то очень скоро овладеете лямбда-выражениями в совершенстве.

Графические выделения

В книге применяются следующие графические выделения:

Курсив
: Новые термины, URL-адреса, адреса электронной почты, имена и расширения имен файлов.

`Моноширинный`
: Листинги программ, а также элементы кода в основном тексте: имена переменных и функций, базы данных, типы данных, переменные окружения, предложения и ключевые слова языка.

Моноширинный полужирный
: Команды и иной текст, который пользователь должен вводить точно в указанном виде.

Моноширинный курсив
: Текст, вместо которого следует подставить значения, заданные пользователем или определяемые контекстом.

 Так обозначаются совет или рекомендация.

 Так обозначаются примечания общего характера.

 Так обозначаются предупреждения или предостережения.

О примерах кода

Дополнительные материалы (примеры кода, упражнения и т. д.) можно скачать с сайта *https://github.com/RichardWarburton/java-8-lambdas-exercises*.

Эта книга призвана помогать вам в работе. Поэтому вы можете использовать приведенный в ней код в собственных программах и в документации. Спрашивать у нас разрешения необязательно, если только вы не собираетесь воспроизводить значительную часть кода. Например, никто не возбраняет включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров из книг издательства O'Reilly на компакт-диске разрешение требуется. Цитировать книгу и примеры в ответах на вопросы можно без ограничений. Но для включения значительных объемов кода в документацию по собственному продукту нужно получить разрешение.

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: «Java 8 Lambdas by Richard Warburton (O'Reilly). Copyright 2014 Richard Warburton, 978-1-449-37077-0».

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу *permissions@oreilly.com*.

Как с нами связаться

Вопросы и замечания по поводу этой книги отправляйте в издательство:

800-998-9938 (в США и Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Для этой книги имеется веб-страница, на которой публикуются списки замеченных ошибок, примеры и прочая дополнительная информация. Адрес страницы: *http://oreil.ly/java_8_lambdas*.

Замечания и вопросы технического характера следует отправлять по адресу *bookquestions@oreilly.com*.

Дополнительную информацию о наших книгах, конференциях и новостях вы можете найти на нашем сайте по адресу *http://www.oreilly.com*.

Читайте нас на Facebook: *http://facebook.com/oreilly*.

Следите за нашей лентой в Twitter: *http://twitter.com/oreillymedia*.

Смотрите нас на YouTube: *http://www.youtube.com/oreillymedia*.

Благодарности

Хотя на обложке книги стоит мое имя, ее выходу в свет немало поспособствовали и многие другие.

Прежде всего хочу сказать спасибо редактору Меган и всему коллективу издательства O'Reilly, благодаря которым процесс оказался очень приятным, а сроки – достаточно гибкими. И если бы Мартин и Бен не представили меня Меган, то эта книга никогда и не появилась бы.

Рецензирование сильно помогло улучшить книгу, поэтому я сердечно благодарен всем, кто принимал участие в официальном и неофициальном рецензировании: Мартину Вербургу (Martijn Verburg), Джиму Гафу (Jim Gough), Джону Оливеру (John Oliver), Эдварду Вонгу (Edward Wong), Брайану Гетцу (Brian Goetz), Даниэлю Брайанту (Daniel Bryant), Фреду Розенбергу (Fred Rosenberger), Джайкиран Пай (Jaikiran Pai) и Мани Саркар (Mani Sarkar). А особенно Мартину, который таки убедил меня написать техническую книгу.

Нельзя также не упомянуть группу разработчиков проекта Project Lambda в Oracle. Модернизировать уже сложившийся язык – задача не из легких, и они отлично справились с этой работой в Java 8, заодно предоставив мне материал, о котором можно писать. Благодарности заслуживает также лондонское сообщество пользователей Java, которое так активно участвовало в тестировании ранних выпусков Java, демонстрируя, какие ошибки допускают разработчики и как их можно исправить.

Помощь и поддержку на протяжении всей работы над книгой мне оказывало множество людей. Особенно хочется выделить родителей, которые приходили на выручку по первому зову. И невыразимо приятно было получать ободрение и позитивные замечания от друзей, в том числе от старых членов компьютерного общества, а особенно от Садика Джаффера (Sadiq Jaffer) и Бойса Бригейда (Boys Brigade).

Глава 1

Введение

Прежде чем вплотную заняться вопросом о том, что такое лямбда-выражения и как ими пользоваться, надо бы понять, для чего вообще они существуют. В этой главе я расскажу об этом, а заодно опишу структуру книги и причины ее появления.

Зачем понадобилось снова изменять Java?

Версия Java 1.0 была выпущена в январе 1996 года, и с тех пор мир программирования претерпел кое-какие изменения. Бизнес требует все более сложных приложений, а программы по большей части исполняются на машинах с мощными многоядерными процессорами. Появление целого ряда виртуальных машин Java (JVM) с эффективными JIT-компиляторами означает, что теперь программисты могут сосредоточиться на создании чистого, удобного для сопровождения кода, а не выжимать из оборудования все до последнего такта процессора, трясясь над каждым байтом памяти.

Все знают о нашествии многоядерных процессоров, но мало кто задумывается об этом. Алгоритмы с применением блокировок чреваты ошибками и требуют много времени на разработку. В пакете `java.util.concurrent` и многочисленных внешних библиотеках предлагаются различные абстракции параллелизма, помогающие писать код, эффективно работающий на многоядерных процессорах. К сожалению, до сих пор мы продвинулись не слишком далеко. Но времена меняются.

В настоящее время есть пределы уровню абстрагирования, доступному авторам библиотек на Java. Хороший пример – отсутствие эффективных параллельных операций с большими коллекциями данных. Java 8 позволяет записывать сложные алгоритмы обработ-

ки коллекций, а путем простого изменения всего лишь одного вызова метода этот код будет эффективно исполняться на многоядерных процессорах. Но чтобы такие библиотеки распараллеливания массовых операций над данными были возможны, в Java пришлось внести дополнение на уровне языка: лямбда-выражения.

Разумеется, за все нужно платить. В данном случае придется научиться читать и писать код с лямбда-выражениями, но это неплохая сделка. Программисту проще изучить новый синтаксис и несколько новых идиом, чем писать вручную горы сложного потокобезопасного кода. Хорошие библиотеки и каркасы существенно сократили временные и финансовые затраты на разработку корпоративных приложений, а теперь следует устранить все барьеры на пути создания простых в использовании и эффективных библиотек.

Идея абстракции знакома всем, кто занимается объектно-ориентированным программированием. Различие же состоит в том, что в объектно-ориентированном программировании абстрагируются главным образом данные, а в функциональном – поведение. В реальном мире, как и в наших программах, все перемешано, поэтому мы можем и должны изучать обе тенденции.

У нового аспекта абстракции есть и другие достоинства – существенные для тех из нас, кому не приходится постоянно писать код, который должен выполняться максимально эффективно. Теперь вы можете писать код, который проще читать, уделяя главное внимание способам выражения своих намерений, а не механизмам их достижения. А код, который легко читать, легко также сопровождать, он более надежен и в меньшей степени подвержен ошибкам.

При создании обратных вызовов и обработчиков событий вы больше не связаны многословностью и неудобочитаемостью анонимных вложенных классов. При таком подходе программисту проще работать с системами обработки событий. Возможность передавать функции из одного места в другое позволяет без особого труда писать отложенный код, в котором значения инициализируются в тот момент, когда это необходимо.

Ко всему прочему, появление в языке методов коллекций по умолчанию (`default`) позволяет программисту лучше сопровождать собственные библиотеки.

Сегодня язык Java не тот, на котором писал ваш дедушка, – и это хорошо.

Что такое функциональное программирование?

Разные люди понимают под словами *функциональное программирование* разные вещи. В его основе лежит осмысление предметной области в терминах неизменяемых значений и функций, которые их преобразуют.

Сообщества, сформировавшиеся вокруг какого-то языка программирования, полагают, что набор средств, включенных в их любимый язык, – единственно правильный. На данном этапе еще слишком рано говорить о том, как определят функциональное программирование программисты, пишущие на Java. Но в определенном смысле это и не важно; существенно то, как писать *хороший* – а не функциональный – код.

В этой книге предметом моего внимания будет прагматичное функциональное программирование, в том числе приемы, которые легко сможет понять и использовать большинство программистов, чтобы создавать программы, более понятные и удобные для сопровождения.

Пример предметной области

Все примеры в этой книге относятся к общей предметной области: музыке. Точнее, мы будем иметь дело с данными, присутствующими в музыкальных альбомах. Ниже приведена краткая сводка терминов.

Исполнитель

Один человек или группа, исполняющая музыкальные произведения:

- *name*: имя или название исполнителя (например, «The Beatles»);
- *members*: другие исполнители, входящие в состав группы (например, «Джон Леннон»), это поле может быть пустым;
- *origin*: место, где возникла группа (например, «Ливерпуль»).

Произведение

Одно музыкальное произведение:

- *name:* название произведения (например, «Yellow Submarine»).

Альбом

Собрание нескольких музыкальных произведений в одном издании:

- *name:* название альбома (например, «Revolver»);
- *tracks:* список произведений;
- *musicians:* список исполнителей, принимавших участие в работе над альбомом.

На примере этой предметной области мы продемонстрируем применение функционального программирования в типичном бизнес-приложении на Java. Возможно, на ваш взгляд, этот пример не идеален, но он простой, а многие приведенные в этой книге фрагменты кода аналогичны встречающимся в реальных задачах.

Глава 2

Лямбда-выражения

Самое серьезное изменение на уровне языка, появившееся в Java 8, – лямбда-выражения – компактный способ передать поведение из одного места программы в другое. Поскольку это тот элемент, который лежит в основе всей книги, поговорим о том, что же он собой представляет.

Наше первое лямбда-выражение

Swing – это платформенно-независимая Java-библиотека для создания графического интерфейса пользователя (ГИП). В ней повсеместно встречается идиома, в соответствии с которой для выяснения того, что сделал пользователь, необходимо зарегистрировать *прослушиватель событий*. Затем прослушиватель сможет выполнить что-то полезное в ответ на действие пользователя (см. пример 2.1).

Пример 2.1 ❖ Использование анонимного внутреннего класса для связывания поведения с нажатием кнопки

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("button clicked");
    }
});
```

Здесь мы создаем новый объект, предоставляющий реализацию интерфейса `ActionListener`. В этом интерфейсе определен единственный метод `actionPerformed`, который вызывается объектом `button`, когда пользователь нажимает кнопку на экране. Анонимный внутренний класс как раз и предоставляет реализацию этого метода. В примере 2.1 он просто печатает сообщение о том, что была нажата кнопка.

На самом деле это пример использования *кода как данных* – мы передаем кнопке объект, который представляет действие.

Анонимные внутренние классы были придуманы, чтобы упростить передачу кода как данных в Java. К сожалению, упрощение оказалось

недостаточным. Мы по-прежнему видим четыре строки стереотипного кода, обрамляющих единственно важную строку, содержащую логику.

Но наличие стереотипного кода – не единственная проблема: этот код довольно трудно читать, потому что он затемняет намерение программиста. Мы не хотим передавать никакой объект, в действительности требуется передать некое поведение. В Java 8 этот код можно переписать в виде лямбда-выражения, как показано в примере 2.2.

Пример 2.2 ❖ Использование лямбда-выражения для связывания поведения с нажатием кнопки

```
button.addActionListener(event -> System.out.println("button clicked"));
```

Вместо объекта, реализующего интерфейс, мы передаем блок кода – функцию без имени. Здесь `event` – имя параметра, такое же, как в примере с анонимным внутренним классом, – а `->` отделяет параметр от тела лямбда-выражения, содержащего код, исполняемый при нажатии кнопки.

Еще одно отличие этого примера от анонимного внутреннего класса – способ объявления переменной `event`. Раньше нужно было явно указать тип – `ActionEvent event`. Теперь тип не указывается вовсе, и тем не менее код компилируется. Подспудно компилятор `javac` выводит тип переменной `event` из контекста – в данном случае из сигнатуры метода `addActionListener`. Это означает, что нет нужды явно выписывать тип в случае, когда он очевиден. Ниже мы рассмотрим механизм выведения типа более подробно, но сначала познакомимся с различными способами записи лямбда-выражений.

Хотя для записи параметров лямбда-метода требуется меньше стереотипного кода, чем раньше, они все равно статически типизированы. Для большей удобочитаемости объявления типов можно включать явно, а иногда компилятор просто не может вывести их автоматически!

Как опознать лямбда-выражение

В примере 2.3 показано несколько вариантов основного формата записи лямбда-выражений.

Пример 2.3 ❖ Различные способы записи лямбда-выражений

```
Runnable noArguments = () -> System.out.println("Hello World");  ❶

ActionListener oneArgument = event -> System.out.println("button clicked");  ❷

Runnable multiStatement = () -> {  ❸
```

```
    System.out.print("Hello");
    System.out.println(" World");
};

BinaryOperator<Long> add = (x, y) -> x + y;  ❹


BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y;  ❺
```

❶ показывает, как можно записать лямбда-выражение вообще без аргументов. Отсутствие аргументов обозначается парой пустых скобок `()`. Это лямбда-выражение реализует интерфейс `Runnable`, единственный метод которого `run` не принимает аргументов и «возвращает» значение типа `void`.

В случае ❷ имеется лямбда-выражение с одним аргументом, и в этой ситуации окружающие его скобки можно опустить. Это именно тот формат, который использовался в примере 2.2.

Лямбда-выражение может включать не только единственное выражение, но и целый блок кода, заключенный в фигурные скобки (`{}`), как в случае ❸. К таким блокам применяются те же правила, что и для обычных методов. В частности, они могут возвращать значения и возбуждать исключения. В фигурные скобки можно заключать и однострочное лямбда-выражение, например чтобы яснее показать, где оно начинается и заканчивается.

С помощью лямбда-выражений можно также представлять методы, принимающие несколько аргументов, как в случае ❹. Сейчас стоит поговорить о том, как *читать* такое лямбда-выражение. В этой строке мы не складываем два числа, а создаем функцию, складывающую два числа. Переменная с именем `add`, имеющая тип `BinaryOperator<Long>`, – это не результат сложения чисел, а код, который их складывает.

До сих пор типы параметров лямбда-выражений выводил за нас компилятор. Это замечательно, но иногда удобно иметь возможность задать тип явно. Поступая так, мы должны заключить аргументы лямбда-выражения в круглые скобки. Скобки необходимы и тогда, когда аргументов несколько. Это показано в случае ❺.

Целевым типом лямбда-выражения называется тип контекста, в котором это выражение встречается, – например, тип локальной переменной, которой оно присваивается, или тип параметра метода, вместо которого оно передается.

Во всех этих примерах неявно подразумевается, что тип лямбда-выражения зависит от контекста. Он выводится компилятором. Выведение целевого типа не является новшеством. Как видно из примера 2.4, типы инициализаторов массива всегда выводились в Java из

контекста. Другой знакомый пример – null. Какой тип имеет null, мы понимаем сразу, как видим его присваивание чему-либо.

Пример 2.4 ❖ Тип правой части не указан, он выводится из контекста

```
final String[] array = { "hello", "world" };
```

Использование значений

В прошлом, используя анонимные внутренние классы, вы, наверное, встречались с ситуацией, когда требуется использовать переменную, определенную в объемлющем методе. Для этого переменную нужно было объявлять с ключевым словом final, как показано в примере 2.5. Переменной, объявленной как final, нельзя присвоить другое значение. Это означает, что, используя final-переменную, мы точно знаем, что она сохранит именно то значение, которое ей было когда-то присвоено.

Пример 2.5 ❖ Локальная final-переменная, захваченная анонимным внутренним классом

```
final String name = getUserName();
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("hi " + name);
    }
});
```

В Java 8 это ограничение немного ослаблено. Теперь разрешено ссылаться на переменные, в объявлении которых нет слова final; однако они должны быть *эффективно* финальными. Хотя включать final в объявление переменной не требуется, использовать переменные, на которые ссылается лямбда-выражение, как нефинальные, запрещено. Попытка сделать это приведет к ошибке компиляции.

Таким образом, эффективно финальной переменной значение можно присвоить только один раз. По-другому осознать это ограничение можно, поняв, что лямбда-выражение захватывает не переменные, а *значения*. В примере 2.6 name – эффективно финальная переменная.

Пример 2.6 ❖ Эффективно final-переменная, захваченная лямбда-выражением

```
String name = getUserName();
button.addActionListener(event -> System.out.println("hi " + name));
```

Лично мне подобный код читать легче, когда слово final опущено, потому что оно воспринимается как лишний шум. Разумеется, быва-

ют ситуации, когда проще понять код, если final явно присутствует. В конечном итоге использовать механизм «эффективной финальности» или нет – дело вкуса.

Если присвоить переменной значение несколько раз, а затем попытаться использовать ее в лямбда-выражении, компилятор выдаст ошибку. Так, при попытке откомпилировать программу из примера 2.7 мы получим сообщение local variables referenced from a lambda expression must be final or effectively final[1].

Пример 2.7 ❖ Не компилируется из-за использования переменной, не являющейся эффективно финальной

```
String name = getUserName();
name = formatUserName(name);
button.addActionListener(event -> System.out.println("hi " + name));
```

Это поведение помогает объяснить одну из причин, по которым некоторые называют лямбда-выражения «замыканиями». Переменная, которой не присвоено значение, *замыкается* в объемлющем состоянии и затем связывается со значением. В интернет-сообществе оживленно дебатировался вопрос, а правда ли в Java есть замыкания, коль скоро ссылаться можно только на эффективно финальные переменные. Перефразируя Шекспира, можно сказать: «Замыкание остается замыканием, хоть замыканием назови его, хоть нет»[2]. Чтобы не вступать в бессмысленные споры, я буду употреблять в книге термин «лямбда-выражение». Так или иначе, лямбда-выражения, как я уже говорил, статически типизированы, поэтому давайте разберемся с типами самих лямбда-выражений; они называются *функциональными интерфейсами*.

Функциональные интерфейсы

Функциональным интерфейсом называет интерфейс с единственным абстрактным методом, который и является типом лямбда-выражения.

В Java у всех параметров метода есть типы; если мы передаем методу значение 3 в качестве аргумента, то параметр должен иметь тип int. Тогда каков же тип лямбда-выражения?

---

[1] Локальные переменные, на которые имеются ссылки в лямбда-выражении, должны быть финальными или эффективно финальными. – *Прим. перев.*

[2] «Что значит имя? Роза пахнет розой, хоть розой назови ее, хоть нет» // Ромео и Джульетта / перевод Б. Л. Пастернака.

Существует старая-престарая идиома использования интерфейса с единственным методом для представления и повторного использования метода. Все мы знакомы с ней по программированию в Swing, и именно это и имеет место в примере 2.2. Никакого нового волшебства здесь нет. Точно такая же идиома используется для лямбда-выражений, а интерфейс такого рода мы называем *функциональным*. В примере 2.8 показан функциональный интерфейс для рассмотренного выше примера.

Пример 2.8 ❖ Интерфейс ActionListener: на входе ActionEvent, на выходе ничего

```
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent event);
}
```

В интерфейсе `ActionListener` имеется только один абстрактный метод, `actionPerformed`, мы используем его, чтобы *представить* действие, которое принимает один аргумент и не возвращает ничего. Напомним, что коль скоро метод `actionPerformed` объявлен в интерфейсе, предпосылать ему ключевое слово `abstract` необязательно – он и так абстрактный. У него также имеется родительский интерфейс `EventListener`, вообще без методов.

Итак, мы имеем функциональный интерфейс. Совершенно не важно, как называется его единственный метод, – он сопоставится с лямбда-выражением при условии совместимости сигнатур. Функциональные интерфейсы также позволяют нам придать полезное имя типу параметра – такое, которое поможет понять, для чего он предназначен.

В данном случае функциональный интерфейс принимает один параметр типа `ActionEvent` и ничего не возвращает (`void`), однако они могут представать и в других обличьях. Например, функциональный интерфейс может принимать два параметра и возвращать значение. В них допустимы также универсальные типы; все зависит от предполагаемого использования.

Начиная с этого момента, я буду использовать диаграммы для представления различных видов функциональных интерфейсов. Входящими стрелками обозначаются аргументы, а исходящей (если она присутствует) – тип возвращаемого значения. Так, на рис. 2.1 изображен функциональный интерфейс `ActionListener`.

Вам еще встретится много функциональных интерфейсов, но в JDK существует несколько основных интерфейсов, повторяющихся снова и снова. Наиболее важные перечислены в табл. 2.1.

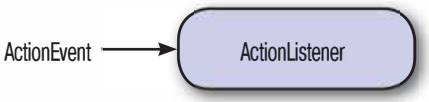

Рис. 2.1 ❖ Интерфейс ActionListener: на входе ActionEvent, на выходе ничего (void)

Таблица 2.1. Важные функциональные интерфейсы в Java

Имя интерфейса	Аргументы	Возвращает	Пример
`Predicate<T>`	`T`	`Boolean`	Этот альбом уже вышел?
`Consumer<T>`	`T`	`Void`	Распечатка значения
`Function<T,R>`	`T`	`R`	Получить имя из объекта `Artist`
`Supplier<T>`	Нет	`T`	Фабричный метод
`UnaryOperator<T>`	`T`	`T`	Логическое НЕТ (!)
`BinaryOperator<T>`	`(T, T)`	`T`	Умножение двух чисел (*)

Я уже говорил о том, какие типы могут принимать функциональные интерфейсы, и отметил, что `javac` умеет автоматически выводить типы параметров и что вы можете указать их вручную. Но как узнать, когда нужно указывать типы явно, а когда нет? Чтобы ответить на этот вопрос, придется познакомиться с деталями выведения типов.

Выведение типов

Есть случаи, когда обязательно указывать типы вручную, а вообще я рекомендую поступать так, как вам и вашим коллегам кажется правильным с точки зрения удобочитаемости. Иногда отсутствие явно указанных типов позволяет устранить лишний шум и прояснить, что происходит на самом деле. А иногда их лучше оставить – с точно такой же целью. Лично мне типы поначалу казались полезными, но со временем я стал указывать их только тогда, когда это действительно необходимо. Чтобы понять, когда это так, нужно освоить несколько простых правил, с которыми мы познакомимся в этой главе.

Выведение типов в лямбда-выражениях – это на самом деле обобщение аналогичного механизма, появившегося в Java 7. Вы, наверное, знаете о ромбовидном операторе (операторе `diamond`) в Java 7, с помощью которого мы просим `javac` *вывести* типы универсальных аргументов. Ниже приведен пример.

Пример 2.9 ❖ Применение ромбовидного оператора для выведения типа переменной

```
Map<String, Integer> oldWordCounts = new HashMap<String, Integer>(); 1
Map<String, Integer> diamondWordCounts = new HashMap<>();  2
```

В объявлении переменной `oldWordCounts` 1 мы явно указали универсальные типы, а в объявлении переменной `diamondWordCounts` 2 воспользовались ромбовидным оператором. Универсальные типы отсутствуют – компилятор сам определил, что вы хотите сделать. Магия!

Конечно, никакой магии тут нет. Универсальные типы-аргументы `HashMap` можно вывести из типа `diamondWordCounts`. Но указывать универсальные типы в объявлении переменной, которой присваивается значение, все равно надо.

Если передавать методу выражение конструирования, то универсальные типы также можно вывести – из сигнатуры метода. В примере 2.10 мы передаем экземпляр `HashMap` в виде аргумента методу, в сигнатуре которого типы уже указаны.

Пример 2.10 ❖ Применение ромбовидного оператора для выведения типа в аргументе метода

```
useHashmap(new HashMap<>());
...
private void useHashmap(Map<String, String> values);
```

Если в Java 7 разрешалось опускать универсальные типы-аргументы в конструкторе, то в Java 8 можно опускать типы параметров лямбда-выражений. И опять-таки никакой магии: `javac` ищет информацию в непосредственной близости от лямбда-выражения и использует ее для принятия решения о типе. Типы по-прежнему проверяются, так что привычная безопасность никуда не девается, но явно указывать типы нет необходимости. Именно это и называется *выведением типов*.

Стоит также отметить, что в Java 8 механизм выведения типов улучшен. Приведенный выше пример, где выражение `new HashMap<>()` передавалось в метод `useHashmap`, в Java 7 не откомпилировался бы, хотя у компилятора имеется вся информация для принятия решения.

Приведем еще несколько примеров, чтобы уточнить детали.

В обоих случаях мы присваиваем значение переменной типа функционального интерфейса, чтобы было проще следить за происходящим. В примере 2.11 мы имеем лямбда-выражение, определяющее, верно ли, что значение типа `Integer` больше 5. На самом деле это `Predicate` – функциональный интерфейс, который возвращает `true` или `false`.

Пример 2.11 ❖ Выведение типа

```
Predicate<Integer> atLeast5 = x -> x > 5;
```

Лямбда-выражение, возвращающее значение, также имеет тип `Predicate` – в отличие от предыдущих примеров, где фигурировал тип `ActionListener`. В данном случае тело лямбда-выражения имеет вид `x > 5`. При вычислении лямбда-выражения будет возвращено значение, получающееся в результате вычисления его тела.

Из примера 2.11 видно, что у типа `Predicate` имеется универсальный тип-аргумент; в данном случае мы взяли тип `Integer`. Поэтому компилятор выводит, что единственный аргумент лямбда-выражения, реализующего `Predicate`, имеет тип `Integer`. `javac` может также проверить, что выражение возвращает значение типа `boolean`, потому что именно таким должен быть тип значения, возвращаемого методом интерфейса `Predicate` (см. рис. 2.2).

Пример 2.12 ❖ Интерфейс Predicate в коде, получающем объект и возвращающем boolean

```
public interface Predicate<T> {
    boolean test(T t);
}
```

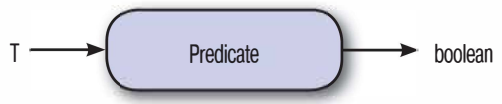

Рис. 2.2 ❖ Диаграмма интерфейса Predicate, который возвращает boolean, получая объект типа T

Рассмотрим еще один, чуть более сложный пример функционального интерфейса: интерфейс `BinaryOperator`, показанный в примере 2.13. Он принимает два аргумента одного типа и возвращает значение того же типа. В примере мы взяли тип `Long`.

Пример 2.13 ❖ Более сложный пример выведения типа

```
BinaryOperator<Long> addLongs = (x, y) -> x + y;
```

Механизм выведения типа достаточно «умный», но если информации недостаточно, он не сможет принять правильное решение. В таких случаях он не занимается гаданием на кофейной гуще, а просто останавливается и просит у вас помощи – в форме сообщения компилятора об ошибке. Например, удалив часть информации о типе в предыдущем примере, мы получим код, показанный ниже.

Пример 2.14 ❖ Этот код не компилируется из-за отсутствия универсального типа

```
BinaryOperator add = (x, y) -> x + y;
```

При попытке компиляции выдается такое сообщение:

```
Operator '&#x002B;' cannot be applied to java.lang.Object, java.lang.Object.
```

Выглядит совершенно непонятно. Но напомним, что `BinaryOperator` – функциональный интерфейс с универсальным типом-аргументом. Именно такой тип имеют оба аргумента, `x` и `y`, и возвращаемое значение. Однако в коде выше мы не указали тип-аргумент в объявлении переменной `add`. Получилось определение *простого* типа. Поэтому компилятор думает, что аргументы и возвращаемое значение имеют тип `java.lang.Object`.

Мы еще вернемся к механизму выведения типов и его взаимосвязи с перегрузкой методов в разделе «Разрешение перегрузки» ниже, а до тех пор уже сказанного будет вполне достаточно.

Основные моменты

- Лямбда-выражение – это безымянный метод, который служит для передачи поведения из одного места программы в другое так, будто это данные.
- Лямбда-выражения выглядят следующим образом: `BinaryOperator<Integer> add = (x, y) → x + y`.
- Функциональным интерфейсом называется интерфейс с единственным абстрактным методом; он используется в качестве типа лямбда-выражения.

Упражнения

В конце каждой главы имеются упражнения, чтобы вы могли проверить, насколько хорошо усвоен изложенный материал. Ответы к упражнениям можно найти на сайте GitHub.

1. Вопросы о функциональном интерфейсе `Function`.

   Пример 2.15 ❖ Функциональный интерфейс Function

   ```
   public interface Function<T, R> {
       R apply(T t);
   }
   ```

   a. Можете ли вы изобразить диаграмму этого функционального интерфейса?

b. Для каких лямбда-выражений можно было бы использовать этот функциональный интерфейс в программе калькулятора?
c. Какие из следующих лямбда-выражений являются корректными реализациями интерфейса `Function<Long, Long>`?

```
x -> x + 1;
(x, y) -> x + 1;
x -> x == 1;
```

2. *Лямбда-выражения и класс ThreadLocal.* В Java имеется класс `ThreadLocal`, который работает как контейнер значения, локального для текущего потока. В Java 8 появился новый фабричный метод для порождения экземпляров `ThreadLocal`, который принимает лямбда-выражение, позволяющее создать объект, не заводя новых подклассов.
   a. Найдите этот метод в Javadoc в своей интегрированной среде разработки (IDE).
   b. Класс `DateFormatter` в Java небезопасен относительно потоков. Используйте конструктор для создания потокобезопасного экземпляра `DateFormatter`, который печатает даты в виде «01-Jan-1970».
3. *Правила выведения типов.* Ниже приведено несколько примеров передачи лямбда-выражений функциям. Может ли `javac` правильно вывести типы аргументов для этих лямбда-выражений? Иначе говоря, будут ли они компилироваться?
   a. `Runnable helloWorld = () -> System.out.println("hello world");`
   b. Лямбда-выражение, выступающее в роли `ActionListener`:

```
JButton button = new JButton();
button.addActionListener(event ->
    System.out.println(event.getActionCommand()));
```

   c. Будет ли выведен тип в выражении `check(x -> x > 5)`, если имеются следующие перегруженные варианты `check`?

```
interface IntPred {
    boolean test(Integer value);
}
boolean check(Predicate<Integer> predicate);
boolean check(IntPred predicate);
```

Чтобы узнать, есть ли у метода несколько перегруженных вариантов, можно изучить документацию Javadoc или посмотреть типы аргументов в IDE.

Глава 3

Потоки

Изменения языка, появившиеся в Java 8, призваны помочь нам в написании более качественного кода. Наибольший вклад в достижение этой цели вносят новые базовые библиотеки, к рассмотрению которых мы и переходим. Наиболее существенные изменения в библиотеках касаются API коллекций и нового понятия *потоков*. Потоки позволяют писать код работы с коллекциями на более высоком уровне абстракции.

В интерфейсе `Stream` объявлены многочисленные функции, которые мы будет изучать в этой главе. Каждая из них соответствует какой-то типичной операции с объектами типа `Collection`.

От внешнего итерирования к внутреннему

Примеры в этой главе и остальной части книги ссылаются на классы предметной области, описанные выше в разделе «Пример предметной области».

В программах на Java часто приходится обходить коллекции, применяя операцию к каждому элементу по очереди. Например, код, показанный в примере 3.1, подсчитывает общее количество исполнителей родом из Лондона.

Пример 3.1 ❖ Подсчет исполнителей из Лондона в цикле for

```
int count = 0;
for (Artist artist : allArtists) {
    if (artist.isFrom("London")) {
        count++;
    }
}
```

Однако у такого подхода есть несколько недостатков. В нем много стереотипного кода, который приходится писать всякий раз, как мы хотим обойти какую-то коллекцию. Кроме того, такой цикл `for` трудно распараллелить – для этого пришлось бы написать несколько циклов.

Наконец, глядя на этот код, не сразу понимаешь, что хотел сделать программист. Стереотипная структура цикла `for` затемняет его смысл; чтобы докопаться до него, необходимо прочитать все тело цикла. Если цикл `for` всего один, то ничего страшного в этом нет, но когда есть значительный объем написанного кода, в котором встречается множество циклов (особенно вложенных), возникает проблема.

Если заглянуть под капот, то окажется, что цикл `for` – на самом деле синтаксическая глазурь, которая обертывает и скрывает итерирование. Стоит немного задержаться и посмотреть, что происходит под капотом. На первом шаге вызывается метод `iterator`, который создает новый объект `Iterator`, управляющий процессом итерирования. Этот механизм называется *внешним итерированием*. Далее в процессе итерирования явно вызываются методы `hasNext` и `next` объекта `Iterator`. В примере 3.2 приведен развернутый таким образом код, а на рис. 3.1 графически показано его выполнение.

Пример 3.2 ❖ Подсчет исполнителей из Лондона с помощью итератора

```
int count = 0;
Iterator<Artist> iterator = allArtists.iterator();
while(iterator.hasNext()) {
    Artist artist = iterator.next();
    if (artist.isFrom("London")) {
        count++;
    }
}
```

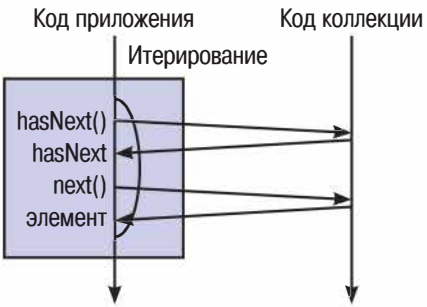

Рис. 3.1 ❖ Внешнее итерирование

Но у внешнего итерирования есть темные стороны. Во-первых, становится трудно абстрагировать различные операции, с которыми мы встретимся далее в этой главе. Кроме того, это принципиально по-

следовательное решение. В общем, выходит, что цикл `for` неразрывно соединяет две разные вещи: что мы хотим сделать и как мы хотим это сделать.

На рис. 3.2 показан альтернативный подход: *внутреннее итерирование*. Прежде всего обратите внимание на вызов метода `stream()`, который играет примерно такую же роль, как метод `iterator()` в предыдущем примере. Но вместо `Iterator` он возвращает эквивалентный интерфейс, предназначенный для внутреннего итерирования: `Stream`.

Пример 3.3 ❖ Подсчет исполнителей из Лондона с помощью внутреннего итерирования

```
long count = allArtists.stream()
                       .filter(artist -> artist.isFrom("London"))
                       .count();
```

На рис. 3.2 показана последовательность вызовов методов относительно библиотеки; сравните с рис. 3.1.

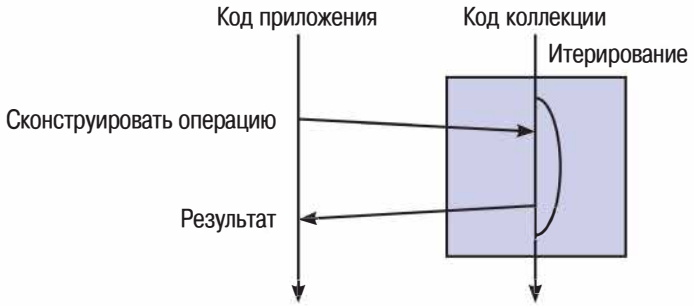

Рис. 3.2 ❖ Внутреннее итерирование

`Stream` – это средство конструирования сложных операций над коллекциями с применением функционального подхода.

Этот пример можно разбить на две более простые операции:

- найти всех исполнителей из Лондона;
- подсчитать количество элементов в списке исполнителей.

Обеим операциям соответствуют методы в интерфейсе `Stream`. Чтобы найти артистов из Лондона, мы вызываем метод `filter`. В данном случае фильтрация означает «оставить только элементы, удовлетворяющие условию». Условие же определяется функцией, которая возвращает `true`, если исполнитель родом из Лондона, и `false` в противном случае. Поскольку при работе с API потоков используется функцио-

нальное программирование, содержимое коллекции не изменяется; мы лишь объявляем, каким будет содержимое потока `Stream`. Метод `count()` подсчитывает количество объектов в данном потоке.

Что происходит на самом деле

В предыдущем примере я выделил две операции: фильтрацию и подсчет. Может показаться, что это чрезмерно расточительно, – ведь в примере 3.1 был только один цикл `for`. Складывается впечатление, что теперь нам понадобятся два цикла, коль скоро имеются две операции. На самом деле библиотека устроена так, что обход списка исполнителей производится только один раз.

Традиционно при вызове метода в Java компьютер что-то выполняет; например, метод `System.out.println("Hello World");` выводит строку на экран терминала. Некоторые методы интерфейса `Stream` работают иначе. Это обычные методы Java, но возвращенный объект типа `Stream` – это не новая коллекция, а рецепт создания коллекции. Подумайте, что делает код из примера 3.4. Не расстраивайтесь, если зайдете в тупик, – я сейчас все объясню!

Пример 3.4 ❖ Просто фильтр, никакого сбора данных нет

```
allArtists.stream()
          .filter(artist -> artist.isFrom("London"));
```

А делает он совсем немного – метод `filter` строит рецепт обработки `Stream`, но нет ничего такого, что привело бы этот рецепт в действие. Методы, которые, подобно `filter`, строят рецепты обработки `Stream`, но не приводят к порождению нового значения, называются *отложенными*. Методы, которые, подобно `count`, порождают конечное значение на основе последовательности `Stream`, называются *энергичными*.

Простейший способ посмотреть, что происходит, – добавить внутрь фильтра вызов `println`, который печатает имена исполнителей. В примере 3.5 приведен вариант программы с такой распечаткой. Если выполнить этот код, то программа не напечатает ничего.

Пример 3.5 ❖ Имена исполнителей не печатаются, потому что вычисление отложено

```
allArtists.stream()
          .filter(artist -> {
             System.out.println(artist.getName());
             return artist.isFrom("London");
          });
```

Если же точно такую же распечатку включить в состав потока, имеющего финальный шаг, например операции подсчета в листинге 3.3, то имена исполнителей будут напечатаны (пример 3.6).

Пример 3.6 ❖ Распечатка имен исполнителей

```
long count = allArtists.stream()
                       .filter(artist -> {
                          System.out.println(artist.getName());
                          return artist.isFrom("London");
                       })
                       .count();
```

Если выполнить код из примера 3.6, подав на вход список членов группы Битлз, то на экране будут напечатаны их имена, как показано в примере 3.7.

Пример 3.7 ❖ Пример распечатки имен исполнителей из группы Битлз

```
John Lennon
Paul McCartney
George Harrison
Ringo Starr
```

Понять, является операция отложенной или энергичной, очень просто; достаточно посмотреть, что она возвращает. Если возвращается `Stream`, значит, операция отложенная, если что-то другое, в том числе `void`, то энергичная. И это понятно, так как идея заключается в том, чтобы сформировать цепочку отложенных операций, а в конец поместить одну энергичную операцию, которая и порождает окончательный результат. Именно так работает пример подсчета, но это самый простой случай – всего две операции.

Подход в целом чем-то напоминает знакомый паттерн *Построитель*, в котором имеется последовательность вызовов, задающих свойства, или конфигурацию, за которыми следует единственный вызов метода `build`. Конечный объект не создается, пока не будет вызван метод `build`.

Уверен, вы задаетесь вопросом: «А зачем нужно это различие между отложенными и энергичными методами?» Отложив вычисление до момента, когда мы будем больше знать о желаемом результате и операциях, мы сможем затем выполнить его более эффективно. Хороший пример – нахождение первого элемента последовательности, который > 10. Для этого не нужно вычислять все элементы – лишь столько, сколько необходимо для получения первого подходящего.

Это также означает, что различные операции над коллекцией можно объединить и произвести ее обход только один раз.

Наиболее распространенные потоковые операции

Сейчас будет уместно сделать обзор наиболее распространенных операций класса `Stream`, чтобы лучше почувствовать, какие возможности предоставляет API. Поскольку мы рассмотрим лишь несколько наиболее важных примеров, я рекомендую заглянуть в документацию Javadoc по новому API и посмотреть, что есть еще.

collect(toList())

`collect(toList())` – энергичная операция, порождающая список из значений в объекте `Stream`.

Значения в `Stream`, над которыми выполняется операция, – это результат применения к исходным значениям рецепта, сконструированного последовательностью предыдущих вызовов методов объекта `Stream`. На самом деле `collect` – весьма общая и мощная конструкция, которую мы будем изучать более детально в главе 5. А пока приведем пример:

```
List<String> collected = Stream.of("a", "b", "c")
                               .collect(Collectors.toList());

assertEquals(Arrays.asList("a", "b", "c"), collected);
```

Здесь показано, как можно использовать `collect(toList())` для построения результирующего списка из объекта `Stream`. Важно помнить, что многие методы `Stream` отложенные, поэтому в конце цепочки вызовов должна находиться какая-нибудь энергичная операция, например `collect`.

На этом примере также виден формат, общий для всех последующих примеров в этом разделе. Сначала создается объект `Stream` из списка `List`. Затем следует какая-то операция и в конце вызов `collect` для преобразования потока в список. Напоследок мы с помощью утверждения проверяем, что результат совпадает с ожидаемым.

Открывающий вызов метода `stream` и закрывающий вызов `collect` или иного финального метода можно сравнить с *половинками булочки*. Они не определяют начинку нашего потокового гамбургера, но показывают, где начинаются и заканчиваются операции.

map

Если имеется функция, которая преобразует значение из одного типа в другой, то метод `map` позволит применить ее к потоку значений и тем самым породить поток новых значений.

Скоро вы поймете, что уже давным-давно занимаетесь отображением одного на другое. Допустим, требуется написать на Java код, который получает список строк и преобразует каждую из них в верхний регистр. Можно было бы обойти все элементы списка и для каждого вызвать метод `toUppercase`. А в конце поместить получившиеся значения в новый список. В примере 3.8 продемонстрирован именно такой подход.

Пример 3.8 ❖ Преобразование строк в верхний регистр в цикле for

```
List<String> collected = new ArrayList<>();
for (String string : asList("a", "b", "hello")) {
    String uppercaseString = string.toUpperCase();
    collected.add(uppercaseString);
}

assertEquals(asList("A", "B", "HELLO"), collected);
```

Операция `map` – одна из наиболее употребительных (см. рис. 3.3). Впрочем, вы, надо думать, и сами догадались об этом, приняв во внимание, как часто приходится писать что-то, подобное приведенному выше циклу `for`. В примере 3.9 показано, как сделать то же самое с помощью потокового API.

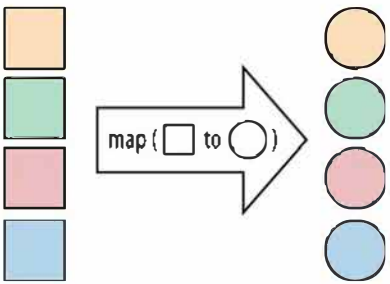

Рис. 3.3 ❖ Операция map

Пример 3.9 ❖ Преобразование строк в верхний регистр с помощью map

```
List<String> collected = Stream.of("a", "b", "hello")
                               .map(string -> string.toUpperCase())
                               .collect(toList());

assertEquals(asList("A", "B", "HELLO"), collected);
```

Лямбда-выражение, переданное `map`, принимает `String` в качестве единственного аргумента и возвращает `String`. Типы аргумента и результата не обязаны совпадать, но лямбда-выражение должно принадлежать типу `Function` (рис. 3.4) – универсальному функциональному интерфейсу с одним аргументом.

Рис. 3.4 ❖ Интерфейс Function

filter

Поймав себя на том, что перебираете коллекцию в цикле и проверяете каждый элемент, подумайте об использовании метода `filter` из интерфейса `Stream` (рис. 3.5).

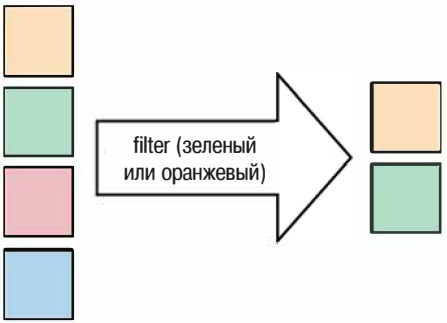

Рис. 3.5 ❖ Операция filter

Пример использования `filter` мы уже видели, так что можете пропустить этот раздел, если считаете, что все поняли. Вы еще здесь? Отлично! Допустим, имеется список строк и нужно найти среди них все, начинающиеся с цифры. То есть строка `"1abc"` подходит, а строка `"abc"` – нет. Можно было бы обойти список в цикле и в предложении `if` проверить первый символ, как показано в примере 3.10.

Пример 3.10 ❖ Обход списка в цикле и использование if

```
List<String> beginningWithNumbers = new ArrayList<>();
for(String value : asList("a", "1abc", "abc1")) {
    if (isDigit(value.charAt(0))) {
```

```
            beginningWithNumbers.add(value);
        }
}

assertEquals(asList("1abc"), beginningWithNumbers);
```

Уверен, вам доводилось писать подобный код, такая методика называется *фильтрацией*. Ее основная идея – оставить одни элементы коллекции и отбросить другие. В примере 3.11 показано, как переписать этот код в функциональном стиле.

Пример 3.11 ❖ Функциональный стиль

```
List<String> beginningWithNumbers
  = Stream.of("a", "1abc", "abc1")
          .filter(value -> isDigit(value.charAt(0)))
          .collect(toList());

assertEquals(asList("1abc"), beginningWithNumbers);
```

Как и `map`, метод `filter` принимает в качестве аргумента одну функцию – в данном случае мы воспользовались лямбда-выражением. Эта функция делает то же самое, что выражение в показанном выше предложении `if`, – возвращает `true`, если строка начинается цифрой. При переработке старого кода обращайте внимание на наличие `if` в середине цикла `for` – это верный признак того, что может пригодиться метод `filter`.

Поскольку задача этой функции такая же, как у предложения `if`, она должна возвращать `true` или `false`. После применения фильтра в потоке `Stream` остаются элементы, для которых функция вернула `true`. Функциональный интерфейс для такого рода функций – наш старый знакомый, `Predicate` (рис. 3.6).

Рис. 3.6 ❖ Интерфейс Predicate

flatMap

Метод `flatMap` (рис. 3.7) позволяет заменить значение объектом `Stream` и конкатенировать все потоки.

Мы уже встречались с операцией `map`, которая заменяет одно значение в потоке другим. Иногда бывает необходим вариант `map`, порож-

Рис. 3.7 ❖ Операция flatMap

дающий в качестве замены новый объект `Stream`. Но зачастую нам ни к чему поток потоков, и вот тогда на помощь приходит `flatMap`.

Рассмотрим простой пример. Имеется объект `Stream`, содержащий несколько списков чисел, и мы хотим получить все числа из всех списков. Эту задачу можно решить, как показано в примере 3.12.

Пример 3.12 ❖ Список потоков

```
List<Integer> together = Stream.of(asList(1, 2), asList(3, 4))
                              .flatMap(numbers -> numbers.stream())
                              .collect(toList());

assertEquals(asList(1, 2, 3, 4), together);
```

Мы заменяем каждый объект `List` объектом `Stream` с помощью метода `stream`, а остальное делает `flatMap`. Этому методу соответствует такой же функциональный интерфейс, как методу `map`, – `Function`, – но возвращать он может только потоки, а не произвольные значения.

max и min

К потокам довольно часто применяются операции нахождения минимума или максимума, для чего очень удобны методы `min` и `max`. Для демонстрации в примере 3.13 приведен код, который ищет самое короткое произведение в альбоме. Чтобы было проще убедиться в правильности результата, я явно перечислил все произведения (признаю, что это не самый известный альбом).

Пример 3.13 ❖ Поиск самого короткого произведения с помощью потоков

```
List<Track> tracks = asList(new Track("Bakai", 524),
                            new Track("Violets for Your Furs", 378),
```

```
                                new Track("Time Was", 451));

Track shortestTrack = tracks.stream()
                        .min(Comparator.comparing(track -> track.getLength()))
                        .get();

assertEquals(tracks.get(1), shortestTrack);
```

При поиске минимального и максимального элементов нужно прежде всего подумать о способе сравнения. Музыкальные произведения сравниваются по длительности звучания.

Чтобы сообщить объекту `Stream` о том, что произведения сравниваются по длительности, мы передаем ему объект `Comparator`. По счастью, в Java 8 добавлен статический метод `comparing`, который позволяет создавать компараторы, зная, как получить доступ к сравниваемым величинам. Раньше приходилось писать уродливый код, в котором мы извлекали поля из обоих сравниваемых объектов и сравнивали их значения. Теперь, чтобы получить одно и то же поле из обоих объектов, достаточно предоставить метод чтения. В данном случае таким методом является `getLength`.

Подумайте, что представляет собой метод `comparing`. Это функция, которая принимает и возвращает функцию. Звучит мудрено, но идея невероятно полезная. Такой метод можно было бы включить в стандартную библиотеку Java давным-давно, но из-за неудобочитаемости и многословности анонимных внутренних классов это оказалось бы непрактично. Теперь же, с появлением лямбда-выражений, выразить идею можно кратко и элегантно.

Метод `max` можно вызывать и для пустого объекта `Stream`, тогда он вернет значение типа `Optional`. Это довольно странный тип: он представляет значение, которое может как присутствовать, так и отсутствовать. Если поток пуст, то максимум не существует, иначе существует. Но не будем пока забивать себе голову деталями типа `Optional`, у нас еще будет случай поговорить о нем ниже. Важно лишь запомнить, что получить значение объекта этого типа можно с помощью метода `get`.

Проявляется общий принцип

Методы `max` и `min` – примеры более общего подхода к кодированию. Проще всего увидеть это, взяв код из примера 3.13 и переписав его в виде цикла `for`, потом можно будет установить общий принцип. Код в примере 3.14 делает то же, что и выше: ищет самое короткое произведение в альбоме, только на этот раз с применением цикла `for`.

Пример 3.14 ❖ Поиск самого короткого произведения с помощью цикла for

```
List<Track> tracks = asList(new Track("Bakai", 524),
                            new Track("Violets for Your Furs", 378),
                            new Track("Time Was", 451));

Track shortestTrack = tracks.get(0);
for (Track track : tracks) {
    if (track.getLength() < shortestTrack.getLength()) {
        shortestTrack = track;
    }
}

assertEquals(tracks.get(1), shortestTrack);
```

Сначала мы инициализируем переменную `shortestTrack`, записывая в нее первый элемент списка. Затем перебираем все произведения. Если найдено более короткое произведение, то мы записываем его в `shortestTrack` вместо текущего значения. Таким образом, по выходе из цикла в `shortestTrack` действительно окажется самое короткое произведение. Не сомневаюсь, что за свою карьеру вы написали тысячи циклов `for`, и многие из них были устроены точно так же. В примере 3.15 показан псевдокод, иллюстрирующий эту общую схему.

Пример 3.15 ❖ Принцип редукции

```
Object accumulator = initialValue;
for(Object element : collection) {
    accumulator = combine(accumulator, element);
}
```

Объект `accumulator` модифицируется в теле цикла и по выходе из него содержит конечное значение, которое мы хотели вычислить. Первоначально `accumulator` содержит значение `initialValue`, а затем комбинируется с каждым элементом списка путем вызова `combine`.

Конкретные реализации этой схемы отличаются только начальным значением `initialValue` и функцией `combine`. В приведенном выше примере в качестве `initialValue` мы брали первый элемент списка, но это совершенно необязательно. Для поиска самого короткого произведения функция `combine` возвращает минимум из двух значений: текущего элемента и аккумулятора.

Теперь можно взглянуть, как этот общий принцип воплощается в операции потокового API.

reduce

Операцию `reduce` стоит использовать, когда имеется коллекция значений, а нужно получить единственное значение в качестве результата. Рассмотренные выше методы `count`, `min` и `max` включены в стандартную библиотеку, потому что это распространенные частные случаи общего принципа редукции.

Продемонстрируем операцию `reduce` на примере суммирования потока чисел. Идея показана на рис. 3.8. Сначала аккумулятор равен 0 (сумма пустого потока), а затем мы комбинируем его с каждым элементом – складываем. По достижении последнего элемента потока аккумулятор будет содержать сумму всех элементов.

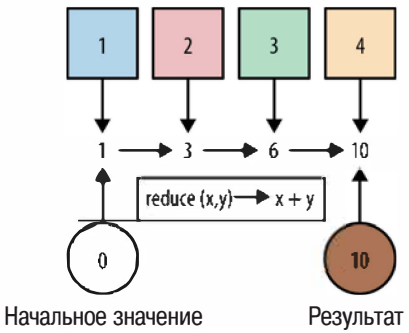

Рис. 3.8 ❖ Реализация сложения с помощью операции редукции

В примере 3.16 показано, как выглядит соответствующий код. Лямбда-выражение, которое называется *редуктором*, принимает два аргумента и возвращает их сумму. Аргумент `acc` – это аккумулятор, в котором накапливается сумма. Кроме него, редуктору передается текущий элемент потока.

Пример 3.16 ❖ Реализация суммирования с помощью reduce

```
int sum = Stream.of(1, 2, 3)
                .reduce(0, (acc, element) -> acc + element);

assertEquals(6, sum);
```

Лямбда-выражение возвращает новое значение `acc` – сумму прежнего значения и текущего элемента. Редуктор имеет тип `BinaryOperator`, с которым мы познакомились в главе 2.


В числе «примитивов», о которых пойдет речь на стр. 57, упоминается и метод `sum` из стандартной библиотеки. В реальных программах я рекомендую пользоваться им, а не показанным выше кодом.

В табл. 3.1 показаны промежуточные значения переменных при обработке каждого элемента потока. Можно было бы развернуть все вызовы функций, порождающие окончательный результат; получится код, показанный в примере 3.17.

Пример 3.17 ❖ Разворачивание reduce в последовательность вызовов функций

```
BinaryOperator<Integer> accumulator = (acc, element) -> acc + element;
int sum = accumulator.apply(
             accumulator.apply(
                accumulator.apply(0, 1),
             2),
          3);
```

Таблица 3.1. Вычисление суммы внутри reduce

element	acc	Результат
Нет	Нет	0
1	0	1
2	1	3
3	3	6

Теперь рассмотрим эквивалентный императивный код на Java (пример 3.18) и сравним функциональную и императивную версии.

Пример 3.18 ❖ Императивная реализация суммирования

```
int acc = 0;
for (Integer element : asList(1, 2, 3)) {
    acc = acc + element;
}
assertEquals(6, acc);
```

Как видно, в императивной версии аккумулятор – это переменная, обновляемая на каждой итерации цикла. А обновление заключается в сложении с элементом. Цикл является внешним по отношению к коллекции, и обновление переменной производится нами самостоятельно.

Объединение операций

При таком изобилии операций в интерфейсе `Stream` поиск нужной иногда оказывается сродни блужданию в лабиринте. Поэтому давай-

те поставим конкретную задачу и посмотрим, как разбить ее на несколько простых потоковых операций.

Первая задача звучит так: для данного альбома определить национальности всех групп-исполнителей. Исполнителем произведения может быть как один солист, так и целая группа. Мы воспользуемся знанием предметной области и с некоторой долей вольности предположим, что исполнитель, название которого начинается словом *The*, – на самом деле группа. Это не совсем так, но довольно близко к истине!

Прежде всего нужно согласиться, что решение не сводится к вызову какого-то одного метода API. Это не преобразование значений, как в `map`, не фильтрация и не получение единственного значения в конце обработки потока. Задачу можно следующим образом разбить на части.

1. Получить всех исполнителей для данного альбома.
2. Определить, какие из них являются группами.
3. Получить национальность каждой группы.
4. Объединить найденные значения.

Теперь уже проще понять, какие вызовы API соответствуют этим шагам.

1. В нашем классе `Album` есть замечательный метод `getMusicians`, который возвращает `Stream`.
2. С помощью метода `filter` оставим в потоке только исполнителей, являющихся группами.
3. С помощью метода `map` сопоставим группе ее национальность.
4. С помощью `collect(toList())` получим список национальностей.

Собрав все вместе, получаем такой код:

```
Set<String> origins = album.getMusicians()
                        .filter(artist -> artist.getName().startsWith("The"))
                        .map(artist -> artist.getNationality())
                        .collect(toSet());
```

В этом примере идиома сцепления операций проявляется более отчетливо. Вызовы `getMusicians`, `filter` и `map` возвращают объекты `Stream`, то есть являются отложенными, тогда как метод `collect` энергичный. Метод `map` – это функция, которая принимает лямбда-выражение и применяет его к каждому элементу `Stream`, возвращая новый объект `Stream`.

Наш предметный класс удобен в том смысле, что возвращает объект `Stream`, когда мы хотим получить список музыкантов, участвовавших

в записи альбома. В ваших собственных предметных классах, скорее всего, нет методов, возвращающих потоки, зато есть методы, которые возвращают коллекции, например `List` или `Set`. Ничего страшного – нужно будет лишь вызвать метод `stream` для `List` или `Set`.

Но, пожалуй, самое время задуматься о том, стоит ли раскрывать в модели предметной области объекты `List` или `Set` явным образом. Быть может, лучше создать фабрику объектов `Stream`. У доступа к коллекциям через интерфейс `Stream` есть большое преимущество – улучшенная инкапсуляция структуры данных модели. Если вы раскрываете только `Stream`, то пользователь ваших классов никоим способом не сможет повлиять на внутреннее содержимое `List` или `Set`.

К тому же у пользователей ваших классов будет стимул писать код в современном стиле Java 8. Никто не мешает производить рефакторинг постепенно – оставить существующие методы чтения и добавить к ним новые, ориентированные на потоковый API. Со временем старый код можно будет переписать и в конце концов полностью избавиться от методов чтения, возвращающих `List` или `Set`. Представьте, как вы будете довольны, почистив свой код от древних наслоений!

Рефакторинг унаследованного кода

Раз уж зашла речь о рефакторинге, рассмотрим пример унаследованного кода работы с коллекциями, основанного на циклах, и покажем процесс его постепенного преобразования в код на основе потоков. После каждого шага рефакторинга все тесты по-прежнему проходят, хотя тут вам придется либо поверить мне на слово, либо тестировать самостоятельно.

В этом примере мы ищем названия всех произведений в заданных альбомах, которые звучат дольше одной минуты. Унаследованный код показан в примере 3.19. Сначала мы инициализируем объект `Set`, в котором будут храниться названия всех произведений. Затем в двух вложенных циклах `for` обходим все альбомы и все произведения в каждом альбоме. Для каждого произведения мы смотрим, звучит ли оно дольше 60 секунд, и если это так, добавляем его название во множество названий.

Пример 3.19 ❖ Унаследованный код поиска названий произведений дольше одной минуты

```
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();
    for(Album album : albums) {
```

```
        for (Track track : album.getTrackList()) {
            if (track.getLength() > 60) {
                String name = track.getName();
                trackNames.add(name);
            }
        }
    }
    return trackNames;
}
```

Мы наткнулись на этот код в своем хранилище и обратили внимание на два вложенных цикла. Беглого взгляда на код не хватило, чтобы понять, что он делает, поэтому мы решили подвергнуть его рефакторингу. (Есть много способов переделать существующий код под потоки, ниже приведен только один из них. Когда вы получше познакомитесь с API, не будет нужды продвигаться вперед такими мелкими шажками. Но в педагогических целях полезно замедлить темп, по сравнению с профессиональной деятельностью.)

Первым делом мы изменим циклы `for`. Оставим пока прежний стиль кодирования в телах циклов, но воспользуемся методом `forEach` интерфейса `Stream`. Это удобно для подготовки к последующим шагам рефакторинга. Чтобы получить первый поток, вызовем метод `stream` списка альбомов. Напомним, что в предметном классе `Album` уже имеется метод `getTracks`, который возвращает поток произведений. Код, получившийся после шага 1, показан в примере 3.20.

Пример 3.20 ❖ Шаг 1 рефакторинга: поиск названий произведений дольше одной минуты

```
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();
    albums.stream()
        .forEach(album -> {
            album.getTracks()
                .forEach(track -> {
                    if (track.getLength() > 60) {
                        String name = track.getName();
                        trackNames.add(name);
                    }
                });
        });
    return trackNames;
}
```

На шаге 1 мы перешли на использование потоков, но еще не раскрыли весь их потенциал. Более того, код даже стал безобразнее, чем был! Что ж, самое время двигаться дальше в сторону потоков. Первой мишенью разумно будет избрать внутренний вызов `forEach`.

Внутри него делаются три вещи: поиск произведений дольше минуты, получение их названий и добавление названий в коллекцию Set. Следовательно, нам понадобятся три операции Stream. Поиск произведений, отвечающих условию, – это работа для filter. С переходом от произведения к его названию отлично справится map. Но нам еще нужно добавлять названия в Set, поэтому финальной операцией пока останется forEach. После разбиения внутреннего вызова forEach на части получается код, показанный в примере 3.21.

Пример 3.21 ❖ Шаг 2 рефакторинга: поиск названий произведений дольше одной минуты

```
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();
    albums.stream()
        .forEach(album -> {
            album.getTracks()
                 .filter(track -> track.getLength() > 60)
                 .map(track -> track.getName())
                 .forEach(name -> trackNames.add(name));
        });
    return trackNames;
}
```

Внутренний цикл стал больше походить на потоковое решение, но все еще напоминает пирамиду судьбы. Не надо нам вложенных потоковых операций, мы хотим иметь одну простую и ясную последовательность вызовов методов.

Что нам в действительности нужно? Каким-то образом преобразовать альбом в поток произведений. Мы знаем, что для *трансформации* или *подмены* предназначена операция map. Здесь же нужен более продвинутый вариант map – операция flatMap, которая возвращает конкатенацию отдельных результирующих потоков. Заменив вызов forEach на flatMap, получаем код, показанный в примере 3.22.

Пример 3.22 ❖ Шаг 3 рефакторинга: поиск названий произведений дольше одной минуты

```
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();
    albums.stream()
          .flatMap(album -> album.getTracks())
          .filter(track -> track.getLength() > 60)
          .map(track -> track.getName())
          .forEach(name -> trackNames.add(name));
    return trackNames;
}
```

Гораздо лучше, правда? Вместо двух вложенных циклов `for` мы имеем одну понятную последовательность вызовов методов, выполняющую всю операцию целиком. Но цели мы еще не достигли. Объект `Set` по-прежнему создается вручную, и элементы в него тоже добавляются вручную. Хотелось бы, чтобы все вычисление состояло только из цепочки потоковых вызовов.

Я еще не показывал рецепта этой трансформации, но с его близкими приятелями мы уже встречались. Чтобы в конце построить список `List` значений, мы вызывали `collect(toList())`. Ну а для построения множества `Set` нужно вызвать `collect(toSet())`. Так что заменим последний вызов `forEach` обращением к `collect`, после чего можно будет удалить переменную `trackNames`.

Пример 3.23 ❖ Шаг 4 рефакторинга: поиск названий произведений дольше одной минуты

```
public Set<String> findLongTracks(List<Album> albums) {
    return albums.stream()
                 .flatMap(album -> album.getTracks())
                 .filter(track -> track.getLength() > 60)
                 .map(track -> track.getName())
                 .collect(toSet());
}
```

Что в итоге? Мы взяли кусок унаследованного кода и, подвергнув его рефакторингу, перешли к идиоматическому представлению с использованием потоков. Начали мы просто с включения потоков в код без применения каких-либо операций над ними. А на каждом шаге идиоматичность повышалась. И после каждого шага я прогонял автономные тесты, дабы удостовериться, что код по-прежнему работает. Это очень полезно при рефакторинге унаследованного кода.

Несколько потоковых вызовов

Можно было бы не сцеплять вызовы методов, а принудительно вычислять результат каждого шага. *Умоляю вас* – не делайте так. В примере 3.24 показано, как в этом стиле можно было бы написать приведенный выше код поиска национальностей участников группы. Для сравнения в примере 3.25 повторен исходный код.

Пример 3.24 ❖ Неправильное использование потоков

```
List<Artist> musicians = album.getMusicians()
                              .collect(toList());

List<Artist> bands = musicians.stream()
```

```
                        .filter(artist -> artist.getName().startsWith("The"))
                        .collect(toList());

Set<String> origins = bands.stream()
                          .map(artist -> artist.getNationality())
                          .collect(toSet());
```

Пример 3.25 ❖ Идиоматическое сцепление потоковых вызовов

```
Set<String> origins = album.getMusicians()
                       .filter(artist -> artist.getName().startsWith("The"))
                       .map(artist -> artist.getNationality())
                       .collect(toSet());
```

Версия из примера 3.24 хуже идиоматической по нескольким причинам.

- Труднее понять, что в ней происходит, потому что доля стереотипного кода, не имеющего отношения к собственно бизнес-логике, выше.
- Она менее эффективна, так как на каждом промежуточном шаге энергично создается новая коллекция объектов.
- Код загромождён лишними переменными, нужными только для хранения промежуточных результатов.
- Операции сложнее автоматически распараллелить.

Разумеется, когда вы только начинаете писать код с потоками, появление такого рода фрагментов можно считать нормальным. Но если вы замечаете, что такие блоки появляются уж слишком часто, то стоит сделать паузу и подумать, нельзя ли преобразовать их в более лаконичную и удобочитаемую форму.

Если от изобилия сцепленных вызовов вам несколько неуютно, не расстраивайтесь – это совершенно нормально. Со временем вы наберетесь опыта, и такой код будет казаться вам вполне естественным. И в любом случае это не причина для того, чтобы разбивать цепочки операций, как показано в примере 3.24. Расположение операций в отдельных строках, как при использовании паттерна Построитель, также вселяет чувство уверенности.

Функции высшего порядка

На протяжении этой главы мы постоянно встречались с тем, что в функциональном программировании называют *функциями высшего порядка*. Это функция, которая либо принимает другую функцию в качестве аргумента, либо возвращает функцию в качестве значения. Выявить функцию высшего порядка очень просто, достаточно взглянуть на ее сигнатуру. Если тип аргумента или возвращаемого

значения – функциональный интерфейс, значит, мы имеем функцию высшего порядка.

`map` – функция высшего порядка, потому что ее аргумент `mapper` – функция. Вообще, почти все рассмотренные нами методы интерфейса `Stream` – функции высшего порядка. В примере сортировки мы использовали также функцию `comparing`, которая не только принимает другую функцию для извлечения сравниваемых полей, но и возвращает новый объект `Comparator`. Можно, конечно, считать `Comparator` объектом, но у него есть единственный абстрактный метод и, значит, это функциональный интерфейс.

На самом деле можно высказать еще более сильное утверждение. Интерфейс `Comparator` был придуман, когда возникла необходимость в функции, но в то время в Java не было ничего, кроме объектов, поэтому был создан тип класса – анонимный класс, с которым можно было обращаться, как с функцией. Тот факт, что это объект – несущественная подробность. Функциональные интерфейсы – шаг в правильном направлении.

Полезное применение лямбда-выражений

Знакомя вас с лямбда-выражениями, я в качестве примера привел обратный вызов для печати какой-то информации. Это вполне допустимое использование лямбда-выражения, но так мы не сделаем код ни более простым, ни более абстрактным, потому что все равно просим компьютер выполнить некую операцию. Устранение стереотипного кода – вещь хорошая, но этим достоинства лямбда-выражений в Java 8 отнюдь не исчерпываются.

Рассмотренные в этой главе идеи позволяют писать более простой код, поскольку описывают операции над данными в терминах того, *какое* преобразование выполнить, а не *как* это сделать. В итоге получается код, в котором меньше шансов для появления ошибок, а намерения программиста выражены более отчетливо.

Еще один аспект перехода к «что» от «как» – идея функции *без побочных эффектов*. Такие функции важны, потому что позволяют оценить все последствия работы функции, взглянув лишь на возвращаемое ей значение.

Функции без побочных эффектов не изменяют состояния программы или внешнего мира. У первого лямбда-выражения, приведенного в этой главе, побочный эффект был, потому что она печатала что-то

на экране, то есть мы *наблюдали* побочный эффект функции. А как насчет следующего примера?

```
private ActionEvent lastEvent;

private void registerHandler() {
    button.addActionListener((ActionEvent event) -> {
        this.lastEvent = event;
    });
}
```

Здесь мы запоминаем параметр `event` в поле. Это более тонкий побочный эффект: присваивание значения переменной. На выводе программы он никак не отражается, но ее состояние изменилось. Но Java полагает определенные пределы в этом отношении. Взгляните на присваивание переменной `localEvent` в следующем фрагменте:

```
ActionEvent localEvent = null;
    button.addActionListener(event -> {
        localEvent = event;
    });
```

Здесь мы пытаемся записать тот же параметр `event` в локальную переменную. Только не надо слать мне уведомления об ошибках в тексте книги – я знаю, что этот код не откомпилируется! И это было сознательное решение проектировщиков языка: лямбда-выражения должны использоваться для захвата значений, а не переменных. Захват значений побуждает писать код без побочных эффектов, поскольку альтернатива труднее. В главе 2 я уже говорил, что хотя внутри лямбда-выражения и можно использовать переменные, объявленные без ключевого слова `final`, они все равно должны быть *эффективно* финальными.

Передавая лямбда-выражения в методы интерфейса `Stream`, являющиеся функциями высшего порядка, стремитесь избегать побочных эффектов. Единственное исключение – метод `forEach`, который является финальной операцией.

Основные моменты

- Внутреннее итерирование – это способ обхода коллекции, при котором у самой коллекции оказывается больше контроля над итерированием.
- `Stream` – аналог `Iterator`, только с внутренним итерированием.

- Многие распространенные операции над коллекциями можно выполнить, комбинируя методы интерфейса `Stream` с лямбда-выражениями.

Упражнения


Ответы к упражнениям можно найти на сайте GitHub.

1. *Распространенные операции Stream*. Реализуйте:
   a. Функцию сложения чисел, то есть `int addUp(Stream<Integer> numbers)`.
   b. Функцию, которая получает исполнителя и возвращает список строк, содержащих имена и место происхождения.
   c. Функцию, которая получает альбомы и возвращает список альбомов, содержащих не более трех произведений.
2. *Итерирование*. Перепишите этот код с использованием внутреннего итерирования вместо внешнего.

```
int totalMembers = 0;
for (Artist artist : artists) {
    Stream<Artist> members = artist.getMembers();
    totalMembers += members.count();
}
```

3. *Вычисление*. Взгляните на сигнатуры следующих методов интерфейса `Stream`. Являются они энергичными или отложенными?
   a. `boolean anyMatch(Predicate<? super T> predicate);`
   b. `Stream<T> limit(long maxSize);`
4. *Функции высшего порядка*. Являются ли следующие методы `Stream` функциями высшего порядка? Почему?
   a. `boolean anyMatch(Predicate<? super T> predicate);`
   b. `Stream<T> limit(long maxSize);`
5. *Чистые функции*. Является ли следующее лямбда-выражение свободным от побочных эффектов, или оно изменяет состояние?

```
x -> x + 1
```

Рассмотрим пример кода:

```
AtomicInteger count = new AtomicInteger(0);
List<String> origins = album.musicians()
                            .forEach(musician -> count.incAndGet();)
```

   a. Лямбда-выражение, переданное в `forEach` в данном примере.

6. Подсчитайте количество строчных букв в строке (подсказка: воспользуйтесь методом `chars` класса `String`).
7. Пусть дан список строк `List<String>`. Найдите в нем строку, содержащую максимальное число строчных букв. Чтобы код правильно работал, когда входной список пуст, можете возвращать объект типа `Optional<String>`.

Упражнения повышенной сложности

1. Напишите реализацию метода `map` интерфейса `Stream`, пользуясь только методом `reduce` и лямбда-выражениями. Если хотите, можете возвращать `List` вместо `Stream`.
2. Напишите реализацию метода `filter` интерфейса `Stream`, пользуясь только методом `reduce` и лямбда-выражениями. Как и раньше, можете возвращать `List` вместо `Stream`.

Глава 4

Библиотеки

Я рассказал о том, как писать лямбда-выражения, но пока не затронул другую сторону медали: как их использовать. А это важно, даже если вы не собираетесь писать такую насыщенную функциональными конструкциями библиотеку, как потоковый API. Даже в самом простом приложении может найтись место для трактовки кода как данных.

Еще одно новшество в Java 8, изменившее наше представление о библиотеках, – появление методов по умолчанию и статических методов в интерфейсах. Это означает, что методы, объявленные в интерфейсах, отныне могут иметь тела и содержать код.

В этой главе я восполню еще несколько пробелов, в частности расскажу о том, что происходит, когда лямбда-выражения перегружают методы, и о том, как используются примитивы. Все это важно знать при написании кода с лямбда-выражениями.

Использование лямбда-выражений в программе

В главе 2 я сказал, что лямбда-выражение имеет тип функционального интерфейса, и описал, как этот тип выводится. С точки зрения вызывающей программы, вызов лямбда-выражения ничем не отличается от вызова метода интерфейса.

Рассмотрим конкретный пример, взятый из области библиотек протоколирования. В нескольких подобных библиотеках на Java, в том числе `slf4j` и `log4j`, имеются методы, которые выводят что-то в журнал, лишь если установленный уровень протоколирования не ниже определенного порога. Например, метод `void debug(String message)` выведет сообщение `message`, только если уровень протоколирования не ниже `debug`.

К сожалению, затраты на само построение сообщения `message` часто не являются пренебрежимо малыми. Поэтому возникает ситуация, когда программист начинает явно вызывать метод `isDebugEnabled`, что-

бы оптимизировать эти затраты. Соответствующий код показан в примере 4.1. Хотя при обращении к методу `debug` сообщение не попало бы в журнал, мы все же вызываем накладный метод `expensiveOperation` и конкатенируем его результат со строкой. Поэтому явная проверка уровня протоколирования в предложении `if` оказывается быстрее.

Пример 4.1 ❖ Вызывается метод isDebugEnabled, чтобы избежать накладных расходов

```
Logger logger = new Logger();
if (logger.isDebugEnabled()) {
    logger.debug("Look at this: " + expensiveOperation());
}
```

А хотелось бы иметь возможность передать лямбда-выражение, которое порождает строку сообщения. Это выражение вызывалось бы лишь в том случае, когда уровень протоколирования действительно не ниже `debug`. При таком подходе мы могли бы переписать показанный выше код в следующем виде (пример 4.2).

Пример 4.2 ❖ Применение лямбда-выражений упрощает код протоколирования

```
Logger logger = new Logger();
logger.debug(() -> "Look at this: " + expensiveOperation());
```

И как реализовать такой метод в классе `Logger`? С точки зрения библиотеки, мы можем просто использовать встроенный функциональный интерфейс `Supplier`, имеющий единственный метод `get`. Затем можно с помощью `isDebugEnabled` определить, нужно ли вызывать этот метод, и если да, то передать результат в метод `debug`. Соответствующий код приведен в примере 4.3.

Пример 4.3 ❖ Реализация регистратора с применением лямбда-выражения

```
public void debug(Supplier<String> message) {
    if (isDebugEnabled()) {
        debug(message.get());
    }
}
```

Вызов метода `get()` в этом примере соответствует вызову лямбда-выражения, переданного в вызываемый метод. Это решение работает и с анонимными внутренними классами, что позволяет сохранить обратную совместимость API для клиентов вашего кода, которые еще не перешли на Java 8. Важно помнить, что фактическое имя метода зависит от конкретного функционального интерфейса. Так, если бы

мы использовали интерфейс `Predicate`, то метод назывался бы `test`, а в интерфейсе `Function` он называется `apply`.

Примитивы

Вы, наверное, обратили внимание, что в предыдущем разделе мы мельком упомянули *примитивные* типы. В Java имеются пары типов – например, `int` и `Integer` – один из которых примитивный, а другой *упакованный*. Примитивные типы встроены в язык и в среду исполнения в качестве фундаментальных структурных единиц; упакованные типы – это обычные классы Java, обертывающие примитивы.

Поскольку механизм универсальных типов в Java основан на *стирании* типа универсального параметра – как будто это экземпляр класса `Object`, – то в роли типов-аргументов могут выступать только упакованные типы. Именно поэтому список целых в Java объявляется как `List<Integer>`, а не `List<int>`.

Но раз упакованные типы – это объекты, с их хранением сопряжены накладные расходы. Так, `int` занимает 4 байта памяти, а `Integer` – уже 16 байтов. Проблема еще обостряется, когда речь идет о массивах чисел, потому что размер каждого элемента массива примитивов равен размеру этого примитива, тогда как элемент массива упакованных типов – это указатель на объект в куче Java. В худшем случае массив `Integer[]` занимает в шесть раз больше памяти, чем массив `int[]` того же размера.

Преобразование примитивного типа в упакованный – *упаковка* – также не обходится без накладных расходов. Как, впрочем, и обратное преобразование – *распаковка*. Если алгоритм подразумевает выполнение большого количества численных операций, то затраты на упаковку и распаковку в сочетании с дополнительным расходом памяти на хранение упакованных объектов могут заметно снизить его производительность.

Из-за описанных накладных расходов в потоковой библиотеке некоторые функции имеют по нескольку версий: для примитивных и для упакованных типов. Функция высшего порядка `mapToLong` и функция `ToLongFunction`, показанная на рис. 4.1, – примеры такого рода решений. В Java 8 специализации существуют только для примитивных типов `int`, `long` и `double`, потому что именно они в наибольшей степени влияют на производительность численных алгоритмов.

К специализациям для примитивных типов применяется очень четкое соглашение об именовании. Если тип возвращаемого значе-

Рис. 4.1 ❖ ToLongFunction

ния примитивный, то имя интерфейса начинается словом `To` и именем примитивного типа, например `ToLongFunction` (как на рис. 4.1). Если аргумент имеет примитивный тип, то имя интерфейса начинается просто с имени типа, например `LongFunction` (рис. 4.2). Если в функции высшего порядка используется примитивный тип, то ее имя заканчивается словом `To`, за которым следует имя примитивного типа, например `mapToLong`.

Рис. 4.2 ❖ LongFunction

Существуют также специализированные версии интерфейса `Stream` для примитивных типов, они начинаются именем типа, например `LongStream`. На самом деле методы наподобие `mapToLong` возвращают не `Stream`, а объекты специализированных потоков. В специализированном потоке реализация `map` также специализирована: она принимает функцию `LongUnaryOperator` (рис. 4.3), которая преобразует `long` в `long`. Можно также перейти от примитивного потока к упакованному с помощью вариантов функции высшего порядка, например `mapToObj`, и метода `boxed`, который возвращает поток упакованных объектов, например `Stream<Long>`.

Рис. 4.3 ❖ LongUnaryOperator

Рекомендуется всюду, где возможно, пользоваться функциями, специализированными для примитивных типов, потому что они работают быстрее. К тому же у специализированных потоков имеются дополнительные функциональные возможности. Это позволяет не реализовывать заново уже имеющуюся функциональность и писать код,

который лучше передает намерение операций над числами. Пример использования дополнительной функциональности приведен ниже.

Пример 4.4 ❖ Использование метода summaryStatistics для получения представления о распределении длительности звучания произведений

```
public static void printTrackLengthStatistics(Album album) {
    IntSummaryStatistics trackLengthStats
        = album.getTracks()
               .mapToInt(track -> track.getLength())
               .summaryStatistics();

    System.out.printf("Max: %d, Min: %d, Ave: %f, Sum: %d",
                  trackLengthStats.getMax(),
                  trackLengthStats.getMin(),
                  trackLengthStats.getAverage(),
                  trackLengthStats.getSum());
}
```

Здесь мы печатаем на экране сводные данные о длительности произведений. Вместо того чтобы производить вычисления самостоятельно, мы отображаем каждое произведение на его длительность с помощью метода потока, специализированного для примитива, `mapToInt`. Поскольку этот метод возвращает объект типа `IntStream`, мы можем вызвать его метод `summaryStatistics`, который вычисляет различные статистические характеристики: минимум, максимум, среднее и сумму.

Этот метод имеется во всех специализированных потоках, в частности в `DoubleStream` и `LongStream`. Можно также вычислять отдельные характеристики, если все сразу не нужны; для этого существуют методы `min`, `max`, `average` и `sum`.

Разрешение перегрузки

В Java разрешается *перегружать* методы, то есть иметь несколько методов с одним и тем же именем, но разными сигнатурами. Однако при этом возникает проблема при выведении типов параметров, так как появляется возможность вывести разные типы. В таких случаях `javac` выбирает *самый специфический* тип. Так, в примере 4.5 из двух методов, показанных в примере 4.6, будет выбран тот, что печатает `String`, а не `Object`.

Пример 4.5 ❖ Метод, при разрешении которого можно выбрать один из двух методов

```
overloadedMethod("abc");
```

Пример 4.6. Два перегруженных метода

```
private void overloadedMethod(Object o) {
    System.out.print("Object");
}

private void overloadedMethod(String s) {
    System.out.print("String");
}
```

Тип `BinaryOperator` – это частный случай типа `BiFunction`, для которого типы аргументов и возвращаемого значения одинаковы. Например, функция сложения двух целых чисел имеет тип `BinaryOperator`.

Поскольку тип лямбда-выражения совпадает с типом его функционального интерфейса, при передаче их в качестве аргументов применимы те же правила. Мы можем иметь два перегруженных варианта метода: с параметром типа `BinaryOperator` и параметром типа расширяющего его интерфейса. При вызове такого метода Java выведет в качестве типа лямбда-выражения тип самого специфичного функционального интерфейса. Так, при наличии двух методов, показанных в примере 4.8, код из примера 4.7 напечатает `IntegerBinaryOperator`.

Пример 4.7 ❖ Еще один пример вызова перегруженного метода

```
overloadedMethod((x, y) -> x + y);
```

Пример 4.8 ❖ Выбор между двумя перегруженными методами

```
private interface IntegerBiFunction extends BinaryOperator<Integer> {
}

private void overloadedMethod(BinaryOperator<Integer> lambda) {
    System.out.print("BinaryOperator");
}

private void overloadedMethod(IntegerBiFunction lambda) {
    System.out.print("IntegerBinaryOperator");
}
```

Разумеется, если есть несколько перегруженных вариантов метода, «самый специфичный тип» существует не всегда. Взгляните на пример 4.9.

Пример 4.9 ❖ Ошибка компиляции из-за невозможности выбрать один из перегруженных методов

```
overloadedMethod((x) -> true);

private interface IntPredicate {
```

```
    public boolean test(int value);
}

private void overloadedMethod(Predicate<Integer> predicate) {
    System.out.print("Predicate");
}

private void overloadedMethod(IntPredicate predicate) {
    System.out.print("IntPredicate");
}
```

Лямбда-выражение, переданное методу `overloadedMethod`, совместимо как с обычным типом `Predicate`, так и с типом `IntPredicate`. И для каждого типа имеется перегруженный вариант метода. В таком случае `javac` отказывается компилировать код, жалуясь на неоднозначность: поскольку `IntPredicate` не расширяет `Predicate`, компилятор не может сказать, какой тип более специфичен.

Чтобы исправить ошибку, нужно привести лямбда-выражение к одному из типов `IntPredicate` или `Predicate<Integer>` – в зависимости от того, какое поведение вам нужно. Конечно, если бы вы проектировали библиотеку сами, то могли бы заподозрить этот код в «попахивании» и прийти к выводу, что перегруженные методы надо бы переименовать.

Подведем итог: параметры-типы лямбда-выражения выводятся из *целевого типа*, при этом действуют следующие правила:

- если существует всего один целевой тип, то тип лямбда-выражения выводится из типа соответствующего аргумента метода функционального интерфейса;
- если возможных целевых типов несколько, выбирается самый специфичный из них;
- если возможных целевых типов несколько и самого специфичного не существует, тип необходимо указать явно.

Аннотация @FunctionalInterface

В главе 2 я уже говорил о том, что такое функциональный интерфейс, но не упомянул об аннотации `@FunctionalInterface`. Этой аннотацией следует снабжать любой интерфейс, который предполагается использовать как функциональный.

А что она означает? Дело в том, что в Java есть интерфейсы с единственным методом, которые вообще-то не предназначены для реализации лямбда-выражениями. Например, может предполагаться, что

у объекта есть какое-то внутреннее состояние, а наличие всего одного метода – случайное совпадение. В качестве примеров приведу интерфейсы `java.lang.Comparable` и `java.io.Closeable`.

Если класс реализует интерфейс `Comparable`, значит, между его экземплярами определено отношение порядка, например лексикографический порядок на строках. Обычно мы не рассматриваем функции как допускающие сравнение объекты, поскольку в них нет полей и внутреннего состояния, а если состояния нет, то что можно сравнивать?

С другой стороны, чтобы объект принадлежал типу `Closeable`, он должен хранить какой-то открытый ресурс, например описатель файла, который надлежит закрыть в будущем. Метод такого интерфейса не может быть чистой функцией, поскольку закрытие ресурса – еще один пример изменения состояния.

В отличие от `Closeable` и `Comparable`, все новые интерфейсы, предназначенные для совместной работы с интерфейсом `Stream`, рассчитаны на реализацию лямбда-выражениями. Они и существуют-то только для того, чтобы можно было оформить код как данные. Поэтому к ним следует применять аннотацию `@FunctionalInterface`.

Наличие этой аннотации заставляет `javac` проверить, отвечает ли интерфейс критерию «функциональности». Если она применена к `enum`, `class` или `annotation`, либо в интерфейсе объявлено более одного абстрактного метода, то `javac` выдаст ошибку. Это очень полезно для обнаружения ошибок во время рефакторинга.

Двоичная совместимость интерфейсов

В главе 3 мы видели, что одно из самых значительных изменений в Java 8 претерпела библиотека коллекций. По мере эволюционирования Java всегда сохранялась обратная двоичная совместимость. На практике это означает, что библиотека или приложение, откомпилированные в любой версии Java от 1 до 7, будут без всяких изменений работать в Java 8.

Разумеется, ошибки случаются, но, по сравнению со многими другими программными платформами, двоичная совместимость неизменно считалась одним из ключевых преимуществ Java. Если не считать добавления новых ключевых слов, например `enum`, всегда прилагались усилия для сохранения также совместимости на уровне исходного кода. Есть гарантия, что исходный код, написанный на Java 1–7, будет компилироваться в Java 8.

Подобные гарантии очень трудно обеспечить при изменении таких основополагающих компонентов, как библиотека коллекций. В качестве умозрительного упражнения рассмотрим конкретный пример. В интерфейс `Collection` в Java 8 был добавлен метод `stream`, а это означает, что в любом классе, реализующем `Collection`, такой метод должен существовать. Для классов из самих базовых библиотек (например, ArrayList) проблему решить легко – нужно лишь реализовать этот метод.

К несчастью, это не поможет устранить нарушение двоичной совместимости, потому что существует немало классов вне JDK, реализующих интерфейс `Collection` (допустим, `MyCustomList`), и в них тоже должен быть реализован метод `stream`. Следовательно, в Java 8 класс `MyCustomList` перестанет компилироваться, а если у вас есть его версия, откомпилированная ранее, то при попытке загрузить `MyCustomList` в виртуальную машину Java (JVM) `ClassLoader` возбудит исключение.

Такого кошмарного развития событий, угрожающего всем сторонним библиотекам, удалось избежать, но ценой введения нового языкового механизма: *методов по умолчанию*.

Методы по умолчанию

Итак, в интерфейсе `Collection` появился новый метод `stream`; каким образом класс `MyCustomList` удается откомпилировать, хотя он представления не имеет о существовании нового метода? В Java 8 эта проблема решается за счет того, что интерфейс `Collection` может сказать: «Если у какого-то из моих потомков нет метода `stream`, пусть он воспользуется вот этим». Такие методы интерфейса называются методами *по умолчанию*. Они могут существовать в любом интерфейсе, а не только функциональном.

Еще один добавленный метод по умолчанию – `forEach` в интерфейсе `Iterable`. Он предоставляет функциональность, аналогичную циклу `for`, но позволяет использовать лямбда-выражение в качестве тела цикла. В примере 4.10 показано, как это могло бы быть реализовано в JDK.

Пример 4.10 ❖ Пример метода по умолчанию, демонстрирующий возможную реализацию forEach

```
default void forEach(Consumer<? super T> action) {
    for (T t : this) {
        action.accept(t);
    }
}
```

Теперь, когда идея о том, что лямбда-выражения можно использовать, просто вызывая методы интерфейсов, не кажется вам чуждой, этот пример не должен вызывать недоумения. Здесь в обычном цикле `for` производится обход объекта `Iterable`, и для каждого значения вызывается метод `accept`.

Но если все так просто, зачем бы и затевать разговор? Важный момент – ключевое слово `default` в начале определения метода. Оно говорит `javac`, что мы хотим добавить метод в интерфейс. Помимо нового ключевого слова, для методов по умолчанию действуют немного отличающиеся правила наследования.

Еще одно существенное отличие состоит в том, что, в отличие от классов, у интерфейсов не может быть полей, поэтому методы по умолчанию могут модифицировать состояние дочерних классов, только вызывая их методы. Это позволяет избежать предположений о конкретной реализации потомков.

Методы по умолчанию и наследование

Существуют некоторые тонкие моменты, касающиеся того, как методы по умолчанию переопределяют другие методы и переопределяются сами. Начнем с простейшего случая: переопределение отсутствует. В примере 4.11 в интерфейсе `Parent` определен метод `welcome`, который отправляет сообщение. В классе `ParentImpl` метод `welcome` не реализован, поэтому наследуется метод по умолчанию.

Пример 4.11 ❖ Интерфейс Parent: метод welcome является методом по умолчанию

```
public interface Parent {

    public void message(String body);

    public default void welcome() {
        message("Parent: Hi!");
    }

    public String getLastMessage();
}
```

При том использовании этого класса, которое показано в примере 4.12, вызывается метод по умолчанию, и утверждение оказывается истинным.

Пример 4.12 ❖ Использование метода по умолчанию в клиентском коде

```
@Test
public void parentDefaultUsed() {
    Parent parent = new ParentImpl();
```

```
    parent.welcome();
    assertEquals("Parent: Hi!", parent.getLastMessage());
}
```

Теперь расширим интерфейс `Parent`, создав интерфейс `Child`, код которого приведен в примере 4.13. В `Child` реализован собственный метод по умолчанию `welcome`. Мы интуитивно ожидаем, что метод по умолчанию в интерфейсе `Child` переопределяет метод по умолчанию в `Parent`. В данном случае класс `ChildImpl` также не предоставляет реализации `welcome`, и, значит, наследуется метод по умолчанию.

Пример 4.13 ❖ Интерфейс Child расширяет Parent

```
public interface Child extends Parent {

    @Override
    public default void welcome() {
        message("Child: Hi!");
    }
}
```

Эта иерархия классов изображена на рис. 4.4.

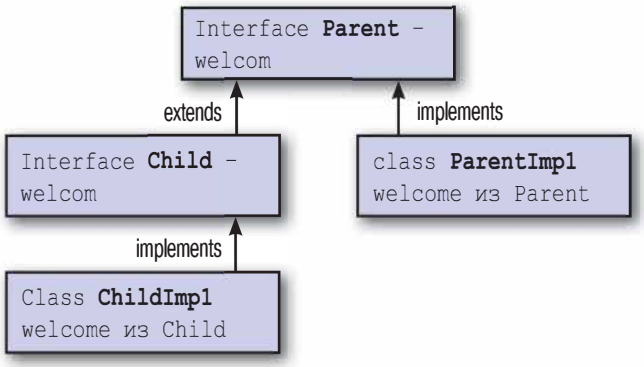

Рис. 4.4 ❖ Иерархия наследования к этому моменту

В примере 4.14 этот метод вызывается и отправляет сообщение `"Child: Hi!"`.

Пример 4.14 ❖ Клиентский код вызывает метод интерфейса Child

```
@Test
public void childOverrideDefault() {
    Child child = new ChildImpl();
    child.welcome();
    assertEquals("Child: Hi!", child.getLastMessage());
}
```

Метод по умолчанию является виртуальным, а не статическим. Это означает, что в случае выбора между ним и методом, переопределенным в классе, предпочтение всегда отдается последнему. Этот принцип проиллюстрирован в примерах 4.15 и 4.16, где выбран метод `welcome` из класса `OverridingParent`, а не из интерфейса `Parent`.

Пример 4.15 ❖ Родительский класс, в котором переопределена реализация welcome по умолчанию

```
public class OverridingParent extends ParentImpl {

    @Override
    public void welcome() {
        message("Class Parent: Hi!");
    }
}
```

Пример 4.16 ❖ Предпочтение отдается конкретному методу, а не методу по умолчанию

```
@Test
public void concreteBeatsDefault() {
    Parent parent = new OverridingParent();
    parent.welcome();
    assertEquals("Class Parent: Hi!", parent.getLastMessage());
}
```

А в примере 4.18 показана ситуация, в которой переопределение метода по умолчанию в конкретном классе выглядит неожиданно. Класс `OverridingChild` наследует метод `welcome` как от `Child`, так и от `OverridingParent`, а сам ничего не делает. В примере 4.17 выбирается метод, унаследованный от `OverridingParent`, хотя тип `Child` более специфичен. Причина в том, что предпочтение отдается конкретному методу, а не методу по умолчанию (см. рис. 4.5).

Пример 4.17 ❖ Как и раньше, в дочернем интерфейсе метод по умолчанию переопределен

```
public class OverridingChild extends OverridingParent implements Child {

}
```

Пример 4.18 ❖ Предпочтение отдается конкретному методу, а не методу по умолчанию, определенному в более специфичном интерфейсе

```
@Test
public void concreteBeatsCloserDefault() {
    Child child = new OverridingChild();
    child.welcome();
    assertEquals("Class Parent: Hi!", child.getLastMessage());
}
```

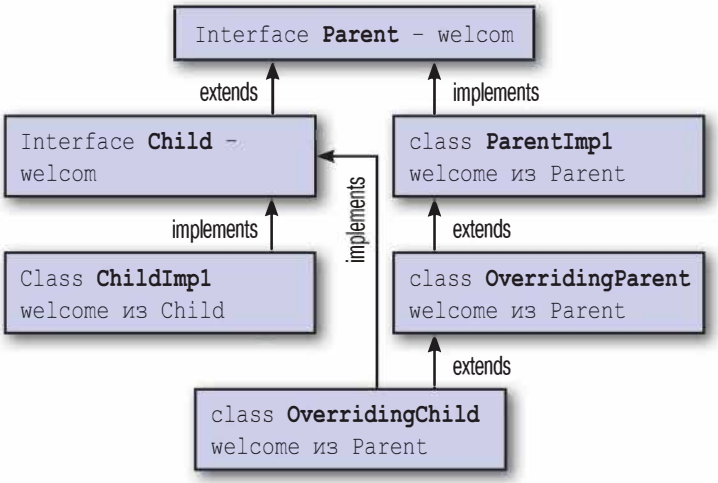

Рис. 4.5 ❖ Полная иерархия наследования

Короче говоря: *класс всегда выигрывает*. Такое решение принято потому, что методы по умолчанию предназначены прежде всего для обеспечения двоичной совместимости при эволюции API. Отдавая предпочтение классам, а не методам по умолчанию, удается избежать многих сложностей в различных сценариях наследования.

Допустим, что у нас была собственная реализация списка `MyCustomList`, в которой мы реализовали метод `addAll`, и что в новом интерфейсе `List` появился метод по умолчанию `addAll`, делегирующий работу методу `add`. Если бы не гарантировалось, что нашему методу `addAll` будет отдано предпочтение, то уже написанные программы могли бы перестать работать.

Множественное наследование

Поскольку разрешено множественное наследование интерфейсов, может возникнуть ситуация, когда в двух интерфейсах имеются методы по умолчанию с одной и той же сигнатурой. В примере ниже в интерфейсах `Carriage` и `Jukebox` имеется метод `rock`, но служит он совершенно разным целям. Имеется также класс `MusicalCarriage`, реализующий оба интерфейса `Jukebox` и `Carriage`, который пытается унаследовать метод `rock`.

Пример 4.19 ❖ Интерфейс Jukebox

```
public interface Jukebox {

    public default String rock() {
        return "... all over the world!";
    }
}
```

Пример 4.20 ❖ Интерфейс Carriage

```
public interface Carriage {

    public default String rock() {
        return "... from side to side";
    }
}

public class MusicalCarriage implements Carriage, Jukebox {
}
```

Поскольку `javac` не уверен, какой метод наследовать, он просто возвращает ошибку компиляции `class MusicalCarriage inherits unrelated defaults for rock() from types Carriage and Jukebox`. Разумеется, ситуацию можно разрешить, реализовав метод `rock`, как показано в примере 4.21.

Пример 4.21 ❖ Реализация метода rock

```
public class MusicalCarriage implements Carriage, Jukebox {

    @Override
    public String rock() {
        return Carriage.super.rock();
    }
}
```

В этом примере используется усовершенствованный синтаксис `super`, чтобы выбрать в качестве предпочтительной реализацию `rock` из интерфейса `Carriage`. В прежних версиях ключевое слово `super` обозначало ссылку на родительский класс, а новый вариант `InterfaceName.super` позволяет указать метод унаследованного интерфейса.

Три правила

Если вы не уверены, что случится при использовании методов по умолчанию или множественного наследования, руководствуйтесь следующими тремя простыми правилами разрешения конфликтов.

1. Классу всегда отдается предпочтение перед интерфейсом. Таким образом, если в цепочке родительских классов существует

метод, содержащий тело, или хотя бы его абстрактное объявление, об интерфейсах вообще можно забыть.

2. Подтипу отдается предпочтение перед супертипом. В ситуации, когда два интерфейса предоставляют один и тот же метод по умолчанию и один интерфейс расширяет другой, выигрывает расширенный интерфейс.
3. *Нет никакого правила 3.* Если предыдущие два правила не дают ответа, то подкласс должен либо реализовать метод, либо объявить его абстрактным.

Правило 1 – это то, что обеспечивает совместимость с ранее написанным кодом.

Компромиссы

Описанные изменения вызывают целый ряд вопросов относительно того, что же такое интерфейс в Java 8, если в нем можно определять методы, имеющие тело. Это означает, что теперь интерфейсы несут с собой некую форму множественного наследования, которое раньше вызывало порицание и изъятие которого из языка считалось важным преимуществом Java над C++.

Но никакой языковой механизм нельзя назвать абсолютным добром или абсолютным злом. Широко распространено мнение, что настоящая проблема связана с множественным наследованием состояния, а не просто кода. Но методы по умолчанию как раз запрещают наследование состояния, то есть обходят самые глубокие ямы, связанные с множественным наследованием в C++.

Возникает сильное искушение как-то обойти и эти ограничения. В блогах уже появляются статьи, в которых делаются попытки реализовать полноценные типажи (traits) с множественным наследованием состояния и методов по умолчанию. Но стремление хитростью обойти намеренно встроенные в Java 8 ограничения ведет прямиком к старым трудностям, свойственным C++.

Ясно также, что существует четкое различие между интерфейсами и абстрактными классами. Интерфейсы открывают возможность множественного наследования, но не имеют полей, тогда как абстрактные классы позволяют наследовать поля, но унаследовать сразу нескольким таким классам нельзя. Разрабатывая модель предметной области, необходимо помнить об этом компромиссе, что в прежних версиях Java было необязательно.

Статические методы в интерфейсах

Мы уже неоднократно встречали вызовы метода `Stream.of`, но в детали я пока не вдавался. Как вы помните, `Stream` – это интерфейс, следовательно, мы имеем дело со статическим методом интерфейса. Это еще одно языковое новшество, появившееся в Java 8, прежде всего чтобы помочь разработчикам библиотек. Впрочем, прикладным программистам оно тоже полезно.

Со временем сформировалась идиома создания классов, содержащих множество статических методов. Иногда класс – подходящее место для служебного кода. Примером может служить класс `Objects`, который был включен в Java 7 и содержит функциональные возможности, которые нельзя отнести к какому-то определенному классу.

Конечно, если метод семантически соотносится с какой-то концепцией, то следует помещать его в соответствующий класс или интерфейс, а не скрывать в служебном классе. Тем самым вы структурируете свой код таким образом, что читателю будет проще найти интересующий его метод.

Например, желая создать простой поток значений, вы, наверное, будете искать подходящий метод в интерфейсе `Stream`. Раньше это было невозможно, но добавление насквозь пронизанного интерфейсами API, конкретно `Stream`, все же стало достаточным основанием для включения в интерфейсы статических методов.

В интерфейсе `Stream` и его вариантах, специализированных для примитивов, есть и другие статические методы. Точнее, `range` и `iterate` предоставляют иные способы порождения потоков.

Тип Optional

До сих пор я обходил молчанием тот факт, что есть две формы `reduce`: одна – мы ее уже видели – принимает начальное значение, а другая не принимает. Если начальное значение опущено, то при первом обращении к редуктору используются первые два элемента потока. Это полезно, когда для операции `reduce` не существует разумного начального значения и возвращается экземпляр типа `Optional`.

`Optional` – это новый тип данных из базовой библиотеки, призванный предложить более удобную альтернативу `null`. Старое доброе значение `null` у многих вызывает ненависть. Даже его изобретатель Тони Хоар признался, что это была «ошибка на миллиард долларов».

Вот ведь как плохо быть авторитетным ученым – можно сделать ошибку на миллиард долларов, а самого миллиарда в глаза не видеть!

Часто null используют, чтобы представить отсутствие значения, и именно в этом случае Optional предпочтительнее. Проблема в том, что в случае, когда null обозначает отсутствие значения, возникает всеми проклинаемое исключение NullPointerException. Стоить обратиться к переменной, содержащей null, как программа «падает». У типа Optional цель двоякая. Во-первых, он поощряет программиста проверять, равна ли переменная null, во избежание ошибок. Во-вторых, он документирует, какие значения могут отсутствовать в API класса. В результате проще искать, где могут таиться мины.

Давайте взглянем на API класса Optional, чтобы понять, как его следует использовать. Чтобы создать экземпляр Optional по имеющемуся значению, нужно воспользоваться фабричным методом of. Отныне экземпляр Optional будет контейнером для этого значения, а достать его можно с помощью метода get, как показано на рис. 4.22.

Пример 4.22 ❖ Создание экземпляра Optional по имеющемуся значению

```
Optional<String> a = Optional.of("a");
assertEquals("a", a.get());
```

Поскольку Optional может также представлять отсутствующее значение, существует еще фабричный метод empty. А допускающее null значение можно преобразовать в Optional с помощью метода ofNullable. Оба способа показаны в примере 4.23 наряду с использованием метода isPresent (который сообщает, содержит ли данный экземпляр Optional какое-нибудь значение).

Пример 4.23 ❖ Создание пустого экземпляра Optional и проверка наличия значения

```
Optional emptyOptional = Optional.empty();
Optional alsoEmpty = Optional.ofNullable(null);

assertFalse(emptyOptional.isPresent());

// a определено выше
assertTrue(a.isPresent());
```

Один из способов использования Optional состоит в том, чтобы перед каждым вызовом get() проверять наличие значение с помощью isPresent(). Более разумный подход – вызывать метод orElse, который возвращает альтернативное значение, если экземпляр Optional

пуст. Если создание альтернативного значения сопряжено с большими накладными расходами, то лучше использовать метод `orElseGet`. Это позволяет передать объект `Supplier`, который вызывается лишь в том случае, когда экземпляр `Optional` действительно пуст. Оба варианта показаны в примере 4.24.

Пример 4.24 ❖ Использование методов orElse и orElseGet

```
assertEquals("b", emptyOptional.orElse("b"));
assertEquals("c", emptyOptional.orElseGet(() -> "c"));
```

`Optional` – обычный класс, который вы можете использовать в собственном коде, а не только совместно с новыми API, появившимися в Java 8. Его определенно стоит иметь в виду, когда вы пытаетесь избежать ошибок, связанных со значением `null`, в том числе неперехваченных исключений.

Основные моменты

- Добиться заметного повышения производительности позволяет использование типов лямбда-выражений и потоков, специализированных для примитивных типов, например `IntStream`.
- Методами по умолчанию называются методы интерфейсов, имеющие тела; их объявления начинаются ключевым словом `default`.
- Класс `Optional` позволяет избежать использования `null` для представления отсутствующего значения.

Упражнения

1. Пусть дан интерфейс `Performance`, показанный в примере 4.25. Добавьте в него метод `getAllMusicians`, который возвращает поток `Stream` исполнителей, принимавших участие в представлении, а в случае, когда исполнителем является группа, – еще и всех входящих в нее исполнителей. Так, если `getMusicians` возвращает *The Beatles*, то следует вернуть *The Beatles*, а также Lennon, McCartney и т. д.

   Пример 4.25 ❖ Интерфейс, описывающий концепцию музыкального представления

   ```
   /** Представление, в котором участвовали музыканты, - например, альбом
    или концерт */.
   public interface Performance {
   ```

```
    public String getName();

    public Stream<Artist> getMusicians();
}
```

2. Принимая во внимание описанные выше правила разрешения, можно ли переопределить `equals` или `hashCode` в методе по умолчанию?
3. Рассмотрим предметный класс `Artists` в примере 4.26, который представляет группу исполнителей. Ваша задача – переработать метод `getArtist`, так чтобы он возвращал объект `Optional<Artist>`. Этот объект должен содержать элемент, если индекс допустимый, и быть пустым в противном случае. Не забудьте, что придется также переработать метод `getArtistName`, сохранив при этом прежнее поведение.

Пример 4.26 ❖ Предметный класс Artists, представляющий несколько исполнителей

```
public class Artists {

    private List<Artist> artists;

    public Artists(List<Artist> artists) {
        this.artists = artists;
    }

    public Artist getArtist(int index) {
        if (index < 0 || index >= artists.size()) {
            indexException(index);
        }
        return artists.get(index);
    }

    private void indexException(int index) {
        throw new IllegalArgumentException(index +
                                " doesn't correspond to an Artist");
    }

    public String getArtistName(int index) {
        try {
            Artist artist = getArtist(index);
            return artist.getName();
        } catch (IllegalArgumentException e) {
            return "unknown";
        }
    }
}
```

Задача для исследования

1. Просмотрите весь код своего проекта или какого-нибудь знакомого вам проекта с открытым исходным кодом и попытайтесь найти классы, содержащие только статические методы, которые можно было бы сделать статическими методами интерфейсов. По возможности обсудите с коллегами, правы вы или нет.

Глава 5

Еще о коллекциях и коллекторах

В библиотеку коллекций внесено куда больше изменений, чем я упомянул в главе 3. Настало время рассмотреть некоторые из более продвинутых изменений, в том числе новую абстракцию `Collector`. Я также расскажу о ссылках на методы – способе использования существующего кода в лямбда-выражениях почти без подготовительных церемоний. Это оказывается чрезвычайно полезным при написании кода, плотно работающего с коллекциями. Будут освещены также другие изменения API, в том числе упорядочение элементов внутри потоков.

Ссылки на методы

Вы уже, наверное, обратили внимание на стандартную идиому – создание лямбда-выражения, которое вызывает метод от имени своего параметра. Так, желая, чтобы лямбда-выражение получало имя исполнителя, мы можем написать:

```
artist -> artist.getName()
```

Эта идиома употребляется настолько часто, что для нее имеется сокращенный синтаксис, который позволяет повторно использовать существующий метод. Называется он *ссылкой на метод*. Воспользовавшись ссылкой на метод, предыдущее лямбда-выражение можно переписать так:

```
Artist::getName
```

В общем случае ссылка на методы имеет вид `Classname::methodName`. Хотя это и метод, указывать скобки не следует, так как мы ничего не вызываем. Это просто эквивалент лямбда-выражения, вызов которого приведет к вызову метода. Ссылки на методы можно использовать всюду, где допустимы лямбда-выражения.

С помощью такого же сокращенного синтаксиса можно вызывать и конструкторы. Лямбда-выражение, создающее объект `Artist`, могло бы выглядеть так:

```
(name, nationality) -> new Artist(name, nationality)
```

То же самое можно выразить и с помощью ссылки на метод:

```
Artist::new
```

Этот код не только короче, но и читается гораздо легче. Запись `Artist::new` сразу же говорит читателю, что создается новый объект `Artist`; и не нужно изучать целую строку кода. Отметим также, что ссылки на методы автоматически поддерживают несколько параметров при условии, что имеется подходящий функциональный интерфейс.

Так можно создавать и массивы. Вот, например, как создается массив строк:

```
String[]::new
```

Начиная с этого момента, мы будем использовать ссылки на методы там, где это уместно, поэтому вскоре вы увидите еще много примеров такого рода. Когда мы только начинали исследовать изменения в Java 8, мой приятель заметил, что ссылки на метод «отдают мошенничеством». Он имел в виду, что после того как было приложено столько усилий для поддержки лямбда-выражений как механизма передачи кода по аналогии с данными, возможность ссылаться на методы напрямую выглядит каким-то обманом.

Успокойтесь – нет тут никакого обмана! Нужно просто помнить, что всякий раз, записывая лямбда-выражение вида `x -> foo(x)`, вы по существу делаете то же самое, что сам метод `foo`. Ссылки на методы – просто упрощенный синтаксис, в котором этот факт используется.

Упорядочение элементов

До сих пор я еще ни слова не сказал о том, как упорядочены элементы в потоках. Вы, наверное, знаете, что для одних типов коллекций, например `List`, порядок определен, а для других, например `HashSet`, – нет. В случае операций с потоками вопрос об упорядочении оказывается несколько более сложным.

Интуитивно представляется, что в потоке имеется определенный порядок, потому что операции производятся над каждым элементом по очереди. Такой порядок называется *порядком поступления*

(encounter order). Как именно определен порядок поступления, зависит от источника данных и от операций, выполняемых над потоком.

Если поток создается из коллекции, в которой определен порядок, то и порядок поступления в потоке тоже определен. Поэтому утверждение в примере 5.1 истинно.

Пример 5.1 ❖ Предположение об упорядоченности в этом утверждении истинно

```
List<Integer> numbers = asList(1, 2, 3, 4);

List<Integer> sameOrder = numbers.stream()
                                 .collect(toList());

assertEquals(numbers, sameOrder);
```

Если изначально порядок не был определен, то и в потоке, созданном на основе такого источника, нет определенного порядка. Примером неупорядоченной коллекции является `HashSet`, и утверждение в примере 5.2 может оказаться ложным.

Пример 5.2 ❖ Истинность предположения об упорядоченности не гарантирована

```
Set<Integer> numbers = new HashSet<>(asList(4, 3, 2, 1));

List<Integer> sameOrder = numbers.stream()
                                 .collect(toList());

// Это утверждение может оказаться ложным
assertEquals(asList(4, 3, 2, 1), sameOrder);
```

Цель потоков – не просто преобразовать одну коллекцию в другую; важно предоставить общий набор операций над данными. И эти операции могут создать порядок поступления там, где его изначально не было. Рассмотрим код в примере 5.3.

Пример 5.3 ❖ Создание порядка поступления

```
Set<Integer> numbers = new HashSet<>(asList(4, 3, 2, 1));

List<Integer> sameOrder = numbers.stream()
                                 .sorted()
                                 .collect(toList());

assertEquals(asList(1, 2, 3, 4), sameOrder);
```

Если порядок поступления существует, то он распространяется на последующие операции; например, если мы выполним операцию `map`

над потоком, в котором есть порядок поступления, то он будет сохранен. Если же во входном потоке нет порядка поступления, то его не будет и в выходном. Рассмотрим два фрагмента кода в примере 5.4. Для коллекции HashSet можно высказать только более слабые утверждения hasItem, потому что отсутствие определенного порядка поступления в HashSet сохраняется и после применения map.

Пример 5.4 ❖ Предположения об упорядочении

```
List<Integer> numbers = asList(1, 2, 3, 4);

List<Integer> stillOrdered = numbers.stream()
                                    .map(x -> x + 1)
                                    .collect(toList());

// Порядок поступления четко определен
assertEquals(asList(2, 3, 4, 5), stillOrdered);

Set<Integer> unordered = new HashSet<>(numbers);

List<Integer> stillUnordered = unordered.stream()
                                        .map(x -> x + 1)
                                        .collect(toList());

// Невозможно ничего утверждать о порядке поступления
assertThat(stillUnordered, hasItem(2));
assertThat(stillUnordered, hasItem(3));
assertThat(stillUnordered, hasItem(4));
assertThat(stillUnordered, hasItem(5));
```

Некоторые операции для упорядоченных потоков оказываются более накладными. Эту проблему можно решить, отказавшись от упорядочения. Для этого достаточно вызвать метод потока unordered. Однако большинство операций, в том числе filter, map и reduce, работают с упорядоченными потоками очень эффективно.

Возможно и неожиданное поведение. Например, forEach не дает никаких гарантий относительно порядка поступления при использовании параллельных потоков (эта тема подробнее обсуждается в главе 6). Если в таких ситуациях необходима гарантия безопасности, пользуйтесь методом forEachOrdered!

Знакомство с интерфейсом Collector

Мы уже использовали идиому collect(toList()) для порождения списков из потоков. Понятно, что List – самая естественная коллекция, порождаемая из Stream, но не всегда самая желательная. Быть может,

вам нужно создать Map или Set. А может быть, вообще имеет смысл завести предметный класс, абстрагирующий нужную вам концепцию?

Вы уже знаете, как по сигнатуре метода интерфейса Stream определить, соответствует ли он энергично вычисляемой финальной операции, порождающей значение. Для этой цели очень даже годится операция reduce. Но иногда хочется зайти дальше, чем позволяет reduce.

Позвольте представить вам *коллектор* – конструкцию общего характера для порождения составных значений из потоков. Коллектор можно использовать с произвольным потоком, передав его в качестве аргумента метода collect.

В стандартной библиотеке имеется ряд готовых полезных коллекторов, поэтому для начала познакомимся с ними. В примерах кода из этой главы коллекторы статически импортируются из класса java.util.stream.Collectors.

Порождение других коллекций

Некоторые коллекторы просто конструируют другие коллекции. Мы уже встречались с коллектором toList, который порождает экземпляры класса java.util.List. Есть также коллекторы toSet и toCollection, порождающие соответственно экземпляры Set и Collection. Я уже много говорил о сцеплении операций Stream, но все же бывают случаи, когда в качестве конечного значения требуется получить Collection. Например:

- при передаче коллекции в существующий код, рассчитанный на работу с коллекциями;
- при создании конечного значения в конце цепочки коллекций;
- при написании в тесте утверждения, относящегося к конкретной коллекции.

Обычно при создании коллекции мы указываем конкретный тип, вызывая соответствующий конструктор:

```
List<Artist> artists = new ArrayList<>();
```

Но при обращении к методу toList или toSet задавать конкретную реализацию List или Set не нужно. Потоковая библиотека сама выберет подходящую реализацию. Ниже в этой книге я расскажу о том, как можно использовать потоковую библиотеку для выполнения параллельных операций; для сбора результатов параллельных операций может потребоваться порождать другой тип Set, чтобы учесть требование потокобезопасности.

Иногда желательно, чтобы метод `collect` создавал коллекцию конкретного типа, если она понадобится в дальнейшем. Например, не исключено, что вам нужен объект класса `TreeSet`, а не того подкласса `Set`, который выбрала бы библиотека. Это можно сделать, воспользовавшись коллектором `toCollection`, который принимает в качестве аргумента функцию, конструирующую коллекцию (см. пример 5.5).

Пример 5.5 ❖ Создание конкретной коллекции с помощью метода toCollection

```
stream.collect(toCollection(TreeSet::new));
```

Порождение других значений

Коллекторы позволяют сворачивать коллекцию в одно значение. Так, коллекторы `maxBy` и `minBy` дают возможность получить одно значение в соответствии с некоторым отношением порядка. В примере 5.6 показано, как найти группу с наибольшим числом участников. Здесь задается лямбда-выражение, отображающее исполнителя на количество участников. Затем оно используется для определения компаратора, который передается в коллектор `maxBy`.

Пример 5.6 ❖ Нахождение группы с наибольшим числом участников

```
public Optional<Artist> biggestGroup(Stream<Artist> artists) {
    Function<Artist,Long> getCount = artist -> artist.getMembers().count();
    return artists.collect(maxBy(comparing(getCount)));
}
```

Коллектор `minBy` аналогично используется для определения минимума.

Существуют также коллекторы, реализующие стандартные операции над числами. Познакомимся с ними, написав компонент, который находит среднее число произведений в альбоме (пример 5.7).

Пример 5.7 ❖ Нахождение среднего числа произведений во всех альбомах из списка

```
public double averageNumberOfTracks(List<Album> albums) {
    return albums.stream()
                 .collect(averagingInt(album -> album.getTrackList().size()));
}
```

Как обычно, мы запускаем конвейер методом `stream`, а затем собираем результаты методом `collect`. После этого мы вызываем метод `averagingInt`, который принимает лямбда-выражение, преобразующее каждый элемент потока в число типа `int`, перед тем как усреднить результаты. Существуют также перегруженные операции для типов

double и long, позволяющие преобразовать элемент в число соответствующего типа.

В разделе «Примитивы» выше мы говорили, что у вариантов потоков, специализированных для примитивных типов, например IntStream, имеется дополнительная функциональность, применимая только к операциям над числами. Существует также и группа коллекторов, предлагающих похожие функциональные возможности, одним из них как раз и является averagingInt. Значения можно складывать с помощью метода summingInt и его вариантов для других типов. С помощью метода summarizingInt и ему подобных можно собирать сводную статистику SummaryStatistics.

Разбиение данных

Еще одна типичная операция интерфейса Stream – разбиение потока на две коллекции значений. Например, поток исполнителей можно разбить на солистов и группы. Один из подходов к решению этой задачи – произвести две операции фильтрации: сначала найти солистов, потом группы.

Но у такого подхода есть два недостатка. Во-первых, для выполнения двух операций потребуются два потока. Во-вторых, если перед фильтрами выполняется длинная последовательность других операций, то все эти операции придется выполнять дважды – в одном и в другом потоке. Такой код чистым не назовешь.

Поэтому существует коллектор partitioningBy, который принимает поток и разбивает его содержимое на две группы (рис. 5.1). Чтобы определить, куда какой элемент поместить, этот коллектор пользуется предикатом Predicate, а возвращает отображение Map, сопоставляю-

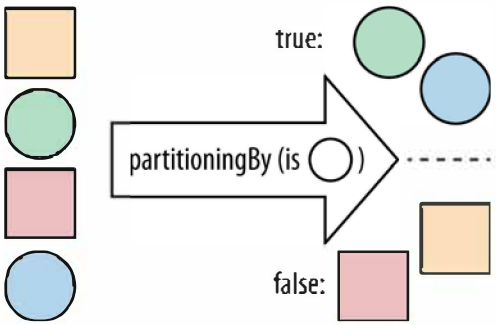

Рис. 5.1 ❖ Коллектор partitioningBy

щее булеву значению список List. Таким образом, в списке, соответствующем значению true, будут находиться элементы потока, для которых предикат вернул true, в списке, соответствующем значению false, – все остальные.

С помощью этого механизма мы можем отделить группы (исполнители с несколькими участниками) от солистов. В данном случае функция разбиения сообщает, является ли исполнитель солистом. Ее реализация показана в примере 5.8.

Пример 5.8 ❖ Разбиение потока исполнителей на группы и солистов

```
public Map<Boolean, List<Artist>> bandsAndSolo(Stream<Artist> artists) {
    return artists.collect(partitioningBy(artist -> artist.isSolo()));
}
```

То же самое можно выразить с помощью ссылки на метод, как показано в примере 5.9.

Пример 5.9 ❖ Разбиение потока исполнителей на группы и солистов с помощью ссылки на метод

```
public Map<Boolean, List<Artist>> bandsAndSoloRef(Stream<Artist> artists) {
    return artists.collect(partitioningBy(Artist::isSolo));
}
```

Группировка данных

Существует естественный способ обобщить разбиение, изменив операцию группировки. Большая общность заключается в том, что вместо двух групп, соответствующих значениям true и false, мы можем создать сколько угодно групп, каждой из которых соответствует некоторое общее значение. Предположим, что вы получили откуда-то поток альбомов и хотите сгруппировать их по имени основного музыканта. Это можно сделать, как показано в примере 5.10.

Пример 5.10 ❖ Группировка альбомов по основному исполнителю

```
public Map<Artist, List<Album>> albumsByArtist(Stream<Album> albums) {
    return albums.collect(groupingBy(album -> album.getMainMusician()));
}
```

Как и в других примерах, мы вызываем метод collect потока Stream, передавая ему объект Collector. Коллектор groupingBy (рис. 5.2) принимает функцию classifier, которая разбивает данные, – точно так же, как коллектор partitioningBy принимал предикат, относивший элементы потока к двум категориям: true и false. Классификатор имеет тип Function – такой же, как применяется в операции map.

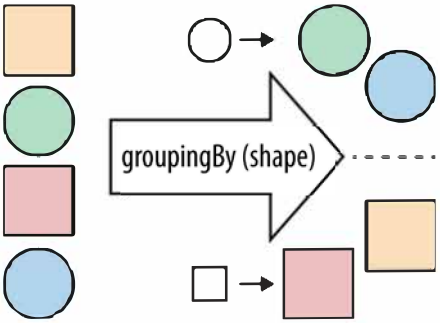

Рис. 5.2 ❖ Коллектор groupingBy

Читатели, работавшие с SQL, знакомы с конструкцией `group by`. Описываемая здесь концепция аналогична, но реализована в соответствии с идиомами потоковой библиотеки.

Строки

Очень часто сбор данных из потоков организуется для того, чтобы в конце сгенерировать строки. Предположим, что мы хотим создать отформатированный список имен исполнителей, участвовавших в записи альбома. Если, например, альбом называется «Let It Be», то на выходе мы ожидаем получить строку `"[George Harrison, John Lennon, Paul McCartney, Ringo Starr, The Beatles]"`.

До выхода Java 8 мы написали бы для решения этой задачи код вроде того, что приведен в примере 5.11. Здесь мы обходим список исполнителей и для построения строки пользуемся объектом `StringBuilder`. На каждой итерации извлекается имя исполнителя и добавляется в `StringBuilder`.

Пример 5.11 ❖ Форматирование имен исполнителей в цикле

```
StringBuilder builder = new StringBuilder("[");
for (Artist artist : artists) {
    if (builder.length() > 1)
        builder.append(", ");

    String name = artist.getName();
    builder.append(name);
}

builder.append("]");
String result = builder.toString();
```

Конечно, этот код шедевром не назовешь. Довольно трудно понять, что здесь происходит, не пройдя по всей программе шаг за шагом. В Java 8 с помощью потоков и коллекторов то же самое можно записать гораздо яснее (пример 5.12).

Пример 5.12 ❖ Форматирование имен исполнителей с помощью потоков и коллекторов

```
String result =
    artists.stream()
           .map(Artist::getName)
           .collect(Collectors.joining(", ", "[", "]"));
```

Здесь мы с помощью `map` извлекаем имена исполнителей, а затем собираем данные из потока с помощью коллектора `Collectors.joining`. Этот метод – удобное средство построения строк из данных, получаемых из потока. В нем можно задать разделитель (символ, вставляемый между элементами), а также начальный и конечный ограничитель.

Композиция коллекторов

Коллекторы, с которыми мы уже познакомились, сами по себе довольно мощные, но их мощь многократно возрастает благодаря возможности композиции.

Ранее мы группировали альбомы по основному исполнителю, а теперь займемся задачей подсчета количества альбомов, в которых участвовал каждый исполнитель. Для этого можно просто применить предыдущую группировку, а затем подсчитать число элементов в группе. Соответствующий код приведен в примере 5.13.

Пример 5.13 ❖ Наивный подход к подсчету количества альбомов для каждого исполнителя

```
Map<Artist, List<Album>> albumsByArtist
    = albums.collect(groupingBy(album -> album.getMainMusician()));

Map<Artist, Integer> numberOfAlbums = new HashMap<>();
for(Entry<Artist, List<Album>> entry : albumsByArtist.entrySet()) {
    numberOfAlbums.put(entry.getKey(), entry.getValue().size());
}
```

Гм, просто, конечно, но как-то неряшливо. Это императивный код, не допускающий автоматического распараллеливания.

А хотелось бы иметь еще один коллектор, который говорит `groupingBy`, что вместо построения списка альбомов для каждого исполнителя их нужно просто пересчитать. И надо же – в базовой биб-

лиотеке как раз такой коллектор есть, и называется он `counting`. Поэтому мы можем переписать этот код, как показано в примере 5.14.

Пример 5.14 ❖ Использование коллекторов для подсчета количества альбомов для каждого исполнителя

```
public Map<Artist, Long> numberOfAlbums(Stream<Album> albums) {
    return albums.collect(groupingBy(album -> album.getMainMusician(),
                          counting()));
}
```

Этот вариант `groupingBy` разбивает множество элементов на *кластеры*. Каждый кластер ассоциируется с ключом, который предоставляет функция классификации: `getMainMusician`. Затем операция `groupingBy` использует подчиненный коллектор для сбора данных из каждого кластера и создает отображение `Map`, содержащее результаты.

Рассмотрим еще один пример, в котором вместо группировки альбомов нам нужны только их названия. Можно было бы снова взять исходный коллектор, а затем подправить результирующие значения в `Map`. Как это сделать, показано в примере 5.15.

Пример 5.15 ❖ Наивный подход к нахождению названий всех альбомов, в записи который участвовал исполнитель

```
public Map<Artist, List<String>> nameOfAlbumsDumb(Stream<Album> albums) {
    Map<Artist, List<Album>> albumsByArtist =
        albums.collect(groupingBy(album ->album.getMainMusician()));

    Map<Artist, List<String>> nameOfAlbums = new HashMap<>();
    for(Entry<Artist, List<Album>> entry : albumsByArtist.entrySet()) {
        nameOfAlbums.put(entry.getKey(), entry.getValue()
                                          .stream()
                                          .map(Album::getName)
                                          .collect(toList()));
    }
    return nameOfAlbums;
}
```

Как и раньше, можно написать более элегантный, быстрый и допускающий распараллеливание код, воспользовавшись еще одним коллектором. Мы уже умеем группировать альбомы по основному исполнителю с помощью коллектора `groupingBy`, однако он порождает объект `Map<Artist, List<Album>>`. А нам нужно ассоциировать с каждым объектом `Artist` не список альбомов, а список строк, содержащих названия альбомов.

В данном случае мы хотели бы выполнить операцию `map` над списком, сопоставив каждому исполнителю название альбома. Но мы не

можем просто воспользоваться методом `map` потока, потому что нужный нам список создан коллектором `groupingBy`. Необходимо как-то сказать коллектору `groupingBy`, чтобы он применил `map` к значениям, помещаемым в результирующий список.

Любой коллектор – это рецепт построения конечного значения. Нам нужен рецепт получения другого рецепта – *еще одного* коллектора. К счастью, умные головы в Oracle предусмотрели такой случай и включили коллектор, который называется `mapping`.

Коллектор `mapping` позволяет выполнять операции типа `map` над контейнером другого коллектора. Еще ему нужно сказать, в какой коллекции сохранять результаты, это можно сделать с помощью коллектора `toList`. На всем пути сплошные коллекторы!

Как и операция `map`, коллектор получает на входе реализацию интерфейса `Function`. После рефакторинга с использованием второго коллектора получается код, показанный в примере 5.16.

Пример 5.16 ❖ Применение коллекторов для нахождения названий альбомов, в записи которых принимал участие исполнитель

```
public Map<Artist, List<String>> nameOfAlbums(Stream<Album> albums) {
    return albums.collect(groupingBy(Album::getMainMusician,
                          mapping(Album::getName, toList())));
}
```

В обоих случаях мы использовали второй коллектор для обработки части конечного результата. Такие коллекторы называются *подчиненными* (downstream). Если коллектор – это рецепт построения конечного значения, то подчиненный коллектор – рецепт построения части этого значения, которая затем используется главным коллектором. Возможность такой композиции коллекторов превращает их в еще более мощный компонент потоковой библиотеки.

Функции, специализированные для примитивных типов, например `averagingInt` или `summarizingLong`, на самом деле не дают ничего нового, по сравнению с вызовом метода самого специализированного потока. А истинное их предназначение – использоваться в качестве подчиненных коллекторов.

Рефакторинг и пользовательские коллекторы

Встроенные в Java коллекторы отлично подходят для выполнения типичных составных операций с потоками, но, вообще говоря, механизм коллекторов весьма общий. В тех, что входят в состав JDK, нет ничего сверхъестественного, написать собственный коллектор совсем нетрудно. Именно этим мы сейчас и займемся.

Вы, наверное, помните, что, рассматривая пример со строками, мы пришли к выводу, что запрограммировать его на Java 7 можно, хотя и не слишком изящно. Давайте постепенно переработаем этот код в коллектор, конкатенирующий строки. Использовать данный код в реальных программах нет нужды – в JDK имеется отлично работающий коллектор `joining`, однако это станет поучительным упражнением, в ходе которого мы узнаем, как устроены пользовательские коллекторы и как подходить к рефакторингу унаследованного кода при переходе на Java 8.

В примере 5.17 повторен код конкатенации строк, написанный для Java 7.

Пример 5.17 ❖ Использование цикла for и класса StringBuilder для форматирования списка имен исполнителей

```
StringBuilder builder = new StringBuilder("[");
for (Artist artist : artists) {
    if (builder.length() > 1)
        builder.append(", ");

    String name = artist.getName();
    builder.append(name);
}
builder.append("]");
String result = builder.toString();
```

Очевидно, что мы можем воспользоваться операцией `map` для преобразования потока исполнителей в поток имен (строк). В примере 5.18 показано, как выглядит этот код после перехода на потоки и `map`.

Пример 5.18 ❖ Использование forEach и класса StringBuilder для форматирования списка имен исполнителей

```
StringBuilder builder = new StringBuilder("[");
artists.stream()
       .map(Artist::getName)
       .forEach(name -> {
           if (builder.length() > 1)
               builder.append(", ");

           builder.append(name);
       });
builder.append("]");
String result = builder.toString();
```

Уже немного лучше, потому что, видя отображение на имена, мы быстрее понимаем, что же здесь строится. Но, к сожалению, остается

еще большой блок forEach, который не отвечает нашей цели: написать понятный с первого взгляда код путем композиции высокоуровневых операций.

Оставим ненадолго поставленную задачу создания пользовательского коллектора и подумаем, какие операции над потоками уже есть в нашем распоряжении. Ближе всего к построению нужной нам строки, пожалуй, операция reduce. Включив ее в пример 5.18, мы получим код, показанный в примере 5.19.

Пример 5.19 ❖ Использование reduce и класса StringBuilder для форматирования списка имен исполнителей

```
StringBuilder reduced =
    artists.stream()
           .map(Artist::getName)
           .reduce(new StringBuilder(), (builder, name) -> {
                 if (builder.length() > 0)
                     builder.append(", ");

                 builder.append(name);
                 return builder;
               }, (left, right) -> left.append(right));

reduced.insert(0, "[");
reduced.append("]");
String result = reduced.toString();
```

Я надеялся, что этот шаг рефакторинга сделает код яснее. Увы, похоже, он ничуть не лучше предыдущего. Но все же посмотрим, что здесь происходит. Вызовы stream и map остались такими же, как и раньше. Операция reduce строит строку имен исполнителей, разделяя их запятыми. Мы начинаем с пустого объекта StringBuilder – начального значения reduce. Следующее далее лямбда-выражение добавляет в построитель имя. Третий аргумент reduce – лямбда-выражение, которое принимает два экземпляра StringBuilder и объединяет их. Последний шаг – добавить ограничители в начало и в конец.

На следующем шаге рефакторинга я попытаюсь оставить редукцию, но убрать неразбериху, то есть абстрагировать детали, скрыв их в классе StringCombiner. В примере 5.20 показано, что получилось.

Пример 5.20 ❖ Использование reduce и класса StringCombiner для форматирования списка имен исполнителей

```
StringCombiner combined =
      artists.stream()
             .map(Artist::getName)
```

```
            .reduce(new StringCombiner(", ", "[", "]"),
                    StringCombiner::add,
                    StringCombiner::merge);

String result = combined.toString();
```

Хотя выглядит этот код совсем не так, как предыдущий, под капотом он делает ровно то же самое. Мы используем `reduce`, чтобы добавить имена и разделители в `StringBuilder`. Но теперь логика добавления элементов делегирована методу `StringCombiner.add`, а логика объединения двух разных комбинаторов – методу `StringCombiner.merge`. Посмотрим, как выглядит код этих методов. Начнем с метода `add` (пример 5.21).

Пример 5.21 ❖ Метод add объекта StringCombiner возвращает этот же объект, в который добавлен новый элемент

```
public StringCombiner add(String element) {
    if (areAtStart()) {
        builder.append(prefix);
    } else {
        builder.append(delim);
    }
    builder.append(element);
    return this;
}
```

Метод `add` делегирует содержательную работу экземпляру `StringBuilder`. Если данная операция комбинирования первая, то мы добавляем начальный ограничитель, иначе – разделитель между элементами. После этого добавляется сам элемент. Мы возвращаем сам объект `StringCombiner`, потому что он проталкивается по всей цепочке редукции. Показанный в примере 5.22 код объединения делегирует свою работу экземпляру `StringBuilder`.

Пример 5.22 ❖ Метод merge объекта StringCombiner объединяет результаты двух комбинаторов StringCombiner

```
public StringCombiner merge(StringCombiner other) {
    builder.append(other.builder);
    return this;
}
```

Мы почти закончили рефакторинг редукции, осталась всего одна мелочь. Хочется вставить в конец цепочки вызовов обращение к методу `toString`, чтобы весь код сводился к единственной цепочке. Для этого достаточно просто добавить вызов `toString` после `reduce`, подготовив код к применению API коллекторов.

Пример 5.23 ❖ Использование reduce и делегирование работы нашему классу StringCombiner

```
String result =
    artists.stream()
               .map(Artist::getName)
               .reduce(new StringCombiner(", ", "[", "]"),
                   StringCombiner::add,
                   StringCombiner::merge)
               .toString();
```

Теперь код выглядит более-менее пристойно, но его крайне трудно будет использовать повторно, если аналогичное комбинирование понадобится в другом месте. Поэтому мы заменим операцию `reduce` коллектором, который можно будет использовать где угодно. Я назову этот коллектор `StringCollector`. Результат очередного шага рефакторинга показан в примере 5.24.

Пример 5.24 ❖ Конкатенация строк с помощью класса StringCollector

```
String result =
    artists.stream()
           .map(Artist::getName)
           .collect(new StringCollector(", ", "[", "]"));
```

Полностью отдав задачу конкатенации строк на откуп специальному коллектору, наш код может ничего не знать о внутреннем устройстве `StringCollector`. Это просто еще один коллектор – такой же, как коллекторы, включенные в базовую библиотеку.

Начнем с реализации интерфейса `Collector` (пример 5.25). Этот интерфейс универсальный, поэтому нужно решить, какие типы будут его аргументами:

- тип собираемых коллектором элементов – `String`;
- тип аккумулятора – `StringCombiner`, мы его уже рассмотрели;
- тип результата – тоже `String`.

Пример 5.25 ❖ Как определяется коллектор строк

```
public class StringCollector implements
          Collector<String, StringCombiner, String> {
```

Коллектор состоит из четырех компонентов. Первый – поставщик `supplier`, то есть фабрика, изготавливающая контейнер, в данном случае `StringCombiner`. Аналогом может служить первый аргумент операции `reduce`, содержащий начальное значение (см. пример 5.26).

Пример 5.26 ❖ Метод supplier – фабрика по изготовлению контейнеров

```
public Supplier<StringCombiner> supplier() {
    return () -> new StringCombiner(delim, prefix, suffix);
}
```

Будем изображать исполнение кода в форме диаграмм, чтобы наглядно видеть, как все увязывается вместе. Поскольку коллекторы могут собирать данные параллельно, мы покажем операцию сборки, в которой два контейнерных объекта (например, `StringCombiner`) используются параллельно.

Все четыре компонента коллектора – функции, поэтому будем представлять их стрелками. Значения в потоке показаны кружочками, а конечное порождаемое значение – овалом. В начале операции сбора поставщик создает два новых контейнерных объекта (рис. 5.3).

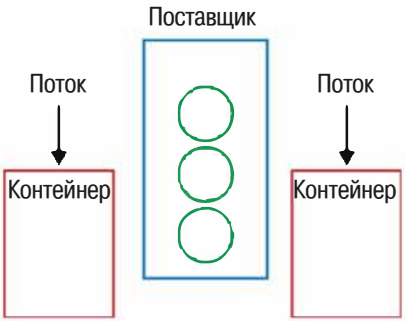

Рис. 5.3 ❖ Поставщик

Аккумулятор `accumulator` нашего коллектора выполняет ту же работу, что второй аргумент `reduce`. Он принимает текущий элемент и результат предыдущей операции, а возвращает новое значение. Эту логику мы уже реализовали в методе `add` класса `StringCombiner`, поэтому просто сошлемся на него (пример 5.27).

Пример 5.27 ❖ Метод accumulator – функция, добавляющая текущий элемент в коллектор

```
public BiConsumer<StringCombiner, String> accumulator() {
    return StringCombiner::add;
}
```

Наш аккумулятор добавляет значения, получаемые из потока, в объекты-контейнеры (рис. 5.4).

Метод `combine` – аналог третьего метода операции `reduce`. У нас должна быть возможность объединить два контейнера. Поскольку и это уже было сделано на предыдущем шаге рефакторинга, мы можем просто воспользоваться методом `StringCombiner.merge` (пример 5.28).

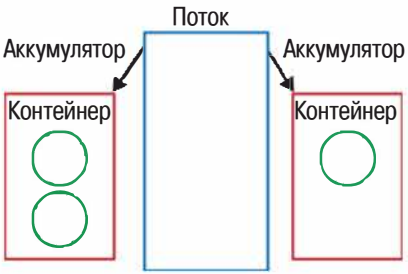

Рис. 5.4 ❖ Аккумулятор

Пример 5.28 ❖ Комбинатор объединяет два контейнера

```
public BinaryOperator<StringCombiner> combiner() {
    return StringCombiner::merge;
}
```

Во время операции сбора контейнерные объекты объединяются попарно с помощью определенного нами метода `combiner`, пока в конце не останется только один контейнер (рис. 5.5).

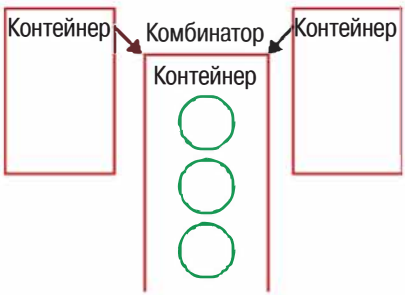

Рис. 5.5 ❖ Комбинатор

Вы, наверное, помните, что на последнем шаге рефакторинга – до того как мы перешли к коллекторам – мы поместили метод `toString` в конец цепочки вызовов. Тем самым мы преобразовали объект `StringCombiner` в строку, к чему, собственно, и стремились (рис. 5.6).

Метод коллектора `finisher` служит той же цели. Мы уже сложили поток значений в изменяемый контейнер, но это еще не конечное значение, которое нам нужно. Теперь один раз вызывается метод `finisher`, чтобы совершить окончательное преобразование. Особенно это полезно, если требуется создать неизменяемое конечное значение, например типа `String`, а контейнер является изменяемым.

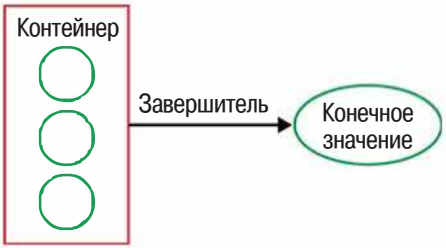

Рис. 5.6 ❖ Завершитель

Чтобы реализовать завершитель для этой операции, мы просто делегируем работу уже написанному методу `toString` (пример 5.29).

Пример 5.29 ❖ Метод finisher порождает конечное значение операции сбора

```
public Function<StringCombiner, String> finisher() {
    return StringCombiner::toString;
}
```

Мы создаем конечное значение из единственного оставшегося контейнера.

У коллекторов есть один аспект, который я пока не описал: *характеристики*. Характеристика – это множество `Set` объектов, описывающих коллектор, зная которые, библиотека может выполнить некоторые оптимизации. Определяется она с помощью метода `characteristics`.

Сейчас имеет смысл вспомнить, что этот код написан только в педагогических целях и немного отличается от внутренней реализации коллектора `joining`. Кроме того, класс `StringCombiner` тоже выглядит вполне полезным. Но не волнуйтесь – вам писать его не придется. В Java 8 уже имеется класс `java.util.StringJoiner`, выполняющий похожие действия и обладающий аналогичным API.

В этом упражнении я хотел не только показать, как работают пользовательские коллекторы, но и написать собственный коллектор. Это бывает полезно, когда имеется предметный класс, экземпляр которого строится в процессе некоторой операции над элементами коллекции, а ни один стандартный коллектор для такого построения не подходит.

В нашем коллекторе `StringCollector` контейнер для сбора значений отличался от конечного значения (объекта `String`). Так часто бывает, когда в результате сбора требуется построить неизменяемое значе-

ние, поскольку иначе на каждом шаге операции сбора пришлось бы создавать новое значение.

Но вполне возможно, что конечное желаемое значение совпадает с контейнером, в который собираются элементы коллекции. Именно так происходит, когда конечное значение само является коллекцией, как в случае коллектора `toList`.

В таком случае методу `finisher` ничего не нужно делать с контейнерным объектом. Точнее, можно сказать, что метод `finisher` – тождественная функция: он возвращает значение, полученное в качестве аргумента. Если это так, то коллектор обладает характеристикой `IDENTITY_FINISH`, о чем следует объявить с помощью метода `characteristics`.

Редукция как коллектор

Как вы только что видели, написать пользовательский коллектор нетрудно, но, подумывая о том, чтобы заняться этим ради построения предметного класса из коллекции, полезно рассмотреть альтернативы. Самое очевидное – построить один или несколько объектов коллекций и передать их конструктору предметного класса. Это действительно просто и годится, если предметный класс – составной объект, содержащий несколько коллекций.

Но, конечно, если предметный класс – не просто композиция коллекций, а должен выполнять какие-то вычисления с имеющимися данными, то такой способ – не выход. Впрочем, даже в этой ситуации без разработки собственного коллектора можно обойтись. Существует коллектор `reducing`, предоставляющий обобщенную реализацию операции редукции потоков. В примере 5.30 показано, как можно было бы записать рассмотренную выше обработку строк с помощью этого коллектора.

Пример 5.30 ❖ Редукция – удобный способ создания пользовательских коллекторов

```
String result =
          artists.stream()
                 .map(Artist::getName)
                 .collect(Collectors.reducing(
                       new StringCombiner(", ", "[", "]"),
                       name -> new StringCombiner(", ", "[", "]").add(name),
                       StringCombiner::merge))
                 .toString();
```

Это очень напоминает основанную на методе `reduce` реализацию, показанную в примере 5.20, что и неудивительно, если принять во

внимание название. Основное отличие – второй аргумент метода `Collectors.reducing`; мы создаем отдельный `StringCombiner` для каждого элемента в потоке. Если это вызывает у вас шок или отвращение, то вы совершенно правы! Такой способ чудовищно неэффективен, потому-то я и предпочитаю разрабатывать пользовательские коллекторы.

Усовершенствование интерфейса коллекций

Введение в язык лямбда-выражений дало возможность реализовать и новые методы коллекций. Познакомимся с некоторыми полезными изменениями в классе `Map`.

При построении любого объекта `Map` есть общее требование – вычислить значение для данного ключа. Классический пример – реализация кэша. Традиционная идиома – попытаться найти значение в `Map`, а если оно отсутствует, то создать.

Допустим, что кэш определен как `Map<String, Artist> artistCache`, а поиск исполнителей производится с помощью накладного обращения к базе данных. Тогда мы могли бы написать код, показанный в примере 5.31.

Пример 5.31 ❖ Кэширование значения с помощью явной проверки на null

```
public Artist getArtist(String name) {
    Artist artist = artistCache.get(name);
    if (artist == null) {
        artist = readArtistFromDB(name);
        artistCache.put(name, artist);
    }
    return artist;
}
```

В Java 8 появился новый метод `computeIfAbsent`, который принимает лямбда-выражение, вычисляющее новое значение, если оно отсутствует. Таким образом, этот пример можно переписать следующим образом.

Пример 5.32 ❖ Кэширование значения с помощью computeIfAbsent

```
public Artist getArtist(String name) {
    return artistCache.computeIfAbsent(name, this::readArtistFromDB);
}
```

Иногда бывает нужен вариант этого кода, который не выполняет вычисление, только если значение отсутствует; в таких случаях полезны новые методы `compute` и `computeIfPresent` в интерфейсе `Map`.

Бывает, что нужно обойти коллекцию `Map`. Исторически для этого использовался метод `values`, возвращающий коллекцию `Set` значений, которая и обходилась. Получающийся код читать было трудно. В примере 5.33 показано, как мы выше в этой главе создавали объект `Map`, сопоставляющий каждому исполнителю количество альбомов, в записи которых он принимал участие.

Пример 5.33 ❖ Некрасивый способ обхода всех элементов Map

```
Map<Artist, Integer> countOfAlbums = new HashMap<>();
for(Map.Entry<Artist, List<Album>> entry : albumsByArtist.entrySet()) {
    Artist artist = entry.getKey();
    List<Album> albums = entry.getValue();
    countOfAlbums.put(artist, albums.size());
}
```

По счастью, появился новый метод `forEach`, который принимает объект типа `BiConsumer` (два элемента на входе, ничего на выходе) и порождает понятный читателю код, основанный на внутреннем итерировании (см. выше раздел «От внешнего итерирования к внутреннему»). Эквивалентный код показан в примере 5.34.

Пример 5.34 ❖ Использование внутреннего итерирования для обхода всех элементов Map

```
Map<Artist, Integer> countOfAlbums = new HashMap<>();
albumsByArtist.forEach((artist, albums) -> {
    countOfAlbums.put(artist, albums.size());
});
```

Основные моменты

- Ссылки на методы – это упрощенная синтаксическая нотация следующего вида: `ClassName::methodName`.
- Коллекторы позволяют вычислять конечные результаты обработки потоков и являются изменяемым аналогом метода `reduce`.
- В Java 8 имеются готовые реализации коллекторов, порождающие различные типы коллекций, а также механизм для построения пользовательских коллекторов.

Упражнения

1. *Ссылки на методы.* Вернитесь к главе 3 и попробуйте переписать следующие примеры из нее с помощью ссылок на методы.
   a. Перевод в верхний регистр.
   b. Реализация `count` с помощью `reduce`.
   c. Конкатенация списков на основе `flatMap`.
2. *Коллекторы.*
   a. Найдите исполнителя с самым длинным именем. В решении воспользуйтесь коллектором и функцией высшего порядка `reduce`, описанной в главе 3. Сравните обе реализации: какую проще писать, а какую – читать? При следующих исходных данных должно быть возвращено имя «Stuart Sutcliffe»:

```
Stream<String> names = Stream.of("John Lennon", "Paul McCartney",
"George Harrison", "Ringo Starr", "Pete Best", "Stuart Sutcliffe");
```

   b. Пусть дан поток, элементы которого – слова. Посчитайте, сколько раз встречается каждое слово. При следующих исходных данных должен быть возвращен такой объект `Map`: `[John → 3, Paul → 2, George → 1]`:

```
Stream<String> names = Stream.of("John", "Paul", "George", "John",
"Paul", "John");
```

   c. Реализуйте метод `Collectors.groupingBy` в виде пользовательского коллектора. Предоставлять подчиненный коллектор необязательно, достаточно реализовать простейший вариант. Заглядывать в исходный код JDK нельзя! Подсказка: имеет смысл начать с `public class GroupingBy<T, K> implements Collector<T, Map<K, List<T>>, Map<K, List<T>>>`. Это трудное упражнение, поэтому займитесь им в последнюю очередь.
3. Усовершенствования Map.
   Найдите эффективный способ вычисления последовательности чисел Фибоначчи, в котором используется только метод `computeIfAbsent` интерфейса `Map`. Под «эффективностью» я понимаю, что ни одно число Фибоначчи не вычисляется дважды.

Глава 6

Параллелизм по данным

Выше я уже неоднократно отмечал, что в Java 8 писать параллельный код стало намного проще. Связано это с тем, что с помощью лямбда-выражений в сочетании с потоковой библиотекой (см. главу 3) мы можем сказать, что должна делать программа, не уточняя, как она это делает: последовательно или параллельно. Я знаю, что это очень походит на то, как вы писали код на Java в течение многих лет, но есть большая разница между описанием того, что требуется вычислить, и того, как это нужно вычислить.

Переход от внешнего итерирования к внутреннему (также описан в главе 3) стал важной вехой на пути к созданию простого и чистого кода, но у этой идеи есть и еще одно существенное достоинство: теперь нам не нужно управлять итерированием вручную. Итерирование необязательно выполняется последовательно. Мы говорим, что хотим сделать, а затем, изменив вызов всего одного метода, просим библиотеку самостоятельно определить, как добиться этого результата.

Вносимые в код изменения настолько малозаметны, что в большей части этой главы мы не будем говорить о том, как изменяется программа. Вместо этого я объясню, зачем может понадобиться распараллеливать код и как при этом повышается производительность. Отмечу также, что эта глава не является общим руководством по производительности в Java; мы лишь рассмотрим простые способы улучшить ее, которые дает Java 8.

Параллелизм и конкурентность

Просматривая оглавление книги, вы, наверное, обратили внимание, что в названии этой главы встречается слово «параллелизм», а в названии главы 9 – слово «конкурентность». Не подумайте, что я повто-

рил один и тот же материал, чтобы увеличить объем книги и заставить вас выложить побольше денег! Параллелизм и конкурентность – разные вещи, предназначенные для достижения разных целей.

Конкурентность означает, что выполнение двух задач перекрывается во времени. В случае параллелизма выполнение двух задач реально происходит в одно и то же время, как, например, на многоядерном процессоре. Если программа запускает две задачи, которые попеременно получают небольшие кванты времени на одноядерном процессоре, то имеет место конкурентность, но не параллелизм. Различие показано на рис. 6.1.

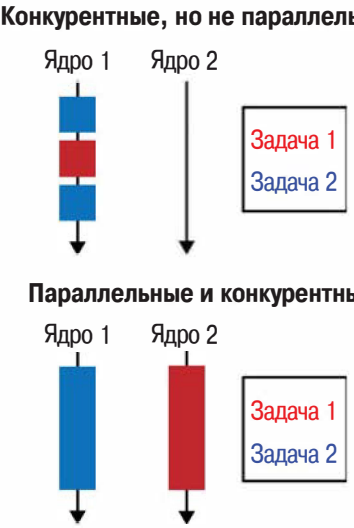

Рис. 6.1 ❖ Сравнение конкурентности и параллелизма

Цель параллелизма – уменьшить время работы конкретной задачи, разбив ее на меньшие части, которые работают параллельно. Это не означает, что общий объем работы уменьшается, по сравнению с последовательным выполнением той же задачи, – просто один и тот же воз тянут больше лошадей и справляются с этим быстрее. На самом деле обычно для распараллеливания задачи процессору приходится делать даже больше работы, чем при последовательном выполнении.

В этой главе мы рассмотрим один очень частный случай параллелизма – *параллелизм по данным*. Распараллеливание при этом дости-

гается за счет того, что рабочий набор данных разбивается на порции, и каждой порции назначается отдельный обрабатывающий блок. Продолжая аналогию с возом и лошадьми, можно сказать, что мы переложили половину груза на другой воз, в который запрягли другую лошадь, и обе лошади движутся к цели одним и тем же маршрутом.

Параллелизм по данным оказывается прекрасным решением, когда одну и ту же операцию требуется выполнить над различными данными. Задачу необходимо подвергнуть декомпозиции, так чтобы порции данных можно было обрабатывать одновременно, а в конце объединить частичные результаты, полученные для каждой порции.

Параллелизму по данным часто противопоставляют *параллелизм на уровне задач*, когда каждый поток выполнения решает свою задачу. Пожалуй, самым известным примером параллелизма на уровне задач является контейнер приложений в Java EE. Различные потоки могут не только работать над задачами разных пользователей, но и выполнять совершенно разные задачи, например один поток занимается аутентификацией пользователя, а другой добавляет товар в корзину.

Почему параллелизм важен?

Исторически мы привыкли полагаться на возрастание тактовой частоты процессоров. Тактовая частота процессора Intel 8086, выпущенного в 1979 году, составляла всего 5 МГц, а у процессора Pentium, появившегося в 1993 году, достигла уже 60 МГц. Поступательный рост тактовой частоты продолжался и в начале 2000-х годов.

Но в последнее десятилетие основные производители процессоров неуклонно двигаются в сторону разработки процессоров со все большим и большим числом ядер. Сейчас не редкость серверы с 32 или 64 ядрами, распределенными по нескольким центральным процессорам. И не похоже, что эта тенденция в обозримом будущем сойдет на нет.

Это оказывает влияние и на проектирование программного обеспечения. Мы больше не можем считать, что масштабировать программу удастся за счет перехода на более быстрый процессор, теперь приходится применять в своих интересах особенности архитектуры современных процессоров. И сделать это можно только путем написания параллельных программ.

Полагаю, вы уже знакомы со следующим положением. И немудрено – ведь оно уже много лет популяризуется многочисленными докладчиками на конференциях, авторами книг и консультантами.

Именно следствия *закона Амдала* и заставили меня всерьез отнестись к важности параллелизма.

Закон Амдала – это простое правило, позволяющее предсказать теоретически максимальное ускорение программы при запуске на машине с несколькими ядрами. Если взять строго последовательную программу и распараллелить только ее половину, то вне зависимости от количества ядер невозможно будет добиться ускорения более чем в 2 раза. Даже при большом числе ядер – а оно уже сейчас достаточно велико – время выполнения определяется последовательной частью программы.

Начав размышлять о производительности в таких терминах, мы быстро приходим к выводу, что для оптимизации любой счетной задачи, то есть связанной преимущественно с вычислениями, необходимо максимально эффективно использовать имеющееся оборудование. Разумеется, не все задачи счетные, но в этой главе мы будем иметь дело с таковыми.

Параллельные потоковые операции

Чтобы распараллелить операцию, реализованную с помощью потоковой библиотеки, достаточно изменить вызов одного метода. Если уже имеется объект `Stream`, то для превращения его в параллельный нужно вызвать его метод `parallel`. Если объект `Stream` создается из `Collection`, то для получения параллельного потока нужно создавать его с помощью метода `parallelStream`.

Для определенности рассмотрим простой пример. В примере 6.1 вычисляется полная длительность звучания последовательности альбомов. Каждый альбом преобразуется в набор составляющих его произведений, после чего длительности произведений суммируются.

Пример 6.1 ❖ Последовательное суммирование длительностей произведений

```
public int serialArraySum() {
    return albums.stream()
                 .flatMap(Album::getTracks)
                 .mapToInt(Track::getLength)
                 .sum();
}
```

Для распараллеливания мы вызываем метод `parallelStream`, как показано в примере 6.2; больше ничего в программе не изменяется. Распараллеливание *просто работает*.

Пример 6.2 ❖ Параллельное суммирование длительностей произведений

```
public int parallelArraySum() {
    return albums.parallelStream()
                 .flatMap(Album::getTracks)
                 .mapToInt(Track::getLength)
                 .sum();
}
```

Я понимаю, что после знакомства с этим кодом возникает желание немедленно заменить повсюду вызов `stream` на вызов `parallelStream` – ведь это так просто! Но попридержите коней! Понятно, что для выжимания максимума из оборудования важно с толком пользоваться параллелизмом, однако потоковая библиотека дает нам лишь параллелизм по данным.

Мы должны спросить себя, что быстрее: выполнить потоковый код последовательно или параллельно, а это отнюдь не простой вопрос. Предыдущая программа вычисления полной длительности звучания произведений из всех альбомов может работать быстрее в параллельном или последовательном варианте в зависимости от обстоятельств.

При замере времени работы примеров 6.1 и 6.2 на 4-ядерной машине при 10 альбомах последовательная версия оказывается в 8 раз быстрее. При 100 альбомах обе версии работают одинаково быстро, а при 10 000 альбомов параллельная версия опережает последовательную в 2,5 раза.

Все результаты измерений в этой главе приводятся только для сведения. На вашей машине они могут оказаться совершенно другими.

Размер входного потока – не единственный фактор, определяющий, даст ли распараллеливание ускорение. Результаты могут также зависеть от способа написания кода и количества доступных ядер. Мы еще вернемся к этой теме в разделе «Производительность» ниже, но сначала рассмотрим более сложный пример.

Моделирование

Параллельная потоковая библиотека особенно хороша для задач, в которых над большим количеством данных производятся простые операции, например для моделирования. В этом разделе мы построим простую модель бросания костей, но те же идеи и подходы применимы к более крупным практическим задачам.

Рассматриваемый здесь способ моделирования называется *методом Монте-Карло*. Его смысл заключается в многократном повторении одного и того же действия со случайными входными данными. Результаты отдельных прогонов сохраняются и агрегируются для получения полного решения. Метод Монте-Карло находит применение в научных и технических расчетах, в построении финансовых моделей.

Если бросить «честную» кость два раза и сложить очки, выпавшие на верхней грани, то получится число от 2 до 12. Минимум равен 2, потому что наименьшее количество очков на одной кости равно 1, а мы бросаем кость дважды. Максимум равен 12, потому что наибольшее количество очков на одной кости равно 6. Мы хотим для каждого числа от 2 до 12 вычислить, какова вероятность выпадения именно такой суммы очков.

Один из подходов к решению этой задачи – перечислить все комбинации очков, которые в сумме дают каждое значение. Например, получить 2 можно только одним способом: два раза подряд выкинуть 1. Всего существует 36 различных комбинаций, поэтому вероятность два раза выкинуть 1 равна 1/36.

Другой способ – смоделировать бросание двух костей с помощью случайных чисел от 1 до 6, затем сложить количество выпадений каждой суммы и разделить результат на количество бросаний. Это и есть простое применение метода Монте-Карло. Чем больше раз мы бросаем кость, тем точнее получается приближение, поэтому число бросаний должно быть велико.

В примере 6.3 показано, как можно реализовать метод Монте-Карло с помощью потоковой библиотеки. Здесь `N` – число прогонов модели, а функция `IntStream` в строке ❶ применяется для создания потока размера `N`. В строке ❷ мы вызываем метод `parallel`, чтобы использовать параллельную версию потоковой библиотеки. Функция `twoDiceThrows` моделирует два бросания кости и возвращает сумму выпавших очков. Метод `mapToObj` в строке ❸ служит для применения этой функции к потоку данных.

Пример 6.3 ❖ Параллельное моделирование бросания кости методом Монте-Карло

```
public Map<Integer, Double> parallelDiceRolls() {
    double fraction = 1.0 / N;
    return IntStream.range(0, N) ❶
                    .parallel()  ❷
                    .mapToObj(twoDiceThrows())   ❸
                    .collect(groupingBy(side -> side,  ❹
                    summingDouble(n -> fraction)));  ❺
}
```

В строке ❹ мы имеем поток Stream всех результатов моделирования, которые необходимо объединить. Чтобы агрегировать все одинаковые результаты, мы воспользовались коллектором groupingBy, представленным в предыдущей главе. Я говорил, что наша цель – подсчитать, сколько раз встретилось каждое число, и разделить эту величину на N. Но при работе с потоковой библиотекой проще отобразить каждое число на 1/N и просуммировать – результат получится тот же самый. Делается это в строке ❺ с помощью функции summingDouble. Возвращаемый в конце объект Map<Integer, Double> сопоставляет каждой сумме выпавших граней ее вероятность.

Вынужден признать, что этот код не вполне тривиален, но реализация параллельного моделирования методом Монте-Карло в пяти строчках – впечатляющее достижение. А поскольку приближение тем точнее, чем больше количество прогонов модели, то у нас есть стимул это количество увеличить. Данная реализация – удачный пример распараллеливания, потому что позволяет получить заметное ускорение.

Не стану вдаваться в детали реализации, но для сравнения в примере 6.4 показано, как выглядит тот же алгоритм параллельного моделирования методом Монте-Карло, написанный вручную. Большая часть кода связана с запуском, планированием и ожиданием завершения задач из пула потоков. Все эти проблемы остаются за кадром при использовании параллельной потоковой библиотеки.

Пример 6.4 ❖ Моделирование бросания костей с помощью явной реализации потоков

```
public class ManualDiceRolls {

    private static final int N = 100000000;

    private final double fraction;
    private final Map<Integer, Double> results;
    private final int numberOfThreads;
    private final ExecutorService executor;
    private final int workPerThread;

    public static void main(String[] args) {
        ManualDiceRolls roles = new ManualDiceRolls();
        roles.simulateDiceRoles();
    }

    public ManualDiceRolls() {
        fraction = 1.0 / N;
        results = new ConcurrentHashMap<>();
        numberOfThreads = Runtime.getRuntime().availableProcessors();
```

```
        executor = Executors.newFixedThreadPool(numberOfThreads);
        workPerThread = N / numberOfThreads;
    }

    public void simulateDiceRoles() {
        List<Future<?>> futures = submitJobs();
        awaitCompletion(futures);
        printResults();
    }

    private void printResults() {
        results.entrySet()
               .forEach(System.out::println);
    }

    private List<Future<?>> submitJobs() {
        List<Future<?>> futures = new ArrayList<>();
        for (int i = 0; i < numberOfThreads; i++) {
            futures.add(executor.submit(makeJob()));
        }
        return futures;
    }

    private Runnable makeJob() {
        return () -> {
            ThreadLocalRandom random = ThreadLocalRandom.current();
            for (int i = 0; i < workPerThread; i++) {
                int entry = twoDiceThrows(random);
                accumulateResult(entry);
            }
        };
    }

    private void accumulateResult(int entry) {
        results.compute(entry, (key, previous) ->
            previous == null ? fraction
                             : previous + fraction
        );
    }

    private int twoDiceThrows(ThreadLocalRandom random) {
        int firstThrow = random.nextInt(1, 7);
        int secondThrow = random.nextInt(1, 7);
        return firstThrow + secondThrow;
    }

    private void awaitCompletion(List<Future<?>> futures) {
        futures.forEach((future) -> {
            try {
                future.get();
```

```
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
            }
        });
        executor.shutdown();
    }
}
```

Подводные камни

Сказав выше, что параллельные потоки «просто работают», я немного слукавил. Существующий код можно выполнить параллельно после минимальной модификации, только если он написан идиоматично. Для оптимального использования параллельной потоковой библиотеки нужно соблюдать несколько правил и ограничений.

Ранее при вызове `reduce` начальный элемент мог быть произвольным значением, однако чтобы эта операция правильно работала в параллельном режиме, этот элемент должен быть *нейтральным* значением комбинирующей функции. Нейтральное значение обладает тем свойством, что при редукции с любым другим элементом оставляет его без изменения. Например, если мы с помощью операции `reduce` суммируем элементы, то комбинирующая функция имеет вид `(acc, element) -> acc + element`. Начальным элементом должен быть `0`, потому что сложение произвольного числа `x` с нулем дает `x`.

С операцией `reduce` связано еще одно ограничение: комбинирующая функция должна быть *ассоциативной*. Это означает, что результат не зависит от порядка применения комбинирующей функции, при условии что сама последовательность значений остается неизменной. Запутались? Не страшно! Взгляните на пример 6.5, где показано, что применение операций + и * к последовательности значений дает один и тот же результат вне зависимости от порядка.

Пример 6.5 ❖ Операции + и * ассоциативны

```
(4 + 2) + 1 = 4 + (2 + 1) = 7
(4 * 2) * 1 = 4 * (2 * 1) = 8
```

Следует избегать захвата блокировок. Потоковая библиотека сама занимается синхронизацией доступа, поэтому блокировать структуры данных нет необходимости. Попытка блокировки тех структур данных, с которыми работает библиотека, например исходной коллекции, скорее всего, приведет к неприятностям.

Выше я говорил, что существующий поток можно преобразовать в параллельный путем вызова метода `parallel`. Если по ходу чтения

книги вы заглядываете также в справочник по API, то, наверное, заметили, что есть еще метод `sequential`. Не существует смешанного режима вычисления потокового конвейера, он либо параллельный, либо последовательный. Если в конвейере встречаются вызовы обоих методов `parallel` и `sequential`, действует последний установленный режим.

Производительность

Я уже мимоходом отмечал, что есть много факторов, определяющих, какая версия потока – последовательная или параллельная – будет работать быстрее; рассмотрим их более подробно. Понимая, что хорошо, а что плохо, вы сможете принять обоснованное решение о том, как и когда использовать параллельные потоки. Существует пять важных факторов, от которых зависит производительность параллельных потоков.

- *Объем данных*. Величина ускорения при параллельной обработке зависит от объема входных данных. С декомпозицией задачи на исполняемые параллельно подзадачи и последующим объединением результатов сопряжены накладные расходы. Поэтому делать это имеет смысл лишь в том случае, когда данных достаточно, чтобы эти расходы амортизировались. Мы изучали эту проблему в разделе «Параллельные потоковые операции» выше.
- *Структура исходных данных*. Любой конвейер операций обрабатывает некоторый источник начальных данных, обычно коллекцию. Ускорение, достигаемое за счет распараллеливания, зависит от того, как источник разбит на несколько участков.
- *Упаковка*. Значения примитивных типов обрабатываются быстрее, чем упакованных.
- *Число ядер*. В крайнем случае, когда имеется всего одно ядро, распараллеливать операции не имеет смысла. Очевидно, что чем больше доступно ядер, тем больше потенциальный выигрыш от распараллеливания. На практике важно не общее число процессорных ядер в машине, а количество ядер, доступных для использования во время выполнения. Поэтому на производительность влияют такие вещи, как число одновременно исполняемых процессов и привязка потоков к ядрам (когда определенные потоки должны выполняться строго на определенных ядрах или процессорах).

- *Стоимость обработки элемента.* Как и объем данных, этот фактор является частью компромисса между выигрышем от распараллеливания и накладными расходами на декомпозицию и объединение. Чем больше времени тратится на обработку каждого элемента потока, тем выше потенциальный выигрыш от распараллеливания.

При использовании параллельной потоковой библиотеки полезно понимать, как производятся декомпозиция задачи и объединение результатов. Это позволит нам заглянуть под капот, не вникая в детали реализации.

Рассмотрим декомпозицию и объединение на конкретном примере. Код в примере 6.6 выполняет параллельное сложение целых чисел.

Пример 6.6 ❖ Параллельное сложение целых чисел

```
private int addIntegers(List<Integer> values) {
    return values.parallelStream()
                 .mapToInt(i -> i)
                 .sum();
}
```

Под капотом параллельные потоки реализованы с помощью разветвления и соединения. На этапе *разветвления* задача рекурсивно разбивается на части. Затем каждая часть обрабатывается параллельно. На стадии *соединения* частичные результаты объединяются.

На рис. 6.2 показано, как этот механизм мог бы быть применен к примеру 6.6.

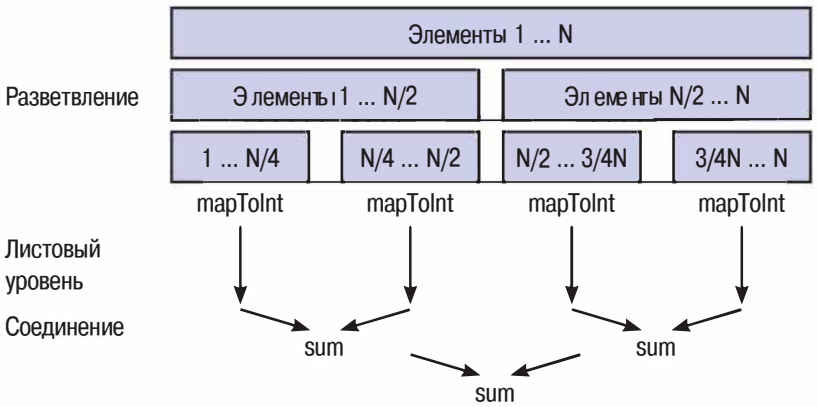

Рис. 6.2 ❖ Реализация декомпозиции и объединения с помощью разветвления и соединения

Предположим, что потоковая библиотека разбивает всю работу на части для параллельной обработки на четырехъядерной машине.

1. Производится декомпозиция источника данных на четыре порции.
2. В каждом потоке из примера 6.6 производятся вычисления над элементами в одном из листьев. Вычисление сводится к отображению `Integer` на `int` и суммированию четверти значений. В идеале хотелось бы как можно больше времени заниматься вычислениями на листовом уровне, потому что при этом выигрыш от распараллеливания максимален.
3. Результаты объединяются. В примере 6.6 для этого используется операция `sum`, но ее место могла бы занять `reduce`, `collect` или еще какая-то финальная операция.

С учетом того, как производится декомпозиция задачи, понятно, что природа источника исходных данных играет огромную роль. Интуитивно очевидно, что чем проще разбить данные на две половины, тем быстрее их можно будет обработать. Кроме того, желательно, чтобы в каждой половине данных было поровну.

Типичные источники данных из базовой библиотеки можно отнести к трем основным группам по характеристикам производительности.

- *Хорошие*. `ArrayList`, массив и конструктор `IntStream.range`. Все эти источники поддерживают произвольную выборку, поэтому их легко разбить на любые части.
- *Нормальные*. `HashSet` и `TreeSet`. Произвести их декомпозицию, сохранив сбалансированность, нелегко, но в большинстве случаев возможно.
- *Плохие*. Некоторые структуры данных с трудом поддаются разбиению, например на это может уйти время порядка *O(N)*. К таковым относится объект `LinkedList`, который вычислительно сложно разбить пополам. Длина объектов `Streams.iterate` и `BufferedReader.lines` изначально неизвестна, поэтому оценить, в каком месте производить разбиение, довольно трудно.

От структуры исходных данных может зависеть очень многое. Если взять крайний случай: 10 000 целых чисел, представленных в виде объектов `ArrayList` и `LinkedList`, – то в первом случае параллельное суммирование займет в 10 раз меньше времени, чем во втором. Это не значит, что для логики вашей программы характерны такие же показатели производительности, но влияние подобного рода факторов налицо. Кроме того, структуры типа `LinkedList`, плохо приспособленные

для декомпозиции, с гораздо большей вероятностью будут медленно работать при распараллеливании.

В идеале, после того как библиотека произвела декомпозицию задачи на меньшие подзадачи, каждая порция данных должна обрабатываться в отдельном потоке, и эти потоки не должны ни взаимодействовать между собой, ни конкурировать. Увы, реальность часто бывает далека от идеала!

Говоря об операциях в потоковом конвейере, применяемых к отдельным порциям данных, мы можем выделить два типа таковых: *с состоянием* и *без состояния*. У операций без состояния нет никаких данных, которые нужно было бы сохранять на протяжении всей операции; у операций с состоянием такие данные есть, и их сохранение влечет за собой определенные накладные расходы и ограничения.

Если удастся обойтись только операциями без состояния, то производительность параллельных вычислений будет выше. Примеры операций без состояния: `map`, `filter` и `flatMap`; у операций `sorted`, `distinct` и `limit` состояние есть.


Тестируйте производительность своей программы. Приведенные в этом разделе соображения подсказывают, на что следует обращать внимание, но ничто не заменит измерения и профилирование.

Параллельные операции с массивами

В Java 8 имеется еще несколько не входящих в потоковую библиотеку параллельных операций с массивами, в которых используются лямбда-выражения. Как и операции из потоковой библиотеки, они реализуют параллелизм по данным. Посмотрим, как с их помощью решаются задачи, которые трудно решить с применением потоков.

Все эти операции находятся в служебном классе `Arrays` наряду с прочей полезной функциональностью, относящейся к массивам и появившейся в предыдущих версиях Java. Они перечислены в табл. 6.1.

Таблица 6.1. Параллельные операции с массивами

Имя	Операция
`parallelPrefix`	Вычисляет накопительный итог элементов массива с помощью произвольной функции
`parallelSetAll`	Обновляет элементы массива с помощью лямбда-выражения
`parallelSort`	Параллельно сортирует элементы

Возможно, вам доводилось раньше писать код наподобие приведенного в примере 6.7, где массив инициализируется в цикле for. В данном случае мы записываем в каждый элемент массива его индекс.

Пример 6.7 ❖ Инициализация массива в цикле for

```
public static double[] imperativeInitilize(int size) {
    double[] values = new double[size];
    for(int i = 0; i < values.length;i++) {
        values[i] = i;
    }
    return values;
}
```

Эту задачу легко распараллелить, воспользовавшись методом parallelSetAll. Как это делается, показано в примере 6.8. Мы подаем на вход инициализируемый массив и лямбда-выражение, которое вычисляет значение элемента по его индексу. В нашем случае то и другое совпадает. Следует отметить, что все указанные методы модифицируют сам переданный массив, а не создают новую копию.

Пример 6.8 ❖ Инициализация массива с помощью параллельной операции

```
public static double[] parallelInitialize(int size) {
    double[] values = new double[size];
    Arrays.parallelSetAll(values, i -> i);
    return values;
}
```

Операция parallelPrefix гораздо полезнее при вычислении частичных итогов последовательностей данных. Она изменяет массив, заменяя каждый элемент *суммой* этого элемента и предшествующих ему. Слово *сумма* не нужно понимать буквально – это необязательно операция сложения, годится любой BinaryOperator.

Пример операции, вычисляемой таким образом, – простое скользящее среднее. В этом случае на временной ряд накладывается скользящее окно и вычисляется среднее арифметическое элементов, оказавшихся внутри окна. Так, если дан ряд 0, 1, 2, 3, 4, 3.5, то в результате вычисления простого скользящего среднего размера 3 получается ряд 1, 2, 3, 3.5. В примере 6.9 показано, как такое скользящее среднее вычисляется с помощью префиксного суммирования.

Пример 6.9 ❖ Вычисление простого скользящего среднего

```
public static double[] simpleMovingAverage(double[] values, int n) {
    double[] sums = Arrays.copyOf(values, values.length);   ❶
    Arrays.parallelPrefix(sums, Double::sum);   ❷
```

```
    int start = n - 1;
    return IntStream.range(start, sums.length)    ❸
                    .mapToDouble(i -> {
                       double prefix = i == start ? 0 : sums[i - n];
                       return (sums[i] - prefix) / n;    ❹
                    })
                    .toArray();  ❺
}
```

Код довольно сложный, поэтому рассмотрим его по частям. Параметр `n` задает размер сдвигающегося окна, по которому мы вычисляем скользящее среднее. В точке ❶ мы создаем копию входных данных. Поскольку префиксное суммирование – модифицирующая операция, то это необходимо сделать, чтобы не изменять исходных данных.

В точке ❷ мы применяем префиксную операцию, чтобы вычислить сумму значений. Теперь в массиве `sums` хранятся частичные суммы. Например, если на вход был подан массив `0`, `1`, `2`, `3`, `4`, `3.5`, то в `sums` окажутся значения `0.0`, `1.0`, `3.0`, `6.0`, `10.0`, `13.5`.

Имея частичные суммы, мы можем найти сумму по временному окну, для чего нужно вычесть частичную сумму в начале этого окна. Для получения среднего нужно разделить результат на `n`. Это вычисление можно проделать с помощью потоковой библиотеки, так не упустим этот шанс! Необходимый нам поток, содержащий индексы нужных нам элементов, создается посредством вызова метода `Intstream.range`.

В точке ❹ мы вычитаем величину частичной суммы в начале окна и выполняем деление, чтобы получить среднее. Отметим граничный случай элемента с индексом `n - 1`, для которого нет никакой частичной суммы, которую можно было бы вычесть. Наконец, мы преобразуем `Stream` снова в массив.

Основные моменты

- Параллелизм по данным подразумевает разбиение всей работы на части, выполняемые одновременно на нескольких ядрах.
- Если при написании программы используется потоковая библиотека, то для реализации параллелизма по данным следует вызвать один из методов `parallel` или `parallelStream`.
- Производительность определяется пятью основными факторами: объем данных, структура исходных данных, являются ли данные упакованными или примитивными, число доступных ядер и время, затрачиваемое на обработку одного элемента.

Упражнения

1. В примере 6.10 последовательно суммируются квадраты элементов потока. Преобразуйте эту программу в параллельную с помощью потоковой библиотеки.

   Пример 6.10 ❖ Последовательное суммирование квадратов элементов списка

   ```
   public static int sequentialSumOfSquares(IntStream range) {
         return range.map(x -> x * x)
                     .sum();
   }
   ```

2. В примере 6.11 все элементы списка перемножаются, а результат умножается на 5. Последовательная версия этого кода работает правильно, но при распараллеливании появляется ошибка. Преобразуйте эту программу в параллельную с помощью потоковой библиотеки и исправьте ошибку.

   Пример 6.11 ❖ Неправильный способ перемножения всех элементов списка и умножения результата на 5

   ```
   public static int multiplyThrough(List<Integer> linkedListOfNumbers) {
         return linkedListOfNumbers.stream()
                     .reduce(5, (acc, x) -> x * acc);
   }
   ```

3. В примере 6.12 также вычисляется сумма квадратов элементов списка. Попробуйте улучшить производительность этой программы, не жертвуя ее качеством. Я хотел бы, чтобы вы внесли всего два простых изменения.

   Пример 6.12 ❖ Медленная реализация суммирования квадратов элементов списка

   ```
   public int slowSumOfSquares() {
         return linkedListOfNumbers.parallelStream()
                                   .map(x -> x * x)
                                   .reduce(0, (acc, x) -> acc + x);
   }
   ```

Не забывайте, что для повышения точности хронометража исследуемый код должен прогоняться несколько раз. В примерах кода на сайте GitHub имеется оснастка для эталонного тестирования, которой вы можете пользоваться.

Глава 7

Тестирование, отладка и рефакторинг

Рост популярности таких методов, как рефакторинг, разработка через тестирование (TDD) и непрерывная интеграция (CI), означает, что если мы намерены использовать лямбда-выражения в повседневном программировании, то нужно понимать, как тестировать содержащий их код.

В материалах по тестированию и отладке программ нет недостатка, и я в этой главе не собираюсь повторять их. Если вам интересно узнать, как правильно применять TDD на практике, горячо рекомендую книги Kent Beck «Test-Driven Development»[1] и Steve Freeman, Nat Pryce «Growing Object-Oriented Software, Guided by Tests».

Я рассмотрю только приемы, относящиеся к использованию в программе лямбда-выражений, и расскажу, когда лучше воздержаться от их (непосредственного) использования. Мы поговорим также о некоторых специальных методах отладки программ, в которых интенсивно применяются лямбда-выражения и потоки.

Но сначала рассмотрим несколько примеров рефакторинга существующего кода с целью использования лямбда-выражений. Я уже демонстрировал некоторые локальные операции рефакторинга, например замену цикла `for` потоковой операцией. Теперь мы более пристально изучим способы улучшения кода, не связанного с коллекциями.

Когда разумно перерабатывать код с использованием лямбда-выражений

Процесс рефакторинга кода с целью использования лямбда-выражений получил неблагозвучное название «точечная лямбдификация»

---

[1] Кент Бек. Экстремальное программирование: разработка через тестирование. – СПб.: Питер, 2003.

(а практикующие его программисты называются «лямбдификаторами», или «ответственными разработчиками»). Именно это было проделано с базовыми библиотеками Java при выпуске версии Java 8. Принимая решение о внутренней структуре приложения, всегда стоит подумать о том, какие методы его API раскрывать подобным образом.

Размышляя над тем, какое место приложения или библиотеки подвергнуть лямбдификации, можно руководствоваться несколькими эвристическими соображениями. Такое место можно рассматривать как проявление какого-то антипаттерна или «запашок» в коде, которые точечная лямбдификация призвана устранить.

Инкапсуляция внутреннего состояния

В примере 7.1 повторен код протоколирования из главы 4. Как видите, булево значение `isDebugEnabled` нужно только для того, чтобы проверить его и в зависимости от результата вызвать метод объекта `Logger`. Обнаружив, что программа в нескольких местах опрашивает некоторое свойство объекта, чтобы решить, нужно ли вызывать его метод, мы можем включить соответствующий код в сам класс объекта.

Пример 7.1 ❖ Использование свойства регистратора isDebugEnabled, чтобы избежать лишних накладных расходов

```
Logger logger = new Logger();
if (logger.isDebugEnabled()) {
    logger.debug("Look at this: " + expensiveOperation());
}
```

Протоколирование – типичный пример такой ситуации, в которой поставленная цель исторически была трудно достижима, потому что в разных местах требовалось разное поведение. В данном случае поведение заключается в построении строки сообщения, которая зависит от того, где происходит протоколирование и какая информация записывается в журнал.

Справиться с этим антипаттерном легко – достаточно передать код как данные. Вместо того чтобы опрашивать свойство объекта, а затем вызывать его метод, мы можем передать лямбда-выражение, которое представляет требуемое поведение, путем вычисления значения. Я повторно привожу такое решение в примере 7.2 – в качестве напоминания. Это лямбда-выражение вызывается, только если установлен уровень протоколирования не выше отладочного, при этом логика проверки остается внутри объекта `Logger`.

Пример 7.2 ❖ Упрощение кода протоколирования с помощью лямбда-выражения

```
Logger logger = new Logger();
logger.debug(() -> "Look at this: " + expensiveOperation());
```

Протоколирование – это также пример использования лямбда-выражений для повышения объектной ориентированности кода. Одна из ключевых концепций ООП – инкапсуляция внутреннего состояния, каковым, в частности, является уровень протоколирования. Обычно оно инкапсулировано не слишком хорошо, потому что раскрывается через свойство `isDebugEnabled`. Но если воспользоваться подходом на основе лямбда-выражения, то коду за пределами класса `Logger` проверять уровень протоколирования вообще не нужно.

Переопределение единственного метода

Этот «запашок» ощущается, когда вы создаете подкласс ради переопределения только одного метода. Хорошим примером может служить класс `ThreadLocal`, который позволяет создать фабрику, генерирующую поточно-локальное значение. Это простой способ гарантировать безопасное использование небезопасного относительно потоков класса в конкурентном окружении. Например, если требуется найти исполнителя в базе данных, но делать это однократно в каждом потоке, то можно написать код наподобие показанного в примере 7.3.

Пример 7.3 ❖ Поиск исполнителя в базе данных

```
ThreadLocal<Album> thisAlbum = new ThreadLocal<Album> () {
    @Override protected Album initialValue() {
        return database.lookupCurrentAlbum();
    }
};
```

В Java 8 мы можем воспользоваться фабричным методом `withInitial`, передав ему экземпляр класса `Supplier`, который занимается созданием, как показано в примере 7.4.

Пример 7.4 ❖ Применение фабричного метода

```
ThreadLocal<Album> thisAlbum
    = ThreadLocal.withInitial(() -> database.lookupCurrentAlbum());
```

Второй вариант предпочтительнее первого по нескольким причинам. Во-первых, здесь можно взять любой существующий экземпляр `Supplier<Album>` без специального конфигурирования для данного случая, что облегчает повторное использование и композицию.

Кроме того, конструкция получается более короткой, что при прочих равных условиях само по себе является преимуществом. Важнее, однако, что краткость – прямое следствие чистоты кода: отношение сигнал/шум в этом коде выше. А это означает, что на решение конкретной задачи вы потратите меньше времени и не придется возиться со стереотипным кодом создания подкласса. Да и виртуальная машина Java должна будет загружать на один класс меньше.

Наконец, читателю этого кода гораздо проще понять, в чем его смысл. Попробуйте произнести вслух слова из второго примера – получится связная фраза. О первом примере такого определенно не скажешь.

Интересно, что до Java 8 подобный способ выражения считался не антипаттерном, а идиоматической записью кода, точно так же, как не считалось антипаттерном создавать анонимные внутренние классы для передачи поведения, – это был единственный способ выразить свое намерение на Java. Но по мере развития языка меняются и применяемые идиомы.

Поведенческий паттерн «пиши все дважды»

Принцип «пиши все дважды» (Write Everything Twice – WET) – противоположность хорошо известному принципу «не повторяйся» (Don't Repeat Yourself – DRY). Этот «запашок» проникает в программу в ситуациях, когда вследствие повторяющихся стереотипных конструкций образуется больше кода – и этот лишний код приходится тестировать, его труднее подвергнуть рефакторингу, и любое изменение чревато «поломкой».

Не все проявления WET – подходящие кандидаты для точечной лямбдификации. Иногда дублирование – единственная альтернатива созданию слишком сильно связанной системы. Но существует простой эвристический способ выявить ситуации, в которых наличие WET настоятельно требует прибегнуть к точечной лямбдификации. Попробуйте добавить лямбда-выражения в те места, где может понадобиться в целом схожее, но различающееся в деталях поведение.

Рассмотрим конкретный пример. Я решил добавить поверх нашей музыкальной предметной области простой класс `Order`, вычисляющий полезные свойства альбомов, которые пожелал купить пользователь. Мы будем подсчитывать суммарное количество музыкантов, произведений и общую длительность звучания по всем вошедшим в заказ альбомам. Императивный код на Java мог бы выглядеть, как показано в примере 7.5.

Пример 7.5 ❖ Императивная реализация класса Order

```
public long countRunningTime() {
    long count = 0;
    for (Album album : albums) {
        for (Track track : album.getTrackList()) {
            count += track.getLength();
        }
    }
    return count;
}

public long countMusicians() {
    long count = 0;
    for (Album album : albums) {
        count += album.getMusicianList().size();
    }
    return count;
}

public long countTracks() {
    long count = 0;
    for (Album album : albums) {
        count += album.getTrackList().size();
    }
    return count;
}
```

Во всех случаях налицо стереотипный код обхода альбомов и прибавления какого-то значения к итоговой сумме – например, продолжительности звучания произведения или количества музыкантов. Мы упускаем возможность повторного использования общих концепций и оставляем больше кода, который нам же придется тестировать и сопровождать. Объем кода можно сократить, воспользовавшись абстракцией потока и библиотекой коллекций в Java 8. В примере 7.6 показано, что получится, если в лоб преобразовать императивный код в потоковый.

Пример 7.6 ❖ Рефакторинг императивного класса Order с использованием потоков

```
public long countRunningTime() {
    return albums.stream()
                 .mapToLong(album -> album.getTracks()
                                          .mapToLong(track -> track.getLength())
                                          .sum())
                 .sum();
}

public long countMusicians() {
```

```
    return albums.stream()
                 .mapToLong(album -> album.getMusicians().count())
                 .sum();
}

public long countTracks() {
    return albums.stream()
                 .mapToLong(album -> album.getTracks().count())
                 .sum();
}
```

Но и этот код страдает теми же проблемами, затрудняющими повторное использование и ухудшающими удобочитаемость, поскольку некоторые абстракции и общие конструкции выразимы только в терминах предметной области. Потоковая библиотека не предоставляет метода, который мог бы подсчитать количество чего-то по множеству альбомов – метод получения «чего-то» относится к предметной области, и мы должны написать его самостоятельно. Причем до выхода Java 8 писать такие методы было трудно, потому что каждый из них делал что-то свое.

Прикинем, как мы стали бы реализовывать подобную функцию. Нам нужно вернуть число типа `long`, равное количеству некоего свойства по всем альбомам. Необходимо также принять лямбда-выражение, которое говорит, как вычислять это свойство. Следовательно, нужен параметр-метод, который возвращает `long` для каждого альбома; по счастливому стечению обстоятельств, в базовых библиотеках Java 8 уже есть метод `ToLongFunction`. Как показано на рис. 7.1, он параметризован типом своего аргумента, так что нас интересует метод `ToLongFunction<Album>`.

Рис. 7.1 ❖ ToLongFunction

После того как решения приняты, написать тело метода уже просто. Мы берем поток альбомов, сопоставляем каждому альбому значение типа `long` и суммируем эти значения. При реализации видимых потребителю методов, например `countTracks`, мы передаем лямбда-выражение, определяющее поведение, специфичное для предметной области. В примере 7.7 показано, на что стал похож код после такого выделения метода подсчета.

Пример 7.7 ❖ Рефакторинг класса Order с выделением методов, специфичных для предметной области

```
public long countFeature(ToLongFunction<Album> function) {
    return albums.stream()
                 .mapToLong(function)
                 .sum();
}

public long countTracks() {
    return countFeature(album -> album.getTracks().count());
}

public long countRunningTime() {
    return countFeature(album -> album.getTracks()
                                      .mapToLong(track -> track.getLength())
                                      .sum());
}

public long countMusicians() {
    return countFeature(album -> album.getMusicians().count());
}
```

Автономное тестирование лямбда-выражений

Автономным[1] тестированием называется методика тестирования отдельных кусков кода с целью удостовериться, что они ведут себя, как задумано.

Обычно из автономного теста вызывается метод, который вызывает также приложение. Подавая на вход определенные данные и, возможно, применяя тестовые двойники, мы хотим удостовериться, что этот метод демонстрирует определенное поведение, и с этой целью описываем, какие изменения должны произойти в результате его вызова.

С точки зрения автономного тестирования кода, с лямбда-выражениями связана одна проблема. Поскольку лямбда-выражение не имеет имени, его невозможно вызвать из тестового кода напрямую.

Можно было бы скопировать тело лямбда-выражения в тестовый код и таким образом протестировать копию, но такой подход страдает тем недостатком, что тестируется не поведение самой реализации.

---

[1] В русскоязычной литературе распространен также термин «модульное тестирование», но, на мой взгляд, он не вполне уместен по различным причинам, в том числе из-за перегруженности слова «модуль». – *Прим. перев.*

Если изменить код реализации, то тест по-прежнему будет проходить, хотя реализация уже ведет себя по-другому.

Есть два заслуживающих внимания решения этой проблемы. Первое – рассматривать лямбда-выражение как блок кода внутри объемлющего его метода. Если пойти по этому пути, то мы должны будем тестировать поведение объемлющего метода, а не самого лямбда-выражения. В примере 7.8 показан метод, преобразующий список строк в список тех же строк, но записанных в верхнем регистре.

Пример 7.8 ❖ Преобразование строк в верхний регистр

```
public static List<String> allToUpperCase(List<String> words) {
    return words.stream()
                .map(string -> string.toUpperCase())
                .collect(Collectors.<String>toList());
}
```

Единственное, что делает лямбда-выражение в теле этого метода, – прямой вызов метода из базовой библиотеки. В силу простоты поведения тестировать это лямбда-выражение как независимую единицу кода вообще не имеет смысла.

Если бы мне нужно было автономно протестировать этот код, я сосредоточился бы на поведении всего метода. Так, в примере 7.9 приведен тест, проверяющий, что если в потоке есть несколько слов, то все они преобразуются в верхний регистр.

Пример 7.9 ❖ Тестирование преобразования слов в верхний регистр

```
@Test
public void multipleWordsToUppercase() {
    List<String> input = Arrays.asList("a", "b", "hello");
    List<String> result = Testing.allToUpperCase(input);
    assertEquals(asList("A", "B", "HELLO"), result);
}
```

Иногда лямбда-выражение обладает сложной функциональностью. Быть может, в нем есть ряд граничных случаев или оно вычисляет чрезвычайно важную для предметной области функцию. Вам абсолютно необходимо протестировать эту часть кода, но она реализована в виде лямбда-выражения, на которое невозможно сослаться.

В качестве примера такой проблемы рассмотрим метод, который лишь немного сложнее преобразования списка строк в верхний регистр. Мы будем преобразовывать не всю строку, а только первый символ. С помощью потоков и лямбда-выражений мы могли бы решить эту задачу, как показано в примере 7.10. Лямбда-выражение, выполняющее преобразование, находится в точке ❶.

Пример 7.10 ❖ Преобразование первого символа каждого элемента списка в верхний регистр

```
public static List<String> elementFirstToUpperCaseLambdas
          (List<String> words) {
    return words.stream()
                .map(value -> {  ❶
                    char firstChar = Character.toUpperCase(value.charAt(0));
                    return firstChar + value.substring(1);
                })
                .collect(Collectors.<String>toList());
}
```

Чтобы протестировать этот метод, нам нужно было бы для каждого интересующего нас случая создать список и проверить результат его преобразования. В примере 7.11 показано, каким громоздким может оказаться такое решение. Но не расстраивайтесь – выход есть!

Пример 7.11 ❖ Тестирование того, что в случае строки из двух символов в верхний регистр преобразуется только первый

```
@Test
public void twoLetterStringConvertedToUppercaseLambdas() {
    List<String> input = Arrays.asList("ab");
    List<String> result = Testing.elementFirstToUpperCaseLambdas(input);
    assertEquals(asList("Ab"), result);
}
```

Не используйте лямбда-выражение. Я понимаю, что такой совет может показаться странным в книге, посвященной лямбда-выражениям, но стоит ли непременно пытаться оседлать корову? Если мы согласимся, что не стоит, то возникает вопрос, как все же автономно протестировать код, не жертвуя удобством библиотек с поддержкой лямбда-выражений.

Используйте ссылки на методы. Любой метод, который можно было бы написать в виде лямбда-выражения, можно также записать в виде обычного метода, а затем обращаться к нему в других местах программы с помощью ссылки на метод.

В примере 7.12 я вынес лямбда-выражение в отдельный метод, который затем вызывается из главного метода, занимающегося преобразованием списка строк.

Пример 7.12 ❖ Преобразование первого символа в верхний регистр и применение этого метода ко всему списку

```
public static List<String> elementFirstToUppercase(List<String> words) {
    return words.stream()
                .map(Testing::firstToUppercase)
```

```
                .collect(Collectors.<String>toList());
}

public static String firstToUppercase(String value) {  ❶
    char firstChar = Character.toUpperCase(value.charAt(0));
    return firstChar + value.substring(1);
}
```

Выделив метод, который отвечает за обработку строки, мы сможем охватить все граничные случаи, тестируя этот метод автономно. В примере 7.13 показан рассмотренный выше тестовый код в упрощенном виде.

Пример 7.13 ❖ Тест строки из двух символов, примененный к одному методу

```
@Test
public void twoLetterStringConvertedToUppercase() {
    String input = "ab";
    String result = Testing.firstToUppercase(input);
    assertEquals("Ab", result);
}
```

Использование лямбда-выражений в тестовых двойниках

При написании автономных тестов нередко применяются *тестовые двойники*, которые описывают ожидаемое поведение других компонентов системы. Это полезно, потому что задача автономного тестирования – проверить работу класса или метода изолированно от всех прочих частей программы, а тестовые двойники позволяют реализовать такую изоляцию для целей тестирования.

Хотя тестовые двойники часто называют подставными объектами (mock), на самом деле есть два типа двойников: подставки и заглушки. Различие в том, что подставки позволяют проверить поведение кода. Самое лучшее изложение этого предмета см. в статье Мартина Фаулера по адресу *http://martinfowler.com/articles/mocksArentStubs.html*.

Один из самых простых способов использования лямбда-выражений в тестовом коде – реализация облегченных заглушек. Это действительно легко и естественно, если компонент, который необходимо заглушить, уже является функциональным интерфейсом.

В разделе «Поведенческий паттерн “пиши все дважды”» выше я показал, как оформить общую предметную логику в виде отдельного метода `countFeature`, а для реализации подсчета различных вещей ис-

пользовать лямбда-выражения. В примере 7.14 демонстрируется, как можно было автономно протестировать это поведение.

Пример 7.14 ❖ Использование лямбда-выражения в качестве тестового двойника, передаваемого методу countFeature

```
@Test
public void canCountFeatures() {
    OrderDomain order = new OrderDomain(asList(
            newAlbum("Exile on Main St."),
            newAlbum("Beggars Banquet"),
            newAlbum("Aftermath"),
            newAlbum("Let it Bleed")));

    assertEquals(8, order.countFeature(album -> 2));
}
```

Ожидаемое поведение состоит в том, что метод `countFeature` возвращает сумму некоторого числа, повторенного столько раз, сколько имеется альбомов. Здесь я передаю четыре разных альбома, а заглушка в моем тесте возвращает для каждого альбома число 2. В утверждении проверяется, что метод возвращает 8, то есть 2×4. Если вы ожидаете, что вашему коду будет передано лямбда-выражение, то обычно имеет смысл написать автономный тест, в котором какое-то выражение действительно передается.

В большинстве тестовых двойников ожидания более сложные. В таких случаях часто применяются каркасы типа *Mockito*, которые автоматически генерируют тестовые двойники. Рассмотрим простой пример, в котором нужно создать тестовый двойник для объекта `List`. Вместо того чтобы возвращать размер `List`, мы хотим вернуть размер другого `List`. Реализуя метод `size` подставного объекта `List`, мы не хотим задавать один-единственный ответ. Мы хотим, чтобы ответ был результатом выполнения некоей операции, поэтому передаем лямбда-выражение (пример 7.15).

Пример 7.15 ❖ Использование лямбда-выражения в сочетании с библиотекой Mockito

```
List<String> list = mock(List.class);

when(list.size()).thenAnswer(inv -> otherList.size());

assertEquals(3, list.size());
```

В библиотеке Mockito применяется интерфейс `Answer`, который позволяет предоставить альтернативную реализацию поведения. Ины-

ми словами, она уже поддерживает нашего старого знакомца: передачу кода как данных. Здесь можно использовать лямбда-выражение, потому что `Answer` – вот ведь как удачно получилось – функциональный интерфейс.

Отложенное вычисление и отладка

Работа с отладчиком обычно подразумевает пошаговое прохождение программы или установку точек останова. Иногда при использовании потоковой библиотеки можно столкнуться с ситуацией, когда отладка усложняется, потому что итерированием управляет сама библиотека, а многие потоковые операции отложены.

При традиционном императивном программировании, когда код является последовательностью действий, направленных на достижение цели, исследование состояния до или после действия имеет прямой смысл. В Java 8 сохраняется доступ ко всем существующим в IDE средствам отладки, но для получения полезных результатов иногда приходится идти на хитрость.

Протоколирование и печать

Допустим, требуется отладить код, в котором над коллекцией выполняется последовательность операций, и мы хотим видеть результат каждой операции по отдельности. Можно было бы распечатывать коллекцию после каждого шага. Но при работе с потоковой библиотекой это затруднительно, так как вычисление промежуточных шагов отложено.

Посмотрим, как можно было бы вывести промежуточные значения в журнал, взяв за образец императивную версию отчета о национальности исполнителей из главы 3. Если вы запамятовали – с кем не бывает? – напомню, что мы пытаемся найти страну, откуда родом каждый исполнитель альбома. В примере 7.16 найденные национальности записываются в журнал.

Пример 7.16 ❖ Протоколирование промежуточных результатов для отладки цикла for

```
Set<String> nationalities = new HashSet<>();
for (Artist artist : album.getMusicianList()) {
    if (artist.getName().startsWith("The")) {
        String nationality = artist.getNationality();
        System.out.println("Found nationality: " + nationality);
```

```
        nationalities.add(nationality);
    }
}
return nationalities;
```

Мы могли бы воспользоваться методом `forEach` для распечатки значений из потока, и это заодно привело к выполнению отложенных вычислений. Увы, у такого решения есть недостаток – после вызова `forEach` продолжить работу с потоком не получится, потому что поток можно использовать только один раз. И если мы настаиваем на применении такого подхода, то поток придется создать заново. В примере 7.17 показано, насколько безобразный при этом получается код.

Пример 7.17 ❖ Наивное применение forEach для протоколирования промежуточных результатов

```
album.getMusicians()
     .filter(artist -> artist.getName().startsWith("The"))
     .map(artist -> artist.getNationality())
     .forEach(nationality -> System.out.println("Found: " + nationality));

Set<String> nationalities
       = album.getMusicians()
              .filter(artist -> artist.getName().startsWith("The"))
              .map(artist -> artist.getNationality())
              .collect(Collectors.<String>toSet());
```

Решение: метод peek

По счастью, в потоковой библиотеке есть метод, который позволяет по очереди просматривать каждое значение и при этом продолжать операции с потоком. Он называется `peek`. В примере 7.18 предыдущий пример переписан с использованием `peek`, чтобы можно было распечатать значения из потока без необходимости заново запускать конвейер операций.

Пример 7.18 ❖ Использование peek для протоколирования промежуточных результатов

```
Set<String> nationalities
   = album.getMusicians()
          .filter(artist -> artist.getName().startsWith("The"))
          .map(artist -> artist.getNationality())
          .peek(nation -> System.out.println("Found nationality: " + nation))
          .collect(Collectors.<String>toSet());
```

Метод peek можно использовать и для вывода в имеющиеся системы протоколирования, например log4j, java.util.logging или slf4j, – точно таким же способом.

Точки останова в середине потока

Протоколирование – лишь один из многих трюков, на которые способен метод peek. Чтобы отлаживать поток поэлементно, по аналогии с пошаговым прохождением цикла мы можем поставить точку останова в теле метода peek.

В данном случае у метода peek может быть пустое тело, внутри которого ставится точка останова. Некоторые отладчики не позволяют поставить точку останова в пустом теле, тогда, чтобы удовлетворить отладчик, я обычно добавляю отображение некоторого значения на само себя. Решение не идеальное, но работает.

Основные моменты

- Подумайте, как подвергнуть рефакторингу унаследованный код, применив лямбда-выражения; на этот счет имеются общие рекомендации.
- Чтобы автономно протестировать лямбда-выражение любой сложности, поместите его внутрь обычного метода.
- Метод peek очень полезен для протоколирования промежуточных значений во время отладки.

Глава 8

Проектирование и архитектурные принципы

> Главным инструментом проектирования программного обеспечения является мозг человека, хорошо знакомого с принципами проектирования. Это не технология.
>
> – *Крейг Ларман*

Я уже показал, что лямбда-выражения – сравнительно простое изменение языка Java и что есть много способов их использования в сочетании со стандартными библиотеками JDK. Большая часть кода на Java написана разработчиками, работающими на уровне базового JDK, – такими же людьми, как и вы. Чтобы получить максимум пользы от лямбда-выражений, следует постепенно включать их в существующий код. Это всего лишь еще один инструмент в арсенале профессионального Java-разработчика, принципиально ничем не отличающийся от интерфейса или класса.

В этой главе мы рассмотрим, как лямбда-выражения помогают придерживаться принципов *SOLID*, в которых выражены рекомендации по качественному объектно-ориентированному программированию. Многие имеющиеся паттерны проектирования можно улучшить с помощью лямбда-выражений, и мы бегло познакомимся с этой темой.

Я уверен, что, работая в команде, вы попадали в ситуацию, когда сами были довольны тем, как реализовали некую функцию или исправили ошибку, но кто-то другой, взглянув на ваш код, – быть может, в ходе рецензирования – высказал замечания. Наличие разных мнений относительно того, что считать хорошим и плохим кодом, – дело обычное.

Когда человек не согласен с чем-то, он чаще всего отстаивает свою точку зрения. Рецензент написал бы код по-другому. Но ниоткуда не следует, что прав именно он, а не вы. Пуская лямбда-выражения в свою жизнь, вы получаете новый способ реализации, который следует принимать во внимание. Дело не в том, что они как-то особенно трудны или спорны, просто это еще один вопрос проектирования, который можно обсуждать и с которым можно не соглашаться.

Настоящая глава призвана помочь в этом отношении. Я расскажу о некоторых общепризнанных принципах и паттернах, с помощью которых создаются надежные и удобные для сопровождения программы, – не просто о новеньких, еще не утративших блеск библиотеках JDK, но и о том, как использовать лямбда-выражения в собственных приложениях и при разработке архитектуры системы.

Паттерны проектирования и лямбда-выражения

Все мы знакомы с одним из краеугольных камней проектирования ПО – идеей *паттернов проектирования*. Паттерны документируют стандартные подходы к решению типичных задач построения архитектуры программного обеспечения. Столкнувшись с задачей, для которой вам известен подходящий паттерн, вы можете применить его к имеющейся ситуации. В каком-то смысле паттерны – это свод правил, признанных наилучшими подходами к решению определенных задач.

Разумеется, никакое правило не остается наилучшим вечно. Типичный пример – популярный когда-то паттерн Одиночка (Singleton), который описывает создание единственного экземпляра класса. Последние десять лет его неоднократно критиковали за то, что он делает приложения более хрупкими и трудными для тестирования. По мере распространения движения за гибкие технологии выдвинулось на передний план тестирование приложений, и проблемы, связанные с паттерном Одиночка, превратили его в *антипаттерн*: паттерн, который никогда не следует использовать.

В этом разделе я не стану толковать о том, как паттерны устаревают. Вместо этого мы посмотрим, как некоторые существующие паттерны можно сделать лучше, проще, а иногда и реализовать по-новому. Во всех случаях силой, направляющей изменение паттернов, являются нововведения в Java 8.

Паттерн Команда

Команда – это объект, который инкапсулирует всю информацию, необходимую для вызова метода в будущем. *Паттерн Команда* – это способ использования такого объекта, позволяющий писать обобщенный код, в котором последовательность вызовов методов определяется на этапе выполнения. В паттерне Команда принимают участие четыре класса, показанные на рис. 8.1.

- Получатель – фактически выполняет работу.
- Команда – инкапсулирует всю информацию, необходимую для вызова получателя.
- Активатор – управляет последовательностью выполнения одной или нескольких команд.
- Клиент – создает конкретные экземпляры команд.

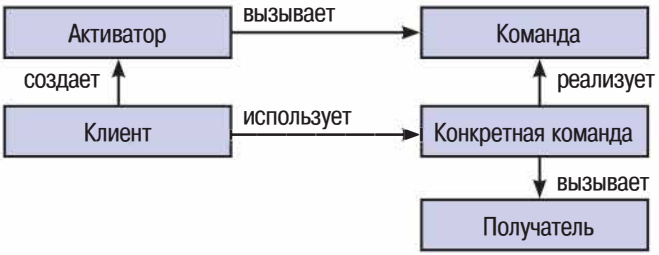

Рис. 8.1 ❖ Паттерн Команда

Рассмотрим конкретный пример использования паттерна Команда и попробуем улучшить его с помощью лямбда-выражений. Предположим, что имеется компонент графического интерфейса `Editor`, и в нем определены действия, которые мы можем вызывать, например `open` или `save`, как в примере 8.1. Мы хотим реализовать функциональность макроса, то есть последовательности операций, которую можно где-то сохранить, а затем выполнить как одну операцию. Это и будет наш получатель.

Пример 8.1 ❖ Типичные функции текстового редактора

```
public interface Editor {

    public void save();

    public void open();

    public void close();
}
```

В этом примере все операции, в частности `open` и `save`, – команды. Нам необходим обобщенный интерфейс команды, которому отвечали бы различные операции. Я назову этот интерфейс `Action`, поскольку он представляет выполнение одного действия в нашей предметной области. Этот интерфейс будут реализовывать все объекты-команды (пример 8.2).

Пример 8.2 ❖ Все наши действия реализуют интерфейс Action

```
public interface Action {

    public void perform();

}
```

Теперь можно реализовать интерфейс `Action` для каждой операции. Каждый класс должен вызывать один какой-то метод редактора `Editor`, обернув этот вызов интерфейсом `Action`. Я буду именовать классы в соответствии с обертываемыми ими операциями, придерживаясь определенного соглашения; так, методу `save` будет соответствовать класс `Save`. В примерах 8.3 и 8.4 приведены наши объекты-команды.

Пример 8.3 ❖ Действие save делегирует работу методу Editor

```
public class Save implements Action {

    private final Editor editor;

    public Save(Editor editor) {
        this.editor = editor;
    }

    @Override
    public void perform() {
        editor.save();
    }
}
```

Пример 8.4 ❖ Действие open также делегирует работу методу Editor

```
public class Open implements Action {

    private final Editor editor;

    public Open(Editor editor) {
        this.editor = editor;
    }

    @Override
```

```
    public void perform() {
        editor.open();
    }
}
```

Теперь можно реализовать класс `Macro`. Этот класс умеет записывать последовательность действий методом `record` и исполнять ее как единую группу. Мы будем хранить последовательность действий в объекте `List`, а для выполнения их по очереди воспользуемся методом `forEach`. В примере 8.5 показан наш активатор.

Пример 8.5 ❖ Макрос состоит из последовательности действий, вызываемых по очереди

```
public class Macro {

    private final List<Action> actions;

    public Macro() {
        actions = new ArrayList<>();
    }

    public void record(Action action) {
        actions.add(action);
    }

    public void run() {
        actions.forEach(Action::perform);
    }
}
```

Создавая макрос в программе, мы добавляем в объект `Macro` экземпляры каждой команды. После этого мы можем запустить макрос, и он вызовет эти команды одну за другой. Будучи ленивым программистом, я высоко ценю возможность определять стандартные цепочки действий в виде макросов. Я сказал «ленивый»? На самом деле я имел в виду «повернутый на продуктивности». Объект `Macro` – это наш клиент, он показан в примере 8.6.

Пример 8.6 ❖ Создание макроса с помощью паттерна Команда

```
Macro macro = new Macro();
macro.record(new Open(editor));
macro.record(new Save(editor));
macro.record(new Close(editor));
macro.run();
```

И как же нам помогут лямбда-выражения? На самом деле все наши классы команд, в частности `Save` и `Open`, – это просто лямбда-выраже-

ния, жаждущие выбраться из своих скорлупок на свет Божий. Это инкапсулированные поведения, для передачи которых из одного места в другое мы написали классы. Благодаря лямбда-выражениям весь паттерн можно значительно упростить, полностью отказавшись от обертывающих классов. В примере 8.7 показано, как можно использовать класс `Macro`, заменив классы команд лямбда-выражениями.

Пример 8.7 ❖ Создание макроса с помощью лямбда-выражений

```
Macro macro = new Macro();
macro.record(() -> editor.open());
macro.record(() -> editor.save());
macro.record(() -> editor.close());
macro.run();
```

Можно поступить еще лучше, осознав, что каждое лямбда-выражение вызывает единственный метод. А раз так, то мы можем применить ссылки на методы, чтобы связать команды редактора с объектом-макросом (см. пример 8.8).

Пример 8.8 ❖ Создание макроса с помощью ссылок на методы

```
Macro macro = new Macro();
macro.record(editor::open);
macro.record(editor::save);
macro.record(editor::close);
macro.run();
```

Паттерн Команда изначально служил заменой отсутствующим в языке лямбда-выражениям. Пользуясь настоящими лямбда-выражениями или ссылками на методы, мы можем сделать код чище, убрав служебные стереотипные конструкции и прояснив его назначение.

Макросы – лишь один пример возможного использования паттерна Команда. Он часто применяется при реализации компонентных систем графического интерфейса пользователя (ГИП), функции отмены, пулов потоков, транзакций и мастеров.

В базовых библиотеках Java уже имеется функциональный интерфейс `Runnable` с такой же структурой, как у нашего интерфейса `Action`. Мы могли бы использовать его для построения класса макроса, но в данном случае слово `Action` лучше отражает специфику предметной области, так что я решил создать собственный интерфейс.

Паттерн Стратегия

Паттерн Стратегия описывает способ изменения алгоритма программы на этапе выполнения. Конкретная реализация этого паттерна зависит от обстоятельств, но в любом случае идея одна и та же: опреде-

лить общую задачу, решаемую разными алгоритмами, а затем скрыть все алгоритмы за единым программным интерфейсом.

В качестве примера инкапсулируемого алгоритма можно взять сжатие файлов. Мы дадим пользователям возможность сжимать файлы с помощью алгоритма *zip* или *gzip* и реализуем обобщенный класс `Compressor`, который сможет выбирать любой из этих алгоритмов.

Прежде всего необходимо определить API нашей стратегии (см. рис. 8.2), которую я назову `CompressionStrategy`. Все алгоритмы сжатия будут реализовывать этот интерфейс. У них будет метод `compress`, который принимает и возвращает `OutputStream`. Выходной объект `OutputStream` – результат сжатия входного (см. пример 8.9).

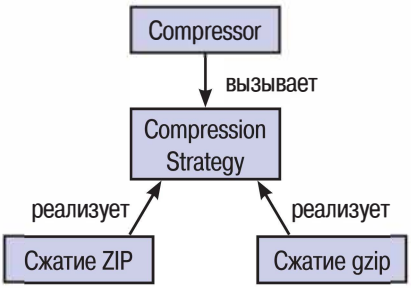

Рис. 8.2 ❖ Паттерн Стратегия

Пример 8.9 ❖ Определение интерфейса стратегии сжатия данных

```
public interface CompressionStrategy {

    public OutputStream compress(OutputStream data) throws IOException;

}
```

У нас есть две конкретные реализации этого интерфейса – для алгоритмов gzip и ZIP, – в которых мы воспользовались библиотечными классами Java для записи файлов в форматах gzip (пример 8.10) и ZIP (пример 8.11).

Пример 8.10 ❖ Применение алгоритма gzip для сжатия данных

```
public class GzipCompressionStrategy implements CompressionStrategy {

    @Override
    public OutputStream compress(OutputStream data) throws IOException {
        return new GZIPOutputStream(data);
    }
}
```

Пример 8.11 ❖ Применение алгоритма ZIP для сжатия данных

```
public class ZipCompressionStrategy implements CompressionStrategy {

    @Override
    public OutputStream compress(OutputStream data) throws IOException {
        return new ZipOutputStream(data);
    }
}
```

Теперь можно реализовать класс `Compressor`, являющийся контекстом, в котором используется наша стратегия. В этом классе есть метод `compress`, который принимает пути к входному и выходному файлам и записывает сжатую версию входного файла в выходной. Конструктору в качестве параметра передается объект `CompressionStrategy`, с помощью которого вызывающая программа на этапе выполнения принимает решение о том, какую стратегию сжатия использовать, — например, спросив пользователя о предпочтительном способе сжатия (см. пример 8.12).

Пример 8.12 ❖ Конструктору объекта Compressor передается стратегия сжатия

```
public class Compressor {

    private final CompressionStrategy strategy;

    public Compressor(CompressionStrategy strategy) {
        this.strategy = strategy;
    }

    public void compress(Path inFile, File outFile) throws IOException {
        try (OutputStream outStream = new FileOutputStream(outFile)) {
            Files.copy(inFile, strategy.compress(outStream));
        }
    }
}
```

При традиционной реализации паттерна Стратегия мы написали бы клиент, который создает объект `Compressor`, задавая нужную стратегию (пример 8.13).

Пример 8.13 ❖ Создание объектов Compressor с конкретными классами стратегий

```
Compressor gzipCompressor = new Compressor(new GzipCompressionStrategy());
gzipCompressor.compress(inFile, outFile);

Compressor zipCompressor = new Compressor(new ZipCompressionStrategy());
zipCompressor.compress(inFile, outFile);
```

Как и в рассмотренном выше паттерне Команда, использование лямбда-выражений или ссылок на методы позволяет исключить целый слой стереотипного кода. В данном случае мы можем отказаться от реализации конкретных стратегий и сослаться на метод, реализующий алгоритм. Алгоритмы будут представлены конструкторами соответствующих реализаций интерфейса `OutputStream`. При таком подходе классы `GzipCompressionStrategy` и `ZipCompressionStrategy` оказываются вообще лишними. В примере 8.14 показано, как выглядит код при использовании ссылок на методы.

Пример 8.14 ❖ Создание экземпляров класса Compressor с помощью ссылок на методы

```
Compressor gzipCompressor = new Compressor(GZIPOutputStream::new);
gzipCompressor.compress(inFile, outFile);

Compressor zipCompressor = new Compressor(ZipOutputStream::new);
zipCompressor.compress(inFile, outFile);
```

Паттерн Наблюдатель

Наблюдатель – еще один поведенческий паттерн, который можно улучшить и упростить за счет лямбда-выражений. В этом паттерне объект, именуемый *субъектом*, хранит список объектов, *наблюдающих* за ним. О любом изменении состояния субъекта уведомляются все наблюдатели. Паттерн часто используется в библиотеках построения графических интерфейсов на основе архитектуры MVC, для того чтобы компоненты-представления можно было обновлять при изменении состояния модели, не организуя сильной связи между двумя классами.

Рассматривать обновление компонентов ГИП скучно, поэтому в качестве субъекта возьмем Луну! Как НАСА, так и пришельцы желают следить за аппаратами, опускающимися на Луну. Задача НАСА – обеспечить безопасную посадку космического корабля Аполлон с астронавтами, а пришельцы хотят вторгнуться на Землю, когда НАСА отвлечется.

Для начала определим API интерфейса наблюдателей, который я назову `LandingObserver`. В нем имеется единственный метод `observeLanding`, который вызывается, когда что-то опускается на Луну (пример 8.15).

Пример 8.15 ❖ Интерфейс наблюдения за аппаратами, садящимися на Луну

```
public interface LandingObserver {
    public void observeLanding(String name);
}
```

В нашем случае классом субъекта является `Moon`, он хранит список экземпляров `LandingObserver`, уведомляет их о посадках и умеет добавлять новые экземпляры `LandingObserver`, шпионящие за Луной (пример 8.16).

Пример 8.16 ❖ Предметный класс Moon – не такой красивый, как настоящая Луна

```
public class Moon {

    private final List<LandingObserver> observers = new ArrayList<>();

    public void land(String name) {
        for (LandingObserver observer : observers) {
            observer.observeLanding(name);
        }
    }

    public void startSpying(LandingObserver observer) {
        observers.add(observer);
    }
}
```

Имеются две конкретные реализации класса `LandingObserver`, соответствующие пришельцам (пример 8.17) и НАСА (пример 8.18). Как уже было сказано, они реагируют на одно и то же событие совершенно по-разному.

Пример 8.17 ❖ Пришельцы могут наблюдать за высадкой людей на Луну

```
public class Aliens implements LandingObserver {
    @Override
    public void observeLanding(String name) {
        if (name.contains("Apollo")) {
            System.out.println("Они отвлеклись, вторгаемся на Землю!");
        }
    }
}
```

Пример 8.18 ❖ НАСА тоже может наблюдать за высадкой людей на Луну

```
public class Nasa implements LandingObserver {
    @Override
    public void observeLanding(String name) {
        if (name.contains("Apollo")) {
            System.out.println("Мы сделали это!");
        }
    }
}
```

Как и в предыдущих случаях, традиционная реализация этого паттерна подразумевает, что клиент создает дополнительные стереотипные классы, которые вполне можно заменить лямбда-выражениями (примеры 8.19 и 8.20).

Пример 8.19 ❖ Клиент создает экземпляр Moon, применяя классы, после чего моделирует посадку на Луну

```
Moon moon = new Moon();
moon.startSpying(new Nasa());
moon.startSpying(new Aliens());

moon.land("An asteroid");
moon.land("Apollo 11");
```

Пример 8.20 ❖ Клиент создает экземпляр Moon, применяя лямбда-выражения, после чего моделирует посадку на Луну

```
Moon moon = new Moon();

moon.startSpying(name -> {
    if (name.contains("Apollo"))
        System.out.println("Мы сделали это!");
    });

moon.startSpying(name -> {
    if (name.contains("Apollo"))
        System.out.println("Они отвлеклись, вторгаемся на Землю!");
    });

moon.land("An asteroid");
moon.land("Apollo 11");
```

Решая, как подойти к реализации паттернов Наблюдатель и Стратегия – с помощью лямбда-выражений или производных классов, – нужно принимать во внимание сложность кода стратегии или наблюдателя. В продемонстрированных выше случаях код был очень простым – всего один-два вызова методов – и неплохо согласовывался с новыми возможностями языка. Но если сам наблюдатель является достаточно сложным классом, то попытка утрамбовать слишком много кода в один метод чревата неудобочитаемостью.

В некотором смысле фраза «попытка утрамбовать слишком много кода в один метод чревата неудобочитаемостью» – золотое правило, определяющее, как следует применять лямбда-выражения. И если я до сих пор не подчеркивал его, то лишь потому, что оно равным образом относится и к написанию обычных методов!

Паттерн Шаблонный метод

При разработке программ часто возникает ситуация, когда имеется некий общий алгоритм с различными вариациями. Мы хотели бы, чтобы все вариации гарантированно реализовывали один и тот же принципиальный алгоритм, а код было легко понять. Уяснив общий принцип, будет проще разобраться в каждой реализации.

Паттерн Шаблонный метод предназначен как раз для таких ситуаций. Структура общего алгоритма представлена *абстрактным классом*. В нем имеется ряд абстрактных методов, соответствующих конкретным шагам алгоритма; при этом общий для всех реализаций код содержится в самом абстрактном классе. Каждый вариант алгоритма реализован в виде *конкретного класса*, который переопределяет абстрактные методы и предоставляет соответствующую своему назначению реализацию.

Следующий сценарий поможет прояснить ситуацию. Представьте, что вы – банк, который выдает кредиты гражданам, компаниям и своим работникам. Процедура рассмотрения заявки для всех категорий заемщиков похожа – необходимо удостоверить личность, проверить кредитную историю и получить сведения о доходах. Эта информация поступает из разных источников, и к ее оценке применяются различные критерии. Например, чтобы удостоверить личность физического лица, достаточно взглянуть на выставленный по его адресу счет. Компании же внесены в официальный реестр, роль которого в США играет Комиссия по ценным бумагам и биржам, а в Великобритании – Регистрационная палата.

Мы можем смоделировать это в программе с помощью абстрактного класса `LoanApplication`, который определяет структуру алгоритма и содержит общий код уведомления об исходе рассмотрения заявки. Для каждой категории заемщиков определены конкретные подклассы: `CompanyLoanApplication`, `PersonalLoanApplication` и `EmployeeLoanApplication`. В примере 8.21 показано, как выглядит класс `LoanApplication`.

Пример 8.21 ❖ Процесс рассмотрения заявки о предоставлении кредита, смоделированный с помощью паттерна Шаблонный метод

```
public abstract class LoanApplication {

    public void checkLoanApplication() throws ApplicationDenied {
        checkIdentity();
        checkCreditHistory();
        checkIncomeHistory();
```

```
        reportFindings();
    }

    protected abstract void checkIdentity() throws ApplicationDenied;

    protected abstract void checkIncomeHistory() throws ApplicationDenied;

    protected abstract void checkCreditHistory() throws ApplicationDenied;

    private void reportFindings() {
```

В классе `CompanyLoanApplication` метод `checkIdentity` реализован путем поиска информации в реестре компаний, например в базе данных Регистрационной палаты. Метод `checkIncomeHistory` должен был бы оценить имеющиеся отчеты компании о прибылях и убытках, а также ее бухгалтерский баланс. Метод `checkCreditHistory` мог бы поинтересоваться имеющимися безнадежными и непогашенными задолженностями.

В классе `PersonalLoanApplication` метод `checkIdentity` реализован путем анализа представленных клиентом бумажных документов с целью проверки существования его адреса проживания. Метод `checkIncomeHistory` оценивает расчетные листки и проверяет, имеется ли у заемщика постоянная работа. Метод `checkCreditHistory` делегирует работу внешнему бюро кредитных историй.

Класс `EmployeeLoanApplication`, по сути дела, совпадает с `PersonalLoanApplication`, только без проверки истории занятости. Так уж сложилось, что банк проверяет историю доходов при найме на работу (пример 8.22).

Пример 8.22 ❖ Частный случай – работник банка обращается за кредитом

```
public class EmployeeLoanApplication extends PersonalLoanApplication {
    @Override
    protected void checkIncomeHistory() {
        // Он же у нас работает!
    }
}
```

Лямбда-выражения и ссылки на методы позволяют взглянуть на паттерн Шаблонный метод в новом свете и реализовать его иначе. Истинный смысл паттерна заключается в выстраивании последовательности вызова методов в определенном порядке. Если представить функции в виде функциональных интерфейсов, а затем использовать лямбда-выражения или ссылки на методы для реализации этих ин-

терфейсов, то мы получим серьезный выигрыш в гибкости, по сравнению с наследованием. Взгляните, как можно было бы реализовать наш алгоритм `LoanApplication` (пример 8.23)!

Пример 8.23 ❖ Частный случай – работник банка обращается за кредитом

```
public class LoanApplication {

    private final Criteria identity;
    private final Criteria creditHistory;
    private final Criteria incomeHistory;

    public LoanApplication(Criteria identity,
                           Criteria creditHistory,
                           Criteria incomeHistory) {
        this.identity = identity;
        this.creditHistory = creditHistory;
        this.incomeHistory = incomeHistory;
    }

    public void checkLoanApplication() throws ApplicationDenied {
        identity.check();
        creditHistory.check();
        incomeHistory.check();
        reportFindings();
    }

    private void reportFindings() {
```

Как видите, вместо набора абстрактных методов мы завели поля с именами `identity`, `creditHistory` и `incomeHistory`. Каждое поле реализует функциональный интерфейс `Criteria`, который проверяет некоторый критерий и в случае его невыполнения возбуждает специальное исключение. Можно было бы возвращать из метода `check` экземпляр специального класса, обозначающий успех или неудачу, но возбуждение исключения следует общему соглашению, принятому в исходной реализации (см. пример 8.24).

Пример 8.24 ❖ Функциональный интерфейс, который возбуждает исключение в случае отклонения заявки

```
public interface Criteria {
    public void check() throws ApplicationDenied;
}
```

Преимущество этого подхода, по сравнению с основанным на наследовании, заключается в том, что мы не привязываем реализацию алгоритма к иерархии классов, наследующих `LoanApplication`, а сохра-

няем гибкость при решении о том, кому делегировать функциональность. Например, мы можем возложить ответственность за проверку всех критериев на класс Company, в котором будут методы с показанными ниже сигнатурами.

Пример 8.25 ❖ Методы проверки критериев в классе Company

```
public void checkIdentity() throws ApplicationDenied;

public void checkProfitAndLoss() throws ApplicationDenied;

public void checkHistoricalDebt() throws ApplicationDenied;
```

Теперь классу CompanyLoanApplication остается только передать ссылки на эти методы, как показано в примере 8.26.

Пример 8.26 ❖ Класс CompanyLoanApplication определяет, какие методы проверяют каждый критерий

```
public class CompanyLoanApplication extends LoanApplication {

    public CompanyLoanApplication(Company company) {
        super(company::checkIdentity,
              company::checkHistoricalDebt,
              company::checkProfitAndLoss);
    }
}
```

В обоснование делегирования поведения классу Company можно привести соображение о том, что поиск информации о компании зависит от страны. В Великобритании Регистрационная палата является каноническим местом регистрации сведений о компании, тогда как в США ситуация различна в разных штатах.

Применение функциональных интерфейсов для проверки критериев вовсе не мешает поместить реализации в подклассы. Мы можем явно использовать лямбда-выражения, в которых участвуют классы реализации, или ссылки на методы в текущем классе.

Нет также необходимости связывать классы EmployeeLoanApplication и PersonalLoanApplication отношением наследования, чтобы повторно использовать функциональность EmployeeLoanApplication в PersonalLoanApplication. Можно просто передать ссылки на одни и те же методы. Является ли один класс подклассом другого, должно определяться тем, действительно ли выдача кредита работнику банка является частным случаем выдачи кредита физическому лицу или еще кому-то. Поэтому описанный подход позволяет точнее моделировать рассматриваемую предметную область.

Предметно-ориентированные языки с поддержкой лямбда-выражений

Предметно-ориентированный язык (DSL) – это язык программирования, спроектированный для одной конкретной части программной системы. Обычно такие языки невелики по объему и менее выразительны, чем языки общего назначения, например Java. DSL-языки высоко специализированы: они умеют делать не всё, но то, что умеют, делают хорошо.

Принято выделять две категории DSL-языков: *внутренние* и *внешние*. Внешний язык не связан с исходным кодом вашей программы, его анализатор и интерпретатор реализованы кем-то другим. Широко известными примерами внешних предметно-ориентированных языков являются каскадные таблицы стилей (CSS) и регулярные выражения.

Внутренние DSL-языки встроены в язык программирования, на котором написаны. Если вам доводилось работать с библиотеками генерации подставных объектов типа JMock или Mockito или с построителями SQL-запросов типа JOOQ или Querydsl, то вы знаете, что такое внутренний DSL-язык. В каком-то смысле это просто обычные библиотеки с текучим API. Несмотря на свою простоту, внутренние DSL-языки весьма ценны, потому что позволяют сделать код более кратким и удобочитаемым. В идеале код, написанный на DSL-языке, читается как высказывание из той предметной области, которую отражает.

С появлением лямбда-выражений стало проще реализовывать DSL-языки с текучим интерфейсом, что добавляет еще один инструмент в арсенал тех, кто желает экспериментировать с предметно-ориентированными языками. Мы изучим эту тему, построив DSL-язык для разработки на основе поведения (behavior-driven development – BDD), который назовем *LambdaBehave*.

BDD – это вариант разработки через тестирование (TDD), в котором упор делается на рассуждениях о поведении программы, а не просто на тестах, которые она должна пройти. За основу проекта языка мы взяли BDD-каркас Jasmine для языка JavaScript BDD, который активно используется разработчиками клиентской части веб-сайтов. В примере 8.27 показан простой скрипт, демонстрирующий, как с помощью Jasmine пишутся тесты.

Пример 8.27 ❖ Jasmine

```
describe("A suite is just a function", function() {
    it("and so is a spec", function() {
```

```
        var a = true;

        expect(a).toBe(true);
    });
});
```

Признаю, что читателям, незнакомым с JavaScript, этот код может показаться загадочным. Разрабатывая эквивалентный язык для Java 8, мы будем продвигаться неспешно. Помните только, что в JavaScript лямбда-выражения синтаксически записываются в виде `function() { … }`.

Рассмотрим все концепции по очереди.

- Каждая *спецификация* описывает одно поведение программы.
- *Ожидание* – это способ описать поведение приложения. Ожидания содержатся в спецификациях.
- Группы спецификаций объединяются в *комплект*.

У каждой из этих концепций есть аналог в традиционных каркасах тестирования, например JUnit. Спецификации соответствует тестовый метод, ожиданию – утверждение, а комплекту – тестовый класс.

Предметно-ориентированный язык на Java

Рассмотрим пример того, что мы собираемся достичь с помощью нашего каркаса BDD, написанного на Java. В примере 8.28 приведена спецификация некоторых поведений класса `Stack`.

Пример 8.28 ❖ Избранные истории, специфицирующие класс Stack

```
public class StackSpec {{

    describe("a stack", it -> {

        it.should("be empty when created", expect -> {
            expect.that(new Stack()).isEmpty();
        });

        it.should("push new elements onto the top of the stack", expect -> {
            Stack<Integer> stack = new Stack<>();
            stack.push(1);

            expect.that(stack.get(0)).isEqualTo(1);
        });

        it.should("pop the last element pushed onto the stack", expect -> {
            Stack<Integer> stack = new Stack<>();
            stack.push(2);
```

```
            stack.push(1);

            expect.that(stack.pop()).isEqualTo(2);
        });
    });
})
```

Комплект спецификаций начинается глаголом describe. Затем мы присваиваем комплекту имя, сообщающее, поведение чего описывается; в данном случае мы назвали комплект "a stack".

Каждая спецификация читается как предложение на английском языке. Все они начинаются фразой it.should, в которой it – ссылка на объект, чье поведение описывается. Далее следует обычное англоязычное предложение, в котором словами сообщается, какое именно поведение проверяется. Затем мы описываем свои ожидания, каждое из которых начинается фразой expect.that.

В результате проверки спецификаций мы получаем простой отчет с информацией о том, оправдалось ожидание или нет. Обратите внимание, что в спецификации «pop the last element pushed onto the stack» (извлечь из стека последний помещенный в него элемент) ожидалось, что pop вернет 2, а не 1, так что ожидание не оправдалось:

```
a stack
should pop the last element pushed onto the stack[expected ❶: but was: ❷]
should be empty when created
should push new elements onto the top of the stack
```

Как это делается

Итак, вы теперь понимаете, что я имел в вид под текучестью DSL-языка, которая обеспечивается применением лямбда-выражений. Посмотрим, как это реализовано. Надеюсь, мое объяснение продемонстрирует, насколько просто пишутся такого рода каркасы.

Описание любого поведения начинается глаголом describe. В действительности это не что иное, как статически импортированный метод. Он создает экземпляр класса Description для всего комплекта и поручает ему обработку спецификации. Классу Description принадлежат параметры it в нашем языке спецификаций (см. пример 8.29).

Пример 8.29 ❖ Метод describe, который начинает определение спецификации

```
public static void describe(String name, Suite behavior) {
    Description description = new Description(name);
    behavior.specifySuite(description);
}
```

В каждом комплекте имеется код, который пишется пользователем с помощью лямбда-выражения. Поэтому нам необходим функциональный интерфейс `Suite`, показанный в примере 8.30, который представляет комплект спецификаций. Обратите внимание, что метод этого интерфейса принимает в качестве аргумента объект `Description`, переданный из метода `describe`.

Пример 8.30 ❖ Комплект тестов – это лямбда-выражение, реализующее данный интерфейс

```
public interface Suite {
    public void specifySuite(Description description);
}
```

Лямбда-выражениями представлены не только комплекты в нашем DSL-языке, но и отдельные спецификации. Они тоже нуждаются в функциональном интерфейсе, который я назову `Specification` (пример 8.31). Переменная `expect` в примере кода выше – экземпляр класса `Expect`, который я опишу ниже.

Пример 8.31 ❖ Спецификация – лямбда-выражение, реализующее данный интерфейс

```
public interface Specification {
    public void specifyBehaviour(Expect expect);
}
```

В этом месте оказывается полезен передаваемый экземпляр `Description`. Мы хотим, чтобы пользователь мог с помощью текучего синтаксиса описывать спецификации в предложении `it.should`. Это означает, что в классе `Description` должен присутствовать метод `should` (см. пример 8.32). Именно в нем и сосредоточена вся работа, поскольку этот метод выполняет лямбда-выражение, вызывая его метод `specifyBehaviour`. Если спецификация не удовлетворяется, она сообщит об этом, возбудив стандартное исключение Java `AssertionError`, а любое другое исключение типа `Throwable` мы будем считать ошибкой.

Пример 8.32 ❖ Лямбда-выражения, выражающие спецификации, передаются методу should

```
public void should(String description, Specification specification) {
    try {
        Expect expect = new Expect();
        specification.specifyBehaviour(expect);
        Runner.current.recordSuccess(suite, description);
    } catch (AssertionError cause) {
        Runner.current.recordFailure(suite, description, cause);
```

```
    } catch (Throwable cause) {
        Runner.current.recordError(suite, description, cause);
    }
}
```

Желая описать в спецификации ожидание, мы используем фразу `expect.that`. Значит, в классе `Expect` должен быть метод `that`, показанный в примере 8.33. Он обертывает переданный ему объект, а обертка может затем раскрывать текучие методы, например `isEqualTo`, которые возбуждают подходящее исключение в случае, когда ожидание не удовлетворяется.

Пример 8.33 ❖ Начало текучей цепочки ожиданий

```
public final class Expect {

    public BoundExpectation that(Object value) {
        return new BoundExpectation(value);
    }

    // Остальная часть класса опущена
```

Возможно, вы обратили внимание на одну деталь, на которой я до сих не акцентировал внимания и которая не имеет отношения к лямбда-выражениям. В классе `StackSpec` нет явно реализованных методов, но тем не менее я написал внутри него код. Тут я немного схитрил – использовал двойные фигурные скобки в начале и в конце определения класса:

```
public class StackSpec {{
    ...
}}
```

Тем самым я определил анонимный конструктор, внутри которого можно выполнить произвольный код на Java. Можно было бы вместо этого написать конструктор полностью, но это увеличило бы объем стереотипного кода:

```
public class StackSpec {
    public StackSpec() {
        ...
    }
}
```

Для реализации всего каркаса BDD придется еще немало поработать, но в этом разделе я ставил целью показать, как с помощью лямбда-выражений можно создавать текучие предметно-ориентированные языки. Я рассмотрел лишь те части DSL-языка, которые взаи-

модействуют с лямбда-выражениями, чтобы вы почувствовали, как такие вещи реализуются.

Оценка

Одна из сторон текучести – приспособленность DSL-языка к IDE. Иными словами, вы должны помнить лишь минимум информации, а остальное подскажет механизм автоматического завершения кода. Именно поэтому мы используем и передаем объекты `Description` и `Expect`. Альтернатива – завести статические методы `it` или `expect`, и такое решение действительно применяется в некоторых DSL-языках. Если передавать лямбда-выражению объект, а не требовать статического импорта, то знающему пользователю IDE будет проще воспользоваться автозавершением кода во время разработки.

Единственное, о чем должен помнить пользователь, – необходимость вызова `describe`. Достоинства такого подхода трудно оценить, просто читая текст книги, но я призываю вас поэкспериментировать с этим каркасом на каком-нибудь простеньком проекте и убедиться.

Еще стоит обратить внимание на то, что в большинстве каркасов тестирования широко применяются аннотации и используется отражение. Нам нет нужды прибегать к таким трюкам. Мы можем непосредственно представить поведение в DSL-языке с помощью лямбда-выражений, обращаясь с ними как с обычными методами Java.

Принципы SOLID и лямбда-выражения

SOLID – это набор принципов проектирования объектно-ориентированных программ. В акрониме скрыты первые буквы названий пяти принципов: Single responsibility (принцип единственной обязанности), Open/closed (принцип открытости-закрытости), Liskov substitution (принцип подстановки Лисков), Interface segregation (принцип разделения интерфейсов) и Dependency inversion (принцип инверсии зависимости). В этих принципах сформулированы рекомендации о том, как писать код, чтобы впоследствии его было легко сопровождать и развивать.

Каждый принцип соответствует определенным «запашкам», которые могут присутствовать в коде, и предлагает способ решения вызываемых ими проблем. На эту тему написано много книг, и я не собираюсь разбирать принципы SOLID во всех деталях. Однако я покажу, как они применяются в контексте лямбда-выражений. В Java 8

некоторые принципы можно даже обобщить, выйдя за рамки исходных ограничений.

Принцип единственной обязанности

У каждого класса или метода в программе должна быть только одна причина для изменения.

Требования к программному обеспечению со временем меняются – от этого факта никуда не деться. Причины могут быть различны: добавляется новая функция, у вас или у заказчика изменяется понимание предметной области, необходимо ускорить работу программы и т. д.

Если изменяются требования к программе, то изменяются также обязанности классов и методов, в которых были реализованы старые требования. Если у класса несколько обязанностей, то при изменении какой-то одной могут оказаться затронуты другие обязанности того же класса. Это может привести к ошибкам и стать препятствием на пути развития кода.

Рассмотрим простой пример программы, генерирующий сводный баланс (`BalanceSheet`). Программа должна построить баланс по списку активов, представить его в виде таблицы и вывести в виде PDF-файла. Если разработчик поместит две обязанности – построение таблицы и форматирование в виде PDF – в один класс, то у этого класса будут две причины для изменения. Возможно, в будущем понадобится другой формат вывода, например HTML. А возможно, нужно будет изменить уровень детализации в самом балансе. Это достаточное основание для того, чтобы разбить задачу на два класса верхнего уровня: представление `BalanceSheet` в виде таблицы и вывод этой таблицы.

Однако принцип единственной обязанности этим не исчерпывается. Мало того что у класса должна быть только одна обязанность, он еще должен ее инкапсулировать. Иначе говоря, если я захочу изменить формат вывода, то должен буду заниматься только классом форматирования, полностью игнорируя класс построения таблицы.

О так спроектированных программах говорят, что они обладают свойством сильной *сцепленности*. Класс называется сцепленным, если его поля и методы следует рассматривать только вместе, потому что они тесно соотносятся друг с другом. Попытавшись разделить сцепленный класс на части, вы получите сильно связанные классы.

Итак, мы познакомились с принципом единственной обязанности, но возникает вопрос: какое отношение это имеет к лямбда-выражениям? А дело в том, что благодаря лямбда-выражениям гораздо проще

реализовать этот принцип на уровне методов. Рассмотрим код, который подсчитывает количество простых чисел, не превосходящих заданного (пример 8.34).

Пример 8.34 ❖ Подсчет простых чисел в методе, имеющем несколько обязанностей

```
public long countPrimes(int upTo) {
    long tally = 0;
    for (int i = 1; i < upTo; i++) {
        boolean isPrime = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                isPrime = false;
            }
        }
        if (isPrime) {
            tally++;
        }
    }
    return tally;
}
```

Нетрудно понять, что здесь делаются две вещи: подсчет количества чисел, обладающих некоторым свойством, и проверка того, что число является простым. В примере 8.35 показано, как переработать этот код, явно выделив обе обязанности.

Пример 8.35 ❖ Подсчет простых чисел после выделения метода isPrime, проверяющего, что число простое

```
public long countPrimes(int upTo) {
    long tally = 0;
    for (int i = 1; i < upTo; i++) {
        if (isPrime(i)) {
            tally++;
        }
    }
    return tally;
}

private boolean isPrime(int number) {
    for (int i = 2; i < number; i++) {
        if (number % i == 0) {
            return false;
        }
    }
    return true;
}
```

К сожалению, наш код по-прежнему имеет две обязанности. Большая его часть занимается перебором чисел в цикле. Если следовать принципу единственной обязанности, то такой перебор необходимо как-то инкапсулировать. Существует и еще одна вполне практическая причина усовершенствовать программу. Если значение `upTo` очень велико, то хорошо бы распараллелить подсчет простых чисел. И конечно же, потоковая модель – обязанность программы!

Этот код можно переработать с использованием потоковой библиотеки Java 8 (см. пример 8.36), поручив управление циклом самой библиотеке. Мы воспользуемся методом `range` для построения множества чисел от `0` до `upTo`, методом `filter` – для проверки числа на простоту и методом `count` – для вычисления результата.

Пример 8.36 ❖ Рефакторинг подсчета простых чисел с использованием потоков

```
public long countPrimes(int upTo) {
    return IntStream.range(1, upTo)
                    .filter(this::isPrime)
                    .count();
}

private boolean isPrime(int number) {
    return IntStream.range(2, number)
                    .allMatch(x -> (number % x) != 0);
}
```

Если мы захотим ускорить операцию ценой большего потребления ресурсов процессора, то сможем воспользоваться методом `parallelStream`, больше ничего не изменяя в коде (пример 8.37).

Пример 8.37 ❖ Параллельный подсчет простых чисел с использованием потоков

```
public long countPrimes(int upTo) {
    return IntStream.range(1, upTo)
                    .parallel()
                    .filter(this::isPrime)
                    .count();
}

private boolean isPrime(int number) {
    return IntStream.range(2, number)
                    .allMatch(x -> (number % x) != 0);
}
```

Таким образом, чтобы упростить реализацию принципа единственной обязанности, мы можем прибегнуть к функциям высшего порядка.

Принцип открытости-закрытости

> Программная единица должна быть открыта для расширения, но закрыта для модификации.
>
> – *Бертран Мейер*

Главная цель принципа открытости-закрытости аналогична цели принципа единственной обязанности: уменьшить хрупкость программы перед лицом изменений. Проблема, как и раньше, заключается в том, что всего одно добавление новой функции или изменение существующей может аукнуться в различных частях кода и стать причиной новых ошибок. Принцип открытости-закрытости – попытка уйти от этой проблемы, проектируя классы так, чтобы их можно было расширять, не изменяя внутренней реализации.

Когда человек впервые слышит о принципе открытости-закрытости, он воспринимает его как пустое прожектерство. Как это – расширить функциональность класса, не меняя его реализации? Ответ прост – нужно опираться на абстракцию и подключать новую функциональность, согласующуюся с этой абстракцией. Рассмотрим конкретный пример.

Допустим, что нам нужно написать программу, которая собирает информацию о производительности системы и представляет результаты измерений в виде графиков. Например, может быть график, показывающий, сколько времени компьютер проводит в режиме пользователя, в режиме ядра и в режиме ввода-вывода. Назовем класс, отвечающий за отображение этих метрик, `MetricDataGraph`.

Класс `MetricDataGraph` можно спроектировать так, чтобы агенты, собирающие данные, добавляли в него каждый новый результат измерения. Тогда его открытый API будет выглядеть, как показано в примере 8.38.

Пример 8.38 ❖ Открытый API класса MetricDataGraph

```
class MetricDataGraph {

    public void updateUserTime(int value);

    public void updateSystemTime(int value);

    public void updateIoTime(int value);

}
```

Но это означает, что всякий раз, как мы захотим добавить в график новый набор результатов измерений, класс `MetricDataGraph` придется модифицировать. Эту проблему можно решить путем заведения абстракции, представляющей ряд моментов времени, которую я назову `TimeSeries`. Теперь API класса `MetricDataGraph` можно упростить, так что он не будет зависеть от конкретных типов отображаемых метрик (пример 8.39).

Пример 8.39 ❖ Упрощенный API класса MetricDataGraph

```
class MetricDataGraph {
    public void addTimeSeries(TimeSeries values);
}
```

Каждый набор результатов измерения метрик можно затем представить в виде класса, реализующего интерфейс `TimeSeries`, и подключить к программе. Например, могут существовать конкретные классы `UserTimeSeries`, `SystemTimeSeries` и `IoTimeSeries`. Если бы мы захотели впоследствии показать еще и заимствованное у виртуализованной машины процессорное время, то нужно было бы добавить еще одну реализацию `TimeSeries` под названием `StealTimeSeries`. Таким образом, класс `MetricDataGraph` расширен, но при этом не модифицирован.

Функции высшего порядка обладают тем же свойством открытости для расширения, хотя и закрыты для модификации. Хорошим примером может служить класс `ThreadLocal`, с которым мы уже встречались. Этот класс предоставляет переменную, уникальную в том смысле, что в каждом потоке имеется отдельный ее экземпляр. Его статический метод `withInitial` – это функция высшего порядка, которая принимает лямбда-выражение, представляющее фабрику для порождения начального значения.

Тем самым реализуется принцип открытости-закрытости, потому что мы можем получить от `ThreadLocal` новое поведение, не внося никаких модификаций. Стоит передать `withInitial` новый фабричный метод, как мы получим экземпляр `ThreadLocal` с новым поведением. Например, с помощью `ThreadLocal` можно породить потокобезопасную переменную типа `DateFormatter` (пример 8.40).

Пример 8.40 ❖ Поточно-локальный форматер данных

```
// Одна реализация
ThreadLocal<DateFormat> localFormatter
    = ThreadLocal.withInitial(() -> new SimpleDateFormat());

// Использование
DateFormat formatter = localFormatter.get();
```

Мы можем получить совершенно иное поведение, передав другое лямбда-выражение. Так, в примере 8.41 создаются уникальные последовательные идентификаторы потоков Java.

Пример 8.41 ❖ Поточно-локальный идентификатор

```
// Или...
AtomicInteger threadId = new AtomicInteger();
ThreadLocal<Integer> localId
    = ThreadLocal.withInitial(() -> threadId.getAndIncrement());

// Использование
int idForThisThread = localId.get();
```

Еще одна интерпретация принципа открытости-закрытости, не укладывающаяся в традиционное его понимание, – идея о том, что неизменяемые объекты этому принципу удовлетворяют. Неизменяемым называется объект, который нельзя модифицировать после создания.

У термина «неизменяемость» есть две возможные интерпретации: *наблюдаемая неизменяемость* и *неизменяемость реализации*. Наблюдаемая неизменяемость означает, что с точки зрения любого другого объекта класс является неизменяемым, а неизменяемость реализации – что объект действительно никогда не изменяется. Из неизменяемости реализации следует наблюдаемая неизменяемость, обратное верно не всегда.

Примером класса, заявляющего о своей неизменяемости, но в действительности обладающего только свойством наблюдаемой неизменяемости, является класс `java.lang.String` – ведь он кэширует хэш-код при первом вызове метода `hashCode`. С точки зрения всех остальных классов, это совершенно безопасно, потому что они не могут увидеть различия между объектом сразу после конструирования и после кэширования хэш-кода.

Я завел речь о неизменяемых объектах в книге, посвященной лямбда-выражениям, потому что обе концепции широко применяются в функциональном программировании, откуда лямбда-выражения, собственно, и пришли. Поэтому они естественно ложатся на тот стиль программирования, о котором рассказывается в этой книге.

Неизменяемые объекты удовлетворяют принципу открытости-закрытости в том смысле, что их внутреннее состояние нельзя модифицировать, а новые методы можно безопасно добавлять. Новые методы не могут изменить внутреннего состояния объекта, поэтому такие объекты закрыты для модификации. Однако они добавляют новое

поведение, и, значит, объекты открыты для расширения. Разумеется, нужно внимательно следить за тем, чтобы ненароком не модифицировать состояние где-то в другом месте программы.

Неизменяемые объекты представляют интерес еще и потому, что по сути своей потокобезопасны. У них нет внутреннего состояния, которое могло бы измениться, поэтому ими могут сообща пользоваться несколько потоков. Размышления о различных подходах определенно завели бы нас в сторону от традиционного принципа открытости-закрытости. На самом деле, впервые формулируя этот принцип, Бертран Мейер говорил, что сам класс нельзя изменять после того, как его разработка завершена. Но современному программисту, практикующему гибкие методики, идея раз и навсегда завершенного класса чужда. Из-за требований заказчиков и характера использования приложения иногда класс приходится использовать так, как заранее не предполагалось. Но это не значит, что о принципе открытости-закрытости можно забыть. Просто надо понимать, что принципы следует воспринимать как рекомендации и эвристические соображения, которым необязательно следовать слепо, до последней запятой.

И последнее, о чем я предлагаю задуматься, – тот факт, что в контексте Java 8 интерпретация принципа открытости-закрытости как абстракции, которой могут соответствовать различные классы, или как призыва к использованию функций высшего порядка – по существу, одно и то же. Поскольку наша абстракция должна быть представлена в виде интерфейса или абстрактного класса, чьи методы вызываются, такой подход к принципу открытости-закрытости – это просто проявление полиморфизма.

В Java 8 любое лямбда-выражение, передаваемое функции высшего порядка, представлено функциональным интерфейсом. Функция высшего порядка вызывает его единственный метод, который приводит к различным результатам в зависимости от того, какое лямбда-выражение передано. А под капотом все тот же полиморфизм, необходимый для реализации принципа открытости-закрытости.

Принцип инверсии зависимости

Абстракции не должны зависеть от деталей – детали должны зависеть от абстракций.

Один из способов получить жесткую и хрупкую программу, которая упорно сопротивляется внесению любых изменений, – прочно связать между собой высокоуровневую бизнес-логику и низкоуровневый код, предназначенный для организации взаимосвязей между

модулями. Дело в том, что это два различных аспекта, которые могут независимо изменяться со временем.

Цель принципа инверсии зависимости заключается в том, чтобы дать программисту возможность разрабатывать высокоуровневую бизнес-логику независимо от низкоуровневого связующего кода. Это позволит повторно использовать высокоуровневый код, не принимая во внимание деталей, от которых он зависит. Достигаемая таким образом модульность работает в обоих направлениях: мы можем заменить одни детали другими при повторном использовании высокоуровневого кода или повторно использовать детали реализации, надстроив над ними другую бизнес-логику.

Рассмотрим конкретный пример традиционного использования принципа инверсии зависимости – высокоуровневую декомпозицию приложения, которое автоматически строит адресную книгу. Это приложение получает на входе последовательность электронных визитных карточек и постепенно создает адресную книгу с помощью какого-то механизма сохранения.

Очевидно, эту программу можно разбить на три крупных модуля:

- модуль считывания визиток, который понимает формат электронной визитки;
- модуль сохранения адресной книги в текстовом файле;
- модуль сбора, который извлекает полезную информацию из электронных визиток и помещает ее в адресную книгу.

Связи между этими модулями изображены на рис. 8.3.

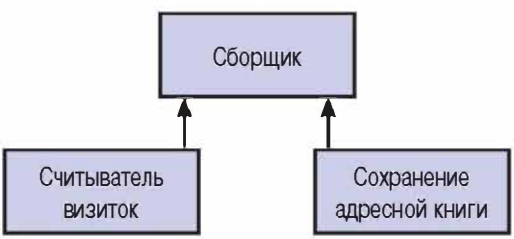

Рис. 8.3 ❖ Зависимости

Повторно использовать модуль сбора из этой системы будет довольно затруднительно, но модули считывания визиток и сохранения адресной книги ни от каких других компонентов не зависят. Поэтому их легко использовать и в других системах. Их можно также подменять; например, можно взять другой считыватель, скажем, получать

данные из учетных записей в Твиттере, или хранить адресную книгу не в текстовом файле, а в базе данных.

Чтобы обеспечить гибкость, необходимую для подмены системных компонентов, мы должны позаботиться о том, чтобы реализация модуля сбора не зависела от конкретных деталей модулей считывания визиток и сохранения адресной книги. Поэтому мы введем абстракцию считывания и абстракцию записи информации. От этих абстракций и будет зависеть реализация модуля сбора. Конкретные детали их реализаций можно будет подставить на этапе выполнения. Это и есть принцип инверсии зависимости в действии.

Если говорить о лямбда-выражениях, то многие функции высшего порядка, с которыми мы встречались выше, обеспечивают инверсию зависимости. Например, функция `map` позволяет повторно использовать код общей концепции преобразования потока значений, подставляя конкретное преобразование. Сама функция `map` зависит не от деталей преобразования, а от абстракции, каковой в данном случае является функциональный интерфейс `Function`.

Более сложный пример инверсии зависимости дает управление ресурсами. Очевидно, что управлять можно самыми разными ресурсами: соединениями с базами данных, пулами потоков, файлами, сетевыми соединениями и т. п. В качестве примера я возьму файлы, потому что это относительно простой ресурс, но общий принцип применим и к более сложным ресурсам, необходимым приложению.

Рассмотрим код, который извлекает заголовки из текста на гипотетическом языке разметки, в котором для обозначения заголовка применяется двоеточие в качестве суффикса. Наш метод будет читать файл, анализировать каждую строку по очереди, отфильтровывать только заголовки и закрывать файл. Кроме того, мы обернем все исключения, относящиеся к файловому вводу-выводу, специальным исключением `HeadingLookupException`, больше соответствующим предметной области. Код приведен в примере 8.42.

Пример 8.42 ❖ Выделение заголовков из файла

```
public List<String> findHeadings(Reader input) {
    try (BufferedReader reader = new BufferedReader(input)) {
        return reader.lines()
                     .filter(line -> line.endsWith(":"))
                     .map(line -> line.substring(0, line.length() - 1))
                     .collect(toList());
    } catch (IOException e) {
        throw new HeadingLookupException(e);
    }
}
```

К сожалению, наш код выделения заголовков сильно связан с кодом управления ресурсами и обработки файла. В действительности нам необходимо написать код, который только ищет заголовки, а детали работы с файлами делегирует другому методу. Например, в качестве абстракции, от которой будет зависеть этот код, можно выбрать не файл, а `Stream<String>`. Поток `Stream` – вещь гораздо более безопасная, которую труднее использовать неправильно. Мы также хотим передать функцию, создающую «предметное» исключение, если обнаружится проблема при работе с файлом. Такой подход, показанный в примере 8.43, позволяет разделить обработку ошибок предметной области и ошибок управления ресурсами.

Пример 8.43 ❖ Работа с файлом отделена от логики предметной области

```
public List<String> findHeadings(Reader input) {
    return withLinesOf(input,
                lines -> lines.filter(line -> line.endsWith(":"))
                              .map(line -> line.substring(0, line.length()-1))
                              .collect(toList()),
                HeadingLookupException::new);
    }
```

Полагаю, вам не терпится узнать, как выглядит метод `withLinesOf`! Он показан в примере 8.44.

Пример 8.44 ❖ Определение метода withLinesOf

```
private <T> T withLinesOf(Reader input,
                          Function<Stream<String>, T> handler,
                          Function<IOException, RuntimeException> error) {

    try (BufferedReader reader = new BufferedReader(input)) {
        return handler.apply(reader.lines());
    } catch (IOException e) {
        throw error.apply(e);
    }
}
```

Метод `withLinesOf` принимает объект-считыватель, который занимается файловым вводом-выводом. Файл обернут объектом `BufferedReader`, который позволяет читать построчно. Функция `handler` представляет код, который мы хотели бы применить к прочитанным строкам. Она принимает в качестве аргумента поток `Stream` строк файла. Метод принимает и еще одну функцию-обработчик `error`, которая вызывается в случае исключения ввода-вывода. Она конструирует

нужное нам «предметное» исключение, которое и возбуждается методом при обнаружении ошибки.

Подведем итог. Функции высшего порядка обеспечивают инверсию управления – одну из форм инверсии зависимости. Их легко использовать в сочетании с лямбда-выражениями. Говоря о принципе инверсии зависимости, стоит также отметить, что абстракция, от которой мы зависим, не обязана быть интерфейсом. В данном случае в качестве абстракции чтения и обработки файла выступал существующий класс `Stream`. Такой подход согласуется со способом управления ресурсами, принятым в функциональных языках программирования, – обычно ресурсом управляет функция высшего порядка, которая принимает функцию обратного вызова, применяет ее к открытому ресурсу, после чего ресурс закрывается. На самом деле если бы во времена Java 7 уже были доступны лямбда-выражения, то конструкцию `try-с-ресурсами`, наверное, можно было бы реализовать в виде одной библиотечной функции.

Что еще почитать

В этой главе затрагивались общие вопросы проектирования, мы говорили скорее о программе в целом, нежели о локальных проблемах, относящихся к отдельным методам. Но поскольку книга посвящена лямбда-выражениям, то мы лишь едва затронули эту тему. Для тех, кого интересуют подробности, существует немало достойных книг, в которых рассматриваются эти и смежные проблемы.

Принципы SOLID уже давно популяризируются «дядюшкой» Бобом Мартином, который много писал и выступал на эту тему. Если вы хотите приобрести его знания задаром, почитайте серию статей, посвященных этим принципам, на сайте компании Object Mentor (*http://www.objectmentor.com/resources/publishedArticles.html*) в разделе «Design Patterns».

Тем, кто хотел бы глубже разобраться в предметно-ориентированных языках, внешних и внутренних, рекомендую книгу Martin Fowler, Rebecca Parsons «Domain-Specific Languages»[1] (издательство Addison-Wesley).

---

[1] Фаулер М. Предметно-ориентированные языки программирования. – М.: Вильямс, 2011.

Основные моменты

- Лямбда-выражения позволяют упростить и сделать более понятными многие существующие паттерны проектирования, особенно паттерн Команда.
- Java 8 предоставляет большую гибкость при разработке предметно-ориентированных языков.
- В Java 8 появились новые возможности для применения принципов SOLID.

Глава 9

Конкурентное программирование и лямбда-выражения

Я уже затрагивал тему параллелизма по данным, а в этой главе покажу, как можно использовать лямбда-выражения для написания конкурентных приложений, которые эффективно обмениваются сообщениями и выполняют неблокирующий ввод-вывод.

Некоторые приведенные в этой главе примеры написаны с использованием каркасов Vert.x и RxJava. Но принципы являются достаточно общими и применимы также к другим каркасам, а равно и к вашему собственному коду – прибегать к каким-то каркасам вообще необязательно.

Зачем нужен неблокирующий ввод-вывод?

Говоря о параллелизме, я уделил много внимания эффективному использованию нескольких ядер. Такой подход, безусловно, полезен, но это не единственная потоковая модель, применяемая для обработки больших объемов данных.

Предположим, что нужно написать чат, способный обслуживать очень много пользователей. Всякий раз как в чат входит новый пользователь, открывается TCP-соединение с сервером. Если придерживаться традиционной потоковой модели, то каждый раз, как возникает необходимость передать что-то пользователю, нужно будет вызвать метод отправки данных. Этот метод заблокирует поток, в котором вызван.

Такой подход называется *блокирующим вводом-выводом*, он широко распространен и прост для понимания, потому что взаимодействие с пользователем происходит строго последовательно и легко

прослеживается в программе. Недостаток же в том, что при увеличении количества пользователей приходится запускать много потоков на сервере для их обслуживания. Это решение плохо масштабируется.

Неблокирующий – или, как его часто называют, *асинхронный* – ввод-вывод можно использовать, когда требуется обработать много одновременных соединений, не резервируя для каждого соединения отдельного потока. В отличие от блокирующего ввода-вывода, методы, которые читают и отправляют данные клиентам чата, возвращают управление немедленно. Собственно ввод-вывод происходит в каком-то другом потоке, а вы тем временем можете заняться полезной работой. Как использовать освободившиеся циклы процессора, решать вам: можно прочитать больше данных, отправленных клиентом, а можно запустить на том же оборудовании игровой сервер Minecraft!

До сих пор я избегал кода, демонстрирующего эти идеи, потому что есть много способов построить API для неблокирующего ввода-вывода. В стандартной библиотеке Java неблокирующий ввод-вывод представлен в виде подсистемы NIO (New I/O). В первоначальной версии NIO использовалась идея селектора (класс `Selector`), который позволял одному потоку выполнения управлять несколькими каналами связи, например сетевыми сокетами, через которые посылаются данные клиентам чата.

Этот подход никогда не пользовался особой популярностью у пишущих на Java, поскольку получающийся код было довольно трудно понять и отлаживать. С пришествием лямбда-выражений появился идиоматический способ проектирования и разработки API, не страдающих такими недостатками.

Обратные вызовы

Чтобы продемонстрировать принципиальную основу нового подхода, мы напишем предельно простое приложение для чата – без всяких «бантиков и рюшечек». Пользователи просто смогут посылать друг другу сообщения и получать их. При первом подключении пользователь должен выбрать себе имя.

Мы реализуем это приложение с помощью каркаса Vert.x и по ходу дела познакомимся с применяемыми техническими приемами. Начнем с кода, который получает запросы на создание нового TCP-соединения.

Пример 9.1 ❖ Получение запроса на создание TCP-соединения

```
public class ChatVerticle extends Verticle {

    public void start() {
        vertx.createNetServer()
            .connectHandler(socket -> {
                container.logger().info("socket connected");
                socket.dataHandler(new User(socket, this));
            }).listen(10_000);

        container.logger().info("ChatVerticle started");
    }

}
```

Объект `Verticle` можно считать отдаленным аналогом сервлета, это атомарная единица развертывания в каркасе Vert.x. Точкой входа в программу является метод `start`, аналогичный методу `main` в обычной программе на Java. В нашем чате мы используем его только для настройки сервера, принимающего запросы на создание TCP-соединений.

Мы передаем лямбда-выражение методу `connectHandler`, который будет вызываться при каждом подключении к чату. Это обратный вызов, который работает, по сути дела, так же, как обратные вызовы в Swing, о которых я рассказывал в главе 1. Достоинство такого подхода в том, что приложению нет дела до потоковой модели, – управлением потоками со всей сопутствующей сложностью занимается каркас Vert.x, а на нашу долю остается продумывание событий и обратных вызовов.

Приложение регистрирует еще один обратный вызов с помощью метода `dataHandler`. Он вызывается, когда нужно прочитать какие-то данные из сокета. В подобном случае мы хотим предоставить более сложную функциональность, поэтому вместо лямбда-выражения передаем экземпляр обычного класса `User`, который реализует необходимый функциональный интерфейс. Обратный вызов класса `User` показан в примере 9.2.

Пример 9.2 ❖ Обработка получения данных от пользователя

```
public class User implements Handler<Buffer> {

    private static final Pattern newline = Pattern.compile("\\n");

    private final NetSocket socket;
    private final Set<String> names;
```

```
private final EventBus eventBus;

private Optional<String> name;

public User(NetSocket socket, Verticle verticle) {
    Vertx vertx = verticle.getVertx();

    this.socket = socket;
    names = vertx.sharedData().getSet("names");
    eventBus = vertx.eventBus();
    name = Optional.empty();
}

@Override
public void handle(Buffer buffer) {
    newline.splitAsStream(buffer.toString())
           .forEach(line -> {
                if (!name.isPresent())
                    setName(line);
                else
                    handleMessage(line);
            });
}

// Продолжение класса ...
```

Буфер `buffer` содержит данные, полученные из сетевого соединения. Мы пользуемся текстовым протоколом, в котором поля отделяются друг от друга знаками новой строки, поэтому необходимо преобразовать поступившие данные в объект `String`, а затем разбить эту строку на части в местах, где находятся знаки новой строки.

Мы завели регулярное выражение `newline`, сопоставляемое со знаками новой строки, – экземпляр класса `java.util.regex.Pattern`. В Java 8 в класс `Pattern` добавлен метод `splitAsStream`, который позволяет разбивать строку `String` с помощью регулярного выражения и порождать поток значений, совпадающих с выделенными подстроками.

Первое, что делает пользователь, подключившийся к серверу чата, – выбирает себе имя. Если мы не знаем имя пользователя `name`, то активируем логику задания имени, в противном случае обрабатываем сообщение как обычно.

Нам также необходим способ получать сообщения от других пользователей и передавать их клиенту чата для прочтения получателем. С этой целью в момент, когда текущий пользователь выбирает себе имя, мы регистрируем еще один обратный вызов – тот, что будет отправлять сообщения (пример 9.3).

Пример 9.3 ❖ Подписка на сообщения в чате

```
eventBus.registerHandler(name, (Message<String> msg) -> {
    sendClient(msg.body());
});
```

Здесь мы пользуемся *шиной событий* каркаса Vert.x – механизмом, который позволяет узлам `Verticle` обмениваться сообщениями без блокировки (см. рис. 9.1). Метод `registerHandler` ассоциирует обработчик с определенным адресом, так что в случае отправки сообщения на указанный адрес происходит вызов этого обработчика с сообщением в качестве аргумента. Здесь адресом служит имя пользователя.

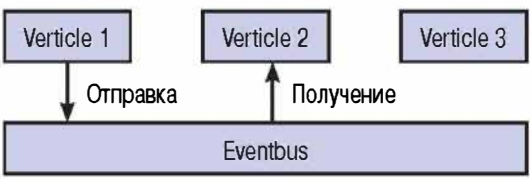

Рис. 9.1 ❖ Отправка с помощью шины событий

Благодаря регистрации обработчиков для адресов и отправки им сообщений мы можем построить весьма развитый и (или) слабо связанные наборы служб, реагирующих на события без какой-либо блокировки выполнения программы. Отметим, что в этом дизайне нет никакого общего состояния.

Шина событий Vert.x позволяет передавать сообщения разных типов, но все они обертываются объектом `Message`. Двухточечный обмен сообщениями реализуется самими объектами `Message`; в объекте-отправителе может храниться обработчик ответа. Поскольку в случае чата нам необходимо фактическое тело сообщения, то есть его текст, то мы просто вызываем метод `body`. Это текстовое сообщение отправляется пользователю-получателю путем передачи по TCP-соединению.

Если приложение хочет отправить сообщение одного пользователя другому, то оно передает сообщение на адрес, представляющий второго пользователя (пример 9.4). И снова адресом является имя пользователя.

Пример 9.4 ❖ Отправка сообщения пользователю чата

```
eventBus.send(user, name.get() + '>' + message);
```

Немного усложним этот простейший чат-сервер, добавив широковещательную рассылку сообщений и «поклонников» (follower). Для этого нам понадобятся две новые команды.

Восклицательный знак будет обозначать команду широковещания, которая посылает следующий за ней текст всем поклонникам. Например, если пользователь bob введет команду «!hello followers», то все его поклонники получат сообщение «bob>hello followers».

Команда follow, которая делает выполнившего ее поклонником указанного вслед за ней пользователя, например: «follow bob».

Далее нам предстоит реализовать методы `broadcastMessage` и `followUser`, соответствующие этим командам.

В данном случае необходим другой тип связи. Вместо того чтобы посылать сообщение одному пользователю, мы должны опубликовать его для нескольких пользователей. К счастью, шина событий Vert.x позволяет публиковать сообщение, адресованное нескольким обработчикам (рис. 9.2). И, таким образом, реализация оказывается похожей на предыдущую.

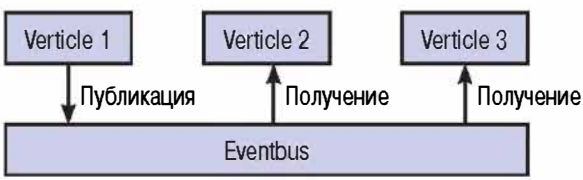

Рис. 9.2 ❖ Публикация с помощью шины событий

Единственное различие состоит в том, что вместо метода `send` шины событий мы вызываем метод `publish`. Чтобы избежать конфликтов с существующими адресами в случае, когда пользователь вводит команду `!`, сообщение публикуется на адрес, составленный из имени пользователя и суффикса `.followers`. Например, сообщение, опубликованное пользователем bob, передается обработчику, зарегистрированному для адреса `bob.followers` (пример 9.5).

Пример 9.5 ❖ Широковещательная рассылка сообщений поклонникам

```
private void broadcastMessage(String message) {
    String name = this.name.get();
    eventBus.publish(name + ".followers", name + '>' + message);
}
```

Что же касается обработчика, то мы хотим выполнить в нем ту же операцию, что и в зарегистрированном ранее для передачи: переправить сообщение клиенту (пример 9.6).

Пример 9.6 ❖ Прием широковещательных сообщений

```
private void followUser(String user) {
    eventBus.registerHandler(user + ".followers", (Message<String> message)->{
        sendClient(message.body());
    });
}
```

Если мы отправляем сообщение на адрес, который прослушивают несколько обработчиков, то используется циклический селектор, который решает, какой обработчик получит сообщение. Это означает, что при регистрации адресов необходимо проявлять осторожность.

Архитектуры на основе передачи сообщений

Выше я описал архитектуру на основе передачи сообщений и реализовал ее на примере простого клиента чата. Детали этого клиента не так интересны, как общий принцип, поэтому поговорим о передаче сообщений как таковой.

Прежде всего отметим, что в этом дизайне нет никакого общего состояния. Коммуникация между узлами `Verticle` производится путем отправки сообщений через шину событий. Это означает, что нам не нужно думать о защите общего состояния, а следовательно, ни к чему всякого рода блокировки или ключевое слово `synchronized`. Механизм обеспечения конкурентности существенно упрощается.

Дабы гарантировать отсутствие общего состояния, разделяемого несколькими узлами `Verticle`, мы наложили ряд ограничений на типы сообщений, передаваемых по шине событий. В данном случае в качестве сообщений фигурировали простые строки Java. Они неизменяемы, а значит, их можно безопасно передавать между узлами `Verticle`. Не будучи в силах изменить состояние `String`, обработчик-получатель не может как-то повлиять на поведение отправителя.

Vert.x не требует, чтобы передавались только строки; можно обмениваться и более сложными JSON-объектами и даже конструировать собственные двоичные сообщения с помощью класса `Buffer`. Но эти сообщения не являются неизменяемыми, а значит, если наивно попытаться передавать их, то у отправителей и обработчиков сообщений может ненароком образоваться общее состояние, если они станут что-то записывать в сообщение.

Каркас Vert.x решает эту проблему, копируя любое изменяемое сообщение в момент его отправки. Таким образом, получатель ви-

дит правильное значение, и никакого обобществления состояния не происходит. Не важно, пользуетесь вы каркасом Vert.x или нет, нельзя допускать, чтобы сообщения случайно стали источником общего состояния. Проще всего добиться этого, когда сообщения в принципе неизменяемы, но и копирование сообщения дает нужный результат.

Модель, основанная на узлах `Verticle`, позволяет к тому же реализовать конкурентную систему, пригодную для тестирования. Связано это с тем, что каждый узел можно тестировать изолированно, посылая ему сообщения и сравнивая возвращенный результат с ожидаемым. Из таких независимо протестированных компонентов мы затем можем собрать систему, не создавая многочисленных проблем, которые неизбежно появились бы, если бы компоненты взаимодействовали с помощью изменения общего состояния. Разумеется, сквозные тесты по-прежнему необходимы, так как позволяют убедиться, что система делает именно то, что от нее ожидает пользователь!

Системы на основе передачи сообщений лучше и с точки зрения простоты изоляции ошибок и написания надежного кода. Если в обработчике сообщения имеется ошибка, то достаточно перезапустить локальный узел `Verticle`, а не всю виртуальную машину Java.

В главе 6 мы видели, как лямбда-выражения применяются в сочетании с потоковой библиотекой для распараллеливания программы по данным. Это позволяет использовать параллелизм для ускорения обработки больших объемов данных. Передача сообщений и реактивное программирование, о котором мы будем говорить далее, составляют другой край спектра. Тут мы сталкиваемся с конкурентными ситуациями, в которых единиц работы по вводу-выводу, например клиентов чата, гораздо больше, чем параллельно исполняемых потоков. В обоих случаях решение одно и то же: использовать лямбда-выражения для представления поведения и проектировать API, которые берут на себя управление конкурентностью. Чем изощреннее библиотека, тем проще код приложения.

Пирамида судьбы

Мы уже видели, как с помощью обратных вызовов и событий создать неблокирующий конкурентный код, но слона-то и не приметили. Код, изобилующий обратными вызовами, очень трудно читать, даже если использовать лямбда-выражения. Рассмотрим конкретный пример, чтобы лучше понять, в чем состоит проблема.

В ходе разработки чат-сервера я написал ряд тестов, описывающих поведение узла Verticle с точки зрения клиента. Один из таких тестов – messageFriend – приведен в примере 9.7.

Пример 9.7 ❖ Тест проверяет, могут ли общаться в чате два приятеля

```
@Test
public void messageFriend() {
    withModule(() -> {
        withConnection(richard -> {
            richard.dataHandler(data -> {
                assertEquals("bob>oh its you!", data.toString());
                moduleTestComplete();
            });

            richard.write("richard\n");
            withConnection(bob -> {
              bob.dataHandler(data -> {
                  assertEquals("richard>hai", data.toString());
                  bob.write("richard<oh its you!");
              });
              bob.write("bob\n");
              vertx.setTimer(6, id -> richard.write("bob<hai"));
            });
        });
    });
}
```

Я соединяю двух клиентов, richard и bob, затем richard говорит «hai» bob'y, а bob отвечает «oh it's you!». Я вынес отдельно общий код создания соединения, но все равно, как видите, вложенные обратные вызовы начинают выстраиваться в *пирамиду судьбы*. Они сдвигаются к правому краю экрана – как пирамида, положенная на боковую грань (не крутите пальцем у виска – не я придумал эту метафору!). Это хорошо известный антипаттерн, затрудняющий чтение и понимание кода. Ко всему прочему логика программы оказывается размазанной между несколькими методами.

В предыдущей главе мы обсуждали, как лямбда-выражения позволяют управлять ресурсами: достаточно передать лямбда-выражение в метод with. В этом тесте я два раза воспользовался таким приемом. У нас есть метод withModule, который развертывает текущий модуль Vert.x, исполняет какой-то код и останавливает модуль. Есть также метод withConnection, который устанавливает соединение с ChatVerticle, а закончив работу с ним, закрывает.

В данном случае вызовы методов with обладают, по сравнению с конструкцией try-с-ресурсами, тем преимуществом, что хорошо со-

гласуются с потоковой моделью, используемой в этой главе. В примере 9.8 показано, как можно переработать этот код, сделав его понятнее.

Пример 9.8 ❖ Тест, проверяющий возможность общения двух приятелей в чате, разбит на несколько методов

```
@Test
public void canMessageFriend() {
    withModule(this::messageFriendWithModule);
}

private void messageFriendWithModule() {
    withConnection(richard -> {
        checkBobReplies(richard);
        richard.write("richard\n");
        messageBob(richard);
    });
}

private void messageBob(NetSocket richard) {
    withConnection(messageBobWithConnection(richard));
}

private Handler<NetSocket> messageBobWithConnection(NetSocket richard) {
    return bob -> {
        checkRichardMessagedYou(bob);
        bob.write("bob\n");
        vertx.setTimer(6, id -> richard.write("bob<hai"));
    };
}

private void checkRichardMessagedYou(NetSocket bob) {
    bob.dataHandler(data -> {
        assertEquals("richard>hai", data.toString());
        bob.write("richard<oh its you!");
    });
}

private void checkBobReplies(NetSocket richard) {
    richard.dataHandler(data -> {
        assertEquals("bob>oh its you!", data.toString());
        moduleTestComplete();
    });
}
```

Агрессивный рефакторинг, показанный в примере 9.8, решил проблему пирамиды судьбы, но ценой разбиения единой логики теста на несколько методов. Теперь у нас на руках не один метод, обладающий

единственной обязанностью, а несколько методов, сообща исполняющих одну и ту же обязанность! Наш код по-прежнему трудно читать, хотя и по другой причине.

Чем больше операций в цепочке, тем острее оказывается проблема. Необходимо решение получше.

Будущие результаты

Еще один способ построения сложных последовательностей конкурентных операций – воспользоваться классом будущих результатов `Future`. Объект `Future` – это долговая расписка на значение. Метод возвращает не само значение, а обязательство вернуть его в будущем – объект `Future`. В момент создания объект `Future` не имеет значения, но впоследствии его можно обменять на значение – точно так же, как долговую расписку можно обменять на деньги.

Чтобы извлечь значение из объекта `Future`, нужно вызвать его метод `get`, который блокирует выполнение, до тех пор пока значение не будет готово. К сожалению, будущие результаты страдают теми же проблемами композиции, что и обратные вызовы. Вкратце рассмотрим, с какими неприятностями можно столкнуться.

Нас будет интересовать задача поиска информации о музыкальном альбоме `Album` с помощью внешних веб-служб. Требуется найти список произведений, включенных в данный альбом, и список исполнителей. Кроме того, для доступа к службам необходимо иметь соответствующие права, а для проверки их наличия представить свои учетные данные. Итак, мы должны аутентифицироваться или хотя бы работать в контексте уже аутентифицированного сеанса.

В примере 9.9 показана реализация этой задачи с применением существующего API класса `Future`. Сначала (точка ❶) мы аутентифицируемся в службах произведений и исполнителей. Обе операции аутентификации возвращают объекты `Future<Credentials>`, содержащие информацию о результате. Интерфейс `Future` универсальный, поэтому `Future<Credentials>` можно рассматривать как долговую расписку на получение объекта `Credentials`.

Пример 9.9 ❖ Получение информации об альбоме от внешних веб-служб с помощью объектов Future

```
@Override
public Album lookupByName(String albumName) {
    Future<Credentials> trackLogin = loginTo("track");  ❶
    Future<Credentials> artistLogin = loginTo("artist");
```

```
    try {
      Future<List<Track>> tracks = lookupTracks(albumName, trackLogin.get()); ❷
      Future<List<Artist>> artists = lookupArtists(albumName,
                                                   artistLogin.get());
      return new Album(albumName, tracks.get(), artists.get()); ❸
    } catch (InterruptedException | ExecutionException e) {
        throw new AlbumLookupException(e.getCause()); ❹
    }
}
```

В точке ❷ мы обращаемся за произведениями и исполнителями, передавая результат аутентификации, полученный из объектов `Future` с помощью метода `get`. В точке ❸ мы конструируем возвращаемый объект `Album`, вновь вызывая `get` и блокируя тем самым существующие объекты `Future`. Возникшие исключения передаются выше в виде предметного исключения в точке ❹.

Вы наверняка заметили, что, желая передать результат объекта `Future` следующей операции, мы блокируем поток выполнения. Это может стать причиной снижения производительности, потому что работа в целом выполняется не параллельно, как было задумано, а последовательно.

А это означает, что в примере 9.9 мы сможем обратиться хотя бы к одной веб-службе поиска не раньше, чем аутентифицируемся в обеих. Но это же лишнее ограничение: метод `lookupTracks` должен дождаться только завершения аутентификации в службе поиска произведений, а метод `lookupArtists` – в службе поиска исполнителей. На рис. 9.3 показано, кто ждет завершения каких действий.

Мы могли бы перенести блокирующие вызовы `get` внутрь методов `lookupTracks` и `lookupArtists`. Это решило бы проблему, но код полу-

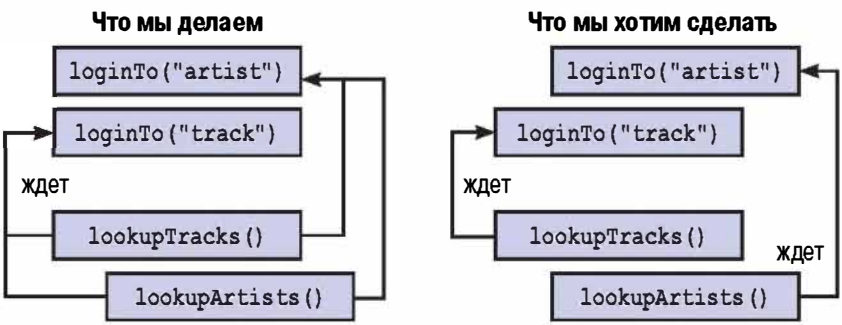

Рис. 9.3 ❖ Ни одно действие поиска не должно дожидаться завершения обоих действий аутентификации

чился бы безобразным, и мы утратили бы возможность повторно использовать результаты аутентификации в нескольких местах.

В действительности нам необходим способ начать действовать, после того как результат объекта Future получен, *не* обращаясь при этом к блокирующему методу get. Требуется объединить Future с обратным вызовом.

Завершаемые будущие результаты

Решение всех вопросов дает класс CompletableFuture. Он объединяет идею долговой расписки, заложенную в классе Future, с использованием обратных вызовов для организации событийно-управляемой программы. Главное в классе CompletableFuture – тот факт, что различные экземпляры можно компоновать, не образуя пирамиду судьбы.

Возможно, вы уже сталкивались с концепцией, лежащей в основе CompletableFuture; в других языках она называется *отложенный объект*, или *обещание*. В библиотеке Google Guava и в каркасе Spring используется название ListenableFuture.

В примере 9.10 я переписал пример 9.9, проиллюстрировав некоторые способы использования объектов CompletableFuture вместо Future.

Пример 9.10 ❖ Получение информации об альбоме от внешних веб-служб с помощью объектов CompletableFuture

```
public Album lookupByName(String albumName) {
    CompletableFuture<List<Artist>> artistLookup
        = loginTo("artist")
          .thenCompose(artistLogin -> lookupArtists(albumName, artistLogin)); ❶

    return loginTo("track")
            .thenCompose(trackLogin -> lookupTracks(albumName, trackLogin)) ❷
            .thenCombine(artistLookup, (tracks, artists)
                -> new Album(albumName, tracks, artists)) ❸
            .join(); ❹
}
```

В примере 9.10 методы loginTo, lookupArtists и lookupTracks возвращают объект CompletableFuture, а не Future. Ключевой «фокус» в API класса CompletableFuture – регистрация лямбда-выражений и сцепление функций высшего порядка. Методы другие, но сама концепция поразительно напоминает дизайн потокового API.

В точке ❶ мы используем метод thenCompose, чтобы преобразовать объект Credentials в CompletableFuture, который содержит исполните-

лей. Это, по сути, то же самое, что взять у приятеля долговую расписку и, получив по ней деньги впоследствии, потратить их на сайте Amazon. Но купленную книгу вы получаете не сразу, вместо этого от Amazon приходит письмо с сообщением, что книга выслана, – еще один вариант долговой расписки.

В точке ❷ мы снова используем метод `thenCompose` и объект `Credentials` от API аутентификации в службе произведений, чтобы создать объект `CompletableFuture`, содержащий произведения. В точке ❸ мы видим еще один метод: `thenCombine`. Он принимает результат от `CompletableFuture` и комбинирует его с другим `CompletableFuture`. Операция-комбинатор предоставляется пользователем в виде лямбда-выражения. Мы хотим сконструировать объект `Album` из произведений и исполнителей, в этом и заключается действие комбинатора.

Сейчас стоит вспомнить, что, как и в потоковом API, мы не выполняем операции немедленно, а лишь составляем рецепт, в котором записано, как их нужно выполнить. Наша методика не может гарантировать, что `CompletableFuture` завершился, пока не будет вызван какой-нибудь финальный метод. Поскольку класс `CompletableFuture` реализует интерфейс `Future`, мы могли бы просто вызвать метод `get`. Но в классе `CompletableFuture` имеется метод `join`, который делает то же самое; его мы и вызываем в точке ❹. В этом случае нет противной проверки исключений, загромождающей код с участием `get`.

Надо полагать, вы уловили основную идею использования объектов `CompletableFuture`, но создание их – совершенно другое дело. У создания `CompletableFuture` есть две стороны: собственно создание объекта и его завершение, то есть запись в него значения, которое он обещал вернуть клиентскому коду.

Из примера 9.11 видно, что создать объект `CompletableFuture` довольно легко. Нужно просто вызвать его конструктор! Клиентский код может использовать этот объект для сцепления операций. Мы также сохраняем ссылку на созданный объект, чтобы можно было завершить работу в другом потоке.

Пример 9.11 ❖ Завершение будущего результата путем записи в него значения

```
CompletableFuture<Artist> createFuture(String id) {
    CompletableFuture<Artist> future = new CompletableFuture<>();
    startJob(future);
    return future;
}
```

Выполнив необходимую работу в том потоке, который используем, мы должны сообщить объекту `CompletableFuture`, какое значение он представляет. Напомним, что выполнить работу можно разными способами. Например, мы можем отправить (методом `submit`) задание службе `ExecutorService`, воспользоваться системой на базе цикла обработки событий типа Vert.x или просто запустить поток `Thread` и сделать в нем всё, что необходимо. В примере 9.12 показано, что для уведомления объекта `CompletableFuture` о его готовности необходимо вызвать метод `complete`. Пришло время платить по долговой расписке.

Пример 9.12 ❖ Завершение будущего результата путем записи в него значения

```
future.complete(artist);
```

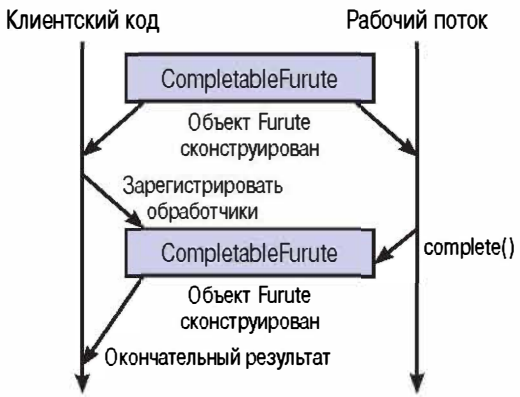

Рис. 9.4 ❖ Завершаемый будущий результат – это долговая расписка, которая может быть обработана обработчиками

Разумеется, широко распространенный способ применения объектов `CompletableFuture` – асинхронное выполнение блока кода. По завершении код возвращает некоторое значение. Чтобы разные люди не занимались реализацией одного и того же кода снова и снова, существует полезный фабричный метод `supplyAsync` для создания объекта `CompletableFuture`, его применение показано в примере 9.13.

Пример 9.13 ❖ Асинхронное создание объекта CompletableFuture

```
CompletableFuture<Track> lookupTrack(String id) {
    return CompletableFuture.supplyAsync(() -> {
        // Здесь выполняется какая-то длительная операция ❶
```

```
        // ...
        return track;  ❷
    ), service);  ❸
}
```

Метод `supplyAsync` принимает и выполняет объект `Supplier`. Ключевой момент, показанный в точке ❶, заключается в том, что `Supplier` может выполнять длительную операцию, не блокируя текущего потока, – отсюда и слово *Async* в имени метода. Значение, возвращенное в точке ❷, используется для завершения объекта `CompletableFuture`. В точке ❸ мы предоставляем объект `Executor`, который называется `service`, сообщающий `CompletableFuture`, где выполнять работу. Если `Executor` не указан, то используется тот же пул потоков с разветвлением и соединением, в котором выполняются параллельные потоки `Stream`.

Само собой, не всякая долговая расписка оплачивается. Иногда исключительные обстоятельства не дают нам заплатить свои долги. Как видно из примера 9.14, API класса `CompletableFuture` предусматривает такие ситуации, давая возможность «завершить с исключением» путем вызова метода `completeExceptionally` вместо `complete`. Вызвать для одного и того же объекта оба метода `complete` и `completeExceptionally` нельзя.

Пример 9.14 ❖ Завершение будущего результата в случае ошибки

```
future.completeExceptionally(new AlbumLookupException("Unable to find " + name));
```

Полный обзор API класса `CompletableFuture` выходит за рамки этой главы, но во многих отношениях это настоящая сокровищница. В API имеется множество полезных методов для компоновки и комбинирования экземпляров `CompletableFuture` всеми мыслимыми способами. К тому же теперь вы уже достаточно знакомы с текучим стилем сцепления функций высшего порядка, позволяющим сообщить компьютеру, что делать.

Вкратце рассмотрим несколько сценариев.

- Если вы хотите завершить цепочку блоком кода, который ничего не возвращает, например `Consumer` или `Runnable`, обратите внимание на методы `thenAccept` и `thenRun`.
- Для преобразования значения объекта `CompletableFuture` по аналогии с методом `map` интерфейса `Stream` можно воспользоваться методом `thenApply`.
- Если вы хотите обработать ситуации, в которых объект `CompletableFuture` завершился с исключением, то метод `exceptio-`

nally позволит восстановиться после ошибки, зарегистрировав функцию, предоставляющую альтернативное значение.

- Если требуется выполнить map, приняв во внимание как нормальное завершение, так и возможность исключения, воспользуйтесь методом handle.
- Для выяснения того, что случилось с объектом CompletableFuture, к вашим услугам методы isDone и isCompletedExceptionally.

Класс CompletableFuture весьма полезен для организации конкурентной работы, но это не единственный игрок на поле. Далее мы рассмотрим родственную идею, которая предлагает несколько большую гибкость ценой усложнения кода.

Реактивное программирование

Концепцию, лежащую в основе класса CompletableFuture, можно обобщить с одиночных значений на потоки данных, с помощью идеи *реактивного программирования*. По существу, это вид декларативного программирования, который позволяет программировать в терминах автоматически распространяемых изменений и потоков данных.

В качестве типичного примера реактивного программирования можно вспомнить электронную таблицу. Если в ячейку C1 ввести формулу =B1+5, то это будет означать, что таблица должна прибавить 5 к содержимому ячейки B1 и поместить результат в C1. Кроме того, электронная таблица будет реагировать на любые изменения в B1, обновляя значение в C1.

Для переноса идей реактивного программирования на виртуальную машину Java написана библиотека RxJava. Здесь мы не будем изучать ее сколько-нибудь подробно, а остановимся только на основных концепциях.

В библиотеке RxJava имеется класс Observable, который представляет последовательность событий, на которую программа может отреагировать. Это долговая расписка для последовательности. Существует явственная связь между классом Observable и интерфейсом Stream, рассмотренным в главе 3.

В обоих случаях мы составляем рецепт, сцепляя функции высшего порядка и используя лямбда-выражения, чтобы ассоциировать поведение с операциями общего вида. На самом деле многие операции, определенные в классе Observable, совпадают с методами интерфейса Stream: map, filter, reduce.

Существенное различие между обоими подходами – сценарий, в котором они используются. Потоки призваны построить последовательность операций обработки коллекций в памяти. Что же касается библиотеки RxJava, то она предназначена для компоновки и выстраивания последовательностей в асинхронных и событийно-ориентированных системах. Данные не вытягиваются, а заталкиваются. Можно также считать, что RxJava относится к последовательности значений точно так же, как `CompletableFuture` относится к одному значению.

На этот раз мы в качестве конкретного примера рассмотрим поиск исполнителя (пример 9.15). Метод `search` фильтрует результаты по имени и национальности. Он хранит локальный кэш имен исполнителей, но прочую информацию, например национальность, должен запрашивать у внешних служб.

Пример 9.15 ❖ Поиск исполнителя по имени и национальности

```
public Observable<Artist> search(String searchedName,
                                 String searchedNationality,
                                 int maxResults) {
    return getSavedArtists()  ❶
            .filter(name -> name.contains(searchedName))  ❷
            .flatMap(this::lookupArtist)  ❸
            .filter(artist -> artist.getNationality()  ❹
                                .contains(searchedNationality))
            .take(maxResults);  ❺
}
```

В точке ❶ мы получаем объект `Observable`, содержащий сохраненные имена исполнителей. Функции высшего порядка в классе `Observable` аналогичны определенным в интерфейсе `Stream`, поэтому в точках ❷ и ❹ мы можем отфильтровать `Observable` по имени и национальности по аналогии с тем, как это делается для потоков типа `Stream`.

В точке ❸ мы заменяем каждое имя объектом `Artist`. Если бы это сводилось просто к вызову конструктора, то мы, очевидно, могли бы воспользоваться операцией `map`. Но в данном случае нам необходимо скомпоновать последовательность обращений к внешним веб-службам, каждое из которых может осуществляться в своем потоке или в потоке, взятом из пула. Следовательно, мы должны заменить каждое имя объектом `Observable`, представляющим одного или нескольких исполнителей. Поэтому используется операция `flatMap`.

Мы также хотим ограничить количество результатов поиска величиной `maxResults`. Для этого служит метод `take` объекта `Observable`.

Как видите, API очень похож на потоковый. Существенное различие заключается в том, что интерфейс `Stream` предназначен для вычисления окончательных результатов, а API библиотеки RxJava больше напоминает класс `CompletableFuture` в части потоковой модели.

В классе `CompletableFuture` мы должны были оплатить долговую расписку вызовом метода `complete`, который записывает значение. Поскольку `Observable` представляет поток событий, нам необходима возможность затолкнуть несколько значений. Как это делается, показано в примере 9.16.

Пример 9.16 ❖ Заталкивание значений в объект Observable и его завершение

```
observer.onNext("a");
observer.onNext("b");
observer.onNext("c");
observer.onCompleted();
```

Мы вызываем метод `onNext` по одному разу для каждого элемента, хранящегося в `Observable`. Это можно сделать в цикле и в любом потоке выполнения, в котором мы собираемся порождать значения. Закончив работу по генерации событий, мы вызываем метод `onCompleted`, чтобы сообщить `Observable` о завершении. Как и в потоковой библиотеке, имеется несколько вспомогательных фабричных методов для создания объектов `Observable` из будущих результатов, итерируемых объектов и массивов.

Как и в случае `CompletableFuture`, API класса `Observable` позволяет завершать работу с исключением. Чтобы сигнализировать об ошибке, в нашем распоряжении имеется метод `onError` (см. пример 9.17). Функциональность, правда, несколько отличается от `CompletableFuture` – мы можем получить все события, предшествующие возникновению исключения, но в обоих случаях завершение происходит нормально или с исключением.

Пример 9.17 ❖ Уведомление объекта Observable об ошибке

```
observer.onError(new Exception());
```

Как и в случае `CompletableFuture`, я лишь в общих чертах описал порядок работы с API класса `Observable`. Если вам интересны детали, обратитесь к полной документации по проекту (*https://github.com/Netflix/RxJava/wiki/Getting-Started*). Сейчас начинается интеграция RxJava в существующую экосистему библиотек Java. Например, в каркас интеграции корпоративных приложений Apache Camel до-

бавлен модуль Camel RX, позволяющий использовать в этом каркасе библиотеку RxJava. Началась также работа по изменению API проекта Vert.x на основе идей RxJava.

Когда и где

В этой главе я много говорил о том, как использовать событийно-ориентированные системы с неблокирующим вводом-выводом. Означает ли это, что прямо с завтрашнего дня надлежит выбросить все существующие корпоративные веб-приложения, написанные с использованием Java EE или Spring? Безусловно, нет.

Класс `CompletableFuture` и библиотека RxJava появились сравнительно недавно, с применением соответствующих идиом сопряжена дополнительная сложность. Они, конечно, проще, чем явные будущие результаты и обратные вызовы, но для многих задач традиционных методов веб-разработки с блокирующим вводом-выводом вполне достаточно. Не надо чинить то, что не сломалось.

Разумеется, я не хочу сказать, что чтение этой главы – впустую потраченный день! Популярность событийно-ориентированных реактивных приложений растет, и нередко они позволяют лучше построить модель предметной области. «Манифест реактивного программирования» (*http://www.reactivemanifesto.org/*) содержит призыв создавать больше приложений в таком стиле, и если вам это кажется правильным, действуйте. Особо отмечу два случая, когда имеет смысл рассуждать в терминах реакции на события, а не блокирующей обработки.

Первый – когда описание предметной области изначально формулируется в терминах событий. Классический пример дает Твиттер, служба подписки на потоки текстовых сообщений. Пользователи отправляют друг другу сообщения, поэтому, проектируя приложение как событийно-ориентированное, вы точно моделируете предметную область. Другой пример – приложение, которое строит график цен на акции. Каждое изменение цены можно представить в виде события.

Второй очевидный случай – ситуация, когда приложение должно выполнять много операций ввода-вывода одновременно, так что блокирующий ввод-вывод потребует запуска слишком большого количества потоков. Что, в свою очередь, приведет к созданию многочисленных блокировок и интенсивному контекстному переключению. Если требуется обслуживать тысячи и более одновременных соединений, то лучше перейти к неблокирующей архитектуре.

Основные моменты

- Событийно-ориентированная архитектура легко реализуется с помощью обратных вызовов в виде лямбда-выражений.
- Класс `CompletableFuture` представляет долговую расписку на получение значения в будущем. Лямбда-выражения позволяют легко компоновать и комбинировать объекты этого класса.
- Класс `Observable` обобщает идею класса `CompletableFuture` на потоки данных.

Упражнения

Эту главу сопровождает только одно упражнение, для его выполнения потребуется подвергнуть рефакторингу код работы с классом `CompletableFuture`. Начнем с класса `BlockingArtistAnalyzer`, показанного в примере 9.18. Он получает имена двух исполнителей, строит объекты `Artist` по именам и возвращает `true`, если в первом исполнителе больше членов, чем во втором; в противном случае возвращается `false`. Через конструктор класса внедрена служба поиска исполнителей `artistLookupService`; на поиск у нее может уйти некоторое время. Поскольку объект `BlockingArtistAnalyzer` два раза подряд блокирует выполнение программы на время обращения к службе, он может работать медленно. Наша задача в этом упражнении – повысить скорость.

Пример 9.18 ❖ Класс BlockingArtistAnalyzer сообщает своим клиентам, в каком исполнителе больше членов

```
public class BlockingArtistAnalyzer {

  private final Function<String, Artist> artistLookupService;

  public BlockingArtistAnalyzer(Function<String, Artist> artistLookupService)  {
      this.artistLookupService = artistLookupService;
  }

  public boolean isLargerGroup(String artistName, String otherArtistName) {
      return getNumberOfMembers(artistName) >
             getNumberOfMembers(otherArtistName);
  }

  private long getNumberOfMembers(String artistName) {
      return artistLookupService.apply(artistName)
                                .getMembers()
                                .count();
  }
}
```

В первой части упражнения вы должны переработать блокирующий код возврата, воспользовавшись интерфейсом обратного вызова. В данном случае он будет иметь тип `Consumer<Boolean>`. Напомню, что `Consumer` – функциональный интерфейс, который входит в состав стандартной библиотеки; он принимает значение и возвращает `void`. Если вы готовы взяться за решение, то измените класс `BlockingArtistAnalyzer`, так чтобы он реализовывал интерфейс `ArtistAnalyzer` (пример 9.19).

Пример 9.19 ❖ Интерфейс ArtistAnalyzer, который должен реализовывать класс BlockingArtistAnalyzer

```
public interface ArtistAnalyzer {
    public void isLargerGroup(String artistName,
                              String otherArtistName,
                              Consumer<Boolean> handler);
}
```

Имея API, согласованный с моделью обратных вызовов, мы можем отказаться от последовательного выполнения блокирующих операций поиска. Воспользовавшись классом `CompletableFuture`, переботайте метод `isLargerGroup`, так чтобы эти операции выполнялись конкурентно.

Глава 10

Что дальше?

Во многих отношениях Java можно считать языком, выдержавшим испытание временем. Эта платформа все еще необычайно популярна и является отличным выбором для разработки корпоративных бизнес-приложений. Существует очень много библиотек и каркасов с открытым исходным кодом для решения самых разнообразных задач, начиная с написания сложных модульных веб-приложений (Spring) и заканчивая такими базовыми вещами, как правильные операции с датой и временем (Jodatime). Диапазон инструментальных средств также не имеет себе равных: от полноценных сред IDE типа Eclipse и Intellij до систем сборки типа gradle и maven.

К сожалению, за прошедшие годы Java заработал репутацию консервативного языка, который перестал развиваться. Отчасти это связано с его длительной популярностью; хорошо знакомое перестает вызывать уважение. Впрочем, эволюция Java действительно протекала не без проблем. Решение во что бы ни стало сохранить обратную совместимость, несмотря на все свои достоинства, тому немало способствовало.

По счастью, выход Java 8 – это не просто косметическое улучшение языка, но новый этап в его развитии. В отличие от Java 6 и 7, эта версия не сводится к мелким усовершенствованиям в библиотеках. Я ожидаю и надеюсь, что в будущих версиях Java взятый темп не будет потерян. И это не просто потому, что мне так нравится писать книги на эту тему! Я действительно думаю, что впереди длинный путь, который приведет к достижению основной цели программирования: писать код, который было бы легко читать, назначение которого очевидно с первого взгляда и который позволяет обеспечить высокую производительность. Я сожалею лишь, что в этой заключительной главе недостаточно места, чтобы подробно рассмотреть потенциальные возможности будущих версий.

Мы подошли к концу книги, но я надеюсь, что это не конец вашего знакомства с Java 8. Я рассмотрел много способов применения лямбда-выражений: улучшенный библиотечный код работы с коллекция-

ми, параллелизм по данным, написание более простого и чистого кода и, наконец, конкурентные программы. Я ответил на вопросы, почему, что и как, но практическое использование остается за вами. Поэтому я предлагаю ряд упражнений, на которые нет ответа. Их выполнение поможет вам лучше усвоить изложенный материал.

- Объясните другому программисту, что такое лямбда-выражения и почему они заслуживают внимания. Собеседником может быть ваш приятель или коллега.
- Попробуйте развернуть программу, над которой работаете, на платформе Java 8. Если у вас уже есть автономные тесты, работающие в системе непрерывной интеграции Jenkins CI, то будет очень просто прогнать сборку для разных версий Java.
- Начните перерабатывать унаследованный код реальной системы с использованием потоков и коллекторов. Это может быть интересный вам проект с открытым исходным кодом или даже программа, над которой вы работаете, если ее тестовое развертывание прошло успешно. Если вы еще не готовы принять окончательное решение, то, быть может, стоит начать с создания прототипа в отдельной ветке.
- Сталкиваетесь ли вы с проблемами конкурентности или с обработкой больших объемов данных? Если да, попробуйте заняться рефакторингом и воспользоваться либо потоками для распараллеливания по данным, либо новыми средствами распараллеливания, появившимися в библиотеке RxJava и в классе `CompletableFuture`.
- Проанализируйте дизайн и архитектуру хорошо знакомой вам программной системы.
  - Можно ли написать лучше на макроуровне?
  - Можно ли упростить ее дизайн?
  - Можете ли вы уменьшить объем кода, необходимого для реализации определенной функции?
  - Можно ли улучшить удобочитаемость кода?

Алфавитный указатель

I

J

L

M

N

O

P

Л

М

Н

О

П

Р

С

Т

У

Ф

Ц

Ш

Э

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Ричард Уорбэртон

Лямбда-выражения в Java 8

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16 .
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 32. Тираж 200 экз.
№

Веб-сайт издательства: www.дмк.рф