O'REILLY®

От Java к Kotlin

Руководство по рефакторингу

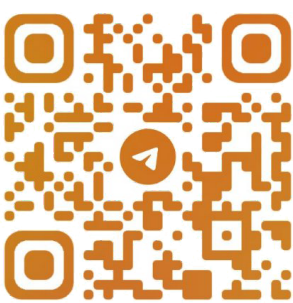
bhv®

Дункан Макгрегор,
Нэт Прайс

Java to Kotlin

A Refactoring Guidebook

Duncan McGregor and Nat Pryce



Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY

Дункан Макгрегор,
Нэт Прайс

От Java к Kotlin

Санкт-Петербург
«БХВ-Петербург»
2023

УДК 004.43
ББК 32.973.26-018.1
М15

Макгрегор, Д.

М15 От Java к Kotlin / Д. Макгрегор, Н. Прайс: Пер. с англ. — СПб.: БХВ-Петербург, 2023. — 448 с.: ил.

ISBN 978-5-9775-6841-8

Книга описывает практические приемы рефакторинга и переноса кода написанных на Java мобильных приложений для Android на язык Kotlin с сохранением совместимости. Приведено подробное сравнение этих двух языков, даны примеры перевода проектов с Java на Kotlin, добавления поддержки Kotlin в сборку Java. Показан переход от классов Java к классам Kotlin, от коллекций Java к коллекциям Kotlin, от объектов JavaBeans к значениям, от статических методов к функциям верхнего уровня. Подробно рассматривается обработка ошибок, приведены практические приемы управления проектами со смешанной кодовой базой. Даны советы по рефакторингу кода и функциональному программированию на Kotlin.

Для программистов

УДК 004.43
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Павел Шалин*
Зав. редакцией	*Людмила Гауль*
Перевод с английского	*Михаила Райтмана*
Редактор	*Григорий Добин*
Компьютерная верстка	*Ольги Сергиенко*
Оформление обложки	*Зои Канторович*

Подписано в печать 31.01 23
Формат 70×100 1/16 Печать офсетная Усл печ л 36,12
Тираж 1000 экз Заказ № 6023
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул , 20

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

ISBN 978-1-492-08227-9 (англ.)
ISBN 978-5-9775-6841-8 (рус.)

Оглавление

Предисловие

Привет, это Дункан и Нэт. Читая это предисловие, вы, наверное, пытаетесь понять, стоит ли наша книга вашего времени. Поэтому давайте сразу к делу:

эта книга не научит вас программировать компьютеры на Kotlin.

Изначально мы писали ее, имея в виду именно это, но вскоре стало ясно, что Kotlin — обширный язык, поэтому на создание такой книги должно было бы уйти больше времени, чем мы планировали. К тому же в этой области уже есть несколько прекрасных книг, а мы не любим соперничать с прекрасным.

Поэтому было принято решение упростить себе жизнь и сосредоточиться на том, чтобы научить языку Kotlin разработчиков на Java, опираясь на курс практических занятий под названием «Refactoring to Kotlin», который мы преподаем. В нем изучение языка Kotlin основано на переписывании существующего кода (согласно нашим рекламным материалам) и ориентировано на команды Java-разработчиков, которые хотят ускорить переход на Kotlin за счет уже имеющихся знаний.

Мы решили написать *такую* книгу, но быстро осознали, что Kotlin — *все-таки* весьма большой язык и что на нее *все так же* пришлось бы потратить много времени. Мы также обнаружили, что мотивированные и опытные разработчики на Java могут очень быстро усвоить большую часть Kotlin. Нам казалось, что постепенный разбор особенностей языка, которые наша целевая аудитория, скорее всего, сразу поймет и начнет применять, создал бы у нее ощущение снисходительности. Поэтому мы отказались и от этой идеи:

наша книга не научит вас языку Kotlin.

Так зачем же ее читать? Потому что это та книга, которую мы бы хотели иметь под рукой, когда только начинали внедрять Kotlin. Мы — опытные программисты с уверенным знанием языка Java и его экосистемы. Надеемся, что то же самое можно сказать и о вас. У вас, как и у нас, наверное, имеется опыт работы с целым рядом других языков. Вы познакомились с основами Kotlin и понимаете, что для извлечения из этого языка максимальной пользы вам нужно изменить свой подход к проектированию систем. Вы заметили, что некоторые моменты, вызывавшие трудности в Java, становятся более доступными в Kotlin, и что некоторых функций — таких как проверяемые исключения, в нем и вовсе нет. Вы вряд ли стремитесь к тому, чтобы писать Java-код с синтаксисом Kotlin.

Возможно, вы лично заинтересованы в этом переходе. Возможно, вы занимаете руководящую должность или успешно убедили свою команду перейти на Kotlin. Воз-

можно, вы воспользовались своим влиянием в компании, чтобы внедрить Kotlin в проект. Теперь вам нужно убедиться, что переход проходит гладко.

Возможно, вы отвечаете за проект на языке Java и хотите убедиться, что внедрение Kotlin не дестабилизирует уже существующий, критически важный для бизнеса код. Или, возможно, вы начали проект на Kotlin с нуля, но осознали, что при проектировании ваша интуиция больше ориентирована на Java и объекты, чем на Kotlin и функции.

Если вам, как и нам, это знакомо, вы пришли по адресу. Эта книга поможет вам адаптировать свой образ мышления и принципы проектирования к использованию Kotlin. Но этого мало, т. к. у вас уже есть код, который нужно поддерживать и развивать. Поэтому мы также покажем, как постепенно и безопасно перевести этот код из Java в синтаксис Kotlin и изменить при этом свой образ мышления, используя автоматические средства рефакторинга, встроенные в IntelliJ IDEA.

Как устроена эта книга?

Эта книга о том, как перейти с Java на Kotlin. Основной акцент в ней делается на код, но затрагиваются также организационные и проектные вопросы. Каждая глава посвящена какому-либо аспекту этого перехода с рассмотрением улучшений, которые по ходу дела можно вносить в типичные проекты на Java. Названия глав построены по шаблону *От подхода на Java к подходу на Kotlin*, и мы поясняем там, почему второй подход предпочтительнее. Kotlin может упростить прием, который сложно реализовать в Java, или затруднить применение распространенной в Java методики, делая архитектуру более лаконичной, устойчивой к ошибкам и совместимой с инструментами разработки.

Но мы не просто *рекомендуем* вам делать так, как принято в Kotlin, — в книге также показано, как выполнить это преобразование. И не только путем переписывания кода на Java, но и за счет его постепенного рефакторинга в Kotlin безопасным путем, который позволит вам поддерживать кодовую базу на двух языках.

Как были выбраны темы?

Мы анализировали, как разработчики используют Java и Kotlin, проводили опросы, чтобы понять, чем эти языки различаются и какие их аспекты вызывают недопонимание. В дополнение к этому мы применили методы машинного обучения для анализа 33 459 открытых проектов на Java и Kotlin. Так был составлен список возможностей, которые мы пометили в формате «один подход вместо другого подхода» и ранжировали по частоте и «коэффициенту проблематичности», чтобы определить, какие из них стоит включить в книгу. Напоследок мы упорядочили полученные темы по...

...хорошо, не будем преувеличивать — это сложно назвать порядком...

По правде говоря, мы в первую очередь выбирали темы, о которых нам хотелось написать и которые, по нашему мнению, были интересными и информативными.

Типичными примерами этого подхода стали *глава 15 «От инкапсулированных коллекций к псевдонимам типов»*, *глава 9 «От многострочных функций к однострочным»* и *глава 20 «От выполнения ввода/вывода к передаче данных»*. Мы также искали аспекты, в которых Kotlin и Java существенно разнились по своим концепциям, поскольку именно о них мы больше всего узнали, расспрашивая о различиях этих языков. В результате появились *глава 4 «От необязательных типов к обнуляемым*, *глава 6 «От коллекций Java к коллекциям Kotlin»* и *глава 8 «От статических методов к функциям верхнего уровня»*.

В процессе подготовки этих глав мы обращали внимание на другие темы и добавляли их в список. В частности, во время работы над этапом рефакторинга в той или иной главе мы нередко замечали, что изменения, вносимые нами в код, по ощущениям заслуживали отдельной главы. Примерами такого подхода могут служить *глава 13 «От потоков к итерируемым объектам и последовательностям»*, *глава 10 «От функций к функциям-расширениям»* и *глава 11 «От методов к свойствам»*.

Результат этого процесса ни в коей мере не является исчерпывающим. Бегло просмотрев оглавление или алфавитный указатель, вы можете заметить, что некоторые важные темы остались без внимания. Возьмем, к примеру, *корутины* — этот абзац является единственным упоминанием в книге такой столь обширной темы, поскольку, как обнаружилось, они не повлияли на то, как мы пишем серверный код. В связи с этим мы не стали их рассматривать. Есть также темы, которые не получилось охватить из-за недостатка времени и места, включая конструкторы, предметно-ориентированные языки, рефлексию, средства внедрения зависимостей, транзакции, — этот список можно продолжать и продолжать!

Надеемся, вам будет интересно читать то, что мы *сумели* написать. Эта книга во многом ориентирована на тактику, а не на стратегию. В ней описываются небольшие сражения, в которых можно победить, отсиживаясь в траншеях, а не масштабные битвы, требующие переброса целых дивизий. Хотя, когда начнут вырисовываться более общие темы, мы попытаемся в заключительной *главе 23 «Продолжение путешествия»* связать их друг с другом, чтобы подытожить все, что рассмотрели, — в ней речь пойдет о том, чему мы научились в процессе подготовки этой книги.

Уровень сложности

Как следует оценивать качество кода нашего программного обеспечения? Предположим, у нас есть две потенциальные реализации, каждая из которых удовлетворяет потребностям наших клиентов, — как их сравнить и как определить, является вносимое изменение положительным или отрицательным? В качестве ответа мы выбрали уровень сложности. При прочих равных условиях мы отдаем предпочтение более простым решениям, которые ведут себя предсказуемо.

Конечно, простота и сложность в какой-то мере являются субъективными понятиями. Наши личные предпочтения немного различаются, поэтому у нас иногда возникают разногласия относительно того, какая из реализаций лучше. В таких случаях в соответствующей главе могут быть представлены альтернативы. Тем не менее мы

оба верим в то, что функциональное программирование способствует упрощению наших систем, особенно в сочетании с объектно-ориентированным (ОО) обменом сообщениями.

В последние годы Java двигается в этом направлении. Язык Scala сделал резкий скачок в сторону функционального программирования, но отдалился от ООП. По нашему мнению, сущность Kotlin позволяет сочетать функциональное и объектно-ориентированное программирование таким образом, что уровень сложности понижается и пробуждает у простых смертных разработчиков их лучшие качества.

Идеальный код

Если уж речь зашла о простых смертных, следует затронуть тему качества кода. Стремление к написанию совершенного кода для примеров в книге выглядит чрезвычайно заманчиво. Знаем, вы будете судить нас по коду, который здесь представлен. К тому же, как это часто бывает с разработчиками, наша самооценка во многом зависит от качества результатов нашей работы.

С другой стороны, мы инженеры, а не художники. Наша работа заключается в соблюдении баланса между объемом проекта, графиком его реализации и его стоимостью для наших клиентов. Никого, кроме нас, не волнует качество кода, за исключением тех случаев, когда оно влияет на одну из этих трех первостепенных характеристик.

В связи с этим мы попытались показать в наших примерах правдоподобный код, используемый в реальных условиях. Не все примеры, которые служат отправными точками, получились такими, как нам бы хотелось, но в конечном итоге мы стараемся показать, как их улучшить. Зачастую, прежде чем дать положительный результат, рефакторинг делает все еще хуже, поэтому нас точно не стоит судить по коду, размещенному в середине глав. Мы стремимся к тому, чтобы к завершению главы наш код стал достаточно хорошим, но не настолько, чтобы нас могли обвинить в пустой трате денег наших клиентов.

Тем не менее даже после рассмотрения темы, которую хотелось проиллюстрировать, мы обычно вносим изменения, которые делают код чище и не требуют больших расходов. Мы неоднократно придумывали тему и писали для нее главу, только чтобы привести код к состоянию, которое бы нас удовлетворяло. В конечном счете мы не только инженеры, но и художники.

Форматирование кода

Мы старались по возможности приводить наши примеры в соответствие с рекомендациями по написанию кода, принятыми в Java и Kotlin (точнее, с нашей интерпретацией этих рекомендаций).

На практике строки примеров кода в печатных изданиях получаются намного короче 120 символов, которые обычно составляют длину строки в современных IDE, и чтобы подогнать код к ширине страницы, нам приходилось использовать перевод строки чаще, чем обычно. В реальных условиях можно уместить в одной строке

четыре-пять параметров или аргументов, а в этой книге зачастую умещается всего один. В процессе форматирования примеров для печатных страниц нам больше пришелся по душе вертикальный стиль. Нам кажется, что Kotlin естественным образом занимает больше вертикального пространства, чем Java, но даже в Java удобочитаемость кода, похоже, улучшилась за счет более коротких строк, большего количества переносов и визуального выравнивания. В IDE горизонтальная прокрутка определенно является почти такой же неудобной, как и в книге, и наши парные сессии удалось оптимизировать благодаря сокращению длины строк и более активному размещению окон в ряд. Кроме того, выделение отдельной строки для каждого параметра существенно улучшает представление о различиях между версиями кода. Надеемся, такой подход как минимум не затруднит чтение кода. Если это действительно так, можете взять его на вооружение.

Иногда мы будем скрывать код, не относящийся к сути дела. Если строка начинается с троеточия, это говорит о том, что часть кода опустили для ясности или краткости. Например:

```
fun Money(amount: String, currency: Currency) =
    Money(BigDecimal(amount), currency)

... и другие удобные перегрузки
```

Условные обозначения

В этой книге используются следующие условные обозначения:

- *Курсив*

  Обозначает новые термины.

- **Полужирный шрифт**

  Обозначает интернет-адреса (URL) и адреса электронной почты.

- Шрифт Arial

  Обозначает имена и расширения файлов.

- `Моноширинный шрифт`

  Используется в листингах кода, а также в тексте, обозначая такие программные элементы, как имена переменных и функций, базы данных, типы данных, переменные окружения, утверждения и ключевые слова.

Этот значок обозначает совет или предложение.

Этот значок обозначает примечание общего характера.

Этот значок обозначает предупреждение или предостережение.

Использование примеров кода

Большинство примеров кода в этой книге (те, что находятся в разделах о рефакторинге) доступны онлайн на GitHub. Ссылка дается над кодом, как показано здесь (пример 0.1).

Пример 0.1
[table-reader.1:src/test/java/travelator/tablereader/TableReaderAcceptanceTests.kt]

```
class TableReaderAcceptanceTests {
@Test
    fun test() {
    }
}
```

Если вы читаете электронную версию английского издания книги, на этой ссылке можно щелкнуть, чтобы перейти к файлу на GitHub. В бумажной версии щелкай не щелкай — ничего не произойдет, извините. Но если взять номер примера (в данном случае 0.1) и ввести его в форму на веб-сайте книги (**https://java-to-kotlin.dev/code.html**), можно получить ссылки, ведущие в то же место.

Примечание от русской редакции

Чтобы облегчить этот процесс читателям бумажной версии книги, мы снабдили все такие ссылки соответствующим QR-кодом. Смартфон сейчас есть у каждого жителя планеты, установить на него считыватель QR-кодов не составляет проблемы, а перегнать, если потребуется, полученную с его помощью ссылку на интернет-ресурс в свой рабочий компьютер тоже не бином Ньютона.

В Git работа над разными примерами кода (которые могут охватывать сразу несколько глав) ведется в отдельных ветвях. Этапы помечены тегами (в данном случае: **table-reader.1**). Ссылка на GitHub ведет к коду с тегом, что позволяет просматривать не только представленный файл (в данном случае: **src/test/java/travelator/tablereader/TableReaderAcceptanceTests.kt**), но и прочие файлы в той же версии примера. Вы также можете выбирать другие теги для просмотра разных версий и другие ветви для просмотра разных примеров. Для более быстрого перемещения по коду вы можете клонировать репозиторий, открыть его в IntelliJ и использовать окно Git для переключения между ветвями и версиями.

Наши примеры кода ненастоящие! Проекты собираются и проходят тесты, но они вымышленные. Некоторые примеры плохо сочетаются друг с другом, а в некоторых, если заглянуть «за занавес», можно увидеть, как мы дергаем за ниточки. Мы стараемся быть честными, но предпочитаем наглядность!

Если у вас возникнут технические вопросы или проблемы при использовании примеров кода, посетите веб-сайт книги (**https://java-to-kotlin.dev**) или напишите нам по адресу: **bookquestions@oreilly.com**.

Предназначение этой книги — помочь вам в решении ваших задач. Вы можете использовать любой пример кода, содержащийся в этой книге, в своих программах и

документации. И да — можете не связываться с нами, если вы не воспроизводите его существенную часть. Это, например, касается ситуаций, когда вы включаете в свою программу несколько фрагментов приведенного здесь кода. Однако продажа или распространение примеров из книг издательства O'Reilly требует отдельного разрешения. Вы можете свободно цитировать эту книгу, включая примеры, при ответе на вопрос, но, если хотите включить существенную часть приведенного здесь кода в документацию своего продукта, вам следует связаться с нами.

Отсылка на оригинал приветствуется, но не является обязательной. Она обычно состоит из названия, имен авторов, издательства и ISBN. Например; «Из Java в Kotlin, Дункан Макгрегор и Нэт Прайс (O'Reilly). Copyright 2021 Duncan McGregor and Nat Pryce, 978-1-492-08227-9».

Если вам кажется, что то, как вы обращаетесь с примерами кода, выходит за рамки добросовестного использования или условий, приведенных ранее, можете обратиться к нам по адресу: **permissions@oreilly.com**.

Платформа онлайн-обучения O'Reilly

O'REILLY На протяжении почти 40 лет издательство O'ReillyMedia предоставляет технические и бизнес-тренинги, знания и опыт, чтобы помочь компаниям достичь успеха.

Его уникальное сообщество экспертов и новаторов делится знаниями и опытом с помощью книг, статей, конференций и нашей платформы онлайн-обучения. Эта платформа предоставляет доступ к учебным курсам, проводимым в прямом эфире, углубленные учебные программы, интерактивные среды для написания кода и обширную коллекцию текстов и видео от O'Reilly и более чем 200 других издательств. Больше информации можно найти на странице **http://oreilly.com**.

Как с нами связаться?

Пожалуйста, адресуйте все комментарии и вопросы относительно этой книги издателю:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в Соединенных Штатах и Канаде)
707-829-0515 (международные или местные звонки)
707-829-0104 (факс)

У этой книги есть веб-страница, где мы приводим ошибки, примеры и любую дополнительную информацию. Она находится по адресу: **https://oreil.ly/java-to-kotlin**.

Для комментариев и технических вопросов об этой книге используйте электронную почту: **bookquestions@oreilly.com**.

Новости и информацию о наших книгах можно найти сайте **http://www.oreilly.com**.

Следите за нами в Twitter: **https://twitter.com/oreillymedia**.

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

Благодарности

Хади Харири поделился с издательством O'Reilly идеей о необходимости написать эту книгу, а Зан Маккуэйд ему поверил, за что мы им благодарны. Спасибо нашему редактору, Саре Грей, которой пришлось иметь дело с последствиями этого решения, а также Керину Форсайту и Кейт Гэллоуэй за подготовку текста и его последующую публикацию.

Многие наши друзья, коллеги и любезные незнакомцы проверяли этот текст, начиная с ранних, несогласованных черновиков и заканчивая почти готовой книгой. Спасибо Яне Афанасьевой, Джеку Боллсу, Дэвиду Дентону, Брюсу Эккелю, Дмитрию Кандалову, Девину Пилу, Джеймсу Ричардсону, Ивану Санчезу, Джордану Стюарту, Роберту Столлу, Кристофу Штурму, Лукашу Выциску, Даниелю Запполду и нашим техническим редакторам: Уберто Барбини, Джеймсу Хармону, Марку Мейнарду и Аугусто Родригезу. Мы крайне признательны за все ваши рекомендации, поощрения и откровенность.

Концепция *экстремального программирования* радикально повлияла на то, как мы пишем программное обеспечение, — мы все в долгу перед Уордом Каннингемом и Кентом Беком. Также спасибо Мартину Фаулеру, без которого эта книга не была бы написана. Британское сообщество eXtreme Tuesday Club развивает эти идеи с 1999 года, объединив вокруг себя ряд разработчиков-единомышленников. Нам посчастливилось поработать и поучиться у многих талантливых участников этой группы. Если у вас есть проблема, с которой никто другой не в состоянии помочь, попробуйте обратиться к ним, если вам удастся их отыскать.

Благодарности Дункана

Думаю, моя жена никогда не поймет, чем я зарабатываю на жизнь, а уж шансов на то, что она дочитает эту книгу до конца, и вовсе нет. Но досюда она, наверное, доберется. Спасибо тебе, Джо Макгрегор, за твое терпение, пока я занимался писательством, вместо того чтобы проводить время с тобой, и за наши обсуждения этой книги в те моменты, когда нам удавалось побыть вместе. Я бы не смог это сделать без твоей поддержки и поощрения. Также спасибо нашим двум сыновьям, Каллуму и Алистеру, которыми мы так гордимся.

Спасибо Викки Кенниш за ее живой интерес к роли мамы писателя и за вопросы о продвижении работы во время наших карантинных прогулок в период COVID-19. Уверен, что мой покойный отец, Джон, вел бы себя более сдержанно, но тем не менее не забывал бы похвастаться моей книгой своим друзьям. Также умер, но не забыт, наш красавец-кот Свитпи — он составлял мне компанию большую часть времени, пока я писал эту книгу, но не дожил до ее окончания.

Робин Холливел — человек, чья дружба и поддержка ощущается мной на протяжении всей моей взрослой жизни. То же самое можно сказать о моей сестре, Люси Сил, и многих других членах семьи, слишком многочисленных для того, чтобы перечислять их поименно. Если говорить о моей профессиональной жизни, то помимо тех, кто оставил свои отзывы, хочу поблагодарить Алана Дайка, Ричарда Кера и Гарета Сильвестра-Брэдли — их влияние и поддержка выходят за рамки служебных обязанностей.

Благодарности Нэта

Когда я сообщил своей жене Ляман о планах написать еще одну книгу, чувство ужаса не было ее первой реакцией. За это и за ее постоянную поддержку я должен сказать ей большое спасибо.

Снимаю шляпу перед своей сестрой, Лоис Прайс, и шурином Остином Винсом, чьи мотоциклетные поездки, книги и фильмы легли в основу приложения для планирования свободных путешествий, которое используется в примерах кода.

Также благодарю Оливера и Алекс. Теперь, когда книга готова, я снова свободен для консультаций в сфере музыки и программирования игр.

ГЛАВА 1

Введение

Сущность языка программирования

В сущность каждого языка программирования заложено свое «направление течения». Будь вы моряк или программист, движение по направлению течения идет гладко. Если же двигаться против течения, все усложняется. Когда разработчик идет против устоев языка программирования, ему приходится писать лишний код. При этом страдает производительность, повышается вероятность допущения ошибок, возникает необходимость переопределения значений, предоставляемых по умолчанию для удобства, а инструментарий доставляет проблемы на каждом шагу.

Движение против течения требует постоянных усилий и приносит сомнительную отдачу.

Например, код на языке Java всегда можно было писать в функциональном стиле, но до выхода Java 8 мало кто так делал, и тому были веские причины.

Перед вами код Kotlin, вычисляющий сумму чисел в списке путем свертки структуры данных по сумме:

```
val sum = numbers.fold(0, Int::plus)
```

Давайте посмотрим, что нужно было сделать в Java 1.0, чтобы добиться того же результата, — не заблудитесь в глубине веков, переносясь в далекий 1995-й...

В Java 1.0 не было функций первого класса, поэтому их приходилось реализовать в виде объектов и определять собственные интерфейсы для разных их типов. Например, функция сложения принимает два аргумента, поэтому мы должны определить тип двухаргументных функций:

```
public interface Function2 {
    Object apply(Object arg1, Object arg2);
}
```

Затем нам надо написать функцию высшего порядка `fold`, скрывающую операции сложения и изменения значения, которые требует класс `Vector` (стандартная библиотека Java образца 1995 года не включала в себя Collections Framework):

```
public class Vectors {
    public static Object fold(Vector l, Object initial, Function2 f) {
        Object result = initial;
        for (int i = 0; i < l.size(); i++) {
            result = f.apply(result, l.get(i));
        }
```

```
        return result;
    }

... и другие операции с векторами
}
```

Затем определить отдельный класс для каждой функции, которую мы хотим передавать нашей функции `fold`. Оператор сложения нельзя передать по значению, и в языке все еще не предусмотрено ссылок на методы лямбд или замыканий. Нет даже внутренних классов. Также в Java 1.0 отсутствуют обобщенные типы и автоупаковка — нам придется самим привести аргументы к нужному типу и написать процедуру упаковки для типов и примитивов:

```
public class AddIntegers implements Function2 {
    public Object apply(Object arg1, Object arg2) {
        int i1 = ((Integer) arg1).intValue();
        int i2 = ((Integer) arg2).intValue();
        return new Integer(i1 + i2);
    }
}
```

И наконец, мы можем применить все это для вычисления суммы:

```
int sum = ((Integer) Vectors.fold(counts, new Integer(0), new AddIntegers()))
    .intValue();
```

То, что отняло у нас столько усилий, в 2020 году можно было записать одним выражением в любом популярном языке.

Но это еще не все. Поскольку в Java отсутствуют стандартные типы функций, мы не можем с легкостью сочетать между собой разные библиотеки, написанные в функциональном стиле. Нам нужно писать классы-адаптеры, чтобы привязывать друг к другу типы функций, определенные в разных библиотеках. И, поскольку у виртуальной машины нет JIT и сборщика мусора, производительность нашего функционального кода хуже, чем у альтернативы в императивной парадигме:

```
int sum = 0;
for (int i = 0; i < counts.size(); i++) {
    sum += ((Integer)counts.get(i)).intValue();
}
```

В 1995 году попросту не было задач, которые бы оправдывали написание кода на Java в функциональном стиле. Программистам было легче писать императивный код, который перебирал коллекции и изменял состояние.

Написание функционального кода *противоречит сущности* Java 1.0.

Сущность языка формируется по мере того, как его создатели и пользователи достигают взаимопонимания о том, каким образом должны вести себя его возможности, и выражают свое видение и предпочтения в библиотеках, которые берут за основу другие программисты. Сущность влияет на то, как программисты пишут код на этом языке, что, в свою очередь, сказывается на процессе его развития,

а также на библиотеках и средствах разработки, а это изменяет сущность языка и то, как программисты пишут на нем свой код. Это постоянный эволюционный цикл взаимной обратной связи.

Например, с выходом версии 1.1 в Java появились анонимные внутренние классы, а в стандартную библиотеку Java 2 был добавлен пакет Collections Framework. Появление анонимных внутренних классов избавило нас от необходимости писать класс с именем для каждой функции, которую нужно передать в `fold`, но это, возможно, затрудняет чтение итогового кода:

```
int sum = ((Integer) Lists.fold(counts, new Integer(0),
    new Function2() {
        public Object apply(Object arg1, Object arg2) {
            int i1 = ((Integer) arg1).intValue();
            int i2 = ((Integer) arg2).intValue();
            return new Integer(i1 + i2);
        }
    })).intValue();
```

Функциональные идиомы по-прежнему противоречат сущности Java 2.

Перенесемся в 2004 год. Следующим релизом, который существенно преобразил язык, стал Java 5. В нем появились обобщения и автоупаковка, которые положительно сказались на безопасности типов и сократили количество шаблонного кода:

```
public interface Function2<A, B, R> {
    R apply(A arg1, B arg2);
}

int sum = Lists.fold(counts, 0,
    new Function2<Integer, Integer, Integer>() {
        @Override
        public Integer apply(Integer arg1, Integer arg2) {
            return arg1 + arg2;
        }
    });
```

Разработчики на Java нередко используют библиотеку Guava от Google (**https://oreil.ly/dMX73**), чтобы добавить распространенные функции высшего порядка для работы с коллекциями (`fold` не входит в их число), однако сами авторы Guava рекомендуют по умолчанию писать императивный код, т. к. он имеет лучшую производительность и его обычно легче читать.

В Java 5 функциональное программирование тоже во многом противоречит сущности языка, но уже начинает вырисовываться тенденция.

В Java 8 были добавлены анонимные функции (они же лямбда-выражения) и ссылки на методы, а стандартная библиотека обзавелась API-интерфейсом Streams. Компилятор и виртуальная машина оптимизируют лямбды, чтобы избежать ухудшения производительности, свойственного анонимным внутренним классам.

Streams API полностью поддерживает функциональные идиомы, что наконец-то позволяет нам написать следующее:

```
int sum = counts.stream().reduce(0, Integer::sum);
```

Но не все так гладко. Эта версия языка все еще не позволяет передать оператор сложения в качестве параметра для функции `reduce` из Streams, хотя в стандартной библиотеке есть функция `Integer::sum`, которая делает то же самое. Система типов Java по-прежнему порождает неловкие ситуации из-за различий между ссылочными и примитивными типами. В Streams API не хватает некоторых востребованных функций высшего порядка, наличие которых можно было бы ожидать в функциональном языке (или даже в Ruby). Проверяемые исключения плохо сочетаются со Streams API и функциональным программированием в целом. А создание неизменяемых классов с семантикой значений по-прежнему требует много шаблонного кода. Тем не менее в восьмой версии Java произошли фундаментальные изменения, которые сделали функциональный стиль пригодным к использованию — он, возможно, не полностью соответствовал ориентации языка, но по крайней мере не противоречил ей.

Версии, последовавшие за Java 8, привнесли в язык и библиотеку ряд менее значимых возможностей с лучшей поддержкой идиом функционального программирования, но на наше вычисление суммы это никак не повлияло. И это возвращает нас обратно в настоящее.

В случае с Java изменение сущности языка (и того, как программисты к ней приспосабливались) характеризовалось несколькими отдельными стилями программирования.

История стилей программирования на Java (на наш взгляд)

Подобно древней поэзии, развитие стиля программирования на Java делится на четыре отдельных периода: первобытный, JavaBeans, корпоративный и современный.

Первобытный стиль

Изначально предназначенный для использования в бытовой технике и интерактивном телевидении, язык Java начал набирать обороты, только когда в крайне популярном на тот момент браузере Netscape Navigator появилась поддержка Java-апплетов. Компания Sun выпустила пакет разработки для Java версии 1.0, компания Microsoft включила Java в Internet Explorer, и внезапно все, у кого был веб-браузер, получили среду выполнения этого языка программирования, что привлекло к нему бурный интерес.

К этому моменту уже сформировались основные элементы Java:

- виртуальная машина, ее байт-код и формат class-файлов;
- примитивные и ссылочные типы, null-ссылки, сборка мусора;

- классы и интерфейсы, методы и инструкции управления логикой;
- проверяемые исключения для обработки ошибок — Abstract Window Toolkit (AWT);
- классы для работы с интернет- и веб-протоколами;
- загрузка и компоновка кода во время выполнения, изолированная диспетчером безопасности.

Тем не менее Java еще не стал полноценным языком программирования общего назначения: JVM была медленной, а стандартная библиотека — бедной. Он был чем-то средним между C++ и Smalltalk, и эти два языка повлияли на стиль программирования на Java того времени. Такие вещи как «getFoo/setFoo» и «AbstractSingletonProxyFactoryBean», над которыми потешались программисты на других языках, еще не были широко распространены.

Одной из незамеченных инноваций Java оказались официальные рекомендации по написанию кода, в которых описывалось, как программисты должны называть свои пакеты, классы, методы и переменные. Программисты на C и C++ руководствовались бесконечным множеством соглашений, в результате чего код, в котором сочеталось несколько библиотек, выглядел ~~слишком вычурно~~ несколько пестро. Общепринятые рекомендации по написанию кода позволили программистам на Java с легкостью интегрировать чужие библиотеки в свои программы и поспособствовали формированию энергичного сообщества разработчиков открытого ПО, которое существует по сей день.

Стиль JavaBeans

После первоначального успеха Java компания Sun задалась целью сделать этот язык практичным инструментом для создания приложений. В Java 1.1 (1996 год) дополнился синтаксис (прежде всего за счет внутренних классов), улучшилась среда выполнения (в основном благодаря JIT-компиляции и рефлексии) и расширилась стандартная библиотека. В Java 1.2 (1998 год) был добавлен стандартный API-интерфейс коллекций и фреймворк Swing для создания кросс-платформенных графических интерфейсов. В результате Java-приложения начали выглядеть и вести себя одинаково неуклюже на всех настольных операционных системах.

В то время компании Sun не давало покоя доминирование Microsoft и Borland в сфере разработки корпоративного программного обеспечения. У Java был потенциал стать серьезным конкурентом Visual Basic и Delphi. В связи с этим программисты Sun представили целый ряд API-интерфейсов, в значительной степени вдохновленных наработками Microsoft: JDBC — для доступа к базам данных (аналог ODBC от Microsoft), Swing — для программирования настольных графических интерфейсов (аналог MFC от Microsoft) и фреймворк JavaBeans, оказавший наибольшее влияние на стиль программирования на Java.

JavaBeans API стал ответом на компонентную модель Microsoft ActiveX, ориентированную на графическое программирование с написанием минимального количества кода и заключавшуюся в перетаскивании компонентов на форму. В Windows

программисты могли использовать компоненты ActiveX в своих программах на Visual Basic, а также встраивать их в офисные документы или веб-страницы в своей внутренней корпоративной сети. Несмотря на простоту в использовании, компоненты ActiveX было на редкость сложно писать. С JavaBeans все стало иначе. Вам просто нужно было соблюдать некоторые дополнительные правила, чтобы создать класс «Bean», экземпляры которого мог создавать и настраивать графический дизайнер. Девиз «напиши один раз — запускай где угодно» также означал, что компоненты JavaBeans можно было использовать (или продавать) в любой операционной системе, а не только в Windows.

Чтобы класс можно было считать JavaBeans, ему следовало иметь конструктор, не принимающий ни одного аргумента. Класс должен был быть сериализуемым и объявлять API-интерфейс, состоящий из публичных свойств, доступных для чтения и, при желании, для записи, методов, которые можно вызывать, и событий, которые генерировались объектами класса. Идея состояла в том, что программисты создавали экземпляры Bean в визуальном редакторе приложения, настраивали их путем задания свойств и соединяли события, генерируемые объектами Bean, с методами других объектов Bean. Каждое свойство в BeansAPI по умолчанию определялось парой методов, имена которых начинались с `get` и `set`. Это можно было переопределить, но в таком случае программисту приходилось писать больше классов с шаблонным кодом. Обычно этим занимались только при модификации существующих классов, чтобы они вели себя как JavaBeans. В новом коде было намного проще плыть по течению.

Недостаток стиля JavaBeans заключался в том, что он во многом опирался на изменяемое состояние, которое должно было быть более публичным, чем в старых добрых объектах Java, т. к. средства визуального редактирования должны были устанавливать свойства объекта вместо того, чтобы передавать параметры его конструктору. Этот подход хорошо работает для компонентов пользовательского интерфейса, т. к. их можно безопасно инициализировать с параметрами по умолчанию, корректируя их внешний вид и поведение уже после создания. Если же у класса нет разумных параметров по умолчанию, обращение с ним в том же ключе чревато ошибками, поскольку механизм проверки типов не знает, предоставили ли мы все необходимые значения. Соглашения по оформлению JavaBeans усложняют написание кода, а изменение зависимостей может незаметно нарушить работу клиентского приложения.

Графический метод разработки JavaBeans так и не получил широкого распространения, но соглашения о написании кода прижились. Программисты на Java соблюдали их, даже когда их классы не предназначались для использования в качестве JavaBeans. Эта технология имела огромное, продолжительное и не вполне позитивное влияние на стиль программирования Java.

Корпоративный стиль

В конце концов популярность к Java пришла из корпоративного мира. Вопреки ожиданиям, этот язык не стал заменой Visual Basic в корпоративных настольных системах, но начал применяться вместо C++ для написания серверных приложений.

В 1998 году компания Sun выпустила версию Java 2 Enterprise Edition (ныне известную как JakartaEE, а в то время носившую название J2EE) — набор стандартных API-интерфейсов для программирования серверных систем обработки транзакций.

Проблему для API-интерфейсов J2EE представляет *инверсия абстракции* (см. далее). Эта проблема также свойственна API-интерфейсам JavaBeans и апплетов (и те и другие, к примеру, не позволяют передавать параметры конструкторам), но в J2EE она носит куда более острый характер. У J2EE-приложений нет единой точки входа. Они состоят из множества мелких компонентов, жизненным циклом которых управляет контейнер приложения, а доступ к ним осуществляется через сервис имен JNDI. Приложениям требуется много шаблонного кода и изменяемое состояние для поиска ресурсов, от которых они зависят. В ответ на эти недостатки программисты изобрели фреймворки для *внедрения зависимостей* (*англ.* dependency injection, DI), которые занимались поиском и привязкой ресурсов, а также управлением жизненным циклом. Самым успешным из них является Spring. Он основан на соглашениях о написании кода, предназначенных для JavaBeans, и использует механизм рефлексии для составления приложений из объектов, подобных компонентам JavaBeans.

Инверсия абстракции

Инверсия абстракции — это архитектурный дефект, из-за которого программная платформа не дает клиентскому коду использовать нужные ему низкоуровневые механизмы. В результате программисты вынуждены заново реализовывать эти низкоуровневые механизмы, используя средства более высокого уровня, предоставляемые API-интерфейсом платформы, а тот в свою очередь опирается на те самые возможности, которые реализуются заново. Это приводит к написанию ненужного кода, ухудшению производительности и увеличению расходов на обслуживание и тестирование.

Возьмем, к примеру, сервлеты J2EE. В Servlet API за создание объектов-сервлетов отвечали не их собственные контейнеры, а контейнер веб-приложения. На самом деле веб-приложение даже не было написано на Java — оно представляло собой XML-файл со списком классов, экземпляры которых должен был создать контейнер сервлета. Поэтому каждому сервлету следовало иметь конструктор, не принимающий аргументов, а приложение не могло инициализировать свои сервлеты, передавая объекты их конструкторам.

Вместо этого приходилось писать класс `ServletContextListener`, создававший объекты, необходимые сервлетам приложения, и хранивший их в виде именованных, нетипизированных атрибутов для `ServletContext`:

```
public class ExampleServletContextListener
    implements ServletContextListener {

    public void contextInitialized(ServletContextEvent contextEvent) {
        ServletContext context = contextEvent.getServletContext();
        context.setAttribute("example.httpClient", createHttpClient());
        context.setAttribute("example.template", new URITemplate(
            context.getInitParameter("example.template")));
        ...
    }

    ...
    }
```

Чтобы себя инициализировать, сервлеты искали в контексте нужные им объекты и приводили их к ожидаемому типу:

```
public class ExampleServlet extends HttpServlet {
    private HttpClient httpClient;
    private URITemplate routeServiceTemplate;
    ...

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        ServletContext context = config.getServletContext();
        this.httpClient =
            (HttpClient) context.getAttribute("example.httpClient");
        this.routeServiceTemplate =
            (URITemplate) context.getAttribute("example.template");
        ...
    }

    ...
}
```

Если бы API-интерфейс сервлетов не запрещал вызывать конструкторы с аргументами, это не потребовало бы столько усилий. К тому же при вызове конструктора происходила бы проверка типов. Неудивительно, что фреймворки для внедрения зависимостей казались Java-программистам шагом вперед!

Лишь в версии 3.0, спустя 20 лет после первого выпуска Servlet API, веб-приложения получили возможность создавать экземпляры сервлетов и передавать зависимости их конструкторам.

Если говорить о стиле программирования, то DI-фреймворки поощряют программистов избегать прямого использования ключевого слова `new`, перекладывая на них ответственность за создание объектов. API-интерфейсам Android тоже свойственна инверсия абстракции, и программы на этой платформе также используют DI-фреймворки для упрощения взаимодействия с этими интерфейсами. Поскольку DI-фреймворки в первую очередь ориентированы на механизмы, а не на моделирование предметной области, имена классов начали приобретать «корпоративный» окрас, как в случае с печально известным классом `AbstractSingletonProxyFactoryBean` в Spring.

Однако в корпоративную эпоху происходили и положительные изменения. Например, выпуск Java 5 привнес в этот язык обобщения и автоупаковку — две самые важные возможности на тот момент. Тогда в сообществе Java наблюдался стремительный рост количества библиотек с открытым исходным кодом, основанных на соглашениях о создании пакетов и центральном репозитории Maven. Это породило благотворный цикл: наличие высококачественных открытых библиотек способствовало применению Java в разработке критически важных приложений и стимулировало создание еще большего количества открытых библиотек. За этим последовало появление первоклассных средств разработки, включая среду IntelliJ IDE, которая используется в этой книге.

Современный стиль

Выход Java 8 ознаменовал следующее крупное нововведение — лямбды. Для совместимости с ними была существенно расширена стандартная библиотека. API-интерфейс Streams поощрял функциональный стиль программирования, в рамках которого обработка данных происходила путем преобразования потоков неизменяемых значений, а не за счет изменения состояния объектов. В новом API-интерфейсе игнорировались соглашения о написании кода, касавшиеся методов доступа к свойствам, — вместо этого использовался стиль, характерный для современной эпохи.

Благодаря развитию облачных платформ программистам больше не нужно было развертывать свои серверы в контейнерах JavaEE-приложений. При использовании легковесных веб-фреймворков для создания приложения достаточно написать функцию `main`. Многие разработчики серверного ПО отказались от использования DI-фреймворков, т. к. процесс композиции функций и объектов был достаточно удобен и без них. В ответ, чтобы не утратить актуальность, создатели DI-фреймворков существенно упростили свои API-интерфейсы. С отсутствием изменяемого состояния и средств внедрения зависимостей необходимость в соглашениях о написании кода, принятых в JavaBeans, становится менее насущной. В пределах одного проекта вполне нормально предоставлять доступ к неизменяемым полям, т. к. IDE может мгновенно инкапсулировать поля, скрытые за методами доступа, если они потребуются.

В Java 9 появились модули, но они пока что не получили широкого распространения за пределами самого пакета JDK. Самыми захватывающими изменениями в последних выпусках Java стали разделение JDK на модули и оформление редко используемых компонентов, таких как CORBA, в виде дополнительных расширений, не входящих в состав JDK.

Будущее

В будущем в Java должны появиться новые возможности, которые упростят применение современного стиля: записи, сопоставление с образцом, пользовательские типы значений и в конечном счете объединение примитивных и ссылочных типов в единую систему.

Но это непростая задача, на выполнение которой уйдут годы. В Java изначально имелись глубоко укоренившиеся противоречия и граничные случаи, которые сложно объединить в элегантные абстракции без потери обратной совместимости. Создатели Kotlin имели возможность оглянуться на предыдущие 25 лет и начать всё с чистого листа.

Сущность Kotlin

Kotlin — молодой язык, но по своей направленности он явно отличается от Java.

Когда мы писали эту книгу, в разделе **Why Kotlin** (Почему стоит выбрать Kotlin) домашней страницы этого языка (**https://oreil.ly/pqZbu**) были приведены четыре

принципа, легших в его основу: лаконичность, безопасность, функциональная совместимость и наличие удобного инструментария. Создатели языка и его стандартной библиотеки также описали приоритеты, способствующие реализации этих принципов, включая следующие:

- *Вместо изменения состояния в Kotlin рекомендуется преобразовывать неизменяемые данные.*

  Классы данных позволяют легко определять новые типы с семантикой значений. Стандартная библиотека облегчает и сокращает процесс преобразования коллекций с неизменяемыми данными по сравнению с перебором и изменением имеющихся значений.

- *В Kotlin отдается предпочтение явно выраженному поведению.*

  Например, в этом языке нет автоматического приведения типов, даже если тип с меньшим диапазоном приводится к типу с большим. Java автоматически преобразует значения `int` в `long`, т. к. это не вызывает потери точности. В Kotlin для этого нужно вручную вызвать `Int.toLong()`. Выбор в пользу ясности особенно ярко проявляется в управлении потоком выполнения. У вас есть возможность перегружать арифметические операции и операторы сравнения для своих собственных типов, однако перегрузка сокращенных логических операторов (`&&` и `||`) невозможна, т. к. это позволило бы вам переопределить логику выполнения.

- *Kotlin отдает предпочтение статическому связыванию перед динамическим.*

  Kotlin поощряет типобезопасный, композиционный стиль написания кода. Функции-расширения привязываются статически. Классы по умолчанию не являются расширяемыми, а методы — не полиморфные. Вы должны явно указать, что вам нужны полиморфизм и наследование. Если вы хотите использовать рефлексию, можете добавить в качестве зависимости библиотеку, предназначенную для определенной платформы. Kotlin изначально разработан для использования с языковой средой IDE, которая статически анализирует код, чтобы направлять программиста, автоматизировать навигацию и преобразование программы.

- *Kotlin не любит особые случаи.*

  Kotlin в сравнении с Java имеет меньше особых случаев, в которых возникает непредсказуемое поведение. Типы не делятся на примитивные и ссылочные. Для функций, которые содержат оператор `return`, но не возвращают значение, не существует типа `void` — в Kotlin функция либо возвращает значение, либо не возвращает ничего. Функции-расширения позволяют добавлять в существующие типы новые операции, которые в момент вызова выглядят точно так же, как старые. Вы можете писать такие управляющие конструкции, как встроенные функции или выражения, а инструкции `break`, `continue` и `return` в них ведут себя так же, как и в стандартных управляющих конструкциях.

- *Чтобы упростить переход, Kotlin нарушает собственные правила.*

  Возможности языка Kotlin позволяют использовать в одном проекте характерный для Kotlin код, написанный как на нем самом, так и на Java. Некоторые из

этих возможностей требуют отказаться от гарантий, которые предоставляются средством проверки типов, и их следует использовать *исключительно* для взаимодействия со старым кодом на Java. Например, `lateinit` открывает дыру в системе типов, чтобы фреймворки для внедрения зависимостей, инициализирующие объекты с помощью рефлексии, могли нарушать границы инкапсуляции, соблюдение которых обычно обеспечивается компилятором. Если вы объявляете свойство как `lateinitvar`, ответственность за то, чтобы оно было инициализировано до того, как его прочитают, ложится на вас. Компилятор не выявит ваши ошибки.

Когда мы, Нэт и Дункан, просматриваем самый давний код, написанный нами на Kotlin, он обычно выглядит словно Java с новым синтаксисом. Мы пришли в мир Kotlin с многолетним опытом программирования на Java, и наши старые привычки сказывались на том, как начать писать код на новом для себя языке. Наш код содержал много ненужных шаблонных участков, плохо использовал стандартную библиотеку и избегал значений `null`, т. к. мы на тот момент еще не привыкли к тому, что работа с этими значениями является безопасной благодаря механизму проверки типов. В нашей команде были программисты на Scala, которых занесло в противоположную сторону, — в их руках язык Kotlin выглядел словно имитация Scala, пытавшаяся подражать Haskell. В то время никто из нас еще не научился использовать Kotlin в соответствии с его устоями.

Наш путь к характерному для Kotlin коду осложнялся тем, что нам все еще нужно было поддерживать в рабочем состоянии код на Java. Знания самого языка оказалось недостаточно. Java и Kotlin имеют разную направленность, и при постепенном переходе с одного языка на другой мы должны уважать особенности *обоих*.

Рефракторинг на Kotlin

В начале нашего пути к Kotlin мы были ответственны за обслуживание и развитие критически важных систем. У нас не было возможности *полностью* сосредоточиться на переносе нашей кодовой базы из Java в Kotlin. Одновременно с переписыванием кода нам приходилось вносить изменения в систему для удовлетворения новых бизнес-потребностей и при этом поддерживать смешанную кодовую базу на Java/Kotlin. Чтобы этот процесс не вышел из-под контроля, каждое вносимое нами изменение было небольшим, что позволяло легко в нем разобраться и отменить его в случае, если оно нарушало работу какого-то компонента. Сначала мы переписывали код с Java на Kotlin, что делало его Java-подобным, но с новым синтаксисом. Затем мы постепенно применяли возможности языка Kotlin, чтобы код становился все более понятным, типобезопасным, компактным и имел композиционную структуру, которую легче изменять без неприятных неожиданностей.

Производя мелкие, безопасные, обратимые изменения, улучшающие архитектуру, мы провели *рефакторинг* из одного языка на другой, соблюдая устои каждого из них.

Рефакторинг *между* языками обычно проходит сложнее, чем рефакторинг *в пределах* одного языка, т. к. соответствующие инструменты плохо с этим справляются или вообще не работают. Перенос логики из одного языка в другой необходимо

осуществлять вручную, что занимает больше времени и представляет собой более рискованную задачу. Барьер между языками затрудняет рефакторинг, поскольку при переписывании кода на одном языке IDE не обновляет зависящий от нее код, написанный на других языках, и не обеспечивает совместимость.

Что делает сочетание Java и Kotlin уникальным, так это (относительно) гладкий переход между этими языками. Благодаря архитектуре языке Kotlin, тому, как он соотносится с платформой JVM, и инвестициям компании JetBrains в инструменты разработки переход с Java на Kotlin и рефакторинг смешанных проектов на этих языках являются почти такими же простыми, как рефакторинг в пределах одной кодовой базы.

Как показывает наш опыт, можно переносить код из Java в Kotlin без отрицательных последствий для продуктивности. И по мере переноса все большего количества кода продуктивность только повышается.

Принципы рефакторинга

Методология рефакторинга претерпела существенные изменения с момента ее начальной популяризации в книге Мартина Фаулера «Refactoring: Improving the Design of Existing Code (Addison Wesley), изданной в 1999 году. Эта книга содержала подробные инструкции даже для простых видов рефакторинга, таких как переименование идентификаторов, но в ней уже отмечалось, что некоторые передовые среды разработки начали предоставлять поддержку автоматического рефакторинга, взяв на себя часть этой тяжелой работы. В наши дни мы ожидаем от своих инструментов автоматизации даже сложных сценариев — таких как извлечение интерфейса или изменение сигнатур функций.

Но рефакторинг редко ограничивается такими отдельными этапами. Теперь, когда эти этапы могут быть автоматизированы, у нас появляются время и энергия для их объединения и внесения в проект серьезных изменений. Когда в пользовательском интерфейсе IDE не предусмотрено специальных действий для крупномасштабных преобразований, которые мы хотим выполнить, нам приходится проводить рефакторинг на более низком уровне путем последовательного внесения изменений. Мы всегда пользуемся автоматическими средствами рефакторинга IDE, если это возможно, и беремся за ручное редактирование текста только тогда, когда IDE не автоматизирует нужные нам преобразования.

Рефакторинг, состоящий в редактировании текста, — утомительный процесс, чреватый ошибками. Чтобы сделать его менее рискованным и скучным, мы сводим его к минимуму. Если нам *необходимо* заниматься ручным редактированием, мы стараемся ограничить его каким-то одним выражением. Таким образом, мы пользуемся автоматическим рефакторингом для преобразования кода, редактируем одно выражение и затем снова выполняем автоматический рефакторинг, чтобы привести код в итоговое состояние, к которому стремимся.

Впервые описывая крупномасштабный рефакторинг, мы пройдемся по всем его этапам и покажем изменения, происходящие на каждом из них. Это занимает довольно много места на странице и требует от читателя дополнительного внимания. Однако на практике такого рода рефакторинг проводится быстро — обычно за насколько секунд или в крайнем случае минут.

Мы полагаем, что процессы рефакторинга, описанные здесь, будут быстро устаревать по мере развития инструментов. Отдельные действия в IDE могут быть переименованы, а некоторые их сочетания реализованы в виде отдельных процедур. Экспериментируйте, чтобы найти в своей среде лучшие способы постепенного и безопасного преобразования кода по сравнению с теми, которые представлены здесь, и затем делитесь ими с окружающим миром.

Подразумеваем хорошее покрытие тестами

Как отмечает Мартин Фаулер в своей книге «Refactoring: Improving the Design of Existing Code»: «Если вы хотите заниматься рефакторингом, важной предпосылкой для этого является наличие надежных тестов». Хорошее покрытие тестами гарантирует, что преобразования кода, направленные исключительно на улучшение архитектуры, не приведут к неожиданным изменениям поведения системы. В этой книге мы исходим из того, что ваш код как следует тестируется. И мы не затрагиваем тему написания автоматических тестов — она рассматривается другими авторами в таких подробностях, которые мы не могли бы предоставить в нашей книге, — например, в книгах: «Test-Driven Development By Example» (Разработка через тестирование) Кента Бека (Addison-Wesley) и «Growing Object-Oriented Software Guided By Tests» (Построение объектно-ориентированного программного обеспечения, руководствуясь тестами) Стива Фримена и Нэта Прайса (Addison-Wesley). Но вместе с тем мы показываем, как применять возможности Kotlin для улучшения наших тестов.

Во время анализа многошаговых преобразований кода мы не всегда упоминаем о том, на каком этапе происходит тестирование. Исходите из того, что наши тесты выполняются после каждого изменения, успешная компиляция которого здесь демонстрируется, каким бы мелким оно ни было.

Если ваша система еще не покрыта тестами как следует, написание тестов задним числом может быть сложным (и затратным), поскольку логика, которую вы хотите тестировать, переплетается с другими аспектами вашей системы. Здесь трудно отделить причину от следствия: чтобы добавить тесты, которые сделают возможным безопасный рефакторинг, нужно провести рефакторинг. Опять же, в других книгах эта тема уже раскрыта подробнее, чем это можно было бы сделать здесь, — например, в книге «Working Effectively with Legacy Code» Майкла Физерса (Pearson).

Больше книг на эти темы приводится в списке литературы, приведенном в конце книги.

Фиксации изменений рассчитаны на Git Bisect

Помимо того что мы не даем четких инструкций о том, когда нужно выполнять тесты, в этой книге вы не найдете ничего конкретного о том, когда следует фиксировать изменения. Исходите из того, что фиксация происходит всякий раз, когда наши изменения, какими бы мелкими они ни были, привносят в код что-то полезное.

Мы знаем, что наш набор тестов не идеален. Если ему не удается выявить ошибку, которую мы случайно сделали, нам нужно найти фиксацию, в которой эта ошибка была добавлена, и как можно скорее ее исправить.

Команда `git bisect` автоматизирует этот поиск. Мы пишем новый тест с демонстрацией ошибки, а `git bisect` ищет в истории изменений первую фиксацию, которая приводит к провалу этого теста.

Если в истории изменений содержатся крупные фиксации, состоящие из мешанины не связанных между собой изменений, польза от `git bisect` будет не такой существенной. Эта команда не может определить, какие именно изменения, внесенные в рамках одной фиксации, привели к ошибке. Если фиксация включает в себя рефакторинг *и* изменение поведения, откат неудачного рефакторинга с большой долей вероятности нарушит работу *других* частей системы.

В связи с этим мы фиксируем небольшие, узконаправленные изменения, которые разделяют разные этапы рефакторинга друг от друга и от изменений поведения. Это помогает понять, что поменялось, и исправить любые ошибочные изменения. По той же причине мы очень редко удаляем коммиты.

Мы предпочитаем сохранять изменения прямо в главной ветви (trunk-based development, разработка на основе главной ветви), однако редактирование кода путем мелких, независимых фиксаций является таким же полезным и при работе с несколькими ветвями, которые объединяются не так часто.

Над чем мы работаем?

В следующих главах используются примеры из проекта `Travelator` — вымышленного приложения для планирования международных наземных путешествий с возможностью бронирования. Наши (тоже вымышленные) пользователи составляют маршруты следования кораблями, поездами и автомобилями, ищут, где бы остановиться на ночь и на что бы посмотреть, сравнивают разные варианты по цене, времени и живописности и, наконец, бронируют билеты и номера в отелях. И все это делается с помощью мобильных и веб-приложений, которые обращаются к серверным компонентам по HTTP.

В каждой главе приводится информативный пример на основе определенной части системы `Travelator`, но во всех этих примерах используются общие понятия предметной области — такие как деньги, перевод из одной валюты в другую, путешествия, маршруты, бронирование и т. д.

Цель этой книги, как и нашего приложения `Travelator`, состоит в том, чтобы помочь вам в планировании вашего путешествия из Java в Kotlin.

Приступим!

Хватит болтать. Вам уже, наверное, не терпится переписать весь свой код с Java на Kotlin. В следующей главе мы сделаем первый шаг в этом направлении: добавим поддержку Kotlin в файл сборки нашего проекта.

ГЛАВА 2

Перевод проектов с Java на Kotlin

С чего начать переход от проекта на чистом языке Java к смешанной кодовой базе с постепенным движением в сторону Kotlin?

Стратегия

Когда мы, Нэт и Дункан, впервые начали внедрять Kotlin в проект, написанный на Java, наша небольшая команда состояла из шести разработчиков, а сам проект содержал относительно мало старого кода. Мы уже имели некоторый опыт развертывания веб-приложений на Kotlin, однако наши корпоративные архитекторы программного обеспечения настояли на том, чтобы новая система была написана на Java 8. Это произошло вскоре после выхода Kotlin 1.0, но еще до того, как компания Google сделала Kotlin официальным языком разработки для Android, поэтому архитекторы по понятным причинам не спешили применять язык с туманным будущим для создания системы, время жизни которой, как они полагали, будет исчисляться десятилетиями.

В Java мы склонялись к функциональному подходу, моделируя базовую предметную область приложения в виде неизменяемых типов данных, которые преобразуются с помощью пайплайнов (конвейеров). Но постоянно упирались в ограничения языка — такие как слишком многословные реализации типов неизменяемых значений, разделение между примитивными и ссылочными типами, null-ссылки и отсутствие распространенных функций верхнего уровня для работы с потоками данных. Тем временем мы наблюдали за постоянно растущими темпами внедрения Kotlin, в том числе и внутри компании. Так что, увидев анонс от Google, мы все же решили начать перевод нашего Java-кода на Kotlin.

Логика перевода заключалась в том, что наибольшую пользу должно было принести преобразование базовой модели предметной области. Классы данных в Kotlin делали код намного компактнее, иногда давая возможность заменить сотни строк кода одним-единственным объявлением. Вначале мы действовали осторожно, используя IntelliJ для преобразования небольших классов значений, которые зависели только от стандартной библиотеки, и анализируя влияние этих изменений на остальной код. Никаких неприятных последствий это не вызывало, и вдохновленные таким успехом, мы начали двигаться ускоренными темпами. Всякий раз, когда новая функция требовала изменений в классе модели домена Java, мы сначала преобразовывали ее в класс данных Kotlin, фиксировали преобразование, а затем внедряли функцию.

По мере того как все большая часть логики модели предметной области становилась написанной на чистом Kotlin, мы могли лучше пользоваться возможностями этого языка. Например, заменили обращения к Stream API стандартными функциями Kotlin для работы с коллекциями и последовательностями. Однако самым значительным улучшением была замена типа `Optional` из Java обнуляемыми ссылками. Это упростило код и укрепило нашу уверенность в том, что обращение со значениями `null` происходит безопасно.

У нашей компании имелся еще один проект, перешедший на Kotlin по другой причине. Это была зрелая Java-система, основанная на фреймворке для внедрения зависимостей. Разработчики обнаружили, что то, как этот фреймворк использует рефлексию и аннотации, затрудняло чтение кода и навигацию в IDE. Легковесный синтаксис замыканий, доступный в Kotlin, позволил им описать структуру своего проекта и разграничить объектные графы для всего приложения, для каждого HTTP-запроса и для каждой транзакции в базе данных. Они постепенно переписали основные части своей системы с фреймворка, скрывавшего ее архитектуру, на язык с поддержкой композиции функций, который ясно выражал архитектуру в коде. Этот проект в итоге перерос в набор инструментов для веб-программирования http4k (**https://http4k.org**).

Как следует их этих двух примеров, выбор того, с чего начать, должен зависеть от ряда факторов: зачем ваша команда внедряет Kotlin, насколько большая у вас кодовая база, как часто она изменяется и т. д. Вы знаете свой проект и можете решить, что в нем следует изменить в первую очередь.

Если выбор Kotlin мотивирован возможностями языка, начинать переход имеет смысл с классов, над которыми вы чаще всего работаете. В нашем первом проекте мы так и поступили. Если ваш выбор продиктован использованием определенной библиотеки, вам лучше начать с кода, который взаимодействует с API-интерфейсом, — задействуйте аннотации, чтобы ваш код было удобно использовать из Java в остальной части приложения, и двигайтесь дальше.

В небольшой команде легко принять стиль написания системы на Kotlin (в дополнение к стандартному руководству по стилям), включая соглашения об обработке ошибок, структуру файлов, выбор объявления верхнего уровня, содержимое объектов и т. д.

Но если ваш проект превышает определенный размер, вы рискуете получить разнородный код на Kotlin, т. к. ваши коллеги могут использовать разные соглашения в разных частях системы. Поэтому лучше, наверное, начинать переход с одной подгруппы разработчиков, отвечающих за какой-то один участок кода, — они должны выработать соглашения и сформировать кодовую базу, которая будет служить примером для остальных. После этого вы сможете привлечь к своей инициативе других членов команды и другие части системы.

С этого момента мы начнем подробное рассмотрение процесса перехода. Вы узнаете, как сделать так, чтобы ваш код на Java оставался удобным в обслуживании, пока его зависимости переписываются на Kotlin, и как воспользоваться возможностя-

ми Kotlin, чтобы продолжить упрощение кода после того, как среда IntelliJ проделала свои магические преобразования. Но сначала нужно сделать первый шажок.

Добавление поддержки Kotlin в сборку Java

Если мы хотим переписать наш Java-код на Kotlin, первое изменение, которое мы должны сделать, — дать себе возможность писать код Kotlin в нашей кодовой базе. К счастью, средства сборки и IDE делают этот этап крайне простым. Достаточно добавить несколько строчек в конфигурацию сборки Gradle, чтобы компилировался не только код на Java, но и код на Kotlin. IntelliJ подхватит эту конфигурацию во время повторной синхронизации файла сборки, что сделает легкодоступными такие возможности, как навигация, автодополнение и рефакторинг в рамках обоих языков.

Чтобы добавить Kotlin в сборку Gradle, нужно воспользоваться специальным дополнением (плагином). Для каждой цели, которую поддерживает Kotlin (JVM, JavaScript и машинный код), равно как и для мультиплатформенных проектов, существуют отдельные дополнения. Поскольку наш проект написан на Java, мы можем игнорировать другие платформы и выбрать дополнение Kotlin JVM.

Нам также нужно добавить в число наших зависимостей стандартную библиотеку Kotlin и указать минимальную версию JVM, которую будет поддерживать итоговый байт-код. Наш проект рассчитан на JDK 11 (последний выпуск с долгосрочной поддержкой на момент подготовки этой книги). В настоящий момент компилятор Kotlin способен генерировать байт-код, совместимый с JDK 1.6 или JDK 1.8. Байт-код JDK 1.8 более эффективен и без проблем работает в JDK 11, поэтому мы выберем его.

Версии Kotlin

Язык и стандартная библиотека Kotlin все еще активно развиваются, но компания JetBrains стремится предоставить четкий путь миграции. Когда мы взялись за написание этой книги, текущей версией Kotlin была 1.3. А когда заканчивали, вышел Kotlin 1.5, и некоторые стандартные API-интерфейсы, используемые в наших примерах кода, оказались устаревшими! Мы решили не переходить на то, что пришло им на смену, поэтому наш код может работать в Kotlin 1.4 и 1.5.

Вот важные части нашего файла build.gradle до внесения изменений (пример 2.1).

Пример 2.1 [projects.0:build.gradle]

```
plugins {
    id("java")
}

java.sourceCompatibility = JavaVersion.VERSION_11
java.targetCompatibility = JavaVersion.VERSION_11
... and other project settings ...

dependencies {
    implementation "com.fasterxml.jackson.core:jackson-databind:2.10.0"
```

```
    implementation "com.fasterxml.jackson.datatype:jackson-datatype-jsr310:2.10.0"
    implementation "com.fasterxml.jackson.datatype:jackson-datatype-jdk8:2.10.0"
    ... and the rest of our app's implementation dependencies

    testImplementation "org.junit.jupiter:junit-jupiter-api:5.4.2"
    testImplementation "org.junit.jupiter:junit-jupiter-params:5.4.2"
    testRuntimeOnly "org.junit.jupiter:junit-jupiter-engine:5.5.2"
    testRuntimeOnly "org.junit.platform:junit-platform-launcher:1.4.2"
    ... and the rest of our app's test dependencies
}

... и прочий код
```

После добавления поддержки Kotlin наш файл сборки стал выглядеть так (пример 2.2).

Пример 2.2 [projects.1:build.gradle]

```
plugins {
    id 'org.jetbrains.kotlin.jvm' version "1.5.0"
}

java.sourceCompatibility = JavaVersion.VERSION_11
java.targetCompatibility = JavaVersion.VERSION_11
... and other project settings ...

dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
    ... and the rest of our app's dependencies
}

tasks.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompile) {
    kotlinOptions {
        jvmTarget = "11"
        javaParameters = true
        freeCompilerArgs = ["-Xjvm-default=all"]
    }
}

... и прочий код
```

Учитывая эти изменения, мы можем перезапустить процесс сборки и увидеть, что...

...все по-прежнему работает!

После выполнения повторной синхронизации проекта Gradle в IntelliJ (это может произойти автоматически при сохранении) мы сможем выполнять наши тесты и программы внутри IDE.

Наши тесты по-прежнему выполняются успешно, т. е. ничего не сломалось. Но мы тем не менее так и не доказали, что в нашем проекте можно использовать Kotlin. Чтобы это проверить, напишем программу «hello, world», для чего создадим в корневом пакете нашего дерева исходного кода на Java (src/main/java) файл HelloWorld.kt (пример 2.3).

Пример 2.3 [projects.2:src/main/java/HelloWorld.kt]

```
fun main() {
    println("hello, world")
}
```

Где хранить исходный код на Kotlin?

Дополнение для сборки Kotlin добавляет новые корневые каталоги для исходников: *src/main/kotlin* и *src/test/kotlin* — и компилирует исходные файлы Kotlin, найденные в их подкаталогах.

Оно также компилирует исходники Kotlin, хранящиеся в дереве исходного кода на Java — в частности: *src/main/java* и *src/test/java*. Вы можете разделить свои исходные файлы по языку, поместив Java-файлы в каталоге *java*, а Kotlin-файлы — в каталоге *kotlin*, но на практике ваши авторы этим не заморачиваются. Приятно, когда можно заглянуть в каталог и увидеть все исходные файлы соответствующего пакета, вместо того чтобы искать их по файловой системе. Однако для этого мы храним исходники Kotlin в каталогах, повторяющих структуру пакета, несмотря на то, что Kotlin позволяет размещать файлы разных пакетов в одном каталоге.

Аналогичным образом, при одновременном использовании в проекте Java и Kotlin мы обычно размещаем каждый класс в отдельном файле, чтобы сделать код согласованным, хотя Kotlin позволяет определять в классе множество публичных классов.

Мы можем выполнить этот код внутри IDE, щелкнув на небольшой зеленой стрелке в поле слева от `fun main()`.

Выполняем сборку и затем запускаем полученный результат из командной строки с помощью команды `java`. В результате компиляции файла HelloWorld.kt получится class-файл Java с именем HelloWorldKt. Подробнее о том, как исходный код на Kotlin переводится в class-файлы Java, мы поговорим позже, а сейчас можно запустить программу, используя команду `java`, как показано здесь:

```
$ java -cp build/classes/kotlin/main HelloWorldKt
hello, world
```

Она ожила!

Давайте удалим файл HelloWorld.kt (он уже сыграл свою роль), зафиксируем и загрузим изменения.

Теперь у нас есть *возможность* использовать Kotlin в нашем проекте — в первой части этой главы даются некоторые указания о том, *где* в первую очередь следует применять этот язык.

Другие системы сборки

Здесь мы решили показать, какие изменения нужно внести в сборку Gradle, чтобы добавить поддержку Kotlin, но вы можете использовать Maven или Ant, следуя инструкциям, предоставленным в разделе **Tools** (Инструменты) документации Kotlin (**https://oreil.ly/bWi9n**). Там же можно найти инструкции по использованию компилятора командной строки `kotlinc`.

Если вы работаете с Gradle, в качестве языка описания сборки вместо традиционного Groovy можно использовать Kotlin. Это обеспечит лучшую поддержку инструментария за счет строгой типизации, но, с другой стороны, вам придется переводить исторические ответы, которые вы будете находить на Stack Overflow, на новый язык.

Поскольку мы являемся разработчиками Java и Kotlin, а не Java и Groovy, то начинаем новые проекты с Kotlin DSL, но при этом не ощущаем необходимости конвертировать существующие сборки Groovy — по крайней мере, не сразу. Мы можем смешивать и сочетать в сборках Kotlin и Groovy, как это делается в рабочем коде с Java и Kotlin, поэтому с таким преобразованием можно не торопиться. Мы не советуем вам переводить вашу сборку с Groovy на Kotlin в качестве первого этапа рефакторинга и уж точно не планируем писать книгу о переводе Gradle с одного языка на другой!

Двигаемся дальше

Мы полагаем, что технический материал, представленный в этой главе, очень быстро утратит свою актуальность, поскольку и система Gradle, и ее дополнения имеют не очень стабильный интерфейс. Ваш текущий файл сборки Java тоже почти наверняка несовместим с нашим примером в каких-то важных аспектах. Тем не менее процесс *добавления* Kotlin в сборку Java в целом является простым и понятным.

Разработать стратегию *переноса* кода из Java в Kotlin будет сложнее, т. к. она зависит от контекста. Или эта сложность станет как минимум варьировать в каждом отдельном случае. Целесообразность применения Java следует оценивать для конкретного проекта, равно как и способность Kotlin решить имеющиеся проблемы и улучшить качество кода. Возможно, вы решите писать на Kotlin с самого начала, или же у вас может быть готовый Java-класс, который нужно преобразовать в Kotlin. В *главе 3 «От классов Java к классам Kotlin»* мы рассмотрим именно второй случай, что соответствует духу этой книги

ГЛАВА 3

От классов Java к классам Kotlin

Класс — основной элемент организации кода в Java. Как преобразовать наши Java-классы в Kotlin и что при этом изменится?

В этой книге мы будем вместе с вами работать над кодом `Travelator` — нашего вымышленного веб-приложения для планирования путешествий. Представьте, что нам нужно реализовать какую-то возможность, но, прежде чем это делать, мы хотим немного улучшить наш код. Вы будете работать в паре с Нэтом или Дунканом (выберите того, кто вам больше по душе, только не говорите Нэту). *Пара* — это то, что имеется в виду под *мы* в наших обсуждениях рефакторинга, — вы будете заниматься проектом `Travelator` наравне с его авторами. Добро пожаловать в команду!

Исходный код

Исходный код `Travelator` опубликован в общедоступном репозитории. В *разд. «Использование примеров кода» предисловия* объясняется, как с ним работать.

Поскольку эта книга посвящена рефакторингу, изменениям (как долгосрочным, так и между отдельными фиксациями) придается большое значение. Мы пытались показать вам все необходимое для того, чтобы вы могли разобраться в напечатанном здесь коде, — когда речь идет об изменениях, которые мы собираемся вносить, представьте, что последний пример кода открыт в активной панели редактора. Если нам не удается как следует описать свои действия, или вам потребуется больше подробностей, можете скачать этот код и повторить все этапы в IntelliJ.

Простой тип значений

Давайте сразу и без лишних слов возьмемся за наш проект и перепишем часть имеющегося кода с Java на Kotlin. Начнем с `EmailAddress`. Это тип значений, хранящий две части, — чего бы вы думали? Правильно, адреса электронной почты (пример 3.1).

Пример 3.1 [classes.0:src/main/java/travelator/EmailAddress.java]

```
public class EmailAddress {
    private final String localPart; ❶
    private final String domain;

    public static EmailAddress parse(String value) { ❷
        var atIndex = value.lastIndexOf('@');
```

```
        if (atIndex < 1 || atIndex == value.length() - 1)
            throw new IllegalArgumentException(
                "EmailAddress must be two parts separated by @"
            );
        return new EmailAddress(
            value.substring(0, atIndex),
            value.substring(atIndex + 1)
        );
    }

    public EmailAddress(String localPart, String domain) { ❸
        this.localPart = localPart;
        this.domain = domain;
    }

    public String getLocalPart() { ❹
        return localPart;
    }

    public String getDomain() { ❹
        return domain;
    }

    @Override
    public boolean equals(Object o) { ❺
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        EmailAddress that = (EmailAddress) o;
        return localPart.equals(that.localPart) &&
            domain.equals(that.domain);
    }

    @Override
    public int hashCode() { ❺
        return Objects.hash(localPart, domain);
    }

    @Override
    public String toString() { ❻
        return localPart + "@" + domain;
    }
}
```

Этот класс очень прост — он всего лишь служит оберткой для двух строк и не предоставляет никаких собственных операций. Несмотря на это, в нем содержится большое количество кода:

❶ Значения не подлежат изменению, поэтому поля класса объявлены как `final`.

❷ Это статический фабричный метод для преобразования строки в `EmailAddress`. Он вызывает первичный конструктор.

❸ Поля инициализируются в конструкторе.

❹ Методы доступа к свойствам соответствуют соглашениям об именовании JavaBeans.

❺ Этот класс реализует методы `equals` и `hashCode`, чтобы при сравнении два значения `EmailAddress` с одинаковыми полями считались равными.

❻ `toString` возвращает адрес в каноническом виде.

Авторы этой книги относятся к числу тех Java-разработчиков, которые предполагают, что все, что передается, сохраняется и возвращается и равно `null` только в случае, если это явно указано. Это соглашение незаметно, поскольку оно выражается в отсутствии аннотаций `@Nullable` и проверок параметров на `null` (обнуляемость рассматривается в *главе 4*). Тем не менее вы *можете видеть*, сколько шаблонного кода необходимо для описания значения, состоящего из двух других значений. К счастью, наша IDE сгенерировала для нас методы `equals` и `hashCode`, но, если вы изменяете поля класса, не забудьте их удалить и сгенерировать заново, чтобы избежать запутанных ошибок.

Но хватит о Java, ведь мы здесь ради Kotlin. Как преобразовать этот код? IntelliJ любезно предоставляет действие под названием **Convert Java File to Kotlin File** (Преобразовать файл Java в файл Kotlin). Если его инициировать, IntelliJ предложит изменить при необходимости другие файлы, чтобы сохранить совместимость. Поскольку в результате этого процесса в вашем проекте могут быть изменены разные файлы, вам лучше согласиться.

Прежде чем переводить исходный код Java на Kotlin, убедитесь в том, что у вас нет незафиксированных изменений. Тогда вы сможете с легкостью увидеть то, как это преобразование повлияло на остальную часть вашего проекта, и откатить изменения, если что-то пойдет не так, как вы ожидали.

В нашем случае среде IntelliJ не нужно модифицировать никакие другие файлы — она создала в том же каталоге файл EmailAddress.kt вместо EmailAddress.java (пример 3.2).

Пример 3.2 [classes.2:src/main/java/travelator/EmailAddress.kt]

```
class EmailAddress(val localPart: String, val domain: String) {
    override fun equals(o: Any?): Boolean {
        if (this === o) return true
        if (o == null || javaClass != o.javaClass) return false
        val that = o as EmailAddress
        return localPart == that.localPart && domain == that.domain
    }
```

```
    override fun hashCode(): Int {
        return Objects.hash(localPart, domain)
    }

    override fun toString(): String {
        return "$localPart@$domain"
    }

    companion object {
        @JvmStatic
        fun parse(value: String): EmailAddress {
            val atIndex = value.lastIndexOf('@')
            require(!(atIndex < 1 || atIndex == value.length - 1)) {
                "EmailAddress must be two parts separated by @"
            }
            return EmailAddress(
                value.substring(0, atIndex),
                value.substring(atIndex + 1)
            )
        }
    }
}
```

Класс Kotlin получился заметно компактнее, т. к. его свойства объявляются в первичном конструкторе, а параметры — после имени класса. Параметры, помеченные как `val`, считаются свойствами и заменяют собой весь этот код на Java (пример 3.3).

Пример 3.3 [classes.1:src/main/java/travelator/EmailAddress.java]

```
private final String localPart;
private final String domain;

public EmailAddress(String localPart, String domain) {
    this.localPart = localPart;
    this.domain = domain;
}

public String getLocalPart() {
    return localPart;
}

public String getDomain() {
    return domain;
}
```

Синтаксис первичного конструктора удобен, но отрицательно сказывается на удобочитаемости класса. В Java классы, соблюдающие стандартные соглашения о на-

писании кода, всегда объявляют свои элементы в одном и том же порядке: имя класса, родительский класс, интерфейсы, и потом, в теле класса, идут поля, конструкторы и методы. Это позволяет быстро находить интересующие вас возможности.

В классе, написанном на Kotlin, не так легко найти отдельные элементы. У определения класса есть заголовок с именем, первичным конструктором (который может содержать параметры и/или определения свойств), родительский класс (который также может представлять собой вызов конструктора родительского класса) и интерфейсы. Затем в теле могут находиться дополнительные свойства, конструкторы, методы и объекты-компаньоны.

Нэт и Дункан пришли из мира Java, поэтому вначале у них определенно возникали трудности с чтением классов, и, хотя в итоге они привыкли к этому синтаксису, им иногда по-прежнему непросто форматировать свои классы с расчетом на максимальную удобочитаемость, особенно если в заголовке много чего происходит. В качестве простого решения можно разбить список параметров конструктора по строкам. Чтобы это сделать, разместите текстовый курсор в списке параметров, нажмите комбинацию клавиш <Alt>+<Enter> и выберите опцию **Put parameters on separate lines** (Разместить параметры в отдельных строках). Иногда также помогает пустая строчка после заголовка (пример 3.4).

Пример 3.4 [classes.3:src/main/java/travelator/EmailAddress.kt]

```
class EmailAddress(
    val localPart: String,
    val domain: String
) {

    override fun equals(o: Any?): Boolean {
        if (this === o) return true
        if (o == null || javaClass != o.javaClass) return false
        val that = o as EmailAddress
        return localPart == that.localPart && domain == that.domain
    }

    override fun hashCode(): Int {
        return Objects.hash(localPart, domain)
    }

    override fun toString(): String {
        return "$localPart@$domain"
    }

    companion object {
        @JvmStatic
        fun parse(value: String): EmailAddress {
            val atIndex = value.lastIndexOf('@')
```

```
            require(!(atIndex < 1 || atIndex == value.length - 1)) {
                "EmailAddress must be two parts separated by @"
            }
            return EmailAddress(
                value.substring(0, atIndex),
                value.substring(atIndex + 1)
            )
        }
    }
}
```

Одной из областей, где Kotlin заметно уступает Java в лаконичности, является использование объектов-компаньонов для размещения статических свойств и методов — в нашем случае `parse()`. В Kotlin мы зачастую предпочитаем состояние и функции верхнего уровня вместо этих членов класса. Плюсы и минусы обоих подходов обсуждаются в *главе 8*.

В настоящий момент наш код на Java использует статический метод `parse` — как, например, в этих тестах (пример 3.5).

Пример 3.5 [classes.0:src/test/java/travelator/EmailAddressTests.java]

```
public class EmailAddressTests {

    @Test
    public void parsing() {
        assertEquals(
            new EmailAddress("fred", "example.com"),
            EmailAddress.parse("fred@example.com")
        );
    }

    @Test
    public void parsingFailures() {
        assertThrows(
                IllegalArgumentException.class,
                () -> EmailAddress.parse("@")
        );
        ...
    }

    ...
}
```

Наличие объекта-компаньона в сочетании с аннотацией `@JVMStatic` означает, что этот код можно не менять при переводе класса на Kotlin, поэтому мы пока что оставим метод `parse` без изменений. О том, как выполнять рефакторинг функций верхнего уровня, речь пойдет в *главе 8*.

Если у вас нет опыта работы с Kotlin, вы можете задаться вопросом: что случилось с методами доступа `getLocalPart()` и `getDomain()`? Объявление свойства `domain` заставляет компилятор сгенерировать частное поле с тем же именем и метод `getDomain()`, чтобы код на Java по-прежнему мог к нему обращаться. В примере 3.6 показан импровизированный код для поддержки маркетингового плана.

Пример 3.6 [classes.3:src/main/java/travelator/Marketing.java]

```
public class Marketing {

    public static boolean isHotmailAddress(EmailAddress address) {
        return address.getDomain().equalsIgnoreCase("hotmail.com");
    }
}
```

Как видите, Java обращается к свойству `domain` через метод `getDomain()`. Если бы у нас был код на Java с явно определенным методом `getDomain()`, обращение к нему из кода на Kotlin выглядело бы как `address.domain`. Мы подробнее обсудим свойства в *главе 11*.

Пока что преобразование нашего класса в Kotlin сэкономило нам 14 строчек кода, но мы еще не закончили. Подобные типы значений крайне полезны, но для их правильного написания и поддержки требуется много усилий, поэтому Kotlin поддерживает их на уровне языка. Если пометить класс с помощью модификатора `data`, компилятор автоматически сгенерирует все методы `equals`, `hashCode` и `toString`, которые мы сами не определили. Это делает класс `EmailAddress` еще компактнее (пример 3.7).

Пример 3.7 [classes.4:src/main/java/travelator/EmailAddress.kt]

```
data class EmailAddress(
    val localPart: String,
    val domain: String
) {

    override fun toString(): String { ❶
        return "$localPart@$domain"
    }

    companion object {
        @JvmStatic
        fun parse(value: String): EmailAddress {
            val atIndex = value.lastIndexOf('@')
            require(!(atIndex < 1 || atIndex == value.length - 1)) {
                "EmailAddress must be two parts separated by @"
            }
```

```
            return EmailAddress(
                value.substring(0, atIndex),
                value.substring(atIndex + 1)
            )
        }
    }
}
```

❶ Нам не нужен сгенерированный метод `toString()`, поэтому мы определяем его так, как нам хочется.

Если честно, то метод `parse` все еще не дает нам покоя — он занимает непропорционально много места, учитывая его скромные обязанности. В *главе 9* мы его в конце концов исправим. А пока что преобразование Java-класса `EmailAddress` в Kotlin завершено.

Ограничения классов данных

Недостаток классов данных состоит в том, что они не предоставляют инкапсуляцию. Мы видели, как компилятор генерирует для них методы `equals`, `hashCode` и `toString`, но не упомянули о методе `copy`, который тоже генерируется и создает новую копию значения, изменяя при этом одно или несколько его свойств.

Например, следующий код создает копию `EmailAddress` со свойством `localPart`, равным `"postmaster"` и тем же значением `domain`:

```
val postmasterEmail = customerEmail.copy(localPart = "postmaster")
```

Для многих типов это удобно. Но, когда класс скрывает детали своей реализации или поддерживает инварианты для своих свойств, этот метод `copy` дает клиентскому коду прямой доступ к внутреннему состоянию значения, что может нарушить его инварианты.

Рассмотрим абстрактный тип данных в классе `Money` приложения `Travelator` (пример 3.8).

Пример 3.8 [values.4:src/main/java/travelator/money/Money.java]

```
public class Money {
    private final BigDecimal amount;
    private final Currency currency;

    private Money(BigDecimal amount, Currency currency) { ❶
        this.amount = amount;
        this.currency = currency;
    }

    public static Money of(BigDecimal amount, Currency currency) { ❶
        return new Money(
```

```
            amount.setScale(currency.getDefaultFractionDigits()),
            currency);
    }

    ... и удобные способы перезагрузки операторов

    public BigDecimal getAmount() { ❷
        return amount;
    }

    public Currency getCurrency() { ❷
        return currency;
    }

    @Override
    public boolean equals(Object o) { ❸
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Money money = (Money) o;
        return amount.equals(money.amount) &&
            currency.equals(money.currency);

    @Override
    public int hashCode() { ❸
        return Objects.hash(amount, currency);
    }

    @Override
    public String toString() { ❹
        return amount.toString() + " " + currency.getCurrencyCode();
    }

    public Money add(Money that) { ❺
        if (!this.currency.equals(that.currency)) {
            throw new IllegalArgumentException(
                "cannot add Money values of different currencies");
        }

        return new Money(this.amount.add(that.amount), this.currency);
    }
}
```

❶ Конструктор является `private`. Другие классы получают значения `Money` путем вызова статического метода `Money.of`, который гарантирует, что величина суммы соответствует количеству разменных денежных единиц валюты. Большинство валют имеют сто разменных единиц (две цифры), но бывают валюты, в которых

их больше или меньше. Например, у японской иены нет разменных денежных единиц, а иорданский динар состоит из тысячи филсов.

Метод `of` соответствует соглашению о написании кода в современных версиях Java, в которых существует различие между идентифицируемыми объектами, созданными с помощью оператора `new`, и значениями, полученными из статических методов. Это соглашение соблюдаются в API-интерфейсе Java для работы с временем (например, `LocalDate.of(2020,8,17)`) и в последних дополнениях к Collections API (например, `List.of(1,2,3)` создает неизменяемый список).

Этот класс предоставляет несколько удобных перегруженных версий метода `of` для строковых и целочисленных сумм.

❷ Значение `Money` делает доступными свои свойства `amount` и `currency` в соответствии с соглашениями JavaBeans, хотя эта технология здесь не используется.

❸ Методы `equals` и `hashCode` реализуют семантику значения.

❹ Метод `toString` предназначен не только для отладки — он возвращает свойства в том виде, в котором их можно показывать пользователю.

❺ `Money` предоставляет операции для вычисления денежных значений. Например, вы можете сложить вместе две денежные суммы. Метод `add` создает новые значения `Money`, напрямую вызывая конструктор (вместо того чтобы использовать `Money.of`), т. к. результат `BigDecimal.add` уже имеет корректную величину. Таким образом можно избежать накладных расходов, связанных с заданием величины в `Money.of`.

Метод `BigDecimal.setScale` может вызвать путаницу. Он назван как сеттер в JavaBeans, но при этом не изменяет объект `BigDecimal`. По примеру наших классов `EmailAddress` и `Money`, `BigDecimal` является неизменяемым типом значений, поэтому `setScale` возвращает новый экземпляр `BigDecimal` с заданной величиной.

Компания Sun добавила класс `BigDecimal` в стандартную библиотеку Java 1.1. В этом выпуске также появилась первая версия JavaBeans API. Ажиотаж вокруг этого API-интерфейса способствовал популяризации принятых в нем соглашений о написании кода. В результате эти соглашения получили широкое распространение даже в таких классах, как `BigDecimal`, не относящихся к JavaBeans (см. *разд. «Стиль JavaBeans» главы 1*). В то время в Java не существовало правил оформления типов значений.

В настоящее время мы избегаем префикса `set` в методах, которые не изменяют их объект-получатель, и вместо этого используем имена, подчеркивающие тот факт, что метод возвращает видоизмененную версию получателя. Для преобразований, затрагивающих какое-то одно свойство, обычно используется префикс `with`. В нашем случае код в классе `Money` выглядел бы так:

```
amount.withScale(currency.getDefaultFractionDigits())
```

В Kotlin для исправления таких исторических недоразумений можно использовать функции-расширения. Если у вас есть много кода, выполняющего вычисления с помощью `BigDecimal`, это поможет сделать его более ясным:

```
fun BigDecimal.withScale(int scale, RoundingMode mode) =
    setScale(scale, mode)
```

Преобразование `Money` в Kotlin дает следующий код (пример 3.9).

Пример 3.9 [values.5:src/main/java/travelator/money/Money.kt]

```
class Money
private constructor(
    val amount: BigDecimal,
    val currency: Currency
)
{
    override fun equals(o: Any?): Boolean {
        if (this === o) return true
        if (o == null || javaClass != o.javaClass) return false
        val money = o as Money
        return amount == money.amount && currency == money.currency
    }

    override fun hashCode(): Int {
        return Objects.hash(amount, currency)
    }

    override fun toString(): String {
        return amount.toString() + " " + currency.currencyCode
    }

    fun add(that: Money): Money {
        require(currency == that.currency) {
            "cannot add Money values of different currencies"
        }
        return Money(amount.add(that.amount), currency)
    }

    companion object {
        @JvmStatic
        fun of(amount: BigDecimal, currency: Currency): Money {
            return Money(
                amount.setScale(currency.defaultFractionDigits),
                currency
            )
        }

        ... и удобные способы перезагрузки операторов
    }
}
```

В Kotlin у этого класса по-прежнему есть первичный конструктор, но теперь он помечен как `private`. Для этого предусмотрен немного неуклюжий синтаксис — мы

переформатировали код, сгенерированный транслятором, в попытке сделать его более удобочитаемым. Как и `EmailAddress.parse`, статические фабричные функции `of` превратились в методы объекта-компаньона с аннотацией `@JvmStatic`. В целом код получился ненамного компактнее по сравнению с оригиналом на Java.

Возможно, если превратить его в класс данных, он все же станет меньше?

Если поменять `class` на `data class`, IntelliJ выведет для ключевого слова `private` в первичном конструкторе предупреждение:

> Private data class constructor is exposed via the generated 'copy' method.

Что это означает?

В реализации `Money` кроется одна важная деталь. У свойств этого класса есть инвариант, гарантирующий, что величина поля `amount` совпадает с количеством цифр разменной единицы, хранящимся в поле `currency`. Частный конструктор не позволяет коду за пределами класса `Money` создавать значения, нарушающие этот инвариант. Метод `Money.of(BigDecimal,Currency)` обеспечивает выполнение инварианта для новых экземпляров `Money`. То же можно сказать и о методе `add`, поскольку результат сложения двух значений `BigDecimal` с одинаковой величиной имеет ту же величину, что позволяет этому методу вызывать конструктор напрямую. Следовательно, конструктору нужно просто инициализировать поля, т. к. он никогда не будет вызван с параметрами, нарушающими инвариант класса.

А вот метод `copy`, принадлежащий классу данных, всегда остается публичным и, следовательно, *позволяет* клиентскому коду создавать значения, нарушающие инвариант. В отличие от `EmailAddress`, абстрактные типы данных, такие как класс `Money`, нельзя реализовать в виде классов данных Kotlin.

Если тип значений должен поддерживать инварианты для своих свойств, не объявляйте его в виде класса данных.

В следующих главах мы познакомимся с возможностями Kotlin, которые позволят сделать этот класс еще лаконичнее и удобнее. А пока оставим его в покое и вернемся к нему *в главе 12*, где он заметно преобразится.

Двигаемся дальше

Большинство классов на Java можно быстро и легко перевести в Kotlin. Результат будет полностью совместимым с существующим Java-кодом.

Если нам нужна семантика значений, то из таких простых классов, как `EmailAddress`, превратив их в классы данных, можно убрать еще большее количество шаблонного кода. Классы данных не нужно обслуживать, а их создание не требует много времени и усилий, поэтому в Kotlin мы используем их для определения типов значений куда чаще, чем в Java. Так мы объявляем «микротипы» уровня приложения, служащие обертками для примитивных значений, храним промежуточные резуль-

таты в процессе вычислений и размещаем данные во временных структурах, упрощающих написание бизнес-логики.

Если же ваши типы значений должны поддерживать инварианты или инкапсулировать свое представление, классы данных вам не подойдут. В таких случаях приходится самостоятельно реализовывать семантику значений.

`EmailAddress` и `Money` все еще выглядят несколько Java-подобно... по-Java-вски?... в духе Java? ... Неважно. В следующих главах речь пойдет о том, как с помощью идиом Kotlin сделать код более компактным, типобезопасным и удобным для дальнейшей разработки. В *главе 9 «От многострочных функций к однострочным»*, показано, как сделать вычислительные функции и методы — такие как метод `toString` в обоих классах или `equals` и `hashCode` в `Money` — более лаконичными путем их преобразования в единое выражение. В *главе 12 «От функций к операторам»* мы повысим удобство использования типа `Money` в Kotlin за счет определения операторов вместо методов.

Не все наши Java-классы являются типами значений. Стиль написания кода, преобладающий в Java, ориентирован на изменяемые объекты. В *главе 5 «От объектов JavaBeans к значениям»* мы рассмотрим преимущества типов значений в ситуациях, в которых в Java предполагали бы использование изменяемых объектов, и покажем, как осуществить переход от таких объектов к изменяющим значениям.

В коде на Java существует множество случаев, оправдывающих использование статических служебных методов. В Kotlin функции и данные являются полноценными элементами языка. Их не нужно объявлять членами классов. В *главе 8 «От статических методов к функциям верхнего уровня»* исследуется процесс преобразования Java-классов со служебными методами в объявления верхнего уровня.

ГЛАВА 4

От необязательных типов к обнуляемым

Тони Хоар, может, и считает изобретение null-ссылок ошибкой на миллиард долларов[1]*, но нам все равно нужно как-то обозначать отсутствие чего-либо в наших программных системах. Как использовать* `null` *в Kotlin, создавая при этом безопасное ПО?*

Как представить отсутствие чего-либо?

Наверное, самой привлекательной возможностью Kotlin для программистов на Java является представление обнуляемости в системе типов. Это еще одна область, в которой Java и Kotlin имеют различия в своей сущности.

До выхода Java 8 для разграничения ссылок, которые могли или не могли быть равны `null`, использовались соглашения, документация и интуиция. Если метод возвращает элемент коллекции, то можно сделать вывод о том, что он должен иметь возможность вернуть `null`, но может ли вернуть `null` метод `addressLine3`? Или же в случае отсутствия информации следует вернуть пустую строку?

За годы работы авторы этой книги и их коллеги пришли к соглашению, в котором ссылки в Java считаются необнуляемыми, если не указано противоположное. Поэтому у нас могут быть поля и методы с именами `addressLine3OrNull` или `previousAddressOrNull`. В пределах проекта это работает достаточно хорошо (несмотря на длинные имена и необходимость постоянно следить за тем, чтобы не напороться на `NullPointerException`).

В некоторых проектах разработчики предпочитают вместо этого использовать аннотации `@Nullable` и `@NotNullable` — зачастую в сочетании с инструментами для проверки корректности. С выходом в 2014 году Java 8 появилась улучшенная поддержка аннотаций, что позволило таким инструментам, как Checker Framework (**https://oreil.ly/qGYlH**), производить статические проверки далеко не только null-безопасности. Но, что еще важнее, в Java 8 появился стандартный тип `Optional`.

В то время многие JVM-разработчики экспериментировали со Scala. Им нравилось, что для значений, которые могли отсутствовать, допускалось задействовать *необязательный* тип (названный `Optional` в стандартной библиотеке Scala), а для всех остальных — обычные ссылки. Компания Oracle спровоцировала неразбериху,

---

[1] «Null References: The Billion Dollar Mistake» на YouTube (**https://oreil.ly/Ue3Ct**).

посоветовав разработчикам отказаться от использования `Optional` для полей и параметров, но, как и многие другие возможности, появившиеся в Java 8, этот тип оказался достаточно хорошим решением, и его начали применять повсеместно.

В вашем Java-коде (в зависимости от того, когда вы его начали писать) для выражения отсутствия могут использоваться все эти стратегии. Безусловно, можно создать проект, в котором практически никогда не встречаются исключения `NullPointerException`, но в реальности это потребовало бы много усилий. Значения `null` тормозят развитие Java, а половинчатое решение этой проблемы в виде типа `Optional` вряд ли можно назвать блестящим.

Для сравнения: в Kotlin `null` является *полноценным* элементом языка. Благодаря тому, что понятие отсутствия реализовано в системе типов, а не в стандартной библиотеке, проекты на Kotlin демонстрируют невиданную однородность в том, как они обращаются с отсутствующими значениями. Но не все так идеально: `Map<K,V>.get(key)` возвращает `null`, если по указанному ключу ничего нет, а вот `List<T>.get(index)` генерирует `IndexOutOfBoundsException`, если не существует значения с заданным индексом. Аналогичным образом `Iterable<T>.first()` генерирует `NoSuchElementException`, вместо того чтобы вернуть `null`. Эти недочеты обычно вызваны желанием обеспечить обратную совместимость с Java.

API-интерфейсы, написанные специально для Kotlin, в целом являются хорошим примером того, как безопасно использовать `null` для представления необязательных свойств, параметров и возвращаемых значений, и при их исследовании можно многое для себя почерпнуть. Тот, кто поработал с полноценной обнуляемостью, чувствует себя небезопасно в языках, в которых она не поддерживается, — вас не покидает мысль о том, что для получения `NullPointerException` достаточно лишь одной операции разыменования и что для поиска безопасного пути по минному полю приходится полагаться на соглашение.

Функциональные программисты могут посоветовать вам отказаться от обнуляемости Kotlin в пользу необязательного типа (известного как `Maybe`). Мы не рекомендуем этого делать, хотя при этом у вас была бы возможность пользоваться теми же (*монадическими* — вот, мы это произнесли) инструментами для представления потенциально отсутствующих значений, ошибок, асинхронности и т. д. Одна из причин, по которым не стоит применять `Optional` в Kotlin, состоит в том, что вы потеряете доступ к возможностям языка, предназначенным специально для поддержки обнуляемости. В этом отношении Kotlin по своей направленности, например, отличается от Scala.

Еще одна важная причина, почему не стоит представлять необязательность с помощью типа-обертки, является не такой очевидной. В системе типов Kotlin `T` выступает подтипом `T?`. Если у вас есть значение `String`, которое не может быть равным `null`, то вы можете использовать его везде, где требуется обнуляемая версия `String`. Для сравнения: `T` не является подтипом `Optional<T>`. Если у вас есть строка и вы хотите присвоить её необязательной переменной, то вам нужно сначала завернуть ее в `Optional`. Что еще хуже, если у вас есть функция, возвращающая `Optional<String>`, и в какой-то момент вы обнаружите, что она всегда может возвра-

щать результат, изменение возвращаемого типа на `String` нарушит работу любого клиентского кода. Если бы она изначально возвращала значения обнуляемого типа `String?`, вы могли бы заменить этот тип более строгим — `String`, сохранив при этом совместимость. То же самое относится к свойствам структур данных — обнуляемость, в отличие от `Optional`, позволяет легко перейти от необязательного типа к обязательному (забавно, учитывая, что само слово «Optional» подразумевает более широкий выбор).

Авторы этой книги обожают поддержку обнуляемости в Kotlin. Они научились полагаться на эту поддержку при решении многих задач. На то, чтобы отучиться избегать значения `null`, потребуется какое-то время, но наградой за это будет возможность исследовать и использовать целое новое измерение выразительности.

Отсутствие этого механизма в `Travelator` выглядит как досадное упущение, поэтому давайте посмотрим, как перейти от кода на Java, в котором используется `Optional`, к Kotlin с обнуляемыми типами.

Рефакторинг от необязательных типов к обнуляемым

В Travelator путешествия делятся на непрерывные этапы типа `Leg`. В примере 4.1 показана одна из служебных функций, которые встретились нам в коде.

Пример 4.1 [nullability.0:src/main/java/travelator/Legs.java]

```
public class Legs {

    public static Optional<Leg> findLongestLegOver(
        List<Leg> legs,
        Duration duration
    ) {
        Leg result = null;
        for (Leg leg : legs) {
            if (isLongerThan(leg, duration))
                if (result == null ||
                    isLongerThan(leg, result.getPlannedDuration())
                ) {
                    result = leg;
                }
        }
        return Optional.ofNullable(result);
    }

    private static boolean isLongerThan(Leg leg, Duration duration) {
        return leg.getPlannedDuration().compareTo(duration) > 0;
    }
}
```

Тесты (пример 4.2) проверяют, работает ли код так, как задумано, и позволяют нам сразу увидеть его поведение.

Пример 4.2 [nullability.0:src/test/java/travelator/LongestLegOverTests.java]

```
public class LongestLegOverTests {

    private final List<Leg> legs = List.of(
        leg("one hour", Duration.ofHours(1)),
        leg("one day", Duration.ofDays(1)),
        leg("two hours", Duration.ofHours(2))
    );
    private final Duration oneDay = Duration.ofDays(1);

    @Test
    public void is_absent_when_no_legs() {
        assertEquals(
            Optional.empty(),
            findLongestLegOver(emptyList(), Duration.ZERO)
        );
    }

    @Test
    public void is_absent_when_no_legs_long_enough() {
        assertEquals(
            Optional.empty(),
            findLongestLegOver(legs, oneDay)
        );
    }

    @Test
    public void is_longest_leg_when_one_match() {
        assertEquals(
            "one day",
            findLongestLegOver(legs, oneDay.minusMillis(1))
                .orElseThrow().getDescription()
        );
    }

    @Test
    public void is_longest_leg_when_more_than_one_match() {
        assertEquals(
            "one day",
            findLongestLegOver(legs, Duration.ofMinutes(59))
                .orElseThrow().getDescription()
        );
    }

    ...
}
```

Давайте посмотрим, как этот код можно улучшить за счет переписывания его на Kotlin. Рефакторинг `Legs.java` дает — после некоторого переформатирования — следующий результат (пример 4.3).

Пример 4.3 [nullability.3:src/main/java/travelator/Legs.kt]

```
object Legs {
    @JvmStatic
    fun findLongestLegOver(
        legs: List<Leg>,
        duration: Duration
    ): Optional<Leg> {
        var result: Leg? = null
        for (leg in legs) {
            if (isLongerThan(leg, duration))
                if (result == null ||
                    isLongerThan(leg, result.plannedDuration))
                    result = leg
        }
        return Optional.ofNullable(result)
    }

    private fun isLongerThan(leg: Leg, duration: Duration): Boolean {
        return leg.plannedDuration.compareTo(duration) > 0
    }
}
```

Метод имеет вполне ожидаемые параметры, т. к. в Kotlin `List<Leg>` прозрачно принимает `java.util.List` (больше о коллекциях в Java и Kotlin можно узнать в *главе 6*). Здесь стоит сказать, что когда функция в Kotlin объявляет необнуляемый параметр (в нашем случае `legs` и `duration`), компилятор вставляет перед ее телом проверку на `null`. Таким образом, если вызывающий код на Java незаметно передаст `null`, мы сразу же об этом узнаем. Благодаря этим защитным проверкам Kotlin выявляет неожиданные значения `null` как можно ближе к их источнику, в то время как в Java ссылка может быть обнулена далеко от того места, где в итоге генерируется исключение.

Вернемся к нашему примеру: цикл `for` в Kotlin очень похож на аналогичный цикл в Java, только вместо : он использует ключевое слово `in`, которое точно так же применятся к любому типу, который наследует `Iterable`.

Итерация и цикл `for`

На самом деле, помимо `Iterable`, Kotlin позволяет использовать в циклах `for` и другие типы, которые:

- наследуют `Iterator`;
- содержат метод `iterator()`, который возвращает `Iterator`;

- **имеют в своей области видимости функцию-расширение** `operator fun T.iterator()`, **возвращающую** `Iterator`.

К сожалению, сам факт наличия корректного цикла `for` **вовсе не означает, что чужие типы реализуют** `Iterable`. **А жаль, ведь если бы интерфейс** `Iterable` **можно было реализовать задним числом, то мы могли бы применять такие операции, как** `map`, `reduce` **и прочие, к любому типу, т. к. они определены в качестве функций-расширений для** `Iterable<T>`.

Преобразованный код `findLongestLegOver` получился не совсем в духе Kotlin (возможно, с момента появления потоков для Java он тоже не очень характерен). Вместо цикла `for` следует подыскать что-то более выразительное, но давайте отложим это на потом, ведь наша основная задача — перейти от `Optional` к обнуляемым типам. Чтобы это проиллюстрировать, мы по очереди перепишем каждый из наших тестов — это позволит нам получить смешанный проект, словно во время миграции. Для использования обнуляемости наш клиентский код должен быть написан на Kotlin, поэтому давайте преобразуем эти тесты (пример 4.4).

Пример 4.4 [nullability.4:src/test/java/travelator/LongestLegOverTests.kt]

```
class LongestLegOverTests {
    ...
    @Test
    fun is_absent_when_no_legs() {
        Assertions.assertEquals(
            Optional.empty<Any>(),
            findLongestLegOver(emptyList(), Duration.ZERO)
        )
    }

    @Test
    fun is_absent_when_no_legs_long_enough() {
        Assertions.assertEquals(
            Optional.empty<Any>(),
            findLongestLegOver(legs, oneDay)
        )
    }

    @Test
    fun is_longest_leg_when_one_match() {
        Assertions.assertEquals(
            "one day",
            findLongestLegOver(legs, oneDay.minusMillis(1))
                .orElseThrow().description
        )
    }
```

```
    @Test
    fun is_longest_leg_when_more_than_one_match() {
        Assertions.assertEquals(
            "one day",
            findLongestLegOver(legs, Duration.ofMinutes(59))
                .orElseThrow().description
        )
    }

    ...
}
```

Теперь, чтобы осуществить постепенную миграцию, нам нужны две версии `findLongestLegOver`: уже имеющаяся, которая возвращает `Optional<Leg>`, и новая, возвращающая `Leg?`. Чтобы их получить, мы заглянем во внутренности текущей реализации и извлечем из них следующее (пример 4.5).

Пример 4.5 [nullability.4:src/main/java/travelator/Legs.kt]

```
@JvmStatic
fun findLongestLegOver(
    legs: List<Leg>,
    duration: Duration
): Optional<Leg> {
    var result: Leg? = null
    for (leg in legs) {
        if (isLongerThan(leg, duration))
            if (result == null ||
                isLongerThan(leg, result.plannedDuration))
                result = leg
    }
    return Optional.ofNullable(result)
}
```

Выполним действие **Extract Function** (Извлечь функцию) для всех инструкций в `findLongestLegOver`, кроме `return`. Имя функции нельзя оставить без изменения, поэтому назовем ее `longestLegOver` и сделаем ее публичной, поскольку это наш новый интерфейс (пример 4.6).

Пример 4.6 [nullability.5:src/main/java/travelator/Legs.kt]

```
@JvmStatic
fun findLongestLegOver(
    legs: List<Leg>,
    duration: Duration
): Optional<Leg> {
    var result: Leg? = longestLegOver(legs, duration)
```

```
    return Optional.ofNullable(result)
}

fun longestLegOver(legs: List<Leg>, duration: Duration): Leg? {
    var result: Leg? = null
    for (leg in legs) {
        if (isLongerThan(leg, duration))
            if (result == null ||
                isLongerThan(leg, result.plannedDuration))
                result = leg
    }
    return result
}
```

После рефакторинга в `findLongestLegOver` осталась лишняя переменная `result`. Функцию можно выделить и заменить встроенным выражением, чтобы код принял следующий вид (пример 4.7).

Пример 4.7 [nullability.6:src/main/java/travelator/Legs.kt]

```
@JvmStatic
fun findLongestLegOver(
    legs: List<Leg>,
    duration: Duration
): Optional<Leg> {
    return Optional.ofNullable(longestLegOver(legs, duration))
}
```

Теперь у нашего интерфейса есть две версии, причем одна основана на другой. Наш клиентский код на Java может и дальше потреблять `Optional` из `findLongestLegOver`, а вот клиентский код на Kotlin должен вызывать функцию `longestLegOver`, возвращающую обнуляемое значение. Давайте продемонстрируем это преобразование с помощью наших тестов.

Начнем с тех, которые выполняют проверку на отсутствие. В настоящий момент они вызывают `assertEquals(Optional.empty<Any>(), findLongestLegOver...)` (пример 4.8).

Пример 4.8 [nullability.6:src/test/java/travelator/LongestLegOverTests.kt]

```
@Test
fun is_absent_when_no_legs() {
    assertEquals(
        Optional.empty<Any>(),
        findLongestLegOver(emptyList(), Duration.ZERO)
    )
}
```

```
@Test
fun is_absent_when_no_legs_long_enough() {
    assertEquals(
        Optional.empty<Any>(),
        findLongestLegOver(legs, oneDay)
    )
}
```

Подставим вместо этого вызова `assertNull(longestLegOver(...)` (пример 4.9).

Пример 4.9 [nullability.7:src/test/java/travelator/LongestLegOverTests.kt]

```
@Test
fun `is absent when no legs`() {
    assertNull(longestLegOver(emptyList(), Duration.ZERO))
}

@Test
fun `is absent when no legs long enough`() {
    assertNull(longestLegOver(legs, oneDay))
}
```

Отметим, что теперь в качестве имен тестов используются *`идентификаторы в обратных кавычках`*. Чтобы внести это изменение в среде IntelliJ, нужно навести курсор на *имя_тестовой_функции_с_подчеркиваниями* и нажать комбинацию клавиш <Alt>+<Enter>.

Теперь перейдем к вызовам, которые всегда что-то возвращают (пример 4.10).

Пример 4.10 [nullability.6:src/test/java/travelator/LongestLegOverTests.kt]

```
@Test
fun is_longest_leg_when_one_match() {
    assertEquals(
        "one day",
        findLongestLegOver(legs, oneDay.minusMillis(1))
            .orElseThrow().description
    )
}

@Test
fun is_longest_leg_when_more_than_one_match() {
    assertEquals(
        "one day",
        findLongestLegOver(legs, Duration.ofMinutes(59))
            .orElseThrow().description
    )
}
```

Эквивалентом `Optional.orElseThrow()` (т. е. `get()` до выхода Java 10) в Kotlin является оператор `!!`. И `orElseThrow` в Java, и `!!` в Kotlin возвращают значение или, если такового нет, генерируют исключение. В обоих случаях исключение `NullPointerException` является вполне логичным, просто эти два языка по-разному относятся к понятию отсутствия! Если не полагаться на тип исключения, `findLongestLegOver(...).orElseThrow()` можно заменить выражением `longestLegOver(...)!!` (пример 4.11).

Пример 4.11 [nullability.8:src/test/java/travelator/LongestLegOverTests.kt]

```
@Test
fun `is longest leg when one match`() {
    assertEquals(
        "one day",
        longestLegOver(legs, oneDay.minusMillis(1))
            !!.description
    )
}

@Test
fun `is longest leg when more than one match`() {
    assertEquals(
        "one day",
        longestLegOver(legs, Duration.ofMinutes(59))
            ?.description
    )
}
```

Мы преобразовали первый из «ненулевых» тестов (`is longest leg when one match`) с помощью оператора `!!`. В случае провала (которого не происходит — просто мы предпочитаем быть готовыми к таким исходам) он сгенерирует `NullPointerException` вместо аккуратного диагностического сообщения. Во втором тесте мы решили эту проблему с помощью оператора безопасного вызова `?.`, который продолжает проверку, только если его получатель не равен `null`. В результате, если отрезок путешествия оказывается равным `null`, ошибка будет иметь следующий вид, что намного приятнее:

```
Expected :one day
Actual   :null
```

Тесты — одно из тех редких мест, где мы применяем `!!` на практике, но даже здесь обычно имеется более удачная альтернатива.

Мы можем распространить этот рефакторинг на наш клиентский код, переписав его на Kotlin и затем добавив `longestLegOver`. После преобразования всего клиентского кода можно удалить вызов `findLongestLegOver`, возвращающий `Optional`.

Рефакторинг с помощью Expand-and-Contract

Мы будем использовать этот метод, также известный как *параллельное программирование* (**https://oreil.ly/jxSPE**) для управления изменениями, которые вносятся в интерфейсы (со строчной «и») на страницах этой книги. Суть его проста: добавить новый интерфейс, сделать так, чтобы он использовался вместо старого, и затем, когда обращений к старому интерфейсу больше не будет, удалить его.

В этой книге рефакторинг нередко сочетается с преобразованием в Kotlin. Мы обычно преобразуем определение и реализацию (реализации) интерфейса в Kotlin (как в этой главе), после чего добавляем в нее новый интерфейс. В процессе перевода клиентского кода на новый интерфейс мы заодно переписываем его на Kotlin.

Несмотря на то что в ходе этого процесса происходит миграция между интерфейсами и переход от одного языка к другому, мы стараемся разделять эти этапы. Подобно скалолазам, которые сохраняют три точки контакта со скалой, не отпускайте сразу обе руки! Сделайте один шаг, убедитесь в успешном прохождении тестов, затем двигайтесь дальше. Если изменение кажется рискованным, это может быть хорошим поводом перестраховаться (выполнить набор тестов перед фиксацией кода, загрузить предыдущие изменения в репозиторий или даже развернуть «канареечный» релиз), чтобы не обжечься, если что-то пойдет не так.

И не забудьте довести начатое до конца. В ходе рефакторинга мы делаем наш код лучше, что почти всегда означает «проще», а «проще» редко сочетается с «больше». Прежде чем улучшить наш код так, чтобы везде использовался новый интерфейс, мы его сознательно ухудшаем (если одна и та же задача может быть выполнена двумя способами). При этом важно не попасть в ситуацию, когда приходится поддерживать обе версии. Если такая ситуация продлится в течение длительного периода, между версиями могут возникнуть расхождения, и чтобы этого не допустить, и ту и другую придется тестировать. Кроме того, у старой версии могут появиться новые потребители. Конечно, код можно пометить как устаревший, но лучше просто завершить начатое. Тем не менее небольшие прослойки для поддержки старого кода вполне допустимы — мы любим Kotlin, но хотим заниматься чем-то полезным вместо переписывания Java-кода, который не требует к себе внимания.

Рефакторинг в код, характерный для Kotlin

Теперь весь код в этом примере написан на Kotlin, и мы уже знаем, как перейти от необязательных типов к обнуляемым. На этом можно было бы остановиться, но мы всегда стараемся сделать чуть больше, чем необходимо, поэтому поднажмем еще немного и посмотрим, чему еще может научить нас этот код.

Вот текущая версия `Legs` (пример 4.12).

Пример 4.12 [nullability.9:src/main/java/travelator/Legs.kt]

```
object Legs {
    fun longestLegOver(
        legs: List<Leg>,
        duration: Duration
    ): Leg? {
        var result: Leg? = null
        for (leg in legs) {
            if (isLongerThan(leg, duration))
                if (result == null ||
                    isLongerThan(leg, result.plannedDuration))
```

```
                result = leg
        }
        return result
    }

    private fun isLongerThan(leg: Leg, duration: Duration): Boolean {
        return leg.plannedDuration.compareTo(duration) > 0
    }
}
```

Функции содержатся в объекте — поскольку в Java наши методы были статическими, в ходе преобразования их нужно было где-то разместить. Как вы увидите в *главе 8*, в Kotlin эта дополнительная изоляция не нужна, поэтому мы можем выполнить для `longestLegOver` действие **Move to top level** (Переместить на верхний уровень). На момент подготовки книги этот способ работал не очень хорошо, потому что среде IntelliJ не удается извлечь функцию `isLongerThan` вместе с кодом, который ее вызывает, и в результате она остается в `Legs`. Но это легко исправить. У нас получилась функция верхнего уровня и откорректированные ссылки в существующем коде (пример 4.13).

Пример 4.13 [nullability.10:src/main/java/travelator/Legs.kt]

```
fun longestLegOver(
    legs: List<Leg>,
    duration: Duration
): Leg? {
    var result: Leg? = null
    for (leg in legs) {
        if (isLongerThan(leg, duration))
            if (result == null ||
                isLongerThan(leg, result.plannedDuration))
                result = leg
    }
    return result
}

private fun isLongerThan(leg: Leg, duration: Duration) =
    leg.plannedDuration.compareTo(duration) > 0
```

Вы, наверное, заметили, что у `isLongerThan` больше нет фигурных скобок и инструкции `return`. О преимуществах и недостатках функций, состоящих из одного выражения, речь пойдет в *главе 9*.

Прежде чем двигаться дальше, стоит сказать, что выражение `isLongerThan(leg, ...)` выглядит немного странно. Оно и по-английски звучит неправильно. Вам рано или поздно надоест увлечение функциями-расширениями (уж точно к концу *главы 10*),

но пока этого не произошло, наведите текстовый курсор на параметр `leg`, нажмите комбинацию клавиш <Alt>+<Enter> и выберите **Convert parameter to receiver** (Преобразовать параметр в получатель), чтобы мы могли написать `leg.isLongerThan(...)` (пример 4.14).

Пример 4.14 [nullability.11:src/main/java/travelator/Legs.kt]

```
fun longestLegOver(
    legs: List<Leg>,
    duration: Duration
): Leg? {
    var result: Leg? = null
    for (leg in legs) {
        if (leg.isLongerThan(duration))
            if (result == null ||
                leg.isLongerThan(result.plannedDuration))
                result = leg
    }
    return result
}

private fun Leg.isLongerThan(duration: Duration) =
    plannedDuration.compareTo(duration) > 0
```

До сих пор все наши изменения были структурными и касались того, где находится определение кода и как мы к нему обращаемся. Структурный рефакторинг по своей природе довольно-таки (т. е. в основном, но не полностью) безопасный. Он может изменить поведение кода, зависящее от полиморфизма (посредством методов или функций) или рефлексии, но в остальном, если код продолжает успешно компилироваться, он, вероятно, ведет себя как нужно.

Теперь обратим наше внимание на *алгоритм* в `longestLegOver`. Рефакторинг алгоритмов более рискованный, особенно если они изменяют значения, т. к. инструменты плохо поддерживают их преобразование. Чтение кода не очень-то помогает понять, что делает эта функция, но у нас есть хорошие тесты, поэтому давайте посмотрим, что мы можем сделать.

IntelliJ предлагает нам лишь один вариант: поменять `compareTo` на `>`. С этого и начнем. Но тут у Дункана иссякло стремление к рефакторингу (возможно, вы могли бы что-то подсказать, если бы мы действительно работали в паре?), поэтому он решил переписать эту функцию с нуля.

Чтобы заново реализовать такую функциональность, мы спрашиваем себя: «что пытается сделать этот код?». Ответ любезно предоставлен в имени самой функции: `longestLegOver` (самый длинный отрезок из чего-то). Чтобы это вычислить, можно найти самый длинный отрезок и вернуть его, если он длиннее значения `duration`, или `null`, если нет. Если ввести `legs.` в начале функции, откроется меню автодополнения, где можно выбрать `maxByOrNull`. В нашем случае самым длинным отрезком будет `legs.maxByOrNull(Leg::plannedDuration)`. Этот API-интерфейс любезно возвра-

щает `Leg?` (и добавляет фразу `orNull`), напоминая нам, что мы не можем получить результат, если список `legs` пустой. Если напрямую выразить наш алгоритм «найти самый длинный отрезок и вернуть его, если он больше `duration`, в противном случае вернуть `null`» непосредственно в коде, получится следующее (пример 4.15).

Пример 4.15 [nullability.12:src/main/java/travelator/Legs.kt]

```
fun longestLegOver(
    legs: List<Leg>,
    duration: Duration
): Leg? {
    val longestLeg: Leg? = legs.maxByOrNull(Leg::plannedDuration)
    if (longestLeg != null && longestLeg.plannedDuration > duration)
        return longestLeg
    else
        return null
}
```

Этот код успешно проходит тесты, но наличие в нем сразу нескольких инструкций `return` выглядит топорно. IntelliJ любезно предлагает убрать `return` из `if` (пример 4.16).

Пример 4.16 [nullability.13:src/main/java/travelator/Legs.kt]

```
fun longestLegOver(
    legs: List<Leg>,
    duration: Duration
): Leg? {
    val longestLeg: Leg? = legs.maxByOrNull(Leg::plannedDuration)
    return if (longestLeg != null && longestLeg.plannedDuration > duration)
        longestLeg
    else
        null
}
```

Теперь поддержка обнуляемости в Kotlin позволяет модифицировать этот код несколькими способами, в зависимости от ваших предпочтений.

Мы можем воспользоваться так называемым *оператором Элвиса* `?:`, который вычисляет правый операнд только в случае, если левый равен `null`. Это позволяет нам покинуть функцию сразу, как только обнаружится, что у нас нет самого длинного отрезка (пример 4.17).

Пример 4.17 [nullability.14:src/main/java/travelator/Legs.kt]

```
fun longestLegOver(
    legs: List<Leg>,
```

```
    duration: Duration
): Leg? {
    val longestLeg = legs.maxByOrNull(Leg::plannedDuration) ?:
        return null
    return if (longestLeg.plannedDuration > duration)
        longestLeg
    else
        null
}
```

Вместо этого можно использовать единое выражение `?.let`. Если оператору `?.` передать `null`, он вернет `null`, в противном случае он автоматически присоединяет самый длинный отрезок к блоку `let` (пример 4.18).

Пример 4.18 [nullability.15:src/main/java/travelator/Legs.kt]

```
fun longestLegOver(
    legs: List<Leg>,
    duration: Duration
): Leg? =
    legs.maxByOrNull(Leg::plannedDuration)?.let { longestLeg ->
    if (longestLeg.plannedDuration > duration)
        longestLeg
    else
        null
}
```

То есть значение `longestLeg` не может быть равно `null` внутри `let`. Это компактное и приятное на вид единое выражение, но его может быть сложно понять с первого взгляда. Перебор вариантов с помощью `when` делает код яснее (пример 4.19).

Пример 4.19 [nullability.17:src/main/java/travelator/Legs.kt]

```
fun longestLegOver(
    legs: List<Leg>,
    duration: Duration
): Leg? {
    val longestLeg = legs.maxByOrNull(Leg::plannedDuration)
    return when {
        longestLeg == null -> null
        longestLeg.plannedDuration > duration -> longestLeg
        else -> null
    }
}
```

Для дальнейшего упрощения нам понадобится прием, который Дункан (автор этих строк) еще не сумел усвоить: `takeIf` возвращает своего получателя, если предикат

равен `true`, в противном случае возвращается `null`. Именно эта логика применялась в нашем предыдущем блоке `let`. Поэтому мы можем написать так, как показано в примере 4.20.

Пример 4.20 [nullability.16:src/main/java/travelator/Legs.kt]

```
fun longestLegOver(
    legs: List<Leg>,
    duration: Duration
): Leg? =
    legs.maxByOrNull(Leg::plannedDuration)?.takeIf { longestLeg ->
        longestLeg.plannedDuration > duration
    }
```

Для команды, которая имеет небольшой опыт работы с Kotlin, такой прием может показаться слишком утонченным. И хотя Нэт с этим не согласен, мы отдадим предпочтение выразительности и остановимся на версии с `when` — по крайней мере пока кто-нибудь другой не возьмется за рефакторинг этого участка.

Напоследок возьмем параметр `legs` в функции-расширении и превратим его в получателя. Это позволит нам дать этой функции более ясное имя (пример 4.21).

Пример 4.21 [nullability.18:src/main/java/travelator/Legs.kt]

```
fun List<Leg>.longestOver(duration: Duration): Leg? {
    val longestLeg = maxByOrNull(Leg::plannedDuration)
    return when {
        longestLeg == null -> null
        longestLeg.plannedDuration > duration -> longestLeg
        else -> null
    }
}
```

Прежде чем завершать эту главу, воспользуемся моментом и сравним эту версию с оригиналом. Имеет ли старая версия (пример 4.22) какие-либо преимущества?

Пример 4.22 [nullability.0:src/main/java/travelator/Legs.java]

```
public class Legs {

    public static Optional<Leg> findLongestLegOver(
        List<Leg> legs,
        Duration duration
    ) {
        Leg result = null;
        for (Leg leg : legs) {
            if (isLongerThan(leg, duration))
```

```
                if (result == null ||
                    isLongerThan(leg, result.getPlannedDuration())
                ) {
                    result = leg;
                }
            }
            return Optional.ofNullable(result);
        }

        private static boolean isLongerThan(Leg leg, Duration duration) {
            return leg.getPlannedDuration().compareTo(duration) > 0;
        }
}
```

Обычно мы не даем однозначного ответа на такие вопросы, но в данном случае нам кажется, что новая версия лучше практически во всем. Она короче и проще, нам легче понять, как она работает, и в большинстве ситуаций она выполняет меньше вызовов `getPlannedDuration()`, которые являются относительно ресурсоемкими. Что, если применить тот же подход в Java? Вот как будет выглядеть дословный перевод (пример 4.23).

Пример 4.23 [nullability.1:src/main/java/travelator/Legs.java]

```
public class Legs {

    public static Optional<Leg> findLongestLegOver(
        List<Leg> legs,
        Duration duration
    ) {
        var longestLeg = legs.stream()
            .max(Comparator.comparing(Leg::getPlannedDuration));
        if (longestLeg.isEmpty()) {
            return Optional.empty();
        } else if (isLongerThan(longestLeg.get(), duration)) {
            return longestLeg;
        } else {
            return Optional.empty();
        }
    }

    private static boolean isLongerThan(Leg leg, Duration duration) {
        return leg.getPlannedDuration().compareTo(duration) > 0;
    }
}
```

Не так уж и плохо, но если сравнить с версией на Kotlin, то можно заметить, что `Optional` добавляет лишний код практически в каждую строчку метода. Из-за этого

версия с `Optional.filter`, наверное, будет предпочтительнее, даже несмотря на те же проблемы с удобочитаемостью, что и у `takeIf` в Kotlin. Иными словами, Нэту она нравится, а Дункану нужно выполнить тесты, чтобы понять, работает она или нет (пример 4.24).

Пример 4.24 [nullability.2:src/main/java/travelator/Legs.java]

```
public static Optional<Leg> findLongestLegOver(
    List<Leg> legs,
    Duration duration
) {
    return legs.stream()
        .max(Comparator.comparing(Leg::getPlannedDuration))
        .filter(leg -> isLongerThan(leg, duration));
}
```

Двигаемся дальше

Понятие отсутствия или наличия информации является неотъемлемой частью нашего кода. В Kotlin ему отводится роль полноценного элемента языка, поэтому отсутствие значения принимается во внимание только тогда, когда нужно, а в остальных случаях оно нам не мешает. По сравнению с ним тип `Optional` в Java кажется топорным. К счастью, мы можем легко перейти с `Optional` на обнуляемый тип и поддерживать их одновременно, если мы не готовы преобразовать в Kotlin весь наш код.

В *главе 10 «От функций к функциям-расширениям»* вы увидите, как обнуляемые типы в сочетании с другими возможностями языка Kotlin (безопасными вызовами, операторами Элвиса и функциями-расширениями) поощряют архитектурные решения, заметно отличающиеся от тех, которые мы применяем в Java.

Но не будем забегать вперед. В следующей главе мы возьмем типичный класс на Java и превратим его в типичный класс на Kotlin. Это преобразование будет не только синтаксическим, т. к. изменяемое состояние в этих языках имеет разную степень допустимости.

ГЛАВА 5

От объектов JavaBeans к значениям

Во многих проектах на Java для представления данных используются соглашения, принятые в изменяемых объектах JavaBeans или POJO (plain old Java object, старых добрых Java-объектах). Однако изменяемость приводит к усложнению. Почему неизменяемые значения являются лучшим выбором и как снизить расходы проекта, связанные с изменяемыми данными?

JavaBeans

Как уже обсуждалось в *разд. «Стиль JavaBeans» главы 1*, технология JavaBeans создавалась для того, чтобы сделать возможной разработку визуальных конструкторов *пользовательских интерфейсов* (*англ.* user interfaces, UI) в стиле Visual Basic. Программист мог перетянуть кнопку на форму, изменить ее текст и значок, затем подключить обработчик щелчков мышью, а конструктор автоматически генерировал код для создания экземпляра этой кнопки и вызывал сеттеры для свойств, измененных разработчиком.

У определения класса JavaBeans должен быть конструктор по умолчанию (без аргументов), геттеры для свойств и сеттеры для тех из них, которые можно изменять (мы не станем говорить здесь о реализации интерфейса `Serializable`, потому что даже компания Sun не относилась к этому требованию серьезно). Для объектов с множеством свойств это выглядит логично. У компонентов пользовательского интерфейса обычно есть цвет текста и фона, шрифт, надпись, границы, выравнивание, внутренние отступы и т. д. Эти свойства в основном имеют адекватные значения по умолчанию, поэтому вызов сеттеров лишь для некоторых из них минимизирует количество генерируемого кода. Даже сегодня модель изменяемых компонентов является хорошим выбором для средств создания пользовательских интерфейсов.

Но на заре появления технологии JavaBeans мы считали изменяемыми большинство объектов, а не только компоненты пользовательского интерфейса. А почему бы и нет? Объекты были нужны для инкапсуляции свойств и управления отношениями между ними. Они предназначались для решения таких задач, как обновление ширины компонента при изменении его границ или общей суммы заказа при добавлении товара в корзину покупок. Объекты служили средством управления изменяемым состоянием. На тот момент сам факт наличия в Java неизменяемого класса `String` считался весьма радикальным (хотя авторы языка не удержались и сделали класс `Date` изменяемым).

За прошедшие годы отношение разработчиков к этому вопросу заметно эволюционировало. Мы ценим возможность использования объектов для представления разных типов значений, сущностей, сервисов, действий, транзакций и т. п. Однако стандартный метод проектирования Java-объектов по-прежнему основан на JavaBeans: объекты являются изменяемыми, а их свойства имеют геттеры и сеттеры. Такое решение, может, и подходит для средств создания пользовательских интерфейсов, но в качестве подхода по умолчанию оно не годится. Для большинства сущностей, которые мы хотим представить в виде объектов, лучше использовать значение.

Значения

Смысл термина *значение* в русском языке может сильно варьироваться. В компьютерных технологиях мы говорим, что переменные, параметры и поля имеют значения: примитивы или ссылки, к которым они привязаны. В этой книге под *значением* понимается *определенный* тип примитивов или ссылок, обладающий семантикой значения. Эта семантика подразумевает, что значение объекта влияет на его поведение, но не на то, что он собой представляет. В Java все примитивы имеют семантику значения: каждое число 7 равно любому другому числу 7. Но в случае с объектами это не всегда так — в частности, такая семантика не распространяется на изменяемые объекты. В следующих главах будут рассмотрены более тонкие различия, а пока что будем считать *значение* неизменяемым фрагментом данных, а *тип значений* — типом, который определяет его поведение.

Таким образом 7 — это значение, а упакованная версия `Integer` — его тип (поскольку упакованные типы являются неизменяемыми); `banana` — значение (поскольку экземпляры `String` являются неизменяемыми) и `URI` — значение (поскольку экземпляры `URI` являются неизменяемыми), а вот `java.util.Date` не является типом значений (поскольку у него есть `setYear` и другие вызовы).

Экземпляр неизменяемого класса `DBConnectionInfo` является значением, а экземпляр `Database` — нет, хотя ни одно из его свойств нельзя изменить. Дело в том, что это не фрагмент данных, а механизм для обращения к ним и их изменения.

Можно ли считать значениями экземпляры JavaBeans? Компоненты пользовательского интерфейса, основанные на JavaBeans, не являются значениями, т. к. они представляют собой не просто данные — две идентичные кнопки не тождественны. Если JavaBeans используется для представления обычных данных, все зависит от того, поддается ли объект изменению. Экземпляры JavaBeans могут быть неизменяемыми, но большинство разработчиков посчитало бы их обычными Java-объектами.

Можно ли считать значениями POJO? Этот термин относится к классам, которые приносят пользу, не наследуя типы, предоставляемые фреймворками. Они обычно представляют данные и соблюдают соглашения о методах доступа в JavaBeans. У многих POJO нет конструктора по умолчанию, вместо этого они определяют конструкторы для инициализации свойств, для которых нельзя придумать разумные стандартные значения. В связи с этим POJO нередко являются неизменяемыми

и могут иметь семантику значения. Однако изменяемые POJO по-прежнему доминируют — настолько, что многие люди ассоциируют объектно-ориентированное программирование на Java с изменяемыми объектами. А изменяемые объекты POJO не являются значениями.

Если подытожить, то теоретически экземпляр JavaBeans может быть значением, но это редкость. POJO чаще обладают семантикой значений, особенно в современном коде на Java. Поэтому, несмотря на заголовок *От объектов JavaBeans к значениям*, эта глава на самом деле посвящена преобразованию изменяемых объектов в неизменяемые данные, поэтому ее, возможно, стоило назвать *Значения вместо изменяемых POJO*. Надеемся, вы простите нам эту неточность.

Почему следует выбирать значения?

Значения — это неизменяемые данные. Почему стоит отдавать предпочтение неизменяемым объектам перед изменяемыми и теми, которые представляют данные для других типов объектов? К этому вопросу мы будем неоднократно возвращаться на страницах нашей книги. Но пока просто скажем, что в неизменяемых объектах легче разобраться, т. к. они не меняются. Поэтому:

- их можно поместить внутрь множества или использовать в качестве ключей для ассоциативного списка;
- нам не нужно беспокоиться о том, что в ходе перебора неизменяемой коллекции она будет модифицирована;
- мы можем исследовать разные сценарии без глубокого копирования начальных состояний (что также упрощает реализацию функции отмены и повтора);
- мы можем безопасно разделять неизменяемые объекты между разными потоками выполнения.

Рефакторинг экземпляров JavaBeans в значения

Давайте рассмотрим рефакторинг кода, использующего изменяемый экземпляр JavaBeans или POJO, в значение.

`Travelator` — мобильное приложение, и его версия для Android написана на Java. В этом коде мы описываем пользовательские настройки с помощью объекта `UserPreferences` (пример 5.1).

Пример 5.1
[beans-to-values.0:src/main/java/travelator/mobile/UserPreferences.java]

```
public class UserPreferences {

    private String greeting;
    private Locale locale;
    private Currency currency;
```

```
    public UserPreferences() {
        this("Hello", Locale.UK, Currency.getInstance(Locale.UK));
    }

    public UserPreferences(String greeting, Locale locale, Currency currency) {
        this.greeting = greeting;
        this.locale = locale;
        this.currency = currency;
    }

    public String getGreeting() {
        return greeting;
    }

    public void setGreeting(String greeting) {
        this.greeting = greeting;
    }

    ... геттеры и сеттеры для locale и currency
}
```

У `Application` есть свойство `preferences`. Оно передается тем представлениям, которые в нем нуждаются (пример 5.2).

Пример 5.2 [beans-to-values.0:src/main/java/travelator/mobile/Application.java]

```
public class Application {

    private final UserPreferences preferences;

    public Application(UserPreferences preferences) {
        this.preferences = preferences;
    }

    public void showWelcome() {
        new WelcomeView(preferences).show();
    }

    public void editPreferences() {
        new PreferencesView(preferences).show();
    }
    ...
}
```

(Любое сходство с реальным UI-фреймворком, живым или мертвым, является чистой случайностью.)

Наконец, `PreferencesView` обновляет свое свойство `preferences`, когда пользователь вносит изменения. О внесении изменений сигнализирует вызов `onThingChange()` (пример 5.3).

Пример 5.3 [beans-to-values.0:src/main/java/travelator/mobile/PreferencesView.java]

```
public class PreferencesView extends View {

    private final UserPreferences preferences;
    private final GreetingPicker greetingPicker = new GreetingPicker();
    private final LocalePicker localePicker = new LocalePicker();
    private final CurrencyPicker currencyPicker = new CurrencyPicker();

    public PreferencesView(UserPreferences preferences) {
        this.preferences = preferences;
    }

    public void show() {
        greetingPicker.setGreeting(preferences.getGreeting());
        localePicker.setLocale(preferences.getLocale());
        currencyPicker.setCurrency(preferences.getCurrency());
        super.show();
    }

    protected void onGreetingChange() {
        preferences.setGreeting(greetingPicker.getGreeting());
    }

    protected void onLocaleChange() {
        preferences.setLocale(localePicker.getLocale());
    }

    protected void onCurrencyChange() {
        preferences.setCurrency(currencyPicker.getCurrency());
    }
    ...
}
```

Этот подход, несмотря на свою простоту, чреват осложнениями, которые свойственны изменяемым данным:

- если и `PreferencesView`, и `WelcomeView` являются активными, то значения, доступные для `WelcomeView`, могут устареть;
- методы `equality` и `hashCode` в объекте `UserPreferences` зависят от значений его свойств, которые могут меняться. Поэтому `UserPreferences` нельзя надежным образом использовать внутри множеств или в качестве ключей для ассоциативных массивов;

- ничто не указывает на то, что `WelcomeView` занимается лишь чтением настроек;
- если чтение и запись происходят в разных потоках выполнения, нам нужно обеспечивать синхронизацию на уровне свойств `UserPreferences`.

Прежде чем переходить на использование неизменяемого значения, давайте переведем `Application` и `UserPreferences` на Kotlin. Это поможет нам увидеть суть нашей модели. С `Application` все просто (пример 5.4).

Пример 5.4 [beans-to-values.1:src/main/java/travelator/mobile/Application.kt]

```
class Application(
    private val preferences: UserPreferences
) {
    fun showWelcome() {
        WelcomeView(preferences).show()
    }

    fun editPreferences() {
        PreferencesView(preferences).show()
    }
    ...
}
```

Чего нельзя сказать о `UserPreferences`. Операция **Convert to Kotlin** (Преобразовать в Kotlin) в IntelliJ дает такой результат (пример 5.5).

Пример 5.5 [beans-to-values.1:src/main/java/travelator/mobile/UserPreferences.kt]

```
class UserPreferences @JvmOverloads constructor(
    var greeting: String = "Hello",
    var locale: Locale = Locale.UK,
    var currency: Currency = Currency.getInstance(Locale.UK)
)
```

Это довольно-таки замысловатое преобразование. Аннотация `@JVMOverloads` заставляет компилятор сгенерировать несколько конструкторов, которые позволяют использовать значения по умолчанию для сочетаний `greeting`, `locale` или `currency`. Наш исходный код на Java этого не делал — у него было всего два конструктора (один из которых был конструктором по умолчанию и не принимал никаких аргументов).

Мы пока что не изменяли поведение нашего приложения, а лишь упростили его реализацию. Когда свойства помечены как `var`, а не как `val`, это признак того, что мы имеем дело с изменяемыми данными. В таких случаях следует помнить о том, что компилятор Kotlin сгенерирует для каждого свойства частное поле, а также геттер и сеттер, чтобы код на Java и дальше мог работать с классом данных, словно с экземпляром JavaBeans. Kotlin оставляет без изменений соглашение об именова-

нии, принятое в JavaBeans, а свойства `var` позволяют нам определять изменяемые экземпляры JavaBeans, плохо это или хорошо.

Мы ничего хорошего в этом не видим. Как теперь сделать класс `UserPreferences` неизменяемым? Ведь в нашем приложении должны быть доступны настройки, отражающие любые изменения, которые вносит пользователь. Ответ прост: нужно переместить изменяющие операции. В духе того подхода, который часто применяется при рефакторинге в этой книге, мы перенесем проблемный аспект (в нашем случае операцию изменения) вверх — т. е. ближе к точке входа или на уровень выше, где находится бизнес-логика приложения.

Вместо того чтобы изменять настройки, обновим ссылку на них в `Application`. Эта ссылка будет указывать на обновленную копию, возвращенную объектом `PreferencesView`. Суть нашей стратегии в том, чтобы перейти от неизменяемой ссылки на изменяемый объект к изменяемой ссылке на неизменяемое значение. Зачем? Потому что это снижает как количество, так и видимость потенциальных составных элементов, а именно видимость изменяющей операции создает нам проблемы.

Мы будем двигаться к этому постепенно, начиная с преобразования `PreferencesView` в Kotlin (пример 5.6).

Пример 5.6 [beans-to-values.3:src/main/java/travelator/mobile/PreferencesView.kt]

```
class PreferencesView(
    private val preferences: UserPreferences
) : View() {
    private val greetingPicker = GreetingPicker()
    private val localePicker = LocalePicker()
    private val currencyPicker = CurrencyPicker()

    override fun show() {
        greetingPicker.greeting = preferences.greeting
        localePicker.locale = preferences.locale
        currencyPicker.currency = preferences.currency
        super.show()
    }

    protected fun onGreetingChange() {
        preferences.greeting = greetingPicker.greeting
    }

    ... onLocaleChange, onCurrencyChange
}
```

Метод `show()` переопределяет метод внутри `View`, который делает представление видимым и блокирует поток выполнения, пока оно не будет закрыто. Чтобы данные не изменялись, нам нужна версия этой функции, возвращающая копию `UserPreferences` со всеми внесенными изменениями, но мы не можем добавить тип

возвращаемого значения в метод класса `View`. Поэтому переименуем `show` в `showModal` и сделаем так, чтобы этот метод возвращал изменяемое свойство `preferences` после возвращения из `super.show()` (пример 5.7).

Пример 5.7 [beans-to-values.4:src/main/java/travelator/mobile/PreferencesView.kt]

```
fun showModal(): UserPreferences {
    greetingPicker.greeting = preferences.greeting
    localePicker.locale = preferences.locale
    currencyPicker.currency = preferences.currency
    show()
    return preferences
}
```

Метод `Application.editPreferences()` вызывал `preferencesView.show()`, полагаясь на тот факт, что `Application` и `PreferencesView` используют общую ссылку на изменяемый объект для получения доступа к любым изменениям. Теперь же мы сделаем свойство `Application.preferences` изменяемым и присвоим ему результат работы `showModal` (пример 5.8).

Пример 5.8 [beans-to-values.4:src/main/java/travelator/mobile/Application.kt]

```
class Application(
    private var preferences: UserPreferences ❶
) {
    ...

    fun editPreferences() {
        preferences = PreferencesView(preferences).showModal()
    }
    ...
}
```

❶ Теперь это `var`.

В настоящий момент метод `showModal` возвращает тот же объект, который передается представлению в конструкторе, поэтому в действительности ничего не изменилось. Более того, мы получили худшую комбинацию из возможных — изменяемую ссылку на изменяемые данные.

Но это еще не все. Чтобы усугубить ситуацию, свойство `preferences` в `PreferencesView` тоже можно сделать изменяемым, чтобы при обновлении элементов пользовательского интерфейса ему можно было присвоить объект `UserPreferences` (пример 5.9).

Пример 5.9 [beans-to-values.5:src/main/java/travelator/mobile/PreferencesView.kt]

```
class PreferencesView(
    private var preferences: UserPreferences
```

```
) : View() {
    private val greetingPicker = GreetingPicker()
    private val localePicker = LocalePicker()
    private val currencyPicker = CurrencyPicker()

    fun showModal(): UserPreferences {
        greetingPicker.greeting = preferences.greeting
        localePicker.locale = preferences.locale
        currencyPicker.currency = preferences.currency
        show()
        return preferences
    }

    protected fun onGreetingChange() {
        preferences = UserPreferences(
            greetingPicker.greeting,
            preferences.locale,
            preferences.currency
        )
    }

    ... onLocaleChange, onCurrencyChange
}
```

И хотя мы хотели все «усугубить», в `UserPreferences` теперь не используется ни единого сеттера. В отсутствии сеттеров можно превратить этот класс в настоящее значение — для этого нужно инициализировать его свойства в его же конструкторе и никогда их больше не изменять. В Kotlin можно просто поменять свойства `var` на `val` и заменить встроенным выражением любое использование конструктора по умолчанию. Это позволяет свести `UserPreferences` к следующему (пример 5.10).

Пример 5.10 [beans-to-values.6:src/main/java/travelator/mobile/UserPreferences.kt]

```
data class UserPreferences(
    val greeting: String,
    val locale: Locale,
    val currency: Currency
)
```

Наблюдательные читатели могут обнаружить, что мы незаметно превратили `UserPreferences` в класс данных. Раньше мы этого не делали, поскольку он был изменяемым. И хотя Kotlin *допускает* создание изменяемых классов данных, к ним следует относиться с еще большей опаской, чем к другим изменяемым классам, т. к. они реализуют `equals` и `hashCode`.

Равенство объектов

Методы `equals` и `hashCode` в классах данных основаны на значениях всех свойств, объявленных в первичном конструкторе. Поэтому два экземпляра одного класса являются равными тогда (и только тогда), когда все их соответствующие свойства имеют одни и те же значения.

Ассоциативные массивы и множества вызывают `equals` и `hashCode` при сохранении объектов, но, если впоследствии эти объекты изменяются, может случиться нечто странное (**https://oreil.ly/keALT**).

Не полагайтесь на эти два метода в изменяемых объектах и не создавайте изменяемые классы данных.

Итак, чего мы достигли на этот момент? Мы заменили две неизменяемые ссылки на разделяемые изменяемые данные двумя изменяемыми ссылками на неизменяемые значения. Теперь сразу видно, какие представления могут обновлять настройки, и, если нам нужно производить обновления в разных потоках, мы можем это сделать на уровне приложения.

Однако наличие изменяемой ссылки на `PreferencesView` немного раздражает. Чтобы это исправить, мы можем совсем отказаться от ссылок и вместо этого передавать настройки методу `showModal`. Объекту `PreferencesView` не нужно свойство `UserPreferences`. Пока он не отображается на экране, мы можем просто сформировать пользовательский интерфейс на основе значений этого свойства и затем снова собрать их вместе по завершении работы `PreferencesView` (пример 5.11).

Пример 5.11 [beans-to-values.7:src/main/java/travelator/mobile/PreferencesView.kt]

```
class PreferencesView : View() {
    private val greetingPicker = GreetingPicker()
    private val localePicker = LocalePicker()
    private val currencyPicker = CurrencyPicker()

    fun showModal(preferences: UserPreferences): UserPreferences {
        greetingPicker.greeting = preferences.greeting
        localePicker.locale = preferences.locale
        currencyPicker.currency = preferences.currency
        show()
        return UserPreferences(
            greeting = greetingPicker.greeting,
            locale = localePicker.locale,
            currency = currencyPicker.currency
        )
    }
}
```

Здесь все еще вносятся изменения, т. к. мы присваиваем значения элементам интерфейса, но у этих элементов есть только конструкторы по умолчанию, так что эти действия в любом случае должны где-то выполняться. В завершение нужно также обновить `Application` путем перемещения аргумента `preferences` из конструктора `PreferencesView` в `showModal` (пример 5.12).

Пример 5.12 [beans-to-values.7:src/main/java/travelator/mobile/Application.kt]

```
class Application(
    private var preferences: UserPreferences
) {
    fun showWelcome() {
        WelcomeView(preferences).show()
    }

    fun editPreferences() {
        preferences = PreferencesView().showModal(preferences)
    }
    ...
}
```

Теперь настройки могут изменяться лишь в одном месте, на что явно указывает операция присваивания в `editPreferences`. Также очевидно, что `showWelcome` может только читать из `UserPreferences`. Создание нового экземпляра `UserPreferences` для возвращения из `showModal`, даже если ничего не изменилось, может показаться расточительным или даже опасным, если вы привыкли разделять изменяемые объекты. Но в мире значений две копии `UserPreferences` с одинаковыми свойствами почти во всех отношениях являются одним и тем же объектом (см. ранее *врезку «Равенство объектов»*), и, чтобы заметить выделение дополнительных ресурсов, нужно находиться в очень ограниченном окружении.

Двигаемся дальше

В этой главе были рассмотрены некоторые преимущества неизменяемых значений перед изменяемыми объектами. Пример с рефакторингом показал, как приблизить операции изменения к обработчикам событий и точкам входа в наше приложение за счет отказа от неизменяемых ссылок на изменяемые объекты в пользу изменяемых ссылок на неизменяемые объекты. В результате последствия и трудности, которые несет с собой изменяемость, затрагивают меньшую часть нашего кода.

Тем не менее технология JavaBeans была рассчитана на применение во фреймворках для создания пользовательских интерфейсов, а пользовательские интерфейсы во многом остаются бастионом изменяемых объектов. Если бы наш UI должен был быть более динамичным (например, если бы при изменении параметра `greeting` нужно было обновлять `WelcomeView`), вместо неизменяемых значений, возможно, стоило бы воспользоваться разделяемыми объектами с событиями, срабатывающими при изменениях.

Переход от изменяемых объектов к значениям и преобразованиям будет встречаться и дальше. В *главе 6 «От коллекций Java к коллекциям Kotlin»* мы продолжим наше обсуждение в контексте коллекций. В *главе 14 «От накопления объектов к преобразованиям»* показано, как отказаться от накопительных параметров в пользу функций верхнего уровня для работы с коллекциями.

ГЛАВА 6

От коллекций Java к коллекциям Kotlin

На первый взгляд, Java и Kotlin имеют очень похожие библиотеки коллекций с подозрительно хорошей совместимостью. Чем они различаются, на что они ориентированы и на что следует обращать внимание при переходе от коллекций Java к коллекциям Kotlin?

Коллекции Java

В *главе 5* рассказывалось о развитии Java во времена, когда считалось, что объекты должны быть изменяемыми и хранить свое состояние. Это в особенности касалось коллекций — ведь какой смысл в списках, если в них нельзя ничего добавить? Сначала создается пустая коллекция, а затем в нее добавляются элементы. Хотите удалить элемент из корзины покупок? Измените список. Перетасовать колоду карт? Очевидно, что это меняет порядок размещения элементов. Мы не стали бы открывать наш бумажный блокнот и составлять новый список дел каждый раз, когда нам нужно сходить за молоком или свозить кошку к ветеринару. Изменяемые коллекции отражают наш реальный опыт.

В момент появления языка Java на него стоило перейти из-за одних только встроенных коллекций. В те дни в стандартной библиотеке многих языков отсутствовали коллекции с возможностью изменения размера. Технология объектно-ориентированного программирования позволяла безопасно определять и использовать изменяемые коллекции. Было бы странно не воспользоваться такой чудесной возможностью, раз уж нам ее предоставили. Поэтому мы начали применять объекты `Vector` и `HashTable` так, как задумывали разработчики Sun. Иными словами, мы сначала их создавали, а затем изменяли. У нас не было выбора, поскольку все конструкторы создавали пустые коллекции.

В Java 2 (на самом деле 1.2 — в то время Java еще не нужно было конкурировать с C# в нумерации версий) библиотека коллекций была переработана. Это сделало классы `Vector`, `Stack` и `Hashtable`, созданные на скорую руку, более аккуратными. Также был добавлен общий интерфейс `Collection` и дополнительные полезные реализации, включая `ArrayList` и `HashSet`. Это позволило создавать одну коллекцию в качестве копии другой. Статический класс `Collections` предоставлял удобные служебные операции — такие как `sort` и `reverse`. В Java 5 появились обобщения, которые были искусно интегрированы в существующие коллекции, что позволило объявлять такие типы, как `List<Journey>`.

Коллекции Java остались в высшей степени изменяемыми. Помимо поддержки добавления и удаления элементов, операции типа сортировки могут быть выполнены *только* путем внесения изменений — в стандартной библиотеке нет функции, которая возвращает отсортированную копию `List`.

Мы не устаем повторять, что изменения — источник многих проблем с усложнением кода: они делают возможной рассинхронизацию состояния в разных местах. Например, в `Travelator` маршрут можно представить в виде списка объектов `Journey`. Также существует такое понятие, как *коэффициент сложности* (*англ.* sufferscore) — чем он ниже, тем приятнее, скорее всего, окажется путешествие. Вот как мы вычисляем его для заданного маршрута (пример 6.1).

Пример 6.1 [collections.0:src/main/java/travelator/Suffering.java]

```
public static int sufferScoreFor(List<Journey> route) {
    Location start = getDepartsFrom(route);
    List<Journey> longestJourneys = longestJourneysIn(route, 3);
    return sufferScore(longestJourneys, start);
}
```

Локальная переменная `start` не играет существенной роли, поэтому мы решили заменить ее встроенным выражением (пример 6.2).

Пример 6.2 [collections.1:src/main/java/travelator/Suffering.java]

```
public static int sufferScoreFor(List<Journey> route) {
    List<Journey> longestJourneys = longestJourneysIn(route, 3);
    return sufferScore(longestJourneys, getDepartsFrom(route));
}
```

Тесты прошли успешно, мы развертываем код в рабочих условиях, но получаем отчет об ошибке, свидетельствующий о проблеме. При дальнейшем исследовании находим ее причину (примеры 6.3 и 6.4).

Пример 6.3 [collections.0:src/main/java/travelator/Routes.java]

```
public static Location getDepartsFrom(List<Journey> route) {
    return route.get(0).getDepartsFrom();
}
```

Пример 6.4 [collections.0:src/main/java/travelator/Suffering.java]

```
public static List<Journey> longestJourneysIn(
    List<Journey> journeys,
    int limit
) {
    journeys.sort(comparing(Journey::getDuration).reversed()); ❶
```

```
    var actualLimit = Math.min(journeys.size(), limit);
    return journeys.subList(0, actualLimit);
}
```

❶ Параметр `journeys`, изменяемый методом `sort`.

Ага, теперь стало видно, что при поиске самого длинного путешествия меняется его отправная точка (`Location`). Разработчик вызвал метод из параметра `journeys`, чтобы решить проблему, и это внезапно нарушило работу какого-то другого участка системы! Вам нужно потратить всего лишь несколько сотен часов своей жизни на отладку проблем, вызванных подобными ошибками псевдонимов (**https://oreil.ly/PeqKs**), чтобы прийти к выводу о том, что по умолчанию лучше использовать неизменяемые данные. Программисты, работающие с JDK, по-видимому, осознали это после выхода Java 2, поэтому мы уже никогда не избавимся от интерфейсов изменяемых коллекций.

Справедливости ради нужно сказать, что коллекции Java с годами становятся все менее изменяемыми в практическом смысле. Даже в самом начале их можно было, к примеру, завернуть в `Collections.unmodifiableList`. Результатом был все тот же экземпляр `List` со всеми его изменяющими методами. Различие заключалось в том, что эти методы генерировали `UnsupportedOperationException`. Мы могли бы обнаружить, что `shortestJourneyIn` изменяет наш список, завернув результат выполнения `loadJourneys` в `UnmodifiableList`. Тесты любого кода, в котором сочетаются эти метод и обертка, быстро бы провалились, хотя и на этапе выполнения, а не компиляции. Жаль, что система типов не может гарантировать корректность, но поскольку мы не можем вернуться назад во времени, это прагматичный выход из положения.

Применение обертки `UnmodifiableList` позволяет устранить зависимость от кода, изменяющего коллекцию. Однако возможность модификации *исходного списка* по-прежнему создает проблемы, т. к. обертка читает элементы своей внутренней коллекции и не гарантирует, что они никогда не изменятся, — мы можем быть уверены лишь в том, что их нельзя изменить, обращаясь к `UnmodifiableList`. В таких случаях, чтобы оградить себя от изменений, необходимо создать копию исходного списка, для подстраховки. В Java 10 появился метод `List.copyOf(collection)` для защитного копирования исходной коллекции в неизменяемый объект `AbstractImmutableList`, который не зависит от изменений, вносимых в эту коллекцию.

Все догадки о потенциальном инициаторе изменения (это может быть как источник коллекции, так и код, которому она передана) и принятие соответствующих мер требует много усилий и чревато ошибками. Отмеченная проблема касается любых изменяемых данных, но изменение коллекций является особенно опасным, т. к. мы нередко формируем значения, способные утратить свою актуальность, если изменится коллекция, из которой их извлекли (как в случае с `departsFrom`). Вместо того чтобы осуществлять защитное копирование на границе каждой функции, многие разработчики, включая авторов этой книги, применяют более простую и эффективную стратегию.

Не изменяйте разделяемые коллекции

Относитесь к любой коллекции, разделяемой между разными частями кода, как к неизменяемой. К разделению в том числе относятся ссылки, принимаемые в виде параметров, возвращаемые в качестве результата и присваиваемые разделяемым переменным. Это касается даже коллекций, которые изначально создавались изменяемыми, несмотря на изменяющие операции, присутствующие в интерфейсах Java.

Если код, который нам не принадлежит, не соблюдает это соглашение, мы можем использовать копии, чтобы оградить наш собственный код от изменений.

Эта стратегия не мешает нам создавать изменяемые коллекции и наполнять их внутри функции, однако коллекция должна изменяться только сразу после создания. После возвращения ссылки в качестве результата мы должны относиться к ней как к неизменяемой — *создавайте, а не изменяйте*. Иногда мы обеспечиваем соблюдение этой неизменяемости за счет создания обертки с помощью `Collections.unmodifiableList(...)` и прочих вызовов, но в слаженной команде делать это не обязательно, поскольку никто бы не стал вносить изменения в разделяемую коллекцию.

У этого правила, конечно же, есть исключения — места, где по соображениям производительности необходимо разделять изменяемую коллекцию. О таких исключениях можно сигнализировать в имени (можете начать с `accumulator`), как можно сильнее ограничивая область разделения. В идеале это лучше делать в рамках функции, или между частными методами класса, что тоже приемлемо, а вот за пределы модуля эти изменения выходят очень редко. В *главе 14* обсуждается, как в таких ситуациях избежать (заметного) изменения коллекции.

Команды, принявшие это соглашение, могут разрабатывать простое и надежное программное обеспечение вопреки изменяемости коллекций. В целом преимущества отказа от изменения коллекции перевешивают проблемы, связанные с тем, что система типов вводит вас в заблуждение, — ведь значения такие значимые. Некоторые библиотеки JVM уходят корнями во времена, когда изменяемость была нормой, однако Java все больше ориентируется на неизменяемые данные, и вам лучше быть на острие этих изменений, чем плестись позади.

Коллекции Kotlin

В отличие от Java, язык и стандартная библиотека Kotlin создавались в эпоху, когда изменяемость вышла из моды. Однако одной из ключевых целей было беспроблемное взаимодействие с Java, где имелись изменяемые коллекции. Создатели Scala пытались разработать собственные развитые коллекции с возможностью постоянного хранения (неизменяемые, но поддерживающие разделение данных), но это вынуждало разработчиков копировать информацию на границе взаимодействия двух языков, что было неэффективно и вызывало раздражение. Как Kotlin удалось избежать этой проблемы и сделать возможным органичное взаимодействие коллекций с Java?

Разработчики Kotlin убрали изменяющие методы из интерфейсов коллекций Java и опубликовали их в пакете `kotlin.collections` как `Collection<E>`, `List<E>` и т. д. Затем

дополнили пакет такими классами, как `MutableCollection<E>`, `MutableList<E>` и прочими, в которых снова появились изменяющие методы из Java. Таким образом в Kotlin у нас есть класс `MutableList`, который является подтипом класса `List`, наследующего `Collection.MutableList` и реализующего `MutableCollection`.

На первый взгляд все просто. Изменяемые коллекции поддерживают те же операции, что и неизменяемые, плюс изменяющие методы. Мы можем безопасно передать объект `MutableList` в качестве аргумента туда, где ожидается `List`, т. к. методы `List` присутствуют в этом объекте и доступны для вызова. Если использовать терминологию *принципа подстановки Лисков* (**https://oreil.ly/8A8KO**), мы можем подставить `MutableList` вместо `List`, не повлияв на корректность нашей программы.

Немного магии на этапе компиляции позволяет коду на Kotlin принимать `java.util.List` в качестве `kotlin.collections.List`:

```
val aList: List<String> = SomeJavaCode.mutableListOfStrings("0", "1")
aList.removeAt(1) // не компилировать
```

Благодаря этой же магии Kotlin может принять `List` из Java как `kotlin.collections.MutableList`:

```
val aMutableList: MutableList<String> = SomeJavaCode.mutableListOfStrings(
    "0", "1")
aMutableList.removeAt(1)
assertEquals(listOf("0"), aMutableList)
```

На самом деле, поскольку тип `List` в Java является изменяемым, мы могли бы выполнить нисходящее приведение к `MutableList` из Kotlin и затем внести изменения (чего почти никогда не следует делать):

```
val aList: List<String> = SomeJavaCode.mutableListOfStrings("0", "1")
val aMutableList: MutableList<String> = aList as MutableList<String>
aMutableList.removeAt(1)
assertEquals(listOf("0"), aMutableList)
```

С другой стороны, компилятор позволяет использовать как `kotlin.collections.MutableList`, так и `kotlin.collections.List` там, где требуется `java.util.List`:

```
val aMutableList: MutableList<String> = mutableListOf("0", "1")
SomeJavaCode.needsAList(aMutableList)


val aList: List<String> = listOf("0", "1")
SomeJavaCode.needsAList(aList)
```

На первый взгляд, все выглядит логично. К сожалению, когда речь идет об изменяемости, подстановка не ограничивается принципом Барбары Лисков. Как показано ранее в *разд. «Коллекции Java»*, тот факт, что нам не виден код, изменяющий нашу ссылку типа `kotlin.collections.List`, не означает, что ее содержимое не может поменяться. Исходным типом может выступать `java.util.List`, который *является* изменяемым. В Kotlin все в каком-то смысле еще хуже, т. к. объект `MutableList` может быть преобразован в `List` при передаче:

```
val aMutableList = mutableListOf("0", "1")
val aList: List<String> = aMutableList
```

Представьте, что мы где-то принимаем список `List<String>`, не сомневаясь в его неизменяемости:

```
class AValueType(
    val strings: List<String>
) {
    val first: String? = strings.firstOrNull()
}
```

Все выглядит хорошо:

```
val holdsState = AValueType(aList)
assertEquals(holdsState.first, holdsState.strings.first())
```

Но постойте, ведь у нас по-прежнему есть ссылка на `MutableList`:

```
aMutableList[0] = "banana"
assertEquals(holdsState.first, holdsState.strings.first()) ❶
```

❶ Ожидали `0`, получили `banana`.

Оказывается, тип `AValueType` все-таки изменяемый! И из-за этого первый элемент, который инициализируется в конструкторе, может устареть. То, что коллекции имеют неизменяемые интерфейсы, не сделало их неизменяемыми!

Неизменяемые, только для чтения, изменяемые

Согласно официальной версии неизменяемые коллекции Kotlin таковыми не являются. Это, скорее, представления коллекций, доступные только для чтения. Аналогично `UnmodifiableList` в Java — коллекции, доступные только для чтения, нельзя изменять с помощью их интерфейсов, однако изменения возможны с использованием других механизмов. Только по-настоящему неизменяемые коллекции гарантируют отсутствие любых изменений.

По-настоящему неизменяемые коллекции могут быть реализованы в JVM (как в случае с результатом `java.util.List.of(...)`), но это (пока еще) не является стандартной возможностью Kotlin.

Это нежелательное последствие того, что изменяемые коллекции наследуют «неизменяемые». Получатель «неизменяемой» коллекции не может ее модифицировать, но при этом никто не гарантирует, что она не изменится, т. к. ссылка («неизменяемого») типа `List` может в действительности указывать на объект типа `MutableList`.

Чтобы кардинально решить эту проблему, можно отделить изменяемые коллекции от неизменяемых за счет отказа от наследования. В таком случае, если у нас есть изменяемый список и нам нужно создать его неизменяемую копию, мы должны скопировать его данные. Хорошей аналогией является `StringBuilder`. Это фактически изменяемая версия `String`, которая не является подтипом `String`. Получив результат, который можно публиковать, мы должны вызвать `.toString()`, и последующие изменения `StringBuilder` не повлияют на предыдущие результаты. Clojure и

Scala применяют этот подход в своих неизменяемых коллекциях — почему же Kotlin не следует их примеру?

Ответ, наверное, состоит в том, что создатели Kotlin, как и авторы этой книги, руководствуются соглашением, описанным во *врезке «Не изменяйте разделяемые коллекции»*. Если любая коллекция, полученная в виде параметра, возвращенная в качестве результата или каким-то другим образом разделяемая между разными частями кода, считается неизменяемой, то наследование изменяемыми коллекциями неизменяемых оказывается достаточно безопасным. Хотя следует признать, что *достаточно* здесь означает *в основном*, а не *полностью*. Тем не менее преимущества перевешивают потенциальный риск.

Коллекции Kotlin делают этот подход еще более действенным. В Java мы имеем ситуацию, когда любая коллекция теоретически может быть изменена, поэтому система типов не показывает, в каких случаях это безопасно, а в каких — нет. А вот в Kotlin, если объявить все обычные ссылки «неизменяемыми», можно использовать `MutableCollection`, чтобы сигнализировать о том, что мы действительно ожидаем внесения изменений в коллекцию. Смирившись с этим по большому счету теоретическим риском, мы получаем взамен крайне простой и эффективный механизм взаимодействия с Java. Прагматизм обычно в духе Kotlin — в данном случае это можно выразить как «безопасность в разумных пределах, но не более того».

Как мы уже говорили, принцип «Не изменяйте разделяемые коллекции» можно выразить и по-другому: сделать так, чтобы изменение коллекций в нашем коде происходило лишь сразу после их создания. Этот подход можно увидеть в действии, если заглянуть в стандартную библиотеку Kotlin. Вот, к примеру, определение `map` (его упрощенная версия):

```
inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {
    val result = ArrayList<R>()
    for (item in this)
        result.add(transform(item))
    return result
}
```

Здесь список формируется путем изменений и затем возвращается с доступом только для чтения. Просто *и* эффективно. Теоретически мы *могли бы* выполнить нисходящее приведение к `MutableList` и изменить результат, но делать этого не следует. Тип результата нужно принимать в том виде, в котором он возвращается. Таким образом, даже если вы разделяете эту коллекцию, вам можно не беспокоиться о ее изменении.

Рефакторинг от коллекций Java к коллекциям Kotlin

Благодаря органичному взаимодействию коллекций Java и Kotlin, описанному ранее, преобразование кода, в котором они используются, обычно не составляет труда, по крайней мере на синтаксическом уровне. Если наш код на Java вносит

изменения в коллекции, нам, возможно, придется приложить дополнительные усилия, чтобы не нарушить инварианты в Kotlin.

Прежде чем переводить наш Java-код на Kotlin, будет полезно привести его в соответствие с соглашением, описанным во *врезке «Не изменяйте разделяемые коллекции»*. Этим мы и займемся.

Подправим код на Java

Вернемся к коду из `Travelator`, который мы рассматривали ранее. Статические методы, о которых шла речь, находятся в классе с именем `Suffering` (пример 6.5).

Пример 6.5 [collections.0:src/main/java/travelator/Suffering.java]

```
public class Suffering {

    public static int sufferScoreFor(List<Journey> route) {
        Location start = getDepartsFrom(route);
        List<Journey> longestJourneys = longestJourneysIn(route, 3);
        return sufferScore(longestJourneys, start);
    }

    public static List<Journey> longestJourneysIn(
        List<Journey> journeys,
        int limit
    ) {
        journeys.sort(comparing(Journey::getDuration).reversed()); ❶
        var actualLimit = Math.min(journeys.size(), limit);
        return journeys.subList(0, actualLimit);
    }

    public static List<List<Journey>> routesToShowFor(String itineraryId) {
        var routes = routesFor(itineraryId);
        removeUnbearableRoutes(routes);
        return routes;
    }

    private static void removeUnbearableRoutes(List<List<Journey>> routes) {
        routes.removeIf(route -> sufferScoreFor(route) > 10);
    }

    private static int sufferScore(
        List<Journey> longestJourneys,
        Location start
    ) {
        return SOME_COMPLICATED_RESULT();
    }
}
```

❶ `longestJourneysIn` изменяет свой параметр, чем нарушает наше правило.

Как здесь показано, `longestJourneysIn` изменяет свой параметр, поэтому у нас нет возможности изменить порядок выполнения `getDepartsFrom`, а также получить `longestJourneysIn` в качестве аргумента `sufferScore`. Прежде чем мы сможем это исправить, необходимо убедиться в том, что никакой другой код не зависит от этого изменения. Это может быть непросто, что само по себе является хорошим поводом запретить изменение коллекций с самого начала. Если мы уверены в своих тестах, то можно попробовать внести изменение и посмотреть, ничего ли не сломалось. В противном случае можно добавить новые тесты и/или провести анализ кода и зависимостей. Давайте внесем изменение в `Travelator`, исходя из того, что это безопасно.

Мы не хотим сортировать имеющуюся коллекцию, поэтому нам нужна функция, которая возвращает отсортированную копию списка, не изменяя оригинал. Даже в Java 16, похоже, такой функции не предусмотрено. Интересно, что `List.sort` создает отсортированную копию и затем изменяет собственный список:

```
@SuppressWarnings({"unchecked", "rawtypes"})
default void sort(Comparator<? super E> c) {
    Object[] a = this.toArray();
    Arrays.sort(a, (Comparator) c);
    ListIterator<E> i = this.listIterator();
    for (Object e : a) {
        i.next();
        i.set((E) e);
    }
}
```

Это еще раз показывает, что во времена выхода версии 8, когда этот код был написан, язык Java ориентировался на изменяемые данные. Сейчас нам доступен метод `Stream.sorted`, но, как показывает наш опыт, потоки данных редко дают хорошие результаты при работе с небольшими коллекциями (см. *главу 13*). Возможно, здесь и не стоит заботиться о производительности, но мы не удержались! И оправдываем свою склонность к применению алгоритма сортировки тем доводом, что в нашем коде есть несколько мест, где сортируются имеющиеся списки, поэтому их пришлось бы отредактировать, чтобы избежать изменения разделяемых коллекций. Авторы `List.sort`, как нам кажется, кое-что понимали в эффективности Java, поэтому мы скопируем их код и напишем следующее (пример 6.6).

Пример 6.6 [collections.3:src/main/java/travelator/Collections.java]

```
@SuppressWarnings("unchecked")
public static <E> List<E> sorted(
    Collection<E> collection,
    Comparator<? super E> by
) {
    var result = (E[]) collection.toArray();
    Arrays.sort(result, by);
    return Arrays.asList(result);
}
```

Прежде чем продолжать, следует подумать о том, как удостовериться в корректности этого кода. Сделать это крайне сложно, учитывая вносимое изменение. Мы должны убедиться, что `Arrays.sort` не влияет на входную коллекцию, — для этого нужно свериться с документацией `Collection.toArray`. Там мы находим заветную фразу: «Следовательно, вызывающему коду позволено изменять возвращенный массив», поэтому все хорошо. Ввод от вывода получилось отделить. Эта функция является классическим примером того, как изменение допускается в том участке кода, где создается коллекция, но не за его пределами — создавайте, а не изменяйте.

Давайте заодно подумаем о том, какой результат мы возвращаем, и является ли он мутирующим (изменяющимся). `Arrays.asList` возвращает экземпляр `ArrayList`, однако он не является стандартным. Этот экземпляр представляет собой частный член `Arrays` и выполняет запись в нашу переменную `result`. Но, поскольку он основан на массиве, добавление и удаление элементов не поддерживается. Мы не можем изменить его размер. Оказывается, коллекции в Java бывают не только изменяемыми (mutable), как бы «неизменяемыми» (nonmutable) и неизменяемыми (immutable) — иногда они могут быть изменяемыми, но при условии, что их структура остается без изменений! Ни одно из этих различий не отражено в системе типов, поэтому наши изменения могут полностью соответствовать типу и при этом провоцировать ошибку во время работы в зависимости от того, какой из путей выполнения даст нам коллекцию, которую мы в последствии попытаемся модифицировать, и характера этих модификаций. Это еще одна причина, чтобы полностью избавиться от этой проблемы и никогда не изменять разделяемые коллекции.

Возвращаясь к нашему рефакторингу, мы можем воспользоваться одной из наших новых функций `sorted` внутри `longestJourneysIn`, чтобы отказаться от модификации разделяемой коллекции.

При использовании `sort` у нас был следующий код (пример 6.7).

Пример 6.7 [collections.2:src/main/java/travelator/Suffering.java]

```
public static List<Journey> longestJourneysIn(
    List<Journey> journeys,
    int limit
) {
    journeys.sort(comparing(Journey::getDuration).reversed());
    var actualLimit = Math.min(journeys.size(), limit);
    return journeys.subList(0, actualLimit);
}
```

Наша новая функция `sorted` позволяет написать (пример 6.8).

Пример 6.8 [collections.3:src/main/java/travelator/Suffering.java]

```
static List<Journey> longestJourneysIn(
    List<Journey> journeys,
```

```
    int limit
) {
    var actualLimit = Math.min(journeys.size(), limit);
    return sorted(
        journeys,
        comparing(Journey::getDuration).reversed()
    ).subList(0, actualLimit);
}
```

Поскольку теперь побочный эффект в `longestJourneysIn` не оказывает влияния на метод `sufferScoreFor`, мы можем подставить его локальные переменные (пример 6.9).

Пример 6.9 [collections.4:src/main/java/travelator/Suffering.java]

```
public static int sufferScoreFor(List<Journey> route) {
    return sufferScore(
        longestJourneysIn(route, 3),
        getDepartsFrom(route));
}
```

Подстановка локальных переменных, может, и не приносит существенной пользы, но иллюстрирует нечто большее. В *главе 7* вы увидите, как отказ от изменения данных позволяет проводить рефакторинг кода такими способами, которые в противном случае были бы небезопасными.

Если перейти на уровень выше и взглянуть на код, вызывающий `sufferScoreFor`, можно обнаружить следующее (пример 6.10).

Пример 6.10 [collections.4:src/main/java/travelator/Suffering.java]

```
public static List<List<Journey>> routesToShowFor(String itineraryId) {
    var routes = routesFor(itineraryId);
    removeUnbearableRoutes(routes);
    return routes;
}

private static void removeUnbearableRoutes(List<List<Journey>> routes) {
    routes.removeIf(route -> sufferScoreFor(route) > 10);
}
```

Хммм, это настолько патологически мутирует, что можно было бы привести в книге в качестве примера! Как минимум за счет возвращения `void` функция `removeUnbearableRoutes` говорит нам о том, что она должна что-то изменить. Мы можем постепенно переписать ее так, чтобы она возвращала изменяемый ею параметр, и использовать этот результат — еще один случай, когда, прежде чем двигаться вперед, нужно сделать шаг назад (пример 6.11).

Пример 6.11 [collections.5:src/main/java/travelator/Suffering.java]

```
public static List<List<Journey>> routesToShowFor(String itineraryId) {
    var routes = routesFor(itineraryId);
    routes = removeUnbearableRoutes(routes);
    return routes;
}

private static List<List<Journey>> removeUnbearableRoutes
    (List<List<Journey>> routes
) {
    routes.removeIf(route -> sufferScoreFor(route) > 10);
    return routes;
}
```

На этот раз мы воспользуемся `Stream.filter` вместо изменяющей операции в методе `removeUnbearableRoutes` и заодно изменим название этого метода (пример 6.12).

Пример 6.12 [collections.6:src/main/java/travelator/Suffering.java]

```
public static List<List<Journey>> routesToShowFor(String itineraryId) {
    var routes = routesFor(itineraryId);
    routes = bearable(routes);
    return routes;
}

private static List<List<Journey>> bearable
    (List<List<Journey>> routes
) {
    return routes.stream()
        .filter(route -> sufferScoreFor(route) <= 10)
        .collect(toUnmodifiableList());
}
```

Обратите внимание, насколько легко подобрать для нашей функции красивое короткое имя: `bearable` вместо `removeUnbearableRoutes`.

Повторное присваивание значения переменной `routes` в `routesToShowFor` выглядит безобразно, но сделано неслучайно, т. к. это позволяет нам провести параллели с рефакторингом в *главе 5*. Там мы отказались от изменения существующих данных в пользу замены ссылки измененным значением, и именно это мы сделали здесь. Конечно, эта локальная переменная совершенно лишняя, поэтому давайте от нее избавимся. Вызов встроенного рефакторинга дважды делает это хорошо (пример 6.13).

Пример 6.13 [collections.7:src/main/java/travelator/Suffering.java]

```
public static List<List<Journey>> routesToShowFor(String itineraryId) {
    return bearable(routesFor(itineraryId));
}

private static List<List<Journey>> bearable
    (List<List<Journey>> routes
) {
    return routes.stream()
        .filter(route -> sufferScoreFor(route) <= 10)
        .collect(toUnmodifiableList());
}
```

Преобразование в Kotlin

Итак, мы убрали все изменяющие операции из наших коллекций в Java. Теперь пришло время перевести код на Kotlin. Действие **Convert Java File to Kotlin File** (Преобразовать файл Java в файл Kotlin) для класса `Suffering` показывает неплохие результаты, но на момент подготовки этой книги оно запуталось и нарушило обнуляемость коллекций и их обобщенных типов. После преобразования нам пришлось убрать знаки `?` из некоторых неприятных на вид типов вроде `List<List<Journey?>>?`, чтобы получить следующий результат (пример 6.14).

Пример 6.14 [collections.8:src/main/java/travelator/Suffering.kt]

```
object Suffering {
    @JvmStatic
    fun sufferScoreFor(route: List<Journey>): Int {
        return sufferScore(
            longestJourneysIn(route, 3),
            Routes.getDepartsFrom(route)
        )
    }

    @JvmStatic
    fun longestJourneysIn(
        journeys: List<Journey>,
        limit: Int
    ): List<Journey> {
        val actualLimit = Math.min(journeys.size, limit)
        return sorted(
            journeys,
            comparing { obj: Journey -> obj.duration }.reversed()
        ).subList(0, actualLimit)
    }
```

```
    fun routesToShowFor(itineraryId: String?): List<List<Journey>> {
        return bearable(Other.routesFor(itineraryId))
    }

    private fun bearable(routes: List<List<Journey>>): List<List<Journey>> {
        return routes.stream()
            .filter { route -> sufferScoreFor(route) <= 10 }
            .collect(Collectors.toUnmodifiableList())
    }

    private fun sufferScore(
        longestJourneys: List<Journey>,
        start: Location
    ): Int {
        return SOME_COMPLICATED_RESULT()
    }
}
```

Мы также переформатировали и почистили некоторые операции импорта. С другой стороны, нам не пришлось изменять код на Java, который обращается к нашему коду на Kotlin. Например, в примере 6.15 показан тест, который передает обычный список `List` из Java методу `longestJourneyIn`, написанному на Kotlin.

Пример 6.15 [collections.8:src/test/java/travelator/LongestJourneyInTests.java]

```
@Test public void returns_limit_results() {
    assertEquals(
        List.of(longJourney, mediumJourney),
        longestJourneysIn(List.of(shortJourney, mediumJourney, longJourney), 2)
    );
}
```

Возвращаясь к Kotlin, мы теперь можем воспользоваться многими служебными методами, доступными в коллекциях этого языка, чтобы упростить наш код. Возьмем в качестве примера метод `longestJourneysIn`. Раньше он выглядел так (пример 6.16).

Пример 6.16 [collections.8:src/main/java/travelator/Suffering.kt]

```
@JvmStatic
fun longestJourneysIn(
    journeys: List<Journey>,
    limit: Int
): List<Journey> {
    val actualLimit = Math.min(journeys.size, limit)
    return sorted(
        journeys,
```

```
        comparing { obj: Journey -> obj.duration }.reversed()
    ).subList(0, actualLimit)
}
```

Использование `sortedByDescending` и `take` вместо `sorted` и соответственно `subList` дает следующий результат (пример 6.17).

Пример 6.17 [collections.9:src/main/java/travelator/Suffering.kt]

```
@JvmStatic
fun longestJourneysIn(journeys: List<Journey>, limit: Int): List<Journey> =
    journeys.sortedByDescending { it.duration }.take(limit)
```

Теперь, если преобразовать `longestJourneysIn` в функцию-расширение (см. *главу 10*), ее имя можно упростить до `longestJourneys` (пример 6.18).

Пример 6.18 [collections.10:src/main/java/travelator/Suffering.kt]

```
@JvmStatic
fun List<Journey>.longestJourneys(limit: Int): List<Journey> =
    sortedByDescending { it.duration }.take(limit)
```

Поскольку функция `longestJourneys` не изменяет свои параметры, мы оформили ее в виде единого выражения (см. *главу 9*). Эту функцию по-прежнему можно вызывать из Java в виде статического метода, но использование ее в Kotlin выглядит куда элегантнее, особенно если указать аргумент с именем (пример 6.19).

Пример 6.19 [collections.10:src/main/java/travelator/Suffering.kt]

```
@JvmStatic
fun sufferScoreFor(route: List<Journey>): Int {
    return sufferScore(
        route.longestJourneys(limit = 3), ❶
        Routes.getDepartsFrom(route)
    )
}
```

❶ Именованный аргумент.

Теперь рассмотрим `bearable` (пример 6.20).

Пример 6.20 [collections.10:src/main/java/travelator/Suffering.kt]

```
private fun bearable(routes: List<List<Journey>>): List<List<Journey>> {
    return routes.stream()
        .filter { route -> sufferScoreFor(route) <= 10 }
        .collect(Collectors.toUnmodifiableList())
}
```

Здесь можно воспользоваться методиками из *главы 13*, чтобы преобразовать `Stream` в синтаксис Kotlin. Мы убрали вызов `.stream()`, т. к. в Kotlin у `List` есть функция-расширение `filter`. Нам также не нужен заключительный вызов `toUnmodifiableList`, поскольку в Kotlin `filter` сразу возвращает `List` (пример 6.21).

Пример 6.21 [collections.11:src/main/java/travelator/Suffering.kt]

```
private fun bearable(routes: List<List<Journey>>): List<List<Journey>> =
    routes.filter { sufferScoreFor(it) <= 10 }
```

Интересно, что в этом участке кода результат может оказаться даже более изменяемым, чем в нашем коде на Java. В Java мы собирали его с помощью `Collectors.toUnmodifiableList()`. В Kotlin функция `filter` объявлена с возвращаемым значением типа `List` (в представлении, доступном только для чтения), однако на этапе выполнения в качестве типа используется `ArrayList`. Если полностью отказаться от нисходящего приведения типов, это не вызовет проблем, особенно учитывая, что мы теперь обращаемся с нашими разделяемыми коллекциями как с неизменяемыми даже в Java.

Вот наш итоговый код (пример 6.22).

Пример 6.22 [collections.11:src/main/java/travelator/Suffering.kt]

```
object Suffering {
    @JvmStatic
    fun sufferScoreFor(route: List<Journey>): Int =
        sufferScore(
            route.longestJourneys(limit = 3),
            Routes.getDepartsFrom(route)
        )

    @JvmStatic
    fun List<Journey>.longestJourneys(limit: Int): List<Journey> =
        sortedByDescending { it.duration }.take(limit)

    fun routesToShowFor(itineraryId: String?): List<List<Journey>> =
        bearable(routesFor(itineraryId))

    private fun bearable(routes: List<List<Journey>>): List<List<Journey>> =
        routes.filter { sufferScoreFor(it) <= 10 }

    private fun sufferScore(
        longestJourneys: List<Journey>,
        start: Location
    ): Int = SOME_COMPLICATED_RESULT()
}
```

Мы называем этот вариант итоговым, но на практике наш рефакторинг на этом бы не остановился. Такие типы, как `List<List<Journey>>`, указывают на то, что некий другой тип стоило бы извлечь на более высокий уровень кода, но в Kotlin мы обычно не используем публичные статические методы в подобных объектах, отдавая предпочтение функциям верхнего уровня. И эта проблема будет наконец-то исправлена в *главе 8*.

Двигаемся дальше

Когда-то программирование на Java ориентировалось на изменяемость. Эта практика уже вышла из моды, но лишь за счет соглашений. Она по-прежнему доступна для использования. В Kotlin применяется очень прагматичный подход к изменяемости коллекций, что обеспечивает беспроблемную работу и простую модель программирования, но только если это не противоречит соглашениям, принятым в Java.

Чтобы сделать взаимодействие между Java и Kotlin более гладким:

- помните, что Java может изменять коллекцию, переданную в код на Kotlin;
- помните, что Java может, по крайней мере, попытаться изменить коллекцию, полученную из Kotlin;
- избегайте изменяющих операций при использовании коллекций в Java. Если без этого не обойтись, создавайте защитные копии.

Обсуждение коллекций будет продолжено в *главе 15 «От инкапсулированных коллекций к псевдонимам типов»*. К примеру кода, рассмотренному здесь, мы вернемся в *главе 8 «От статических методов к функциям верхнего уровня»*.

ГЛАВА 7

От действий к вычислениям

Ни в Java, ни в Kotlin не существует формального различия между императивным и функциональным кодом, хотя тот акцент, который Kotlin делает на неизменяемости и выражениях, в целом приводит к написанию более функциональных программ. Можем ли мы улучшить свой код, сделав его в большей степени функциональным?

Функции

Программисты придумали много разных терминов для обозначения участков кода, вызываемых внутри программы. Так, мы имеем очень общий термин *подпрограмма*. Некоторые языки (в частности, Pascal) различают *функции* и *процедуры*: первые возвращают результат, а вторые — нет, но большинство разработчиков относятся к ним как с синонимам. Есть также *методы* — подпрограммы, привязанные к объекту (или к классу — в случае со статическими методами).

Язык C называет все это *функциями*, но предлагает специальный тип `void` для представления отсутствия возвращаемого значения. Этот подход был унаследован языком Java. В Kotlin почти такую же роль играет тип `Unit`, только это не отсутствие возвращаемого значения, а объект-одиночка, который возвращается вместо него.

В этой книге термин *функция* применяется по отношению к подпрограммам независимо от того, возвращают ли они что-нибудь и привязаны ли они к объекту. В случаях, когда их принадлежность к объекту имеет значение, будем называть их *методами*.

Как бы мы их ни называли, функции являются одним из основополагающих составных элементов нашего программного обеспечения. И определяем мы их в каком-то формате обычно в рамках используемого языка программирования. Они также, как правило, остаются без изменений на протяжении работы программы — по крайней мере, в статических языках функции на лету обычно не переопределяют.

С другой стороны, существует совершенно другой основополагающий составной элемент — *данные*. Ожидается, что в ходе выполнения программы данные будут варьироваться, и мы привязываем их к разным *переменным*. Переменные носят такое название, потому что они, что бы вы думали? Правильно, меняются. Даже будучи окончательными (`final`), они обычно привязываются к разным данным во время разных вызовов функции.

Чуть ранее мы упоминали о том, что функции делятся на те, что возвращают результат, и те, которые этого не делают. Это может показаться фундаментальным различием, но на практике функции лучше разделять на *вычисления* и *действия*.

Действие — это функция, зависящая от того, сколько раз она вызывается. У вычислений такой зависимости нет — они вневременные. Большинство написанных нами функций являются действиями, т. к. написание кода, который всегда выполняется одинаково, требует к себе особого внимания. Но если бы нам нужно было это сделать, как бы мы к этому подошли?

Вычисления

Чтобы считаться вычислением, функция всегда должна возвращать один и тот же результат при одних и тех же входных данных. Входными данными для функции являются ее *параметры*, которые привязываются к *аргументам* при вызове функции. Таким образом, вычисление всегда возвращает один и тот же результат при вызове с одними и теми же аргументами.

Возьмем функцию `fullName`:

```
fun fullName(customer: Customer) = "${customer.givenName} ${customer.familyName}"
```

`fullName` здесь является вычислением — она всегда будет возвращать одно и то же значение при отправке одному и тому же `Customer`.

Это справедливо только в том случае, если `Customer` является немутирующим, или, по крайней мере, `givenName` и `family Name` не могут быть изменены. Чтобы упростить задачу, мы скажем, что вычисления могут иметь только те параметры, которые являются значениями, как определено в *главе 5*.

Методы и скрытые методы, являющиеся свойствами членов, также могут быть вычислениями:

```
data class Customer(
    val givenName: String,
    val familyName: String
) {
    fun upperCaseGivenName() = givenName.toUpperCase()

    val fullName get() = "$givenName $familyName"
}
```

Для метода или расширения получатель `this` и любое свойство, доступ к которому осуществляется через `this`, также являются входными данными. То есть функции `upperCaseGivenName` и `fullName` обе будут вычислениями, потому что `given Name` и `familyName` — это значения.

Функция-расширение или свойство также могут быть вычислениями, если данные, от которых они зависят, являются значениями:

```
fun Customer.fullName() = "$givenName $familyName"

val Customer.fullName get() = "$givenName $familyName"
```

Вычисляемое свойство или функция?

Возможно, вы задавались вопросом, когда следует определять вычисляемое свойство, а когда использовать функцию, возвращающую результат. Вычисляемые свойства сбивают с толку, если они возвращают разные результаты в разное время, — по крайней мере, когда они определены для типов значений (и вы уже понимаете: ваши авторы считают, что большинство наших типов должны быть типами значений). Поэтому хорошим эмпирическим правилом является резервирование рассчитанных свойств для вычислений. Мы подробно остановимся на этой теме в *главе 11*.

Результат вычисления может зависеть от данных, которые не передаются в качестве параметров, но только в том случае, если эти данные не изменяются. В противном случае результат функции отличался бы до и после изменения, что сделало бы его действием. Даже если функция всегда возвращает один и тот же результат для одних и тех же параметров, она все равно может быть действием, если она что-то изменяет (либо параметр, либо внешний ресурс — такой как глобальная переменная или база данных). Например:

```
println("hello")
```

Функция `println` всегда возвращает один и тот же результат `Unit`, выдающий одни и те же введенные данные `hello`, но она не является вычислением, она — действие.

Действия

`println` — это действие, потому что оно зависит от того, когда и сколько раз *выполняется*. Если мы не вызываем его, ничего не выводится в качестве результата, что отличается от вызова действия один раз и отличается от вызова действия дважды. Порядок, в котором мы вызываем `println` с разными аргументами, также влияет на результаты, представленные на экране.

Мы вызываем `println` из-за его *побочного эффекта* — эффекта, который он оказывает на окружающую его среду. Побочный эффект — это немного вводящий в заблуждение термин. От побочных эффектов лекарств он отличается тем, что побочные эффекты действия нам полезны. Возможно, лучшим названием было бы *внешние эффекты*. Это подчеркнет, что они находятся за пределами параметров функции, локальных переменных и возвращаемого значения. В любом случае функции с наблюдаемыми побочными эффектами — это действия, а не вычисления. Функции, возвращающие `void` или `Unit`, почти всегда являются действиями, потому что, если они что-то выполняют, они должны выполнять это за счет побочного эффекта.

Как мы видели ранее, код, считывающий данные из внешнего мутирующего состояния, также должен быть действием (при условии, когда что-либо действительно изменяет состояние).

Давайте рассмотрим сервис `Customers`:

```
class Customers {
    fun save(data: CustomerData): Customer {
        ...
    }
```

```
    fun find(id: String): Customer? {
        ...
    }
}
```

Функции `save` и `find` являются действиями — `save` создает новую запись клиента в базе данных и возвращает данные записи. Это именно действие, потому что состояние нашей базы данных зависит от того, когда мы ее вызываем. Результат `find` также чувствителен к времени вызова, поскольку зависит от предыдущих вызовов `save`.

Функции, у которых нет параметров (это не включает методы или функции-расширения, которые могут иметь неявные параметры, доступные через `this`), должны либо возвращать константу, либо считываться из какого-либо другого источника и поэтому классифицироваться как действия. Не глядя на ее источник, мы можем сделать вывод, что функция верхнего уровня `requestRate` почти наверняка является действием, считываемым из некоторого глобального мутирующего состояния:

```
fun requestRate(): Double {
    ...
}
```

Если функция с той же очевидной сигнатурой определена как метод, она, вероятно, представляет собой вычисление, которое зависит от свойств `Metrics` (при условии, что `Metrics` является неизменяемым):

```
class Metrics(
    ...
) {

    fun requestRate(): Double {
        ...
    }
}
```

Мы говорим, *вероятно*, потому что в таких языках, как Java или Kotlin, которые позволяют вводить, выводить или получать доступ к глобальным мутирующим данным из любого кода, невозможно быть уверенным, представляет ли функция вычисление или действие, если не проверять ее и все функции, которые она вызывает. Мы вскоре вернемся к этой проблеме.

Почему это должно нас волновать?

Очевидно, что мы должны обратить особое внимание на некоторые действия в нашем программном обеспечении. Отправка одного и того же электронного письма каждому пользователю дважды — это ошибка, так же как и его отсутствие вообще. Нас волнует, сколько именно раз оно будет отправлено. Мы можем даже позаботиться об отправке письма ровно в 8:07 утра, чтобы наше предложение о бесплат-

ном повышении класса обслуживания находилось в верхней части почтового ящика наших клиентов, когда они проверяют электронную почту за завтраком.

Другие, казалось бы, безвредные действия могут быть более вредоносными, чем мы думаем. Изменение порядка действий чтения и записи приводит к ошибкам параллелизма. Обработка ошибок сильно усложняется, если второе из двух последовательных действий завершается неудачей после успешного выполнения первого. Действия не дают нам полной свободы в рефакторинге кода, поскольку на этот процесс может повлиять время или вероятность их вызова.

Вычисления, с другой стороны, могут быть вызваны в любое время, без каких-либо последствий их повторного вызова с одними и теми же аргументами, кроме пустой траты времени и энергии. Когда мы проводим рефакторинг кода и обнаруживаем, что нам не нужен результат вычисления, то можем смело не вызывать его. Если это ценное вычисление, мы можем смело кешировать его результат. Если оно неценное, можем безопасно пересчитать его по требованию, если это упрощает ситуацию. Именно это чувство безопасности вызывает самодовольную улыбку на лицах функциональных программистов (ну, и знание того, что монада — это всего лишь моноид в категории эндофункторов). У функциональных программистов также есть термин для обозначения свойства функции, которое делает ее вычислением, — *ссылочная прозрачность*. Если функция ссылочно прозрачна, мы можем заменить ее вызов ее результатом, но можем сделать это только в том случае, если не имеет значения, когда или будем ли вообще ее вызывать.

Процедурное программирование

Нэт и Дункан — достаточно взрослые, и они учились программировать на языке Sinclair BASIC на компьютере ZX81. Этот диалект не имел неизменяемых данных и не поддерживал подпрограммы, параметры или локальные переменные. Программирование в такой системе требует настоящего порядка, потому что практически каждая строка кода является действием и поэтому потенциально влияет на функционирование любого другого оператора.

Это очень близко к тому, как на самом деле работают наши компьютеры с мутирующими значениями. Они хранятся в регистрах и глобальной памяти и управляются действиями машинного кода. Эволюция языков программирования была процессом ограничения максимальной гибкости этой модели, чтобы люди могли лучше рассуждать о создаваемом ими коде.

Почему предпочтение отдается вычислениям?

Нам нравятся вычисления, потому что с ними намного проще работать, но в конечном счете программное обеспечение должно оказывать влияние на мир, а это уже действие. Однако здесь нет места дублированию. Код не может быть действием и вычислением, как вневременными, так и зависящими от времени. Если мы возьмем некоторый код, представляющий собой вычисление, и заставим его вызвать действие, тогда оно станет действием, потому что теперь вычисление будет зависеть от того, будет ли оно вызвано или когда. Мы можем думать о вычислениях как о более чистом коде, где код наследует наиболее испорченный уровень всех своих зависимостей. То же самое мы видим и в случае с восприимчивостью к ошибкам (см. *гла-*

ву 19). Если мы ценим чистоту (которая во всех этих случаях обеспечивает простоту рассуждений и рефакторинга), то должны стремиться перенести границу между нечистым и чистым кодом на внешние уровни нашей системы — те, которые ближе всего к точкам входа. И если добьемся успеха, то значительная часть нашего кода может представлять собой вычисления и, следовательно, может быть легко протестирована, аргументирована и переработана.

Что, если нам не удастся сохранить действия в нижней части нашего стека вызовов? Тогда мы сможем все исправить с помощью рефакторинга!

Рефракторинг действий в вычисления

Давайте посмотрим на распознавание и рефакторинг действий в существующем коде.

Существующий код

В `Travelator` есть конечная точка HTTP, которая позволяет клиентскому приложению получать информацию о текущей поездке клиента (пример 7.1).

Пример 7.1 [actions.0:src/main/java/travelator/handlers/CurrentTripsHandler.java]

```
public class CurrentTripsHandler {

    private final ITrackTrips tracking;
    private final ObjectMapper objectMapper = new ObjectMapper();

    public CurrentTripsHandler(ITrackTrips tracking) {
        this.tracking = tracking;
    }

    public Response handle(Request request) {
        try {
            var customerId = request.getQueryParam("customerId").stream()
                .findFirst();
            if (customerId.isEmpty())
                return new Response(HTTP_BAD_REQUEST);
            var currentTrip = tracking.currentTripFor(customerId.get());
            return currentTrip.isPresent() ?
                new Response(HTTP_OK,
                    objectMapper.writeValueAsString(currentTrip)) :
                new Response(HTTP_NOT_FOUND);
        } catch (Exception x) {
            return new Response(HTTP_INTERNAL_ERROR);
        }
    }
}
```

Действия — это код, чувствительный ко времени, поэтому такие фрагменты, как `current` в `CurrentTripsHandler`, являются верным признаком действия. Метод `handle` — это действие, и это нормально: операции на границе наших систем часто являются таковыми.

Обработчик делегирует некоторую бизнес-логику, реализованную в `Tracking` (пример 7.2).

Пример 7.2 [actions.0:src/main/java/travelator/Tracking.java]

```
class Tracking implements ITrackTrips {

    private final Trips trips;

    public Tracking(Trips trips) {
        this.trips = trips;
    }

    @Override
    public Optional<Trip> currentTripFor(String customerId) {
        var candidates = trips.currentTripsFor(customerId).stream()
            .filter((trip) -> trip.getBookingStatus() == BOOKED)
            .collect(toList());
        if (candidates.size() == 1)
            return Optional.of(candidates.get(0));
        else if (candidates.size() == 0)
            return Optional.empty();
        else
            throw new IllegalStateException(
                "Unexpectedly more than one current trip for " + customerId
        );
    }
}
```

Правило `current` дает понять, что `Tracking.currentTripFor`, очевидно, является действием, как и `Trips.currentTripsFor`. Вот его реализация в `InMemoryTrips`, которая используется для тестирования вместо версии, реализованной с помощью запросов к базе данных (пример 7.3).

Пример 7.3 [actions.0:src/test/java/travelator/InMemoryTrips.java]

```
public class InMemoryTrips implements Trips {

    ...
    @Override
    public Set<Trip> currentTripsFor(String customerId) {
        return tripsFor(customerId).stream()
```

```
            .filter(trip -> trip.isPlannedToBeActiveAt(clock.instant()))
            .collect(toSet());
    }
}
```

Преобразования из `Set<Trip>` (результат выполнения `Trips.currentTripsFor`) и в `Optional<Trip>` (возвращенное из `Tracking.currentTripFor`), по-видимому, связаны с тем, что существует какое-то бизнес-правило. Оно обусловлено тем, что в любое время может быть только одна активная поездка — правило, которое не применяется на уровне сохранения.

Пока мы не дошли до этого этапа, то полагались на наши знания значений слов (в частности, `current`), чтобы сделать вывод, какие методы Java представляют собой действия, а не вычисления. Здесь, однако, мы получили неопровержимые доказательства. Вы это заметили?

Да, функция `clock.instant()` определенно зависит от того, когда мы ее вызовем (если вы нашли другое действие — отлично, но пока держите его при себе. Мы еще вернемся к этому). Даже если бы мы решили не продолжать оставшуюся часть рефакторинга, есть одно изменение, которое необходимо внести сейчас. Мы обсуждали вычисления и действия, применимые к именованным блокам кода, но они также применимы на уровне выражений. Как только вы начнете отличать действия от вычислений, имеет смысл не использовать случайное действие в чистом вычислении. Давайте удалим действие так, чтобы оставшаяся часть выражения осталась чистой: выберите `clock.instant()` и **Представьте переменную** (Introduce Variable), назвав ее `now` (пример 7.4).

Пример 7.4 [actions.1:src/test/java/travelator/InMemoryTrips.java]

```
@Override
public Set<Trip> currentTripsFor(String customerId) {
    return tripsFor(customerId).stream()
        .filter(trip -> {
            Instant now = clock.instant();
            return trip.isPlannedToBeActiveAt(now);
        })
        .collect(toSet());
}
```

Она все еще в середине выражения — давайте поднимем ее выше (и по пути преобразуем в `var`), как показано в примере 7.5.

Пример 7.5 [actions.2:src/test/java/travelator/InMemoryTrips.java]

```
@Override
public Set<Trip> currentTripsFor(String customerId) {
    var now = clock.instant();
```

```
    return tripsFor(customerId).stream()
        .filter(trip -> trip.isPlannedToBeActiveAt(now))
        .collect(toSet());
}
```

Это простое действие позволило нам понять, что раньше мы сравнивали каждую поездку с немного разным временем! Было ли это проблемой? Вероятно, не здесь, но вы, может быть, работали над системами, где это было бы актуально. Дункан, например, недавно закончил диагностику проблемы, из-за которой половина банковской транзакции может учитываться за один день, а другая половина — за следующий.

Помимо того, что действия затрудняют рефакторинг нашего кода, они также усложняют его тестирование. Давайте посмотрим, как это проявляется (пример 7.6).

Пример 7.6 [actions.0:src/test/java/travelator/TrackingTests.java]

```
public class TrackingTests {

    final StoppedClock clock = new StoppedClock();

    final InMemoryTrips trips = new InMemoryTrips(clock);
    final Tracking tracking = new Tracking(trips);

    @Test
    public void returns_empty_when_no_trip_planned_to_happen_now() {
        clock.now = anInstant();
        assertEquals(
            Optional.empty(),
            tracking.currentTripFor("aCustomer")
        );
    }

    @Test
    public void returns_single_active_booked_trip() {
        var diwaliTrip = givenATrip("cust1", "Diwali",
            "2020-11-13", "2020-11-15", BOOKED);
        givenATrip("cust1", "Christmas",
            "2020-12-24", "2020-11-26", BOOKED);

        clock.now = diwaliTrip.getPlannedStartTime().toInstant();
        assertEquals(
            Optional.of(diwaliTrip),
            tracking.currentTripFor("cust1")
        );
    }

    ...
}
```

Чтобы получить предсказуемые результаты, нам пришлось использовать поддельные часы, введенные в `InMemoryTrips`.

Ранее говорилось, что `clock.instant()` зависит от того, когда мы его вызываем. В наших тестах это не так (по крайней мере не таким же образом). Вместо этого мы могли бы настроить поездки относительно времени выполнения тестов, но это затруднило бы понимание наших тестов и могло привести к сбою, если вы запустите тесты около полуночи.

2015 был концом времени

Дункан и Нэт вернулись к работе после рождественских каникул в начале 2015 года и обнаружили множество ранее пройденных модульных тестов, которые теперь выдают ошибку. Оказывается, 2015-01-01T00:00:00 было воспринято как время, которое всегда будет в будущем. Когда наступил 2015 год, все тесты, которые были связаны с данными до и после этой даты, начали давать сбои.

Они исправили тесты путем рефакторинга, описанного в этой главе.

Необходимо ли вводить часы по принципу «повреждение конструкции, вызванное испытаниями» (test-induced design damage — см. **https://oreil.ly/YZx1T**)? В этом случае — да. Фальшивые часы позволили нам решить проблему тестирования, но за счет усложнения кода. Это также позволило нам избежать переосмысления, которое могло бы привести к...

Улучшенный проект

Как бы в таком случае выглядел улучшенный проект?

Чтобы сделать этот код менее зависящим от времени, мы можем указать время в качестве аргумента метода. Хотя это заставляет *абонента* знать время, для нашего абонента так же легко запросить время, как и для этого метода. Это частный случай того, как проводится рефакторинг, чтобы избежать зависимостей от другого глобального состояния. Вместо того, чтобы считывать значение внутри функции, мы передаем в нее значение.

Функция, в которую мы хотим передать время, переопределяет `Trips.currentTripsFor`, поэтому начнем с добавления к ней параметра `Instant`. Раньше было так (пример 7.7).

Пример 7.7 [actions.0:src/main/java/travelator/Trips.java]

```
public interface Trips {
    ...
    Set<Trip> currentTripsFor(String customerId);
}
```

Мы используем рефакторинг IntelliJ **Изменить сигнатуру** (Change Signature), чтобы добавить параметр с названием `at`. Когда мы добавляем параметр, нам нужно

сообщить IntelliJ, какое значение она должна использовать при обновлении абонентов, вызывающих нашу функцию. Поскольку мы еще не используем значение в методе (а это Java), то должны иметь возможность использовать `null`, ничего не нарушая. Запуск тестов показывает нашу правоту — они все еще проходят успешно. `Trips` теперь выглядит так (пример 7.8).

Пример 7.8 [actions.3:src/main/java/travelator/Trips.java]

```
public interface Trips {
    ...
    Set<Trip> currentTripsFor(String customerId, Instant at);
}
```

Здесь осуществляется вызов метода (пример 7.9).

Пример 7.9 [actions.3:src/main/java/travelator/Tracking.java]

```
class Tracking implements ITrackTrips {
    ...

    @Override
    public Optional<Trip> currentTripFor(String customerId) {
        var candidates = trips.currentTripsFor(customerId, null) ❶
            .stream()
            .filter((trip) -> trip.getBookingStatus() == BOOKED)
            .collect(toList());
        if (candidates.size() == 1)
            return Optional.of(candidates.get(0));
        else if (candidates.size() == 0)
            return Optional.empty();
        else
            throw new IllegalStateException(
                "Unexpectedly more than one current trip for " + customerId
            );
    }
}
```

❶ IntelliJ представила `null` в виде значения аргумента.

Помните, что значение времени в нашем выполнении `Trips` еще не используется. Мы просто пытаемся предоставить его снаружи нашей системы, чтобы преобразовать как можно больше кода в вычисления. `Tracking` не является внешней стороной нашего взаимодействия, поэтому мы выбираем `null Instant` и **Ввести параметр** (Introduce Parameter), чтобы добавить его к сигнатуре `Tracking.currentTripFor` (пример 7.10).

Пример 7.10 [actions.4:src/main/java/travelator/Tracking.java]

```
@Override
public Optional<Trip> currentTripFor(String customerId, Instant at) { ❶
    var candidates = trips.currentTripsFor(customerId, at) ❶
        .stream()
        .filter((trip) -> trip.getBookingStatus() == BOOKED)
        .collect(toList());
        ...
}
```

❶ Новый параметр `Instant`.

Когда мы применяем **Ввести параметр**, IntelliJ перемещает выражение (в нашем случае `null`) из тела метода в абонента, поэтому `CurrentTripsHandler` все еще компилируется (пример 7.11).

Пример 7.11 [actions.4:src/main/java/travelator/handlers/CurrentTripsHandler.java]

```
public Response handle(Request request) {
    try {
        var customerId = request.getQueryParam("customerId").stream()
            .findFirst();
        if (customerId.isEmpty())
            return new Response(HTTP_BAD_REQUEST);
        var currentTrip = tracking.currentTripFor(customerId.get(), null); ❶
        return currentTrip.isPresent() ?
            new Response(HTTP_OK,
                objectMapper.writeValueAsString(currentTrip)) :
            new Response(HTTP_NOT_FOUND);
    } catch (Exception x) {
        return new Response(HTTP_INTERNAL_ERROR);
    }
}
```

❶ Значение аргумента `null`.

Аналогично исправлен `nullTrackingTests` (пример 7.12).

Пример 7.12 [actions.4:src/test/java/travelator/TrackingTests.java]

```
@Test
public void returns_empty_when_no_trip_planned_to_happen_now() {
    clock.now = anInstant();
    assertEquals(
        Optional.empty(),
```

```
        tracking.currentTripFor("cust1", null) ❶
    );
}

@Test
public void returns_single_active_booked_trip() {
    var diwaliTrip = givenATrip("cust1", "Diwali",
        "2020-11-13", "2020-11-15", BOOKED);
    givenATrip("cust1", "Christmas",
        "2020-12-24", "2020-11-26", BOOKED);

    clock.now = diwaliTrip.getPlannedStartTime().toInstant();
    assertEquals(
        Optional.of(diwaliTrip),
        tracking.currentTripFor("cust1", null) ❶
    );
}
```

❶ Значение аргумента `null`.

На этом этапе все компилируется и проходит тесты, но на самом деле мы не используем время (`null`), которое передаем из нашего обработчика.

Давайте исправим это в `InMemoryTrips`, с чего мы и начинали. У нас действительно было так (пример 7.13).

Пример 7.13 [actions.4:src/test/java/travelator/InMemoryTrips.java]

```
public class InMemoryTrips implements Trips {

    ...
    @Override
    public Set<Trip> currentTripsFor(String customerId, Instant at) {
        var now = clock.instant();
        return tripsFor(customerId).stream()
            .filter(trip -> trip.isPlannedToBeActiveAt(now))
            .collect(toSet());
    }
}
```

Теперь, когда у нас есть время в качестве параметра, мы можем использовать его вместо того, чтобы задавать `clock` (пример 7.14).

Пример 7.14 [actions.5:src/test/java/travelator/InMemoryTrips.java]

```
public class InMemoryTrips implements Trips {

    ...
    @Override
```

```
    public Set<Trip> currentTripsFor(String customerId, Instant at) {
        return tripsFor(customerId).stream()
            .filter(trip -> trip.isPlannedToBeActiveAt(at))
            .collect(toSet());
    }
}
```

Поскольку метод применяет значение параметра, это приводит к сбою в тестах, использующих `InMemoryTrips` с `NullPointerException`, и тесты проходят с результатом `null` (пример 7.15).

Пример 7.15 [actions.5:src/test/java/travelator/TrackingTests.java]

```
@Test
public void returns_empty_when_no_trip_planned_to_happen_now() {
    clock.now = anInstant();
    assertEquals(
        Optional.empty(),
        tracking.currentTripFor("cust1", null) ❶
    );
}

@Test
public void returns_single_active_booked_trip() {
    var diwaliTrip = givenATrip("cust1", "Diwali",
        "2020-11-13", "2020-11-15", BOOKED);
    givenATrip("cust1", "Christmas",
        "2020-12-24", "2020-11-26", BOOKED);

    clock.now = diwaliTrip.getPlannedStartTime().toInstant();
    assertEquals(
        Optional.of(diwaliTrip),
        tracking.currentTripFor("cust1", null) ❶
    );
}
```

❶ Эти `nulls` теперь разыменовываются внутри `InMemoryTrips`.

Вместо `null` нам нужно передать значение, которое тесты задавали в `clock`. Хитрый рефакторинг заключается в замене значений `null` на `clock.now` (пример 7.16).

Пример 7.16 [actions.6:src/test/java/travelator/TrackingTests.java]

```
@Test
public void returns_empty_when_no_trip_planned_to_happen_now() {
    clock.now = anInstant();
```

```
    assertEquals(
        Optional.empty(),
        tracking.currentTripFor("cust1", clock.now)
    );
}

@Test
public void returns_single_active_booked_trip() {
    var diwaliTrip = givenATrip("cust1", "Diwali",
        "2020-11-13", "2020-11-15", BOOKED);
    givenATrip("cust1", "Christmas",
        "2020-12-24", "2020-11-26", BOOKED);

    clock.now = diwaliTrip.getPlannedStartTime().toInstant();
    assertEquals(
        Optional.of(diwaliTrip),
        tracking.currentTripFor("cust1", clock.now)
    );
}
```

Это позволяет нашим тестам пройти, потому что теперь мы передаем правильное время в качестве аргумента, хотя и с помощью настройки и немедленного чтения поля в `StoppedClock`. Чтобы исправить это, мы заменяем считывание `clock.now` значениями из записи `clock.now`. Теперь `clock` не используется, и мы можем удалить его (пример 7.17).

Пример 7.17 [actions.8:src/test/java/travelator/TrackingTests.java]

```
public class TrackingTests {

    final InMemoryTrips trips = new InMemoryTrips();
    final Tracking tracking = new Tracking(trips);

    @Test
    public void returns_empty_when_no_trip_planned_to_happen_now() {
        assertEquals(
            Optional.empty(),
            tracking.currentTripFor("cust1", anInstant())
        );
    }

    @Test
    public void returns_single_active_booked_trip() {
        var diwaliTrip = givenATrip("cust1", "Diwali",
            "2020-11-13", "2020-11-15", BOOKED);
        givenATrip("cust1", "Christmas",
            "2020-12-24", "2020-11-26", BOOKED);
```

```
        assertEquals(
            Optional.of(diwaliTrip),
            tracking.currentTripFor("cust1",
                diwaliTrip.getPlannedStartTime().toInstant())
        );
    }

    ...
}
```

Это шаблон, который мы часто видим при рефакторинге в сторону более функционального кода. По мере того как мы сокращаем объем действий, тесты становятся проще, потому что они могут выражать больше различий в параметрах, чем в состоянии тестирования. Мы снова вернемся к этому в *главе 17*.

Конец игры

Мы уже почти закончили (рефакторинг никогда не может быть завершен полностью).

Уделив столько внимания тестам, мы уже собирались зарегистрироваться, прежде чем поняли, что не завершили наш рефакторинг в `CurrentTripsHandler` (пример 7.18).

Пример 7.18 [actions.8:src/main/java/travelator/handlers/CurrentTripsHandler.java]

```
public Response handle(Request request) {
    try {
        var customerId = request.getQueryParam("customerId").stream()
            .findFirst();
        if (customerId.isEmpty())
            return new Response(HTTP_BAD_REQUEST);
        var currentTrip = tracking.currentTripFor(customerId.get(), null); ❶
        return currentTrip.isPresent() ?
            new Response(HTTP_OK,
                objectMapper.writeValueAsString(currentTrip)) :
            new Response(HTTP_NOT_FOUND);
    } catch (Exception x) {
        return new Response(HTTP_INTERNAL_ERROR);
    }
}
```

❶ Мы все еще получаем `null`.

Теперь, когда ни один из наших методов `currentTripFor` не получает значение времени, `CurrentTripHandler` является единственным действием — местом, в котором нам нужно вызвать `Instant.now()`. Мы можем все исправить, вставив вызов и получив следующее (пример 7.19).

Пример 7.19
[actions.9:src/main/java/travelator/handlers/CurrentTripsHandler.java]

```
public class CurrentTripsHandler {
    private final ITrackTrips tracking;
    private final ObjectMapper objectMapper = new ObjectMapper();

    public CurrentTripsHandler(ITrackTrips tracking) {
        this.tracking = tracking;
    }

    public Response handle(Request request) {
        try {
            var customerId = request.getQueryParam("customerId").stream()
                .findFirst();
            if (customerId.isEmpty())
                return new Response(HTTP_BAD_REQUEST);
            var currentTrip = tracking.currentTripFor(
                customerId.get(),
                Instant.now() ❶
            );
            return currentTrip.isPresent() ?
                new Response(HTTP_OK,
                objectMapper.writeValueAsString(currentTrip)) :
                new Response(HTTP_NOT_FOUND);
        } catch (Exception x) {
            return new Response(HTTP_INTERNAL_ERROR);
        }
    }
}
```

❶ Теперь наше действие находится в точке входа в приложение.

Просматривая наш код, мы с ужасом обнаруживаем, что у нас нет никаких модульных тестов для обработчика. Если мы хотим их добавить, то в тот уровень, на котором мы бы сейчас внедрили `Clock`, а не в отдельные сервисы. Моки (имитации) или стабы (заглушки) позволяют нам тестировать действия, но редко требуются для тестирования вычислений.

Мы не будем показывать это, но мы также должны рассмотреть производственную имплементацию `Trips` — ту, которая считывает данные из нашей базы данных. Нам повезло, и мы обнаружили, что она передает текущее время в свой SQL-запрос, так что теперь вместо нее можно просто использовать значение параметра `at` в `Trips.currentTripsFor(String customerId, Instant at)`.

Это было бы сложнее, если бы SQL-запрос использовал текущее время сервера базы данных из специфичного для базы данных выражения — такого как `CURRENT_TIMESTAMP` или `NOW`. Как и в случае с кодом, отличным от SQL, такие действия настолько распространены, что могут считаться обычной практикой, хотя это

усложняет тестирование, а сам код получается менее универсальным. Если бы в нашем запросе использовалось время базы данных, нам пришлось бы переписать его, чтобы получить время из нашей функции в качестве параметра, и сделать мысленную заметку, чтобы не повторять ту же ошибку снова.

Сделав все это, мы просматриваем наши изменения и обнаруживаем, что не преобразовали никакой код в Kotlin!

Это очень существенно. Такой способ размышлений о вычислениях и действиях не зависит от нашего языка имплементации, и со временем сущность Java становится все более функциональной. Однако мы обнаруживаем, что более естественная поддержка Kotlin немутирующих данных и других функциональных конструкций означает, что затраты на реализацию различий будут ниже, и поэтому соотношение затрат и выгод выглядит более благоприятным. Также обратите внимание, что многие шаги по рефакторингу, предпринятые в этой главе (и других), безопасны только потому, что премещения происходят вокруг вызова вычислений, а не действий.

Прежде чем закончить эту главу, вспомним про другое действие, на которое мы намекали. Вот реализация `InMemoryTrips`, преобразованная в Kotlin (пример 7.20).

Пример 7.20 [actions.10:src/test/java/travelator/InMemoryTrips.kt]

```
class InMemoryTrips : Trips {
    private val trips: MutableMap<String, MutableSet<Trip>> = mutableMapOf()

    fun addTrip(trip: Trip) {
        val existingTrips = trips.getOrDefault(trip.customerId, mutableSetOf())
        existingTrips.add(trip)
        trips[trip.customerId] = existingTrips
    }

    override fun tripsFor(customerId: String) =
        trips.getOrDefault(customerId, emptySet<Trip>())

    override fun currentTripsFor(customerId: String, at: Instant): Set<Trip> =
        tripsFor(customerId)
            .filter { it.isPlannedToBeActiveAt(at) }
            .toSet()
}
```

Функция `MutableMap` из сетов `MutableSet` представляет собой признак того, что со временем что-то может измениться. Если у них один и тот же клиент, результат `tripsFor` будет другим после того, как мы вызовем `addTrip`. Таким образом, `tripsFor` является действием, а не вычислением. Если `tripsFor` — это действие, то и все, что его вызывает, является действием, включая наш `currentTripsFor`. То же самое, очевидно, относится и к рабочей версии `Trips`, которая считывает и записывает данные в базу данных. Проделав всю эту работу мы фактически не продвинули наше действие в сторону расчета...

Должны ли мы падать духом? Нет. Несмотря на наше предыдущее утверждение о том, что функции являются вычислениями *или* действиями, истина заключается в том, что на практике действие является постепенным, и действия могут быть более или менее восприимчивыми ко времени. В этом случае, если другой код *в этом взаимодействии* также не будет извлекать поездки для клиента и не обнаружит несоответствие, мы можем рассматривать данные `Trips` как фактически неизменяемые. То есть `tripsFor` и, как следствие, `currentTrips For`, фактически являются вычислениями. В этом отношении класс `InMemoryTrips` более опасен, чем имплементация базы данных, поскольку при обращении в несколько потоков он может преобразоваться в коллекции, возвращаемые функцией `tripsFor`. Это приводит к потенциальным исключениям `ConcurrentModification Exceptions` в осуществлении `filter`. Классификация нашего кода по вычислениям и действиям помогла нам увидеть эти проблемы и дала нам основу для принятия решения о том, важны ли они в контексте.

Наконец, обратите внимание, что предпочтение Kotlin немутирующих данных облегчает эту классификацию. Например, когда вы видите `List` в Java, вы должны найти места, в которых он создан или на которые ссылается, чтобы установить его изменяемость и, следовательно, вероятность того, что код, обращающийся к этому списку, может быть действием. В Kotlin, когда вы видите `MutableList`, вы можете вывести действие, хотя, как показано на примере `InMemoryTrips`, предоставление мутирующей коллекции с псевдонимом, доступным только для чтения, может привести к действиям, притворяющимся вычислениями.

Двигаемся дальше

Категоризация кода на вычисления и действия (наряду с данными) — это формализм, представленный Эриком Нормандом в его книге «Grokking Simplicity: Taming Complex Software with Functional Thinking» (Manning Publications). Как разработчики, мы интуитивно чувствуем разницу и вскоре учимся полагаться на свою интуицию, но часто не понимаем, как и почему. Присвоение названий категориям и изучение их качеств позволяет нам рассуждать на более сознательном и эффективном уровне.

В *главе 5 «От объектов JavaBeans к значениям»* мы перешли от мутирующего компонента к немутирующему значению. Аналогично, в *главе 6 «От коллекций Java к коллекциям Kotlin»* мы перешли от мутирующих коллекций к немутирующим. В обоих случаях мы обмениваем способность объекта к изменению на возврат скорректированной копии, преобразуя действие в вычисление. Поступая таким образом, мы получаем преимущества, которые рассмотрели в этой главе: лучшее понимание, более простое тестирование и предсказуемый рефакторинг. Чем больше в нашем коде вычислений, тем нам лучше.

Мы вернемся к теме перехода от действий к вычислениям в *главе 14 «От накопления объектов к преобразованиям»*. А в *главе 15 «От инкапсулированных коллекций к псевдонимам типов»* мы увидим, как немутирующие данные сочетаются с функциями-расширениями Kotlin и псевдонимами типов и позволяют нам организовать наш код способами, невозможными в Java.

ГЛАВА 8

От статических методов к функциям верхнего уровня

Автономные функции являются одним из фундаментальных строительных блоков программного обеспечения. Они должны быть объявлены как методы класса в Java, но в Kotlin можно объявить их как объекты верхнего уровня. Когда мы должны отдать предпочтение функциям верхнего уровня и как можем осуществить рефракторинг из Java?

Модификатор Static в Java

Все значения и функции в программе Java должны принадлежать классу — они являются *членами* этого класса. Java вызывает *поля* значений-членов и *методы* функций-членов. По умолчанию значения полей задаются для каждого экземпляра класса — разные экземпляры имеют разные значения. Методы также являются индивидуальными для каждого экземпляра в том смысле, что они имеют доступ к состоянию экземпляра, в котором они вызываются. Однако, если пометить поля как `static`, они будут общими для всех экземпляров класса. Статические методы имеют доступ только к этому общему состоянию (и видимым статическим полям в других классах), но в обмен на это ограничение мы можем вызывать их, не требуя экземпляра класса.

Чтобы упростить Java, разработчики языка привязали весь код и данные к классам. У нас есть статическое состояние с классовой областью, поэтому нам нужны статические методы с классовой областью. Они могли бы добавить автономные данные и функции, но предпочли статические поля и методы. Если бы у языка были варианты, то разработчикам пришлось бы выбирать между ними, а меньшее количество возможностей выбора часто бывает полезнее. Затем разработчики перенесли это решение о разработке языка на виртуальную машину Java, которая, в свою очередь, не имеет возможности выразить код или данные верхнего уровня.

Статическое состояние

В ранний период существования Java статическое состояние было намного более распространенным явлением, чем сегодня. Мы писали синглтоны и спорили о том, как их инициализировать ленивым (отложенным), но безопасным способом. Мы использовали статические поля для реализации кешей экземпляров для каждого класса. Затем увлечение тестированием кода в новом тысячелетии практически уничтожило статическое состояние. Это потому, что очень трудно изолировать один тест от другого, когда они связаны по состоянию, и очень трудно разъединить их, когда это состояние статично и поэтому разделяется между всеми тестами в JVM. (Обратите внимание, что состояние здесь относится к мутирующим данным. Немутирующие данные — константы — представляют меньшую проблему.) Итак, мы научи-

лись сохранять статическое состояние в полях объекта и использовали внедрение зависимостей, чтобы в нашем приложении был только один общий экземпляр объекта. (Когда здесь говорится «внедрение зависимостей», то имеется в виду «передача объекта конструктору», а не использование фреймворка.)

В этой главе мы сосредоточимся на коде, а не на состоянии.

Иногда у нас есть класс как с нестатическими, так и со статическими методами, действующими на один и тот же тип — например, класс `email` со статическим парсингом, который рассмотрен в *главе 3*. Однако часто мы получаем класс, состоящий только из статических методов. Когда у них нет общего статического состояния, эти методы фактически являются просто автономными функциями, сгруппированными вместе и вызываемыми по имени их класса, как методы из класса `java.util.Collections`, например:

```
var max = Collections.max(list);
```

Удивительно, но программистское сообщество на самом деле не заметило, какую боль вызывало существование префикса `Collections.`. Дело в том, что мы писали наши программы, добавляя все больше и больше методов к типам, которыми владели, поэтому нам редко требовались статические функции. Статические функции полезны, когда мы хотим добавить функциональность *без* добавления метода к типу, с которым они работают. Это может быть связано с тем, что наши классы уже прогибались под тяжестью всех добавленных нами методов. Или потому, что мы не владеем классом и по этой причине не можем добавить к нему метод. Еще одна причина использования статических функций, а не методов, заключается в том, что функциональность применяется только к некоторым экземплярам обобщенного типа, поэтому она не может быть объявлена как член дженерика. Например, `Collections.max` применяется только к коллекции с сопоставимыми элементами.

Со временем мы начали ценить преимущества использования стандартных интерфейсов — таких как коллекции Java — вместо того, чтобы работать с нашими собственными абстракциями. Java 5 (с ее дженериками) была первым релизом, который позволил нам использовать коллекции напрямую, а не оборачивать их нашими собственными классами. Также не случайно, что Java 5 предоставила доступ к `import static java.util.Collections.max`, чтобы мы могли затем написать:

```
var m = max(list);
```

Обратите внимание, что на самом деле это всего лишь удобство, предоставляемое компилятором, поскольку JVM по-прежнему фактически поддерживает только статические методы, а не истинные функции верхнего уровня.

Функции верхнего уровня в Kotlin, объекты и объекты-компаньоны

Kotlin позволяет объявлять функции (а также свойства и константы) вне классов. В этом случае, поскольку JVM больше ничего поделать не может, компилятор генерирует класс со статическими членами для этих объявлений верхнего уровня.

По умолчанию он выводит имя класса из имени файла, определяющего функции. Например, функции, определенные в `top-level.kt`, заканчиваются как статические методы в классе с названием `Top_levelKt`. Если мы знаем имя класса, то можем ссылаться на него из Java либо статическим импортом `Top_LevelKt.foo`, либо прямым вызовом `Top_levelKt.foo()`. Если нам не нравится уродливое `Top_LevelKt`, мы можем явно присвоить сгенерированному классу имя, добавив аннотацию `@file:JvmName` в начало файла, как будет показано дальше в этой главе.

Помимо этих функций верхнего уровня, Kotlin также позволяет определять свойства и функции, охватываемые, подобно статике Java, классом, а не экземпляром. Вместо того чтобы просто помечать их как `static`, Kotlin заимствует их из Scala и собирает в объявления `object`. Этот тип объявления `object` (в отличие от выражения `object`, которое создает анонимные типы) определяет *синглтон* — тип только с одним экземпляром, который предоставляет глобальную точку доступа к этому экземпляру. Все члены `object` будут скомпилированы в члены класса с именем объекта. Однако на самом деле они не будут статическими методами, если специально не помечены `@JvmStatic`. Это связано с тем, что Kotlin позволяет этим объектам расширять классы и реализовывать интерфейсы, а это несовместимо со статическими объявлениями.

Когда нам нужно сгруппировать статические и нестатические элементы в одном классе, мы можем объявить статические (или соответственно нестатические) части в `companion object` внутри объявления класса. Это группирует их в файле, и код в объекте-компаньоне может получить доступ к закрытому состоянию в экземплярах содержащего его класса. Объекты-компаньоны также могут расширять другой класс и реализовывать интерфейсы — то, чего не может сделать статика Java. Однако по сравнению со статикой Java объекты-компаньоны громоздки, если просто требуется определить один или два статических метода.

Так что в Kotlin мы можем писать функции, не относящиеся к экземпляру: либо как функции верхнего уровня, либо как методы для синглтон-объекта, и этот объект может быть объектом-компаньоном или не иметь области действия для типа.

Из всего этого, при прочих равных условиях, мы должны по умолчанию использовать функции верхнего уровня. Их проще всего объявлять и ссылаться на них, и их можно перемещать из файла в файл внутри пакета, не затрагивая клиентский код Kotlin (с учетом предупреждения, приведенного далее во *врезке «Перемещение функций верхнего уровня»*). Мы оставляем за собой объявление функций в качестве методов для синглтон-объекта вместо функций верхнего уровня для случаев, когда нам нужна возможность реализовать интерфейс или иным образом более тесно сгруппировать функции. И используем объект-компаньон, когда нам нужно смешать статическое и нестатическое поведение внутри класса или написать фабричные методы с именами типа `MyType.of(...)`.

Как и во многих аспектах программирования, мы начинаем с самой простого момента, который работает, — обычно это функция верхнего уровня, и проводим рефакторинг до более сложного решения только тогда, когда оно приносит преимущества, такие как более выразительный API для наших клиентов или лучшая ремонтопригодность для нас.

Рефакторинг от статических методов к функциям верхнего уровня

Хотя мы предпочитаем использовать объявления верхнего уровня, встроенное в IntelliJ преобразование Java в Kotlin этого не делает. Оно преобразует нашу статику Java в объектные методы. Давайте посмотрим, как выполнить рефакторинг с Java через объявления объектов к функциям верхнего уровня.

В `Travelator` мы позволяем нашим клиентам составлять короткие списки — такие как краткий список маршрутов при планировании поездки или краткий список гостиничных номеров на этих маршрутах. Пользователь может ранжировать элементы в коротком списке по различным критериям и отбрасывать элементы, чтобы уменьшить число результатов до окончательного выбора. Согласно рекомендациям, приведенным во *врезке «Не изменяйте разделяемые коллекции»* (см. *разд. «Коллекции Java» главы 6*), краткий список сохраняется как неизменяемый. Функции для управления коротким списком (возврат измененной копии, а не изменение списка) реализованы как статические методы класса `Shortlists` (пример 8.1).

Пример 8.1 [static-to-top-level.0:src/main/java/travelator/Shortlists.java]

```
public class Shortlists {
    public static <T> List<T> sorted(
        List<T> shortlist,
        Comparator<? super T> ordering
    ) {
        return shortlist.stream()
            .sorted(ordering)
            .collect(toUnmodifiableList());
    }

    public static <T> List<T> removeItemAt(List<T> shortlist, int index) {
        return Stream.concat(
            shortlist.stream().limit(index),
            shortlist.stream().skip(index + 1)
        ).collect(toUnmodifiableList());
    }

    public static Comparator<HasRating> byRating() {
        return comparingDouble(HasRating::getRating).reversed();
    }

    public static Comparator<HasPrice> byPriceLowToHigh() {
        return comparing(HasPrice::getPrice);
    }

    ... и другие компараторы
}
```

Удобные функции

Функции `sorted` и `removeItemAt` слишком сложны для своих задач. Нэт открыл для себя Kotlin, когда увидел, как Дункан пытается использовать Java Streams API для анализа некоторых публикуемых данных. Он был так напуган сложностью выполнения базовых операций, что отправился на поиски языка JVM, который облегчил бы его труд. Создается впечатление, что, хотя API-интерфейсы Java в последнее время улучшились, в течение многих лет разработчики, похоже, придерживались политики никогда сознательно не добавлять в них удобные функции.

Напротив, стандартная библиотека Kotlin, похоже, старается изо всех сил предоставлять функциональность именно там, где и когда она нам нужна, часто в форме функций-расширений (см. *главу 10*) для существующих типов.

Функции в `Shortlists` являются статическими методами, и на них необходимо ссылаться как на таковые. Написанный от руки код выглядит так (пример 8.2).

Пример 8.2 [static-to-top-level.5:src/test/java/travelator/ShortlistsTest.java]

```
var reordered = Shortlists.sorted(items, Shortlists.byValue());
```

Однако мы обычно применяем `static import` к методам, и они названы так, чтобы лучше читались (пример 8.3).

Пример 8.3 [static-to-top-level.5:src/test/java/travelator/ShortlistsTest.java]

```
var reordered = sorted(items, byPriceLowToHigh());
```

Преобразуя Java в Kotlin с помощью IntelliJ, мы получаем следующий код (пример 8.4).

Пример 8.4 [static-to-top-level.5:src/main/java/travelator/Shortlists.kt]

```
object Shortlists {
    @JvmStatic
    fun <T> sorted(shortlist: List<T>, ordering: Comparator<in T>): List<T> {
        return shortlist.stream().sorted(ordering)
            .collect(toUnmodifiableList())
    }

    @JvmStatic
    fun <T> removeItemAt(shortlist: List<T>, index: Int): List<T> {
        return Stream.concat(
            shortlist.stream().limit(index.toLong()),
            shortlist.stream().skip((index + 1).toLong())
        ).collect(toUnmodifiableList())
    }
```

```
    @JvmStatic
    fun byRating(): Comparator<HasRating> {
        return comparingDouble(HasRating::rating).reversed()
    }

    @JvmStatic
    fun byPriceLowToHigh(): Comparator<HasPrice> {
        return comparing(HasPrice::price)
    }

    ... и другие компараторы
}
```

На самом деле это не совсем так. На момент подготовки книги конвертер добавил некоторую ложную возможность обнуления к типам, отменил статический импорт (оставив нас с `Collectors.to UnmodifiableList()`, например) и создал список импорта, который не компилировался. Исправление файла вручную дает нам уверенность в том, что машины не вытеснят нас с наших рабочих мест еще несколько лет.

В *главе 3* показано, что преобразование класса Java со статическими и нестатическими методами привело к созданию класса Kotlin с объектом-компаньоном. Здесь преобразование привело к созданию только объекта верхнего уровня. Поскольку в этом классе Java не было нестатических методов или состояний, нет необходимости в том, чтобы перевод на Kotlin включал создаваемый класс. Классы со статическими и нестатическими методами менее подходят для преобразования в функции верхнего уровня.

Хотя преобразование не прошло на уровне Kotlin полностью гладко, есть в этом и положительная сторона — в процессе не пострадал ни один фрагмент Java-кода. Клиентский код остается неизменным, поскольку аннотации `@JvmStatic` позволяют Java-коду видеть в классе `Shortlists` методы как статические — какими они и были до преобразования.

Мы хотим преобразовать методы в функции верхнего уровня, но не можем просто переместить их, потому что Java понимает только методы, а не функции. Если бы эти функции были скомпилированы в методы класса с именем `Shortlists`, Java была бы счастлива, и это задача аннотации `@file:JvmName`, о которой мы упоминали ранее. Можно вручную добавить аннотацию в верхней части файла и удалить область видимости `object` и аннотации `@JvmStatic`, чтобы получить следующий вариант (пример 8.5).

Пример 8.5 [static-to-top-level.6:src/main/java/travelator/Shortlists.kt]

```
@file:JvmName("Shortlists")
package travelator

...
```

```
fun <T> sorted(shortlist: List<T>, ordering: Comparator<in T>): List<T> {
    return shortlist.stream().sorted(ordering)
        .collect(toUnmodifiableList())
}

fun <T> removeItemAt(shortlist: List<T>, index: Int): List<T> {
    return Stream.concat(
        shortlist.stream().limit(index.toLong()),
        shortlist.stream().skip((index + 1).toLong())
    ).collect(toUnmodifiableList())
}

... и т. п.
```

Это приносит счастье Java, но, что раздражает, нарушает некоторый код Kotlin, вызывающий методы. Вот, например, импорт для теста (пример 8.6).

Пример 8.6
[static-to-top-level.6:src/test/java/travelator/hotels/ShortlistScenarioTest.kt]

```
import org.junit.jupiter.api.Test
import travelator.Shortlists.byPriceLowToHigh
import travelator.Shortlists.byRating
import travelator.Shortlists.byRelevance
import travelator.Shortlists.byValue
import travelator.Shortlists.removeItemAt
import travelator.Shortlists.sorted
```

Здесь импортировались статические методы Java, но Kotlin не может импортировать свои собственные функции верхнего уровня таким же способом, поэтому эти строки завершаются ошибкой с `Unresolved reference: Short lists`. Дело в том, что в Kotlin функции определяются в области пакета, а не в классе этого пакета. Компилятор может скомпилировать их в статические методы класса JVM с именем `ShortlistsKt`, но этот класс является частью реализации того, как компилятор сопоставляет концепции языка Kotlin с платформой JVM, и не виден нашему коду Kotlin во время компиляции.

Мы могли бы просмотреть все ошибки компиляции и вручную исправить импорт, чтобы ссылаться на функцию в области пакета. Например, нам пришлось бы изменить `import travelator.Shortlists.sorted` на `import travelator.sorted`. Это достаточно просто, если изменение затрагивает несколько файлов. Но если изменение оказало обширное влияние, исправление всего импорта является утомительной работой, хотя и *может* быть достигнуто с помощью одного глобального поиска и замены.

К счастью, пока мы писали эту книгу, IntelliJ получил рефакторинг **Перейти на верхний уровень** (Move to top level). Давайте отменим последнее изменение Kotlin, вернемся к объявлению объекта и попробуем еще раз.

Перемещение на верхний уровень

Сейчас, когда мы пишем эту главу, рефакторинг еще настолько новый, что недоступен в меню IntelliJ. Но нажатие комбинации клавиш <Alt>+<Enter> на имени метода объекта дает возможность **Перейти на верхний уровень**. Сначала отсортируем всё методом `sorted` — IntelliJ переместит метод из области видимости объекта на уровень файла (пример 8.7).

Пример 8.7 [static-to-top-level.7:src/main/java/travelator/Shortlists.kt]

```
@JvmStatic
fun <T> sorted(shortlist: List<T>, ordering: Comparator<in T>): List<T> {
    return shortlist.stream().sorted(ordering)
        .collect(toUnmodifiableList())
}
```

К сожалению, коду не удалось удалить аннотацию `@JvmStatic`, поэтому мы должны удалить ее сами, чтобы получить код для компиляции. Как только мы это сделаем, то обнаружим, что удаление, по крайней мере, исправило абонентов. Это было проблемой, с которой мы столкнулись, когда только перенесли метод вручную. Там, где мы явно ссылались на метод в Java как `ShortLists.sorted`, теперь получим следующее (пример 8.8).

Пример 8.8 [static-to-top-level.8:src/test/java/travelator/ShortlistsTest.java]

```
var reordered = ShortlistsKt.sorted(items, Shortlists.byValue());
```

По какой-то причине там, где у нас был статический импорт, все стало хуже (пример 8.9).

Пример 8.9 [static-to-top-level.8:src/test/java/travelator/ShortlistsTest.java]

```
var reordered = travelator.ShortlistsKt.sorted(items, byPriceLowToHigh());
```

Мы можем исправить это с помощью комбинации клавиш <Alt>+<Enter> и опции **Добавить статический импорт по требованию**. И нам придется сделать так один раз в каждом затронутом файле. Но надо проверить это перед рефакторингом, чтобы увидеть, какие файлы он изменил (пример 8.10).

Пример 8.10 [static-to-top-level.9:src/test/java/travelator/ShortlistsTest.java]

```
var reordered = sorted(items, byPriceLowToHigh());
```

По сравнению с предыдущим ручным подходом к добавлению аннотации `@file:JvmName("Short lists")` наши Java-клиенты теперь имеют доступ к столь неприглядному имени `ShortlistsKt`. Поскольку имена методов были разработаны для ис-

пользования со статическим импортом, они почти всегда скрыты в блоке импорта, куда никто никогда не заглядывает, поэтому мы готовы смириться с этим. В обмен на эту жертву рефакторинг также исправил абонентов Kotlin из `sorted`. Теперь в Kotlin на него ссылаются как на `travelator.sorted`, а не на `travalator.Shortlists.sorted`, в чем и заключался смысл операции.

Сейчас мы можем переместить остальные методы в `Shortlists` тем же способом. Это немного утомительно, но, утешает лишь тот факт, что, переместив последний метод, IntelliJ удаляет пустой объект, оставляя нам следующий код (пример 8.11).

Пример 8.11 [static-to-top-level.10:src/main/java/travelator/Shortlists.kt]

```
fun <T> sorted(shortlist: List<T>, ordering: Comparator<in T>): List<T> {
    return shortlist.stream().sorted(ordering)
        .collect(toUnmodifiableList())
}

fun <T> removeItemAt(shortlist: List<T>, index: Int): List<T> {
    return Stream.concat(
        shortlist.stream().limit(index.toLong()),
        shortlist.stream().skip((index + 1).toLong())
    ).collect(toUnmodifiableList())
}

fun byRating(): Comparator<HasRating> {
    return comparingDouble(HasRating::rating).reversed()
}

fun byPriceLowToHigh(): Comparator<HasPrice> {
    return comparing(HasPrice::price)
}

... и другие компараторы
```

Когда мы пишем код таким образом, рефакторинг **Перейти на верхний уровень** ограничен одним методом за раз. И если методы зависят друг от друга, это может привести к некоторым проблемам, как будет показано в *главе 10*.

Kotlin'изация

Конечно, мы не переносили наши методы в функции верхнего уровня только ради этого. Ну, во всяком случае, не *только* ради этого. Теперь, когда функции находятся в характерном для Kotlin месте, давайте закончим присущую Kotlin работу.

В *главе 13* представлены рекомендации по преобразованию потоков Java в Kotlin. В случае с `sorted` можно просто использовать функцию-расширение Kotlin `sortedWith` (пример 8.12).

Пример 8.12 [static-to-top-level.11:src/main/java/travelator/Shortlists.kt]

```
fun <T> sorted(shortlist: List<T>, ordering: Comparator<in T>): List<T> {
    return shortlist.sortedWith(ordering)
}
```

Это создает, как показано в примере 8.13, очень логичную функцию-расширение (см. *главу 10*).

Пример 8.13 [static-to-top-level.12:src/main/java/travelator/Shortlists.kt]

```
fun <T> List<T>.sorted(ordering: Comparator<in T>): List<T> {
    return sortedWith(ordering)
}
```

Java по-прежнему вызывает ее как статический метод (пример 8.14).

Пример 8.14 [static-to-top-level.12:src/test/java/travelator/ShortlistsTest.java]

```
var reordered = sorted(items, byPriceLowToHigh());
```

Вызов из Kotlin тоже читается хорошо (пример 8.15).

Пример 8.15 [static-to-top-level.12:src/test/java/travelator/hotels/ShortlistScenarioTest.kt]

```
val hotelsByPrice = hotels.sorted(byPriceLowToHigh())
```

Эти варианты использования Kotlin на самом деле ничего не дают нам по сравнению с необработанным API Kotlin, поэтому мы можем просто встроить их (пример 8.16).

Пример 8.16 [static-to-top-level.13:src/test/java/travelator/hotels/ShortlistScenarioTest.kt]

```
val hotelsByPrice = hotels.sortedWith(byPriceLowToHigh())
```

Это оставляет функцию `sorted` для вызова Java. Судя по всему, наш результат действительно больше не имеет ничего общего с краткими списками. Должны ли мы переместить его в обобщенное пространство имен? Может быть, позже. Сейчас просто проследим за остальной частью файла, чтобы получить следующий код (пример 8.17).

Пример 8.17 [static-to-top-level.14:src/main/java/travelator/Shortlists.kt]

```
fun <T> Iterable<T>.sorted(ordering: Comparator<in T>): List<T> =
    sortedWith(ordering)
```

```
fun <T> Iterable<T>.withoutItemAt(index: Int): List<T> =
    take(index) + drop(index + 1)

fun byRating(): Comparator<HasRating> =
    comparingDouble(HasRating::rating).reversed()

fun byPriceLowToHigh(): Comparator<HasPrice> =
    comparing(HasPrice::price)

... и другие компараторы
```

Возможно, вы заметили, что мы переименовали `removeItemAt` в `withoutItemAt`. Такие предлоги, как `with` (с) и `without` (без), являются полезным средством, позволяющим читателю понять, что мы не изменяем объект, а возвращаем копию.

Перемещение функций верхнего уровня

Функция Kotlin `withoutItemAt` выглядит полезной, и мы удивляемся, почему не можем найти ее версию в стандартной библиотеке. Теперь, когда это функция верхнего уровня, мы можем переместить ее из `Shortlists.kt` в другой файл в том же пакете, не изменяя исходный код Kotlin, который ее вызывает. Однако, если мы это сделаем, функция будет определена как статический метод другого класса JVM в скомпилированном выводе.

Если мы опубликуем код библиотеки Kotlin в файле JAR, все будет в порядке при условии, что зависимый код будет перекомпилирован для новой версии. Компилятор Kotlin генерирует файлы классов с разделами метаданных, которые позволяют сопоставлять имена Kotlin с классами и методами JVM. Все может быть *не* так хорошо, если зависимый код использует наш JAR в качестве двоичной зависимости и обновляется без перекомпиляции. JVM не задействует метаданные, сгенерированные компилятором Kotlin. Если функция верхнего уровня переместилась из одного класса в другой между версиями двоичной зависимости, зависимый код может обнаружить ошибку `NoSuchMethodError`, появившуюся во время выполнения.

Мы можем использовать аннотации `@JvmMultifileClass` и `@JvmName` для управления отображением наших объявлений верхнего уровня в файлах классов JVM, чтобы перемещение объявлений между исходными файлами не нарушало двоичную совместимость. Однако ни один из тестов или примеров кода Kotlin не изменился, поэтому мы не можем полагаться на то, что получим предупреждение о такого рода поломках. Нам просто нужно проявлять особую осторожность, когда мы перемещаем функции верхнего уровня, опубликованные в библиотеке, или используем инструменты для проверки двоичной совместимости.

Двигаемся дальше

Статические функции — это хлеб с маслом наших программ. Они должны быть объявлены как методы класса в Java, но в Kotlin мы можем объявить их как объекты верхнего уровня.

Автоматическое преобразование класса статических методов Java в Kotlin создаст объявление `object`, доступное как из Java, так и из Kotlin. Мы можем переместить методы объекта на верхний уровень по отдельности, оставляя их доступными для обоих языков, прежде чем применять другие методы рефакторинга, чтобы воспользоваться преимуществами Kotlin.

Наиболее вероятным следующим шагом в рефакторинге является преобразование наших функций верхнего уровня в функции-расширения. Этот вопрос является предметом *главы 10 «От функций к функциям-расширениям»*.

ГЛАВА 9

От многострочных функций к однострочным

И Нэт, и Дункан любят определение однострочных функций в Kotlin. Когда мы должны использовать эту форму, почему мы можем предпочесть ее и какие функции Kotlin мы можем задействовать, чтобы сделать больше функций однострочными выражениями?

Как и в случае с Java, код в функции Kotlin обычно {помещен внутри фигурных скобок} и использует `return` чтобы определить результат функции (если это не `Unit` — псевдоним Kotlin для `void`):

```
fun add(a: Int, b: Int): Int {
    return a + b
}
```

Однако, если верхний уровень кода представляет собой однострочное выражение, можно при желании удалить тип результата и определить код с помощью выражения после знака равенства:

```
fun addToo(a: Int, b: Int): Int = a + b
```

Можно прочесть это так: результат функции `add` равен `a + b`. Такой подход имеет смысл для однострочного выражения, которое может быть легко прочитано, когда само состоит из подвыражений:

```
fun max(a: Int, b: Int): Int =
    when {
        a > b -> a
        else -> b
    }
```

Но такая интерпретация имеет меньше смысла для функций, имеющих побочные эффекты, особенно для тех, которые выполняют ввод/вывод или запись в мутирующее состояние. Рассмотрим пример:

```
fun printTwice(s: String): Unit = println("$s\n$s")
```

Мы не можем прочесть это так: результат `printTwice` равен `println(..)`, потому что у `println` нет результата, или, по крайней мере, функция его не возвращает. Суть этой функции полностью заключается в побочных эффектах, как было показано в *главе 7*.

Сохранение однострочных функций для вычислений

Если принять соглашение о резервировании однострочных функций для вычислений (см. *разд. «Вычисления» главы 7*), то у нас будет способ сообщить о наших намерениях, когда мы их используем. И когда мы встретим однострочную функцию, то будем знать, что это не действие (см. *разд. «Действия» главы 7*), и поэтому гораздо безопаснее проводить рефакторинг.

На практике это означает, что однострочные функции не должны возвращать `Unit` или выполнять чтение или запись из мутирующего состояния, включая выполнение операций ввода/вывода.

Ваши авторы обнаружили, что попытка объединить как можно больше функций в однострочные выражения улучшает качество кода. Во-первых, если мы зарезервируем форму однострочных функций (single expression) для вычислений, то это уменьшит долю нашего кода, который является действиями, что облегчит его понимание и модификацию. Однострочные выражения также будут иметь тенденцию к укорочению, по сравнению с их альтернативой, что ограничивает сложность каждой функции. Когда функция становится настолько большой, что в ней оказывается трудно разобраться, стиль с однострочными выражениями позволяет легче проводить рефакторинг, повышающий ее понятность, поскольку существует меньший риск нарушения логики, зависящей от побочных эффектов и порядка выполнения действий.

Мы также предпочитаем выражения, а не инструкции. Выражения являются *объявляемыми* — мы объявляем, *что* хотим, чтобы функция вычисляла, и позволяем компилятору Kotlin и среде выполнения решать, *как* осуществлять это вычисление. Нам не нужно запускать код в воображаемом компьютере в наших головах, чтобы понять, что делает функция.

Вспомним, что в конце *главы 3* мы остались для `EmailAddress` с таким кодом (пример 9.1).

Пример 9.1 [single-expressions.0:src/main/java/travelator/EmailAddress.kt]

```
data class EmailAddress(
    val localPart: String,
    val domain: String
) {

    override fun toString() = "$localPart@$domain"

    companion object {
        @JvmStatic
        fun parse(value: String): EmailAddress {
            val atIndex = value.lastIndexOf('@')
            require(!(atIndex < 1 || atIndex == value.length - 1)) {
                "EmailAddress must be two parts separated by @"
            }
```

```
            return EmailAddress(
                value.substring(0, atIndex),
                value.substring(atIndex + 1)
            )
        }
    }
}
```

Метод `toString` уже является отличным простым однострочным выражением. Однако, как мы отмечали тогда, объем кода, требуемый в методе `parse`, сам по себе вредоносен, поскольку необходимо объявлять статические методы в объекте-компаньоне. Может быть, помогло бы решение сосредоточиться на превращении функции в простое однострочное выражение?

Прежде чем продолжить, непременно стоит пояснить, что многие последовательности рефакторинга, представленные в этой книге, имеют смысл «Вот одна из тех, которые мы подготовили заранее». То есть мы показываем вам успешные попытки. Реальные операции рефакторинга — такие как написание кода с нуля — не похожи на них. Мы пробуем то, что вообще не работает, или выбираем гораздо более окольный путь, чем показываем вам в финальной версии. Пример с `EmailAddress` — небольшой, и мы воспользовались им чтобы показать, что на самом деле происходило, когда мы пытались преобразовать `parse` в однострочное выражение. Мы считаем, что из этого «хождения по мукам» можно извлечь ценные уроки, но, если вы просто хотите побыстрее добраться до финиша, вам следует перейти к *разд. «Попытка № 4: Отступление»*.

Попытка № 1: Замена встроенным выражением

Давайте проанализируем код из примера 9.1 и посмотрим, что мешает нашей функции быть хорошим однострочным выражением (пример 9.2).

Пример 9.2 [single-expressions.1:src/main/java/travelator/EmailAddress.kt]

```
fun parse(value: String): EmailAddress {
    val atIndex = value.lastIndexOf('@') ❶
    require(!(atIndex < 1 || atIndex == value.length - 1)) { ❷
        "EmailAddress must be two parts separated by @"
    }
    return EmailAddress( ❸
        value.substring(0, atIndex),
        value.substring(atIndex + 1)
    )
}
```

❶ Присвоение `atIndex` — это инструкция.

❷ Вызов `require` — это инструкция.

❸ Создание `EmailAddress` — это однострочное выражение, зависящее от `value` и `atIndex`.

Первая инструкция — это присвоение `atIndex`. В языке Kotlin присвоение является инструкцией, а не выражением (в отличие от Java, где мы можем соединять назначения в цепочку). Ее положение в коде также имеет значение — она должна находиться в этом месте кода, чтобы значение `atIndex` было доступно для компиляции остальной части функции. Однако выражение, привязанное к переменной `value.lastIndexOf(Char)`, является вычислением. Это означает, что оно всегда будет возвращать один и тот же результат для одних и тех же аргументов (`this` считается аргументом, когда мы вызываем методы). В результате мы можем встроить переменную `atIndex` без изменения результата функции, получая следующий вариант (пример 9.3).

Пример 9.3 [single-expressions.2:src/main/java/travelator/EmailAddress.kt]

```
fun parse(value: String): EmailAddress {
    require(!(
        value.lastIndexOf('@') < 1 ||
            value.lastIndexOf('@') == value.length - 1)) {
        "EmailAddress must be two parts separated by @"
    }
    return EmailAddress(
        value.substring(0, value.lastIndexOf('@')),
        value.substring(value.lastIndexOf('@') + 1)
    )
}
```

Эта версия не будет выдавать тот же байт-код и не будет работать так быстро (общеизвестно, как трудно угадать хот-спот), но она вернет тот же результат. Однако нам все еще нужно разобраться с вызовом `require`, и, похоже, мы уже сделали все немного менее понятным, так что давайте отменим изменение и попробуем другой подход.

Попытка № 2: Ввести функцию

Другой способ удалить оператор присваивания — это выделить область видимости, в которой всегда определяется `atIndex`. Мы могли бы использовать функцию в качестве такой области видимости, потому что функция привязывает единственную оценку своих аргументов к своим параметрам. Мы можем увидеть это, выбрав все, кроме кода, перед назначением и извлекая функцию `emailAddress` (пример 9.4).

Пример 9.4 [single-expressions.3:src/main/java/travelator/EmailAddress.kt]

```
fun parse(value: String): EmailAddress {
    val atIndex = value.lastIndexOf('@')
    return emailAddress(value, atIndex)
}

private fun emailAddress(value: String, atIndex: Int): EmailAddress {
    require(!(atIndex < 1 || atIndex == value.length - 1)) {
        "EmailAddress must be two parts separated by @"
    }
    return EmailAddress(
        value.substring(0, atIndex),
        value.substring(atIndex + 1)
    )
}
```

Теперь мы можем заменить встроенным выражением переменную `atIndex` в `parse`, потому что параметр `atIndex` уловил ее значение для нас (пример 9.5).

Пример 9.5 [single-expressions.4:src/main/java/travelator/EmailAddress.kt]

```
fun parse(value: String): EmailAddress {
    return emailAddress(value, value.lastIndexOf('@'))
}

private fun emailAddress(value: String, atIndex: Int): EmailAddress {
    require(!(atIndex < 1 || atIndex == value.length - 1)) {
        "EmailAddress must be two parts separated by @"
    }
    return EmailAddress(
        value.substring(0, atIndex),
        value.substring(atIndex + 1)
    )
}
```

Теперь `parse` является однострочным выражением, но не `emailAddress(...)`. Поэтому мы пока не можем объявить о своей победе. Функция `require` всегда будет вызывать у нас некоторые проблемы, потому что ее задача состоит в том, чтобы не допустить продолжения оценки. Это противоположно выражению, которое должно быть вычислено до значения.

Часто, когда мы заходим в такого рода тупик при рефакторинге, замена встроенным выражением причины проблемы позволит нам увидеть дальнейший путь. Так что давайте заменим `require` встроенным выражением (пока что отложите недоверие — всё будет становиться только хуже, прежде чем станет лучше), как показано в примере 9.6.

Пример 9.6 [single-expressions.5:src/main/java/travelator/EmailAddress.kt]

```
private fun emailAddress(value: String, atIndex: Int): EmailAddress {
    if (!!(atIndex < 1 || atIndex == value.length - 1)) {
        val message = "EmailAddress must be two parts separated by @"
        throw IllegalArgumentException(message.toString())
    }
    return EmailAddress(
        value.substring(0, atIndex),
        value.substring(atIndex + 1)
    )
}
```

Код очень избыточен, и мы можем это устранить. Нажатие комбинации клавиш <Alt>+<Enter> на условии `if` удалит двойное отрицание `!!`, а следующее нажатие этой комбинации — на лишнем `toString` — удалит и его. Это позволит нам заменить `message` встроенным выражением, что приводит к следующему результату: (пример 9.7).

Пример 9.7 [single-expressions.6:src/main/java/travelator/EmailAddress.kt]

```
private fun emailAddress(value: String, atIndex: Int): EmailAddress {
    if ((atIndex < 1 || atIndex == value.length - 1)) {
        throw IllegalArgumentException(
            "EmailAddress must be two parts separated by @"
        )
    }
    return EmailAddress(
        value.substring(0, atIndex),
        value.substring(atIndex + 1)
    )
}
```

Теперь можно ввести `else` для просмотра структуры (пример 9.8).

Пример 9.8 [single-expressions.7:src/main/java/travelator/EmailAddress.kt]

```
private fun emailAddress(value: String, atIndex: Int): EmailAddress {
    if ((atIndex < 1 || atIndex == value.length - 1)) {
        throw IllegalArgumentException(
            "EmailAddress must be two parts separated by @"
        )
    } else {
        return EmailAddress(
            value.substring(0, atIndex),
```

```
            value.substring(atIndex + 1)
        )
    }
}
```

К этому моменту у нас есть функция с двумя инструкциями, выбранными с помощью `if`. Она настолько соблазнительно близка к однострочному выражению, что даже IDE может это почувствовать, — нажмите комбинацию клавиш <Alt>+<Enter> на `if`, и IntelliJ предложит вам **Поднимите return из 'if'** (Lift return out of 'if'), что и показано в примере 9.9.

Пример 9.9 [single-expressions.8:src/main/java/travelator/EmailAddress.kt]

```
private fun emailAddress(value: String, atIndex: Int): EmailAddress {
    return if ((atIndex < 1 || atIndex == value.length - 1)) {
        throw IllegalArgumentException(
            "EmailAddress must be two parts separated by @"
        )
    } else {
        EmailAddress(
            value.substring(0, atIndex),
            value.substring(atIndex + 1)
        )
    }
}
```

Вот оно — наше однострочное выражение. Нажмите комбинацию клавиш <Alt>+<Enter> на `return` и выберите **Преобразовать в тело выражения** (Convert to expression body), что и показано в примере 9.10.

Пример 9.10 [single-expressions.9:src/main/java/travelator/EmailAddress.kt]

```
private fun emailAddress(value: String, atIndex: Int): EmailAddress =
    if ((atIndex < 1 || atIndex == value.length - 1)) {
        throw IllegalArgumentException(
            "EmailAddress must be two parts separated by @"
        )
    } else {
        EmailAddress(
            value.substring(0, atIndex),
            value.substring(atIndex + 1)
        )
}
```

Когда мы определяем функцию как однострочное выражение, `when` часто понятнее, чем `if`. IntelliJ сделает это для нас, если нажать комбинацию клавиш <Alt>+<Enter> на `if`. Здесь мы также удалили ненужные фигурные скобки, заменили встроенным

выражением `message` и, наконец, преобразовали `parse` в однострочное выражение тоже (пример 9.11).

Пример 9.11 [single-expressions.10:src/main/java/travelator/EmailAddress.kt]

```
fun parse(value: String) =
    emailAddress(value, value.lastIndexOf('@'))

private fun emailAddress(value: String, atIndex: Int): EmailAddress =
    when {
        atIndex < 1 || atIndex == value.length - 1 ->
            throw IllegalArgumentException(
                "EmailAddress must be two parts separated by @"
            )
        else -> EmailAddress(
            value.substring(0, atIndex),
            value.substring(atIndex + 1)
        )
    }
```

Перед вами — для сравнения — оригинал (пример 9.12).

Пример 9.12 [single-expressions.11:src/main/java/travelator/EmailAddress.kt

```
fun parse(value: String): EmailAddress {
    val atIndex = value.lastIndexOf('@')
    require(!(atIndex < 1 || atIndex == value.length - 1)) {
        "EmailAddress must be two parts separated by @"
    }
    return EmailAddress(
        value.substring(0, atIndex),
        value.substring(atIndex + 1)
    )
}
```

Мы довольны результатом?

Не совсем. Теперь у нас *больше* кода, и функция `emailAddress`, похоже, не добавляет никакого значения, кроме захвата `atIndex`.

Рефакторинг часто представляет собой процесс исследования. У нас есть в голове цель, но мы не всегда знаем, чем она обернется. По опыту ваших авторов, попытка найти форму однострочной функции часто улучшает код, но мы не можем, смотря вам в глаза, сказать, что это произошло в нашем случае.

Мы могли бы отказаться от этой идеи или двигаться дальше и пытаться достичь цели из текущего положения. Однако вместо этого давайте вернемся и попробуем третий подход, основанный на только что полученном опыте.

Попытка № 3: Блок *Let*

Причина, по которой мы извлекли функцию `emailAddress`, заключалась в том, чтобы предоставить нам область, в которой значение `atIndex` определяется по всему блоку, — вместо того чтобы присваиваться локальной переменной. Там, где у нас есть только одна переменная, которую нужно заменить, блок `let` предоставляет такую возможность без необходимости определять функцию. Мы можем пройти этот путь маленькими шагами, сначала окружив код после присваивания с помощью оператора `let` (пример 9.13).

Пример 9.13 [single-expressions.12:src/main/java/travelator/EmailAddress.kt]

```
fun parse(value: String): EmailAddress {
    val atIndex = value.lastIndexOf('@')
    atIndex.let {
        require(!(atIndex < 1 || atIndex == value.length - 1)) {
            "EmailAddress must be two parts separated by @"
        }
        return EmailAddress(
            value.substring(0, atIndex),
            value.substring(atIndex + 1)
        )
    }
}
```

Теперь мы можем отменить возврат из `let`. К сожалению, IntelliJ на этот раз не предлагает нам помощи (пример 9.14).

Пример 9.14 [single-expressions.13:src/main/java/travelator/EmailAddress.kt]

```
fun parse(value: String): EmailAddress {
    val atIndex = value.lastIndexOf('@')
    return atIndex.let {
        require(!(atIndex < 1 || atIndex == value.length - 1)) {
            "EmailAddress must be two parts separated by @"
        }
        EmailAddress(
            value.substring(0, atIndex),
            value.substring(atIndex + 1)
        )
    }
}
```

В настоящее время `atIndex` в блоке `let` относится к локальному объекту, который мы пытаемся удалить. Если мы добавим лямбда-параметр с тем же именем, он будет привязан к нему (пример 9.15).

Пример 9.15 [single-expressions.14:src/main/java/travelator/EmailAddress.kt]

```
fun parse(value: String): EmailAddress {
    val atIndex = value.lastIndexOf('@')
    return atIndex.let { atIndex -> ❶
        require(!(atIndex < 1 || atIndex == value.length - 1)) {
            "EmailAddress must be two parts separated by @"
        }
        EmailAddress(
            value.substring(0, atIndex),
            value.substring(atIndex + 1)
        )
    }
}
```

❶ `Warning Name shadowed: atIndex` — в этом и заключается суть.

Заменив встроенным выражением переменную `atIndex`, мы получим наше однострочное выражение (пример 9.16).

Пример 9.16 [single-expressions.15:src/main/java/travelator/EmailAddress.kt]

```
fun parse(value: String): EmailAddress {
    return value.lastIndexOf('@').let { atIndex ->
        require(!(atIndex < 1 || atIndex == value.length - 1)) {
            "EmailAddress must be two parts separated by @"
        }
        EmailAddress(
            value.substring(0, atIndex),
            value.substring(atIndex + 1)
        )
    }
}
```

Теперь выполнение команды <Alt>+<Enter> на `return` предлагает **Преобразовать в тело выражения** (Convert to expression body), что и показано в примере 9.17.

Пример 9.17 [single-expressions.16:src/main/java/travelator/EmailAddress.kt]

```
fun parse(value: String): EmailAddress =
    value.lastIndexOf('@').let { atIndex ->
        require(!(atIndex < 1 || atIndex == value.length - 1)) {
            "EmailAddress must be two parts separated by @"
        }
        EmailAddress(
            value.substring(0, atIndex),
```

```
            value.substring(atIndex + 1)
        )
    }
```

Мы достигли точки невозврата! И что — довольны результатом?

Это сделал Дункан, и он очень рад, что добрался сюда после 15 шагов рефакторинга. Приведенный в попытке № 3 пример, безусловно, послужил своей цели — показать некоторые приемы, позволяющие перейти к однострочным функциям. Тем не менее Дункан не верит, что смог привести пример того, что поиск однострочных выражений дает значительную отдачу. Количество кода все еще кажется большим, и ни один фрагмент не выглядит так, как будто он оправдывает свой объем. Можем ли мы улучшить ситуацию, повысив уровень абстракции? Давайте сделаем еще одну попытку.

Попытка № 4: Отступление

Если отойти от механического рефакторинга, то станет понятно, что мы создаем `EmailAddress` из двух непустых строк, разделенных определенным символом, — в нашем случае `@`. Нахождение двух непустых строк, разделенных символом, представляется как концепция более высокого уровня, к которой мы могли бы перейти. Взглянем еще раз на оригинал (пример 9.18).

Пример 9.18 [single-expressions.17:src/main/java/travelator/EmailAddress.kt]

```
fun parse(value: String): EmailAddress {
    val atIndex = value.lastIndexOf('@')
    require(!(atIndex < 1 || atIndex == value.length - 1)) {
        "EmailAddress must be two parts separated by @"
    }
    return EmailAddress(
        value.substring(0, atIndex),
        value.substring(atIndex + 1)
    )
}
```

На этот раз мы сосредоточимся не на `atIndex`, а на вызовах `substring`, и разделим их на переменные (пример 9.19).

Пример 9.19 [single-expressions.18:src/main/java/travelator/EmailAddress.kt]

```
fun parse(value: String): EmailAddress {
    val atIndex = value.lastIndexOf('@')
    require(!(atIndex < 1 || atIndex == value.length - 1)) {
        "EmailAddress must be two parts separated by @"
    }
```

```
    val leftPart = value.substring(0, atIndex)
    val rightPart = value.substring(atIndex + 1)
    return EmailAddress(
        leftPart,
        rightPart
    )
}
```

А теперь еще раз и с чувством, что мы можем извлечь функцию из всего, кроме оператора `return` (пример 9.20).

Пример 9.20 [single-expressions.19:src/main/java/travelator/EmailAddress.kt]

```
fun parse(value: String): EmailAddress {
    val (leftPart, rightPart) = split(value)
    return EmailAddress(
        leftPart,
        rightPart
    )
}

private fun split(value: String): Pair<String, String> {
    val atIndex = value.lastIndexOf('@')
    require(!(atIndex < 1 || atIndex == value.length - 1)) {
        "EmailAddress must be two parts separated by @"
    }
    val leftPart = value.substring(0, atIndex)
    val rightPart = value.substring(atIndex + 1)
    return Pair(leftPart, rightPart)
}
```

IntelliJ оказывается здесь действительно довольно умным, выдавая результат `Pair`, потому что у него есть два возвращаемых значения.

`split` была бы хорошей обобщенной функцией, которую мы могли бы использовать в других местах, если бы она была параметризована с помощью символа. Опция **Ввести параметр** (Introduce Parameter) для `'@'` делает это. Мы также **Преобразуем параметр в получатель** (Convert parameter to receiver) для `value`, если хотим получить небольшую локальную функцию-расширение (пример 9.21).

Пример 9.21 [single-expressions.20:src/main/java/travelator/EmailAddress.kt]

```
fun parse(value: String): EmailAddress {
    val (leftPart, rightPart) = value.split('@')
    return EmailAddress(
        leftPart,
        rightPart
    )
}
```

```
private fun String.split(divider: Char): Pair<String, String> {
    val atIndex = lastIndexOf(divider)
    require(!(atIndex < 1 || atIndex == length - 1)) {
        "EmailAddress must be two parts separated by @"
    }
    val leftPart = substring(0, atIndex)
    val rightPart = substring(atIndex + 1)
    return Pair(leftPart, rightPart)
```

Теперь можно ввести `let`, как мы делали ранее, чтобы получить следующий код (пример 9.22).

Пример 9.22 [single-expressions.21:src/main/java/travelator/EmailAddress.kt]

```
fun parse(value: String): EmailAddress =
    value.split('@').let { (leftPart, rightPart) ->
        EmailAddress(leftPart, rightPart)
    }
```

Это, наконец, та однострочная функция, которая стоит затраченных усилий!

Чтобы завершить попытку, мы можем применить методы из этой главы к `split` и тоже сделать его однострочным выражением. Итак, вот заключительный `EmailAddress.kt` (пример 9.23).

Пример 9.23 [single-expressions.22:src/main/java/travelator/EmailAddress.kt]

```
data class EmailAddress(
    val localPart: String,
    val domain: String
) {

    override fun toString() = "$localPart@$domain"

    companion object {
        @JvmStatic
        fun parse(value: String): EmailAddress =
            value.splitAroundLast('@').let { (leftPart, rightPart) ->
                EmailAddress(leftPart, rightPart)
            }
    }
}

private fun String.splitAroundLast(divider: Char): Pair<String, String> =
    lastIndexOf(divider).let { index ->
        require(index >= 1 && index != length - 1) {
            "string must be two non-empty parts separated by $divider"
        }
```

```
        substring(0, index) to substring(index + 1)
    }
```

Здесь `splitAroundLast` кажется приемлемым именем, которое не противоречит стандарту `String.split` и намекает на то, что обе стороны разделения должны быть непустыми. Такие фрагменты имен, как `Around`, которые редки в обозначениях, должны побудить читателей кода отказаться от своих предположений о том, что делает функция, и просто ее искать.

Несмотря на то что `splitAroundLast` кажется общеприменимой полезной функцией, если мы хотим сделать ее общедоступной, нам, вероятно, следует написать для нее несколько модульных тестов. Тем не менее сегодня мы потратили на все это уже достаточно времени, поэтому мысленно отметим, что у нас есть `String.splitAroundLast` (если он нам когда-нибудь понадобится) и, наконец, зафиксируем изменение.

Что должен возвращать метод `Parse`?

Прежде чем завершить рассмотрение этого примера, обратите внимание, что показанный рефакторинг был бы проще, если бы `parse` не выдавал исключение при ошибке. `throw` — это выражение, которое возвращает `Nothing`. Оно не завершается и поэтому не подходит, когда мы пытаемся разложить его на отдельные выражения. В *главе 19* об этом подробно рассказывается, но, если бы мы написали `EmailAddress` с нуля в Kotlin, мы бы, вероятно, вернули `EmailAddress` из `parse`, со значением `null` при сбое. Это плохо сочетается с Java-клиентами, где система типов не предупреждает о возможности обнуления. Таким образом, мы, вероятно, получили бы два метода парсинга: один для устаревшего кода и один для Kotlin, удалив версию исключения, когда не осталось Java-клиентов. В *главе 12* рассмотрим, как поддерживать соглашения обоих языков во время постепенного перехода с Kotlin на Java.

Двигаемся дальше

Определение наших вычислений как однострочных функций позволяет нам сообщить, что они отличаются от побочных действий. Попытка выразить функции в виде простого однострочного выражения — полезный прием, который может привести к созданию хорошо продуманного, чистого кода. Чтобы получить форму с однострочным выражением, нам обычно приходится разбивать подвыражения на их собственные функции.

Форма с однострочным выражением является декларативной: выражение описывает результат функции в терминах ее параметров, а не действий, которые компьютер должен выполнить для вычисления результата. Разложение подвыражений на их собственные функции побуждает нас задуматься о том, что должны обозначать эти подвыражения, и таким образом помогает нам писать более четкий код. Например, `String.splitAroundLast('@')` лучше описывает, что мы хотим вычислить, чем `emailAddress(value: String, atIndex: Int)`.

На более глубоком уровне эта глава посвящена не только однострочным выражениям. Речь идет о том, как мы можем перестроить наш код, не меняя его поведения.

Множество различных компоновок утверждений и выражений будут вести себя одинаково. Рефакторинг — это искусство поиска лучшего и безопасного достижения цели. Чем больше механизмов мы сможем рассмотреть и чем более безопасные маршруты сможем спланировать, тем больше у нас будет возможностей улучшить наш код.

Рефакторинг не всегда получается с первого, второго или даже третьего раза, когда мы пробуем его провести.

Как разработчики, мы не всегда можем позволить себе роскошь повторять попытки, но чем больше мы практикуемся в совершенствовании нашей ориентации в коде, тем чаще достигаем цели, прежде чем сдаться и двигаться дальше.

ГЛАВА 10

От функций к функциям-расширениям

В Kotlin есть особый вид процедуры, называемый функцией-расширением, которая вызывается как метод, но на самом деле (обычно) является функцией верхнего уровня. Легко преобразовать обычную функцию в функцию-расширение и обратно. Когда мы должны предпочесть одно другому?

Функции и методы

Объектно-ориентированное программирование — это искусство решения проблем путем отправки сообщений объектам. Хотите знать длину `myString`? Запросите ее, отправив запрос `myString.length()`. Хотите вывести строку на экран? Поместите строку в сообщение и попросите другой объект, представляющий консоль, отобразить ее для вас: `System.out.println(myString)`. В классических ОО-языках мы определяем, как объект реагирует на сообщение, определяя методы в классах. Методы привязаны к своему классу и имеют доступ к членам (полям и другим методам), связанным с конкретным экземпляром. Когда мы вызываем метод, среда выполнения организует вызов правильной версии (в зависимости от типа среды выполнения объекта) и предоставляет ей доступ к состоянию экземпляра.

Напротив, в функциональном программировании мы решаем проблемы, вызывая функции со значениями. Мы находим длину `myString`, передавая ее в функцию: `length(myString)`. Выводим на экран с помощью `println(myString)`. А если хотим вывести данные где-то еще, мы сообщаем об этом функции: `println(myString, System.err)`. Функции не определены *для* типа, но *тип* есть у параметров функций и результатов.

У обеих парадигм есть свои плюсы и минусы, но сейчас давайте просто рассмотрим возможность обнаружения и расширяемости:

Перед вами тип `Customer`:

```
data class Customer(
    val id: String,
    val givenName: String,
    val familyName: String
) {
    ...
}
```

Это класс, поэтому мы заранее знаем, что можем отправлять сообщения с запросом id, givenName и familyName. А как насчет других операций? В системе, основанной на классах, нам нужно только прокрутить листинг вниз, чтобы увидеть другое доступное для отправки сообщение:

```
data class Customer(
    val id: String,
    val givenName: String,
    val familyName: String
) {
    val fullName get() = "$givenName $familyName"
}
```

Часто мы даже не доходим до того, чтобы взглянуть на определение. Если у нас есть переменная `val customer:Customer`, то можно написать `customer.`, и наша среда IDE сразу же сообщит, что мы можем запросить `id`, `givenName`, `familyName` или `fullName`. Фактически это автозаполнение во многих отношениях лучше, чем просмотр определения класса, потому что оно также показывает нам другие операции (`equals`, `copy`, и т. д.), которые определены в супертипах или неявны в языке.

В функциональной декомпозиции `fullName` было бы функцией, и, если мы подозреваем, что она существует, нам пришлось бы искать ее в нашей кодовой базе. Тогда это была бы функция, где единственным аргументом является тип `Customer`. На удивление трудно заставить IntelliJ помочь нам. Опция **Найти способы использования** (Find Usages), сгруппированные по типу параметра, справится с такой задачей, но вряд ли этот способ можно считать удобным. На практике мы ожидаем найти определение `Customer` и его основные операции близко друг к другу в исходном коде, возможно, в одном файле или, по крайней мере, в пространстве имен. Поэтому можно было бы перейти туда и самим найти функции там, где мы их ожидали найти, поскольку наши инструменты оказались для этого не очень полезными.

В таком случае мы ставим один балл ООП за возможность *обнаружения*. А как насчет *расширяемости*? Что происходит, когда мы хотим добавить операцию `Customer`? Особенности маркетинга требуют отобразить имя `familyName` для отчета или в других целях в обратном порядке да еще и в верхнем регистре (вы можете заметить, что всякий раз, когда нам нужен простой, но произвольный пример, мы перекладываем ответственность за него на маркетинг). Если у нас есть код, мы можем просто добавить метод:

```
data class Customer(
    val id: String,
    val givenName: String,
    val familyName: String
) {
    val fullName get() = "$givenName $familyName"
    fun nameForMarketing() = "${familyName.uppercase()}, $givenName}"
}
```

Если мы не владеем кодом, то не можем добавить метод, и нам приходится прибегать к функции. В Java у нас может быть коллекция этих статических функций

в классе под названием `Marketing` или `CustomerUtils`. В Kotlin мы можем сделать их функциями верхнего уровня (см. *главу 8*), но принцип тот же:

```
fun nameForMarketing(customer: Customer) =
    "${customer.familyName.uppercase()}, $customer.givenName}"
```

Что ж, это тоже функциональное решение. Так что функциональное решение, возможно, лучше для расширяемости, потому что операции расширения неотличимы от операций (например, `fullName`), предоставленных авторами изначально, тогда как решение ООП заставляет нас искать два разных типа выполнения: методы и функции.

Даже если мы *действительно* владеем кодом для класса `Customer`, мы должны с осторожностью относиться к добавлению таких методов, как `nameForMarketing`. Если класс `Customer` является в нашем приложении фундаментальным доменным классом, от него будет зависеть множество другого кода. Добавление отчета для маркетинга не должно заставлять нас перекомпилировать и повторно тестировать всё, но это произойдет, если мы добавим метод. Так что будет лучше, если мы сохраним `Customer` как можно меньшего размера и неосновные операции запишем в качестве внешних функций. Даже если это означает, что их не так просто будет обнаружить, как методы.

Однако в Kotlin эти функции не обязательно так сложно найти, как мы предполагали, — они могут быть *функциями-расширениями*.

Функции-расширения

Функции-расширения Kotlin выглядят как методы, но на самом деле являются просто функциями. (Как показано в *главе 8*, технически они также являются методами, потому что в JVM весь код должен быть определен как метод. Далее, в *разд. «Функции-расширения в качестве методов»*, мы увидим, что функции-расширения на самом деле также могут быть нестатическими методами другого класса.) Как следует из их названия, функции-расширения дают нам возможность *расширять* операции, доступные для типа. Они делают это, поддерживая интуитивно понятное соглашение о вызове методов с использованием точек, означающих отправку сообщения, что позволяет их обнаруживать одним и тем же способом с помощью комбинации клавиш <Ctrl>+<Space>.

Так мы можем определить функцию-расширение:

```
fun Customer.nameForMarketing() = "${familyName.uppercase()}, $givenName}"
```

И вызвать ее, как если бы это был метод:

```
val s = customer.nameForMarketing()
```

IntelliJ автоматически предложит функции-расширения вместе с фактическими методами, даже если их необходимо импортировать, чтобы включить в область видимости.

Java не так услужлива — она просто рассматривает функцию-расширение как статическую функцию:

```
var s = MarketingStuffKt.nameForMarketing(customer);
```

Здесь `MarketingStuffKt` — это имя класса, содержащего наши объявления верхнего уровня в качестве статических методов (см. *главу 8*).

Интересно, что мы не можем вызвать функцию таким же образом из Kotlin:

```
nameForMarketing(customer) // не компилируется
```

Это приводит к сбою компиляции с ошибкой:

```
Unresolved reference. None of the following candidates is applicable
because of receiver type mismatch: public fun Customer.nameForMarket
ing(): String ....
```

Кстати `receiver` (*получатель* или *приемник*) — это термин, который Kotlin использует для объекта с именем `this` в функции-расширении (или обычном методе), — т. е. это объект, который получает сообщения.

Обратите внимание, что функции-расширения не имеют никакого специального доступа к закрытым членам класса, который они расширяют, у них есть только те же привилегии, что и у обычных функций в их области видимости.

Расширения и типы функций

Хотя мы не можем вызывать функции-расширения как обычные функции в Kotlin, мы можем назначать их обычным ссылкам на функции. Пусть компилируется следующий код:

```
val methodReference: (Customer.() -> String) =
    Customer::fullName
val extensionFunctionReference: (Customer.() -> String) =
    Customer::nameForMarketing
val methodAsFunctionReference: (Customer) -> String =
    methodReference
val extensionAsFunctionReference: (Customer) -> String =
    extensionFunctionReference
```

Мы можем вызвать тогда их, как и предполагалось, следующим образом:

```
customer.methodReference()
customer.extensionFunctionReference()
methodAsFunctionReference(customer)
extensionAsFunctionReference(customer)
```

Мы также можем использовать ссылки *with-receiver* (*с получателем*), как если бы они брали получателя в качестве первого аргумента:

```
methodReference(customer)
extensionFunctionReference(customer)
```

Однако мы не можем вызывать простые ссылки так, как если бы у них был получатель. То есть обе эти строки не компилируются с ошибкой `Unresolved reference`:

```
customer.methodAsFunctionReference()
customer.extensionAsFunctionReference()
```

Свойства-расширения

Kotlin также поддерживает *свойства-расширения*. Как мы увидим в *главе 11*, акцессоры свойств Kotlin на самом деле являются вызовами методов. Точно так же, как функции-расширения являются статическими функциями, которые вызываются как методы, свойства-расширения — это статические функции, которые вызываются как свойства, которые, в свою очередь, являются методами. Свойства-расширения не могут хранить никаких данных, потому что они на самом деле не добавляют поля в свой класс, — их значение может быть только вычислено.

Функцию `nameForMarketing` можно определить в качестве свойства-расширения так:

```
val Customer.nameForMarketing get() = "${familyName.uppercase()}, $givenName}"
```

Вероятно, она *должна быть* свойством (рассмотрим это в *главе 11*).

Большая часть того, что мы должны сказать о функциях-расширениях, относится к свойствам-расширениям, если специально не проводить различие между ними.

Расширения не являются полиморфными

Хотя вызов функции-расширения выглядит как вызов метода, на самом деле это не отправка сообщения объекту. При вызове полиморфных методов Kotlin во время выполнения использует динамический тип получателя для выбора исполняемого метода. В случае же расширений Kotlin задействует во время компиляции статический тип получателя для выбора вызываемой функции.

Если есть необходимость использовать расширения полиморфным способом, обычно этого можно достичь, вызывая полиморфные методы из наших функций-расширений.

Преобразования

До сих пор мы видели функции-расширения, добавляющие операции к типу. Преобразования из одного типа в другой — обычное дело. Приложению `Travelator` необходимо преобразовать данные о клиентах в JSON и XML и обратно. Как мы должны конвертировать данные из `JsonNode` в `Customer`?

Мы могли бы добавить конструктор: `Customer(JsonNode)` знает, как извлекать соответствующие данные. Но нам кажется неправильным засорять наш класс `Customer` зависимостями от определенной библиотеки JSON, а затем синтаксическим анализатором XML и т. д. Тот же аргумент применим к добавлению преобразований в класс `JsonNode`. Даже если бы мы *могли* изменить его код, довольно скоро он стал бы неуправляемым со всеми методами `JsonNode.toMyDo mainType()`.

В Java мы бы написали класс служебных функций вида:

```
static Customer toCustomer(JsonNode node) {
        ...
}
```

Или с предпочтительным соглашением об именовании Нэта и Дункана:

```
static Customer customerFrom(JsonNode node) {
        ...
}
```

Как именовать преобразования?

Функцию преобразования `JsonNode` в `Customer` можно назвать `nodeToCustomer`, `createCustomer`, `toCustomer`, `customerFrom` или `customerFor`. Почему стоит выбрать `customerFrom`? Давайте рассмотрим альтернативы, в которых они вызываются. Вариант `nodeToCustomer` достаточно хорош, но при вызове повторение `node` раздражает:

```
var customer = nodeToCustomer(node)
```

Вариант `createCustomer` тоже хорош, но не отражает связи между `node` и `customer`:

```
var customer = createCustomer(node)
```

Вариант `toCustomer` дает понять, что `node` содержит в себе всё, что требуется для создания записи `Customer`, но плохо звучит на английском:

```
var customer = toCustomer(node)
```

Наше предпочтение отдано `customerFrom` — хорошо звучит и указывает, что мы извлекаем данные для `customer` из `node`:

```
var customer = customerFrom(node)
```

Можно также попробовать благозвучное `customerFor`:

```
var customer = customerFor(node)
```

Но `customerFor` предполагает иную взаимосвязь, чем парсинг. `For` подразумевает операцию поиска: `phoneNumberFor(customer)` или структуру `wheelFor(bicycle)`.

Действительно ли эти различия имеют значение? В большинстве случаев нет, и нам не следует полагаться на тонкости английского языка, когда наши друзья по команде и клиенты могут не быть его носителями. Но есть смысл взять наилучшие из возможных слов. По сравнению с `createCustomer(node)` использование `customerFrom(node)` может помочь читателю понять, что процедура выполняется за один проход, а не за два, или предотвратить неправильное допущение, приводящее к ошибке. Мы можем добиться небольших, но значительных улучшений, оптимизировав то, как наш код читается в контексте.

Вызов преобразований по отдельности не так уж ужасен:

```
var customer = customerFrom(node);
var marketingName = nameForMarketing(customer);
```

Однако, если нам нужно объединить функции, все начинает идти наперекосяк:

```
var marketingLength = nameForMarketing(customerFrom(node)).length();
```

Мы все здесь разработчики и привыкли читать вызовы функций. Поэтому нам легко недооценить когнитивную нагрузку, связанную с поиском самого внутреннего

вызова и поиском выхода с помощью вызовов функций и методов для расчета того, как вычисляется выражение. Не то, что оно вычисляет, а просто порядок, в котором выражение вычисляется. В Kotlin мы можем записать преобразование как расширение для `JsonNode` и наслаждаться успокаивающим потоком слева направо:

```
fun JsonNode.toCustomer(): Customer = ...

val marketingLength = jsonNode.toCustomer().nameForMarketing().length
```

Ну да... Это гораздо более читабельно.

Обнуляемые параметры

Расширения действительно вступают в свои права, когда мы работаем с необязательными типами данных. Отправляя сообщения потенциально `null`-объекту, мы можем использовать оператор безопасного вызова `?.`, о котором шла речь в *главе 4*. Однако это не помогает с параметрами. Чтобы передать обнуляемую ссылку в качестве аргумента функции, которая принимает необнуляемый параметр, мы должны обернуть вызов в условную логику:

```
val customer: Customer? = loggedInCustomer()
val greeting: String? = when (customer) {
    null -> null
    else -> greetingForCustomer(customer)
}
```

Функции определения области применения Kotlin — такие как `let`, `apply` и `also` — могут помочь в этом случае. В частности, `let` преобразует свой получатель в лямбда-параметр:

```
val customer: Customer? = loggedInCustomer()
val greeting: String? = customer?.let { greetingForCustomer(it) }
```

Здесь оператор `?.` обеспечивает то, что `let` вызывается только тогда, когда значением клиента не является `null`. Это подразумевает, что значение лямбда-параметра `it` никогда не является нулевым и может быть передано в функцию в лямбда-теле. Вы можете думать о `?.let` как об операторе безопасного вызова для одиночных аргументов.

Если функция возвращает результат с нулевым значением, и мы должны передать этот результат другой функции, которая ожидает ненулевой параметр, функции определения области приобретут громоздкий вид:

```
val customer: Customer? = loggedInCustomer()

val reminder: String? = customer?.let {
    nextTripForCustomer(it)?.let {
        timeUntilDepartureOfTrip(it, currentTime)?.let {
            durationToUserFriendlyText(it) + " until your next trip!"
        }
    }
}
```

Даже когда мы можем сгладить вложенные в конвейер вызовов `let` проверки нулевых значений, весь этот дополнительный механизм добавляет синтаксический шум и скрывает основную *задачу* кода:

```
val reminder: String? = customer
    ?.let { nextTripForCustomer(it) }
    ?.let { timeUntilDepartureOfTrip(it, currentTime) }
    ?.let { durationToUserFriendlyText(it) }
    ?.let { it + " until your next trip!" }
```

Если мы преобразуем проблемные параметры в получатели функций-расширений, то сможем напрямую связывать вызовы, выводя логику приложения на передний план:

```
val reminder: String? = customer
    ?.nextTrip()
    ?.timeUntilDeparture(currentTime)
    ?.toUserFriendlyText()
    ?.plus(" until your next trip!")
```

Когда Нэт и Дункан впервые приняли Kotlin, они вскоре обнаружили, что расширения и возможность обнуления образуют добродетельный круг. Было проще обрабатывать необязательные данные с помощью функций-расширений, поэтому они извлекали расширения, закрытые для файла, или преобразовывали функции в расширения, где это облегчало написание логики. Они нашли также, что названия этих расширений могут быть более краткими, чем названия эквивалентных функций, не скрывая их задачи. В результате они написали больше расширений, чтобы сделать логику своего приложения лаконичной. Частные расширения оказались полезными в других местах программы, поэтому авторы перенесли их в общие модули, чтобы облегчить к ним доступ. Это упростило использование необязательных данных в других частях приложения, что привело к написанию большего количества расширений, что сделало логику приложения более лаконичной ... и т. д..

Хотя функции-расширения продвигаются как способ расширения сторонних типов, краткие имена, которые они позволяют применять, и возможность обнуления значений в системе типов побуждают нас определять расширения и для наших собственных типов. Часть сути Kotlin заключается в том, как эти функции взаимодействуют, сглаживая трудности на нашем пути.

Обнуляемые получатели

Одно из основных различий между вызовами метода и функции заключается в обработке `null`-ссылок. Если у нас есть ссылка, значение которой равно `null`, мы не можем отправить в нее сообщение, потому что отправлять сообщение некуда, — JVM выдает `Null PointerException` при попытке это делать. С другой стороны, у нас могут быть `null`-параметры. Мы можем не знать, что с ними делать, но они не мешают среде выполнения находить код для вызова.

Поскольку получатель в функции-расширении на самом деле является параметром, он *может* иметь значение `null`. Хотя `anObject.method()` и `anObject.extensionFunction()` выглядят как одинаковые вызовы, метод `method` никогда нельзя вызвать, если `anObject` имеет значение `null`. В то время как функция `extensionFunction` может быть вызвана со значением `null`, если получатель обнуляемый.

Мы могли бы использовать это, чтобы отбросить шаги, которые генерируют в предыдущем конвейере напоминание в расширение для `Trip?`:

```
fun Trip?.reminderAt(currentTime: ZonedDateTime): String? =
    this?.timeUntilDeparture(currentTime)
        ?.toUserFriendlyText()
        ?.plus(" until your next trip!")
```

Обратите внимание, что мы должны использовать оператор безопасного вызова для разыменования `this` внутри расширения. Хотя `this` никогда не принимает значение `null` внутри метода, оно может находиться внутри расширения обнуляемого типа. Значение `null this` может удивить, если вы пришли из Java, где этого никогда не может произойти. Но Kotlin рассматривает `this` для расширений как просто еще один обнуляемый параметр.

Мы можем вызвать эту функцию с обнуляемым значением `Trip` без оператора `?.`:

```
val reminder: String? = customer.nextTrip().reminderAt(currentTime)
```

С другой стороны, мы усложнили понимание потока обнуления в вызывающей функции, потому что, несмотря на проверку типа, он не виден в коде конвейеров, вызывающих расширение.

У `Trip?.reminderAt` есть еще один, более навязчивый недостаток: возвращаемый тип всегда имеет обнуляемое значение `String?`, даже если вызывается с помощью ненулевого `Trip`. В этом случае мы обнаружим, что пишем такой код:

```
val trip: Trip = ...
val reminder: String = trip.reminderAt(currentTime) ?: error("Should never happen")
```

Эта ошибка ожидает своего появления, когда код вокруг нее изменится, потому что мы сделали невозможным обнаружение несовместимых изменений с помощью средства проверки типов.

Не пишите расширения для типов с обнуляемым значением, которые возвращают `null`, если получатель имеет значение `null`. Напишите расширение для ненулевого типа и используйте для него оператор безопасного вызова.

Однако расширения для типов с обнуляемым значением могут быть полезны, когда они возвращают ненулевой результат. Их действие в этом случае определяет путь выхода из области обнуляемых значений обратно в область ненулевых значений, завершая конвейер безопасных вызовов. Например, мы можем заставить расширение `reminderAt` возвращать некоторый значимый текст, даже если у клиента нет следующей поездки:

```
fun Trip?.reminderAt(currentTime: ZonedDateTime): String =
    this?.timeUntilDeparture(currentTime)
        ?.toUserFriendlyText()
        ?.plus(" until your next trip!")
        ?: "Start planning your next trip. The world's your oyster!"
```

Аналогию представляют и две следующие функции-расширения, которые мы, вероятно, должны были привести еще в *главе 4*. Первая определяется для любого типа с обнуляемым значением, но всегда возвращает ненулевой результат:

```
fun <T : Any> T?.asOptional(): Optional<T> = Optional.ofNullable(this)
fun <T : Any> Optional<T>.asNullable(): T? = this.orElse(null)
```

Это аккуратно поднимает тему обобщенных расширений.

Обобщенные расширения

Как и в случае с обычными функциями, расширения могут иметь обобщенные параметры, и всё становится крайне интересным, когда получатель является обобщенным.

Вот полезная функция-расширение, которая по какой-то причине не является частью стандартной библиотеки. Она определяется как расширение для любого типа, включая `null`-ссылки:

```
fun <T> T.printed(): T = this.also(::println)
```

Мы можем использовать это, когда хотим отладить значение выражения на месте. Например, запомните следующую строку:

```
val marketingLength = jsonNode.toCustomer().nameForMarketing().length
```

Если нам понадобится увидеть значение клиента для отладки, то в общем случае нужно будет извлечь переменную:

```
val customer = jsonNode.toCustomer()
println(customer)
val marketingLength = customer.nameForMarketing().length
```

У нас есть функция `printed`, которая выводит значение получателя и возвращает его без изменений, так что мы можем записать:

```
val marketingLength = jsonNode.toCustomer().printed().nameForMarketing().length
```

что гораздо удобнее в поиске и создает меньше помех, пока мы не найдем ее.

Обратите внимание, что даже если бы мы смогли добавить метод в `Any?`, то он не смог бы утверждать, что возвращает тот же тип, что и его получатель. Если бы мы написали:

```
class Any {
    fun printed() = this.also(::println)
}
```

возвращаемый тип был бы `Any`, следовательно, мы не смогли бы вызвать `nameForMarketing()` и т. д. для результата.

Также можно определить функции-расширения для специализированных обобщенных типов — например, `Iterable<Customer>`:

```
fun Iterable<Customer>.familyNames(): Set<String> =
    this.map(Customer::familyName).toSet()
```

Такая функция-расширение применима к любому `Collection<Customer>`, но не к коллекциям других типов. Это позволяет нам использовать коллекции для представления концепций предметной области, а не для определения наших собственных типов, как мы увидим в *главе 15*. Мы также можем извлекать части коллекций конвейеров в именованные операции (см. *разд. «Извлечение части конвейера» главы 13*).

Функции-расширения в качестве методов

Мы обычно определяем функции-расширения как функции верхнего уровня. Однако они могут быть определены *внутри* определения класса. В этом случае они способны получить доступ к членам своего собственного класса и расширить другой тип:

```
class JsonWriter(
    private val objectMapper: ObjectMapper,
) {
    fun Customer.toJson(): JsonNode = objectMapper.valueToTree(this)
}
```

Здесь у `Customer.toJson` есть доступ к двум значениям `this`. Он может ссылаться на получатель функции-расширения `Customer` или экземпляр `JsonWriter` метода. В обычном формате функция выглядит так:

```
fun Customer.toJson(): JsonNode =
    this@JsonWriter.objectMapper.valueToTree(this@toJson)
```

Это не тот метод, который мы должны использовать слишком часто (без помощи IDE может быть трудно интерпретировать, какой получатель применяется), но он может упростить код, разрешая простое чтение функций-расширений слева направо и скрывая детали, которые усложнили бы ситуацию. В частности, это позволяет DSL скрывать детали (например, `ObjectMapper`), которые не должны беспокоить клиентов.

Рефракторинг к функциям-расширениям

Фактическая механика преобразования статического метода в функцию-расширение проста, но мы должны сосредоточиться, чтобы понять, где функция-расширение улучшит ситуацию. Давайте пройдемся по части кода `Travelator` и посмотрим, как это у нас получится.

Умные люди из отдела маркетинга разработали сводную таблицу, которая дает клиентам оценку в зависимости от того, насколько они ценны для компании, — на основе их ожидаемых будущих расходов. Отдел маркетинга постоянно совершенствует алгоритм, поэтому они не хотят, чтобы мы это автоматизировали. Вместо этого они экспортируют разделенный табуляцией файл с данными о клиентах, оценках и расходах, а мы создаем из этого файла сводный отчет. Вот наши тесты (пример 10.1).

Пример 10.1 [extensions.0:src/test/java/travelator/marketing/HighValueCustomersReportTests.java]

```
class HighValueCustomersReportTests {

    @Test
    public void test() throws IOException {
        List<String> input = List.of(
            "ID\tFirstName\tLastName\tScore\tSpend",
            "1\tFred\tFlintstone\t11\t1000.00",
            "4\tBetty\tRubble\t10\t2000.00",
            "2\tBarney\tRubble\t0\t20.00",
            "3\tWilma\tFlintstone\t9\t0.00"
        );
        List<String> expected = List.of(
            "ID\tName\tSpend",
            "4\tRUBBLE, Betty\t2000.00",
            "1\tFLINTSTONE, Fred\t1000.00",
            "\tTOTAL\t3000.00"
        );
        check(input, expected);
    }

    @Test
    public void emptyTest() throws IOException {
        List<String> input = List.of(
            "ID\tFirstName\tLastName\tScore\tSpend"
        );
        List<String> expected = List.of(
            "ID\tName\tSpend",
            "\tTOTAL\t0.00"
        );
        check(input, expected);
    }

    @Test
    public void emptySpendIs0() {
        assertEquals(
            new CustomerData("1", "Fred", "Flintstone", 0, 0D),
```

```
            HighValueCustomersReport.customerDataFrom("1\tFred\tFlintstone\t0")
        );
    }

    private void check(
        List<String> inputLines,
        List<String> expectedLines
    ) throws IOException {
        var output = new StringWriter();
        HighValueCustomersReport.generate(
            new StringReader(String.join("\n", inputLines)),
            output
        );
        assertEquals(String.join("\n", expectedLines), output.toString());
    }
}
```

Мы не сильно углублялись в их выкладки, потому что у маркетологов есть привычка менять свои предпочтения, но, по сути, в отчете должен быть представлен список клиентов, набравших 10 или более баллов и отсортированных по расходам с итоговой строкой. Вот этот код (пример 10.2).

Пример 10.2
[extensions.0:src/main/java/travelator/marketing/HighValueCustomersReport.java]

```
public class HighValueCustomersReport {

    public static void generate(Reader reader, Writer writer) throws IOException {
        List<CustomerData> valuableCustomers = new BufferedReader(reader).lines()
            .skip(1) // заголовок
            .map(line -> customerDataFrom(line))
            .filter(customerData -> customerData.score >= 10)
            .sorted(comparing(customerData -> customerData.score))
            .collect(toList());

        writer.append("ID\tName\tSpend\n");
        for (var customerData: valuableCustomers) {
            writer.append(lineFor(customerData)).append("\n");
        }
        writer.append(summaryFor(valuableCustomers));
    }

    private static String summaryFor(List<CustomerData> valuableCustomers) {
        var total = valuableCustomers.stream()
            .mapToDouble(customerData -> customerData.spend)
            .sum();
        return "\tTOTAL\t" + formatMoney(total);
    }
```

```
    static CustomerData customerDataFrom(String line) {
        var parts = line.split("\t");
        double spend = parts.length == 4 ? 0 :
            Double.parseDouble(parts[4]);
        return new CustomerData(
            parts[0],
            parts[1],
            parts[2],
            Integer.parseInt(parts[3]),
            spend
        );
    }

    private static String lineFor(CustomerData customer) {
        return customer.id + "\t" + marketingNameFor(customer) + "\t" +
            formatMoney(customer.spend);
    }

    private static String formatMoney(double money) {
        return String.format("%#.2f", money);
    }

    private static String marketingNameFor(CustomerData customer) {
        return customer.familyName.toUpperCase() + ", " + customer.givenName;
    }
}
```

Вы можете видеть, что здесь мы имеем уже вполне функциональное (в отличие от объектно-ориентированного) выражение решения. Это упростит преобразование в функции верхнего уровня, а функции верхнего уровня легко преобразовать в функции-расширения.

Но сначала `CustomerData` (пример 10.3).

Пример 10.3 [extensions.0:src/main/java/travelator/marketing/CustomerData.java]

```
public class CustomerData {
    public final String id;
    public final String givenName;
    public final String familyName;
    public final int score;
    public final double spend;

    public CustomerData(
        String id,
        String givenName,
        String familyName,
        int score,
```

```
        double spend
    ) {
        this.id = id;
        this.givenName = givenName;
        this.familyName = familyName;
        this.score = score;
        this.spend = spend;
    }

    ... методы equals и hashcode
}
```

Это попытка представить не всю информацию о клиенте, а только данные, которые нас интересуют в этом отчете, поэтому тот, кто его написал, просто использовал поля (в *главе 11* обсуждается этот компромисс). Я сомневаюсь, что мы (да кто бы ни написал это) вообще стали бы возиться с `equals` и `hashCode`, если бы не тест `emptySpendIs0`. Число `double` для расходов тоже выглядит подозрительно, но пока оно не вызвало у нас никаких проблем, поэтому мы отложим наше недоверие и просто преобразуем всё это в класс данных Kotlin (см. *главу 5*), прежде чем продолжить.

Обычно такое преобразование представляет собой действительно простую работу из-за отличного взаимодействия, но оказывается, что (на момент подготовки книги) конвертер не может поверить, что кто-то опустится до доступа к необработанным полям. Следовательно, он не обновляет код Java, который обращается, например, к `customer Data.score` для вызова `customerData.getScore()` (свойство Kotlin), что приводит к множеству сбоев компиляции. Вместо того, чтобы исправлять их, мы возвращаемся и используем рефакторинг **Инкапсулировать поля** (Encapsulate Fields) для преобразования всех полей и доступа полей в `Customer` к геттерам (пример 10.4).

Пример 10.4 [extensions.1:src/main/java/travelator/marketing/CustomerData.java]

```
public class CustomerData {
    private final String id;
    private final String givenName;
    private final String familyName;
    private final int score;
    private final double spend;

    ...

    public String getId() {
        return id;
    }

    public String getGivenName() {
        return givenName;
    }
    ...
}
```

Рефакторинг также обновил клиентский код для вызова геттеров (пример 10.5).

Пример 10.5
[extensions.1:src/main/java/travelator/marketing/HighValueCustomersReport.java]

```
private static String lineFor(CustomerData customer) {
    return customer.getId() + "\t" + marketingNameFor(customer) + "\t" +
        formatMoney(customer.getSpend());
}
```

Геттеры позволяют нам преобразовывать `CustomerData` в класс данных Kotlin, не нарушая Java. Опция **Преобразовать Java-файл в Kotlin-файл** (Convert Java File to Kotlin File) с последующим добавлением модификатора `data` и удалением перезаписей `equals` и `hashCode` даст нам следующее (пример 10.6).

Пример 10.6 [extensions.2:src/main/java/travelator/marketing/CustomerData.kt]

```
data class CustomerData(
    val id: String,
    val givenName: String,
    val familyName: String,
    val score: Int,
    val spend: Double
)
```

Теперь мы можем идти дальше и конвертировать `HighValueCustomerReport` в Kotlin тоже. Он полностью самодостаточен. Но процедура эта не проходит блестяще, потому что `customerDataFrom` не компилируется после преобразования (пример 10.7).

Пример 10.7
[extensions.3:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
object HighValueCustomersReport {
    ...
    @JvmStatic
    fun customerDataFrom(line: String): CustomerData {
        val parts = line.split("\t".toRegex()).toTypedArray()
        val spend: Double = if (parts.size == 4) 0 else parts[4].toDouble() ❶
        return CustomerData(
            parts[0],
            parts[1],
            parts[2], parts[3].toInt(), ❷
            spend
        )
    }
    ...
}
```

❶ Целочисленный литерал не соответствует ожидаемому типу `Double`.

❷ Странное форматирование.

Конвертер недостаточно умен, чтобы знать, что Kotlin не принуждает к двойной точности целое число 0, и это приводит к ошибке компиляции. Давайте поможем IntelliJ, нажав на ошибку и применив комбинацию клавиш <Alt>+<Enter>, чтобы исправить ее, в надежде, что программа вернет должок, когда машины будут править миром. После переформатирования это дает нам следующий вариант (пример 10.8).

Пример 10.8
[extensions.4:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
object HighValueCustomersReport {
    ...
    @JvmStatic
    fun customerDataFrom(line: String): CustomerData {
        val parts = line.split("\t".toRegex()).toTypedArray()
        val spend: Double = if (parts.size == 4) 0.0 else parts[4].toDouble()
        return CustomerData(
            parts[0],
            parts[1],
            parts[2],
            parts[3].toInt(),
            spend
        )
    }
    ...
}
```

Как мы обсуждали в *главе 8*, преобразование поместило функции в `object HighValueCustomersReport`, так что Java-код все еще может их найти.

Если мы попытаемся преобразовать их в функции верхнего уровня, используя методы, описанные ранее в этой главе, то обнаружим, что зависимости, имеющиеся между методами, ведут к тому, что код иногда не компилируется. Мы можем решить проблему либо путем перемещения частных методов, либо просто игнорируя компилятор до тех пор, пока `HighValueCustomersReport` не будет очищен и удален (пример 10.9).

Пример 10.9
[extensions.5:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
package travelator.marketing

...
```

```
@Throws(IOException::class)
fun generate(reader: Reader?, writer: Writer) {
    val valuableCustomers = BufferedReader(reader).lines()
        .skip(1) // заголовок
        .map { line: String -> customerDataFrom(line) }
        .filter { (_, _, _, score) -> score >= 10 }
        .sorted(Comparator.comparing { (_, _, _, score) -> score })
        .collect(Collectors.toList())
    writer.append("ID\tName\tSpend\n")
    for (customerData in valuableCustomers) {
        writer.append(lineFor(customerData)).append("\n")
    }
    writer.append(summaryFor(valuableCustomers))
}

private fun summaryFor(valuableCustomers: List<CustomerData>): String {
    val total = valuableCustomers.stream()
        .mapToDouble { (_, _, _, _, spend) -> spend }
        .sum()
    return "\tTOTAL\t" + formatMoney(total)
}

fun customerDataFrom(line: String): CustomerData {
    val parts = line.split("\t".toRegex()).toTypedArray()
    val spend: Double = if (parts.size == 4) 0.0 else parts[4].toDouble()
    return CustomerData(
        parts[0],
        parts[1],
        parts[2],
        parts[3].toInt(),
        spend
    )
}

private fun lineFor(customer: CustomerData): String {
    return customer.id + "\t" + marketingNameFor(customer) + "\t" +
        formatMoney(customer.spend)
}

private fun formatMoney(money: Double): String {
    return String.format("%#.2f", money)
}

private fun marketingNameFor(customer: CustomerData): String {
    return customer.familyName.toUpperCase() + ", " + customer.givenName
}
```

Хорошо, пришло время искать места, где функции-расширения могут улучшить код. В конце приведенного в примере 10.9 кода находится функция `marketingNameFor`, которую мы видели (в немного другой версии) ранее. Если мы нажмем комбинацию клавиш <Alt>+<Enter> на параметре `customer`, IntelliJ предложит **Преобразовать параметр в получатель** (Convert parameter to receiver). Это даст нам следующий вариант (пример 10.10).

Пример 10.10
[extensions.6:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
private fun lineFor(customer: CustomerData): String {
    return customer.id + "\t" + customer.marketingNameFor() + "\t" +
        formatMoney(customer.spend)
}

...
private fun CustomerData.marketingNameFor(): String {
    return familyName.toUpperCase() + ", " + givenName
}
```

Слово `For` в `marketingNameFor` сбивает с толку, поскольку мы задали параметр в качестве получателя, а `For` не имеет субъекта. Давайте выполним команду **Преобразовать функцию в свойство** (Convert function to property) с именем `marketingName` (*глава 11* объяснит, как и почему), затем **Преобразовать в тело выражения** (Convert to expression body) и далее **Преобразовать конкатенацию в шаблон** (Convert concatenation to template) в обеих строках! Вжик, этот шквал альтернативного ввода с помощью комбинации клавиш <Alt>+<Enter> дает следующий вариант (пример 10.11).

Пример 10.11
[extensions.7:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
private fun lineFor(customer: CustomerData): String =
    "${customer.id}\t${customer.marketingName}\t${formatMoney(customer.spend)}"

private fun formatMoney(money: Double): String {
    return String.format("%#.2f", money)
}

private val CustomerData.marketingName: String
    get() = "${familyName.toUpperCase()}, $givenName"
```

Теперь нас подводит `formatMoney`, поэтому мы снова можем выполнить команду **Преобразовать параметр в получатель** (Convert parameter to receiver), переименовать `formatMoney` в `toMoneyString` и выполнить команду **Преобразовать в тело выражения** (Convert to expression body), как показано в примере 10.12.

Пример 10.12
[extensions.8:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
private fun lineFor(customer: CustomerData): String =
    "${customer.id}\t${customer.marketingName}\t${customer.spend.toMoneyString()}"

private fun Double.toMoneyString() = String.format("%#.2f", this)
```

Здесь нас немного раздражает `String.format`. Kotlin позволил бы нам писать `"%#.2f".format(this)`, но мы предпочитаем менять местами параметр и получатель (пример 10.13).

Пример 10.13
[extensions.9:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
private fun Double.toMoneyString() = this.formattedAs("%#.2f")

private fun Double.formattedAs(format: String) = String.format(format, this)
```

Здесь `Double.formattedAs` — первая функция-расширение, содержащая параметр и получатель, которую мы написали. И всё потому, что другие преобразования были очень специфическими, а это более общее. Поскольку мы думаем об общих преобразованиях, `formattedAs` может одинаково хорошо применяться к любому типу, включая `null`, поэтому мы можем его обновить (пример 10.14).

Пример 10.14
[extensions.10:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
private fun Any?.formattedAs(format: String) = String.format(format, this)
```

После этого он становится хорошим кандидатом для перехода в нашу библиотеку общеполезных функций Kotlin.

Теперь в поле нашего зрения попадает `customerDataFrom`. В настоящее время это выглядит так (пример 10.15).

Пример 10.15
[extensions.11:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun customerDataFrom(line: String): CustomerData {
    val parts = line.split("\t".toRegex()).toTypedArray()
    val spend: Double = if (parts.size == 4) 0.0 else parts[4].toDouble()
    return CustomerData(
        parts[0],
        parts[1],
        parts[2],
        parts[3].toInt(),
```

```
        spend
    )
}
```

Прежде чем продолжить, давайте заметим, что `CharSequence.split()`, `String.toRegex()`, `Collection<T>.toTypedArray()`, `String.toDouble()` и `String.toInt()` — это все функции-расширения, предоставляемые стандартной библиотекой Kotlin.

Мы можем многое привести в порядок, прежде чем обратимся к сигнатуре `customerDataFrom`. В Kotlin есть функция `CharSequence.split(delimiters)`, которую можно применить вместо `regex` (регулярного выражения). Затем мы можем заменить `spend` встроенным выражением, после чего, нажав комбинацию клавиш <Alt>+<Enter>, выбрать опцию **Добавить имена к аргументам вызова** (Add names to call arguments), чтобы помочь разобраться в вызове конструктора (пример 10.16).

Пример 10.16
[extensions.12:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun customerDataFrom(line: String): CustomerData {
    val parts = line.split("\t")
    return CustomerData(
        id = parts[0],
        givenName = parts[1],
        familyName = parts[2],
        score = parts[3].toInt(),
        spend = if (parts.size == 4) 0.0 else parts[4].toDouble()
    )
}
```

В *главе 9* приводятся аргументы в пользу однострочных функций. Приведенный здесь код, конечно, не обязательно должен быть однострочным выражением, но давайте все равно попрактикуемся (пример 10.17).

Пример 10.17
[extensions.13:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun customerDataFrom(line: String): CustomerData =
    line.split("\t").let { parts ->
        CustomerData(
            id = parts[0],
            givenName = parts[1],
            familyName = parts[2],
            score = parts[3].toInt(),
            spend = if (parts.size == 4) 0.0 else parts[4].toDouble()
        )
    }
```

Наконец, мы можем перейти к преобразованию в функцию-расширение. Мы снова меняем имя (на `toCustomerData`), чтобы оно имело смысл в месте вызова (пример 10.18).

Пример 10.18
[extensions.14:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun String.toCustomerData(): CustomerData =
    split("\t").let { parts ->
        CustomerData(
            id = parts[0],
            givenName = parts[1],
            familyName = parts[2],
            score = parts[3].toInt(),
            spend = if (parts.size == 4) 0.0 else parts[4].toDouble()
        )
    }
```

Обратите внимание, что Java в наших тестах все еще может вызывать это как статический метод (пример 10.19).

Пример 10.19
[extensions.13:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
@Test
public void emptySpendIs0() {
    assertEquals(
        new CustomerData("1", "Fred", "Flintstone", 0, 0D),
        HighValueCustomersReportKt.toCustomerData("1\tFred\tFlintstone\t0")
    );
}
```

Теперь давайте обратимся к `summaryFor` (пример 10.20).

Пример 10.20
[extensions.15:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
private fun summaryFor(valuableCustomers: List<CustomerData>): String {
    val total = valuableCustomers.stream()
        .mapToDouble { (_, _, _, _, spend) -> spend }
        .sum()
    return "\tTOTAL\t" + total.toMoneyString()
}
```

Это необычное деструктурирование, но мы можем избавиться от него, вручную преобразовав поток в Kotlin. Это не та задача, с которой IntelliJ могла справиться,

когда мы писали этот раздел, и мы дадим соответствующие рекомендации в *главе 13*. Сейчас же давайте удалим конкатенацию строк, пока мы в этой части кода (пример 10.21).

Пример 10.21
[extensions.16:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
private fun summaryFor(valuableCustomers: List<CustomerData>): String {
    val total = valuableCustomers.sumByDouble { it.spend }
    return "\tTOTAL\t${total.toMoneyString()}"
}
```

Теперь произведем знакомую комбинацию преобразования в функцию-расширение одиночного выражения с соответствующим названием (пример 10.22).

Пример 10.22
[extensions.17:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
private fun List<CustomerData>.summarised(): String =
    sumByDouble { it.spend }.let { total ->
        "\tTOTAL\t${total.toMoneyString()}"
    }
```

На этом этапе только `generate` остается без доработок (пример 10.23).

Пример 10.23
[extensions.18:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
@Throws(IOException::class)
fun generate(reader: Reader?, writer: Writer) {
    val valuableCustomers = BufferedReader(reader).lines()
        .skip(1) // заголовок
        .map { line: String -> line.toCustomerData() }
        .filter { (_, _, _, score) -> score >= 10 }
        .sorted(Comparator.comparing { (_, _, _, score) -> score })
        .collect(Collectors.toList())
    writer.append("ID\tName\tSpend\n")
    for (customerData in valuableCustomers) {
        writer.append(lineFor(customerData)).append("\n")
    }
    writer.append(valuableCustomers.summarised())
}
```

Опять же, в настоящее время нам приходится вручную преобразовывать потоки Java в операции Kotlin со списком (пример 10.24).

Пример 10.24
[extensions.19:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
@Throws(IOException::class)
fun generate(reader: Reader, writer: Writer) {
    val valuableCustomers = reader.readLines()
        .drop(1) // заголовок
        .map(String::toCustomerData)
        .filter { it.score >= 10 }
        .sortedBy(CustomerData::score)
    writer.append("ID\tName\tSpend\n")
    for (customerData in valuableCustomers) {
        writer.append(lineFor(customerData)).append("\n")
    }
    writer.append(valuableCustomers.summarised())
}
```

`Appendable.AppendLine()` — это еще одна функция-расширение, которая позволяет нам упростить этап вывода (пример 10.25).

Пример 10.25
[extensions.20:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
@Throws(IOException::class)
fun generate(reader: Reader, writer: Writer) {
    val valuableCustomers = reader.readLines()
        .drop(1) // header
        .map(String::toCustomerData)
        .filter { it.score >= 10 }
        .sortedBy(CustomerData::score)
    writer.appendLine("ID\tName\tSpend")
    for (customerData in valuableCustomers) {
        writer.appendLine(lineFor(customerData))
    }
    writer.append(valuableCustomers.summarised())
}
```

Здесь мы чувствуем, что должны иметь возможность удалить комментарий `// header` (заголовок) путем извлечения функции. В *разд. «Извлечение части конвейера» главы 13* подробно описано, как извлечь функцию из цепочки, но посмотрите, что происходит, когда мы пробуем этот метод, но не преобразуем `withoutHeader` в функцию расширения (пример 10.26).

Пример 10.26
[extensions.21:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
@Throws(IOException::class)
fun generate(reader: Reader, writer: Writer) {
```

```
    val valuableCustomers = withoutHeader(reader.readLines())
        .map(String::toCustomerData)
        .filter { it.score >= 10 }
        .sortedBy(CustomerData::score)
    writer.appendLine("ID\tName\tSpend")
    for (customerData in valuableCustomers) {
        writer.appendLine(lineFor(customerData))
    }
    writer.append(valuableCustomers.summarised())
}

private fun withoutHeader(list: List<String>) = list.drop(1)
```

Мы потеряли хороший конвейер в потоке слева направо и сверху вниз: `withoutHeader` идет перед `readLines` в тексте, но находится после нее в порядке выполнения. Нажмите комбинацию клавиш <Alt>+<Enter> и примените опцию **Преобразовать параметр в получатель** (Convert parameter to receiver) к параметру `list` в функции `withoutHeader`. Это восстановит поток (пример 10.27).

Пример 10.27
[extensions.22:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
@Throws(IOException::class)
fun generate(reader: Reader, writer: Writer) {
    val valuableCustomers = reader.readLines()
        .withoutHeader()
        .map(String::toCustomerData)
        .filter { it.score >= 10 }
        .sortedBy(CustomerData::score)
    writer.appendLine("ID\tName\tSpend")
    for (customerData in valuableCustomers) {
        writer.appendLine(lineFor(customerData))
    }
    writer.append(valuableCustomers.summarised())
}

private fun List<String>.withoutHeader() = drop(1)
```

Мы можем сделать эту часть более выразительной с помощью еще двух расширений: `List<String>.toValuableCustomers()` и `CustomerData.outputLine` (пример 10.28).

Пример 10.28
[extensions.23:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
@Throws(IOException::class)
fun generate(reader: Reader, writer: Writer) {
    val valuableCustomers = reader
        .readLines()
```

```
        .toValuableCustomers()
        .sortedBy(CustomerData::score)
    writer.appendLine("ID\tName\tSpend")
    for (customerData in valuableCustomers) {
        writer.appendLine(customerData.outputLine)
    }
    writer.append(valuableCustomers.summarised())
}

private fun List<String>.toValuableCustomers() = withoutHeader()
    .map(String::toCustomerData)
    .filter { it.score >= 10 }
...

private val CustomerData.outputLine: String
    get() = "$id\t$marketingName\t${spend.toMoneyString()}"
```

Тем не менее всё это выглядит еще не так приятно, как нам хотелось бы, но мы доказали важность функций-расширений. В *главах 20* и *21* мы завершим этот рефракторинг, а пока вот весь файл целиком (пример 10.29).

Пример 10.29
[extensions.23:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
@Throws(IOException::class)
fun generate(reader: Reader, writer: Writer) {
    val valuableCustomers = reader
        .readLines()
        .toValuableCustomers()
        .sortedBy(CustomerData::score)
    writer.appendLine("ID\tName\tSpend")
    for (customerData in valuableCustomers) {
        writer.appendLine(customerData.outputLine)
    }
    writer.append(valuableCustomers.summarised())
}

private fun List<String>.toValuableCustomers() = withoutHeader()
    .map(String::toCustomerData)
    .filter { it.score >= 10 }

private fun List<String>.withoutHeader() = drop(1)

private fun List<CustomerData>.summarised(): String =
    sumByDouble { it.spend }.let { total ->
        "\tTOTAL\t${total.toMoneyString()}"
    }

internal fun String.toCustomerData(): CustomerData =
    split("\t").let { parts ->
```

```
        CustomerData(
            id = parts[0],
            givenName = parts[1],
            familyName = parts[2],
            score = parts[3].toInt(),
            spend = if (parts.size == 4) 0.0 else parts[4].toDouble()
        )
    }

private val CustomerData.outputLine: String
    get() = "$id\t$marketingName\t${spend.toMoneyString()}"

private fun Double.toMoneyString() = this.formattedAs("%#.2f")

private fun Any?.formattedAs(format: String) = String.format(format, this)

private val CustomerData.marketingName: String
    get() = "${familyName.toUpperCase()}, $givenName"
```

Обратите внимание, что каждая функция, кроме точки входа, является однострочной функцией-расширением. Мы не сделали `generate` функцией-расширением, потому что для нее нет естественного параметра для создания получателя и она не похожа на естественную операцию с `Reader` или `Writer`. Всё может измениться, когда мы продолжим рефакторинг этого фрагмента кода в *главе 20*. Что ж, когда доберемся до нее, тогда и выясним.

Двигаемся дальше

Функции-расширения и свойства-расширения — невоспетые герои языка Kotlin. Их каноническое использование заключается в добавлении операций к типам, которые мы не можем изменить сами.

Однако языковые возможности и инструментарий Kotlin в совокупности побуждают нас, и довольно настойчиво, писать функции-расширения и для наших *собственных* типов. Оператор безопасного вызова Kotlin делает вызов функции-расширения через потенциально нулевую ссылку более удобным, чем через передачу ссылки на функцию в качестве параметра, когда она не является нулевой. Тип автономного обобщенного расширения может выражать отношения между получателем и его результатом, которые не могут быть выражены открытыми методами. Автозаполнение в IntelliJ включает в себя функции-расширения наряду с методами, которые могут вызывать значение, но оно не показывает нам функции, которым мы можем передать значение в качестве параметра.

В результате функции-расширения позволяют нам писать код, который легче обнаружить, понять и поддерживать. Многие другие методы, представленные в этой книге, основаны на функциях-расширениях, как мы увидим в *главе 15 «От инкапсулированных коллекций к псевдонимам типов»*, *главе 18 «От открытых классов к запечатанным»* и других.

ГЛАВА 11

От методов к свойствам

Java не делает различий между методами доступа к свойствам и другими типами. Kotlin, с другой стороны, обрабатывает свойства иначе, чем функции-члены. Когда мы должны предпочесть вычисляемое свойство функции, возвращающей результат?

Поля, акцессоры и свойства

Большинство языков программирования позволяют нам каким-либо образом группировать данные, присваивая имена (и часто типы) свойствам составного элемента.

Вот, например, *запись*, состоящая из трех *полей*, в ALGOL W — одном из первых языков общего назначения, поддерживающих типы записей. (ALGOL W также был языком, на котором Тони Хоар ввел нулевые ссылки.)

```
RECORD PERSON (
    STRING(20) NAME;
    INTEGER AGE;
    LOGICAL MALE;
);
```

Тогда все было по-другому: у настоящих программистов были только ЗАГЛАВНЫЕ БУКВЫ, а пол был логическим (булевым) значением.

В ALGOL W мы можем (ну хорошо, могли бы) обновить возраст, указанный в записи `PERSON`:

```
AGE(WILMA) := AGE(WILMA) + 1;
```

В этом случае компилятор выдаст инструкции для доступа к памяти записи, найдет байты, представляющие возраст Вильмы, и увеличит его. Записи, также известные как *structs* (для структуры) на других языках, удобны для группировки связанных данных. Здесь нет никакой скрытой информации, только композиция.

Большинство ранних объектно-ориентированных систем (в частности, C++) были основаны на этом механизме записи. Экземпляры переменных представляли собой просто поля записи, а методы (они же функции-члены) — поля, содержащие указатели на функции. Иным был Smalltalk. Объекты Smalltalk могут иметь экземпляры переменных, но доступ к паттерну состояния осуществляется путем отправки объекту сообщения с запросом значения. Сообщения, а не поля, являются в нем фундаментальной абстракцией.

Разработчики Java использовали понемногу каждый подход. Объекты могут иметь общедоступные поля, но клиенты не могут просто залезть в свою память, чтобы извлечь их. Они должны вызывать инструкции байт-кода для доступа к их значениям. Это позволяет нам рассматривать классы как записи, позволяя среде выполнения обеспечивать доступ к закрытым полям.

Хотя прямой доступ к полям был *разрешен*, с самого начала это не поощрялось. Мы не можем изменить внутреннее представление данных, если клиенты обращаются к полям напрямую, по крайней мере, без одновременного изменения этих клиентов. Мы также не можем поддерживать какие-либо инвариантные отношения между полями, если клиенты могут изменять их напрямую, и, как показано в *главе 5*, в те дни все объекты были связаны с мутацией. Доступ к полям также не является полиморфным, поэтому подклассы не могут изменять свою имплементацию. В те дни все объекты тоже были связаны с подклассами.

Поэтому вместо прямого доступа к полю в Java мы обычно пишем *методы доступа* (акцессоры): геттеры и (при необходимости) сеттеры. Геттеры обычно ничего не делают, кроме возвращения значения поля, но вместо этого они могут вычислять значение из других полей. Сеттеры способны поддерживать инварианты или события запуска, а также обновлять поле или, возможно, несколько полей.

Иногда, однако, данные — это просто данные. В таком случае хорошо осуществлять прямой запрос к общедоступным полям, особенно когда у нас есть немутирующие значения (т. е. конечные поля неизменяемых типов). Для более сложных моделей полиморфное поведение и/или единый способ доступа к значениям либо из поля, либо из вычисления становятся полезными, и методы доступа вступают в свои права.

Разработчики Kotlin предпочли не принимать решения от нас и поддерживают только методы доступа. В результате язык не обеспечивает предоставление прямого доступа к полям. Kotlin будет генерировать код для доступа к общедоступным полям классов Java, но сам не определяет общедоступные поля. (Специальная аннотация `@JvmField` предоставляет запасной выход, если вам это действительно нужно.) Это было сделано, чтобы побудить нас для получения возможности изменять представления использовать акцессоры, не затрагивая клиентов.

Чтобы еще больше стимулировать применение акцессоров, Kotlin позволяет нам генерировать как частные переменные-члены, так и акцессор в одном объявлении *свойства*. Итак, в Java мы можем предоставить доступ к полю напрямую:

```
public class PersonWithPublicFields {
    public final String givenName;
    public final String familyName;
    public final LocalDate dateOfBirth;

    public PersonWithPublicFields(
        String givenName,
        String familyName,
        LocalDate dateOfBirth
```

```
    ) {
        this.givenName = givenName;
        this.familyName = familyName;
        this.dateOfBirth = dateOfBirth;
    }
}
```

Или написать свой собственный акцессор:

```
public class PersonWithAccessors {
    private final String givenName;
    private final String familyName;
    private final LocalDate dateOfBirth;

    public PersonWithAccessors(
        ...
    }

    public String getGivenName() {
        return givenName;
    }

    public String getFamilyName() {
        return familyName;
    }

    ...
}
```

В Kotlin доступны только свойства:

```
data class PersonWithProperties(
    val givenName: String,
    val familyName: String,
    val dateOfBirth: LocalDate
) {
}
```

Это объявление создаст частные поля: `givenName`, `familyName` и `dateOfBirth`, методы-акцессоры `getGivenName()` и т. д., а также конструктор для инициализации остальных полей.

В Java можно получить доступ к (видимым) полям напрямую или вызвать акцессоры:

```
public static String accessField(PersonWithPublicFields person) {
    return person.givenName;
}

public static String callAccessor(PersonWithAccessors person) {
    return person.getGivenName();
}
```

```
public static String callKotlinAccessor(PersonWithProperties person) {
    return person.getGivenName();
}
```

В Kotlin мы можем получать доступ к видимым полям (из классов Java) напрямую или вызывать акцессоры, как если бы они были полями:

```
accessField(person: PersonWithPublicFields): String =
    person.givenName

fun callAccessor(person: PersonWithAccessors): String =
    person.givenName

fun callKotlinAccessor(person: PersonWithProperties): String =
    person.givenName
```

Свойства — это удобство, подкрепленное некоторой магией компилятора. Они делают использование полей и акцессоров в Kotlin таким же простым, как и старые добрые поля в Java, поэтому мы, естественно, напишем код, который может использовать преимущества инкапсуляции. Например, мы можем столкнуться с тем, что требуется определить свойство в интерфейсе или вычислить свойство, которое мы ранее сохранили.

Вычисляемые свойства — это те, которые не подкреплены полем. Если в нашем случае `givenName` и `family Name` подкреплены полем, то нет никакой нужды хранить `fullName`. Мы вычислим это свойство, когда потребуется:

```
public class PersonWithPublicFields {
    public final String givenName;
    public final String familyName;
    public final LocalDate dateOfBirth;

    public PersonWithPublicFields(
        ...
    }

    public String getFullName() {
        return givenName + " " + familyName;
    }
}
```

Если мы используем прямой доступ к полю в Java, существует разница между тем, как мы получаем доступ к сохраненным и вычисляемым свойствам:

```
public static String fieldAndAccessor(PersonWithPublicFields person) {
    return
        person.givenName + " " +
        person.getFullName();
}
```

В Kotlin это не так даже при доступе к полям и методам Java, что приятно:

```
fun fieldAndAccessor(person: PersonWithPublicFields) =
    person.givenName + " " +
    person.fullName
```

В Kotlin мы определяем вычисляемые свойства вне конструктора:

```
data class PersonWithProperties(
    val givenName: String,
    val familyName: String,
    val dateOfBirth: LocalDate
) {
    val fullName get() = "$givenName $familyName"
}
```

Итак, в Java мы *можем* определять классы, обеспечивающие прямой доступ к полям, но по преимуществу *должны* использовать акцессоры, которые представляют собой просто методы, содержащие в названии (по соглашению, но не всегда) префикс `get` или `set`. В Kotlin же мы не можем определять поля и акцессоры раздельно. Когда мы определяем свойство в Kotlin, компилятор генерирует поле и акцессоры, соответствующие соглашению об именовании Java. Когда мы ссылаемся на свойство в Kotlin, синтаксис совпадает с синтаксисом Java для доступа к полю, но компилятор генерирует вызов акцессора. Это применимо даже к границе взаимодействия: когда мы ссылаемся на свойства объектов Java, компилятор генерирует вызов акцессора, если таковой существует, и следует соглашению об именовании Java.

Как выбрать?

Вернемся к вопросу, заданному в начале главы: когда мы должны выбрать вычисляемое свойство, а когда метод, учитывая, что вычисляемые свойства — это просто методы, «присыпанные сахарной пудрой»?

Хорошее эмпирическое правило для этого — использовать свойство, когда оно зависит только от других свойств типа и является простым для вычисления. Например, таким свойством может быть `fullName`, поскольку оно легко вычисляется. А как насчет возраста человека?

Мы можем легко вычислить возраст (игнорируя часовые пояса) из свойства `dateOfBirth`, поэтому у нас может возникнуть соблазн написать в Java `fred.getAge()`. Но эта функция зависит не только от других свойств, она также зависит от того, когда мы ее вызываем. Хотя это маловероятно, но `fred.age == fred.age` может вернуть значение `false`.

Возраст — это действие (см. *разд «Действия» главы 7*). Его результат зависит от того, когда оно вызывается. Свойства же должны быть вычислениями (см. *разд. «Вычисления» главы 7*), неподвластными времени и зависящими только от их входных данных, — в нашем случае свойства `dateOfBirth`. Следовательно, выражение `age()` должно быть функцией, а не свойством:

```
data class PersonWithProperties(
    val givenName: String,
```

А что можно сказать о криптографическом хеше всех остальных свойств объекта? Он предствляет собой вычисление (для немутирующих объектов), но если вычислять его затратно, это должен быть метод `hash()`, а не свойство `hash`. Возможно, мы даже захотим намекнуть на затратность метода в его названии:

```
    val familyName: String,
    val dateOfBirth: LocalDate
) {
    fun age() = Period.between(dateOfBirth, LocalDate.now()).years
}
```

Мы могли бы создать свойство, предварительно рассчитав его и сохранив в поле:

```
data class PersonWithProperties(
    val givenName: String,
    val familyName: String,
    val dateOfBirth: LocalDate
) {
    fun computeHash(): ByteArray =
        someSlowHashOf(givenName, familyName, dateOfBirth.toString())
}
```

Недостаток этого подхода заключается в том, что создание каждого экземпляра замедляется, независимо от того, осуществляется ли когда-либо доступ к его `hash` или нет. Мы могли бы разделить расхождения с помощью отложенного свойства:

```
data class PersonWithProperties(
    val givenName: String,
    val familyName: String,
    val dateOfBirth: LocalDate
) {
    val hash: ByteArray =
        someSlowHashOf(givenName, familyName, dateOfBirth.toString())
}
```

В ограниченной области это было бы прекрасным решением. Но если класс используется более широко, мы должны по крайней мере намекнуть на потенциальную проблему с производительностью при первом вызове, скрыв вычисляемое свойство за функцией:

```
data class PersonWithProperties(
    val givenName: String,
    val familyName: String,
    val dateOfBirth: LocalDate
) {
    private val hash: ByteArray by lazy {
        someSlowHashOf(givenName, familyName, dateOfBirth.toString())
    }
```

```
    fun hash() = hash
}
```

В этом случае мы могли бы рассмотреть свойство-расширение. Однако, как показано в *главе 10*, свойства-расширения могут быть только вычислены, а не подкреплены полем, и поэтому они не могут быть отложенными. Кроме того, большая часть обсуждения здесь также относится к функциям-расширениям в сравнении со свойствами-расширениями.

Мутирующие свойства

Так что же такое мутирующие (изменяющиеся) свойства? Kotlin позволяет нам определять свойства для переменной как `var`.

Если вы дочитали до этого места, то знаете, что ваши авторы предпочитают сохранять свои данные (см. *главу 5*) и коллекции (см. *главу 6*) немутирующими. Мы можем представить себе использование Kotlin в целях определения изменяемого свойства для интеграции с некоторым кодом Java, который требовал этого, но очень, очень редко задействуем мутирующие публичные свойства на практике. Иногда мы можем определять свойство, которое будет меняться со временем (например, для предоставления доступа к счетчику), но почти никогда не такое, которое могут задать клиенты. На практике обнаруживается, что классы данных с методами копирования работают лучше почти во всех ситуациях, которые могли бы потребовать вызов сеттера. Фактически можно было бы зайти дальше и сказать, что разрешение свойств `var` в классах данных было ошибкой языковой конструкции.

Рефакторинг к свойствам

IntelliJ обеспечивает отличную поддержку рефакторинга для преобразования между методами и свойствами Kotlin. Это, с одной стороны, просто, потому что оба варианта являются просто вызовами методов, а с другой — сложно, потому что взаимодействие Java полагается на соглашения об именах для идентификации методов доступа (акцессоров). Давайте рассмотрим пример из программы `Travelator`.

Некоторые из наших более мелких клиентов любят разбивать палаточный лагерь, поэтому мы указываем в приложении места для кемпинга (пример 11.1).

Пример 11.1 [methods-to-properties.0:src/main/java/travelator/CampSite.java]

```
public class CampSite {
    private final String id;
    private final String name;
    private final Address address;
    ...

    public CampSite(
        String id,
```

```
        String name,
        Address address
        ...
    ) {
        this.id = id;
        this.name = name;
        this.address = address;
        ...
    }

    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getCountryCode() {
        return address.getCountryCode();
    }

    public String region() {
        return address.getRegion();
    }

    ...
}
```

Это типично для класса домена, который рос с годами. У него есть множество свойств, и некоторые из них подкреплены такими полями, как `id` и `name`, а некоторые — вычислены (для небольших значений вычислений) как `countryCode` и `region`. Кто-то проигнорировал соглашения о компонентах, присвоив акцессору название `region` вместо `getRegion`. Но нам ясно, что имелось в виду. Перед вами фрагмент кода, использующий акцессоры (пример 11.2).

Пример 11.2 [methods-to-properties.0:src/main/java/travelator/CampSites.java]

```
public class CampSites {

    public static Set<CampSite> sitesInRegion(
        Set<CampSite> sites,
        String countryISO,
        String region
    ) {
        return sites.stream()
```

```
            .filter( campSite ->
                campSite.getCountryCode().equals(countryISO) &&
                    campSite.region().equalsIgnoreCase(region)
            )
            .collect(toUnmodifiableSet());
    }
}
```

Давайте преобразуем `Campsite` в Kotlin при помощи IntelliJ (а затем сделаем его классом данных), как показано в примере 11.3.

Пример 11.3 [methods-to-properties.1:src/main/java/travelator/CampSite.kt]

```
data class CampSite(
    val id: String,
    val name: String,
    val address: Address,
    ...
) {
    val countryCode: String
        get() = address.countryCode

    fun region(): String {
        return address.region
    }

    ...
}
```

Наши свойства, поддерживаемые полями, стали свойствами конструктора, а вычисляемый `countryCode` — вычисляемым свойством. Однако IntelliJ не понял, что `region` является свойством, потому что в нем не соблюдено соглашение об именовании геттера, и просто преобразовал метод. В итоге клиентский код не был изменен. Чтобы исправить оплошность, нажмите на `region` комбинацию клавиш <Alt>+ +<Enter> и выберите опцию **Преобразовать функцию в свойство** (Convert function to property), как показано в примере 11.4.

Пример 11.4 [methods-to-properties.2:src/main/java/travelator/CampSite.kt]

```
val region: String
    get() {
        return address.region
    }
```

Как и в случае с большинством вычисляемых свойств, это лучше использовать в виде однострочного выражения (см. *главу 9*), как показано в примере 11.5.

Пример 11.5 [methods-to-properties.3:src/main/java/travelator/CampSite.kt]

```
val region: String get() = address.region
```

Изменение метода Kotlin `region` на свойство означает, что метод акцессора теперь будет называться `getRegion`. К счастью, IntelliJ достаточно умен, чтобы подправить наших клиентов для нас (пример 11.6).

Пример 11.6 [methods-to-properties.3:src/main/java/travelator/CampSites.java]

```
public static Set<CampSite> sitesInRegion(
    Set<CampSite> sites,
    String countryISO,
    String region
) {
    return sites.stream()
        .filter( campSite ->
            campSite.getCountryCode().equals(countryISO) &&
                campSite.getRegion().equalsIgnoreCase(region) ❶
        )
        .collect(toUnmodifiableSet());
}
```

❶ `campsite.region()` **заменен на** `campsite.getRegion()`.

Если сейчас мы преобразуем `sitesInRegion` в Kotlin, то получим следующий вариант (пример 11.7).

Пример 11.7 [methods-to-properties.4:src/main/java/travelator/CampSites.kt]

```
object CampSites {
    fun sitesInRegion(
        sites: Set<CampSite>,
        countryISO: String,
        region: String?
    ): Set<CampSite> {
        return sites.stream()
            .filter { campSite: CampSite ->
                campSite.countryCode == countryISO &&
                    campSite.region.equals(region, ignoreCase = true) ❶
            }
            .collect(Collectors.toUnmodifiableSet())
    }
}
```

❶ `campsite.region` **теперь вызывает** `campsite.getRegion()`.

В *главе 8* было показано, как переместить `sitesInRegion` на верхний уровень, а в *главе 10* — как преобразовать `sitesInRegion` в функцию-расширение (пример 11.8).

Пример 11.8 [methods-to-properties.5:src/main/java/travelator/CampSites.kt]

```
fun Set<CampSite>.sitesInRegion(
    countryISO: String,
    region: String
): Set<CampSite> {
    return stream()
        .filter { campSite: CampSite ->
            campSite.countryCode == countryISO &&
                campSite.region.equals(region, ignoreCase = true)
        }
        .collect(Collectors.toUnmodifiableSet())
}
```

В *главе 13 «От потоков к итерируемым объектам и последовательностям»* и в *главе 9 «От многострочных функций к однострочным»* показано, как завершить работу, чтобы получить следующий результат (пример 11.9).

Пример 11.9 [methods-to-properties.6:src/main/java/travelator/CampSites.kt]

```
fun Iterable<CampSite>.sitesInRegion(
    countryISO: String,
    region: String
): Set<CampSite> =
    filter { site ->
        site.countryCode == countryISO &&
            site.region.equals(region, ignoreCase = true)
    }.toSet()
```

Благодаря наличию отличных инструментов и взаимодействию с методами, акцессорами и свойствами у нас получился приятно короткий рефакторинг. Поскольку мы жаждем вашего одобрения, добавим еще одну настройку.

`sitesInRegion` — это немного странный метод. Он восполняет недостаток в нашем моделировании, который заключается в том, что регионы — это просто строки, а не объекты. Без кода страны, если мы будем фильтровать только по названию региона (например, `Hampshire`), то рискуем вернуть перечень мест, большинство из которых находятся в английском графстве, но один (Moonlight Camping — звучит красиво) расположен на острове в Канаде. Пока мы не исправим это, может быть вынести предикат фильтра в отдельный метод (пример 11.10)?

Пример 11.10 [methods-to-properties.7:src/main/java/travelator/CampSites.kt]

```
fun Iterable<CampSite>.sitesInRegion(
    countryISO: String,
    region: String
): Set<CampSite> =
    filter { site ->
        site.isIn(countryISO, region)
    }.toSet()

fun CampSite.isIn(countryISO: String, region: String) =
    countryCode == countryISO &&
        this.region.equals(region, ignoreCase = true)
```

Теперь, когда у нас есть `Campsite.isIn(...)`, `sitesInRegion` может быть заменен встроенным выражением в местах, где его вызывают, потому что код теперь действительно не требует пояснений. Мы предпочитаем находить и публиковать фундаментальные операции, на которых могут основываться клиенты, а не скрывать их внутри функций. Потянув за эту нить, мы могли бы расширить функциональность `isIn`, сделав `region` необязательным (пример 11.11).

Пример 11.11 [methods-to-properties.8:src/main/java/travelator/CampSites.kt]

```
fun CampSite.isIn(countryISO: String, region: String? = null) =
    when (region) {
        null -> countryCode == countryISO
        else -> countryCode == countryISO &&
            region.equals(this.region, ignoreCase = true)
    }
```

Нэт предпочитает то же самое, но с оператором Элвиса (пример 11.12).

Пример 11.12 [methods-to-properties.9:src/main/java/travelator/CampSites.kt]

```
fun CampSite.isIn(countryISO: String, region: String? = null) =
    countryCode == countryISO &&
        region?.equals(this.region, ignoreCase = true) ?: true
```

Дункану нравится хороший оператор Элвиса, но он считает, что код и без него понятен. В вашей команде, вероятно, будут по этому поводу вестись маленькие сражения (выберите путь Дункана).

Фундаментальная операция, подобная `isIn`, теперь может быть повышена до метода (в противоположность функции-расширению) `Campsite` или, даже лучше, `Address`. Таким образом, проблема регионов, не являющихся объектами, ограничивается

типом, наиболее близким к проблеме, и ее исправление там окажет наименьшее влияние на остальную часть базы кода.

Двигаемся дальше

Kotlin предоставляет удобный синтаксис как для свойств с поддержкой полей, так и для вычисляемых свойств, позволяющий нам выразить разницу между доступом к свойству и вызовом функции, даже если они представляют собой один и тот же механизм передачи сообщений «под капотом».

Мы должны предпочесть свойство методу, когда оно применяется к типу значения. Свойство зависит только от значения и не требует больших затрат на вычисление. В этих случаях рефакторинг от метода к свойству прост и облегчает понимание нашего кода.

ГЛАВА 12

От функций к операторам

Если у нас есть большая база кода Java, нашим Java и Kotlin придется сосуществовать в течение некоторого времени. Что мы можем сделать для поддержки соглашений обоих языков по мере постепенного перевода системы на Kotlin?

До сих пор мы показывали, как перевод кода с Java на Kotlin происходит за одну процедуру. Мы использовали автоматический рефакторинг для безопасного выполнения перевода, но к концу операции весь затронутый код оказывался преобразованным в код, характерный для Kotlin.

В объемных кодовых базах это не всегда возможно. В процессе перехода на Kotlin мы должны продолжать работать с Java. Там, где между ними есть пограничная зона, мы хотим использовать обычную Java с одной стороны и обычный Kotlin — с другой. Этот метод отлично подходит, когда мы преобразуем базовые классы, поддерживающие большую часть функциональности нашей системы.

Базовый класс: *Money*

Каждая система содержит некоторые базовые классы, которые используются во многих частях кодовой базы. Примером такого класса в `Travelator` может служить класс `Money`, с которым мы впервые повстречались в *главе 3*. Путешественникам необходимо планировать бюджет своих поездок. Они хотят сравнить стоимость различных их вариантов, посмотреть, как расходы на них конвертируются в предпочитаемую ими валюту, забронировать что-то, оплатить что-то и т. д. Класс `Money` используется настолько широко, что мы не можем преобразовать его и весь зависящий от него код Java в код Kotlin одним махом. Мы должны продолжать работать над функциями, которые используют `Money` как в Java, так и в Kotlin, пока продолжается процесс конвертации.

Это помещает нас между Сциллой и Харибдой. Оставляем ли мы `Money` как класс Java, пока преобразуем зависящий от него код в Kotlin, но в то же время ограничиваем возможности Kotlin, которые мы можем использовать в этом зависимом коде? Или мы преобразуем класс `Money` в Kotlin, пока у нас все еще есть использующий его Java-код, что позволяет нам использовать функции Kotlin в зависимом коде, но делает оставшийся Java-код непоследовательным и нетрадиционным?

Тот факт, что у нас даже есть эти варианты выбора, свидетельствует о том, насколько хорошо взаимодействие Kotlin/Java работает в обоих направлениях. Но на

практике нам не приходится выбирать. С помощью некоторых хитрых тактик рефакторинга и нескольких аннотаций для управления тем, как компилятор Kotlin генерирует код для JVM, нам удается получить лучшее из обоих миров. Мы можем определить `Money` в Kotlin, что позволит нам воспользоваться преимуществами функций Kotlin, и при этом предоставить соответствующий API для кода Java, который мы поддерживаем.

Итак, мы преобразовали класс `Money` в Kotlin в *главе 3*. Поскольку эта операция производилась в конце той главы, мы (извините, без вас) смогли сделать код более кратким, не затрагивая код Java, который зависит от него. Мы преобразовали большинство методов в форму *однострочного выражения* (см. *главу 9*) и воспользовались преимуществами Kotlin для вывода типов, зависящих от потока, чтобы значительно упростить метод `equals`.

Класс `Money` теперь выглядит так (пример 12.1). Он существенно не отличается от варианта, приведенного в *главе 3*, но имеет намного меньше синтаксического шума.

Пример 12.1 [operators.0:src/main/java/travelator/money/Money.kt]

```
class Money private constructor(
    val amount: BigDecimal,
    val currency: Currency
) {
    override fun equals(other: Any?) =
        this === other ||
            other is Money &&
            amount == other.amount &&
            currency == other.currency

    override fun hashCode() =
        Objects.hash(amount, currency)

    override fun toString() =
        amount.toString() + " " + currency.currencyCode

    fun add(that: Money): Money {
        require(currency == that.currency) {
            "cannot add Money values of different currencies"
        }
        return Money(amount.add(that.amount), currency)
    }

    companion object {
        @JvmStatic
        fun of(amount: BigDecimal, currency: Currency) = Money(
            amount.setScale(currency.defaultFractionDigits),
            currency
        )
```

```
        ... и удобные способы перезагрузки операторов
    }
}
```

Тем не менее он по-прежнему сохраняет сущность Java, как и код Kotlin, который его использует. Класс `Money` соответствует соглашениям для типов значений, которые распространены в современной Java, но не так, как обычно делается в Kotlin. В частности, он использует методы объекта-компаньона для создания значений и задействует методы, а не операторы, для выполнения арифметических действий.

В одноязычной кодовой базе было бы довольно просто решить эти проблемы. Однако у нас все еще есть много Java-кода, который использует класс `Money`. Мы будем продолжать вносить изменения *и* в Java, *и* в Kotlin до тех пор, пока Kotlin полностью не вытеснит Java. В то же время мы хотим убедиться, что код, использующий `Money` на любом языке, достаточно общепризнан, чтобы не играть с огнем.

Добавление определяемого пользователем оператора

Код Kotlin, ведущий расчеты со значениями `Money`, все еще довольно неуклюжий:

```
val grossPrice = netPrice.add(netPrice.mul(taxRate))
```

Он не существенно отличается от своего эквивалента на Java:

```
final var grossPrice = netPrice.add(netPrice.mul(taxRate));
```

Использование *методов* для арифметических операций затрудняет чтение вычислений. Это лучшее, что можно сделать на Java, но в Kotlin мы можем определять арифметические операторы для наших собственных классов, что позволяет записывать это вычисление так:

```
val grossPrice = netPrice + netPrice * taxRate
```

Возьмем в качестве примера сложение и посмотрим, как мы можем дать классу `Money` арифметические операторы.

Для этого мы присваиваем классу `Money` оператор `+`, написав операторный метод или функцию-расширение с именем `plus`, а также переименовываем для этого класса на `plus` существующий метод `add` и добавляем модификатор `operator` (пример 12.2).

Пример 12.2 [operators.2:src/main/java/travelator/money/Money.kt]

```
class Money private constructor(
    val amount: BigDecimal,
    val currency: Currency
) {
    ...
```

```
    operator fun plus(that: Money): Money {
        require(currency == that.currency) {
            "cannot add Money values of different currencies"
        }
        return Money(amount.add(that.amount), currency)
    }

    ...
}
```

С этим изменением наш код Kotlin может добавлять значения `Money` с помощью оператора `+`, тогда как код Java вызывает `plus` в качестве метода.

Однако проверка показывает, что наше переименование распространяется на сотни файлов Java-кода, внедряя в них имя, которое не соответствует соглашениям Java. Все Java-классы в стандартной библиотеке с арифметическими операциями — такие как `BigDecimal` и `BigInteger` — используют имя `add`, а не `plus`.

Мы можем сделать так, чтобы функция эта имела разные имена в Java и Kotlin, снабдив ее определение аннотацией `@JvmName`. Давайте отменим только что внесенное изменение и попробуем выполнить его еще раз, начав с аннотирования метода с помощью конструкции `@JvmName("add")` (пример 12.3).

Пример 12.3 [operators.3:src/main/java/travelator/money/Money.kt]

```
@JvmName("add")
fun add(that: Money): Money {
    require(currency == that.currency) {
        "cannot add Money values of different currencies"
    }
    return Money(amount.add(that.amount), currency)
}
```

Теперь, когда мы переименовываем метод `add` в `plus`, наш Java-код остается неизменным, и пометка его как оператора позволяет коду Java и Kotlin вызывать этот метод согласно с их соответствующими языковыми соглашениями (пример 12.4).

Пример 12.4 [operators.4:src/main/java/travelator/money/Money.kt]

```
@JvmName("add")
operator fun plus(that: Money): Money {
    require(currency == that.currency) {
        "cannot add Money values of different currencies"
    }
    return Money(amount.add(that.amount), currency)
}
```

Желательно ли это? Не добавит ли путаницы прием, в результате которого один и тот же метод появляется под разными именами в разных частях одной и той же ко-

довой базы? С другой стороны, поскольку это операторный метод, имя `plus` должно появляться только в определении метода, а все виды использования метода из Kotlin должны осуществляться с помощью оператора `+`. Фрагмент `operator fun plus` больше похож на ключевое слово языка, чем на имя метода, а IntelliJ беспрепятственно переходит от вызовов `add` в Java к определением `operator plus` в Kotlin. И как бы там ни было, но ваши авторы считают, что в этом случае использовать аннотацию `@JvmName` имеет смысл. Однако в целом вам надо будет прийти внутри вашей команды к соглашению о том, как использовать аннотацию `@JvmName` при настройке классов Kotlin для клиентов Java.

Вызов нашего оператора из существующего кода Kotlin

Просматривая клиентский код Kotlin, мы обнаруживаем, что у нас все еще есть проблема. На момент подготовки книги IntelliJ не имел автоматического рефакторинга для замены всех прямых вызовов операторного метода на использование соответствующего оператора. Любой из наших фрагментов кода Kotlin, который был вызовом метода `Money.add` до того, как мы преобразовали его в оператор, будет все равно вызывать `Money.plus` как метод вместо использования оператора `+`. IntelliJ может автоматически рефакторить в оператор каждое из мест вызова метода, но нам придется проходить их, вызывая рефакторинг поштучно.

Чтобы решить эту проблему, мы можем использовать последовательность шагов рефакторинга, чтобы переключить весь наш код Kotlin на одновременное использование оператора `+` и оставить в коде возможность повторять шаги по мере преобразования всё большего количества классов Java в Kotlin. Итак, давайте снова отменим наше изменение и проведем еще одну попытку преобразования.

На этот раз мы извлечем все тело метода `add` как метод с названием `plus` и сделаем его общедоступным операторным методом (пример 12.5).

Пример 12.5 [operators.6:src/main/java/travelator/money/Money.kt]

```
fun add(that: Money): Money {
    return plus(that)
}

operator fun plus(that: Money): Money {
    require(currency == that.currency) {
        "cannot add Money values of different currencies"
    }
    return Money(amount.add(that.amount), currency)
}
```

Используя автоматический рефакторинг IntelliJ, мы делаем `this` явным в вызове `plus` (пример 12.6).

Пример 12.6 [operators.7:src/main/java/travelator/money/Money.kt]

```
fun add(that: Money): Money {
    return this.plus(that)
}
```

Из этой формы IntelliJ позволяет нам автоматически выполнять рефакторинг от вызова метода к оператору (пример 12.7).

Пример 12.7 [operators.8:src/main/java/travelator/money/Money.kt]

```
fun add(that: Money): Money {
    return this + that
}
```

Наконец, мы можем преобразовать метод `add` в форму однострочного выражения (пример 12.8).

Пример 12.8 [operators.9:src/main/java/travelator/money/Money.kt]

```
fun add(that: Money) = this + that

operator fun plus(that: Money): Money {
    require(currency == that.currency) {
        "cannot add Money values of different currencies"
    }
    return Money(amount.add(that.amount), currency)
}
```

Теперь у нас есть два метода сложения. Оператор `plus` реализует логику сложения и является тем, что мы хотели бы задействовать во всем нашем коде Kotlin в будущем, но пока ничего не вызывает его напрямую. Метод `add` используется нашим кодом Java, пока он существует, и его тело содержит идеальный синтаксис, который мы хотели бы иметь в нашем коде Kotlin.

Мы можем попробовать преобразовать весь наш код Kotlin, добавляющий значения `Money`, чтобы использовать синтаксис оператора, заменив метод `Money.add` встроенным выражением. Но когда мы это делаем, IntelliJ сообщает, что ему не удалось заменить встроенными выражениями все варианты использования `add`. А ведь это как раз то, чего мы хотели! Мы не можем заменить встроенным выражением код Kotlin в Java, поэтому IntelliJ встроил тело метода `add` только в места вызовов Kotlin. А также сохранил его определение в классе `Money`, потому что он все еще вызывается Java. Весь наш код Kotlin теперь использует оператор `+`, и Java-код не изменился.

В будущем, когда станем переводить больше Java-классов, добавляющих значения `Money` в Kotlin, мы сможем снова заменить встроенным выражением метод `add`, что-

бы заставить преобразованный класс Kotlin использовать оператор + вместо синтаксиса вызова метода. До тех пор пока в нашей кодовой базе есть Java-код, который его вызывает, IntelliJ сохранит метод add. А когда мы преобразуем последний класс Java, который добавляет Money, IntelliJ удалит уже неиспользуемый метод add, выполнив это в качестве части рефакторинга по замене методов встроенными выражениями. После чего наша кодовая база будет использовать только оператор +.

Операторы для существующих классов Java

Пока идет работа над методом plus, у нас есть шанс воспользоваться возможностью применить оператор + *внутри* метода. Класс Money представляет свойство amount в виде BigDecimal — класса стандартной библиотеки Java. И мы можем заменить вызов метода BigDecimal.add оператором + (пример 12.9).

Пример 12.9 [operators.11:src/main/java/travelator/money/Money.kt]

```
operator fun plus(that: Money): Money {
    require(currency == that.currency) {
        "cannot add Money values of different currencies"
    }
    return Money(this.amount + that.amount, currency)
}
```

Наш код продолжает компилироваться. Как такое возможно?

Стандартная библиотека Kotlin включает функции-расширения, которые определяют операторы для классов в стандартной библиотеке Java: математические классы — например, BigInteger и BigDecimal, и коллекции — например, List<T> или Set<T>. Поскольку эти функции-расширения определены в пакете kotlin, они автоматически доступны для любого пакета — нам не нужно их импортировать.

Соглашения для аннотаций значений

Статические функции of объекта-компаньона, используемые для указания значений Money, также нарушают соглашения Kotlin.

Синтаксис Java различает создание экземпляра класса с помощью оператора new и получение объекта в результате вызова метода. Современное соглашение Java заключается в том, что объекты с сохранением состояния, для которых важна идентификация, создаются с помощью оператора new, а значения указываются при помощи вызова статических фабричных функций. Например, выражение new ArrayList<>() создает новый мутирующий список, отличный от любого другого изменяемого списка, в то время как выражение List.of("a","b","c") определяет немутирующее значение списка.

Kotlin не проводит различия между созданием объектов и вызовом функций — синтаксис для создания экземпляра класса такой же, как и для вызова функции.

Также не существует соглашений о кодировании, позволяющих проводить различие между созданием нового объекта с сохранением состояния, который имеет четкую идентификацию, и указанием значений, которые этого не делают.

Хотя код Kotlin для вызова функции и создания экземпляра класса выглядит одинаково, он реализован при помощи разного байт-кода JVM. Совместимое с исходным кодом изменение между вызовом конструктора и функции не будет совместимым с двоичным кодом.

Там, где классу требуется несколько фабричных функций, как это делается в нашем классе `Money`, они обычно определяются как функции верхнего уровня, а не в объекте-компаньоне класса. IntelliJ вносит свой вклад в то, чтобы подтолкнуть человека к этому стилю, — он намного лучше справляется с автоназначением функций верхнего уровня, чем методов объекта-компаньона.

Итак, было бы более общепринятым, если бы мы создали экземпляры `Money` с помощью таких выражений, как `Money(...)` или, в качестве альтернативы, `moneyOf(...)`, нежели `Money.of(...)`.

Как показано в *главе 3*, класс `Money` не является классом данных и имеет закрытый (`private`) конструктор, обеспечивающий сохранение взаимосвязи между валютой класса и точностью его значения `amount`. Таким образом, самым простым вариантом было бы определить функцию верхнего уровня `moneyOf` из того же файла-источника, что и класс `Money`. Однако функции `moneyOf` должны будут вызывать конструктор класса `Money`. А они не могут вызвать его, если он все еще объявлен как `private`, но смогут, если мы изменим конструктор на `internal`.

Внутренняя видимость сделала бы конструктор видимым для любого кода Kotlin в том же модуле компиляции (подпроекте Gradle или модуле IntelliJ), но предотвратила бы его вызов кодом Kotlin в других блоках компиляции. В результате модуль компиляции, а не класс, стал бы отвечать за обеспечение инвариантов класса `Money`, никогда не вызывая его конструктор ненадлежащим образом. Это было бы достаточно безопасно, если бы не те части нашей системы Java, которые мы будем продолжать поддерживать во время перехода системы на Kotlin.

Java и JVM не имеют понятия внутренней видимости. Компилятор Kotlin преобразует внутренние функции класса в общедоступные функции в файлах классов JVM, которые он генерирует, и записывает внутреннюю видимость в виде дополнительных метаданных, которые обрабатываются компилятором Kotlin, но игнорируются компилятором Java. В результате функции Kotlin, объявленные как внутренние, окажутся общедоступными для компилятора Java и JVM, что позволит нам случайно создавать недопустимые значения `Money`, когда мы работаем в проекте с кодом Java. Это делает функции верхнего уровня `moneyOf` непривлекательным вариантом.

Вместо этого мы можем снова опереться на перегрузку оператора Kotlin. Если мы определим оператор вызова функции для объекта-компаньона класса `Money`, код Kotlin сможет создавать значения `Money`, используя тот же синтаксис, как если бы они вызывали конструктор напрямую:

```
val currently = Money.of(BigDecimal("9.99"), GBP))

val proposal = Money(BigDecimal("9.99"), GBP))
```

Однако на самом деле это не будет вызовом конструктора — в сокращенном виде это вот что:

```
val proposal = Money.Companion.invoke(BigDecimal("9.99", GBP))
```

При переименовании метода `add` в `plus` обнаружилось, что, если попытаться достичь этого путем простого переименования `of` в `invoke`, мы получим для Java-кода эффект домино. Код Java, который создает значения `Money`, изменяется с написания:

```
Money.of(BigDecimal(100), EUR)
```

на

```
Money.invoke(BigDecimal(100), EUR)
```

Методы `of` имели две обязанности: обеспечить соблюдение инвариантов класса при построении значений `Money` и предоставить в вызывающем объекте «синтаксический сахар», который соответствует современным соглашениям Java для обозначения значений. Переименование `of` в `invoke` не влияет на первое, но портит второе.

Мы можем использовать ту же комбинацию метода извлечения и реорганизовать вызов извлеченного метода и встроенного метода, чтобы избежать каких-либо негативных последствий для нашего кода Java, поскольку мы рефакторим наш код Kotlin, чтобы следовать соглашениям Kotlin.

Сначала извлеките все тело метода `of` в виде метода с именем `invoke` (пример 12.10).

Пример 12.10 [operators.12:src/main/java/travelator/money/Money.kt]

```
class Money private constructor(
    val amount: BigDecimal,
    val currency: Currency
) {
    ...

    companion object {
        @JvmStatic
        fun of(amount: BigDecimal, currency: Currency) =
            invoke(amount, currency)

        private fun invoke(amount: BigDecimal, currency: Currency) =
            Money(
                amount.setScale(currency.defaultFractionDigits),
                currency
            )

            ... и удобные способы перезагрузки операторов
    }
}
```

Затем сделайте invoke общедоступным операторным методом (пример 12.11).

Пример 12.11 [operators.13:src/main/java/travelator/money/Money.kt]

```
@JvmStatic
fun of(amount: BigDecimal, currency: Currency) =
    invoke(amount, currency)

operator fun invoke(amount: BigDecimal, currency: Currency) =
    Money(
        amount.setScale(currency.defaultFractionDigits),
        currency
    )
```

Теперь мы можем вызвать объект-компаньон Money в качестве функции, которая выглядит как конструктор. Так почему же вызов Money(...) в теле метода invoke не переполняет стек вызовов? Да потому что вызов Money(...) внутри метода invoke не является рекурсивным вызовом invoke, а на самом деле вызывает закрытый конструктор Money. Вне класса обращение к Money(...) вызывает метод invoke объекта-компаньона, поскольку закрытый конструктор не виден. Так что у нас есть лучшее из обоих миров: обычный синтаксис для создания экземпляров класса и граница инкапсуляции, которая гарантирует инварианты класса.

Чтобы заставить существующий код Kotlin использовать новый синтаксис, нам нужно сначала научить метод of объекта-компаньона вызывать себя как функцию (пример 12.12).

Пример 12.12 [operators.14:src/main/java/travelator/money/Money.kt]

```
@JvmStatic
fun of(amount: BigDecimal, currency: Currency) =
    this(amount, currency)
```

Затем мы заменяем встроенным выражением метод of, чтобы встроить его в код Kotlin. Опять же, Java-код не будет затронут, и когда окажется, что ни один фрагмент Java-кода не вызывает метод of, IDE удалит его.

Перед рефакторингом по замене встроенными выражениями код Kotlin, который создает значения Money, выглядит следующим образом (пример 12.13).

Пример 12.13 [operators.16:src/main/java/travelator/money/ExchangeRates.kt]

```
interface ExchangeRates {
    fun rate(fromCurrency: Currency, toCurrency: Currency): BigDecimal

    @JvmDefault
    fun convert(fromMoney: Money, toCurrency: Currency): CurrencyConversion {
        val rate = rate(fromMoney.currency, toCurrency)
```

```
        val toAmount = fromMoney.amount * rate
        val toMoney = Money.of(toAmount, toCurrency)
        return CurrencyConversion(fromMoney, toMoney)
    }
}
```

А после рефракторинга по замене встроенными выражениями он выглядит так (пример 12.14).

Пример 12.14 [operators.17:src/main/java/travelator/money/ExchangeRates.kt]

```
interface ExchangeRates {
    fun rate(fromCurrency: Currency, toCurrency: Currency): BigDecimal

    @JvmDefault
    fun convert(fromMoney: Money, toCurrency: Currency): CurrencyConversion {
        val rate = rate(fromMoney.currency, toCurrency)
        val toAmount = fromMoney.amount * rate
        val toMoney = Money(toAmount, toCurrency)
        return CurrencyConversion(fromMoney, toMoney)
    }
}
```

То есть у нас остается обычный и удобный класс — независимо от того, используем ли мы его из Kotlin или из Java.

Двигаемся дальше

Java и Kotlin имеют разные соглашения, которые работают с разными компонентами двух языков.

Мы не хотим, чтобы наше применение Kotlin оказало негативное влияние на Java или оставило код Kotlin простым кодом Java в синтаксисе Kotlin.

Используя аннотации и делегирование, мы можем гарантировать, что код Kotlin и Java будут следовать соответствующим языковым соглашениям при переходе на Kotlin. Комбинация извлечения и рефакторинга с заменой встроенными выражениями позволяет легко добавлять код в нашу кодовую базу и удалять его, когда в нем больше нет необходимости.

От потоков к итерируемым объектам и последовательностям

Java и Kotlin позволяют нам преобразовывать и сокращать коллекции. Однако у них разные цели проектирования и реализации. Что Kotlin использует вместо потоков Java, когда мы должны конвертировать их под Kotlin, и как?

Потоки в Java

Java 8 представила *потоки* (streams) в 2014 году, эффективно используя новые лямбда-выражения. Допустим, мы хотим вычислить среднюю длину некоторых строк, учитывая, что строки, содержащие только пробелы, следует обрабатывать так, как если бы они были пустыми. Ранее мы могли написать:

```
public static double averageNonBlankLength(List<String> strings) {
    var sum = 0;
    for (var s : strings) {
        if (!s.isBlank())
            sum += s.length();
    }
    return sum / (double) strings.size();
}
```

С помощью потоков Java мы можем выразить этот алгоритм через `filter`, `map` и `reduce` путем конвертации `List` в `Stream` и применения преобразований:

```
public static double averageNonBlankLength(List<String> strings) {
    return strings
        .stream()
        .filter(s -> !s.isBlank())
        .mapToInt(String::length)
        .sum()
        / (double) strings.size();
}
```

Вместо того чтобы мысленно запускать цикл `for`, чтобы представить, что делает этот код, можно рассмотреть шаги алгоритма, объявленные построчно, и полагаться на среду выполнения для реализации этих шагов за нас.

Если мы действительно спешим получить эти результаты, мы можем даже написать:

```
public static double averageNonBlankLength(List<String> strings) {
    return strings
        .parallelStream() ❶
        .filter(s -> !s.isBlank())
        .mapToInt(String::length)
        .sum()
        / (double) strings.size();
}
```

❶ `parallelStream` разделит работу на несколько потоков.

Здесь выполняются различные типы фундаментальных операций: `map` изменяет типы элементов, но не их количество, `filter` сохраняет или отклоняет элементы в зависимости от некоторого свойства, но сохраняет их тип неизменным, а `sum` сводит коллекции к одному свойству. В этом примере не показаны операции `skip(n)` и `limit(n)` — обратные потоки без первого и последнего элемента `n` соответственно.

Потоки Java *отложены* (lazy, «ленивы»): код `strings.filter(...).mapToInt(...)` ничего не делает, кроме настройки конвейера для некоторой терминальной операции, в нашем случае `sum`, которая нужна, чтобы пропускать через себя переменные. *Отложенность* (Laziness) означает, что более поздние этапы конвейера могут ограничить объем работы, которую приходится выполнять на более ранних этапах. Рассмотрим перевод списка слов, закончив его, когда дело дойдет до слова `STOP`. Версия с циклом может выглядеть следующим образом:

```
public static List<String> translatedWordsUntilSTOP(List<String> strings) {
    var result = new ArrayList<String>();
    for (var word: strings) {
        String translation = translate(word);
        if (translation.equalsIgnoreCase("STOP"))
            break;
        else
            result.add(translation);
    }
    return result;
}
```

Выходя из цикла, мы переводим не все слова, а только тот минимум, который нам нужен. В Java 9 были внедрены методы `dropWhile` и `takeWhile`, которые позволяют выразить это так:

```
public static List<String> translatedWordsUntilSTOP(List<String> strings) {
    return strings
        .stream()
        .map(word -> translate(word))
        .takeWhile(translation -> !translation.equalsIgnoreCase("STOP"))
        .collect(toList());
}
```

Это работает, поскольку `collect` вызывает процесс проникновения значений через конвейер, а `takeWhile` перестает вытягивать данные от своего предшественника, когда его предикат возвращает `false`.

Что касается предмета проникновения, потоки для небольших коллекций могут быть удивительно медленными. Они отлично подходят для крупномасштабного анализа данных, когда мы хотим использовать все доступные ядра для решения проблемы, но не так хороши для суммирования стоимости пяти товаров в корзине покупок. Проблема в том, что потоки Java были разработаны для обеспечения общих преобразований коллекции, отложенной оценки *и* параллельной обработки и к ним предъявляются разные требования. Kotlin не пытается реализовать параллельные операции, оставляя две абстракции: итераторы хороши для преобразования и сокращения коллекций, а последовательности дают отложенную оценку.

Итераторы в Kotlin

Вместо создания нового интерфейса для определения операций с коллекциями Kotlin предоставляет функции-расширения для `Iterable`. Простейшим выражением Kotlin того же алгоритма может быть следующий код:

```
fun averageNonBlankLength(strings: List<String>): Double =
    (strings
        .filter { it.isNotBlank() }
        .map(String::length)
        .sum()
        / strings.size.toDouble())
```

Здесь функцией-расширением для `Iterable` служит `filter`. В отличие от `Stream.filter`, возвращающей другой аргумент `Stream`, в Kotlin `filter` возвращает список `List` (который является `Iterable`, что дает нам возможность продолжить цепочку). Функция `map` тоже возвращает `List`, так что это однострочное выражение создает два дополнительных списка в памяти.

Первый список `List` — для непустых строк, а второй список `List` содержит длину этих строк. Когда (и если) мы заботимся о производительности, это может стать проблемой, потому что заполнение обоих этих списков потребует времени и памяти для их поддержки.

Список `List`, содержащий длины, — это особая проблема, потому что целые числа должны быть заключены в *коробку* (обернуты в объект `Integer`), чтобы стать пригодными для включения в список. В пример потоков Java для ухода от этой проблемы мы использовали `mapToInt(String::length)`. Для предотвращения необходимости упаковывать и распаковывать потоки были созданы `IntStream` (и `LongStream`, и `Double Stream`, но, что любопытно, не `BooleanStream` или `CharStream`), однако вы должны помнить о необходимости использовать их и учитывать, что `IntStream` не является `Stream<Integer>`.

Должны ли мы заботиться о производительности? В основном нет — этот код Kotlin будет быстрым, *пока* у нас нет объемных коллекций, в отличие от потоков,

которые быстры только в том случае, *если* у нас большие коллекции. Когда же мы столкнемся с действительно объемными коллекциями, то сможем переключиться на последовательности.

Последовательности в Kotlin

Абстракция Kotlin `Sequence` предлагает ту же ленивую (отложенную) оценку, что и потоки Java. Операция `map` над `Sequence` возвращает другую `Sequence` — операции в цепочке выполняются только тогда, когда какая-либо терминальная операция требует их оценки. Если у нас есть `Collection`, `Iterable` или даже `Iterator`, значит, есть и функция-расширение `asSequence()` для преобразования. После этого API становится подозрительно знакомым:

```
fun averageNonBlankLength(strings: List<String>): Double =
    (strings
        .asSequence()
        .filter { it.isNotBlank() }
        .map(String::length)
        .sum()
        / strings.size.toDouble())
```

Знакомство это подозрительно тем, что все присутствующие здесь операции (`filter`, `map`, `sum`) теперь являются расширениями не для `Iterable`, а для `Sequence`. Они возвращают не список `List`, а другую последовательность `Sequence`. За исключением `sum`, которая даже не может делать вид, что выполняет свою работу, не прочитав все данные, так что `sum` — это терминальная операция. Приведенный здесь код читается так же, как и его итеративная версия, но каждая из его функций фактически от присутствующих в итеративной отличается.

Замена итераций и последовательностей

Последовательности `Iterable<T>` и `Sequence<T>` имеют одну и ту же сигнатуру единого метода: `public operator fun iterator(): Iterator<T>`. Обе они содержат функции-расширения для `map`, `filter`, `reduce` и т. д., которые принимают те же параметры. Но они не относятся к одн(му и тому же типу, потому что их семантика очень разная. Операции над `Iterable` выполняются сразу же, в то время как операции над `Sequence` могут быть отложенными, поэтому мы не можем безнаказанно менять одно на другое (как позже будет показано в этой главе).

Тем не менее тот факт, что у них есть такие похожие API, означает, что в подобных ситуациях нам придется изменить очень мало исходного кода, когда захотим перейти с одного на другой.

Версия с последовательностями `averageNonBlankLength` не будет окупать затраты на создание промежуточных списков для хранения результатов каждого этапа, но для небольшого количества элементов затраты на настройку и выполнение конвейера могут быть выше, чем на создание списков. В этом случае длины `Int` по-прежнему будут упакованы как `Integer`, хотя и по одному за раз, а не путем создания из них целого списка. Во многих случаях разработчики API предоставят разумное решение для распаковки. В таком случае `sumBy` будет выглядеть так:

```
fun averageNonBlankLength(strings: List<String>): Double =
    (strings
        .asSequence()
        .filter { it.isNotBlank() }
        .sumBy(String::length)
        / strings.size.toDouble())
```

Функция `sumBy` (так же доступная, как расширение для `Iterable`) избегает упаковки, принимая функцию, которая возвращает `Int`. Она может это сделать, потому что `sumBy` — операция терминальная, поскольку не возвращает другую последовательность или коллекцию.

Множественные итерации

Если вы используете потоки Java, то, вероятно, пытались сделать примерно так:

```
public static double averageNonBlankLength(List<String> strings) {
    return averageNonBlankLength(strings.stream());
}

public static double averageNonBlankLength(Stream<String> strings) {
    return strings
        .filter(s -> !s.isBlank())
        .mapToInt(String::length)
        .sum()
        / (double) strings.count();
}
```

Это выглядит очень правдоподобно: мы только что извлекли функцию, взяв параметр `Stream` вместо использования `List`. У последовательности `Stream` нет свойства `size`, но функция `count()` дает тот же результат, поэтому мы ее и применяем. Однако, запустив этот код на выполнение, мы получаем ошибку:

```
java.lang.IllegalStateException: stream has already been operated upon or closed.
```

Проблема в том, что у `Stream` есть скрытое состояние. Как только мы используем все его элементы (и `sum` делает именно это), мы не сможем снова обойти их, чтобы реализовать `count`. Несмотря на то что `sum` на самом деле является методом `IntStream`, каждый поток в конвейере задействует своего предшественника, поэтому входные строки поглощаются функцией `sum`.

В Java этого достаточно, чтобы отвлечь нас от извлечения операций `Stream` в функции. Давайте попробуем проделать то же самое с `Sequence` в Kotlin:

```
fun averageNonBlankLength(strings: List<String>): Double =
    averageNonBlankLength(strings.asSequence())

fun averageNonBlankLength(strings: Sequence<String>): Double =
    (strings
        .filter { it.isNotBlank() }
```

```
    .sumBy(String::length)
    / strings.count().toDouble())
```

В Kotlin мы можем вызвать версию `Sequence` из версии `List`, и все в порядке... пока.

Однако мы накапливаем неприятности. Чтобы понять почему, давайте перейдем к другому слою и добавим функцию, которая принимает `Iterator`:

```
fun averageNonBlankLength(strings: Iterator<String>): Double =
    averageNonBlankLength(strings.asSequence())
```

Если мы вызовем эту функцию, то получим такую ошибку:

```
java.lang.IllegalStateException: This sequence can be consumed only once
```

(сравнивая ее с ошибкой потоков, мы подмечаем, что разработчики Kotlin кажутся более грамматически педантичными, чем разработчики JVM). Теперь `Sequence` действует как `Stream` в Java, хотя раньше это было не так. Что изменилось?

Оказывается, некоторые последовательности можно безопасно повторять несколько раз — например, те, которые поддерживаются коллекцией, хранящейся в памяти. А другие нельзя. Сейчас, когда наша последовательность `Sequence` предоставлена итератором `Iterator`, первый прогон (для расчета `sum`) продолжается до тех пор, пока `Iterator.hasNext()` не вернет `false`. Если бы мы попытались снова запустить выполнение через `Sequence` (для `count`), состояние `Iterator` не изменилось бы, и функция `hasNext()` немедленно вернула бы `false`. Это привело бы к тому, что функция `strings.count()` вернула бы 0, в результате чего `mediumNonBlankLength` всегда стала бы возвращать `Infinity` (если есть входящие данные).

Такого рода поведение, э-э, *нежелательно*, поэтому итераторы, обертывающие последовательности, намеренно ограничиваются `Sequence.constrainOnce()`, чтобы это предотвратить. Это та функция `constrainOnce()`, которая выдает исключение `IllegalStateException`, если мы пытаемся выполнить код дважды.

Другим каноническим примером последовательности `Sequence`, которая не может быть использована больше одного раза, является та, что поддерживается чтением из внешнего источника, — такого как файл или сетевой сокет. В подобных случаях мы, как правило, не можем просто вернуться назад и воспроизвести ввод для повторной итерации. К сожалению, разница между двумя типами `Sequence` не отражена в системе типов, поэтому мы обнаружим любую несовместимость между нашим алгоритмом и нашими входными данными только во время выполнения. Как мы увидим в *главе 20*, это усугубляется распространенной техникой использования `sequenceOf(...)` или `List.asSequence()` в качестве тестовых данных. Эти последовательности поддерживают несколько итераций и не будут предупреждать нас о проблеме.

На практике такая проблема обычно вызывает лишь раздражение из-за некоторой потери времени на переработку кода. И она не возникнет, если вы конвертируете код из потоков, потому что в первую очередь она проявляется не в этих случаях, а, скорее, при применении `Sequence` с нуля или преобразовании из `Iterable`.

В нашей конкретной ситуации мы можем заставить все работать, ведя подсчет элементов по мере их прохождения в первой итерации вместо того, чтобы снова считать их в конце:

```
fun averageNonBlankLength(strings: Sequence<String>): Double {
    var count = 0
    return (strings
        .onEach { count++ }
        .filter { it.isNotBlank() }
        .sumBy(String::length)
        / count.toDouble())
}
```

Это первая проблема с мутирующей локальной переменной, которую мы решили в нашей книге! Мы можем скрыть свой стыд внутри более общего полезного служебного класса `CountingSequence`:

```
class CountingSequence<T>(
    private val wrapped: Sequence<T>
) : Sequence<T> {
    var count = 0
    override fun iterator() =
        wrapped.onEach { count++ }.iterator()
}

fun averageNonBlankLength(strings: Sequence<String>): Double {
    val countingSequence = CountingSequence(strings)
    return (countingSequence
        .filter { it.isNotBlank() }
        .sumBy(String::length)
        / countingSequence.count.toDouble())
}
```

Это повторяющаяся тема в алгоритмах Kotlin: иногда нам может потребоваться прибегнуть к мутации, чтобы реализовать что-то разумным или эффективным способом, но обычно мы можем скрыть мутацию таким образом, чтобы уменьшить ее видимость и создать полезную абстракцию. В нашем случае этому способствует тот факт, что `Sequence` — это интерфейс только с одним методом, что позволяет очень легко реализовать его самостоятельно. Java `Stream` также представляет собой интерфейс, но имеющий 42 метода и не включающий класс `AbstractStream`, обеспечивающий реализации по умолчанию!

Прежде чем покинуть этот раздел, задумаемся о том, что вы, возможно, молча кипели от злости с тех пор, как мы представили `Stream.count()`. Если нет, можете ли вы понять, в чем проблема?

Одним из преимуществ `Stream` и `Sequence` является то, что они позволяют нам работать с произвольно большими наборами данных, а определение размера этих наборов данных путем их индивидуального подсчета не очень эффективно, даже если иногда это можно сделать. В общем, даже если мы сможем на практике повторить

Sequence более одного раза, это, скорее всего, будет неэффективно как раз в тех случаях использования, которые заставили нас к Sequence обратиться.

Повторяйте последовательность только один раз

Как правило, если работа ведется с Sequence, наши алгоритмы должны завершаться за один проход. Так они смогут работать с последовательностями, которые не поддерживают многократную итерацию, и могут быть эффективными с большим количеством элементов.

Мы можем использовать Sequence.constrainOnce() в наших тестах, чтобы убедиться, что мы снова случайно не пойдем на следующий проход.

Выбор между потоками, итераторами и последовательностями

Если у нас уже есть код, использующий потоки Java, он продолжит отлично работать на JVM, даже при преобразовании в Kotlin. Это будет выглядеть немного приятнее, т. к. Kotlin может переместить лямбду за пределы метода и разрешить использование неявного лямбда-параметра it:

```
fun averageNonBlankLength(strings: List<String>): Double =
    (strings
        .stream()
        .filter { it.isNotBlank() }
        .mapToInt(String::length)
        .sum()
        / strings.size.toDouble())
```

Кроме того, мы можем использовать функции-расширения для *добавления* операций в потоки таким же образом, как Kotlin определяет свои операции Sequence.

Если наш код работает с большими коллекциями и, в частности, использует parallelStream(), то по умолчанию следует оставить потоки в покое, потому что в этих случаях они хорошо оптимизированы JVM. Стандартная библиотека Kotlin даже предоставляет расширения Stream<T>.asSequence() и Sequence<T>.asStream(), что позволит нам «сменить лошадей» в середине, э-э, Stream.

Если мы решим перейти к абстракции Kotlin, то можем выбрать Iterable или Sequence — в зависимости от того, использует ли поточный код преимущества отложенной оценки. Отложенная оценка требуется, если:

- нам нужно получить результаты до того, как мы закончим чтение входных данных;
- нам нужно обработать больше данных, чем мы можем вместить в память (включая промежуточные результаты).

Отложенная оценка может повысить производительность для:

- больших коллекций со множеством уровней конвейера, где создание промежуточных коллекций может быть медленным;

- конвейеров, где ранние этапы могут быть пропущены в зависимости от информации, доступной только на более поздних этапах.

Проиллюстрируем последний пункт на том же примере перевода, к которому мы обращались при работе с потоками:

```
public static List<String> translatedWordsUntilSTOP(List<String> strings) {
    return strings
        .stream()
        .map(word -> translate(word))
        .takeWhile(translation -> !translation.equalsIgnoreCase("STOP"))
        .collect(toList());
}
```

Мы можем преобразовать это в эквивалентное повторяющееся выражение:

```
fun translatedWordsUntilSTOP(strings: List<String>): List<String> =
    strings
        .map { translate(it) }
        .takeWhile { !it.equals("STOP", ignoreCase = true) }
```

Но тогда *все* слова во входных данных `List` будут переведены в другой `List` с помощью `map` — даже находящиеся после `STOP`. Использование `Sequence` позволяет избежать перевода слов, которые мы не собираемся возвращать:

```
fun translatedWordsUntilSTOP(strings: List<String>): List<String> =
    strings
        .asSequence()
        .map { translate(it) }
        .takeWhile { !it.equals("STOP", ignoreCase = true) }
        .toList()
```

Если нам не нужна отложенная оценка, а также для небольших коллекций или при написании с нуля в Kotlin, конвейеры `Iterable` просты, как правило, быстры и легко понятны. Ваши авторы часто преобразуют потоки в итераторы, чтобы воспользоваться гораздо более богатым API, предоставляемым Kotlin. Если итераторы оказываются слишком медленными (или иногда слишком «жадными» до памяти) с объемными коллекциями, то мы можем конвертировать их в последовательности. Если этого все еще недостаточно, мы можем перейти (надеемся, не обратно) к потокам и, возможно, даже воспользоваться преимуществами параллелизма.

Алгебраическая трансформация

Отложенность и параллелизм, конечно, повлияют на то, *когда* будут вызваны этапы нашего конвейера. Когда какой-либо из наших алгоритмов зависит от порядка операций, он может быть нарушен, если мы поменяем местами потоки, итераторы и последовательности. То, что нам нужно, — это код с предсказуемой *алгеброй*: набором правил для управления операциями при сохранении поведения.

В *главе 7* было показано, что мы можем классифицировать функции (на самом деле любой код, включая лямбда-выражения) в зависимости от того, зависят ли они от

времени их запуска. Вычисления (см. *разд. «Вычисления» главы 7*) безопасны для рефакторинга, потому что мы можем перемещать их вызовы, не влияя на результат вычислений или результат любого другого кода. Напротив, перемещение действия (см. *разд. «Действия» главы 7*) из итераторов в последовательность или наоборот может изменить результат при его вызове и, следовательно, результат работы нашей программы. Чем больше нашего кода выражено в виде вычислений, тем больше мы можем рассматривать его представление как нечто, что можно преобразовать в соответствии с правилами.

Мы также можем применить другую алгебру — арифметику, для упрощения нашего определения `average NonBlankLength`. В настоящее время:

```
class CountingSequence<T>(
    private val wrapped: Sequence<T>
) : Sequence<T> {
    var count = 0
    override fun iterator() =
        wrapped.onEach { count++ }.iterator()
}

fun averageNonBlankLength(strings: Sequence<String>): Double {
    val countingSequence = CountingSequence(strings)
    return (countingSequence
        .filter { it.isNotBlank() }
        .sumBy(String::length)
        / countingSequence.count.toDouble())
}
```

Все эти сложности возникают из-за того, что нам нужно не простое среднее значение, а среднее значение, в котором пробельные строки считаются пустыми. Одним из способов сделать это является фильтрация пробелов в сумме, а не в счете. Математически, однако, это равнозначно следующему:

```
fun averageNonBlankLength(strings: Sequence<String>): Double =
    strings
        .map { if (it.isBlank()) 0 else it.length }
        .average()
```

Такая математическая перестановка, как и в случае с нашим рефакторингом кода, работает только в том случае, если все операции являются вычислениями. Она также опасно притягательна, потому что возвращает нас к упаковыванию целых чисел, чтобы передать их в `average`.

Что нам нужно, так это `averageBy` — аналог `sumBy`. Мы можем осуществить такую замену, сопоставив определения среды выполнения Kotlin для `Sequence.sumBy` с `Sequence.average`, чтобы получить:

```
inline fun <T> Sequence<T>.averageBy(selector: (T) -> Int): Double {
    var sum: Double = 0.0
    var count: Int = 0
```

```
    for (element in this) {
        sum += selector(element)
        checkCountOverflow(++count)
    }
    return if (count == 0) Double.NaN else sum / count
}
```

Это снова сводится к мутации во имя эффективности и, наконец, позволяет нам написать:

```
fun averageNonBlankLength(strings: Sequence<String>): Double =
    strings.averageBy {
        if (it.isBlank()) 0 else it.length
    }
```

Почему мы просто не написали код таким образом с самого начала? Что ж, иногда мы видим эти соответствия, иногда нет! Помните, что мы начали здесь:

```
public static double averageNonBlankLength(List<String> strings) {
    var sum = 0;
    for (var s : strings) {
        if (!s.isBlank())
            sum += s.length();
    }
    return sum / (double) strings.size();
}
```

Учитывая этот код, естественно заменить инструкцию `if` на `filter`:

```
public static double averageNonBlankLength(List<String> strings) {
    return strings
        .stream()
        .filter(s -> !s.isBlank())
        .mapToInt(String::length)
        .sum()
        / (double) strings.size();
}
```

А если бы наш исходный код был более функциональным? Вместо того чтобы использовать инструкцию `if` для принятия решения, *что* необходимо добавить, он мог бы использовать тернарное выражение для вычисления добавляемого *количества*:

```
public static double averageNonBlankLength(List<String> strings) {
    var sum = 0;
    for (var s : strings) {
        sum += s.isBlank() ? 0 : s.length();
    }
    return sum / (double) strings.size();
}
```

И тогда наше первоначальное преобразование, вероятно, было бы таким:

```
public static double averageNonBlankLength(List<String> strings) {
    return strings
        .stream()
        .mapToInt(s -> s.isBlank() ? 0 : s.length())
        .average()
        .orElse(Double.NaN);
}
```

Что ж, в этом случае у нас была бы более короткая глава, но мы узнали бы меньше.

Рефракторинг от потоков к итераторам и последовательностям

`Travelator` регистрирует операционные события во время выполнения, поэтому мы знаем, что программа работает так, как ожидается. Эти события отправляются в формате JSON на сервер индексации, который может генерировать красивые графики и оповещения, заданные с помощью собственного языка запросов. Однако каким-то образом эти милые люди из отдела маркетинга всегда задают вопросы, запросы для которых мы написать не можем...

В этих случаях мы извлекаем события с сервера и обрабатываем их локально. Запросы, маршалинг (упорядочивание) и подкачки событий скрыты за простым интерфейсом `EventStore`, который возвращает `Iterator<Map<String, Object>>`, где `Map<String, Object>` представляет собой JSON-объекты (пример 13.1).

Пример 13.1
[streams-to-sequences.0:src/main/java/travelator/analytics/EventStore.java]

```
public interface EventStore {

    Iterator<Map<String, Object>> query(String query);

    default Stream<Map<String, Object>> queryAsStream(String query) {
        Iterable<Map<String, Object>> iterable = () -> query(query);
        return StreamSupport.stream(iterable.spliterator(), false);
    }
}
```

Интерфейс содержит собственное преобразование `Iterator` в `Stream` для нашего большего удобства. (Удивительно, но в JDK нет встроенной функции преобразования.)

Вот то, что мы не смогли написать на языке запросов сервера индексации. Этот код вычисляет среднее количество взаимодействий, которые совершают клиенты для успешного завершения бронирования (пример 13.2).

Пример 13.2
[streams-to-sequences.0:src/main/java/travelator/analytics/MarketingAnalytics.java]

```
public double averageNumberOfEventsPerCompletedBooking(
    String timeRange
) {
    Stream<Map<String, Object>> eventsForSuccessfulBookings =
        eventStore
            .queryAsStream("type=CompletedBooking&timerange=" + timeRange)
            .flatMap(event -> {
                String interactionId = (String) event.get("interactionId");
                return eventStore.queryAsStream("interactionId=" + interactionId);
            });
    Map<String, List<Map<String, Object>>> bookingEventsByInteractionId =
        eventsForSuccessfulBookings.collect(groupingBy(
            event -> (String) event.get("interactionId"))
        );
    var averageNumberOfEventsPerCompletedBooking =
        bookingEventsByInteractionId
            .values()
            .stream()
            .mapToInt(List::size)
            .average();
    return averageNumberOfEventsPerCompletedBooking.orElse(Double.NaN);
}
```

Работая над этим кодом, мы сделали все возможное, чтобы он был понятным. Мы давали имена промежуточным переменным и указывали их типы тогда и только тогда, когда это казалось полезным, и тщательно форматировали. И все равно он выглядит так, будто кто-то смахнул его на пол и сгреб обратно в надежде, что мы этого не заметим. Иногда мы оказывались совсем в проигрышной позиции — например, можно было бы извлечь функцию для упрощения кода на месте вызова, но, если не получалось дать этой функции хорошее имя, мы просто выбрасывали ее из исходного файла.

Явный или неявный тип

Иногда тип переменной важен для понимания того, как работает код. А в других случаях он просто загромождает и без того многословный блок. В этом отношении явные (explicit) типы подобны комментариям, но у них есть дополнительное преимущество, заключающееся в том, что они проверяются и применяются компилятором. Как и в случае с комментариями, мы должны попытаться написать код, который не нуждается в явных типах переменных. Хорошее именование способно помочь, как и рефакторинг, в функции, где может быть отображен возвращаемый тип.

Однако, если так не получается, нет ничего постыдного в том, чтобы показывать тип переменных, если это улучшает читаемость кода, и мы, безусловно, должны предпочесть взаимодействие в типах, а не в комментариях.

Так что сейчас мы собираемся преобразовать этот код в Kotlin, будучи воодушевлены надеждой, что наш любимый язык позволит нам выполнить эту работу лучше. Вот результат автоматического преобразования (пример 13.3).

Пример 13.3
[streams-to-sequences.1:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
fun averageNumberOfEventsPerCompletedBooking(
    timeRange: String
): Double {
    val eventsForSuccessfulBookings = eventStore
        .queryAsStream("type=CompletedBooking&timerange=$timeRange")
        .flatMap { event: Map<String?, Any?> ->
            val interactionId = event["interactionId"] as String?
            eventStore.queryAsStream("interactionId=$interactionId")
        }
    val bookingEventsByInteractionId = eventsForSuccessfulBookings.collect(
        Collectors.groupingBy(
            Function { event: Map<String, Any> ->
                event["interactionId"] as String?
            }
        )
    )
    val averageNumberOfEventsPerCompletedBooking = bookingEventsByInteractionId
        .values
        .stream()
        .mapToInt { obj: List<Map<String, Any>> -> obj.size }
        .average()
    return averageNumberOfEventsPerCompletedBooking.orElse(Double.NaN)
}
```

На момент подготовки книги конвертер Java в Kotlin оказался не так умен, как мог бы быть, сопоставляя лямбды на двух языках. Это особенно заметно в поточном коде, потому что именно там можно найти большинство Java-лямбд. Практически все такие проблемы можно снять, нажав на сомнительном коде комбинацию клавиш <Alt>+<Enter> и приняв быстрое исправление. Давайте начнем с приведения в порядок возможности обнуления, удаления рудиментарной `Function` и упрощения этой уродливой лямбды `mapToInt` (пример 13.4).

Пример 13.4
[streams-to-sequences.2:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
fun averageNumberOfEventsPerCompletedBooking(
    timeRange: String
): Double {
    val eventsForSuccessfulBookings = eventStore
        .queryAsStream("type=CompletedBooking&timerange=$timeRange")
```

```
        .flatMap { event ->
            val interactionId = event["interactionId"] as String
            eventStore.queryAsStream("interactionId=$interactionId")
        }
    val bookingEventsByInteractionId = eventsForSuccessfulBookings.collect(
        groupingBy { event -> event["interactionId"] as String }
    )
    val averageNumberOfEventsPerCompletedBooking = bookingEventsByInteractionId
        .values
        .stream()
        .mapToInt { it.size }
        .average()
    return averageNumberOfEventsPerCompletedBooking.orElse(Double.NaN)
}
```

Код Java перед преобразованием смешал некоторые явно типизированные переменные старого стиля — например, `Stream<Map<String, Object>>` с неявным `var averageNumberOfEventsPerCompletedBooking`. Преобразование отбросило явные типы. Это, без сомнения, пугает меньше, но и менее понятно, если нас действительно интересует, как программа делает то, что она делает. Мы пока оставим все как есть, но пересмотрим наше решение до того, как закончим.

Итак, к настоящему моменту у нас есть код Kotlin, использующий потоки Java и работающий просто отлично, так что мы могли бы оставить его в покое. `Travelator` пользуется огромным успехом, обеспечивая множество полноценных бронирований в день, а потоки — хороший выбор для пропускной способности, так зачем переходить на Kotlin?

Однако, покупая эту книгу, вы надеялись найти в ней иной подход к таким ситуациям, поэтому мы продолжим двигаться вперед, замеряя производительность на каждом этапе, и остановимся, когда увидим, что она значительно снизилась.

Сначала итераторы

Глядя на код, мы видим, что его работа состоит из двух этапов. На первом этапе обрабатываются входные данные неопределенной длины и создается коллекция в памяти (пример 13.5).

Пример 13.5
[streams-to-sequences.2:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
val eventsForSuccessfulBookings = eventStore
    .queryAsStream("type=CompletedBooking&timerange=$timeRange")
    .flatMap { event ->
        val interactionId = event["interactionId"] as String
        eventStore.queryAsStream("interactionId=$interactionId")
    }
```

```
val bookingEventsByInteractionId = eventsForSuccessfulBookings.collect(
    groupingBy { event -> event["interactionId"] as String }
)
```

На втором этапе эта коллекция обрабатывается (пример 13.6).

Пример 13.6
[streams-to-sequences.2:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
val averageNumberOfEventsPerCompletedBooking = bookingEventsByInteractionId
    .values
    .stream()
    .mapToInt { it.size }
    .average()
return averageNumberOfEventsPerCompletedBooking.orElse(Double.NaN)
```

Как было показано ранее, Java для обоих этих случаев использует потоки, тогда как в Kotlin мы склонны задействовать `Sequence` для обработки входных данных неизвестной длины и `Iterable` для обработки данных в памяти. Действовать с данными в памяти проще, поэтому сначала преобразуем `averageNumberOfEventsPerCompletedBooking`.

До тех пор пока в IntelliJ не добавят автоматический рефакторинг, нам придется выполнять такие процедуры вручную. В другой ситуации мы использовали бы тесты, чтобы реализовать все это безопасным способом. Но у нас имеется быстро развивающийся и произвольный аналитический код, так что можно попытаться «срезать углы». Впрочем, прежде чем начать непосредственно рефакторинг, мы напишем быстрый тест, который связывается с производством и показывает, что результат за вчерашний день был 7.44. Теперь мы можем начать рефакторинг, проверяя, что этот результат не меняется.

Нам известно, что в Kotlin есть возможность применять операции с коллекциями непосредственно к `Map.values` (те, что для `Iterable`) — т. е. можно удалить `.stream()`. Известно также, что `average()` является операцией для `IntStream` в Java, но Kotlin удобно объявляет `Iterable<Int>.average()` — следовательно, нам не требуется `mapToInt`, а только `map`. Наконец, `IntStream.average()` возвращает пустой `OptionalDouble`, если поток не имеет элементов. `Iterable<Int>.average()` в Kotlin возвращает `NaN` (не число), а это означает, что мы можем использовать результат преобразования напрямую (пример 13.7).

Пример 13.7
[streams-to-sequences.3:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
val averageNumberOfEventsPerCompletedBooking = bookingEventsByInteractionId
    .values
    .map { it.size }
    .average()
return averageNumberOfEventsPerCompletedBooking
```

Тем не менее было ли это хорошей заменой?

В нашем коде мы сейчас создаем промежуточный `List<Int>`, для которого вызываем `average()`. Это приведет к упаковыванию каждого значения, но на этот раз у нас нет `averageBy()` (чем была `sumBy()` в предыдущем примере), чтобы это предотвратить.

Станет ли этот код работать лучше или хуже, чем потоковая версия, будет зависеть от количества значений в `Map`, от того, как наша конкретная JVM оптимизирует упаковывание, и насколько сильно HotSpot оптимизировала этот путь. Узнаем мы это, только измерив все в реальных условиях. Если нам придется выбрать обобщенное решение, то следует написать свою собственную коллекцию `Collection.averageBy`. Тогда мы сможем использовать знание размера `Collection`. Мы могли бы задействовать ту коллекцию, которую подготовили ранее в этой главе (хотя и для `Sequence`), или провести рефакторинг отсюда. Давайте проведем рефакторинг отсюда, извлекая `values` и применяя `sumBy()` (пример 13.8).

Пример 13.8
[streams-to-sequences.4:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
val values = bookingEventsByInteractionId.values
return values.sumBy { it.size } / values.size.toDouble()
```

Теперь применим опцию **Извлечь функцию** (Extract Function) `averageBy` для возвращаемого выражения (примеры 13.9 и 13.10).

Пример 13.9
[streams-to-sequences.5:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
val values = bookingEventsByInteractionId.values
return averageBy(values)
```

Пример 13.10
[streams-to-sequences.5:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
private fun averageBy(
    values: MutableCollection<MutableList<MutableMap<String, Any>>>
): Double {
    return values.sumBy { it.size } / values.size.toDouble()
}
```

Ой! Оказывается, тип `bookingEventsByInteractionId` был намного более изменчивым, чем мы хотели. Он появился из `Collectors.groupingBy` — потоковой операции, которая в конце концов возвращает только коллекции Java. Мы изменим ее сейчас для использования `Collection` на месте `MutableCollection`. А потом применим опцию **Ввести параметр** (Introduce Parameter) с названием `selector` для лямбды (пример 13.11).

Пример 13.11
[streams-to-sequences.6:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
private fun averageBy(
    values: Collection<MutableList<MutableMap<String, Any>>>,
    selector: (MutableList<MutableMap<String, Any>>) -> Int
): Double {
    return values.sumBy(selector) / values.size.toDouble()
}
```

Здесь мы не хотим задумываться о фактическом типе элементов в `Collection`. Если выделить `MutableList<MutableMap<String, Any>>` и применить опцию **Извлечь/Ввести параметр типа** (Extract/Introduce Type Parameter), то получим следующее (пример 13.12).

Пример 13.12
[streams-to-sequences.7:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
private fun <T : MutableList<MutableMap<String, Any>>> averageBy(
    values: Collection<T>,
    selector: (T) -> Int
): Double {
    return values.sumBy(selector) / values.size.toDouble()
}
```

Этот рефакторинг достаточно умен, и мы не жалем о необходимости сообщать IntelliJ, что `T` может быть на самом деле чем угодно (удаляя ограничения типа `MutableList<MutableMap<String, Any>>`), как показано в примере 13.13.

Пример 13.13
[streams-to-sequences.8:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
private fun <T> averageBy(
    values: Collection<T>,
    selector: (T) -> Int
): Double {
    return values.sumBy(selector) / values.size.toDouble()
}
```

IntelliJ также по какой-то причине добавил тип к вызову (пример 13.14).

Пример 13.14
[streams-to-sequences.7:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
val values = bookingEventsByInteractionId.values
return averageBy<MutableList<MutableMap<String, Any>>>(values) { it.size }
```

Таким образом, мы удаляем `MutableList<MutableMap<String, Any>>` и оттуда.

Наконец, мы можем сделать `averageBy` крошечной — встроив однострочную функцию-расширение, созданную ранее (см. *главы 9* и *10*), как показано в примере 13.15.

Пример 13.15
[streams-to-sequences.9:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
inline fun <T> Collection<T>.averageBy(selector: (T) -> Int): Double =
    sumBy(selector) / size.toDouble()
```

Эта версия не содержит целых чисел и не повторяется более одного раза, так что она, вероятно, настолько эффективна, насколько мы хотели бы. Но, опять же, только измерение в наших конкретных обстоятельствах покажет это наверняка.

Обратите внимание, что когда мы ранее написали `Sequence.averageNonBlankLength`, то должны были посчитать количество элементов. Определяя `averageBy` в качестве расширения для `Collection`, а не `Iterable`, мы можем избежать скучного счетоводства, учитывая, что нам доступен запрос размера `size` для коллекций в памяти.

Последовательности

До сих пор мы преобразовывали конвейер (пайплайн) в памяти. А сейчас мы остаемся с кодом, который считывает неизвестное количество событий из `eventStore`, и хотим сохранить этот код отложенным.

Возвратимся к точке входа — там мы имеем следующее (пример 13.16).

Пример 13.16
[streams-to-sequences.9:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
fun averageNumberOfEventsPerCompletedBooking(
    timeRange: String
): Double {
    val eventsForSuccessfulBookings = eventStore
        .queryAsStream("type=CompletedBooking&timerange=$timeRange")
        .flatMap { event ->
            val interactionId = event["interactionId"] as String
            eventStore.queryAsStream("interactionId=$interactionId")
        }
    val bookingEventsByInteractionId = eventsForSuccessfulBookings.collect(
        groupingBy { event -> event["interactionId"] as String }
    )
    return bookingEventsByInteractionId.values.averageBy { it.size }
}
```

Здесь переменная `bookingEventsByInteractionId` на самом деле существует только для того, чтобы указать контрольную точку в алгоритме, — она называет промежуточное звено в надежде, что это поможет пониманию. Перемещаемая наверх функция

`eventsForSuccessfulBookings` является потоком `Stream`, следовательно, мы можем преобразовать `collect(groupingBy(...))` в Kotlin с помощью `asSequence().groupBy {...}`. Лямбда-выражение при этом остается неизменным (пример 13.17).

Пример 13.17
[streams-to-sequences.10:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
val bookingEventsByInteractionId = eventsForSuccessfulBookings
    .asSequence()
    .groupBy { event ->
        event["interactionId"] as String
    }
```

Замена одного метода на другой (или функцию-расширение) с аналогичным именем, который использует совместимый лямбда-код, является хорошим признаком того, что мы на правильном пути.

Теперь для функции `flatMap`, используемой при извлечении всех событий для любого взаимодействия с завершенным бронированием, мы имеем (пример 13.18).

Пример 13.18
[streams-to-sequences.10:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
val eventsForSuccessfulBookings = eventStore
    .queryAsStream("type=CompletedBooking&timerange=$timeRange")
    .flatMap { event ->
        val interactionId = event["interactionId"] as String
        eventStore.queryAsStream("interactionId=$interactionId")
    }
```

Это также, вероятно, «просто сработало бы» (*probably just work*™), если бы у нас были последовательности, а не потоки. К счастью, мы знаем, как конвертировать из `Stream` в `Sequence` — для этого используется расширение `.asSequence()`, предоставляемое Kotlin JDK interop. Нам нужно применить его к обоим потокам (пример 13.19).

Пример 13.19
[streams-to-sequences.11:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
val eventsForSuccessfulBookings = eventStore
    .queryAsStream("type=CompletedBooking&timerange=$timeRange")
    .asSequence()
    .flatMap { event ->
        val interactionId = event["interactionId"] as String
        eventStore
            .queryAsStream("interactionId=$interactionId")
            .asSequence()
    }
```

Удивительно, но код продолжает компилироваться и проходить наш (весьма поверхностный) тест! Он компилируется, потому что мы имеем возможность вызвать `eventsForSuccessfulBookings.asSequence()`, несмотря на то, что изменили тип `eventsForSuccessfulBookings` с потока `Stream` на `Sequence` (пример 13.20).

Пример 13.20
[streams-to-sequences.11:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
val bookingEventsByInteractionId = eventsForSuccessfulBookings
    .asSequence()
    .groupBy { event ->
        event["interactionId"] as String
    }
```

Это разрешается `Sequence.asSequence()`, которая не является операцией (no-op). Мы можем встроить `asSequence`, чтобы доказать такую возможность (пример 13.21).

Пример 13.21
[streams-to-sequences.12:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
val bookingEventsByInteractionId = eventsForSuccessfulBookings
    .groupBy { event ->
        event["interactionId"] as String
    }
```

Возвратимся к `eventsForSuccessfulBookings` — теперь у нас есть следующее (пример 13.22).

Пример 13.22
[streams-to-sequences.11:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
val eventsForSuccessfulBookings = eventStore
    .queryAsStream("type=CompletedBooking&timerange=$timeRange")
    .asSequence()
    .flatMap { event ->
        val interactionId = event["interactionId"] as String
        eventStore
            .queryAsStream("interactionId=$interactionId")
            .asSequence()
    }
```

Чего мы действительно хотели, так это чтобы `EventStore` поддерживала `queryAsSequence`. Мы можем сделать это, не изменяя ее, если введем функцию-расширение (пример 13.23).

Пример 13.23
[streams-to-sequences.12:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
fun EventStore.queryAsSequence(query: String) =
    this.queryAsStream(query).asSequence()
```

Это позволяет нам удалить вызовы `asSequence` из вызова функции (пример 13.24).

Пример 13.24
[streams-to-sequences.12:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
fun averageNumberOfEventsPerCompletedBooking(
    timeRange: String
): Double {
    val eventsForSuccessfulBookings = eventStore
        .queryAsSequence("type=CompletedBooking&timerange=$timeRange")
        .flatMap { event ->
            val interactionId = event["interactionId"] as String
            eventStore
                .queryAsSequence("interactionId=$interactionId")
        }
    val bookingEventsByInteractionId = eventsForSuccessfulBookings
        .groupBy { event ->
            event["interactionId"] as String
        }
    return bookingEventsByInteractionId.values.averageBy { it.size }
}
```

Хорошо, настало время посмотреть, что мы сделали. Мы преобразовали нашу Java в Kotlin и используем итераторы — для обработки операций в памяти и последовательности (поддерживаемые потоками в `EventStore`) — для обработки неограниченных операций. Однако мы действительно не можем утверждать, что структура алгоритма стала намного понятнее. Немного менее замусоренной, да, но вряд ли выразительной.

В настоящее время функция разделена на три части, и, если честно, они довольно условны. Иногда мы можем получить более глубокое понимание, заменив всё встроенным выражением и увидев, что у нас есть, так что давайте сделаем это (пример 13.25).

Пример 13.25
[streams-to-sequences.13:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
fun averageNumberOfEventsPerCompletedBooking(
    timeRange: String
): Double {
    return eventStore
```

```
        .queryAsSequence("type=CompletedBooking&timerange=$timeRange")
        .flatMap { event ->
            val interactionId = event["interactionId"] as String
            eventStore
                .queryAsSequence("interactionId=$interactionId")
        }.groupBy { event ->
            event["interactionId"] as String
        }.values
        .averageBy { it.size }
}
```

Похоже, что тот фрагмент, который начинается с `flatMap` и заканчивается перед `groupBy`, может быть автономным. Давайте посмотрим, как извлечь часть конвейера в его собственную функцию.

Извлечение части конвейера

Сначала выбираем код от начала конвейера и до последнего этапа, который мы хотим включить. Поэтому выделяем код от `eventStore` до `.groupBy` (но не включая его) и применяем опцию **Извлечь функцию** (Extract Function), назвав ее (в нашем случае) `allEventsInSameInteractions` (пример 13.26).

Пример 13.26
[streams-to-sequences.14:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
fun averageNumberOfEventsPerCompletedBooking(
    timeRange: String
): Double {
    return allEventsInSameInteractions(timeRange)
        .groupBy { event ->
            event["interactionId"] as String
        }.values
            .averageBy { it.size }
}

private fun allEventsInSameInteractions(timeRange: String) = eventStore
    .queryAsSequence("type=CompletedBooking&timerange=$timeRange")
    .flatMap { event ->
        val interactionId = event["interactionId"] as String
        eventStore
            .queryAsSequence("interactionId=$interactionId")
    }
```

Затем выбираем биты конвейера, которые в новой функции нам не нужны: от `eventStore` до `.flatMap`, и применяем опцию **Ввести параметр** (Introduce Parameter). Принимайте любое имя, которое выберет IntelliJ, — оно долго не проживет (пример 13.27).

Пример 13.27
[streams-to-sequences.15:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
fun averageNumberOfEventsPerCompletedBooking(
    timeRange: String
): Double {
    return allEventsInSameInteractions(
        eventStore
            .queryAsSequence("type=CompletedBooking&timerange=$timeRange")
    )
        .groupBy { event ->
            event["interactionId"] as String
        }.values
        .averageBy { it.size }
}

private fun allEventsInSameInteractions(
    sequence: Sequence<MutableMap<String, Any?>>
) = sequence
    .flatMap { event ->
        val interactionId = event["interactionId"] as String
        eventStore
            .queryAsSequence("interactionId=$interactionId")
    }
```

Это действительно некрасиво, но как только мы преобразуем параметр `sequence` из `allEventsInSameInteractions` в получатель и переформатируем код, у нас получится следующее (пример 13.28).

Пример 13.28
[streams-to-sequences.16:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
fun averageNumberOfEventsPerCompletedBooking(
    timeRange: String
): Double {
    return eventStore
        .queryAsSequence("type=CompletedBooking&timerange=$timeRange")
        .allEventsInSameInteractions()
        .groupBy { event ->
            event["interactionId"] as String
        }.values
        .averageBy { it.size }
}

fun Sequence<Map<String, Any?>>.allEventsInSameInteractions() =
    flatMap { event ->
        val interactionId = event["interactionId"] as String
```

```
        eventStore
            .queryAsSequence("interactionId=$interactionId")
    }
```

Как обсуждалось в *главе 10*, функции-расширения действительно вступают в свои права, когда мы объединяем операции в цепочки. В Java мы не смогли расширить Streams API с помощью `allEventsInSameInteractions()` и закончили тем, что разорвали цепочку, либо вызвав функцию, либо введя объясняющую переменную.

Итоговая уборка

Весь наш код еще немного громоздкий, и мы могли бы сделать его более эффективным, не создавая списков в группе, и этого, пожалуй, будет достаточно. Ну разве что за исключением создания «тонкого как бумага» псевдонима и свойства-расширения (пример 13.29).

Пример 13.29
[streams-to-sequences.17:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
typealias Event = Map<String, Any?>

val Event.interactionId: String? get() =
    this["interactionId"] as? String
```

Это позволяет нам сосредоточиться на сложных моментах, когда мы читаем окончательную версию кода (пример 13.30).

Пример 13.30
[streams-to-sequences.17:src/main/java/travelator/analytics/MarketingAnalytics.kt]

```
class MarketingAnalytics(
    private val eventStore: EventStore
) {
    fun averageNumberOfEventsPerCompletedBooking(
        timeRange: String
    ): Double = eventStore
        .queryAsSequence("type=CompletedBooking&timerange=$timeRange")
        .allEventsInSameInteractions()
        .groupBy(Event::interactionId)
        .values
        .averageBy { it.size }

    private fun Sequence<Event>.allEventsInSameInteractions() =
        flatMap { event ->
            eventStore.queryAsSequence(
                "interactionId=${event.interactionId}"
            )
        }
    }
```

```
    inline fun <T> Collection<T>.averageBy(selector: (T) -> Int): Double =
        sumBy(selector) / size.toDouble()

fun EventStore.queryAsSequence(query: String) =
    this.queryAsStream(query).asSequence()
```

Попутно отметим, что `allEventsInSameInteractions` — пример функции-расширения, выступающей в качестве метода, который мы обсуждали в *главе 10*. У нее есть доступ к `this` из `MarketingAnalytics` (для доступа к `eventStore`) и к `this` из `Sequence<Event>`.

Двигаемся дальше

Мы не собираемся утверждать, что переработанный код Kotlin в этом примере прекрасен, но мы действительно считаем, что он значительно улучшен по сравнению с оригинальной Java. Функции-расширения, лямбда-синтаксис Kotlin и улучшенный вывод типов в совокупности уменьшают значительную часть синтаксического мусора, связанного с потоками Java. Когда у нас есть коллекции в памяти, использование итераторов вместо потоков также может быть более эффективным и чистым.

ГЛАВА 14

От накопления объектов к преобразованиям

Программы на Java обычно в значительной степени зависят от изменяемого (мутирующего) состояния, потому что в Java очень сложно определить типы значений и преобразовать значения даже с помощью Streams API. Как лучше всего перевести код Java, основанный на изменяемых объектах и побочных эффектах, в код Kotlin, преобразующий неизменяемые значения?

Вычисление с использованием параметров накопления

Один из самых важных моментов, который хотят знать наши путешественники, — во что им обойдутся их приключения. Международные поездки делают такие расчеты довольно сложными. Поездка — по мере пересечения границ — будет сопряжена с расходами в нескольких валютах, но путешественник хочет иметь возможность сравнивать общие затраты, чтобы принимать решения о маршрутах и о том, где остановиться. Поэтому `Travelator` суммирует расходы в местной валюте и предпочитаемой валюте путешественника, а затем показывает общую сумму в предпочитаемой валюте. Программа делает это при помощи классов `CostSummary` и `CostSummaryCalculator`. Давайте разберемся, как они работают, а затем рассмотрим их реализацию.

У класса `Itinerary` есть операция для суммирования затрат с помощью `CostSummaryCalculator`. Она выполняется так (пример 14.1).

Пример 14.1 [accumulator.0:src/test/java/travelator/itinerary/Itinerary_CostTest.kt]

```
val fx: ExchangeRates = ...
val userCurrency = ...
val calculator = CostSummaryCalculator(userCurrency, fx) ❶

fun costSummary(i: Itinerary): CostSummary {
    i.addCostsTo(calculator) ❷
    return calculator.summarise() ❸
}
```

❶ Здесь код создает `CostSummaryCalculator` с предпочтительной валютой путешественника и источником курсов обмена валюты.

❷ Это говорит о том, что `Itinerary` должен добавить свои расходы в калькулятор. В ответ `Itinerary` добавляет стоимость своих элементов: поездки по маршруту, проживание и другие платные услуги.

❸ Это вызывает метод `summarise` калькулятора для получения `CostSummary` после того, как все расходы будут собраны.

Кто-то уже преобразовал класс `Itinerary` в Kotlin, но имплементации `addCostsTo` все еще имеют привкус Java (пример 14.2).

Пример 14.2 [accumulator.0:src/main/java/travelator/itinerary/Itinerary.kt]

```
data class Itinerary(
    val id: Id<Itinerary>,
    val route: Route,
    val accommodations: List<Accommodation> = emptyList()
) {
    ...

    fun addCostsTo(calculator: CostSummaryCalculator) {
        route.addCostsTo(calculator)
        accommodations.addCostsTo(calculator)
    }

    ...
}

fun Iterable<Accommodation>.addCostsTo(calculator: CostSummaryCalculator) {
    forEach { a ->
        a.addCostsTo(calculator)
    }
}
```

Логика здесь основана на побочных эффектах для накопления затрат в мутирующем состоянии `CostSummaryCalculator`.

Преимущество такой конструкции заключается в том, что мы можем использовать калькулятор для суммирования затрат на любой объект в нашей модели предметной области, не зная структуры этого объекта. Объект отвечает за добавление своих затрат в калькулятор и передачу калькулятора своим дочерним элементам, чтобы они могли добавлять *свои* затраты. Это отделяет код, который нуждается в затратах, от кода, который обеспечивает затраты, позволяя нам разрабатывать их независимо друг от друга.

Например, класс `Route` (маршрут) содержит список путешествий (`Journey`), каждое из которых имеет свою стоимость, а класс `Accommodation` (размещение) включает стоимость номера, количество ночей и дополнительные расходы — такие как питание и гостиничные услуги. `Itinerary` не нужно знать или заботиться о том, как структурированы эти объекты или как собирать соответствующие затраты, — это знание заключено в классах `Route` и `Accommodation`.

Однако использование мутирующего состояния имеет два существенных недостатка.

Во-первых, оно вводит возможность появления ошибок наложения имен (псевдонимов) — *aliasing errors* (**https://oreil.ly/PeqKs**). Ошибки псевдонимов создают «жуткое действие на расстоянии» (так Эйнштейн образно описал квантовую запутанность), которое не сразу видно из исходного кода. Мы приводили в *главе 6* пример, когда функция сортировала мутирующий параметр списка и прерывала его вызывающий объект.

В случае с `CostSummaryCalculator`, если мы повторно используем калькулятор для суммирования затрат нескольких объектов, то должны сбрасывать его состояние после каждого вычисления. Если мы не сбросим состояние калькулятора, затраты, собранные во время одного расчета, будут включены в следующий. Система типов не может помочь нам избежать этой ошибки.

Пример, приведенный в начале этой главы, *может* привести к этой ошибке. Калькулятор не является локальным для метода `costSummary`, и `costSummary` не сбрасывает калькулятор перед каждым вычислением. Мы не можем сказать, является ли это проблемой, просто взглянув на метод `costSummary`. Мы должны понимать, как этот метод работает в его более широком контексте, и, внося изменения в этот контекст, должны убедиться, что эти изменения не нарушают наши предположения о том, как используется метод `costSummary`.

Вторая проблема с мутирующим состоянием заключается в том, что оно разбрасывает реализацию наших алгоритмов по всему коду. Мы вернемся к этому позже в текущей главе.

Прежде чем рассмотреть `CostSummaryCalculator`, давайте взглянем на `CostSummary`, который его вычисляет (пример 14.3). Он использует `CurrencyConversion` (к счастью, уже Kotlin).

Пример 14.3
[accumulator.0:src/main/java/travelator/money/CurrencyConversion.kt]

```
data class CurrencyConversion(
    val fromMoney: Money,
    val toMoney: Money
)
```

Класс `CostSummary` представляет собой мутирующий POJO-объект (см. *главу 5*) и содержит список конвертации `CurrencyConversion` из местных валют в предпочитаемую путешественником валюту (пример 14.4).

Пример 14.4 [accumulator.0:src/main/java/travelator/itinerary/CostSummary.java]

```
public class CostSummary {
    private final List<CurrencyConversion> lines = new ArrayList<>();
    private Money total;
```

```
    public CostSummary(Currency userCurrency) {
        this.total = Money.of(0, userCurrency);
    }

    public void addLine(CurrencyConversion line) {
        lines.add(line);
        total = total.add(line.getToMoney());
    }

    public List<CurrencyConversion> getLines() {
        return List.copyOf(lines);
    }

    public Money getTotal() {
        return total;
    }
}
```

`CostSummary` также сообщает общую стоимость в предпочитаемой валюте. Он сохраняет общую стоимость в поле, а не вычисляет ее в `getTotal`, потому что приложение часто сортирует позиции по их `CostSummary.total` и пересчитывает каждый раз, когда мы проводим сравнение, а это как раз и есть его «узкое место», поскольку `CostSummary` необходимо обновлять `total` всякий раз при добавлении `CurrencyConversion`.

`CostSummary`, кроме того, фактически является общей мутирующей коллекцией. Поскольку это нарушает наше эмпирическое правило, приведенное во *врезке «Не изменяйте разделяемые коллекции»* (см. *разд. «Коллекции Java» главы 6*), функция выполняет копирование в `getLines`, чтобы ограничить ущерб.

Перейдем к `CostSummaryCalculator`. Он сохраняет текущую сумму для каждого курса `Currency` в поле `currencyTotals` при вызове `addCost`. Метод `summarise` создает `CostSummary`, используя источник обменных курсов для конвертации местных расходов в предпочитаемую путешественником валюту (пример 14.5).

Пример 14.5
[accumulator.0:src/main/java/travelator/itinerary/CostSummaryCalculator.java]

```
public class CostSummaryCalculator {
    private final Currency userCurrency;
    private final ExchangeRates exchangeRates;
    private final Map<Currency, Money> currencyTotals = new HashMap<>();

    public CostSummaryCalculator(
        Currency userCurrency,
        ExchangeRates exchangeRates
    ) {
        this.userCurrency = userCurrency;
        this.exchangeRates = exchangeRates;
    }
```

```
    public void addCost(Money cost) {
        currencyTotals.merge(cost.getCurrency(), cost, Money::add);
    }

    public CostSummary summarise() {
        var totals = new ArrayList<>(currencyTotals.values());
        totals.sort(comparing(m -> m.getCurrency().getCurrencyCode()));

        CostSummary summary = new CostSummary(userCurrency);
        for (var total : totals) {
            summary.addLine(exchangeRates.convert(total, userCurrency));
        }
        return summary;
    }

    public void reset() {
        currencyTotals.clear();
    }
}
```

Таким образом, вычисление `CostSummary` распределяется между двумя классами, в которых переплетаются следующие задачи:

- хранение информации из контекста вычисления, необходимой для вычисления сводки;
- расчет итоговых значений по каждой валюте, чтобы при вычислении не накапливались ошибки округления;
- конвертация расходов в предпочитаемую путешественником валюту;
- расчет общей суммы в предпочитаемой путешественником валюте;
- сортировка конвертации валют в алфавитном порядке исходного кода валюты;
- хранение конвертации валют и общей суммы, чтобы их можно было отобразить для путешественника.

Такое распределение задач между классами является обычным явлением, когда мы осуществляем вычисления путем мутации общего состояния. Но мы хотели бы разделить задачи и упростить их реализацию. К какой окончательной структуре мы должны стремиться?

Одна подсказка кроется в названии класса `CostCurrencyCalculator`. На лингвистическом жаргоне название `CostCurrencyCalculator` является *отглагольным существительным* — производным от глагола существительным, которое означает не более чем нечто, выполняющее действие, обозначенное глаголом, — например, *водитель*, *пекарь* или *калькулятор*. `CostCurrencyCalculator` — это так называемый класс исполнителей.

Еще один ключ к разгадке кроется в данных, хранящихся в классе. Предпочтительная валюта путешественника и источник обменных курсов являются контекстом

для расчета. Они управляются в другом месте приложения и хранятся в `CostCurrencyCalculator` — следовательно, они будут под рукой для его вычислений. Карта итогов по валютам (`currencyTotals`) содержит временные промежуточные результаты вычисления, которые не имеют значения после его завершения и, по сути, должны быть отброшены, чтобы избежать ошибок псевдонимов. Класс не *владеет* никакими данными, только временно хранит их по оперативным соображениям.

Таким образом, класс `CostCurrencyCalculator` представляет собой в нашей модели предметной области приложения не концепцию, а функцию, которую мы выполняем с элементами этой модели предметной области. А в Kotlin мы обычно реализуем функции не с помощью объектов, а с помощью — ну да — функций.

Давайте проведем рефракторинг от вычисления с мутирующими классами к функциям, которые работают с немутирующими данными.

Рефакторинг функций над немутирующими данными

Преобразование двух классов в Kotlin оставляет нас с Java в синтаксисе Kotlin. Перед вами `CostSummary` после небольшой очистки и перестановки (пример 14.6).

Пример 14.6 [accumulator.1:src/main/java/travelator/itinerary/CostSummary.kt]

```
class CostSummary(userCurrency: Currency) {

    private val _lines = mutableListOf<CurrencyConversion>()

    var total: Money = Money.of(0, userCurrency)
        private set

    val lines: List<CurrencyConversion>
        get() = _lines.toList()

    fun addLine(line: CurrencyConversion) {
        _lines.add(line)
        total += line.toMoney
    }
}
```

Автоматическое преобразование `CostSummaryCalculator` меньше нуждается в очистке (пример 14.7).

Пример 14.7 [accumulator.1:src/main/java/travelator/itinerary/CostSummaryCalculator.kt]

```
class CostSummaryCalculator(
    private val userCurrency: Currency,
    private val exchangeRates: ExchangeRates
```

```
) {
    private val currencyTotals = mutableMapOf<Currency, Money>()

    fun addCost(cost: Money) {
        currencyTotals.merge(cost.currency, cost, Money::add)
    }

    fun summarise(): CostSummary {
        val totals = ArrayList(currencyTotals.values)
        totals.sortWith(comparing { m: Money -> m.currency.currencyCode })

        val summary = CostSummary(userCurrency)
        for (total in totals) {
            summary.addLine(exchangeRates.convert(total, userCurrency))
        }
        return summary
    }

    fun reset() {
        currencyTotals.clear()
    }
}
```

Можно начать отсюда и осуществить рефракторинг мутации. Мы будем работать изнутри, делая `CostSummary` немутирующим типом значения и постепенно продвигая неизменность наружу через `CostSummaryCalculator`. Но прежде, чем реализовать это, вспомним, что нас в какой-то степени поглотила одержимость сортировкой коллекций Java на месте, поэтому сначала мы это исправим (пример 14.8).

Пример 14.8
[accumulator.2:src/main/java/travelator/itinerary/CostSummaryCalculator.kt]

```
fun summarise(): CostSummary {
    val totals = currencyTotals.values.sortedBy {
        it.currency.currencyCode
    }
    val summary = CostSummary(userCurrency)
    for (total in totals) {
        summary.addLine(exchangeRates.convert(total, userCurrency))
    }
    return summary
}
```

Теперь мы видим шаблон, который часто встречается в мутирующем коде: создайте объект (`CostSummary` в нашем случае) и вызовите несколько этапов инициализации, а затем верните его. Всякий раз, когда мы видим подобные этапы инициализации, то должны стремиться к `apply` (пример 14.9).

Пример 14.9
[accumulator.3:src/main/java/travelator/itinerary/CostSummaryCalculator.kt]

```
fun summarise(): CostSummary {
    val totals = currencyTotals.values.sortedBy {
        it.currency.currencyCode
    }
    val summary = CostSummary(userCurrency).apply {
        for (total in totals) {
            addLine(exchangeRates.convert(total, userCurrency))
        }
    }
    return summary
}
```

Применение `apply` позволяет нам сгруппировать шаги инициализации в блок, чтобы лучше выразить наше намерение. Это похоже на мини-конструктор: функция `summarise` никогда не видит ссылку на частично инициализированный `CostSummary`, а только завершенный объект.

Таково мелкомасштабное функциональное мышление — попытка ограничить область мутации даже внутри функции. Функциональное мышление также помогает нам увидеть, что перебор `totals` с созданием `CurrencyConversion` для каждого и вызовом `addLine` каждый раз — это то же самое, что и создание списка `conversions`, и запуск цикла для него (пример 14.10).

Пример 14.10
[accumulator.4:src/main/java/travelator/itinerary/CostSummaryCalculator.kt]

```
fun summarise(): CostSummary {
    val conversions = currencyTotals.values.sortedBy {
        it.currency.currencyCode
    }.map { exchangeRates.convert(it, userCurrency) }

    return CostSummary(userCurrency).apply {
        conversions.forEach(this::addLine)
    }
}
```

Зачем вносить это изменение? Затем, что мы хотим разобрать функцию `CostSummary` до ее немутирующей сути. Если бы `CostSummary` была неизменяемой, клиентский код должен был бы передавать список строк своему конструктору вместо вызова его метода `addLine`. `CostSummary` не должен отвечать за конвертацию валюты, поэтому мы заставляем блок `apply` выглядеть так, как мы хотим, чтобы выглядел его конструктор. Поэтому мы добавляем вторичный конструктор, который дублирует эту логику инициализации (пример 14.11).

Пример 14.11 [accumulator.5:src/main/java/travelator/itinerary/CostSummary.kt]

```
class CostSummary(userCurrency: Currency) {
    private val _lines = mutableListOf<CurrencyConversion>()

    var total: Money = Money.of(0, userCurrency)
        private set

    val lines: List<CurrencyConversion>
        get() = _lines.toList()

    constructor(
        userCurrency: Currency,
        lines: List<CurrencyConversion>
    ): this(userCurrency) {
        lines.forEach(::addLine)
    }

    fun addLine(line: CurrencyConversion) {
        _lines.add(line)
        total += line.toMoney
    }
}
```

Теперь мы можем изменить метод `CostSummaryCalculator.summarise` на вызов нового конструктора, обрабатывая класс `CostSummary` так, как если бы он был значением немутирующего типа (пример 14.12).

Пример 14.12 [accumulator.5:src/main/java/travelator/itinerary/CostSummaryCalculator.kt]

```
fun summarise(): CostSummary {
    val conversions = currencyTotals.values.sortedBy {
        it.currency.currencyCode
    }.map { exchangeRates.convert(it, userCurrency) }

    return CostSummary(userCurrency, conversions)
}
```

Это, в свою очередь, позволяет нам сделать класс `CostSummary` фактически немутирующим, по крайней мере извне (пример 14.13).

Пример 14.13 [accumulator.6:src/main/java/travelator/itinerary/CostSummary.kt]

```
class CostSummary(
    userCurrency: Currency,
    val lines: List<CurrencyConversion>
) {
```

```
    var total: Money = Money.of(0, userCurrency)
        private set

    init {
        lines.forEach {
            total += it.toMoney
        }
    }
}
```

Иногда трудно уйти от мутации после ее возникновения, особенно для таких накапливающих структур, как показанная здесь. Это заметно по неприятным `var` и `init`, поэтому имеет смысл воспользоваться дружественным нам оператором `fold`. У нас была серия действий (см. *разд. «Действия» главы 7*), влияющих на мутирующую переменную `total`. А `fold` преобразует действия в одно вычисление (см. *разд. «Вычисления» главы 7*), которое мы можем использовать для инициализации немутирующей переменной (пример 14.14).

Пример 14.14 [accumulator.7:src/main/java/travelator/itinerary/CostSummary.kt]

```
class CostSummary(
    userCurrency: Currency,
    val lines: List<CurrencyConversion>
) {
    val total = lines
        .map { it.toMoney }
        .fold(Money.of(0, userCurrency), Money::add)
}
```

Теперь наш класс `CostSummary` полностью немутирующий. Если мы сможем назначить `total` основным параметром конструктора, то можно будет сделать `CostSummary` классом данных. Мы могли бы организовать это, преобразовав текущий конструктор во вторичный, но вместо этого мы собираемся переместить все вычисления в `CostSummaryCalculator`, оставив `CostSummary` просто для хранения результатов этих вычислений.

Чтобы реализовать наше намерение, мы сначала выбираем выражение справа от знака равенства в определении свойства `total` и используем рефакторинг IDE **Ввести параметр** (Introduce Parameter), чтобы вывести выражение в качестве параметра конструктора (пример 14.15).

Пример 14.15 [accumulator.8:src/main/java/travelator/itinerary/CostSummary.kt]

```
class CostSummary(
    val lines: List<CurrencyConversion>,
    total: Money
```

```
) {
    val total = total
}
```

Свойство `total` теперь выделено как предупреждение о стиле — IDE обнаружила, что это свойство может быть объявлено в параметре конструктора. Быстрое применение комбинации клавиш <Alt>+<Enter> на предупреждении оставляет объявление класса в таком виде (пример 14.16).

Пример 14.16 [accumulator.9:src/main/java/travelator/itinerary/CostSummary.kt]

```
class CostSummary(
    val lines: List<CurrencyConversion>,
    val total: Money
)
```

А сейчас вернемся к `CostSummaryCalculator`. IntelliJ отправляет вычисления в `summarise` (пример 14.17).

Пример 14.17 [accumulator.9:src/main/java/travelator/itinerary/CostSummaryCalculator.kt]

```
fun summarise(): CostSummary {
    val lines = currencyTotals.values
        .sortedBy { it.currency.currencyCode }
        .map { exchangeRates.convert(it, userCurrency) }

    val total = lines
        .map { it.toMoney }
        .fold(Money.of(0, userCurrency), Money::add)

    return CostSummary(lines, total)
}
```

Теперь мы можем сделать `CostSummary` классом данных. Его единственной задачей будет хранение результатов вычислений для фильтрации, сортировки и отображения (пример 14.18).

Пример 14.18 [accumulator.10:src/main/java/travelator/itinerary/CostSummary.kt]

```
data class CostSummary(
    val lines: List<CurrencyConversion>,
    val total: Money
)
```

Ранее мы говорили, что мутирующее состояние может скрывать алгоритмы, разнося их по коду. Теперь можно оглянуться назад и увидеть, что так было и в случае

с CostSummary. Когда мы достигли нужной точки, вычисление общей суммы было разделено на инициализацию мутирующего свойства total и его обновление в методе addLine (пример 14.19).

Пример 14.19 [accumulator.1:src/main/java/travelator/itinerary/CostSummary.kt]

```
class CostSummary(userCurrency: Currency) {
    private val _lines = mutableListOf<CurrencyConversion>()

    var total: Money = Money.of(0, userCurrency)
        private set

    val lines: List<CurrencyConversion>
        get() = _lines.toList()

    fun addLine(line: CurrencyConversion) {
        _lines.add(line)
        total += line.toMoney
    }
}
```

Теперь вычисление представляет собой однострочное выражение в summarise (пример 14.20).

Пример 14.20 [accumulator.9:src/main/java/travelator/itinerary/CostSummaryCalculator.kt]

```
val total = lines
    .map { it.toMoney }
    .fold(Money.of(0, userCurrency), Money::add)
```

Давайте проделаем это снова

Точно так же все, что происходит с валютами, остается пока скрыто в оставшихся мутациях в CostSummaryCalculator (пример 14.21).

Пример 14.21 [accumulator.9:src/main/java/travelator/itinerary/CostSummaryCalculator.kt]

```
class CostSummaryCalculator(
    private val userCurrency: Currency,
    private val exchangeRates: ExchangeRates
) {
    private val currencyTotals = mutableMapOf<Currency, Money>()

    fun addCost(cost: Money) {
        currencyTotals.merge(cost.currency, cost, Money::add)
    }
```

```
    fun summarise(): CostSummary {
        val lines = currencyTotals.values
            .sortedBy { it.currency.currencyCode }
            .map { exchangeRates.convert(it, userCurrency) }

        val total = lines
            .map { it.toMoney }
            .fold(Money.of(0, userCurrency), Money::add)

        return CostSummary(lines, total)
    }

    fun reset() {
        currencyTotals.clear()
    }
}
```

Для их устранения мы можем прибегнуть к процессу, аналогичному показанному в предыдущем разделе, но на этот раз не станем добавлять вторичный конструктор. Вместо этого мы применим *«Рефракторинг с помощью Expand-and-Contract»* (см. соответствующую врезку в *разд. «Рефракторинг от необязательных типов к обнуляемым» главы 4*), добавив перегрузку метода `summarise`, который берет на себя затраты (пример 14.22).

Пример 14.22
[accumulator.11:src/main/java/travelator/itinerary/CostSummaryCalculator.kt]

```
fun summarise(costs: Iterable<Money>): CostSummary {
    val delegate = CostSummaryCalculator(userCurrency, exchangeRates)
    costs.forEach(delegate::addCost)
    return delegate.summarise()
}
```

Приемчик получился довольно-таки подлым... Старый метод `summarise` является действием: его результат зависит от предшествующей истории вызовов `addCost` и `reset`. А новый `summarise` — это расчет: его результат зависит только от значений входных данных (параметр `costs` плюс доступные свойства `userCurrency` и `exchangeRates`). И все же новый `summarise` использует старый. Он просто ограничивает область действия мутации локальной переменной, преобразуя ее в вычисление.

Когда мы используем эту версию `summarise`, то делаем различие между *контекстом* вычислений итоговых затрат, которые мы передаем в конструктор в качестве `userCurrency` и `exchangeRates`, и параметром *конкретного* расчета (`costs`, которые мы передаем в метод `summarise`). Важность этого момента мы почувствуем чуть позже — в *разд. «Обогащение обнаруженной нами абстракции» этой главы*).

Теперь, когда у нас есть два метода `summarise`, мы можем переместить наших абонентов в новый метод. Чтобы перейти к использованию нового `summarise`, нам при-

дется извлекать из объектов затраты (costs), которые мы хотим суммировать, вместо того, чтобы указывать им добавлять свои затраты в мутирующий калькулятор, который передаем. То есть вместо того, чтобы попросить добавление своих расходов в CostSummaryCalculator у дочерних методов, родительские методы будут запрашивать их расходы и объединять их.

В итоге мы используем калькулятор следующим образом (пример 14.23).

Пример 14.23
[accumulator.12:src/test/java/travelator/itinerary/Itinerary_CostTest.kt]

```
val fx: ExchangeRates = ...
val userCurrency = ...
val calculator = CostSummaryCalculator(userCurrency, fx)

fun costSummary(i: Itinerary) =
    calculator.summarise(i.costs())
```

А сообщим мы о затратах по нашим доменным моделям следующим образом (пример 14.24).

Пример 14.24 [accumulator.12:src/main/java/travelator/itinerary/Itinerary.kt]

```
data class Itinerary(
    val id: Id<Itinerary>,
    val route: Route,
    val accommodations: List<Accommodation> = emptyList()
) {
    ...

    fun costs(): List<Money> = route.costs() + accommodations.costs()
    ...
}

fun Iterable<Accommodation>.costs(): List<Money> = flatMap { it.costs() }
```

Когда все применения CostSummaryCalculator в приложении станут использовать новый метод summarise, мы сможем встроить вычисления currencyTotals и CostSummary в тот метод, который в настоящее время используется как локальный для выполнения этой задачи (пример 14.25).

Пример 14.25
[accumulator.11:src/main/java/travelator/itinerary/CostSummaryCalculator.kt]

```
fun summarise(costs: Iterable<Money>): CostSummary {
    val delegate = CostSummaryCalculator(userCurrency, exchangeRates)
    costs.forEach(delegate::addCost)
    return delegate.summarise()
}
```

Мы можем эффективно заменить встроенным выражением весь класс в этом методе, используя вместо этого локальные переменные (пример 14.26).

Пример 14.26
[accumulator.13:src/main/java/travelator/itinerary/CostSummaryCalculator.kt]

```
fun summarise(costs: Iterable<Money>): CostSummary {
    val currencyTotals = mutableMapOf<Currency, Money>()
    costs.forEach {
        currencyTotals.merge(it.currency, it, Money::plus)
    }
    val lines = currencyTotals.values
        .sortedBy { it.currency.currencyCode }
        .map { exchangeRates.convert(it, userCurrency) }
    val total = lines
        .map { it.toMoney }
        .fold(Money(0, userCurrency), Money::add)
    return CostSummary(lines, total)
}
```

Наши тесты все еще проходят, и IntelliJ сообщает, что все остальные методы `CostSummaryCalculator` в настоящее время не используются, как и поле `currencyTotals`. Следовательно, удалив все это, мы наконец добьемся успеха в удалении всех мутирующих состояний из класса.

Но не с помощью этого метода — у нас все еще есть мутирующая карта! Это последний остаток размытия алгоритма, о котором мы упоминали ранее. Мы наконец-то объединили всю логику в этот один метод, и поскольку вся наша логика находится в одном месте, мы знаем, что это происходит в одно время и безопасно для преобразования в любую эквивалентную форму.

Что это за форма? Подумав об этом, мы придем к выводу, что общую сумму по каждой валюте накапливает `MutableMap.merge`. Когда у нас есть все данные сразу, как сейчас, мы можем выполнить тот же расчет, сгруппировав их по валютам и суммировав списки (пример 14.27).

Пример 14.27
[accumulator.14:src/main/java/travelator/itinerary/CostSummaryCalculator.kt]

```
class CostSummaryCalculator(
    private val userCurrency: Currency,
    private val exchangeRates: ExchangeRates
) {
    fun summarise(costs: Iterable<Money>): CostSummary {
        val currencyTotals: List<Money> = costs
            .groupBy { it.currency }
            .values
            .map { moneys -> moneys.reduce(Money::add) }
```

```
        val lines: List<CurrencyConversion> = currencyTotals
            .sortedBy { it.currency.currencyCode }
            .map { exchangeRates.convert(it, userCurrency) }
        val total = lines
            .map { it.toMoney }
            .fold(Money(0, userCurrency), Money::add)
        return CostSummary(lines, total)
    }
}
```

Здесь немного раздражает то, что нам приходится использовать `reduce` для суммирования денежных средств вместо того, чтобы иметь приятную функцию-расширение `Iterable<Money>.sum()`. Наверное, нам следует это исправить. И теперь, когда все вычисления находятся в одном месте, можно было бы задуматься, есть ли смысл в том факте, что мы используем `reduce` в одном выражении и `fold` — в другом (подсказка: есть!). Но эти мысли могут появиться только потому, что код теперь изложен в одном месте.

Ключевым моментом является то, что теперь мы можем более четко видеть форму итогового вычисления `summarise`. Это чистая функция, которая применяется к набору затрат и оценивается в контексте ряда обменных курсов и предпочитаемой путешественником валюты. Функция преобразует вложенные объекты нашей предметной модели в простую коллекцию затрат, а затем преобразует затраты в карту итогов для каждой валюты, преобразует итоги для каждой валюты в список `CurrencyConversion` и, наконец, преобразует список валютных конверсий в `CostSummary`.

Функциональная программа преобразует свои входные данные в выходные данные

Если вы не можете так легко написать это за один этап, преобразуйте входные данные в промежуточное представление, которое легко преобразовать в выходные данные.

Вводите промежуточные формы и преобразования до тех пор, пока у вас не появится конвейер простых преобразований между промежуточными формами, которые создаются для преобразования имеющихся входных данных в нужные выходные данные.

Мы подробнее рассмотрим чистые функции, оцениваемые в контексте, в *главе 16*.

Обогащение обнаруженной нами абстракции

Программа `Travelator` делает больше с обменными курсами и предпочитаемой валютой путешественника, чем просто расчет суммы затрат. Например, когда пользователь просматривает гостиничные номера, она показывает ему стоимость каждого номера как в местной, так и в предпочитаемой валюте. То есть страница обзора

гостиничного номера выполняет конвертацию валюты по индивидуальной стоимости. Функции `CostSummaryCalculator` также необходимо выполнить конвертацию валют по отдельным затратам для расчета сводной информации. Если извлечь эту функциональность в качестве общедоступного метода, который мы можем назвать `toUserCurrency`, то можно инициализировать страницу обзора гостиничного номера с помощью `CostSummaryCalculator` вместо того, чтобы передавать ей как обменные курсы, так и предпочитаемую валюту. Мы также можем удалить из страницы обзора гостиничного номера расчет конвертации валюты, который, как мы теперь видим, представляет собой дублирующий код.

На этом этапе наш класс больше не является калькулятором итоговых затрат. В нем содержится контекст для любых цен, которые мы устанавливаем для отдельного путешественника. Так что давайте переименуем его, чтобы отразить новообретенную задачу. И сейчас мы не можем придумать для него лучшего названия, чем `Pricing Context` (Ценовой контекст), в результате чего наш класс приобретает следующий вид (пример 14.28).

Пример 14.28 [accumulator.16:src/main/java/travelator/itinerary/PricingContext.kt]

```
class PricingContext(
    private val userCurrency: Currency,
    private val exchangeRates: ExchangeRates
) {
    fun toUserCurrency(money: Money) =
        exchangeRates.convert(money, userCurrency)

    fun summarise(costs: Iterable<Money>): CostSummary {
        val currencyTotals: List<Money> = costs
            .groupBy { it.currency }
            .values
            .map {
                it.sumOrNull() ?: error("Unexpected empty list")
            }
        val lines: List<CurrencyConversion> = currencyTotals
            .sortedBy { it.currency.currencyCode }
            .map(::toUserCurrency)
        val total = lines
            .map { it.toMoney }
            .sum(userCurrency)
        return CostSummary(lines, total)
    }
}
```

По поводу имен...

Название, на наш взгляд, отчасти излишне обобщенное, но сойдет, пока мы не придумаем что-нибудь получше. По крайней мере, оно не вводит в заблуждение. Переименовать будет несложно даже в смешанной кодовой базе Java/Kotlin, так что небольшое постепенное улучшение лучше, чем отсутствие улучшений.

При этом остается код, который использовался для применения `CostSummaryCalculator` (пример 14.29).

Пример 14.29
[accumulator.16:src/test/java/travelator/itinerary/Itinerary_CostTest.kt]

```
val fx: ExchangeRates = ...
val userCurrency = ...
val pricing = PricingContext(userCurrency, fx)

fun costSummary(i: Itinerary) = pricing.summarise(i.costs())
```

Теперь, когда в нашей кодовой базе реализована эта концепция, мы можем определить другие части приложения, которые могут ее использовать. Мы можем перенести логику из этих частей в `Pricing Context`, сделав его «универсальным магазином» для операций, которым необходимо конвертировать денежные суммы в предпочитаемую путешественником валюту. И если в конечном итоге он будет заполнен разрозненными методами для разных вариантов использования, мы сможем перенести операции из методов в функции-расширения, чтобы они были ближе к тому месту, где они необходимы (см. *главу 10*).

Двигаемся дальше

Мы начали эту главу с вычисления, которое основывалось на общем мутирующем состоянии. Это дублировало логику из стандартной библиотеки и создавало риск ошибок псевдонимов. К концу главы мы преобразовали то же самое вычисление в преобразование немутирующих данных.

Чтобы сделать это, мы переместили мутацию из нашего кода в двух направлениях: наружу и внутрь. Перемещение наружу было очевидно: мы заставили `CostSummaryCalculator` видеть класс `CostSummary` в качестве неизменяющегося типа значения, а затем сделали `CostSummary` немутирующим. После этого мы заставили пользователей `CostSummaryCalculator` рассматривать его как немутирующий контекст для вычисления, а затем сделали `CostSummaryCalculator` немутирующим. А перемещение внутрь? Мы заменили императивный код, который изменял коллекции и поля, вызовами стандартных функций более высокого порядка — таких как `groupingBy`, `fold` и `reduce`. «Под капотом» эти функции могут изменять состояние, но они скрывают эту мутацию от своих абонентов. Извне функции представляют собой вычисления.

Мы можем использовать тот же подход в нашем собственном коде, когда это потребуется. Иногда изменить коллекцию — это самое простое, что можно сделать. Стандартная библиотека не всегда имеет функцию более высокого порядка, которая преобразует данные так, как мы хотим. Если нам действительно нужна мутирующая коллекция, мы можем скрыть эту мутацию внутри вычисления, чтобы

ограничить зону поражения любых потенциальных ошибок псевдонимов. Однако каждый релиз добавляет в стандартную библиотеку все больше функций, поэтому со временем потребность в них уменьшается.

Функциональное программирование не устраняет мутирующее состояние, а вместо этого *возлагает ответственность за него на среду выполнения*. Функциональная программа объявляет, что должна вычислять среда выполнения, и позволяет среде выполнения отвечать за расчет этого вычисления. Kotlin не является чисто функциональным языком, но мы извлекаем выгоду, следуя этому принципу там, где можем.

От инкапсулированных коллекций к псевдонимам типов

В Java мы инкапсулируем коллекции объектов в классы, чтобы контролировать мутацию и добавлять операции. Управление мутациями в Kotlin менее важно, и для добавления операций мы можем использовать функции-расширения. Насколько наши проекты были бы лучше без инкапсуляции и как мы этого добьемся?

В *главе 6* мы рассмотрели различия Java и Kotlin, когда речь идет о коллекциях. Интерфейсы коллекций Java в соответствии с ее объектно-ориентированными корнями, фундаментально мутирующие, в то время как Kotlin рассматривает коллекции как типы значений. Там было показано, что если изменить общие (разделяемые) коллекции, то можно столкнуться со всевозможными проблемами. Мы могли бы избежать этих проблем, не изменяя общие коллекции (см. *врезку «Не изменяйте разделяемые коллекции»* в *разд. «Коллекции Java» главы 6*), но в Java это трудно реализовать, когда методы `add` и `set` находятся всего в нескольких шагах от автозаполнения. Вместо дисциплинированного соблюдения соглашений большая часть Java-кода разумно выбирает более безопасный подход, заключающийся в том, чтобы просто не делиться необработанными коллекциями. Вместо этого коллекции скрываются внутри другого объекта.

Вот, например, перед вами класс `Route` из `Travelator` (пример 15.1).

Пример 15.1
[encapsulated-collections.0:src/main/java/travelator/itinerary/Route.java]

```
public class Route {
    private final List<Journey> journeys; ❶

    public Route(List<Journey> journeys) {
        this.journeys = journeys; ❷
    }

    public int size() { ❸
        return journeys.size();
    }

    public Journey get(int index) { ❸
        return journeys.get(index);
    }
```

```
    public Location getDepartsFrom() { ❹
        return get(0).getDepartsFrom();
    }

    public Location getArrivesAt() { ❹
        return get(size() - 1).getArrivesAt();
    }

    public Duration getDuration() { ❹
        return Duration.between(
            get(0).getDepartureTime(),
            get(size() - 1).getArrivalTime());
    }

    ...
}
```

❶ `Route` инкапсулирует список `List` из `Journey`.

❷ Необработанные данные передаются в конструктор.

❸ Доступ к данным — например, для отображения в пользовательском интерфейсе обеспечивается методами `size` и `get`.

❹ Класс Route реализует логику приложения, которая использует содержимое инкапсулированного списка.

Защитные копии

Чтобы полностью инкапсулировать список, конструктор `Route` может использовать защитные копии параметра `journeys`. Однако мы «знаем», что наша система создает объекты `Route` только в десериализаторе JSON или в тестах, ни один из которых не сохраняется в списке поездок после создания маршрута `Route`, который его использует. Следовательно, нет никакого риска *ошибок наложения* (**https://oreil.ly/PeqKs**), и мы можем сэкономить на затратности создания копии всякий раз, когда создаем `Route`.

Но если кто-то придет и создаст `Route` с коллекцией, которую он позже изменит, мы можем пожалеть о такой оптимизации. В этом и заключается проблема с использованием соглашений, которые не прописаны в системе типов.

Как только у нас будет класс `Route`, он станет удобным пространством имен для размещения операций с маршрутами — таких как `getDepartsFrom` и `getDuration`. В этом случае все показанные методы используют только другие общедоступные методы, и полиморфного поведения нет, поэтому эти операции *могут* быть определены в качестве статических методов, принимающих параметр `Route`. Мы можем рассматривать `Route` скорее, как пространство имен, чем как класс, — операции не обязательно должны быть методами. Просто удобнее, чтобы они ими были, — по крайней мере, в Java, где статические функции гораздо менее доступны для обнаружения, чем методы. В Kotlin, как показано в *главе 10*, преобразование операций в функции-расширения позволило бы нам находить и вызывать их так, как если бы они были методами. `Route` как класс тогда не добавлял бы никакого значения к спи-

ску `List` из `Journey`, просто препятствуя людям его изменить. И в кодовой базе, созданной полностью на Kotlin, этот `List` в любом случае был бы фактически немутирующим.

На самом деле `Route` осуществляет гораздо более неприятную операцию, чем добавление отсутствующего значения в `List<Journey>`, — он удаляет значение. Если бы у нас была последовательность `List<Journey>`, наш фронтенд-код мог бы задействовать ее `Iterator` при рендеринге (пример 15.2).

Пример 15.2 [encapsulated-collections.0:src/main/java/travelator/UI.java]

```
public void render(Iterable<Journey> route) {
    for (var journey : route) {
        render(journey);
    }
}
```

Используя `Route`, мы возвращаемся к программированию 1980-х годов (пример 15.3).

Пример 15.3 [encapsulated-collections.0:src/main/java/travelator/UI.java]

```
public void render(Route route) {
    for (int i = 0; i < route.size(); i++) {
        var journey = route.get(i);
        render(journey);
    }
}
```

Если мы инкапсулируем коллекцию, то *сокращаем* перечень операций, доступных нам для работы с ее содержимым, до операций, определенных инкапсулирующим классом. Когда мы хотим обработать эти данные по-новому, путь наименьшего сопротивления заключается в добавлении новых методов в класс. Чем больше методов мы добавляем в класс, тем больше класс *усиливает* соединение между различными частями нашего приложения. Прежде чем мы это узнаем, добавление операции для поддержки новой функции пользовательского интерфейса приведет к перекомпиляции нашего уровня доступа к данным.

Создание коллекций доменов

Если мы не инкапсулируем коллекцию — когда заставляем нашу модель предметной области *быть* соответствующей структурой данных, а не скрываем ее внутри границ другого класса, — то *расширяем* доступные нам операции для работы с данными. В таком случае у нас есть операции, специфичные для приложения, *и* все операции, определенные для коллекции. Клиентский код может определять необходимые ему операции в терминах обширных API Collections без необходимости добавлять их в класс.

Вместо класса `Route`, объединяющего всю функциональность маршрута и, в свою очередь, связывающего все части нашего приложения вместе, мы можем рассматривать функциональность как операции, которые должны быть составлены путем импорта функций-расширений. Пользовательский интерфейс может определять функции, отображающие коллекцию `List<Journey>`, которая, в свою очередь, импортирует функции, преобразующие `Iterable<Journey>`. Уровень сохраняемости может преобразовывать ответы базы данных в `List<Journey>` и вообще не иметь никакого особого понятия о «рутинности».

Мы можем программировать подобным образом на Java, но плохая обнаруживаемость статических функций в сочетании с мутирующими коллекциями противоречит сути этого языка. В Kotlin же есть функции-расширения, позволяющие сделать статические функции более доступными для обнаружения, и немутирующие коллекции, так что разделение нашей модели предметной области на типы коллекций и отдельные операции становится «путем к счастью».

Если нам не нужно контролировать доступ к коллекции, чтобы предотвратить неудобную мутацию, и не нужно писать класс для размещения операций над коллекциями определенного типа, то делает ли класс `Route` что-нибудь для нас? Ну, он дает название `List<Journey>`, а также дает тип этой `List<Journey>`, что поможет отличить ее от другой `List<Journey>` — в отчетах о путешествиях наших туристов, которые забронировали их за неделю, например. Кроме того, в некотором смысле это действительно мешает нам. Мы увидим это далее — в *разд. «Замена псевдонима типа»*.

Там, где различия между разными типами списков поездок *не* являются критичными, Kotlin позволяет нам задействовать *псевдонимы типов* для сопоставления имени `Route` с `List<Journey>` вместо того, чтобы использовать для этого класс:

```
typealias Route = List<Journey>
```

Таким образом, в Kotlin были устранены препятствия для использования коллекций в качестве типов доменов. Инкапсуляция немутирующих коллекций должна быть скорее исключением, чем правилом.

Коллекции с другими свойствами

Конечно, мы не всегда можем просто заменять классы псевдонимами типов. Взглянем, к примеру, на класс `Itinerary` (пример 15.4).

Пример 15.4
[encapsulated-collections.0:src/main/java/travelator/itinerary/Itinerary.kt]

```
class Itinerary(
    val id: Id<Itinerary>,
    val route: Route
) {
    ...
}
```

В дополнение к `Journey`, в настоящее время скрывающемуся в `route`, `Itinerary` содержит идентификатор `Id`, что позволяет нам рассматривать его как объект. В этих случаях мы не можем просто заменить класс его коллекцией.

Но здесь можно получить множество преимуществ неинкапсулированных коллекций, осуществив для `List<Journey>` имплементацию `Itinerary`. Такую процедуру трудно осуществить прямо сейчас, потому что `Route` не реализует сам этот интерфейс, но это хорошая стратегия, поскольку большая часть нашего домена выражена в виде полных коллекций. Мы доберемся до нее далее — в *разд. «Рефракторинг коллекций с другими свойствами»*.

Рефракторинг инкапсулированных коллекций

Одной из основных услуг нашего приложения `Travelator` является планирование маршрута.

`Route`, показанный ранее, представляет собой последовательность маршрутов, которые могут привести путешественника из одного места в другое. Мы хотели бы добавить некоторую функциональность, которая позволит нам продавать проживание, где `Route` будет разделен на несколько дней, но в качестве абстракции ключевого домена. `Route` разрушается под тяжестью всех операций, которые мы уже добавили к нему, а также соединяющих разрозненные части кодовой базы вместе. Давайте посмотрим, сможем ли мы реорганизовать `Route`, чтобы освободить место, прежде чем начнем работу над новой функцией. Перед вами снова класс `Route` из Java (пример 15.5).

Пример 15.5
[encapsulated-collections.1:src/main/java/travelator/itinerary/Route.java]

```
public class Route {
    private final List<Journey> journeys;

    public Route(List<Journey> journeys) {
        this.journeys = journeys;
    }

    public int size() {
        return journeys.size();
    }

    public Journey get(int index) {
        return journeys.get(index);
    }

    public Location getDepartsFrom() {
        return get(0).getDepartsFrom();
    }
```

```
    ... другие методы
}
```

Преобразование операций в расширения

Мы собираемся сделать `Route` менее крупногабаритным (возможно, скорее даже малогабаритным), переместив его операции из методов в функции. Функции-расширения делают эту стратегию разумной, но только в Kotlin, где их гораздо легче обнаружить. Следовательно, мы попробуем этот трюк только после того, как большинство наших применений `Route` будут перенесены в Kotlin. К счастью, нашей команде действительно нравится конвертировать Java в Kotlin, и она усердно работает над главами этой книги, поэтому мы готовы попробовать такой рефакторинг прямо сейчас.

В конечном счете мы хотим деинкапсулировать коллекцию, чтобы наши клиенты работали в терминах `List<Journey>`, а не использовали `Route`. И чтобы операции предоставлялись функциями-расширениями над этим `List<Journey>`.

Начнем с преобразования `Route` в Kotlin, что после небольшого наведения порядка дает следующий код (пример 15.6).

Пример 15.6
[encapsulated-collections.2:src/main/java/travelator/itinerary/Route.kt]

```
class Route(
    private val journeys: List<Journey>
) {
    fun size(): Int = journeys.size

    operator fun get(index: Int) = journeys[index]

    val departsFrom: Location
        get() = get(0).departsFrom

    ... другие методы
}
```

Как и в предыдущих случаях, вы должны иметь в виду, что между рефакторингами мы проводим тесты, помогающие убедиться, что ничего не сломалось. На текущий момент все пока в порядке.

Как только класс окажется в Kotlin, IntelliJ может преобразовывать методы в методы-расширения. Давайте попробуем этот рефакторинг для свойства `departsFrom`: выделите его, нажмите комбинацию клавиш <Alt>+<Enter> и выберите опцию **Преобразовать элемент в расширение** (Convert member to extension) — метод исчезнет и снова появится на верхнем уровне файла (пример 15.7).

Пример 15.7
[encapsulated-collections.3:src/main/java/travelator/itinerary/Route.kt]

```
val Route.departsFrom: Location
    get() = get(0).departsFrom
```

Код Kotlin по-прежнему будет иметь доступ к `route.departsFrom` в качестве свойства, а Java-код — нет. IntelliJ услужливо исправил одно использование Java, чтобы видеть свойство как статический метод (пример 15.8).

Пример 15.8 [encapsulated-collections.3:src/main/java/travelator/UI.java]

```
public void renderWithHeader(Route route) {
    renderHeader(
        RouteKt.getDepartsFrom(route), ❶
        route.getArrivesAt(),
        route.getDuration()
    );
    for (int i = 0; i < route.size(); i++) {
        var journey = route.get(i);
        render(journey);
    }
}
```

❶ Вызов статического метода в `Route.kt`.

Опция **Преобразовать элемент в расширение** хорошо работает для методов, которые вызывают только `Route` из публичных API. Программа потерпит неудачу, если мы попробуем то же самое, например, с `withJourneyAt` (пример 15.9).

Пример 15.9
[encapsulated-collections.3:src/main/java/travelator/itinerary/Route.kt]

```
fun withJourneyAt(index: Int, replacedBy: Journey): Route {
    val newJourneys = ArrayList(journeys)
    newJourneys[index] = replacedBy
    return Route(newJourneys)
}
```

Здесь есть ссылка на свойство `journeys`, которое в настоящее время является закрытым и поэтому невидимо для функции-расширения. Но мы можем сделать свойство общедоступным (при условии, что не злоупотребляем этим, изменяя `List` из Java-кода). Такой подход исправит функцию-расширение (пример 15.10).

Пример 15.10
[encapsulated-collections.4:src/main/java/travelator/itinerary/Route.kt]

```
fun Route.withJourneyAt(index: Int, replacedBy: Journey): Route {
    val newJourneys = ArrayList(journeys)
```

```
    newJourneys[index] = replacedBy
    return Route(newJourneys)
}
```

Мы можем продолжать процесс преобразования элементов в расширения до тех пор, пока не останется ни одного элемента. Даже `size` и `get` могут быть удалены, при условии, что мы будем рады использовать их статически в любых оставшихся клиентах Java (пример 15.11).

Пример 15.11 [encapsulated-collections.5:src/main/java/travelator/UI.java]

```
public void render(Route route) {
    for (int i = 0; i < RouteKt.getSize(route); i++) {
        var journey = RouteKt.get(route, i);
        render(journey);
    }
}
```

Обратите внимание, что, поскольку мы преобразовали метод `size` в свойство-расширение `size`, Java видит функцию `getSize`.

Вот и все, что осталось от некогда раздутого класса `Route` (пример 15.12).

Пример 15.12 [encapsulated-collections.5:src/main/java/travelator/itinerary/Route.kt]

```
class Route(
    val journeys: List<Journey>
)

val Route.size: Int
    get() = journeys.size

operator fun Route.get(index: Int) = journeys[index]

...
```

Все его операции (запрещающие доступ к `journeys`) теперь стали расширениями, хотя и в том же файле. Но теперь, когда они *являются* расширениями, мы можем переместить их из этого файла в другие, даже в разных модулях, чтобы лучше отделить наши зависимости.

Замена псевдонима типа

Теперь, когда мы достигли нашей цели по отделению функциональности `Route` от класса, не стал ли этот класс лишним? На самом деле обертывание `List` хуже, чем лишнее: оно мешает нам легко использовать все полезные функции-расширения

в стандартной библиотеке Kotlin для построения, преобразования и обработки маршрутов. Процитирую одно из высказываний Алана Перлиса из «Epigrams of Programming» (**https://oreil.ly/QDOJz**): «Лучше, чтобы 100 функций работали с одной структурой данных, чем 10 функций с 10 структурами данных». Мы не хотим, чтобы у маршрута (`Route`) *имелся* список (`List`) путешествий (`Journey`) — мы хотим, чтобы он *был* списком путешествий (`List<Journey>`). Этого очень легко достичь в Kotlin с помощью делегирования (пример 15.13).

Пример 15.13
[encapsulated-collections.6:src/main/java/travelator/itinerary/Route.kt]

```
class Route(
    val journeys: List<Journey>
) : List<Journey> by journeys
```

На самом деле, можно хотеть большего, чем чтобы маршрут (`Route`) был списком путешествий (`List<Journey>`), — у нас может появиться желание, чтобы список путешествий (`List<Journey>`) был маршрутом (`Route`). Чтобы понять зачем, давайте посмотрим на функцию `withJourneyAt`, о которой мы упоминали ранее.

Когда путешественник решает, что он предпочел бы не путешествовать на верблюде, мы не можем просто заменить `Journey`, потому что функция `Route` является немутирующей. Вместо этого мы возвращаем новую функцию `Route`, в которой `journeys` — это копия, заменяющая соответствующее путешествие `Journey` (пример 15.14).

Пример 15.14
[encapsulated-collections.5:src/test/java/travelator/itinerary/RouteTests.kt]

```
@Test
fun replaceJourney() {
    val journey1 = Journey(waterloo, alton, someTime(), someTime(), RAIL)
    val journey2 = Journey(alton, alresford, someTime(), someTime(), CAMEL)
    val journey3 = Journey(alresford, winchester, someTime(), someTime(), BUS)
    val route = Route(listOf(journey1, journey2, journey3))

    val replacement = Journey(alton, alresford, someTime(), someTime(), RAIL)
    val replaced = route.withJourneyAt(1, replacement)

    assertEquals(journey1, replaced.get(0))
    assertEquals(replacement, replaced.get(1))
    assertEquals(journey3, replaced.get(2))
}
```

Попутно обратите внимание, что этот тест был усложнен тем, что для доступа к компонентам `route` использовался только `get`. Мы можем исправить это теперь, когда у нас есть прямой доступ к свойству `journeys` (пример 15.15).

Пример 15.15
[encapsulated-collections.4:src/main/java/travelator/itinerary/Route.kt]

```
fun Route.withJourneyAt(index: Int, replacedBy: Journey): Route {
    val newJourneys = ArrayList(journeys)
    newJourneys[index] = replacedBy
    return Route(newJourneys)
}
```

Мы не можем просто управлять `journeys`, потому что `Route` обертывает `journeys`. Нам необходимо ее развернуть, осуществить взаимодействие и снова завернуть обратно. Если бы `List<Journey>` была бы `Route`, тогда мы могли бы применить отличную обобщенную функцию (пример 15.16).

Пример 15.16
[encapsulated-collections.7:src/main/java/travelator/itinerary/Route.kt].

```
fun <T> Iterable<T>.withItemAt(index: Int, replacedBy: T): List<T> =
    this.toMutableList().apply {
    this[index] = replacedBy
}
```

Как бы то ни было, даже используя `withItemAt`, нам все равно приходится работать с обертыванием (пример 15.17).

Пример 15.17
[encapsulated-collections.7:src/main/java/travelator/itinerary/Route.kt]

```
fun Route.withJourneyAt(index: Int, replacedBy: Journey): Route =
    Route(journeys.withItemAt(index, replacedBy))
```

Любая операция, которая преобразует `Route`, столкнется с проблемой — проблемой, которой не существовало бы, если бы мы просто использовали псевдоним типа, чтобы сказать, что `Route` и `List<Journey>` относятся к одному типу.

Чтобы добраться до этого, нам придется удалить все вызовы конструктора `Route` и запросы свойства `journeys`, эффективно развернув нашу тщательно разработанную инкапсуляцию. Существует хитрость, позволяющая сделать это автоматически, но она основана на преобразовании всех клиентов `Route` в Kotlin. То же самое относится и к использованию псевдонима типа — поэтому, если у нас есть какие-либо оставшиеся Java-клиенты, нам придется смириться с некоторым редактированием вручную.

Что мы собираемся сейчас сделать? — заменить класс псевдонимом типа и в то же время добавить временные определения, которые эмулируют API класса. Этот API в настоящее время выглядит так (пример 15.18).

Пример 15.18
[encapsulated-collections.6:src/main/java/travelator/itinerary/Route.kt]

```
class Route(
    val journeys: List<Journey>
) : List<Journey> by journeys
```

Мы имитируем его с помощью следующего кода (пример 15.19).

Пример 15.19
[encapsulated-collections.8:src/main/java/travelator/itinerary/Route.kt]

```
typealias Route = List<Journey>

fun Route(journeys: List<Journey>) = journeys

val Route.journeys get() = this
```

Мы можем эмулировать вызов конструктора `Route(...)` с помощью функции с тем же именем, потому что в Kotlin нет ключевого слова `new`. Аналогично, мы заменяем свойство `journeys` на свойство-расширение, которое возвращает получатель. Конечным результатом является то, что наши клиенты Kotlin продолжают компилироваться с использованием этого нового API (пример 15.20).

Пример 15.20
[encapsulated-collections.8:src/test/java/travelator/itinerary/RouteTests.kt]

```
val route = Route(listOf(journey1, journey2, journey3)) ❶

val replacement = Journey(alton, alresford, someTime(), someTime(), RAIL)

assertEquals(
    listOf(journey1, replacement, journey3),
    route.withJourneyAt(1, replacement).journeys ❷
)
```

❶ Наша новая функция — не конструктор.

❷ Свойство-расширение — не свойство класса.

Замена встроенным выражением обеих функций и свойств завершает рефакторинг. Инкапсулированная коллекция теперь просто коллекция (пример 15.21).

Пример 15.21
[encapsulated-collections.9:src/test/java/travelator/itinerary/RouteTests.kt]

```
val route = listOf(journey1, journey2, journey3) ❶

val replacement = Journey(alton, alresford, someTime(), someTime(), RAIL)
```

```
assertEquals(
    listOf(journey1, replacement, journey3),
    route.withJourneyAt(1, replacement) ❷
)
```

❶ `Route` не работал.

❷ Как и `journeys`.

Все связи оставшихся клиентов Java будут сломаны, когда мы заменим класс `Route` псевдонимом типа, потому что Java не понимает псевдонимы типов. Мы исправили это вручную, заменив `Route` на `List<Journey>` (пример 15.22).

Пример 15.22 [encapsulated-collections.8:src/main/java/travelator/UI.java]

```
public void render(List<Journey> route) {
    for (int i = 0; i < RouteKt.getSize(route); i++) {
        var journey = RouteKt.get(route, i);
        render(journey);
    }
}
```

Наша трансформация почти завершена, но у нас все еще остались функции `size` и `get` (пример 15.23).

Пример 15.23 [encapsulated-collections.9:src/main/java/travelator/itinerary/Route.kt]

```
val Route.size: Int
    get() = this.size

operator fun Route.get(index: Int) = this[index]
```

Поскольку эти функции имеют ту же сигнатуру, что и их двойники методов в `List`, компилятор предупреждает нас об этом, затеняя их, — наш код Kotlin будет вызывать методы, а не расширения. Это означает, что если бы у нас не было никакого клиентского кода Java, вызывающего расширения как статические, мы могли бы удалить их.

Однако у нас есть Java-клиент — надоедливый код рендеринга, который по-прежнему вызывает в `RouteKt` расширения `getSize` и `get`. Эти расширения вызывают методы, которые мы хотим использовать, но мы не можем заменить встроенным выражением код из Kotlin в Java, поэтому просто удалим их. Теперь компилятор подскажет нам, где нам нужно исправить Java, и мы можем сделать это вручную (пример 15.24).

Пример 15.24 [encapsulated-collections.10:src/main/java/travelator/UI.java]

```
public void render(List<Journey> route) {
    for (int i = 0; i < route.size(); i++) {
        var journey = route.get(i);
        render(journey);
    }
}
```

На самом деле, конечно, мы бы заменили это на следующий код (пример 15.25).

Пример 15.25 [encapsulated-collections.10:src/main/java/travelator/UI.java]

```
public void render(Iterable<Journey> route) {
    for (var journey : route) {
        render(journey);
    }
}
```

Клиенты Kotlin не зависят от удаления расширений, потому что они всегда вызывали методы в `List`, так что преобразование почти завершено. Теперь мы также можем заменить элемент `withJourneyAt` встроенным выражением, поскольку он тоже не работает. Это оставляет нас с `Route` в таком виде (пример 15.26).

Пример 15.26 [encapsulated-collections.10:src/main/java/travelator/itinerary/Route.kt]

```
typealias Route = List<Journey>

val Route.departsFrom: Location
    get() = first().departsFrom

val Route.arrivesAt: Location
    get() = last().arrivesAt

val Route.duration: Duration
    get() = Duration.between(
        first().departureTime,
        last().arrivalTime
    )
... прочие операции
```

Наше применение Kotlin — это просто операции `List` (пример 15.27).

Пример 15.27
[encapsulated-collections.10:src/test/java/travelator/itinerary/RouteTests.kt]

```
val route = listOf(journey1, journey2, journey3)
assertEquals(
    listOf(journey1, replacement, journey3),
    route.withItemAt(1, replacement)
)
```

Любая остаточная Java читаема, хотя и немного уродлива (пример 15.28).

Пример 15.28 [encapsulated-collections.10:src/main/java/travelator/UI.java]

```
public void renderWithHeader(List<Journey> route) {
    renderHeader(
        RouteKt.getDepartsFrom(route),
        RouteKt.getArrivesAt(route),
        RouteKt.getDuration(route)
    );
    for (var journey : route) {
        render(journey);
    }
}
```

Рефракторинг коллекций с другими свойствами

Как было показано ранее, нельзя использовать псевдонимы типов, когда типы имеют коллекции с другими атрибутами. Мы также рассмотрели класс `Itinerary`, который сочетает `id` и `Route` (пример 15.29).

Пример 15.29
[encapsulated-collections.11:src/main/java/travelator/itinerary/Itinerary.kt]

```
class Itinerary(
    val id: Id<Itinerary>,
    val route: Route
) {
    fun hasJourneyLongerThan(duration: Duration) =
        route.any { it.duration > duration }

    ...
}
```

Получить преимущества от возможности запрашивать различные `Journey` можно напрямую, реализуя `Route` с делегированием (пример 15.30).

Пример 15.30
[encapsulated-collections.12:src/main/java/travelator/itinerary/Itinerary.kt]

```
class Itinerary(
    val id: Id<Itinerary>,
    val route: Route
) : Route by route { ❶

    fun hasJourneyLongerThan(duration: Duration) =
        any { it.duration > duration }

    ...
}
```

❶ Функция `by route` заявляет, что объект `Itinerary` делегирует все методы интерфейса `Route` параметру `route`, переданному в конструктор. Класс может переопределить это поведение, предоставив свою собственную реализацию методов делегированного интерфейса, но мы не хотим делать это для `Itinerary`.

Теперь, можно рассматривать `Itinerary` как `Route` — мы можем вынести `hasJourneyLongerThan` в качестве расширения и оставить его доступным для любого `Route`, а не только для `Itinerary` (пример 15.31).

Пример 15.31
[encapsulated-collections.13:src/main/java/travelator/itinerary/Itinerary.kt]

```
fun Route.hasJourneyLongerThan(duration: Duration) =
    any { it.duration > duration }
```

Все эти расширения `Route` (известного как `List<Journey>`), которые мы переместили из методов расширения, также применимы и к `Itinerary` (пример 15.32).

Пример 15.32
[encapsulated-collections.13:src/main/java/travelator/itinerary/Itineraries.kt]

```
fun Iterable<Itinerary>.shortest() =
    minByOrNull {
        it.duration ❶
    }
```

❶ Это `Route.duration`, она же `List<Journey>.duration`.

Что мы не можем сделать так же легко, так это создать новый `Itinerary` из существующего. Такая процедура сейчас легко выполняется для `Route`, потому что стандартный API для операций на `List<Journey>` (на самом деле, обычно `Iterable<Journey>`, как показано в *главе 6*) возвращает `List<Journey>`, которая по-другому называется `Route` (пример 15.33).

Пример 15.33
[encapsulated-collections.13:src/main/java/travelator/itinerary/itineraries.kt]

```
fun Route.withoutJourneysBy(travelMethod: TravelMethod) =
    this.filterNot { it.method == travelMethod }
```

Мы должны создать новый `Itinerary` для `Itinerary`, чтобы заново запаковать результат (пример 15.34).

Пример 15.34
[encapsulated-collections.13:src/main/java/travelator/itinerary/itineraries.kt]

```
fun Itinerary.withoutJourneysBy(travelMethod: TravelMethod) =
    Itinerary(
        id,
        this.filterNot { it.method == travelMethod }
    )
```

Это еще один пример того, как классы данных приходят на помощь (пример 15.35).

Пример 15.35
[encapsulated-collections.14:src/main/java/travelator/itinerary/Itinerary.kt]

```
data class Itinerary(
    val id: Id<Itinerary>,
    val route: Route
) : Route by route {

    ...
}
```

Превращение `Itinerary` в класс данных означает, что мы можем создать копию только с измененным маршрутом, — независимо от того, сколько у него других свойств (пример 15.36).

Пример 15.36
[encapsulated-collections.14:src/main/java/travelator/itinerary/itineraries.kt]

```
fun Itinerary.withoutJourneysBy(travelMethod: TravelMethod) =
    copy(route = filterNot { it.method == travelMethod } )
```

А будет лучше, если еще добавить метод `withTransformedRoute` (пример 15.37).

Пример 15.37
[encapsulated-collections.15:src/main/java/travelator/itinerary/Itinerary.kt]

```
data class Itinerary(
    val id: Id<Itinerary>,
```

```
    val route: Route
) : Route by route {

    fun withTransformedRoute(transform: (Route).() -> Route) =
        copy(route = transform(route))

    ...
}
```

Это позволяет нам создавать преобразованный `Itinerary` почти так же легко, как мы могли создать преобразованный `Route` (пример 15.38).

Пример 15.38
[encapsulated-collections.15:src/main/java/travelator/itinerary/Itineraries.kt]

```
fun Itinerary.withoutJourneysBy(travelMethod: TravelMethod) =
    withTransformedRoute {
        filterNot { it.method == travelMethod }
    }

fun Itinerary.withoutLastJourney() =
    withTransformedRoute { dropLast(1) }
```

Двигаемся дальше

Мы начали эту главу с класса Java, который инкапсулировал мутирующую коллекцию, чтобы гарантировать семантику значений. Поскольку мы перевели большую часть нашего кода на Kotlin, то смогли положиться на систему типов Kotlin, чтобы предотвратить изменение коллекции и не инкапсулировать ее внутри класса. Это позволило нам преобразовать операции из методов в расширения и переместить их определения ближе к тому месту, где они используются. Поскольку наш класс инкапсулировал единую коллекцию, мы смогли полностью исключить класс и заменить его псевдонимом типа.

Немутирующие коллекции и расширения позволяют нам организовывать код способами, недоступными в Java. Мы можем сгруппировать всю логику, требуемую определенной функцией приложения, в одном модуле, независимо от классов домена, к которым применяется эта логика. Однако если бы мы хотели, чтобы методы этих классов домена были полиморфными методами, нам пришлось бы определять их в этих классах, а не в нашем функциональном модуле. В *главе 18 «От открытых классов к запечатанным»* мы рассмотрим закрытые классы — альтернативу объектно-ориентированному полиморфизму, которая более удобна при определении иерархии типов в одной части кода и операциях над этими типами в другой.

Наконец, обратите внимание, что повторное использование встроенных типов — таких как `List`, вместо определения конкретного типа не обходится без затрат. Мы могли бы хранить элементы в `List` как детали имплементации, а не как выбор моделирования. Также намного проще «найти способы использования» определенного класса-оболочки, чем обобщенную специализацию. Тем не менее стандартные типы коллекций широко распространены, потому что они являются действительно хорошими абстракциями — настолько хорошими, что мы, как правило, не должны их скрывать. В *главе 22 «От классов к функциям»* рассматривается, что произойдет, если мы возьмем эту идею на вооружение и будем следовать ей.

ГЛАВА 16

От интерфейсов к функциям

В Java мы используем интерфейсы для указания контракта между участком кода, который определяет некоторую функциональность, и кодом, который в ней нуждается. Эти интерфейсы соединяют обе стороны вместе, что может усложнить поддержку нашего программного обеспечения. Как типы функций помогают решить эту проблему?

Представьте, что вам нужно отправить электронное письмо из какого-то кода, который вы пишете. Всего лишь только это — не получить почту или открыть список отправленных сообщений — просто отправить письмо и забыть. Код, описывающий электронное письмо, достаточно прост:

```
data class Email(
    val to: EmailAddress,
    val from: EmailAddress,
    val subject: String,
    val body: String
)
```

Написав этот код, мы *хотели бы* вызвать простейшую возможную функцию для его отправки:

```
fun send(email: Email) {
    ...
}
```

Но, приступив к реализации этой функции, мы обнаружим, что для фактической отправки электронного письма нам требуется еще и другая информация. Не информация о самом электронном письме, а скорее конфигурация его отправки. Такие сведения, как имя хоста отправляющего сервера и учетные данные безопасности — все то, чего не знает ваш не обладающий техническими знаниями родственник, которому вы настраиваете его новый компьютер. Так что мы добавим три дополнительных параметра в `sendEmail`, чтобы поддержать всю эту конфигурацию:

```
fun sendEmail(
    email: Email,
    serverAddress: InetAddress,
    username: String,
    password: String
) {
    ...
}
```

С точки зрения клиента, все просто стало намного менее удобным. Везде, где требуется отправить электронное письмо, должна быть известна эта конфигурация. Нам придется передавать ее сверху вниз по кодовой базе. Решение этой проблемы путем сокрытия деталей в глобальных переменных работает нормально, пока мы не обнаружим, что каждый запуск набора модульных тестов теперь отправляет 50 электронных писем! Должен быть лучший способ скрыть эти мелкие детали.

Объектно-ориентированная инкапсуляция

Объектно-ориентированные языки имеют готовое решение этой проблемы — объекты могут инкапсулировать данные:

```
class EmailSender(
    private val serverAddress: InetAddress,
    private val username: String,
    private val password: String
) {
    fun send(email: Email) {
        sendEmail(
            email,
            serverAddress,
            username,
            password
        )
    }
}
```

Теперь, когда мы хотим отправить электронное письмо, нам нужен доступ к `EmailSender` (а не к статической функции). Получив `EmailSender`, вместо вызова *функции* мы вызываем *метод*, и нам не нужно сообщать методу все мелкие детали, потому что он уже знает их — это поля его класса:

```
// Место, где описана конфигурация письма
val sender: EmailSender = EmailSender(
    inetAddress("smtp.travelator.com"),
    "username",
    "password"
)

// Место, куда мы его отправляем
    fun sendThanks() {
        sender.send(
            Email(
                to = parse("support@internationalrescue.org"),
                from = parse("support@travelator.com"),
                subject = "Thanks for your help",
                body = "..."
            )
```

```
        )
    }
}
```

Однако место, где описана конфигурация письма, и место, куда мы хотим его отправить, зачастую могут быть разделены в нашем коде многими слоями кода. Обычно в ООП `sender` записывается как свойство класса и пользуется его методами:

```
// Место, где описана конфигурация письма
val subsystem = Rescuing(
    EmailSender(
        inetAddress("smtp.travelator.com"),
        "username",
        "password"
    )
)
// Место, куда мы его отправляем
class Rescuing(
    private val emailSender: EmailSender
) {
    fun sendThanks() {
        emailSender.send(
            Email(
                to = parse("support@internationalrescue.org"),
                from = parse("support@travelator.com"),
                subject = "Thanks for your help",
                body = "..."
            )
        )
    }
}
```

Мы также часто будем извлекать интерфейс:

```
interface ISendEmail {
    fun send(email: Email)
}

class EmailSender(
        ...
) : ISendEmail {
    override fun send(email: Email) {
        sendEmail(
            email,
            serverAddress,
            username,
            password
        )
    }
}
```

Если наш клиентский код будет зависеть от интерфейса `ISendEmail`, а не от класса `EmailSender`, мы сможем сконфигурировать тесты, использующие поддельные реализации `ISendEmail`, который на самом деле не станет отправлять электронные письма, но вместо этого позволит нам проверить, что будет отправлено, если мы инициируем отправку.

Мы можем предоставить не только подделки (имитации), которые вообще не отправляют электронные письма, но и различные подлинные реализации — такие как `SmtpEmailSender` и `X400EmailSender`, которые скрывают от своих клиентов как собственную конфигурацию, так и реализацию. Мы начали с сокрытия информации, а пришли к сокрытию реализации.

Когда мы говорим о *сокрытии*, это звучит немного уничижительно, но сокрытие полезно как для клиента, так и для разработчика. У первого нет проблемы с необходимостью предоставления деталей конфигурации в момент ее использования. Последний способен развиваться отдельно от своих пользователей (при условии, что он не изменяет API, выраженный в интерфейсе).

Прежде чем покинуть объектно-ориентированную область, обратите внимание, что для реализации `ISendEmail` нет необходимости создавать именованный класс. Мы можем сделать это анонимно:

```
fun createEmailSender(
    serverAddress: InetAddress,
    username: String,
    password: String
): ISendEmail =
    object : ISendEmail {
        override fun send(email: Email) =
            sendEmail(
                email,
                serverAddress,
                username,
                password
            )
    }
```

Почему нам может это понадобиться? Дело в том, что когда мы не контролируем всех клиентов нашего кода (например, публикуем библиотеку, внешнюю по отношению к нашей организации), это дает нам гибкость в изменении нашей реализации: зная, что клиенты не могут зависеть от конкретного класса реализации, мы можем выполнять понижающее приведение к нему и вызывать другие методы. Мы называем объект, который возвращаем здесь, *замыканием* (closure), потому что он как бы «замыкает» значения, которые требует из своего окружающего контекста (вызов функции), фиксируя их для последующего использования.

В Kotlin 1.4 допускается объявить наш интерфейс `ISendEmail` в виде `fun interface` (один и только с одним абстрактным методом). Таким образом, мы можем определить имплементацию одной операции с помощью лямбды, а не с помощью объекта с одним методом:

```
fun interface ISendEmail {
    fun send(email: Email)
}

fun createEmailSender(
    serverAddress: InetAddress,
    username: String,
    password: String
) = ISendEmail { email ->
    sendEmail(
        email,
        serverAddress,
        username,
        password
    )
}
```

Опять же, лямбда здесь является замыканием, фиксирующим значения параметров его заключающей функции.

Функциональная инкапсуляция

Мы увидели, как ОО-программист решает проблему инкапсуляции надоедливых деталей, чтобы клиентам не приходилось предоставлять их в момент использования, но как бы к той же проблеме подошел функциональный программист?

Помните, что мы пытаемся добраться до функции с такой сигнатурой:

```
fun send(email: Email) {
    ...
}
```

Но на самом деле, чтобы отправить сообщение, нам нужна вся следующая информация:

```
fun sendEmail(
    email: Email,
    serverAddress: InetAddress,
    username: String,
    password: String
) {
    ...
}
```

В функциональном плане это пример *частичного применения* (**https://oreil.ly/V1KOm**) — фиксация некоторых аргументов функции для получения функции с меньшим количеством аргументов. Хотя некоторые языки предоставляют для этого встроенную поддержку, в Kotlin самый простой подход — написать функцию для частичного применения нашей конфигурации.

То есть нам нужна функция, которая принимает конфигурацию и возвращает функцию, содержащую информацию о том, как отправить электронное письмо:

```
fun createEmailSender(
    serverAddress: InetAddress,
    username: String,
    password: String
): (Email) -> Unit { ❶
    ...
}
```

❶ Тип возвращаемого нашей функции сам по себе является функцией, которая принимает `Email` и возвращает `Unit`.

Итак, `createEmailSender` — это конструктор. Не конструктор класса, а функция, выполняющая ту же роль. И `createEmailSender`, и `::EmailSender` — это функции, которые возвращают объект, который знает, как отправить сообщение.

Чтобы увидеть, как это работает в функциях, можно сначала записать это от руки, определив внутреннюю функцию, которая фиксирует аргументы, требуемые от родительского элемента:

```
fun createEmailSender(
    serverAddress: InetAddress,
    username: String,
    password: String
): (Email) -> Unit {

    fun result(email: Email) {
        sendEmail(
            email,
            serverAddress,
            username,
            password
        )
    }
    return ::result
}
```

Затем мы можем превратить результат в лямбда-выражение:

```
fun createEmailSender(
    serverAddress: InetAddress,
    username: String,
    password: String
): (Email) -> Unit {

    val result: (Email) -> Unit =
        { email ->
            sendEmail(
                email,
                serverAddress,
```

```
                username,
                password
            )
        }
    return result
}
```

Если заменить `result` встроенным выражением и преобразовать всю функцию в однострочное выражение, у нас останется это функциональное определение:

```
fun createEmailSender(
    serverAddress: InetAddress,
    username: String,
    password: String
): (Email) -> Unit =
    { email ->
        sendEmail(
            email,
            serverAddress,
            username,
            password
        )
    }
```

Здесь `createEmailSender` — функция, которая возвращает лямбда-выражение, вызывающее `sendEmail`, комбинируя единственный аргумент `Email` лямбда-выражения с конфигурацией из собственных параметров. То есть представляет собой замыкание в функциональной области, и не случайно, что все это очень похоже на версии ООП с определением `fun interface` или `object`.

Чтобы использовать эту функцию, можно создать ее в одном месте и вызвать в другом точно так же, как мы делали в случае с объектным решением:

```
// Место, где описана конфигурация письма
val sender: (Email) -> Unit = createEmailSender(
    inetAddress("smtp.travelator.com"),
    "username",
    "password"
)
// Место, куда мы его отправляем
fun sendThanks() {
    sender( ❶
        Email(
            to = parse("support@internationalrescue.org"),
            from = parse("support@travelator.com"),
            subject = "Thanks for your help",
            body = "..."
        )
    )
}
```

❶ Здесь спрятан неявный вызов `invoke`.

Это те же самые формы, которые мы применяли в случае с ООП (если заменим спрятанный `invoke` на `send`):

```
fun sendThanks() {
    sender.send(
        Email(
            to = parse("support@internationalrescue.org"),
            from = parse("support@travelator.com"),
            subject = "Thanks for your help",
            body = "..."
        )
    )
}
```

В том маловероятном случае, если вы присоединяетесь к нам, придя из JavaScript или Clojure, функциональная форма будет вам знакома, но если вы пришли в Kotlin из Java, то это решение, вероятно, покажется вам весьма странным.

Функциональные типы в Java

Как объектные, так и функциональные формы позволили нам инкапсулировать данные (в рассмотренном случае — конфигурацию электронного письма, но с таким же успехом это могла бы быть информация о сотрудниках), чтобы перенести их из места, где они записаны, в место, где они используются. Это может сделать любая структура данных, но поскольку и объект, и функция имеют операцию, которую можно выполнить (`send` и `invoke` соответственно), клиент может не обращать внимания на детали конфигурации, а просто передавать информацию, специфичную для каждого вызова (`Email`).

Один из способов объединения функциональных и ОО-решений — рассматривать функцию как объект с помощью одного метода `invoke`. Это именно то, что сделала Java 8, когда ввела лямбды. Для ссылки на тип функции Java использует интерфейсы с *единым абстрактным методом* (Single Abstract Method, SAM), у которого есть желаемая сигнатура. Лямбды в Java — это специальный синтаксис для реализации интерфейса SAM. Среда выполнения Java определяет интерфейсы SAM, именуемые по их роли: `Consumer`, `Supplier`, `Function`, `BiFunction`, `Predicate` и т. д. Он также предоставляет примитивные специализации — такие как `DoublePredicate`, чтобы избежать проблем с упаковыванием.

Выразим на Java наше функциональное решение:

```
// Место, где описана конфигурация письма
Consumer<Email> sender = createEmailSender(
    inetAddress("example.com"),
    "username",
    "password"
);
```

```
// Место, куда мы его отправляем
public void sendThanks() {
    sender.accept( ❶
        new Email(
            parse("support@internationalrescue.org"),
            parse("support@travelator.com"),
            "Thanks for your help",
            "..."
        )
    );
}
```

❶ `accept` — это имя единственного абстрактного метода в интерфейсе `Consumer`.

`createEmailSender` может быть реализован с помощью лямбда:

```
static Consumer<Email> createEmailSender(
    InetAddress serverAddress,
    String username,
    String password
) {
    return email -> sendEmail(
        email,
        serverAddress,
        username,
        password
    );
}
```

Это эквивалентно созданию анонимной реализации интерфейса — метод, который очень хорошо знаком тем из нас, кто программировал на Java до Java 8:

```
static Consumer<Email> createEmailSender(
    InetAddress serverAddress,
    String username,
    String password
) {
    return new Consumer<Email>() {
        @Override
        public void accept(Email email) {
            sendEmail(
                email,
                serverAddress,
                username,
                password
            );
        }
    };
}
```

Мы говорим: «Это эквивалентно созданию анонимной реализации интерфейса» — но «под капотом» реализация более сложная, поскольку надо уйти от ненужного определения классов и создания экземпляров объектов.

Обратите внимание, что мы не можем назначить результат `(Email) -> Unit` функции Kotlin `createEmailSender` переменной типа `Consumer<Email>`. Это связано с тем, что среда выполнения Kotlin использует свои собственные типы функций, а компилятор компилирует `(Email) -> Unit` в `Function1<Email, Unit>`. Существует целая серия интерфейсов `FunctionN` в Kotlin для различного количества параметров.

Поскольку интерфейсы несовместимы, чтобы смешать Java и Kotlin на этом функциональном уровне, нам иногда придется потрудиться. Зададим тип функции Kotlin `(Email) -> Unit`:

```
// Тип функции Kotlin
val sender: (Email) -> Unit = createEmailSender(
    inetAddress("smtp.travelator.com"),
    "username",
    "password"
)
```

Мы не можем просто назначить `sender` для `Consumer<Email>`:

```
val consumer: Consumer<Email> = sender // Не компилируется ❶
```

❶ Несоответствие типов. Требуется: `Consumer<Email>` Found: `(Email) -> Unit`.

Однако мы можем осуществить преобразование с помощью лямбды:

```
val consumer: Consumer<Email> = Consumer<Email> { email ->
    sender(email)
}
```

Существует ситуация, когда нам не понадобится это преобразование. Она заключается в вызове метода Java, принимающего параметр SAM, — например, этот конструктор:

```
class Rescuing {
    private final Consumer<Email> emailSender;

    Rescuing(Consumer<Email> emailSender) {
        this.emailSender = emailSender;
    }
    ...
}
```

Здесь компилятор *может* преобразовать `(Email) -> Unit` в `Consumer<Email>`, потому что Kotlin автоматически преобразует параметры, так что мы можем написать:

```
Rescuing(sender)
```

Сочетание и совмещение

У абстракции есть две стороны: клиентский код и код реализации. До сих пор и клиент, и разработчик были либо объектно-ориентированными, либо функциональными. В случае ООП поля содержат конфигурацию, а клиент вызывает метод. В функциональной схеме функция замыкается поверх конфигурации, а клиент вызывает функцию.

Можем ли мы объединить эти подходы, передав выполнение ООП клиенту, ожидающему функции, или наоборот? Или, выражаясь языком Kotlin, можем ли мы преобразовать `ISendEmail` в `(Email) -> Unit` и обратно? Что ж, да, можем!

Помните, что в Java и Kotlin типы функций — это просто интерфейсы. Из-за этого `EmailSender` может имплементировать соответственно тип `Consumer<Email>` или `(Email) -> Unit`, определив метод с сигнатурой типа функции.

Итак, на Java мы можем написать:

```
public class EmailSender
    implements ISendEmail,
        Consumer<Email> ❶
{
        ...
    @Override
    public void accept(Email email) { ❷
        send(email);
    }

    @Override
    public void send(Email email) {
        sendEmail(email, serverAddress, username, password);
    }
}
```

❶ Объявление.

❷ Имплемент.

Эквивалент этого на Kotlin:

```
class EmailSender(
        ...
) : ISendEmail,
    (Email) -> Unit ❶
{
    override operator fun invoke(email: Email) =
        send(email) ❷

    override fun send(email: Email) {
        sendEmail(
            email,
```

```
            serverAddress,
            username,
            password
        )
    }
}
```

❶ Объявление.

❷ Имплемент.

Если мы сделаем это, то сможем использовать отправитель на основе класса вместо функционального. Теперь мы будем придерживаться Kotlin:

```
// Место, где описана конфигурация письма
val sender: (Email) -> Unit = EmailSender(
    inetAddress("smtp.travelator.com"),
        "username",
        "password"
)
// Место, куда мы его отправляем
fun sendThanks() {
    sender( ❶
        Email(
            to = parse("support@internationalrescue.org"),
            from = parse("support@travelator.com"),
            subject = "Thanks for your help",
            body = "..."
        )
    )
}
```

❶ Здесь есть неявная функция `invoke`.

Теперь наша объектно-ориентированная реализация получила метод `invoke`, который соответствует функциональному подходу. Это ставит под сомнение полезность интерфейса `ISendEmail`. Мы видим, что он эквивалентен типу функции `(Email) -> Unit`. Все, что он делает, — это присваивает имя `send` тому, что происходит, когда вы его вызываете. Может быть, мы могли бы просто использовать тип `(Email) -> Unit` везде вместо `ISendEmail`?

Если вы считаете, что это недостаточно выразительно, то, возможно, вы не функциональный программист. К счастью, есть золотая середина: мы можем использовать *псевдоним типа*, чтобы дать имя функциональному типу, сообщая таким образом о наших намерениях:

```
typealias EmailSenderFunction = (Email) -> Unit

class EmailSender(
    ...
```

```
) : EmailSenderFunction {
    override fun invoke(email: Email) {
        sendEmail(
            email,
            serverAddress,
            username,
            password
        )
    }
}
```

На самом деле мы, вероятно, назвали бы `EmailSenderFunction` как `EmailSender`. Здесь же мы дали ему другое название, чтобы избежать путаницы с версией ООП, но тот факт, что мы хотим называть их одинаково, показывает, что они служат одной и той же цели, с точки зрения клиента.

Выразительные типы функций

Как только мы придем к функциональному мышлению, тип `(Email) -> Unit` может стать достаточно выразительным, чтобы рассказать нам о роли функции, — особенно, когда она привязана к переменной с именем `sender`. Что вы можете сделать с электронным письмом, которое не возвращает результат? Что ж, если оно не возвращает результат, это должно быть действие (см. *разд. «Действия» главы 7*), но удаление электронного письма также может иметь аналогичную сигнатуру. Объекты имеют это преимущество перед функциями — они могут именовать свои методы (`send(email)`, очевидно, отличается от `delete(email)`) — за счет того, что в них также необходимо указывать тип объекта или интерфейса.

В Kotlin есть особенность, которая может помочь сделать типы функций более выразительными, — возможность именовать параметры. Так что, если это поможет, мы могли бы написать: `(toSend: Email) -> Unit`. Здесь это не имеет большого значения, но может быть очень полезно в ситуациях с параметрами одного и того же типа — например: `(username: String, password: String) -> AuthenticatedUser`. Если мы используем эту форму, IntelliJ даже поможет назвать параметры, когда мы реализуем функцию с помощью лямбды или класса.

Есть еще один способ преодолеть разрыв между ООП и ФП — он не требует, чтобы наши классы реализовывали типы функций: создайте ссылку на функцию в точке ее применения. Перед вами наше старое решение на основе классов:

```
class EmailSender(
    private val serverAddress: InetAddress,
    private val username: String,
    private val password: String
) {
    fun send(email: Email) {
        sendEmail(
            email,
            serverAddress,
            username,
            password
        )
    }
}
```

Мы можем преобразовать экземпляр `EmailSender` в тип функции с помощью лямбда-выражения:

```
val instance = EmailSender(
    inetAddress("smtp.travelator.com"),
    "username",
    "password"
)
val sender: (Email) -> Unit = { instance.send(it) }
```

или просто использовать ссылку на метод:

```
val sender: (Email) -> Unit = instance::send
```

Хотя мы показали эти преобразования в Kotlin, они также работают в Java (с немного другим синтаксисом). Они работают и с методом `send` в интерфейсе `ISendEmail`, хотя неясно, что этот интерфейс делает для нас, если мы используем тип функции.

Можем ли мы сделать обратное и передать нашего функционального отправителя во что-то, что ожидает `ISendEmail`? Это требует дополнительных церемоний, потому что для выполнения перехода мы должны создать анонимный объект, реализующий `IsendEmail`:

```
val function: (Email) -> Unit = createEmailSender(
    inetAddress("smtp.travelator.com"),
    "username",
    "password"
)

val sender: ISendEmail = object : ISendEmail {
    override fun send(email: Email) {
        function(email)
    }
}
```

Если бы мы использовали `fun interface` Kotlin 1.4, то могли бы снова удалить некоторые шаблоны:

```
fun interface ISendEmail {
    fun send(email: Email)
}

val sender = ISendEmail { function(it) }
```

Сравнение подходов

Давайте вспомним о подходе ООП. Сначала мы определяем тип:

```
class EmailSender(
    private val serverAddress: InetAddress,
```

```
    private val username: String,
    private val password: String
) {
    fun send(email: Email) {
        sendEmail(
            email,
            serverAddress,
            username,
            password
        )
    }
}
```

Затем создаем экземпляры и вызываем методы:

```
// Место, где описана конфигурация письма
val sender: EmailSender = EmailSender(
    inetAddress("smtp.travelator.com"),
    "username",
    "password"
)
// Место, куда мы его отправляем
fun sendThanks() {
    sender.send(
        Email(
            to = parse("support@internationalrescue.org"),
            from = parse("support@travelator.com"),
            subject = "Thanks for your help",
            body = "..."
        )
    )
}
```

В функциональной среде нам не нужно определять тип, потому что `(Email) -> Unit` изначально существует (т. е. предоставляется средой выполнения), поэтому мы можем просто написать:

```
// Место, где описана конфигурация письма
val sender: (Email) -> Unit = createEmailSender(
    inetAddress("smtp.travelator.com"),
    "username",
    "password"
)
// Место, куда мы его отправляем
fun sendThanks() {
    sender( ❶
        Email(
            to = parse("support@internationalrescue.org"),
            from = parse("support@travelator.com"),
```

```
            subject = "Thanks for your help",
            body = "..."
        )
    )
}
```

❶ С или без `invoke`.

Клиенты, использующие *объект*, должны знать о вызове метода `send` для отправки электронной почты. В отличие от них, клиенты, использующие *функцию*, просто должны вызвать ее, но они знают лишь то, что функция отправляет электронное письмо, потому что ей присвоено имя `sender`. Если это имя затерялось в иерархии вызовов, нам остается гадать, что произойдет, исходя из сигнатуры функции.

Плата за услугу ОО-клиентов, которые должны знать, как вызывать метод `send`, заключается в том, что мы можем упаковать ряд операций, связанных с электронной почтой, в систему `EmailSystem` с такими методами, как `send`, `list` и `delete`, и передать все эти функции клиентам на одном дыхании. Затем клиенты могут выбрать, что им нужно и в каком контексте:

```
interface EmailSystem {
    fun send(email: Email)
    fun delete(email: Email)
    fun list(folder: Folder): List<Email>
    fun move(email: Email, to: Folder)
}
```

Для достижения этой функциональной цели потребуется либо передача отдельных функций, либо какая-то карта имен для функции — возможно, экземпляр самого класса:

```
class EmailSystem(
    val send: (Email) -> Unit,
    val delete: (Email) -> Unit,
    val list: (folder: Folder) -> List<Email>,
    val move: (email: Email, to: Folder) -> Unit
)
```

Учитывая такой объект, клиенты могли бы относиться к нему так, как к реализации интерфейса:

```
fun sendThanks(sender: EmailSystem) {
    sender.send(
        Email(
            to = parse("support@internationalrescue.org"),
            from = parse("support@travelator.com"),
            subject = "Thanks for your help",
            body = "..."
        )
    )
}
```

Но это не то же самое, что код ООП. Вместо вызова *метода* `send` на самом деле происходит вот что: мы вызываем `getSender` для доступа к свойству типа функции, а затем вызываем `invoke` для этой функции:

```
fun sendThanks(sender: EmailSystem) {
    sender.send.invoke(
        Email(
            to = parse("support@internationalrescue.org"),
            from = parse("support@travelator.com"),
            subject = "Thanks for your help",
            body = "..."
        )
    )
}
```

Этот код, может, и выглядит очень похоже, но генерирует он совершенно другой и принципиально несовместимый байт-код.

Соединение

Тонкое различие между выражением зависимости либо как реализации `ISendEmail`, либо как реализации типа функции `(Email) -> Unit` заключается в соединении клиента и реализации, в частности, когда они находятся в разных модулях кода.

Функция `ISendEmail` должна быть где-то определена. Клиент не может определить ее, потому что исполнитель будет зависеть от интерфейса, а клиент — от реализации, что приведет к циклической зависимости. Таким образом, интерфейс должен быть определен либо вместе с реализацией, либо в отдельном расположении (пакете или файле JAR), от которого зависит как реализация, так и ее клиенты. Последнее — применение *принципа инверсии зависимостей* (dependency inversion principle, **https://oreil.ly/AcrWj**), теоретически предпочтительнее, но на практике требует больше работы, и поэтому им часто пренебрегают.

С инверсией зависимостей или без нее результатом является то, что клиент и реализация связаны интерфейсом таким образом, что это может затруднить работу с системами и их рефакторинг. Любое изменение методов в `EmailSystem` может повлиять на весь код, зависящий от интерфейса.

Напротив, в функциональном мире среда выполнения определяет все типы функций, поэтому во время компиляции они не создают зависимости между клиентом и реализацией. В отличие от `ISendEmail`, которую мы должны куда-то определить, `(Email) -> Unit` (или `Consumer<Email>` в Java) является частью языка. Конечно, будет существовать зависимость во время выполнения — код конструктора должен быть виден там, где создается зависимость, и клиент должен иметь возможность вызывать код реализации, — но это приводит к ухудшению качества соединения. Например, когда зависимость выражается как тип функции, мы можем переименовать `EmailSystem.send`, и единственным изменением в нашем клиентском коде будет использование другой ссылки на метод. Внутренние компоненты `sendThanks` при этом не затрагиваются.

Передавайте только те типы, которые принадлежат вам или которые определяет среда выполнения

Раннее эмпирическое правило для ОО-систем заключалось в том, что внутри наших систем мы должны программировать в терминах тех типов, которые нам принадлежат, а не тех, которые предоставляются библиотеками. Тогда мы будем изолированы от изменений, которые не контролируем, и с большей вероятностью напишем код с возможностью повторного его использования в разных реализациях.

Исключением из этого правила является зависимость от типов, предоставляемых средой выполнения, — маловероятно, что они изменятся. Типы функций дают нам возможность легко переходить от нестабильных интерфейсов к стабильным, позволяя частям наших систем развиваться с разной скоростью.

Объектно-ориентированный или функциональный?

Как объектно-ориентированный, так и функциональный подход может достигать одних и тех же целей и с одинаковым уровнем выразительности. И какой же нам выбрать?

Давайте рассмотрим это в контексте клиентского кода. Если нашему клиенту нужно только увидеть список электронной почты, то у него должна быть зависимость от единственной функции: `(Folder) -> List<Email>`. В этом случае он не связан с реализацией, и зависимость может быть выполнена любым способом, реализующим тип функции, включая:

- простую функцию;
- объект, реализующий тип функции;
- ссылку на метод, выбирающую метод с желаемой сигнатурой;
- лямбда-выражение с желаемой сигнатурой.

Даже если у нас уже есть интерфейс — скажем, `EmailSystem`, который определяет желаемый метод вместе с `send`, `move` и `delete`:

```
interface EmailSystem {
    fun send(email: Email)
    fun delete(email: Email)
    fun list(folder: Folder): List<Email>
    fun move(email: Email, to: Folder)
}
```

мы не должны без необходимости подключать нашего клиента к этому интерфейсу, когда тип функции будет выполнять следующее:

```
class Organiser(
    private val listing: (Folder) -> List<Email>
) {
    fun subjectsIn(folder: Folder): List<String> {
```

```
        return listing(folder).map { it.subject }
    }
}

val emailSystem: EmailSystem = ...
val organiser = Organiser(emailSystem::list)
```

В зависимости от более широкого интерфейса здесь упускается возможность точно сообщить, какие операции нам требуются, что заставляет клиентов предоставлять реализацию всего интерфейса. Это особенно раздражает в тестах, где нам придется вводить поддельные объекты только для того, чтобы скомпилировать наш тестовый код.

Средства связи и урезанное соединение настолько сильны, что даже если нашему клиенту необходимо отправлять и удалять электронную почту, а на практике это будет предоставляться одной `EmailSystem`, клиент, вероятно, должен зависеть от двух функций, а не от интерфейса:

```
class Organiser(
    private val listing: (Folder) -> List<Email>,
    private val deleting: (Email) -> Unit
) {
    fun deleteInternal(folder: Folder) {
        listing(rootFolder).forEach {
            if (it.to.isInternal()) {
                deleting.invoke(it)
            }
        }
    }
}

val organiser = Organiser(
    emailSystem::list,
    emailSystem::delete
)
```

И только когда клиенту требуются три связанные операции, создается впечатление, что по умолчанию должен использоваться мультиметодный интерфейс:

```
class Organiser(
    private val emails: EmailSystem
) {
    fun organise() {
        emails.list(rootFolder).forEach {
            if (it.to.isInternal()) {
                emails.delete(it)
            } else {
                emails.move(it, archiveFolder)
            }
```

```
        }
    }
}

val organiser = Organiser(emailSystem)
```

Даже здесь клиенту, возможно, было бы лучше принять объект, который поддерживает только нужные операции. Мы можем сделать это на основе нового интерфейса (здесь `Dependencies`), реализованного с помощью `object`:

```
class Organiser(
    private val emails: Dependencies
) {
    interface Dependencies {
        fun delete(email: Email)
        fun list(folder: Folder): List<Email>
        fun move(email: Email, to: Folder)
    }

    fun organise() {
        emails.list(rootFolder).forEach {
            if (it.to.isInternal()) {
                emails.delete(it)
            } else {
                emails.move(it, archiveFolder)
            }
        }
    }
}

val organiser = Organiser(object : Organiser.Dependencies {
    override fun delete(email: Email) {
        emailSystem.delete(email)
    }

    override fun list(folder: Folder): List<Email> {
        return emailSystem.list(folder)
    }

    override fun move(email: Email, to: Folder) {
        emailSystem.move(email, to)
    }
})
```

Выглядит это малоприятно... Возможно, здесь было бы лучше использовать класс функций:

```
class Organiser(
    private val emails: Dependencies
```

```
) {
    class Dependencies(
        val delete: (Email) -> Unit,
        val list: (folder: Folder) -> List<Email>,
        val move: (email: Email, to: Folder) -> Unit
    )

    fun organise() {
        emails.list(rootFolder).forEach {
            if (it.to.isInternal()) {
                emails.delete(it)
            } else {
                emails.move(it, archiveFolder)
            }
        }
    }
}

val organiser = Organiser(
    Organiser.Dependencies(
        delete = emailSystem::delete,
        list = emailSystem::list,
        move = emailSystem::move
    )
)
```

Итак, пока это не станет слишком трудной задачей, мы должны по умолчанию выражать потребности нашего клиента в виде типов функций. Тогда наша реализация может быть просто функцией, или чем-то, реализующим тип функции, или методом, преобразованным в тип функции с помощью ссылок на методы или лямбд, — в зависимости от того, что имеет наибольший смысл в контексте.

Наследие Java

Хотя ранее мы говорили, что «наша среда выполнения определяет все типы функций», для Java это было не так, пока в Java 8 не были представлены `Supplier`, `Consumer`, `Predicate` и т. д., а также возможность реализовать их с помощью ссылок на методы или лямбда-выражений.

Из-за этого устаревший код Java обычно выражает зависимости с помощью тех же мультиметодных интерфейсов, которые мы использовали бы для группировки их по подсистемам (например, `EmailSystem`), даже если для реализации функциональности требуется только один из указанных методов.

Это приводит к проблемам со связью, описанным ранее, а также к необходимости имитировать (или, попросту говоря, подделывать) фреймворки для создания тестовых реализаций широких интерфейсов, где на практике будет вызываться только один метод. Такие имитации (моки) впоследствии приводят к преждевременному

сбою функциональности, если вы вызываете метод, который не собирались вызывать, — проблеме, которая была бы решена во время компиляции, если бы одна функция выражала зависимость.

Как только мы внедрим в нашу кодовую базу фиктивный фреймворк (или, чаще, два или три фиктивных фреймворка на любой вкус), они позволят нам решать проблемы создания реализаций для неиспользуемых методов и заглушки взаимодействий с внешними системами. Однако обычно наш код улучшают путем реструктуризации, чтобы избежать необходимости в имитациях (моках). Одним из примеров является выражение зависимостей в виде типов функций, другим — перенос взаимодействия с внешними системами на внешние уровни нашего кода, как будет показано в *главе 20*. А в *главе 17* рассматриваются пути сокращения использования моков путем рефакторинга наших тестов в более функциональную форму.

Прослеживаемость

У выражения зависимостей с помощью типов функций есть обратная сторона, и это обычная проблема с добавлением уровня косвенности. Если мы используем IntelliJ для поиска абонентов, вызывающих `EmailSystem.send`, след обрывается в том месте, где `EmailSystem::send` преобразуется в `(Email) -> Unit`. Среда IDE не знает, что вызовы функции на самом деле вызывают метод. Это похоже на ситуацию, когда беглец движется по реке, и отряд, выслеживающий его, должен прочесать оба берега вверх и вниз по течению, чтобы найти, где он выйдет.

Такова цена, которую мы платим также и за косвенное обращение с вызовами методов, но наш инструментарий для этого пригоден и может как минимум найти все места, где реализован конкретный метод и где реализация вызывается через интерфейс. Как и в случае использования неинкапсулированных коллекций (см. *главу 15*), цена, которую мы платим за разделение и универсальность, сводится к тому, что у инструментов и разработчиков остается меньше контекста для анализа. Мы надеемся, что поддержка IDE улучшит функциональный анализ, и в то же время мы можем помочь ему, если станем передавать типы функций туда, где они используются, не слишком далеко от того места, где они инициализируются.

Рефракторинг от интерфейсов к функциям

Программа `Travelator` хорошо сконструирована в стиле Java с интерфейсами, выражающими отношения между компонентами. Например, движок `Recommendations` зависит от `FeaturedDestinations` и `DistanceCalculator` (пример 16.1).

Пример 16.1 [interfaces-to-funs.0:src/main/java/travelator/recommendations/Recommendations.java]

```
public class Recommendations {
    private final FeaturedDestinations featuredDestinations;
    private final DistanceCalculator distanceCalculator;
```

```
    public Recommendations(
        FeaturedDestinations featuredDestinations,
        DistanceCalculator distanceCalculator
    ) {
        this.featuredDestinations = featuredDestinations;
        this.distanceCalculator = distanceCalculator;
    }
    ...
}
```

Интерфейс `FeaturedDestinations` содержит несколько методов, объединяющих функциональность, которая обеспечивает доступ к удаленной службе (пример 16.2).

Пример 16.2 [interfaces-to-funs.0:src/main/java/travelator/destinations/FeaturedDestinations.java]

```
public interface FeaturedDestinations {
    List<FeaturedDestination> findCloseTo(Location location);
    FeaturedDestination findClosest(Location location);

    FeaturedDestination add(FeaturedDestinationData destination);
    void remove(FeaturedDestination destination);
    void update(FeaturedDestination destination);
}
```

Похоже, мы уже преобразовали интерфейс `DistanceCalculator` в Kotlin. Он тоже содержит более одного метода и скрывает другую внешнюю службу (пример 16.3).

Пример 16.3 [interfaces-to-funs.0:src/main/java/travelator/domain/DistanceCalculator.kt]

```
interface DistanceCalculator {
    fun distanceInMetersBetween(
        start: Location,
        end: Location
    ): Int

    fun travelTimeInSecondsBetween(
        start: Location,
        end: Location
    ): Int
}
```

Несмотря на зависимость в общей сложности от семи методов, `Recommendations` фактически использует для своей реализации только два из них (пример 16.4).

Пример 16.4 [interfaces-to-funs.0:src/main/java/travelator/recommendations/Recommendations.java]

```
public List<FeaturedDestinationSuggestion> recommendationsFor(
    Set<Location> journey
) {
    var results = removeDuplicates(
        journey.stream()
            .flatMap(location ->
                recommendationsFor(location).stream()
            )
    );
    results.sort(distanceComparator);
    return results;
}

public List<FeaturedDestinationSuggestion> recommendationsFor(
    Location location
) {
    return featuredDestinations
        .findCloseTo(location) ❶
        .stream()
        .map(featuredDestination ->
            new FeaturedDestinationSuggestion(
                location,
                featuredDestination,
                distanceCalculator.distanceInMetersBetween( ❷
                    location,
                    featuredDestination.getLocation()
                )
            )
        ).collect(toList());
}
```

❶ Метод для `FeaturedDestinations`.

❷ Метод для `DistanceCalculator`.

`RecommendationsTests` использует моки (имитации) для предоставления реализаций `DistanceCalculator` и `FeaturedDestinations`, которые отправляются в тестируемый экземпляр `Recommendations` (пример 16.5).

Пример 16.5 [interfaces-to-funs.0:src/test/java/travelator/recommendations/RecommendationsTests.java]

```
public class RecommendationsTests {

    private final DistanceCalculator distanceCalculator =
        mock(DistanceCalculator.class);
```

```
    private final FeaturedDestinations featuredDestinations =
        mock(FeaturedDestinations.class);

    private final Recommendations recommendations = new Recommendations(
        featuredDestinations,
        distanceCalculator
    );
    ...
}
```

Тесты указывают, что ожидаемые взаимодействия с моками выполняются с использованием двух методов: `givenFeaturedDestinationsFor` и `givenADistanceBetween`, которыми мы не будем вас утомлять (пример 16.6).

Пример 16.6 [interfaces-to-funs.0:src/test/java/travelator/recommendations/RecommendationsTests.java]

```
@Test
public void returns_recommendations_for_multi_location() {
    givenFeaturedDestinationsFor(paris,
        List.of(
            eiffelTower,
            louvre
        ));
    givenADistanceBetween(paris, eiffelTower, 5000);
    givenADistanceBetween(paris, louvre, 1000);

    givenFeaturedDestinationsFor(alton,
        List.of(
            flowerFarm,
            watercressLine
        ));
    givenADistanceBetween(alton, flowerFarm, 5300);
    givenADistanceBetween(alton, watercressLine, 320);

    assertEquals(
        List.of(
            new FeaturedDestinationSuggestion(alton, watercressLine, 320),
            new FeaturedDestinationSuggestion(paris, louvre, 1000),
            new FeaturedDestinationSuggestion(paris, eiffelTower, 5000),
            new FeaturedDestinationSuggestion(alton, flowerFarm, 5300)
        ),
        recommendations.recommendationsFor(Set.of(paris, alton))
    );
}
```

Введение функций

Прежде чем мы начнем переходить от интерфейсов к функциям, мы преобразуем `Recommendations` в Kotlin. Это класс, который в настоящее время выражает свои зависимости с помощью интерфейсов, а типы функций Kotlin менее неуклюжие, чем у Java.

Преобразование в Kotlin и применение способов рефакторинга, представленных в *главах 10* и *13*, показаны в примере 16.7.

Пример 16.7
[interfaces-to-funs.3:src/main/java/travelator/recommendations/Recommendations.kt]

```
class Recommendations(
    private val featuredDestinations: FeaturedDestinations,
    private val distanceCalculator: DistanceCalculator
) {
    fun recommendationsFor(
        journey: Set<Location>
    ): List<FeaturedDestinationSuggestion> =
        journey
            .flatMap { location -> recommendationsFor(location) }
            .deduplicated()
            .sortedBy { it.distanceMeters }

    fun recommendationsFor(
        location: Location
    ): List<FeaturedDestinationSuggestion> =
        featuredDestinations.findCloseTo(location)
            .map { featuredDestination ->
                FeaturedDestinationSuggestion(
                    location,
                    featuredDestination,
                    distanceCalculator.distanceInMetersBetween(
                        location,
                        featuredDestination.location
                    )
                )
            }
}

private fun List<FeaturedDestinationSuggestion>.deduplicated() =
    groupBy { it.suggestion }
        .values
        .map { suggestionsWithSameDestination ->
            suggestionsWithSameDestination.closestToJourneyLocation()
        }
```

```
private fun List<FeaturedDestinationSuggestion>.closestToJourneyLocation() =
    minByOrNull { it.distanceMeters } ?: error("Unexpected empty group")
```

Чтобы увидеть, как внутренние компоненты `Recommendations` будут использовать функцию, а не интерфейс, без необходимости изменять этот интерфейс, мы можем добавить свойство, инициализированное из метода интерфейса. В примере 16.8 мы добавляем свойство для `featuredDestinations::findCloseTo`, назвав его `destinationFinder`.

Пример 16.8 [interfaces-to-funs.4:src/main/java/travelator/recommendations/Recommendations.kt]

```
class Recommendations(
    private val featuredDestinations: FeaturedDestinations,
    private val distanceCalculator: DistanceCalculator
) {
    private val destinationFinder: ❶
        (Location) -> List<FeaturedDestination> =
        featuredDestinations::findCloseTo

    ...

    fun recommendationsFor(
        location: Location
    ): List<FeaturedDestinationSuggestion> =
        destinationFinder(location) ❷
            .map { featuredDestination ->
                FeaturedDestinationSuggestion(
                    location,
                    featuredDestination,
                    distanceCalculator.distanceInMetersBetween(
                        location,
                        featuredDestination.location
                    )
                )
            }
}
```

❶ Извлечение функции из интерфейса.

❷ Использование ее вместо метода.

Этот код проходит тесты, так что мы к чему-то приближаемся. Такое ощущение, что для перемещения `destinationFinder` в конструктор нужен рефакторинг, но мы не нашли ничего лучше, чем просто вырезать определение и вставить его туда, где нам хочется, чтобы оно было (пример 16.9).

Пример 16.9
[interfaces-to-funs.5:src/main/java/travelator/recommendations/Recommendations.kt

```
class Recommendations(
    private val featuredDestinations: FeaturedDestinations,
    private val distanceCalculator: DistanceCalculator,
    private val destinationFinder:
        (Location) -> List<FeaturedDestination> =
        featuredDestinations::findCloseTo
) {
```

Это, опять же, тот самый *expand*, о котором шла речь в *разд. «Рефракторинг с помощью Expand-and-Contract» главы 4*. К сожалению, Java не понимает параметр по умолчанию, поэтому нам приходится исправлять места вызовов, чтобы добавить аргумент функции. Впрочем, это именно то, чего мы действительно хотим (пример 16.10).

Пример 16.10 [interfaces-to-funs.5:src/test/java/travelator/recommendations/RecommendationsTests.java]

```
private final Recommendations recommendations = new Recommendations(
    featuredDestinations,
    distanceCalculator,
    featuredDestinations::findCloseTo
);
```

Теперь ничего в `Recommendations` не использует свойство `featuredDestinations`, следовательно — тот самый *contract* — мы можем удалить его (пример 16.11).

Пример 16.11
[interfaces-to-funs.6:src/main/java/travelator/recommendations/Recommendations.kt]

```
class Recommendations(
    private val distanceCalculator: DistanceCalculator,
    private val destinationFinder: (Location) -> List<FeaturedDestination>
) {
```

Места в нашем коде, которые создают `Recommendations`, теперь выглядят так (пример 16.12).

Пример 16.12 [interfaces-to-funs.6:src/test/java/travelator/recommendations/RecommendationsTests.java]

```
private final Recommendations recommendations = new Recommendations(
    distanceCalculator,
    featuredDestinations::findCloseTo
);
```

Если вы привыкли к рефакторингу тестов с помощью моков, вас может удивить, что тесты продолжали проходить этот рефакторинг. Мы можем предполагать, что они *должны* пройти — эффект вызова функции, привязанной к `featuredDestinations::findCloseTo`, по-прежнему заключается в вызове метода в интерфейсе-пустышке, — но наши предположения так часто при выполнении тестов оказывались неверными, что нам не стоит радоваться раньше времени.

Однако нам нравится единая структура, так что давайте проделаем то же самое с `distanceCalculator` — на этот раз одним махом, что бы там ни было (пример 16.13).

Пример 16.13 [interfaces-to-funs.7:src/main/java/travelator/recommendations/Recommendations.kt]

```
class Recommendations(
    private val destinationFinder: (Location) -> List<FeaturedDestination>,
    private val distanceInMetersBetween: (Location, Location) -> Int
) {
    ...
    fun recommendationsFor(
        location: Location
    ): List<FeaturedDestinationSuggestion> =
        destinationFinder(location)
            .map { featuredDestination ->
                FeaturedDestinationSuggestion(
                    location,
                    featuredDestination,
                    distanceInMetersBetween( ❶
                        location,
                        featuredDestination.location
                    )
                )
            }
}
```

❶ Вызов новой функции.

Вызовы конструктора теперь выглядят так (пример 16.14).

Пример 16.14 [interfaces-to-funs.7:src/test/java/travelator/recommendations/RecommendationsTests.java]

```
private final Recommendations recommendations = new Recommendations(
    featuredDestinations::findCloseTo,
    distanceCalculator::distanceInMetersBetween
);
```

Обратите внимание, что небольшое размышление о том, как называть функциональные переменные, может иметь большое значение для того, чтобы они казались

естественными в использовании, хотя иногда это слегка скрывает их от нашего внимания там, где они определены.

Опять же, тесты все еще проходят, что дает нам уверенность в том, что рабочий код увидит преобразование таким же образом. Особенно приятно, что мы показали возможность одновременного пересечения границы метода/функции и границы Java/Kotlin.

Может быть, это взаимодействие все-таки сработает хорошо!

Двигаемся дальше

Мы хотим, чтобы наш код был простым и гибким. С этой целью библиотекам необходимо скрывать детали реализации от клиентского кода, и мы хотим иметь возможность заменять одну реализацию некоторой функциональности другой.

В ООП мы прячем конфигурацию и реализацию внутри классов и выражаем заменяемую функциональность с помощью интерфейсов. В функциональном программировании функции выполняют обе роли. Мы могли бы рассматривать функцию как более фундаментальную, но мы также можем рассматривать объект как набор функций и функцию как объект с одним методом. И Kotlin, и Java позволяют нам перемещаться между областями на границах между реализациями и клиентами, но родной синтаксис типов функций Kotlin поощряет использование типов функций, а не интерфейсов. Это позволяет еще дальше уйти от определения наших собственных интерфейсов и должно стать нашим подходом по умолчанию.

Мы продолжим рефакторинг рассмотренного в этой главе примера и исследуем описанные здесь взаимосвязи в *главе 17 «От мокинга к маппингу»*.

ГЛАВА 17

От мокинга к маппингу

Моки (имитации) — это распространенный метод отделения объектно-ориентированного кода от его производственных зависимостей. Доступны ли в Kotlin решения лучше?

Это короткая бонусная глава, как бы продолжающая *главу 16*. Там было показано, что в наших тестах использовались моки (имитации), потому что с их помощью можно было реализовать два интерфейса с несколькими методами, хотя большинство из этих методов не задействовались. Мы ушли от рефакторинга, заменив зависимости от мультиметодных интерфейсов зависимостью только от двух операций, которые были необходимы для выполнения задачи. Тесты, однако, по-прежнему имитируют весь интерфейс, а затем передают ссылку на требуемые методы тестируемому объекту (`Recommendations`), как показано в примере 17.1.

Пример 17.1 [interfaces-to-funs.7:src/test/java/travelator/recommendations/RecommendationsTests.java]

```
public class RecommendationsTests {

    private final DistanceCalculator distanceCalculator =
        mock(DistanceCalculator.class);
    private final FeaturedDestinations featuredDestinations =
        mock(FeaturedDestinations.class);
    private final Recommendations recommendations = new Recommendations(
        featuredDestinations::findCloseTo,
        distanceCalculator::distanceInMetersBetween
    );
    ...
}
```

Тесты абстрагируются от мокинга над методами `givenFeaturedDestinationsFor` и `givenADistanceBetween` (пример 17.2).

Пример 17.2 [interfaces-to-funs.7:src/test/java/travelator/recommendations/RecommendationsTests.java]

```
@Test
public void returns_recommendations_for_single_location() {
```

```
    givenFeaturedDestinationsFor(paris,
        List.of(
            eiffelTower,
            louvre
        ));
    givenADistanceBetween(paris, eiffelTower, 5000);
    givenADistanceBetween(paris, louvre, 1000);

    assertEquals(
        List.of(
            new FeaturedDestinationSuggestion(paris, louvre, 1000),
            new FeaturedDestinationSuggestion(paris, eiffelTower, 5000)
        ),
        recommendations.recommendationsFor(Set.of(paris))
    );
}
```

Вот реализация `givenADistanceBetween` (пример 17.3).

Пример 17.3 [interfaces-to-funs.7:src/test/java/travelator/recommendations/RecommendationsTests.java]

```
private void givenADistanceBetween(
    Location location,
    FeaturedDestination destination,
    int result
) {
    when(
        distanceCalculator.distanceInMetersBetween(
            location,
            destination.getLocation())
    ).thenReturn(result);
}
```

Повреждение теста, вызванное моком

Дэвид Хайнемайер Ханссон ввел в обиход термин «повреждение конструкции, вызванное испытанием» (test-induced design damage — см. **https://oreil.ly/8vgJU**) для обозначения вреда, причиненного системам при тестировании. На практике ваши авторы не считают это большой проблемой — мы, как правило, видим, что системы обычно улучшаются за счет разделения, необходимого для их хорошего тестирования. Однако часто случается так, что тесты ломаются из-за моков — причем настолько часто, что моки впали в наших кругах в немилость.

Проблема в том, что, помимо реализации интерфейсов, фиктивные фреймворки позволяют нам указывать ожидаемые вызовы их методов и то, что должно быть возвращено в этих случаях. Часто, однако, ожидаемые вызовы и их результаты не соответствуют описанию, удобному для чтения человеком, — мы это увидим, если посмотрим на замененную встроенным выражением версию последних ожиданий (пример 17.4).

Пример 17.4 [interfaces-to-funs.1:src/test/java/travelator/recommendations/RecommendationsTests.java]

```
when(featuredDestinations.findCloseTo(paris))
    .thenReturn(List.of(
        eiffelTower,
        louvre
    ));
when(distanceCalculator.distanceInMetersBetween(
    paris, eiffelTower.getLocation())
).thenReturn(5000);
when(distanceCalculator.distanceInMetersBetween(
    paris, louvre.getLocation())
).thenReturn(1000);

when(featuredDestinations.findCloseTo(alton))
    .thenReturn(List.of(
        flowerFarm,
        watercressLine
    ));
when(distanceCalculator.distanceInMetersBetween(
    alton, flowerFarm.getLocation())
).thenReturn(5300);
when(distanceCalculator.distanceInMetersBetween(
    alton, watercressLine.getLocation())
).thenReturn(320);
```

Определение методов, таких как `givenADistanceBetween`, позволяет нам выразить взаимосвязь между фиктивными ожиданиями и нашим тестом: они могут скрыть *как*, чтобы разоблачить *почему*. На практике, однако, очень немногие разработчики идут на этот шаг, что приводит к появлению загадочных тестов, которые обвиняются в использовании мокинга.

Нэт подчеркивает, что моки, о которых он и Стив Фриман написали в книге «Growing Object-Oriented Sotware Guided by Tests», никогда не предполагалось использовать для реализации функций запросов, таких как `findCloseTo` и `DistanceInMetersBetween`, а только для методов, которые изменяют состояние. Дункан не помнит, чтобы замечал это, и лично не против использования моков таким образом, потому что они по-прежнему являются хорошим способом определить, чего мы ожидаем от соавторов, когда практикуем внешнюю разработку, основанную на тестировании, будь то считывание или запись. В конце концов, возможно, это не имеет значения, потому что, по опыту, в большинстве кодовых баз Java есть моки, которые используются таким образом, а большинству кодовых баз Kotlin было бы лучше без них.

Однако мы пока еще применяем мокинг, но предыдущий рефакторинг привел к тому, что мы передали ограниченные интерфейсы (типы функций) тестируемому коду. Теперь, когда нам не надо реализовывать невызываемые методы, нужны ли нам все еще моки? Давайте посмотрим, куда приведет нас этот вопрос.

Замена мокинга маппингом

Прежде чем продолжить, мы преобразуем тесты в Kotlin, потому что он имеет лучшую поддержку типов функций. Мы могли бы остаться в Java, но тогда нам пришлось бы решить, какой из типов функций Java (`Function`, `BiFunction` и т. д.) выражает операции. И у нас все еще была бы Java.

Автоматическое преобразование происходит достаточно плавно, хотя по какой-то причине конвертер создал лямбды вместо того, чтобы использовать ссылки на методы в вызове конструктора `Recommendations`, которые мы должны заменить вручную, оставив установку (пример 17.5).

Пример 17.5 [mocks-to-maps.0:src/test/java/travelator/recommendations/RecommendationsTests.kt]

```
class RecommendationsTests {
    private val distanceCalculator = mock(DistanceCalculator::class.java)
    private val featuredDestinations = mock(FeaturedDestinations::class.java)

    private val recommendations = Recommendations(
        featuredDestinations::findCloseTo,
        distanceCalculator::distanceInMetersBetween
    )
    ...
```

Мы могли бы использовать овеществленные типы Kotlin, чтобы избежать этих аргументов `::class.java`, но мы уходим от моков, а не идем к ним, поэтому сопротивляемся такому решению.

Термин `when` является ключевым словом в Kotlin, но конвертер достаточно умен, чтобы указывать его там, где это необходимо (пример 17.6).

Пример 17.6 [mocks-to-maps.0:src/test/java/travelator/recommendations/RecommendationsTests.kt]

```
private fun givenFeaturedDestinationsFor(
    location: Location,
    result: List<FeaturedDestination>
) {
    Mockito.`when`(featuredDestinations.findCloseTo(location))
        .thenReturn(result)
}
```

Рассмотрение процесса удаления моков помогает воспринять тип функции как *сопоставление* (mapping) ее входных параметров (в виде кортежа) с ее результатом. Следовательно `destinationFinder` представляет собой маппинг отдельных значений `Location` и последовательности `List<FeaturedDestination>`, `distanceInMetersBetween` —

это маппинг `Pair<Location, Location>` и `Int`. Структура данных `Map` (карты) — это наш способ выразить набор *маппингов* (и название это не случайно). Таким образом, мы можем имитировать функцию, заполнив карту (`Map`) ключами параметров и значениями результатов, а также заменить вызов функции поиском по предоставленным параметрам. Возможно, вы использовали этот трюк для кеширования результатов затратных вычислений. Здесь мы не станем кешировать, а будем заполнять `Map` параметрами и результатом, которые ожидаем увидеть.

Принимая первым случай `destinationFinder`, мы создадим свойство для хранения `Map` — `featuredDestinations` (пример 17.7).

Пример 17.7 [mocks-to-maps.1:src/test/java/travelator/recommendations/RecommendationsTests.kt]

```
private val featuredDestinations =
    mutableMapOf<Location, List<FeaturedDestination>>()
        .withDefault { emptyList() }
```

Параметр `givenFeaturedDestinationsFor` может заполнить `destinationLookup Map` вместо того, чтобы устанавливать ожидания для мока (пример 17.8).

Пример 17.8 [mocks-to-maps.1:src/test/java/travelator/recommendations/RecommendationsTests.kt]

```
private fun givenFeaturedDestinationsFor(
    location: Location,
    destinations: List<FeaturedDestination>
) {
    featuredDestinations[location] = destinations.toList()
}
```

Если заставить `Recommendations` считывать данные из `featuredDestinations Map`, мы успешно пройдем этап тестирования (пример 17.9).

Пример 17.9 [mocks-to-maps.1:src/test/java/travelator/recommendations/RecommendationsTests.kt]

```
private val recommendations =
    Recommendations(
        featuredDestinations::getValue,
        distanceCalculator::distanceInMetersBetween
    )
```

Параметр `getValue` представляет собой расширение для `Map`. Он действует как `get`, но соблюдает заданные `Map.withDefault` параметры по умолчанию (в нашем случае для возврата `emptyList()`) и, следовательно, не возвращает обнуляемый результат.

Вас не удивит, когда мы проделаем то же самое для `distanceInMetersBetween`, удаляя все зависимости от `Mockito` (пример 17.10).

Пример 17.10 [mocks-to-maps.2:src/test/java/travelator/recommendations/RecommendationsTests.kt]

```
class RecommendationsTests {

    private val featuredDestinations =
        mutableMapOf<Location, List<FeaturedDestination>>()
            .withDefault { emptyList() }

    private val distanceInMetersBetween =
        mutableMapOf<Pair<Location, Location>, Int>()
            .withDefault { -1 }

    private val recommendations =
        Recommendations(
            featuredDestinations::getValue,
            { l1, l2 -> distanceInMetersBetween.getValue(l1 to l2) }
        )
    ...
}
```

Может потребоваться пара проходов, чтобы увидеть, как это работает (пример 17.11). Здесь представлены детали, которые скрывают для нас имитирующие фреймворки. Вы можете спокойно игнорировать их и вернуться сюда, если когда-нибудь выполните этот рефакторинг самостоятельно.

Пример 17.11 [mocks-to-maps.2:src/test/java/travelator/recommendations/RecommendationsTests.kt]

```
private fun givenADistanceFrom(
    location: Location,
    destination: FeaturedDestination,
    distanceInMeters: Int
) {
    distanceInMetersBetween[location to destination.location] =
        distanceInMeters
}
```

Необходимость использовать лямбду, а не ссылку на метод в вызове конструктора `Recommendations`, немного раздражает. Мы можем это подправить, применив локальную функцию-расширение `getValue` (пример 17.12). Помните, как сильно нам нравятся функции-расширения?

Пример 17.12 [mocks-to-maps.3:src/test/java/travelator/recommendations/RecommendationsTests.kt]

```
private fun <K1, K2, V> Map<Pair<K1, K2>, V>.getValue(k1: K1, k2: K2) =
    getValue(k1 to k2)
```

Теперь мы можем сделать следующее (пример 17.13).

Пример 17.13 [mocks-to-maps.3:src/test/java/travelator/recommendations/RecommendationsTests.kt]

```
private val recommendations =
    Recommendations(
        featuredDestinations::getValue,
        distanceInMetersBetween::getValue
)
```

И да, мы можем улучшить читаемость методов тестирования с помощью некоторых разумных именований параметров и вспомогательных методов. Раньше у нас были простые вызовы функций (пример 17.14).

Пример 17.14 [mocks-to-maps.3:src/test/java/travelator/recommendations/RecommendationsTests.kt]

```
@Test
fun deduplicates_using_smallest_distance() {
    givenFeaturedDestinationsFor(
        alton,
        flowerFarm, watercressLine
    )
    givenFeaturedDestinationsFor(
        froyle,
        flowerFarm, watercressLine
    )
    givenADistanceFrom(alton, flowerFarm, 5300)
    givenADistanceFrom(alton, watercressLine, 320)
    givenADistanceFrom(froyle, flowerFarm, 0)
    givenADistanceFrom(froyle, watercressLine, 6300)
    assertEquals(
        listOf(
            FeaturedDestinationSuggestion(froyle, flowerFarm, 0),
            FeaturedDestinationSuggestion(alton, watercressLine, 320)
        ),
        recommendations.recommendationsFor(setOf(alton, froyle))
    )
}
```

Небольшое усилие дает нам следующее (пример 17.15).

Пример 17.15 [mocks-to-maps.4:src/test/java/travelator/recommendations/RecommendationsTests.kt]

```
@Test
fun deduplicates_using_smallest_distance() {
    givenFeaturedDestinationsFor(alton, of(flowerFarm, watercressLine))
    givenADistanceFrom(alton, to = flowerFarm, of = 5300)
    givenADistanceFrom(alton, to = watercressLine, of = 320)

    givenFeaturedDestinationsFor(froyle, of(flowerFarm, watercressLine))
    givenADistanceFrom(froyle, to = flowerFarm, of = 0)
    givenADistanceFrom(froyle, to = watercressLine, of = 6300)

    assertEquals(
        listOf(
            FeaturedDestinationSuggestion(froyle, flowerFarm, 0),
            FeaturedDestinationSuggestion(alton, watercressLine, 320)
        ),
        recommendations.recommendationsFor(setOf(alton, froyle))
    )
}
```

Иногда определение крошечной локальной функции, такой как `of`, может иметь большое значение для того, чтобы позволить нашему мозгу просто читать код, а не тратить усилия на его интерпретацию (пример 17.16).

Пример 17.16 [mocks-to-maps.4:src/test/java/travelator/recommendations/RecommendationsTests.kt]

```
private fun of(vararg destination: FeaturedDestination)
    = destination.toList()
```

Имитации в Kotlin

Будут времена, даже в Kotlin, когда мы захотим реализовать только некоторые методы интерфейса для тестирования. В JVM мы можем комбинировать динамические прокси с анонимными объектами, делегированием и выборочным переопределением, чтобы написать следующее:

```
inline fun <reified T> fake(): T =
    Proxy.newProxyInstance(
        T::class.java.classLoader,
        arrayOf(T::class.java)
    ) { _, _, _ ->
        TODO("not implemented")
    } as T
```

```
val sentEmails = mutableListOf<Email>()
val testCollaborator: EmailSystem =
    object : EmailSystem by fake() {
        override fun send(email: Email) {
            sentEmails.add(email)
        }
    }
```

Но действительно ли мы хотим уйти от моков?

Ах, вот это хороший вопрос!

В некотором смысле мы только что реализовали плохую имитацию мок-фреймворка: у нас нет сопоставителей параметров, нет способа разрешения сбоя, если метод не вызывается, и нет способа выразить порядок выполнения.

С другой стороны, мы реализовали зависимости механизма рекомендаций в виде двух карт. При этом `Recommendations.recommendationsFor` начинает выглядеть как простое вычисление (см. *разд. «Вычисления» главы 7*). Результат этого вычисления зависит от параметра `journey` и от содержимого тех карт, которые позволяют нам искать рекомендуемые пункты назначения и расстояния. Мы знаем, что в реальности имеет значение, *когда* мы вызываем `recommendationsFor`. Это на самом деле действие (см. *разд. «Действия» главы 7*). Расстояние между местоположениями, вероятно, не изменится со временем, но пункты назначения, которые мы найдем вокруг местоположения, будут меняться по мере добавления или удаления их из любой базы данных, в которой они хранятся. Однако в наших тестах это различие является спорным, и мы могли бы рассматривать `recommendationsFor` в качестве вычисления во многом таким же образом, как показано в случае с `InMemoryTrips` в *главе 7*. Вычисления легче протестировать, чем действия: мы просто проверяем, что заданный ввод возвращает заданный вывод, — так что давайте продолжим следовать за этой мыслью.

В настоящий момент важно, *когда* мы вызываем `recommendationsFor`, потому что результат будет зависеть от содержания карт `featuredDestinations` и `distanceInMetersBetween`. Они изначально пусты и заполняются вызовами `givenFeaturedDestinationsFor` и `givenADistanceFrom`.

Это напрямую чувствительно ко времени. Что нам нужно, так это какой-то способ преобразовать действие в вычисление, и мы можем сделать это, манипулируя областью действия.

В *главе 16* было показано, что можно рассматривать методы как функции с применением некоторыми из их аргументов, записывая их в виде полей. В тестах есть возможность обратить этот процесс вспять. Мы можем написать функцию, которая создает объект из своих зависимостей один раз для каждого вызова. Если мы называем заполненный объект *предметом* тестов, то можем воссоздать его из состояния тестирования следующим образом (пример 17.17).

Пример 17.17 [mocks-to-maps.5:src/test/java/travelator/recommendations/RecommendationsTests.kt]

```
private fun subjectFor(
    featuredDestinations: Map<Location, List<FeaturedDestination>>,
    distances: Map<Pair<Location, Location>, Int>
): Recommendations {
    val destinationsLookup = featuredDestinations.withDefault { emptyList() }
    val distanceLookup = distances.withDefault { -1 }
    return Recommendations(destinationsLookup::getValue, distanceLookup::getValue)
}
```

Здесь мы создаем новый экземпляр `Recommendations` с каждым вызовом, чтобы он мог захватывать неизменяемые карты, представляющие состояние системы.

Теперь можно написать функцию `resultFor`, которая использует `subjectFor` (пример 17.18).

Пример 17.18 [mocks-to-maps.5:src/test/java/travelator/recommendations/RecommendationsTests.kt]

```
private fun resultFor(
    featuredDestinations: Map<Location, List<FeaturedDestination>>,
    distances: Map<Pair<Location, Location>, Int>,
    locations: Set<Location>
): List<FeaturedDestinationSuggestion> {
    val subject = subjectFor(featuredDestinations, distances)
    return subject.recommendationsFor(locations)
}
```

За пределами области видимости функции `resultFor` нет чувствительности ко времени, так что это фактически вычисление.

Теперь, когда существует прямой маппинг ввода с выводом (`resultFor`), мы можем написать простые тесты для его вызова. Каждый тест может просто указать входные параметры и проверить, что результат соответствует ожиданиям, без необходимости указывать состояние в тесте вообще. Тогда каждый тест может принять следующую форму (пример 17.19).

Пример 17.19 [mocks-to-maps.5:src/test/java/travelator/recommendations/RecommendationsTests.kt]

```
private fun check(
    featuredDestinations: Map<Location, List<FeaturedDestination>>,
    distances: Map<Pair<Location, Location>, Int>,
    recommendations: Set<Location>,
    shouldReturn: List<FeaturedDestinationSuggestion>
```

```
) {
    assertEquals(
        shouldReturn,
        resultFor(featuredDestinations, distances, recommendations)
    )
}
```

Это придает приятную простоту ранее запутанным тестам (пример 17.20).

Пример 17.20 [mocks-to-maps.5:src/test/java/travelator/recommendations/RecommendationsTests.kt]

```
class RecommendationsTests {
    companion object {
        val distances = mapOf(
            (paris to eiffelTower.location) to 5000,
            (paris to louvre.location) to 1000,
            (alton to flowerFarm.location) to 5300,
            (alton to watercressLine.location) to 320,
            (froyle to flowerFarm.location) to 0,
            (froyle to watercressLine.location) to 6300
        )
    }

    ...

    @Test
    fun returns_no_recommendations_when_no_featured() {
        check(
            featuredDestinations = emptyMap(),
            distances = distances,
            recommendations = setOf(paris),
            shouldReturn = emptyList()
        )
    }

    ...

    @Test
    fun returns_recommendations_for_multi_location() {
        check(
            featuredDestinations = mapOf(
                paris to listOf(eiffelTower, louvre),
                alton to listOf(flowerFarm, watercressLine),
            ),
            distances = distances,
            recommendations = setOf(paris, alton),
```

```
            shouldReturn = listOf(
                FeaturedDestinationSuggestion(alton, watercressLine, 320),
                FeaturedDestinationSuggestion(paris, louvre, 1000),
                FeaturedDestinationSuggestion(paris, eiffelTower, 5000),
                FeaturedDestinationSuggestion(alton, flowerFarm, 5300)
            )
        )
    }
    ...
}
```

Поучительно сравнить это с исходным тестом (пример 17.21).

Пример 17.21 [interfaces-to-funs.0:src/test/java/travelator/recommendations/RecommendationsTests.java]

```
@Test
public void returns_recommendations_fore_multi_location() {
    givenFeaturedDestinationsFor(paris,
        List.of(
            eiffelTower,
            louvre
        ));
    givenADistanceBetween(paris, eiffelTower, 5000);
    givenADistanceBetween(paris, louvre, 1000);

    givenFeaturedDestinationsFor(alton,
        List.of(
            flowerFarm,
            watercressLine
        ));
    givenADistanceBetween(alton, flowerFarm, 5300);
    givenADistanceBetween(alton, watercressLine, 320);

    assertEquals(
        List.of(
            new FeaturedDestinationSuggestion(alton, watercressLine, 320),
            new FeaturedDestinationSuggestion(paris, louvre, 1000),
            new FeaturedDestinationSuggestion(paris, eiffelTower, 5000),
            new FeaturedDestinationSuggestion(alton, flowerFarm, 5300)
        ),
        recommendations.recommendationsFor(Set.of(paris, alton))
    );
}
```

Конечно, это Java, и она немного разбита вызовами `givenADistanceBetween`, но вы можете видеть, как этот рефакторинг перенес наши тесты с запутанных функций,

которые могут иметь или не иметь общую структуру, на четкое тестирование входных данных по сравнению с выходными данными.

Двигаемся дальше

Моки занимают свое место в программном обеспечении, и *разработка через тестирование* (Test-Driven Development, TDD), безусловно, может улучшить наши проекты, позволив создавать прототипы того, как распределять функциональность между сотрудничающими объектами без необходимости выполнять полные реализации. Однако у них есть привычка маскировать проблемы проектирования, заставляя нас тестировать проекты, выраженные в виде взаимодействий объектов, которые лучше рассматривать как потоки данных.

В приведенном здесь примере мы увидели, как сосредоточение внимания на данных может упростить наши тесты, особенно там, где мы только считываем значения. В *главе 20 «От выполнения ввода/вывода к передаче данных»* мы разберемся, как можно применить эту технику и к их написанию.

ГЛАВА 18

От открытых классов к запечатанным

Наши системы состоят из типов и операций — как бы из существительных и глаголов. В Java существительные выражаются в виде классов и интерфейсов, глаголы — в виде методов, а Kotlin добавляет к этому еще и запечатанные иерархии классов и автономные функции. Что они принесут на нашу вечеринку?

Изменения — это постоянная проблема при разработке программного обеспечения. Чем больше людей используют наше ПО, тем больше они думают о том, какие функции оно еще должно выполнять. Для поддержки новых вариантов использования нам необходимо добавлять новые функции, которые работают с существующими типами данных, и новые типы данных, которые работают с существующими функциями. Если наше улучшение хорошо укладывается в направление развития системы, мы можем спокойно добавлять новые функции, дописывая новый код и внося небольшие локализованные изменения в существующий. Если же оно идет вразрез с ним, то нам при добавлении нового типа данных придется изменять многие функции или изменять многие типы данных, когда нам нужно добавить новую функцию.

Это противоречие, связанное с изменяемостью типов данных и функций, мы наиболее остро ощущаем в основных объектах нашей модели предметной области. Так, основным элементом нашего приложения `Travelator` является маршрут путешественника. Многие функции приложения отображают и изменяют содержимое или рассчитывают информацию о маршрутах. Поэтому неудивительно, что многие запросы на функции от пользователей влияют на тип `Itinerary` своего маршрута. Наши путешественники хотят включать в свои маршруты больше моментов — не только поездки и размещение, как показано в *главе 10*, но и бронирование ресторанов и осмотр достопримечательностей по ходу маршрута. Кроме того, им желательно иметь возможность вносить в свои маршруты дополнительные изменения. В *главе 14* мы разобрались с оценкой стоимости маршрутов, но наши клиенты также хотят сравнивать их варианты по стоимости, времени или предоставляемому комфорту, просмотреть маршрут на карте, импортировать его в свой календарь, поделиться им со своими друзьями... их воображение безгранично.

Когда мы в последний раз обращались к классу маршрута `Itinerary` в *главе 14*, то моделировали маршрут в качестве класса данных с одним свойством для маршрута и с другим — для размещения, необходимого на маршруте (пример 18.1).

Пример 18.1 [accumulator.17:src/main/java/travelator/itinerary/Itinerary.kt]

```
data class Itinerary(
    val id: Id<Itinerary>,
    val route: Route,
    val accommodations: List<Accommodation> = emptyList()
) {
    ...
}
```

С тех пор мы добавили в приложение больше функций и, следовательно, больше типов элементов в маршрут. И обнаружили, что хранение каждого типа элементов маршрута в отдельной коллекции становится все более громоздким, поскольку слишком большая часть нашего кода связана с объединением этих коллекций или применением одних и тех же фильтров и преобразований к отдельным коллекциям. Поэтому мы решили, что `Itinerary` будет поддерживать единую коллекцию элементов маршрута `ItineraryItem` вместо того, чтобы хранить каждый тип элемента в отдельной коллекции (пример 18.2).

Пример 18.2 [open-to-sealed.0:src/main/java/travelator/itinerary/Itinerary.kt]

```
data class Itinerary(
    val id: Id<Itinerary>,
    val items: List<ItineraryItem>
) : Iterable<ItineraryItem> by items
```

То есть `ItineraryItem` — это интерфейс, реализованный конкретными типами элементов, — как теми, которые мы видели ранее: `Journey` и `Accommodation`, так и новыми типами: `RestaurantBooking` и `Attraction` (пример 18.3).

Пример 18.3 [open-to-sealed.0:src/main/java/travelator/itinerary/ItineraryItem.kt]

```
interface ItineraryItem {
    val id: Id<ItineraryItem>
    val description: String
    val costs: List<Money>
    val mapOverlay: MapOverlay
    ... и другие методы
}
```

Операции над `Itinerary` не зависят от конкретных типов элементов. Например, чтобы отобразить маршрут на карте, мы создаем элемент `MapOverlay`, который будет отображаться поверх фрагментов карты во внешнем интерфейсе. Наложение (overlay) для `Itinerary` — это группа наложений для всех содержащихся в нем элементов. Класс `Itinerary` и его клиенты не знают или не должны знать, как каждый элемент представляет себя в виде наложения на карту (пример 18.4).

Пример 18.4 [open-to-sealed.0:src/main/java/travelator/itinerary/Itinerary.kt]

```
val Itinerary.mapOverlay
    get() = OverlayGroup(
        id = id,
        elements = items.map { it.mapOverlay })
```

Этот полиморфизм позволяет очень легко добавлять новые типы `ItineraryItem` к системе без необходимости изменять части приложения, которые используют тип `Itinerary`.

Однако с некоторых пор нам этого делать не доводилось. И недавно мы обнаружили, что большая часть новых функций, которые мы добавляем в `Travelator`, включает добавление новых операций в `Itinerary` и `ItineraryItem`, а не новых типов в `ItineraryItem`. Изменения в интерфейсе `ItineraryItem` и его реализациях являются распространенным источником конфликтов слияния между членами команды, которые работают над разными функциями. С каждой новой функцией `ItineraryItem` становится объемнее.

Похоже, что это вызывает режим поддержки отдаленно связанных частей приложения со свойствами для рендеринга, оценки стоимости, ранжирования по комфорту, рисования карт и многого другого, скрытого за этим «`...и другие методы`». Парадоксально, но в ядре нашего приложения объектно-ориентированный полиморфизм *усиливает* связанность!

Объектно-ориентированный полиморфизм обеспечивает изменчивость типов данных при нечасто меняющемся наборе операций. Какое-то время это было тем, в чем нуждалась наша кодовая база, но теперь, когда она стабилизировалась, нам нужно обратное: изменчивость операций, применяемых к редко меняющемуся набору типов данных.

Если бы мы писали на Java (по крайней мере до Java 16), то у нас не было бы языковой функции, которая помогла бы справиться с изменчивостью в этом измерении. Основной особенностью Java для поддержки изменчивости является объектно-ориентированный полиморфизм, и он не помогает, когда операции меняются чаще, чем набор типов данных.

Мы могли бы задействовать паттерн *двойной отправки* (double dispatch — см. **https://oreil.ly/8m2HL**), но он включает в себя много шаблонного кода и, поскольку плохо работает с проверяемыми исключениями, широкого применения в Java не нашел. Вместо этого Java-программисты часто прибегают к проверке типов во время выполнения, используя операторы `instanceof` и `downcast` для исполнения разного кода для разных классов объектов (пример 18.5).

Пример 18.5 [open-to-sealed.0:src/main/java/travelator/itinerary/ItineraryItems.java]

```
if (item instanceof Journey) {
    var journey = (Journey) item;
    return ...
```

```
} else if (item instanceof Accommodation) {
    var accommodation = (Accommodation) item;
    return ...
} else if (item instanceof RestaurantBooking) {
    var restaurant = (RestaurantBooking) item;
    return ...
} else {
    throw new IllegalStateException("should never happen");
}
```

Функция `IllegalStateException` показывает рискованность такого подхода. Несмотря на то, что компилятор может проверять типы наших вызовов полиморфных методов, написанные нами вручную проверки типов и приведения типа во время выполнения явно обходят проверки во время компиляции. Проверка типов не может определить, являются ли наши приведения типов правильными, а условный оператор — *исчерпывающим* (применяется ли он ко всем возможным подклассам). Если метод возвращает значение, мы должны написать предложение `else`, чтобы вернуть фиктивное значение или вызвать исключение, даже если у нас есть ветки для каждого подкласса `ItineraryItem`, а предложение `else` «не может быть выполнено» (*cannot possibly be executed*™).

Даже в случае охвата при написании кода всех подтипов `ItineraryItem`, если позже мы добавим новые типы, то нам придется найти весь такой код и обновить его. Оказывается, мы здесь этого не делали, поэтому, если мы добавим `Attraction` к `Itinerary`, наш код выдаст ошибку `IllegalArgumentException`. ООП решает эту проблему, но мы обошли такое решение, потому что устали от необходимости обновлять множество классов при добавлении операции.

Проверка типов и понижающее приведение также возможны и в Kotlin, хотя тоже сопряжены с теми же накладными затратами и рисками. Однако в Kotlin есть еще один механизм организации классов и поведения, который делает проверку типов во время выполнения безопасной и удобной — *запечатанные классы*. Запечатанный класс — это абстрактный класс с фиксированным набором прямых подклассов. Мы должны определить запечатанный класс и его подклассы в одном и том же модуле компиляции и пакете. А компилятор не позволит нам осуществлять расширение запечатанного класса в другом месте. Благодаря этому ограничению проверки типов во время выполнения в запечатанных иерархиях классов не имеют той же проблемы, что и проверки типов во время выполнения в Java. Статическая проверка типов может гарантировать, что выражения `when`, которые выполняют проверку типов во время выполнения для подтипа запечатанного класса, охватывают все возможные случаи и только возможные случаи.

Когда инструкции не проверяются на исчерпываемость

Компилятор проверяет выражения `when` на исчерпываемость, но не проверяет *инструкции* `when`. `when` становится инструкцией, если внутреннее значение выражения `when` не используется. Вы можете заставить компилятор проверить исчерпываемость, используя результат `when`, даже в том случае, если оно относится к типу `Unit`.

Если when — только инструкция в теле функции, вы можете осуществить рефакторинг функции к форме однострочного выражения. Если when — это последняя инструкция в мультиинструкционной функции, вы можете явно использовать ее значение при помощи ключевого слова return. Когда when находится в середине тела функции, извлечение ее в его отдельную функцию может иметь смысл.

Если ни одна из этих опций не применяется, вы можете использовать следующую служебную функцию для принудительной проверки на исчерпываемость:

```
val <T> T.exhaustive get() = this
```

При таком использовании это предотвратит компиляцию, когда when неисчерпываемая:

```
when (instanceOfSealedClass) {
    is SubclassA -> println("A")
    is SubclassB -> println("B")
}.exhaustive
```

По сравнению с полиморфными методами запечатанные классы и выражения when упрощают добавление новых операций, которые применяются к иерархии фиксированных типов. Впрочем, нам все равно придется изменять все эти операции, если мы добавим в иерархию новый тип. Но на этом этапе компилятор поможет нам, проверив, что все такие операции охватывают в иерархии все возможные типы.

Полиморфизм или запечатанные классы?

В некоторых языках есть механизмы, которые позволяют нам изменять типы и операции без изменения существующего кода. У Haskell есть классы типов, у Scala — неявные параметры, у Rust — трейты, у Swift — протоколы, а у Clojure и Common Lisp — полиморфные функции, которые отправляют классы с несколькими аргументами.

У Kotlin нет никакого такого эквивалента. Когда мы разрабатываем в Kotlin, нам приходится выбирать между объектно-ориентированным полиморфизмом или запечатанными классами, основанными на типах измерений или операциях, — мы ожидаем, что они будут меняться чаще всего по мере развития программы. Объектно-ориентированный полиморфизм предпочтительнее, когда набор типов данных меняется чаще, чем набор операций над этими типами данных, а запечатанные иерархии классов — когда набор операций меняется чаще, чем набор типов данных, к которым они применяются.

Только типизация по запечатанной иерархии классов

Используйте приведение типов только для приведения из корня запечатанной иерархии классов к одному из дочерних элементов в исчерпывающем выражении when. В противном случае отказываться от статического типа рискованно. Фактический класс, используемый для реализации значения, может иметь операции, которые нарушают ограничения, выраженные его статическим типом.

Например, как показано в *главе 6*, статический тип List предотвращает мутацию, но функции более высокого порядка Kotlin возвращают списки, которые могут быть изменены, если вы осуществите понижающее приведение от List к MutableList.

Функция, которая приводит значение аргумента списка от `List` к `MutableList` и изменяет его, скорее всего, вызовет ошибки в коде, поскольку она нарушает ожидания своих абонентов. Это может привести к ошибкам псевдонимов, которые очень трудно обнаружить, потому что возможность неверного действия на дистанции не указана явно в объявлениях типов сигнатуры функции. Если будущая версия стандартной библиотеки Kotlin вернет неизменяемые списки из своих функций более высокого порядка, то функция продолжит успешно компилироваться, но завершится сбоем во время выполнения.

Вы не обязаны выполнять приведение из супертипа в подтип просто потому, что вы можете это сделать. Эта возможность, скорее всего, будет простой деталью реализации. Запечатанная иерархия классов сигнализирует о том, что понижающее приведение типов возможно, поддерживается и его безопасность, которая обеспечивается с помощью проверок компилятора на исчерпываемость.

Преобразование интерфейса в запечатанный класс

Мы собираемся добавить еще одну функцию, которая включает маршруты и элементы маршрута, — отображение маршрута `Itinerary` в приложении календаря путешественника. Мы не хотим добавлять дополнительные методы к уже раздутому интерфейсу `ItineraryItem` и связывать основные классы домена нашего приложения с потребностями другого периферийного модуля. Пришло время сделать решительный шаг и преобразовать `ItineraryItem` из интерфейса полиморфных методов в запечатанную иерархию классов и автономные функции, а также переместить эти автономные функции в модули, которые их используют.

Когда мы писали эту книгу, актуальным был Kotlin версии 1.4, поэтому нам требовалось определить запечатанный класс и его прямые подклассы в одном файле. Таким образом, наш первый шаг — использовать рефакторинг IDE **Переместить класс** (Move Class), чтобы переместить реализации `ItineraryItem` в тот же файл, что и интерфейс. Как только мы это сделаем, то сможем превратить интерфейс и его реализации в запечатанную иерархию классов. IntelliJ не имеет автоматического рефакторинга для этой операции, поэтому мы должны сделать это вручную, отредактировав определения классов (пример 18.6). Перемещение всех классов в один и тот же файл как минимум упростило нашу задачу.

Пример 18.6 [open-to-sealed.2:src/main/java/travelator/itinerary/ItineraryItem.kt]

```
sealed class ItineraryItem { ❶
    abstract val id: Id<ItineraryItem> ❷
    abstract val description: String
    abstract val costs: List<Money>
    abstract val mapOverlay: MapOverlay
    ... и другие методы
}
```

```
data class Accommodation(
    override val id: Id<Accommodation>,
    val location: Location,
    val checkInFrom: ZonedDateTime,
    val checkOutBefore: ZonedDateTime,
    val pricePerNight: Money
) : ItineraryItem() { ❸
    val nights = Period.between(
        checkInFrom.toLocalDate(),
        checkOutBefore.toLocalDate()
    ).days
    val totalPrice: Money = pricePerNight * nights

    override val description
        get() = "$nights nights at ${location.userReadableName}"
    override val costs
        get() = listOf(totalPrice)
    override val mapOverlay
        get() = PointOverlay(
            id = id,
            position = location.position,
            text = location.userReadableName,
            icon = StandardIcons.HOTEL
        )

    ... и другие методы
}

... и другие подклассы
```

❶ Мы объявляем ItineraryItem в качестве sealed class вместо interface.

❷ Поскольку теперь это класс, мы должны явно пометить его методы как abstract. Если бы в интерфейсе были какие-либо методы с реализацией по умолчанию, нам пришлось бы объявить их как open, чтобы подклассы все равно могли их переопределять.

❸ Мы заменяем объявление интерфейса в классах конкретных элементов вызовом конструктора суперкласса.

Kotlin 1.5 (выпущенный после завершения работы над книгой) поддерживает запечатанные интерфейсы, которые упрощают этот рефакторинг, — более нет необходимости перемещать подклассы в один и тот же файл или вызывать конструктор.

Теперь `ItineraryItem` — запечатанный класс. Его операции по-прежнему являются полиморфными методами, но мы можем добавлять *новые* операции без изменения классов `ItineraryItem`, написав функции-расширения, которые используют выражение `when` для безопасной отправки по конкретному типу элемента.

Сначала мы напишем функции-расширения, которые нам нужны для перевода `Itinerary` в календарь (пример 18.7). Сделав это, мы продолжим рефакторинг, чтобы другие операции над `ItineraryItem` работали таким же образом.

Пример 18.7
[open-to-sealed.3:src/main/java/travelator/calendar/ItineraryToCalendar.kt]

```
fun ItineraryItem.toCalendarEvent(): CalendarEvent? = when (this) {
    is Accommodation -> CalendarEvent(
        start = checkInFrom,
        end = checkOutBefore,
        description = description,
        alarms = listOf(
            Alarm(checkInFrom, "Check in open"),
            Alarm(checkOutBefore.minusHours(1), "Check out")
        )
    )
    is Attraction -> null
    is Journey -> CalendarEvent(
        start = departureTime,
        end = arrivalTime,
        description = description,
        location = departsFrom,
        alarms = listOf(
            Alarm(departureTime.minusHours(1)))
    )
    is RestaurantBooking -> CalendarEvent(
        start = time,
        description= description,
        location = location,
        alarms = listOf(
            Alarm(time.minusHours(1)))
    )
}
```

Теперь давайте применим рефакторинг остальных методов `ItineraryItem` из полиморфных методов, определенных в (теперь запечатанном) классе, в функции-расширения, которые используют выражения `when` для переключения типа элемента. Мы пройдем через весь процесс с помощью свойства `mapOverlay`.

Нажатие комбинации клавиш <Alt>+<Enter> на определении `mapOverlay` в `ItineraryItem` открывает контекстное меню, содержащее действие **Преобразовать элемент в расширение** (Convert member to extension). Неужели это действительно так просто? К сожалению, нет. На момент подготовки книги действие IDE позволяет пройти только часть пути и оставляет нас с кодом, который не компилируется (пример 18.8).

Пример 18.8 [open-to-sealed.4:src/main/java/travelator/itinerary/ItineraryItem.kt]

```
sealed class ItineraryItem {
    abstract val id: Id<ItineraryItem>
    abstract val description: String
    abstract val costs: List<Money> ❶
    ... и другие методы
}

val ItineraryItem.mapOverlay: MapOverlay ❷
    get() = TODO("Not yet implemented")

data class Accommodation(
    override val id: Id<Accommodation>,
    val location: Location,
    val checkInFrom: ZonedDateTime,
    val checkOutBefore: ZonedDateTime,
    val pricePerNight: Money
) : ItineraryItem() {
    val nights = Period.between(
        checkInFrom.toLocalDate(),
        checkOutBefore.toLocalDate()
    ).days
    val totalPrice: Money = pricePerNight * nights

    override val description
        get() = "$nights nights at ${location.userReadableName}"
    override val costs
        get() = listOf(totalPrice)
    override val mapOverlay ❸
        get() = PointOverlay(
            id = id,
            position = location.position,
            text = location.userReadableName,
            icon = StandardIcons.HOTEL
        )

    ... и другие методы
}
```

❶ IDE удалила метод `mapOverlay` из класса `ItineraryItem`...

❷ ...и заменила его функцией-расширением. К сожалению, функция-расширение содержит только `TODO`, выдающее ошибку `UnsupportedOperationException`.

❸ IDE оставляет модификаторы `override` для свойств `mapOverlay` в подклассах, у которых больше нет методов для переопределения в суперклассах.

Мы можем снова скомпилировать код, удалив модификаторы `override` в подклассах. Этим мы заставим код фактически работать с реализацией тела функции-расширения в качестве выражения `when`, которое переключается на тип `ItineraryItem` и вызывает теперь уже мономорфный геттер `mapOverlay` для каждого конкретного класса (пример 18.9).

Пример 18.9 [open-to-sealed.5:src/main/java/travelator/itinerary/ItineraryItem.kt]

```
val ItineraryItem.mapOverlay: MapOverlay get() = when (this) {
    is Accommodation -> mapOverlay
    is Attraction -> mapOverlay
    is Journey -> mapOverlay
    is RestaurantBooking -> mapOverlay
}
```

Выражение `when` не скомпилируется до тех пор, пока не будут рассмотрены все классы `ItineraryItem`.

IntelliJ также подчеркивает каждое прочтение подкласса свойства `mapOverlay`, чтобы показать, что чувствительный к потоку вывод компилятора интеллектуально приводит неявную ссылку `this` из `ItineraryItem` к корректному подклассу.

Итак, смысл этого рефакторинга состоял в недопущении того, чтобы каждая реализация `ItineraryItem` знала о наложениях карты. В настоящее время все они все еще работают, потому что у каждой есть свое собственное свойство `mapOverlay` — то, которое изначально переопределяло свойство в интерфейсе (пример 18.10).

Пример 18.10 [open-to-sealed.5:src/main/java/travelator/itinerary/ItineraryItem.kt]

```
data class Accommodation(
...
) : ItineraryItem() {
    ...
    val mapOverlay
        get() = PointOverlay(
            id = id,
            position = location.position,
            text = location.userReadableName,
            icon = StandardIcons.HOTEL
    )
...
```

Мы можем решить эту проблему, преобразовав свойство `mapOverlay` путем выполнения действия **Преобразовать элемент в расширение** (пример 18.11).

Пример 18.11 [open-to-sealed.6:src/main/java/travelator/itinerary/ItineraryItem.kt]

```
data class Accommodation(
...
```

```
) : ItineraryItem() {
    ...
}

val Accommodation.mapOverlay
    get() = PointOverlay(
        id = id,
        position = location.position,
        text = location.userReadableName,
        icon = StandardIcons.HOTEL
    )
```

Теперь, похоже, `ItineraryItem.mapOverlay` вообще не изменилась (пример 18.12).

Пример 18.12 [open-to-sealed.6:src/main/java/travelator/itinerary/ItineraryItem.kt]

```
val ItineraryItem.mapOverlay: MapOverlay get() = when (this) {
    is Accommodation -> mapOverlay
    is Attraction -> mapOverlay
    is Journey -> mapOverlay
    is RestaurantBooking -> mapOverlay
}
```

Однако присмотритесь повнимательнее (ну, наведите курсор в IntelliJ), и вы увидите, что эти обращения к свойствам теперь являются свойствами-расширениями, а не вызовами методов. `Accommodation` и остальные больше не зависят от `MapOverlay`. И теперь, когда `ItineraryItem.mapOverlay` и все свойства подкласса являются расширениями, их не нужно определять в том же файле, что и запечатанные классы. Мы можем переместить их в модуль или пакет, где они используются, и они не будут загромождать нашу основную доменную абстракцию (пример 18.13).

Пример 18.13 [open-to-sealed.7:src/main/java/travelator/geo/ItineraryToMapOverlay.kt]

```
package travelator.geo

import travelator.itinerary.*

val ItineraryItem.mapOverlay: MapOverlay get() = when (this) {
    is Accommodation -> mapOverlay
    is Attraction -> mapOverlay
    is Journey -> mapOverlay
    is RestaurantBooking -> mapOverlay
}

private val Accommodation.mapOverlay
    get() = PointOverlay(
        id = id,
```

```
        position = location.position,
        text = location.userReadableName,
        icon = StandardIcons.HOTEL
    )

... Attraction.mapOverlay и пр.
```

Мы можем проделать то же самое с другими членами `ItineraryItem`, пока запечатанный класс не объявит только фундаментальные свойства типа. Для `ItineraryItem` в настоящий момент только свойство `id` поистине фундаментально — объявление `id` в качестве абстрактного свойства в запечатанном классе принуждает каждый подкласс иметь идентификатор.

Некоторые из других свойств явно предназначены только для поддержки определенных функций приложения — таких как `mapOverlay` и `toCalendar`. А некоторые — такие как `description`, находятся в серой зоне: они поддерживают многие функции приложения, но не являются фундаментальным свойством `ItineraryItem`. Так, каждый подтип получает свое описание из своих фундаментальных свойств. Нэт предпочитает определять подобные свойства как расширения, в то время как Дункан — в качестве членов класса. Поскольку этот пример пишет Нэт, `description` здесь указан в качестве расширения (пример 18.14), а вам придется в своем коде принять собственное решение.

Пример 18.14 [open-to-sealed.8:src/main/java/travelator/itinerary/ItineraryDescription.kt]

```
val ItineraryItem.description: String
    get() = when (this) {
        is Accommodation ->
            "$nights nights at ${location.userReadableName}"
        is Attraction ->
            location.userReadableName
        is Journey ->
            "${departsFrom.userReadableName} " +
                "to ${arrivesAt.userReadableName} " +
                "by ${travelMethod.userReadableName}"
        is RestaurantBooking -> location.userReadableName
    }
```

Итак, у нас остается запечатанный класс `ItineraryItem`, объявляющий только свойство `id`, и его подклассы, объявляющие свои фундаментальные свойства. Вся иерархия при этом выглядит следующим образом (пример 18.15).

Пример 18.15 [open-to-sealed.8:src/main/java/travelator/itinerary/ItineraryItem.kt]

```
sealed class ItineraryItem {
    abstract val id: Id<ItineraryItem>
}
```

```
data class Accommodation(
    override val id: Id<Accommodation>,
    val location: Location,
    val checkInFrom: ZonedDateTime,
    val checkOutBefore: ZonedDateTime,
    val pricePerNight: Money
) : ItineraryItem() {
    val nights = Period.between(
        checkInFrom.toLocalDate(),
        checkOutBefore.toLocalDate()
    ).days
    val totalPrice: Money = pricePerNight * nights
}

data class Attraction(
    override val id: Id<Attraction>,
    val location: Location,
    val notes: String
) : ItineraryItem()

data class Journey(
    override val id: Id<Journey>,
    val travelMethod: TravelMethod,
    val departsFrom: Location,
    val departureTime: ZonedDateTime,
    val arrivesAt: Location,
    val arrivalTime: ZonedDateTime,
    val price: Money,
    val path: List<Position>,
    ... и другие поля
) : ItineraryItem()

data class RestaurantBooking(
    override val id: Id<RestaurantBooking>,
    val location: Location,
    val time: ZonedDateTime
) : ItineraryItem()
```

Модель `ItineraryItem` теперь представляет собой запечатанную иерархию классов из чистых классов данных. Все операции, необходимые для функций нашего приложения, являются функциями-расширениями в модулях для этих функций. Только свойство `id` остается полиморфным `val`, поскольку оно является фундаментальным свойством типа, которое неспецифично для какой-либо одной функции приложения.

Двигаемся дальше

По мере развития нашего программного обеспечения приходится добавлять в систему новые типы данных и новые операции. В Kotlin, как и в Java, объектно-ориентированный полиморфизм позволяет легко добавлять новые типы данных без изменения кода существующих функций. Мы также можем использовать запечатанные классы и безопасные проверки типов, встроенные в среду выполнения, чтобы легко добавлять новые функции поверх существующих типов данных, не изменяя код, определяющий эти типы.

То, что мы выбираем, зависит от наших ожиданий: что будет меняться чаще всего по мере развития кода — типы данных или операции. Управление изменчивостью в Kotlin включает в себя понимание того, когда применять оба эти механизма к нашим моделям предметной области.

Если наше предположение окажется неверным, мы должны пойти по другому пути. Когда весь код находится в одной кодовой базе, Kotlin и IntelliJ упрощают рефакторинг между двумя указанными вариантами. В этой главе описан переход от объектно-ориентированного полиморфизма, который мы бы написали на Java, к запечатанным классам Kotlin. Другой путь включает в себя этапы рефакторинга, описанные в книге Мартина Фаулера «Refactoring: Improving the Design of Existing Code», — такие как «Замена условного выражения полиморфизмом», поэтому мы не будем рассматривать этот подход в нашей книге.

ГЛАВА 19

От проверяемых исключений к типам результатов

Java использует проверяемые и непроверяемые исключения для представления и обработки ошибок. Kotlin поддерживает исключения в целом, но проверяемые исключения в этот язык не встроены. Почему Kotlin отверг подход Java и что нам следует использовать вместо него?

Вам не нужно долго заниматься разработкой программного обеспечения для компьютеров, чтобы обнаружить, что что-то идет не так... во *многих* отношениях.

В начале своей карьеры ваши авторы были склонны замалчивать ошибки. Мы и сейчас часто поступаем так, по крайней мере на ранней стадии проекта. Однако по мере роста системы мы узнаем, как сбои влияют на приложение, и начинаем добавлять код, чтобы справиться с этим: сначала навскидку, а затем с некоторой стратегией, основанной на опыте. В этом плане наш подход к обработке ошибок основан на тех же тенденциях, что и другие аспекты создания нами программ. Иногда мы разрабатываем решения заранее, используя наш опыт работы с подобными системами, а в других случаях позволяем процессу написания программного кода учить нас тому, что ему требуется.

В отсутствие более или менее продуманной стратегии большинство систем по умолчанию создают исключения, когда что-то идет не так, — перехватывают и регистрируют эти исключения на каком-то внешнем уровне. Утилиты командной строки в таких случаях просто завершают работу — надеемся, предоставив пользователю достаточно информации, чтобы устранить проблему и повторить попытку. Серверное приложение или графический интерфейс с циклом событий обычно прерывают текущее взаимодействие и переходят к следующему.

Зачастую сбои ПО просто портят нашим пользователям настроение — не более того, но иногда ошибка приводит к повреждению постоянного состояния системы, поэтому исправление возникшей проблемы и повторная попытка идти дальше не срабатывают. Отсюда и проистекает мудрый совет: «выключить и снова включить». Наши системы в основном запускаются в безопасном состоянии, так что после перезагрузки повторная попытка должна быть успешной. Если нет, что ж, вы, вероятно, бывали в ситуации, когда единственным решением становилась переустановка операционной системы — лучший способ уйти от поврежденного постоянного состояния.

Перезагрузка интернет-соединения

У Дункана была проблема, когда подключение между его термостатом Nest и системой автоматизации процессов IFTTT (IfThisThenThat) не работало. IFTTT получала уведомления,

когда Nest переходил в домашний режим, но не в режим ожидания. Серьезное отключение облачного веб-сервиса AWS (Amazon Web Services) 28 февраля 2017 года таинственным образом устранило проблему. Оказывается, все, что для этого требовалось, — это перезагрузка интернет-соединений.

Если ошибки не устраняются должным образом, но, несмотря на это, система остается работоспособной, диагностика и устранение повреждений, вызванных ошибками, могут занять все время работы команды, что отнюдь не является самым подходящим делом для программного проекта. Спросите нас, откуда мы про это знаем!

Нам не нужны сбои, потому что они раздражают наших пользователей и могут привести к повреждению ПО, для исправления которого потребуется много усилий, если мы вообще сможем его исправить. Какие сбои случаются?

В программе что-то может пойти не так по многим причинам. Когда мы говорим *программа*, то также имеем в виду ее функции, методы, процедуры — любой вызываемый код. И когда мы говорим *пойти не так*, то имеем в виду, что они не справляются с той работой, которая на них возлагается. Причинами сбоев могут быть следующие моменты:

- иногда программам необходимо взаимодействовать с другими системами, и связь с ними каким-то образом прерывается;
- зачастую мы не даем программе правильных входных данных, необходимых ей для выполнения своих задач;
- случается, что и программисты допускают ошибки — дают, например, компьютеру инструкции разыменовывать нулевые ссылки или читать дальше концов коллекций!
- среда, в которой мы работаем, по какой-то причине дает сбой — например, ей может не хватить памяти, или она не может загрузить класс.

Есть сбои, которые не вписываются в эти категории, но большинство все же в них попадают.

Этот список не кажется очень уж длинным, и все же, как у отрасли, у нас не очень хорошая репутация в плане надежности, — обработка ошибок всем представляется сложной задачей. Почему так?

Ну, во-первых, мы часто не знаем, может ли операция завершиться неудачно, и если да, то как. Хорошо, пусть мы это знаем, но знание того, как обрабатывать ошибку, может находиться в коде далеко от места обнаружения проблемы. Тогда код, который обнаруживает ошибку, и код, который восстанавливает работу программы после ее устранения, трудно отделить от «счастливого пути», и поэтому их сложно протестировать. Объедините это с тенденцией ошибок оставлять нашу систему в неисправном состоянии, и мы получим ситуацию, когда большинство разработчиков скорее будут надеяться на лучшее, чем возьмутся за тяжелую работу и все равно ошибутся.

Тяжелая работа *и* склонность к ошибкам? Разве компьютеры не должны были освобождать нас от подобных ситуаций и брать на себя тяжелую работу, чтобы мы

могли сосредоточиться на веселых творческих процессах? Да, так оно и есть, поэтому мы сфокусируемся на обработке ошибок через призму того, как наш язык программирования может сделать работу программистов безопаснее и проще.

Обработка ошибок до появления методов, основанных на исключениях

Большая часть обработки ошибок в наши дни основана на исключениях, но существуют и другие методы, которые могут быть применимы в определенных обстоятельствах. Сначала мы рассмотрим плюсы и минусы этих методов. Минусы покажут нам, почему исключения сейчас доминируют. Плюсы могут подсказать нам варианты, когда можно обойтись без исключений.

- *Игнорирование ошибок.*

  Мы можем игнорировать ошибки: либо сбойная процедура ничего не делает, чтобы привлечь к ним внимание вызывающей стороны (абонента), либо вызывающая сторона не утруждает себя проверкой состояния.

  Это может привести к повреждению постоянных данных и скрытому сбою в работе, поэтому в большинстве случаев нам нужно стремиться все же к исправлению ситуации.

- *Простое падение программы.*

  Некоторые программы просто завершают работу при обнаружении ошибки.

  В сочетании с супервизором для перезапуска при ошибке и тщательным кодированием для предотвращения повреждения постоянного состояния — это проверенная в боевых условиях стратегия, которая может оказаться подходящей. Генерация исключения для прерывания операции как раз и есть применение этого метода к процедуре, а не ко всей программе.

- *Возврат специального значения.*

  Полезным методом может стать возврат специального значения для обозначения ошибки. Например, функция может возвращать `-1` вместо индекса, когда элемент не найден в списке.

  Однако этот метод нельзя использовать, когда все возвращаемые значения являются допустимыми результатами для функции. Это также может быть опасно, потому что вызывающая сторона должна знать (и помнить) соглашение. Если мы попытаемся вычислить расстояние между двумя элементами в списке, вычитая их индексы, когда один из них не найден и возвращает `-1`, наше вычисление будет неверным, если мы явно не обработаем особый случай. Кроме того, мы не можем полагаться на средство проверки типов в надежде избежать ошибок.

  Особым случаем возврата специального значения является возврат `null` при ошибке. Это весьма опасно для большинства языков, потому что если вызывающая сторона явно не проверяет наличие `null`, то использование такого результата вызовет исключение `NullPointerException`, что может быть еще хуже,

чем первоначальная проблема. Однако в Kotlin средство проверки типов заставляет вызывающие программы иметь дело с `null`, что делает этот метод безопасным и эффективным.

- *Установка глобального флага.*

  Одна из проблем с возвратом специальных значений заключается в том, что они затрудняют определение того, какая из нескольких возможных ошибок произошла. Чтобы решить эту проблему, мы можем объединить специальное значение с установкой глобальной переменной. При обнаружении специального значения вызывающая сторона может прочитать `errno` — например, чтобы установить, в чем заключалась проблема.

  Этот метод был популярен в C, но сейчас в значительной степени вытеснен обработкой ошибок на основе исключений.

- *Возврат кода состояния.*

  Еще один метод, применявшийся до появления исключений, — это возврат кода состояния. Такое возможно, когда функция либо не возвращает никакого значения (это полностью побочный эффект), либо возвращает значение другим способом — часто путем изменения параметра, передаваемого по ссылке.

- *Вызов специальной функции.*

  Приемлемым подходом при возникновении ошибки иногда может стать вызов специальной функции. Обычно функция ошибки передается в качестве параметра вызываемой функции. При обнаружении проблемы вызывается функция ошибки со значением, представляющим ошибку в качестве параметра. Иногда функция ошибки может сигнализировать своим возвращаемым значением, что неудачную операцию следует повторить или прервать. Можно также сделать так, чтобы функция ошибок предоставляла значение, которое должно быть возвращено вызванной функцией.

  Этот подход также представляет собой пример стратегии, применяемой для обработки ошибок. Даже когда доступны исключения, он весьма полезен в определенных ситуациях.

Обработка ошибок с исключениями

Все приведенные в предыдущем разделе методы страдают тем недостатком, что вызывающий код способен в большей или меньшей степени игнорировать факт возникновения ошибки.

Исключения решают эту проблему. Операция автоматически прерывается при ошибке, и вызывающий объект явно обрабатывает исключение. Если вызывающий объект не может выполнить обработку исключения, оно распространяется далее по стеку вызовов до тех пор, пока что-то не произойдет, и если никакая функция это исключение не обработает, поток завершится.

Java и проверяемые исключения

Исключения, на момент выпуска Java, были относительной новинкой, и разработчики языка решили внедрить в эту область собственные инновации. При создании средств обработки ошибок они предусмотрели условие, что метод должен быть способен выдавать часть своей сигнатуры. В результате вызывающая сторона будет знать, что, например, метод может завершиться ошибкой из-за того, что считываемый сетевой ресурс перестал был доступным. Если метод объявляет, что он может завершиться из-за чего-то неудачей, то каждый вызывающий этот метод должен либо иметь дело с ошибкой (указав в блоке `catch`, как ее следует обрабатывать), либо объявить, что он тоже может завершиться неудачей с тем же исключением. Этот подход гарантирует, что программист учитывает возможность возникновения подобных ошибок. Такие исключения называются *проверяемыми исключениями*, потому что компилятор проверяет факт их обработки (или что они повторно объявлены для вывода вызывающим методом).

Проверяемые исключения были разработаны для тех случаев, когда программист может разумно найти способ восстановления — например, организовать повторную попытку записи в базу данных или повторное открытие сокета. Разработчики языка определили также два других типа неудач: ошибки и исключения во время выполнения.

- *Ошибки (Error).*

  Подклассы `java.lang.Error` зарезервированы для сбоев настолько серьезных, что JVM больше не может гарантировать правильное функционирование среды выполнения. Возможно, в системе заканчивается память, или класс не может быть загружен. Такие условия могут возникнуть в любой момент выполнения программы и поэтому приведут к сбою любой функции. Поскольку любой метод может дать сбой в результате подобных ситуаций, нет смысла включать их в сигнатуру каждого метода, поэтому и объявлять их не требуется.

  Эта схема вынуждает разработчиков иметь дело с операциями, которые могут завершиться сбоем из-за ошибок ввода/вывода или других факторов, которые находятся вне их контроля (проверяемые исключения), позволяя реализовать защитное программирование там, где это выгодно. С другой стороны, если выдается `Error`, лучший подход по умолчанию — выйти из процесса как можно быстрее, прежде чем будет нанесен еще какой-либо ущерб постоянному состоянию.

- *Исключения среды выполнения (Runtime Exceptions).*

  Подклассы `RuntimeException` представляют другие ошибки. Замысел состоял в том, чтобы отделить их от проблем, вызванных ошибками программиста — такими как доступ к null-ссылке или попытка чтения за пределами коллекции. В обоих этих случаях программист мог бы быть более внимательным. Однако ошибкам программиста подвержен любой фрагмент кода, поэтому в таких ситуациях нет необходимости объявлять `RuntimeExceptions`.

  `RuntimeException` — это золотая середина. Если сбой вызван ошибкой программиста, нам следует понять, что мы не знаем, что происходит в нашей программе,

и прервать текущую операцию или все приложение. В противном случае мы могли бы предпринять попытку восстановления, особенно если система разработана так, чтобы ограничить ущерб, который может быть нанесен постоянному состоянию.

Вашим авторам очень понравились проверяемые исключения, но мы оказались в меньшинстве, потому что такие исключения с годами впали в Java в немилость. Затруднения для проверяемых исключений с самого начала были обусловлены странным решением создать непроверяемый подкласс `RuntimeException` из проверяемого в противном случае `Exception`. Так что код, который должен обрабатывать все проверяемые исключения, на практике также улавливает непроверяемые исключения, скрывая ошибки программирования.

Также не помог тот факт, что API-интерфейсы Java использовали их непоследовательно. Возьмем, к примеру, извлечение данных из строки `URL` — конструктор `URL(String)` выдает *проверенную* ошибку `MalformedURLException`, в то время как `Integer.parseInt(String)` выдает *непроверенную* ошибку `NumberFormatException`.

Как должен быть оформлен сбой `parseInt`?

Это интересный случай, который показывает, почему обработка ошибок так сложна.

Просматривая наши варианты стратегии, мы приходим к выводу, что `parseInt` не может возвращать специальное целочисленное значение, потому что все целые числа являются допустимыми результатами. Он мог бы возвращать значение `null`, упакованное в `Integer`, но для этого была бы необходима критичная в отношении производительности нежелательная операция упаковки и распаковки — действительно фундаментальная низкоуровневая операция, особенно на JVM середины 1990-х годов.

Вызов функции ошибки аналогично повлек бы за собой неэффективный процесс, поэтому нам остается создать исключение. Должно ли это исключение быть проверяемым или нет?

Разработчики языка решили, что `parseInt` должна выдавать `NumberFormatException` и что `NumberFormatException` должна быть `IllegalArgumentException`, которая является `RuntimeException`, а значит, непроверяемой.

Оба эти решения являются разумными в отдельности. В сочетании, однако, они приводят к тому, что `parseInt` не заставляет вызывающих ее пользователей (абонентов) учитывать, что она может завершиться ошибкой, как это было бы, если бы она объявила проверяемое исключение.

Мы подозреваем, что программисты JVM очень привыкли разбирать целые числа из строк на C (где нет исключений), используя функцию `atoi`, которая услужливо возвращает 0, если не может быть успешно завершена. Они бы сочли, что непредусмотренность этого сбоя является ошибкой программиста, а не сбоем самой функции. Ваши авторы, однако, были бы признательны, если бы им напомнили о возможности сбоя и указали бы проверяемое исключение.

Путаница в отношении того, какой тип исключения использовать, умножалась, и вскоре по умолчанию было установлено, что единственными проверяемыми исключениями, объявленными большинством библиотек Java, будут `IOExceptions`. Но даже тогда библиотеки баз данных, такие как Hibernate, которые определенно общались по сети и явно подвергались `IOExceptions`, генерировали только `RuntimeExceptions`.

Когда значительная часть вызываемого кода просто использует непроверяемые исключения, игра заканчивается. Вы больше не можете рассчитывать на проверяемые исключения, которые предупредят вас о том, что функция может дать сбой. Вместо этого ваши действия должны быть сведены к некоторому тактическому защитному программированию и старой технике запуска его в производство, просмотру ошибок, которые вы сможете отловить, и добавлению кода для их обработки.

Последним гвоздем в крышку гроба проверяемых исключений стало введение лямбд в Java 8. Было принято решение не объявлять тип исключения в сигнатуре функциональных интерфейсов, введенных для поддержки лямбд (`Producer`, `Consumer` и т. д.), поэтому они не могут объявлять проверяемые исключения. Это не стало непреодолимой проблемой, но, честно говоря, ваши авторы тоже сдались. В конечном результате старый стандартный Java API объявляет проверяемые исключения (в частности, `IOException`), которые новый стандартный API (в частности, потоки) заставляет разработчиков отклонять.

Kotlin и исключения

В Kotlin также предусмотрены исключения, потому что он работает на JVM, а исключения встроены в эту платформу. Однако он не обрабатывает проверяемые исключения, потому что Java уже проиграла эту битву, и, как и в случае с Java, их трудно согласовать с функциями более высокого порядка. Kotlin способен в значительной степени игнорировать проверяемые исключения, поскольку они являются не функцией JVM, а принадлежат компилятору Java. Компилятор записывает в байт-код, какие проверяемые исключения объявляет метод (чтобы иметь возможность их проверять), но самой JVM все равно.

В результате программы Kotlin по умолчанию ничем не лучше и не хуже большинства программ Java, когда дело доходит до обработки ошибок.

Исключением из этого правила является то, что, как мы отмечали ранее, Kotlin может использовать `null` для указания ошибки, зная, что вызывающим сторонам придется принять во внимание возможность `null`. Примером может служить в среде выполнения строка:

```
<T> Iterable<T>.firstOrNull():T?
```

Что, однако, характерно, среда выполнения также определяет функцию `first()`, которая выдает `NoSuchElementException`, если коллекция пустая.

За пределами исключений: функциональная обработка ошибок

Статически типизированные языки функционального программирования часто отвергают исключения в пользу другого метода обработки ошибок, основанного на типе *Either*. Вскоре мы увидим, что такое тип Either, но почему функциональные программисты не любят исключения?

Отличительной особенностью функционального программирования является *ссылочная прозрачность*. Когда выражение является ссылочно прозрачным, мы можем безопасно заменить его результатом его вычисления. Итак, если мы напишем:

```
val secondsIn24hours = 60 * 60 * 24
```

то можем заменить `60 * 60` на `3600`, или `60 * 24` на `1440`, не влияя на результаты. Да и компилятор может решить заменить всё выражение на `86400`, и (если не проверить байт-код или не воспользоваться отладчиком) мы ничего про это не узнаем.

По контрасту выражение:

```
secondsIn(today())
```

не является ссылочно прозрачным, потому что функция `today()` сегодня даст результат не такой, как вчера, поскольку в любой день к расчету может быть добавлена дополнительная секунда. В результате значение `secondsIn(today())` может различаться в зависимости от того, когда мы ее вызываем, и мы не можем просто подставлять одно и то же значение для выражения каждый раз, когда его используем.

Это та же концепция, которую мы рассматривали в *главе 7*: *вычисления* являются ссылочно прозрачными, а *действия* — нет.

Почему это должно нас волновать? Потому что ссылочная прозрачность намного облегчает понимание поведения программы, что, в свою очередь, приводит к меньшему количеству ошибок и большему количеству возможностей для рефакторинга и оптимизации. Если мы хотим этого (и, по крайней мере, мы не хотим большего количества ошибок и меньшего количества возможностей), тогда нам следует стремиться к ссылочной прозрачности.

Какое это имеет отношение к обработке ошибок? Давайте вернемся к примеру `Integer.parseInt(String)`. Для заданного допустимого ввода элемент `parseInt` всегда будет возвращать одно и то же значение — следовательно, он может быть ссылочно прозрачным. Однако в тех случаях, когда `String` не представляет собой целое число, `parseInt` выдает исключение, а не возвращает результат. Мы не можем заменить результат вызова функции исключением, потому что тип выражения `Int`, а исключение `Exception` не относится к `Int`. То есть исключения нарушают ссылочную прозрачность.

Если бы вместо использования исключений мы вернулись к старому приему использования специального значения для представления ошибок, у нас была бы ссылочная прозрачность, потому что это значение ошибки может заменить выражение. В Kotlin `null` было бы отличным решением, чтобы мы могли заставить `parseInt` возвращать `Int?`. Но что, если нам нужно знать, каким был первый символ, не являющийся цифрой? Мы можем передать эту информацию в исключении, но не в возвращаемом типе `Int?`.

Можем ли мы найти способ, чтобы наша функция возвращала либо (*either*) `Int`, либо способ, которым она завершилась неудачей?

Ответ, как говорится, кроется в вопросе. Мы определяем тип `Either`, который может содержать один или два типа, но только в определенный момент времени:

```
sealed class Either<out L, out R>

data class Left<out L>(val l: L) : Either<L, Nothing>()

data class Right<out R>(val r: R) : Either<Nothing, R>()
```

В Kotlin запечатанные классы (см. *главу 18*) отлично подходят для этого, потому что мы можем определять наши собственные подтипы, но знаем, что никто другой этого не может.

Когда `Either` используется для обработки ошибки, соглашение заключается в том, что `Right` применяется для результата, а `Left` — для ошибки. Если придерживаться этого соглашения, то можно определить:

```
fun parseInt(s: String): Either<String, Int> = try {
    Right(Integer.parseInt(s))
} catch (exception: Exception) {
    Left(exception.message ?: "No message")
}
```

Как бы мы это использовали? В *главе 18* показано, что выражения `when` и интеллектуальное приведение работают очень хорошо, позволяя нам писать такой код:

```
val result: Either<String, Int> = parseInt(readLine() ?: "")
when (result) {
    is Right -> println("Your number was ${result.r}")
    is Left -> println("I couldn't read your number because ${result.l}")
}
```

Вернув `Either`, мы заставляем клиентов смириться с тем фактом, что мы, возможно, потерпели неудачу. Это дает некоторые преимущества проверяемым исключениям в функциональной форме. Чтобы использовать этот стиль, мы заставляем все функции, которые в Java мы бы объявили для создания проверяемого исключения, возвращать `Either`. Затем вызывающие абоненты либо распаковываются успешно и действуют в соответствии с этим, либо передают любую ошибку:

```
fun doubleString(s: String): Either<String, Int> {
    val result: Either<String, Int> = parseInt(s)
    return when (result) {
        is Right -> Right(2 * result.r)
        is Left -> result
    }
}
```

Хотя использование `when` для распаковки `Either` логично, но оно многословно. Этот конкретный шаблон встречается так часто, что мы определяем `map` как:

```
inline fun <L, R1, R2> Either<L, R1>.map(f: (R1) -> R2): Either<L, R2> =
    when (this) {
        is Right -> Right(f(this.r))
        is Left -> this
    }
```

Это позволяет нам записать предыдущую функцию так:

```
fun doubleString(s: String): Either<String, Int> = parseInt(s).map { 2 * it }
```

Почему эта функция называется `map`, а не `invokeUnlessLeft`? Ну если прищуриться, то можно заметить, что это примерно то же самое, что `List.map`. Здесь функция применяется к содержимому контейнера, возвращая результат в другом контейнере. В случае `Either map` применяет функцию, только если она является `Right` (безошибочной). В противном случае она проходит как `Left` на прежнем уровне.

Попрактикуйтесь в этом прищуривании, потому что сейчас мы собираемся определить:

```
inline fun <L, R1, R2> Either<L, R1>.flatMap(
    f: (R1) -> Either<L, R2>
): Either<L, R2> =
    when (this) {
        is Right -> f(this.r)
        is Left -> this
    }
```

Этот код распаковывает наше значение и использует его для вызова функции, которая, в свою очередь, может завершиться ошибкой (поскольку она возвращает `Either`). Что мы можем с этим сделать? Ну, допустим, мы хотим прочитать из `Reader` и вывести двойной результат. Мы можем определить оболочку для `readLine`, которая возвращает `Either`, а не терпит провал с исключением:

```
fun BufferedReader.eitherReadLine(): Either<String, String> =
    try {
        val line = this.readLine()
        if (line == null)
            Left("No more lines")
        else
            Right(line)
    } catch (x: IOException) {
        Left(x.message ?: "No message")
    }
```

Это позволяет нам объединять `eitherReadLine` и `doubleString` с `flatMap`:

```
fun doubleNextLine(reader: BufferedReader): Either<String, Int> =
    reader.eitherReadLine().flatMap { doubleString(it) }
```

Приведенный код вернет `Left` с ошибкой, если `eitherReadLine` завершится провалом. В противном случае он вернет результат `doubleString`, который сам по себе может быть либо `Left` — для неудачи, либо `Right` — с итоговым результатом `Int`. Таким образом, цепочка `map` и/или вызовы `flatMap` действуют как серия выражений, которые могут выдавать исключение. Первый сбой прерывает остальную часть вычисления.

Если вы пришли из объектно-ориентированного окружения, к этому стилю действительно нужно немного привыкнуть. По нашему опыту, никакое чтение не помо-

гает. Вам просто нужно собраться с духом и начать писать код таким образом, пока он не станет для вас менее странным. Позже мы разделим вашу боль, прорабатывая вместе с вами соответствующий пример.

Обработка ошибок в Kotlin

Теперь, когда мы знаем, какие варианты обработки ошибок для нас доступны, что мы должны использовать в проектах Kotlin и как перенести в них наш Java-код?

Как обычно, все зависит от обстоятельств.

Использование обнуляемых типов для представления сбоя очень эффективно при условии, что вам не нужно передавать какую-либо информацию о причине сбоя.

Вас не уволят за использование исключений в качестве стратегии по умолчанию. Однако отсутствие проверки типов затрудняет передачу информации о том, какой код какому сбою подвержен, что, в свою очередь, затрудняет создание надежных систем. Подливая масла в огонь, скажем, что вы потеряете преимущества ссылочной прозрачности, это затруднит рефакторинг и исправление вашей ненадежной системы.

Мы предпочитаем возвращать тип `Either` из тех операций, которые вызвали бы проверяемое исключение в Java, либо из-за проблем с вводом/выводом, либо потому, что, как и `parseInt`, они не могут выдать результат для всех входных данных. Это позволяет нам зарезервировать использование исключений для более опасных проблем. `Errors` по-прежнему подходят для неустранимых программных ошибок — в этом случае мы должны спроектировать наши системы таким образом, чтобы программа завершалась и перезапускалась каким-либо другим процессом. `RuntimeExceptions` все еще хороши для оповещения, когда мы, как программисты, допускаем ошибку: `IndexOutOfBounds` и тому подобные. Если мы тщательно спроектировали нашу систему, она должна быть способна справиться с этими проблемами и обрабатывать другие входные данные, которые не сталкиваются с той же проблемой.

Какой тип `Either` следует выбрать? Встроенный в Kotlin тип `Result` — это на момент подготовки книги неприятный заполнитель, который просто дразнит и мешает. Он предназначен для сопрограмм, ограничен `Exception` (на самом деле `Throwable`) в качестве значения ошибки, и IntelliJ «стонет», если вы используете его в качестве типа свойства. Было бы неплохо, если бы он не был опубликован в пакете `kotlin`.

Но он там есть. Так что, если вы попытаетесь использовать более полезные типы под названием `Result`, то получите странные сообщения об ошибке. Пока не вспомните, что компилятор считает `Result` относящимся к типу `kotlin.Result`, вы не должны его использовать.

В доступе есть множество других типов результатов, но для этой книги мы будем использовать тип Result4k (**https://oreil.ly/F5Y4M**) — не случайно, что он написан Нэтом. В отличие от обощенного типа `Either`, описанного ранее, Result4k задает `Result<SuccessType, FailureType>` с подтипами `Success` и `Failure`, а не `Left` и `Right`. Поскольку он специализирован для представления ошибок, Result4k отменяет согла-

шение `Either`, содержащее успешный тип в качестве первого из обобщенных параметров. Он также может предлагать такие операции, как `onFailure` и `recover`, которые не имели бы смысла с `Either`. Мы увидим некоторые из этих операций в процессе рефакторинга.

Рефракторинг от исключений к обработке ошибок

Теперь, когда мы знаем доступные нам варианты обработки ошибок, давайте выполним рефракторинг части Java-кода в Kotlin, преобразуя обработку ошибок по ходу работы.

В `Travelator` есть конечная точка HTTP, которая позволяет клиентскому приложению регистрировать `Customer` (пример 19.1).

Пример 19.1
[errors.0:src/main/java/travelator/handlers/CustomerRegistrationHandler.java]

```
public class CustomerRegistrationHandler {

    private final IRegisterCustomers registration;
    private final ObjectMapper objectMapper = new ObjectMapper();

    public CustomerRegistrationHandler(IRegisterCustomers registration) {
        this.registration = registration;
    }

    public Response handle(Request request) {
        try {
            RegistrationData data = objectMapper.readValue(
                request.getBody(),
                RegistrationData.class
            );
            Customer customer = registration.register(data);
            return new Response(HTTP_CREATED,
                objectMapper.writeValueAsString(customer)
            );
        } catch (JsonProcessingException x) {
            return new Response(HTTP_BAD_REQUEST);
        } catch (ExcludedException x) {
            return new Response(HTTP_FORBIDDEN);
        } catch (DuplicateException x) {
            return new Response(HTTP_CONFLICT);
        } catch (Exception x) {
            return new Response(HTTP_INTERNAL_ERROR);
        }
    }
}
```

Работа `CustomerRegistrationHandler` заключается в извлечении данных из тела запроса, отправке их в `registration` для обработки и возврате ответа либо с JSON-представлением записи `Customer`, либо с подходящим кодом состояния ошибки.

HTTP

Мы бы предпочли не привязывать наш пример кода к конкретному фреймворку Java HTTP, поэтому абстрагировали входящие вызовы с помощью простой функции, которая принимает `Request` и возвращает `Response`.

Коды статуса HTTP — это еще один пример типа результата. Протокол HTTP возвращает ошибки 4*xx*, когда запрос не выполняется, потому что он был в чем-то неправильным, ошибки 5*xx*, когда запрос не может быть обработан по причинам, связанным с сервером. Коды статуса 2*xx* — это случаи успешного выполнения, а коды 1*xx* и 3*xx* служат для оповещения о продолжающемся взаимодействии.

Если мы ценим способность правильно обрабатывать различные типы ошибок, то должны в нашем приложении позаботиться о том, чтобы правильно сопоставлять типы ошибок с кодами статуса и наоборот при проектировании систем, которые обмениваются данными по протоколу HTTP.

`CustomerRegistration` реализует бизнес-правила, которые заключаются в том, что потенциальные клиенты должны быть проверены на соответствие `ExclusionList`. Мы не хотим позволять заведомо нежелательным лицам регистрироваться и злоупотреблять нашими услугами, поэтому на текущий момент отклоняем их (пример 19.2).

Пример 19.2 [errors.0:src/main/java/travelator/CustomerRegistration.java]

```
public class CustomerRegistration implements IRegisterCustomers {
    private final ExclusionList exclusionList;
    private final Customers customers;

    public CustomerRegistration(
        Customers customers,
        ExclusionList exclusionList
    ) {
        this.exclusionList = exclusionList;
        this.customers = customers;
    }

    public Customer register(RegistrationData data)
        throws ExcludedException, DuplicateException {
        if (exclusionList.exclude(data)) {
            throw new ExcludedException();
        } else {
            return customers.add(data.name, data.email);
        }
    }
}
```

Взгляните на положение `throws` в `register`. Это говорит нам о том, что метод может завершиться неудачей из-за явного исключения, но также и о том, что `customers.add`

может потерпеть неудачу с DuplicateException. Перед вами интерфейс Customers (пример 19.3).

Пример 19.3 [errors.0:src/main/java/travelator/Customers.java]

```
public interface Customers {

    Customer add(String name, String email) throws DuplicateException;

    Optional<Customer> find(String id);
}
```

Наконец, Customer — это еще один тип значения. Вот он после преобразования в Kotlin (пример 19.4).

Пример 19.4 [errors.1:src/main/java/travelator/Customer.kt]

```
data class Customer(
    val id: String,
    val name: String,
    val email: String
)
```

Это типично для стиля Java ваших авторов. Он выражает все то, что могло бы пойти не так, если бы проверялось на ExcludedException и DuplicateException. Эти исключения ловят ошибки на верхнем уровне в handle, где они доводятся до принимающей стороны — в нашем случае в качестве кодов статуса HTTP. Ваш стиль может заключаться в использовании непроверяемых исключений, и в этом случае этот код будет похож, но без исключений как части сигнатур методов.

Один момент, который мы не замечаем, — это какое-либо проверяемое исключение, связанное со сбоями при сохранении Customer в Customers::add. Этот метод будет взаимодействовать по сети с базой данных, но наш код запроса, очевидно, в какой-то момент явно проглатывает IOException и на его место вызывает RuntimeException. Они будут распространяться из CustomerRegistration::register, но окажутся пойманы на верхнем уровне CustomerRegistrationHandler и отправлены обратно клиенту в виде сообщения HTTP_INTERNAL_ERROR (500). Жаль, что мы не регистрируем никакой информации об этих блуждающих RuntimeException, потому что они могут выявлять систематические проблемы с подключением или скрывать частые NullPointerException в каком-то низкоуровневом коде. Вероятно, кому-то следует заняться этим вопросом, но пока что у нас есть более короткий пример, который можно показать в этой книге.

Наша стратегия преобразования

Если бы мы просто преобразовали этот код в Kotlin, то потеряли бы преимущества проверяемых исключений, которые сообщают нам, что может пойти не так, и пока-

зывают, как мы решаем эти проблемы. Так что по мере преобразования мы заменим обработку ошибок на основе исключений функциональной альтернативой, использующей Result4k.

В этом примере мы начнем с самого низкого уровня и будем продвигаться вверх, поддерживая работу более высоких уровней до тех пор, пока предсказуемые случаи ошибок (те, которые в настоящее время выражаются как проверяемые исключения) больше не будут использовать исключения. В то же время мы должны помнить, что практически любая инструкция в JVM может привести к сбою, поэтому нам необходимо защититься от этих проблем во время выполнения.

Начнем с самого низа

Если сейчас мы преобразуем `Customers` в Kotlin, то получим следующее (пример 19.5).

Пример 19.5 [errors.3:src/main/java/travelator/Customers.kt]

```
interface Customers {

    @Throws(DuplicateException::class) ❶
    fun add(name: String, email: String): Customer

    fun find(id: String): Optional<Customer>
}
```

❶ Хотя в Kotlin нет проверяемых исключений, аннотация `@Throws` взаимодействует с кодом Java, добавляя исключение к сигнатуре метода в байт-коде. Без этого Java-реализация `Customers`, выдающая `DuplicateException`, не может выполнить метод. Хуже того, код Java, который вызывает метод в интерфейсе, не сможет перехватить исключение или объявить, что оно передается, потому что это будет ошибкой компиляции кода Java для обработки проверяемого исключения, которое компилятор считает невозможным.

Наша стратегия состоит в том, чтобы добавить в наш интерфейс версию `Customers::add`, которая вместо исключения возвращает `Result<Customer, DuplicateException>`. Если бы мы начинали с нуля, то не использовали бы `DuplicateException` в качестве типа ошибки, но здесь это позволяет нам легко взаимодействовать с Java. Сейчас мы собираемся пока сохранить текущую версию, чтобы не нарушать работу существующих вызывающих объектов. Затем мы преобразуем эти объекты для использования версии `Result` и удаления старой версии, когда появится такая возможность. Совершенно верно — это наш старый друг *«Рефакторинг с помощью Expand-and-Contract»* (см. соответствующую врезку в *разд. «Рефакторинг от необязательных типов к обнуляемым» главы 4*).

Как мы должны назвать метод, который работает как `Customers::add`, но возвращает `Result`? Мы не можем тоже назвать его `add`, поскольку они оба имеют одинаковые

параметры, поэтому сейчас присвоим ему имя addToo. Если новый метод делегирует add, мы можем сделать его методом по умолчанию, доступным для всех реализаций (пример 19.6).

Пример 19.6 [errors.5:src/main/java/travelator/Customers.kt]

```
interface Customers {

    @Throws(DuplicateException::class)
    fun add(name: String, email: String): Customer

    fun addToo(name:String, email:String)
        : Result<Customer, DuplicateException> =
        try {
            Success(add(name, email))
        } catch (x: DuplicateException) {
            Failure(x)
        }

    fun find(id: String): Optional<Customer>
}
```

Именование

Немного раздражает, что мы не можем назвать новый метод тоже add, но JVM не позволяет методам, которые отличаются только типом возвращаемого значения, иметь одинаковые имена.

Если мы в подобных ситуациях не можем придумать достаточно хорошее название, лучше уж тогда выбрать плохое. По всей вероятности, позже нам удастся придумать название получше, а плохое название снизит риск того, что мы согласимся на недостаточно хорошее.

В нашем же случае это не должно иметь никакого значения, потому что к концу этого этапа рефакторинга исходный метод мы удалим и сможем воспользоваться его именем.

Теперь, когда у нас есть и версия исключения, и версия результата метода, мы можем перенести вызывающие объекты версии исключения. Хотя у нас есть возможность использовать Result4k из Java, это намного удобнее сделать из Kotlin. Итак, давайте возьмем вызывающий объект add — CustomerRegistration (пример 19.7).

Пример 19.7 [errors.5:src/main/java/travelator/CustomerRegistration.java]

```
public class CustomerRegistration implements IRegisterCustomers {

    private final ExclusionList exclusionList;
    private final Customers customers;

    public CustomerRegistration(
        Customers customers,
```

```
        ExclusionList exclusionList
    ) {
        this.exclusionList = exclusionList;
        this.customers = customers;
    }

    public Customer register(RegistrationData data)
        throws ExcludedException, DuplicateException {
        if (exclusionList.exclude(data)) {
            throw new ExcludedException();
        } else {
            return customers.add(data.name, data.email);
        }
    }
}
```

И преобразуем этот код в Kotlin (пример 19.8).

Пример 19.8 [errors.6:src/main/java/travelator/CustomerRegistration.kt]

```
class CustomerRegistration(
    private val customers: Customers,
    private val exclusionList: ExclusionList
) : IRegisterCustomers {

    @Throws(ExcludedException::class, DuplicateException::class)
    override fun register(data: RegistrationData): Customer {
        return if (exclusionList.exclude(data)) {
            throw ExcludedException()
        } else {
            customers.add(data.name, data.email)
        }
    }
}
```

Выражение `customers.add` является тем, что может выдать `DuplicateException`. Мы собираемся заменить его вызовом `addToo`, но сохраняя прежнее поведение. Так что извлекаем `result` как локальный (пример 19.9).

Пример 19.9 [errors.7:src/main/java/travelator/CustomerRegistration.kt]

```
@Throws(ExcludedException::class, DuplicateException::class)
override fun register(data: RegistrationData): Customer {
    return if (exclusionList.exclude(data)) {
        throw ExcludedException()
    } else {
        val result = customers.add(data.name, data.email)
```

```
        result
    }
}
```

Если мы теперь вызовем вместо этого `addToo`, он больше не станет выдавать ошибку, но исключение будет возвращено в `Result`. Этот код еще пока не может быть скомпилирован (пример 19.10).

Пример 19.10 [errors.8:src/main/java/travelator/CustomerRegistration.kt]

```
@Throws(ExcludedException::class, DuplicateException::class)
override fun register(data: RegistrationData): Customer {
    return if (exclusionList.exclude(data)) {
        throw ExcludedException()
    } else {
        val result: Result<Customer, DuplicateException> =
            customers.addToo(data.name, data.email)
        result ❶
    }
}
```

❶ Несоответствие типов. Требуется:

```
Customer Found: Result<Customer, DuplicateException>.
```

Мы получили результат `Result`, и теперь нам необходимо распаковать его. Когда внутри него заключено значение `Success`, мы хотим вернуть обернутое значение. Когда же значение `Failure`, выдать обернутое `DuplicateException` — чтобы сохранить текущее поведение `register` (пример 19.11).

Пример 19.11 [errors.9:src/main/java/travelator/CustomerRegistration.kt]

```
@Throws(ExcludedException::class, DuplicateException::class)
override fun register(data: RegistrationData): Customer {
    return if (exclusionList.exclude(data)) {
        throw ExcludedException()
    } else {
        val result: Result<Customer, DuplicateException> =
            customers.addToo(data.name, data.email)
        when (result) {
            is Success<Customer> ->
                result.value
            is Failure<DuplicateException> ->
                throw result.reason
        }
    }
}
```

На случай, когда тип ошибки `Exception`, у Result4k есть функция для сокращения такой ситуации: `Result::orThrow` (пример 19.12).

Пример 19.12 [errors.10:src/main/java/travelator/CustomerRegistration.kt]

```
@Throws(ExcludedException::class, DuplicateException::class)
override fun register(data: RegistrationData): Customer {
    return if (exclusionList.exclude(data)) {
        throw ExcludedException()
    } else {
        val result: Result<Customer, DuplicateException> =
            customers.addToo(data.name, data.email)
        result.orThrow()
    }
}
```

Теперь мы можем использовать встроенное выражение, чтобы вернуться к более короткой форме представления (пример 19.13).

Пример 19.13 [errors.11:src/main/java/travelator/CustomerRegistration.kt]

```
@Throws(ExcludedException::class, DuplicateException::class)
override fun register(data: RegistrationData): Customer {
    return if (exclusionList.exclude(data)) {
        throw ExcludedException()
    } else {
        customers.addToo(data.name, data.email).orThrow()
    }
}
```

Полученное вложение слишком запутанно и неудобно, поэтому давайте упростим его, нажимая комбинацию клавиш <Alt>+<Enter> для каждой следующей команды: **Заменить 'if' на 'when'** (Replace 'if' with 'when'), **Заменить return выражением 'when'** (Replace return with 'when' expression) и **Удалить фигурные скобки из всех записей 'when'** (Remove braces from all 'when' entries). Результат их выполнения показан в примере 19.14.

Пример 19.14 [errors.12:src/main/java/travelator/CustomerRegistration.kt]

```
@Throws(ExcludedException::class, DuplicateException::class)
override fun register(data: RegistrationData): Customer {
    when {
        exclusionList.exclude(data) -> throw ExcludedException()
        else -> return customers.addToo(data.name, data.email).orThrow()
    }
}
```

Великолепно. Мы заменили одно из применений исключений типом результата. Давайте немного отдохнем.

Контракт

Готовы снова отправиться в путь? Хорошо.

Теперь нам нужно выбрать, какой алгоритм применять первым: *поиск в глубину* (depth-first) или *поиск в ширину* (breadth-first). Поиск в глубину обратился бы к вызывающему объекту `CustomerRegistration::register`, поиск в ширину сначала бы исправил другие вызывающие объекты `Customers::add`, чтобы мы могли удалить их. Так получилось, что в нашем примере кода нет других вызывающих объектов `add`, поэтому сначала осуществить поиск в ширину — это не вариант, и мы можем перейти к этапу контракта метода «Expand-and-Contract».

К настоящему моменту у нас есть две реализации `Customers::add`. Одна из них является производственной реализацией, которая взаимодействует с базой данных, другая — тестовая реализация. Наш код теперь вызывает их через реализацию по умолчанию `Customers::addToo`, которую мы добавили в интерфейс. Мы хотим удалить реализации `add`, поэтому нам нужно выполнить `addToo` напрямую. Давайте посмотрим на (непотокобезопасную) тестовую версию (пример 19.15).

Пример 19.15 [errors.12:src/test/java/travelator/InMemoryCustomers.java]

```
public class InMemoryCustomers implements Customers {

    private final List<Customer> list = new ArrayList<>();
    private int id = 0;

    @Override
    public Customer add(String name, String email) throws DuplicateException {
        if (list.stream().anyMatch( item -> item.getEmail().equals(email)))
            throw new DuplicateException(
                "customer with email " + email + " already exists"
            );
        int newId = id++;
        Customer result = new Customer(Integer.toString(newId), name, email);
        list.add(result);
        return result;
    }

    @Override
    public Optional<Customer> find(String id) {
        return list.stream()
            .filter(customer -> customer.getId().equals(id))
            .findFirst();
    }
```

```
    // для теста
    public void add(Customer customer) {
        list.add(customer);
    }

    public int size() {
        return list.size();
    }
}
```

Самый простой способ выполнить `addToo` здесь — вероятно, это просто дублировать `add` и исправить его, вернув `Failure` там, где мы выдавали значение `Success` для «счастливого пути» (пример 19.16).

Пример 19.16 [errors.13:src/test/java/travelator/InMemoryCustomers.java]

```
@SuppressWarnings("unchecked")
@Override
public Result<Customer, DuplicateException> addToo(
    String name, String email
) {
    if (list.stream().anyMatch( item -> item.getEmail().equals(email)))
        return new Failure<>(
            new DuplicateException(
                "customer with email " + email + " already exists"
            )
        );
    int newId = id++;
    Customer result = new Customer(Integer.toString(newId), name, email);
    list.add(result);
    return new Success<Customer>(result);
}
```

Мы также можем использовать эту стратегию для добавления `addToo` к нашим рабочим реализациям `Customers`. Опустим детали. Мы сможем удалить неиспользуемые `add` из реализаций и интерфейса, как только закончим процедуру, а затем переименуем `addToo` в `add` (пример 19.17).

Пример 19.17 [errors.14:src/main/java/travelator/Customers.kt]

```
interface Customers {

    fun add(name:String, email:String): Result<Customer, DuplicateException>

    fun find(id: String): Optional<Customer>
}
```

Клиенты `Customers` снова будут вызывать `add`, хотя эта версия возвращает `Result`, а не объявляет проверяемые исключения (пример 19.18).

Пример 19.18 [errors.14:src/main/java/travelator/CustomerRegistration.kt]

```
class CustomerRegistration(
    private val customers: Customers,
    private val exclusionList: ExclusionList
) : IRegisterCustomers {

    @Throws(ExcludedException::class, DuplicateException::class)
    override fun register(data: RegistrationData): Customer {
        when {
            exclusionList.exclude(data) -> throw ExcludedException()
            else -> return customers.add(data.name, data.email).orThrow()
        }
    }
}
```

Мы оставили `InMemoryCustomers` на Java только лишь с тем, чтобы продемонстрировать, что мы можем возвращать типы Result4k из нашего старого кода, но мы не можем сопротивляться преобразованию, потому что в коде теперь есть ряд предупреждений типа `Not annotated [X] overrides @NotNull [X]`.

После преобразования, включая переход от потоков к операциям с коллекциями Kotlin (см. *главу 13*), мы получим следующий код (пример 19.19).

Пример 19.19 [errors.15:src/test/java/travelator/InMemoryCustomers.kt]

```
class InMemoryCustomers : Customers {

    private val list: MutableList<Customer> = ArrayList()

    private var id = 0

    override fun add(name: String, email: String)
        : Result<Customer, DuplicateException> =
        when {
            list.any { it.email == email } -> Failure(
                DuplicateException(
                    "customer with email $email already exists"
                )
            )
            else -> {
                val result = Customer(id++.toString(), name, email)
                list.add(result)
                Success(result)
            }
        }
```

```
    override fun find(id: String): Optional<Customer> =
        list.firstOrNull { it.id == id }.toOptional()

    // для теста
    fun add(customer: Customer) {
        list.add(customer)
    }

    fun size(): Int = list.size
}
```

Давайте подведем итоги: `Customers` теперь в Kotlin, и `add` возвращает `Result` вместо того, чтобы выдавать `DuplicateException` (пример 19.20).

Пример 19.20 [errors.15:src/main/java/travelator/Customers.kt]

```
interface Customers {

    fun add(name:String, email:String): Result<Customer, DuplicateException>

    fun find(id: String): Optional<Customer>
}
```

Но `IRegisterCustomers` все еще в Java и все еще выдает два типа исключений (пример 19.21).

Пример 19.21 [errors.15:src/main/java/travelator/IRegisterCustomers.java]

```
public interface IRegisterCustomers {
    Customer register(RegistrationData data)
        throws ExcludedException, DuplicateException;
}
```

Однако `CustomerRegistration` теперь в Kotlin, и это то место кода, в котором мы сейчас находимся, — между `Result.Error` и `DuplicateException`. Поэтому применим `orThrow` (пример 19.22).

Пример 19.22 [errors.15:src/main/java/travelator/CustomerRegistration.kt]

```
class CustomerRegistration(
    private val customers: Customers,
    private val exclusionList: ExclusionList
) : IRegisterCustomers {

    @Throws(ExcludedException::class, DuplicateException::class)
    override fun register(data: RegistrationData): Customer {
        when {
            exclusionList.exclude(data) -> throw ExcludedException()
```

```
            else -> return customers.add(data.name, data.email).orThrow()
        }
    }
}
```

Мы преобразовали целый уровень нашего взаимодействия, чтобы использовать тип результата, и можем перейти к следующему.

Отступление

Если мы хотим в случае с `IRegisterCustomers::register` следовать той же схеме, какую мы применяли с `Customers`, — предоставлять реализацию переходника по умолчанию между обработкой исключений и возвратом ошибок — нам придется решить вопрос о том, как выразить результат функции, которая может выйти из строя. Это связано с тем, что в настоящее время `register` объявляет, что выдает проверяемые исключения `ExcludedException` и `DuplicateException`. Но в коде мы хотим видеть что-то вроде `Result<Customer, Either<ExcludedException, DuplicateException>>`.

Мы *могли бы* использовать обобщенный тип `Either`, но это только заводит нас так же далеко, как и указанная стратегия. В отличие от Java, где порядок объявления исключений не имеет значения, `Either<ExcludedException, DuplicateException>` — не то же самое, что `Either<DuplicateException, ExcludedException>`. `Either` в лучшем случае сбивает с толку, но станет еще хуже, если у нас когда-нибудь будет более двух исключений: `OneOf<ExcludedException, Duplicate Exception, SomeOtherProblem>` — это просто ужасно.

Другой вариант — перейти к общему суперклассу двух исключений и объявить возвращаемый тип как `Result<Customer, Exception>`. Это не пройдет коммуникационный тест — мы не можем взглянуть на сигнатуру и получить какие-либо подсказки об ожидаемых типах ошибок.

Вместо этого наша лучшая стратегия заключается не в том, чтобы пытаться выразить ошибку в терминах существующих типов, а в сопоставлении с новым типом.

Поскольку *exception* (исключение) и *error* (ошибка) — это перегруженные термины, мы выбрали `RegistrationProblem` с подтипами `Excluded` (который не несет никакой дополнительной информации и поэтому может быть `object`) и `Duplicate` (который несет любое сообщение из исходного `DuplicateException`), как показано в примере 19.23.

Пример 19.23 [errors.16:src/main/java/travelator/IRegisterCustomers.kt]

```
sealed class RegistrationProblem

object Excluded : RegistrationProblem()

data class Duplicate(
    val message: String?
) : RegistrationProblem()
```

Делая `RegistrationProblem` запечатанным классом, мы во время компиляции будем знать, какие подклассы могут существовать и, следовательно, какие ошибки должны быть обработаны, — это очень похоже на проверяемую сигнатуру исключения метода.

Следуя предыдущему шаблону, мы можем использовать `RegistrationProblem`, добавив реализацию по умолчанию `registerToo` к интерфейсу, который возвращает `Result<Customer, RegistrationProblem>` (пример 19.24).

Пример 19.24 [errors.16:src/main/java/travelator/IRegisterCustomers.kt]

```
interface IRegisterCustomers {

    @Throws(ExcludedException::class, DuplicateException::class)
    fun register(data: RegistrationData): Customer

    fun registerToo(data: RegistrationData):
        Result<Customer, RegistrationProblem> =
        try {
            Success(register(data))
        } catch (x: ExcludedException) {
            Failure(Excluded)
        } catch (x: DuplicateException) {
            Failure(Duplicate(x.message))
        }
}
```

Теперь мы можем перенести вызывающие объекты из `register` в `registerToo`. Начнем с конвертации `Customer RegistrationHandler` в Kotlin (пример 19.25).

Пример 19.25 [errors.17:src/main/java/travelator/handlers/CustomerRegistrationHandler.kt]

```
class CustomerRegistrationHandler(
    private val registration: IRegisterCustomers
) {
    private val objectMapper = ObjectMapper()

    fun handle(request: Request): Response {
        return try {
            val data = objectMapper.readValue(
                request.body,
                RegistrationData::class.java
            )
            val customer = registration.register(data)
            Response(
                HTTP_CREATED,
                objectMapper.writeValueAsString(customer)
            )
```

```
        } catch (x: JsonProcessingException) {
            Response(HTTP_BAD_REQUEST)
        } catch (x: ExcludedException) {
            Response(HTTP_FORBIDDEN)
        } catch (x: DuplicateException) {
            Response(HTTP_CONFLICT)
        } catch (x: Exception) {
            Response(HTTP_INTERNAL_ERROR)
        }
    }
}
```

Затем, как и делали раньше, мы переходим к вызову нового метода (`registerToo`) вместо старого (`register`) и интерпретируем тип возвращаемого выражения, как `when` (пример 19.26).

Пример 19.26
[errors.18:src/main/java/travelator/handlers/CustomerRegistrationHandler.kt]

```
class CustomerRegistrationHandler(
    private val registration: IRegisterCustomers
) {
    private val objectMapper = ObjectMapper()

    fun handle(request: Request): Response {
        return try {
            val data = objectMapper.readValue(
                request.body,
                RegistrationData::class.java
            )
            val customerResult = registration.registerToo(data)
            when (customerResult) {
                is Success -> Response(
                    HTTP_CREATED,
                    objectMapper.writeValueAsString(customerResult.value)
                )
                is Failure -> customerResult.reason.toResponse()
            }
        } catch (x: JsonProcessingException) {
            Response(HTTP_BAD_REQUEST)
        } catch (x: ExcludedException) {
            Response(HTTP_FORBIDDEN)
        } catch (x: DuplicateException) {
            Response(HTTP_CONFLICT)
        } catch (x: Exception) {
            Response(HTTP_INTERNAL_ERROR)
        }
    }
}
```

```
private fun RegistrationProblem.toResponse() = when (this) {
    is Duplicate -> Response(HTTP_CONFLICT)
    is Excluded -> Response(HTTP_FORBIDDEN)
}
```

Наконец, мы можем удалить ненужные случаи исключения и упростить случай ошибки с помощью `map` и `recover`. `Result::recover` — это функция-расширение Result4k, которая распаковывает результат, если он положительный `Success`. В противном случае она возвращает результат маппинга провала `reason` (пример 19.27).

Пример 19.27
[errors.19:src/main/java/travelator/handlers/CustomerRegistrationHandler.kt]

```
fun handle(request: Request): Response =
    try {
        val data = objectMapper.readValue(
            request.body,
            RegistrationData::class.java
        )
        registration.registerToo(data)
            .map { value ->
                Response(
                    HTTP_CREATED,
                    objectMapper.writeValueAsString(value)
                )
            }
            .recover { reason -> reason.toResponse() }
    } catch (x: JsonProcessingException) {
        Response(HTTP_BAD_REQUEST)
    } catch (x: Exception) {
        Response(HTTP_INTERNAL_ERROR)
    }
```

Обратите внимание, что этот код по-прежнему не свободен от исключений. Во-первых, `ObjectMapper` все еще может выдавать `JSONProcessingException`. Такова реальность API-интерфейсов Java (и, откровенно говоря, большинства из Kotlin), но код безопасен и хорошо работает, поскольку выдача ошибок и их перехват выполняются одним и тем же методом. Во-вторых, мы все еще должны рассмотреть другие исключения `RuntimeException`, которые могут выдаваться в любом месте кода: `NullPointerException` и т. д. Они могли бы пересечь границы функций и просочиться сюда, где проверка останавливается на верхнем уровне, возвращающем `HTTP_INTERNAL_ERROR`. Реальность такова, что у нас все еще могут появиться неожиданные исключения, но ожидаемые случаи сбоев теперь выражаются в `Results`, о чем и сообщается в нашем коде.

Еще исправления

Теперь мы можем признаться, что `RegistrationHandlerTests` был нарушен несколько этапов назад. В рабочей ситуации мы бы сразу же исправили его, но это прервало бы наше объяснение.

Проблема в том, что тесты проводились с применением имитаций (моков), и они ожидают вызовов `IRegister.register`, но мы уже обращаемся к `registerToo` (пример 19.28).

Пример 19.28
[errors.20:src/test/java/travelator/handlers/CustomerRegistrationHandlerTests.java]

```
public class CustomerRegistrationHandlerTests {

    final IRegisterCustomers registration =
        mock(IRegisterCustomers.class);
    final CustomerRegistrationHandler handler =
        new CustomerRegistrationHandler(registration);

    final String fredBody = toJson(
        "{ 'name' : 'fred', 'email' : 'fred@bedrock.com' }"
    );
    final RegistrationData fredData =
        new RegistrationData("fred", "fred@bedrock.com");

    @Test
    public void returns_Created_with_body_on_success()
        throws DuplicateException, ExcludedException {
        when(registration.register(fredData))
            .thenReturn(
                new Customer("0", fredData.name, fredData.email)
            );

        String expectedBody = toJson(
            "{'id':'0','name':'fred','email':'fred@bedrock.com'}"
        );
        assertEquals(
            new Response(HTTP_CREATED, expectedBody),
            handler.handle(new Request(fredBody))
        );
    }

    @Test
    public void returns_Conflict_for_duplicate()
        throws DuplicateException, ExcludedException {
```

```
        when(registration.register(fredData))
            .thenThrow(
                new DuplicateException("deliberate")
            );

        assertEquals(
            new Response(HTTP_CONFLICT),
            handler.handle(new Request(fredBody))
        );
    }
    ...

    private String toJson(String jsonIsh) {
        return jsonIsh.replace('\'', '"');
    }
}
```

Чтобы исправить тесты, нам нужно изменить вызов с `register`, возвращающего `Customer` или фиксирующего ошибку, на `registerToo`, возвращающий `Result<Customer, RegistrationProblem>` (пример 19.29).

Пример 19.29
[errors.21:src/test/java/travelator/handlers/CustomerRegistrationHandlerTests.java]

```
@Test
public void returns_Created_with_body_on_success() {

    when(registration.registerToo(fredData))
        .thenReturn(new Success<>(
            new Customer("0", fredData.name, fredData.email)
        ));

    String expectedBody = toJson(
        "{'id':'0','name':'fred','email':'fred@bedrock.com'}"
    );
    assertEquals(
        new Response(HTTP_CREATED, expectedBody),
        handler.handle(new Request(fredBody))
    );
}

@Test
public void returns_Conflict_for_duplicate() {

    when(registration.registerToo(fredData))
        .thenReturn(new Failure<>(
            new Duplicate("deliberate")
        ));
```

```
    assertEquals(
        new Response(HTTP_CONFLICT),
        handler.handle(new Request(fredBody))
    );
}
    ...
```

Тесты на самом деле упрощены, потому что вместо выбора `thenReturn` или `thenThrow`, мы теперь постоянно реализуем имитацию при помощи `thenReturn` с `Success` или `Failure` соответственно.

Теперь, когда наши тесты снова проходят, мы можем вернуться к производственному коду и реализовать `CustomerRegistration::registerToo` непосредственно. Вместо любой более умной идеи мы делаем это, дублируя метод `register` и усложняя обработку ошибок при помощи `Result::mapFailure` (часть Result4k), преобразующей `DuplicateException` в `Duplicate` (пример 19.30).

Пример 19.30 [errors.22:src/main/java/travelator/CustomerRegistration.kt]

```
class CustomerRegistration(
    private val customers: Customers,
    private val exclusionList: ExclusionList
) : IRegisterCustomers {
    @Throws(ExcludedException::class, DuplicateException::class)
    override fun register(data: RegistrationData): Customer {
        when {
            exclusionList.exclude(data) -> throw ExcludedException()
            else -> return customers.add(data.name, data.email).orThrow()
        }
    }

    override fun registerToo(
        data: RegistrationData
    ): Result<Customer, RegistrationProblem> {
        return when {
            exclusionList.exclude(data) -> Failure(Excluded)
            else -> customers.add(data.name, data.email)
                .mapFailure { exception: DuplicateException -> ❶
                    Duplicate(exception.message)
                }
        }
    }
}
```

❶ Обратите внимание, что мы явно указываем тип лямбда-параметра в `mapFailure`. Позже мы увидим, что в случае изменения возвращаемого типа `add` на другой тип сбоя компилятор заставит нас изменить способ его обработки.

С этим связаны две проблемы. Во-первых, `registerToo` не содержит тестовый код, а во-вторых, у нас есть дубликат логики, вызванный нашим дублированием `register` для создания `registerToo`. Мы можем исправить и то и другое, внедрив `register` в рамках `registerToo` — в противоположность тому, что мы сделали с `Customers` (пример 19.31).

Пример 19.31 [errors.23:src/main/java/travelator/CustomerRegistration.kt]

```
class CustomerRegistration(
    private val customers: Customers,
    private val exclusionList: ExclusionList
) : IRegisterCustomers {

    @Throws(ExcludedException::class, DuplicateException::class)
    override fun register(data: RegistrationData): Customer =
        registerToo(data).recover { error -> ❶
            when (error) {
                is Excluded -> throw ExcludedException()
                is Duplicate -> throw DuplicateException(error.message)
            }
        }

    override fun registerToo(
        data: RegistrationData
    ): Result<Customer, RegistrationProblem> {
        return when {
            exclusionList.exclude(data) -> Failure(Excluded)
            else -> customers.add(data.name, data.email)
                .mapFailure { exception: DuplicateException ->
                    Duplicate(exception.message)
                }
        }
    }
}
```

❶ Передача `registerToo` и обработка типа `Error`.

Теперь `CustomerRegistrationTests`, работающий в рамках `register`, будет для нас тестировать `registerToo` (пример 19.32).

Пример 19.32 [errors.23:src/test/java/travelator/CustomerRegistrationTests.java]

```
public class CustomerRegistrationTests {

    InMemoryCustomers customers = new InMemoryCustomers();
    Set<String> excluded = Set.of(
        "cruella@hellhall.co.uk"
    );
```

```
    CustomerRegistration registration = new CustomerRegistration(customers,
        (registrationData) -> excluded.contains(registrationData.email)
    );

    @Test
    public void adds_a_customer_when_not_excluded()
        throws DuplicateException, ExcludedException {
        assertEquals(Optional.empty(), customers.find("0"));

        Customer added = registration.register(
            new RegistrationData("fred flintstone", "fred@bedrock.com")
        );
        assertEquals(
            new Customer("0", "fred flintstone", "fred@bedrock.com"),
            added
        );
        assertEquals(added, customers.find("0").orElseThrow());
    }

    @Test
    public void throws_DuplicateException_when_email_address_exists() {
        customers.add(new Customer("0", "fred flintstone", "fred@bedrock.com"));
        assertEquals(1, customers.size());

        assertThrows(DuplicateException.class,
            () -> registration.register(
                new RegistrationData("another name", "fred@bedrock.com")
            )
        );
        assertEquals(1, customers.size());
    }

    ...
}
```

Это отличный способ сохранить и `register`, и `registerToo`, пока мы переходим от Java и исключений к Kotlin и типу ошибки. В этом случае тесты на самом деле являются последними вызывающими объектами `register`, так что давайте преобразуем их для вызова `registerToo`. Мы могли бы потратить время на то, чтобы показать, как использовать Result4k в Java, но сейчас мы все уже достаточно устали от текущего примера, поэтому преобразуем тесты в Kotlin, а затем попросим их вызвать `register` с бессмертными словами: «Вот один из вариантов рефракторинга, который мы сделали ранее» (пример 19.33).

Пример 19.33 [errors.24:src/test/java/travelator/CustomerRegistrationTests.kt]

```
@Test
fun `adds a customer when not excluded`() {
    assertEquals(Optional.empty<Any>(), customers.find("0"))
```

```
    val added = registration.registerToo(
        RegistrationData("fred flintstone", "fred@bedrock.com")
    ).valueOrNull()
    assertEquals(
        Customer("0", "fred flintstone", "fred@bedrock.com"),
        added
    )
    assertEquals(added, customers.find("0").orElseThrow())
}

@Test
fun `returns Duplicate when email address exists`() {
    customers.add(Customer("0", "fred flintstone", "fred@bedrock.com"))
    assertEquals(1, customers.size())
    val failure = registration.registerToo(
        RegistrationData("another name", "fred@bedrock.com")
    ).failureOrNull()
    assertEquals(
        Duplicate("customer with email fred@bedrock.com already exists"),
        failure
    )
    assertEquals(1, customers.size())
}

    ...
```

Теперь, когда у нас нет абонентов `register`, мы наконец-то можем удалить его и переименовать `registerToo` в `register`, заканчивая рефакторинг в код Kotlin, не содержащий исключений (примеры 19.34 и 19.35).

Пример 19.34 [errors.25:src/main/java/travelator/IRegisterCustomers.kt]

```
interface IRegisterCustomers {
    fun register(data: RegistrationData):
        Result<Customer, RegistrationProblem>
}

sealed class RegistrationProblem
object Excluded : RegistrationProblem()

data class Duplicate(
    val message: String?
) : RegistrationProblem()
```

Пример 19.35 [errors.25:src/main/java/travelator/Customers.kt]

```
interface Customers {

    fun add(name:String, email:String): Result<Customer, DuplicateException>

    fun find(id: String): Optional<Customer>
}
```

Хм, не совсем без исключений — из-за `DuplicateException`. Но на самом деле это исключение больше ниоткуда *не выдается*, просто создается и помещается в `Failure`. Мы можем исправить это, либо переименовав класс в `DuplicateCustomerProblem` и прекратив его расширение `Exception`, либо использовав существующий подкласс `Duplicate` из `RegistrationProblem`. Что лучше?

Слои

Если размышлять в категориях слоев, то `Customers` более низкий слой, чем зависящий от него `Registration`. Следовательно, `Customers` не должен зависеть от `RegistrationProblem` верхних уровней. Мы могли бы попробовать инвертировать зависимость так, чтобы подкласс `Duplicate` из `RegistrationProblem` стал подтипом (иногда даже того же типа) `DuplicateCustomerProblem`, объявленного в слое хранилища.

Такое решение здесь сработало бы, но, скорее всего, это тупик, если `Customers::add` когда-либо нужно будет объявить другой способ, который может привести к сбою. Если, например, мы хотим показать в нашем результате, что связь с базой данных может завершиться сбоем, то не можем (ну, не должны) сделать это подтипом `DuplicateCustomerProblem`. То есть мы вернемся к проблеме выражения более одного типа ошибки в одном результате.

Давайте разберемся с этим до конца. Если для `Customers::add` необходимо объявить более одного варианта сбоя — предыдущий класс `DuplicateCustomerProblem` и новый `DatabaseCustomerProblem` — мы вводим запечатанный `CustomersProblem` в качестве типа ошибки и делаем две известные проблемы лишь его подклассами (пример 19.36).

Пример 19.36 [errors.27:src/main/java/travelator/Customers.kt]

```
interface Customers {

    fun add(name:String, email:String): Result<Customer, CustomersProblem>

    fun find(id: String): Optional<Customer>
}

sealed class CustomersProblem

data class DuplicateCustomerProblem(val message: String): CustomersProblem()

data class DatabaseCustomerProblem(val message: String): CustomersProblem()
```

Далее `CustomerRegistration` вызывает `Customers::add` и обрабатывает только `DuplicateCustomerProblem` в `mapFailure` (пример 19.37).

Пример 19.37 [errors.26:src/main/java/travelator/CustomerRegistration.kt]

```
class CustomerRegistration(
    private val customers: Customers,
    private val exclusionList: ExclusionList
) : IRegisterCustomers {

    override fun register(
        data: RegistrationData
    ): Result<Customer, RegistrationProblem> {
        return when {
            exclusionList.exclude(data) -> Failure(Excluded)
            else -> customers.add(data.name, data.email)
                .mapFailure { duplicate: DuplicateCustomerProblem ->
                    Duplicate(duplicate.message)
                }
        }
    }
}
```

Этот код больше не компилируется, потому что тип сбоя здесь является базовым классом `CustomersProblem`. Обратите внимание, что мы получаем преимущества проверяемых исключений — код сообщает о том, как он может выйти из строя, и заставляет нас разбираться с этими случаями.

Теперь `Customers::add` признает, что он может потерпеть неудачу новым и интересным способом, да и `register` также вынужден придерживаться истинности. Он решает передать знания своим абонентам (ну хорошо, мы решаем за него), добавив новый подтип `DatabaseProblem` существующего запечатанного класса `RegistrationProblem` (пример 19.38).

Пример 19.38 [errors.27:src/main/java/travelator/IRegisterCustomers.kt]

```
sealed class RegistrationProblem

object Excluded : RegistrationProblem()

data class Duplicate(val message: String) : RegistrationProblem()

data class DatabaseProblem(val message: String) : RegistrationProblem()
```

Затем можно исправить `register` путем преобразования между вариантами, которые могут дать сбой `add` (`DuplicateCustomerProblem` и `DatabaseCustomerProblem`), и вариантами, которые могут дать сбой `register` (`Duplicate` и `DatabaseProblem` соответственно).

Теперь это делает выбор `map Failure` понятным (пример 19.39).

Пример 19.39 [errors.27:src/main/java/travelator/CustomerRegistration.kt]

```
override fun register(
    data: RegistrationData
): Result<Customer, RegistrationProblem> {
    return when {
        exclusionList.exclude(data) -> Failure(Excluded)
        else -> customers.add(data.name, data.email)
            .mapFailure { problem: CustomersProblem ->
                when (problem) {
                    is DuplicateCustomerProblem ->
                        Duplicate(problem.message)
                    is DatabaseCustomerProblem ->
                        DatabaseProblem(problem.message)
                }
            }
    }
}
```

Наконец, из-за добавления запечатанной иерархии `RegistrationProblem` компилятор заставляет нас рассмотреть `DatabaseProblem` в вышеследующем слое, не сумев скомпилировать `CustomerRegistrationHandler` (пример 19.40).

Пример 19.40 [errors.27:src/main/java/travelator/handlers/CustomerRegistrationHandler.kt]

```
private fun RegistrationProblem.toResponse() = when (this) {
    is Duplicate -> Response(HTTP_CONFLICT)
    is Excluded -> Response(HTTP_FORBIDDEN)
    is DatabaseProblem -> Response(HTTP_INTERNAL_ERROR) ❶
}
```

❶ Мы должны добавить условие для `DatabaseProblem`, чтобы выражение `when` скомпилировалось.

Учитывая, что `CustomerRegistrationHandler` является отправной точкой для этого взаимодействия, наша работа завершена.

Двигаемся дальше

Это была длинная глава, но ее длина пропорциональна ее важности.

Возможно, ваш Java-проект уже объявил о несостоятельности исключений без систематического использования проверяемых исключений. В этом случае политика Kotlin, согласно которой все рассматривается как непроверяемое исключение, отлично подходит.

Если вы действительно полагаетесь на проверяемые исключения и хотите перевести такой код на Kotlin или хотите активизировать свои умения по обработке ошибок в качестве части преобразования, то использование типа результата — лучшая стратегия. Там, где операция может завершиться несколькими вариантами неудачи, мы можем использовать запечатанные классы для перечисления видов сбоя за счет невозможности распространения одного и того же типа через несколько уровней. Когда у нас есть несколько слоев, код становится утомительным, но, по крайней мере, не очень подвержен ошибкам.

Мы могли бы (и, возможно, должны) написать целую книгу об обработке ошибок, но пока вы можете проследить за путешествием Дункана по этой «кроличьей норе» в его блоге (**https:// oreil.ly/kfvAn**). Наряду с рассмотренным материалом здесь показано, как уменьшить количество функций, которые подвержены сбоям, поскольку они являются частично применяемыми (**https://oreil.ly/8RoO4**).

Сокращение числа функций, которые могут завершиться сбоем, важно, потому что код, подверженный ошибкам, очень похож на действия, рассмотренные в *главе 7 «От действий к вычислениям»*. Действия засоряют своих вызывающих абонентов — по умолчанию код, вызывающий действие, становится действием. Точно так же код, вызывающий подверженный сбою код, сам подвержен сбою. Мы можем смягчить последствия как действий, так и ошибок, переместив их как можно ближе к точкам входа в нашу систему, чтобы они намного меньше загрязняли код.

В этой главе мы кратко коснулись того, как сделать код устойчивым к ошибкам, когда они возникают. Действия тоже являются проблемой, потому что они влияют на состояние нашей системы. Состояние может быть нарушено, когда необходимо обновить два места, и первое действие записывает обновление, а второе — нет, потому что ошибка произошла до того, как оно было вызвано. Строгое внимание к различию между действиями и вычислениями является ключом к созданию надежного программного обеспечения.

Мы вернемся к обработке ошибок в *главе 21 «От исключений к значениям»*.

От выполнения ввода/вывода к передаче данных

Присутствие в коде операций ввода и вывода чревато проблемами. При общении с внешним миром наша программа подвержена ошибкам, когда файлы исчезают или сетевые сокеты выходят из строя. Ввод/вывод также является действием и поэтому ограничивает наши возможности по реорганизации кода. Как мы можем уменьшить масштабы проблем, вызываемых вводом/выводом?

Теперь, когда в предыдущих главах были заложены некоторые основы, здесь мы собираемся ускорить темп, перейдя непосредственно к рефакторингу и усвоению материала по ходу дела.

Прослушивание тестов

В *главе 10* мы рассмотрели некоторый Java-код, который создавал отчет для маркетинга. Заканчивая работу с кодом, мы ввели в `HighValueCustomersReport` функции-расширения, что дало нам следующий вариант (пример 20.1).

Пример 20.1
[io-to-data.0:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
@Throws(IOException::class)
fun generate(reader: Reader, writer: Writer) {
    val valuableCustomers = reader
        .readLines()
        .toValuableCustomers()
        .sortedBy(CustomerData::score)
    writer.appendLine("ID\tName\tSpend")
    for (customerData in valuableCustomers) {
        writer.appendLine(customerData.outputLine)
    }
    writer.append(valuableCustomers.summarised())
}

private fun List<String>.toValuableCustomers() = withoutHeader()
    .map(String::toCustomerData)
    .filter { it.score >= 10 }
```

```
private fun List<String>.withoutHeader() = drop(1)

private fun List<CustomerData>.summarised(): String =
    sumByDouble { it.spend }.let { total ->
        "\tTOTAL\t${total.toMoneyString()}"
    }
```

Вот его тест после преобразования в Kotlin (пример 20.2).

Пример 20.2
[lo-to-data.1:src/test/java/travelator/marketing/HighValueCustomersReportTests.kt]

```
class HighValueCustomersReportTests {

    @Test
    fun test() {
        check(
            inputLines = listOf(
                "ID\tFirstName\tLastName\tScore\tSpend",
                "1\tFred\tFlintstone\t11\t1000.00",
                "4\tBetty\tRubble\t10\t2000.00",
                "2\tBarney\tRubble\t0\t20.00",
                "3\tWilma\tFlintstone\t9\t0.00"
            ),
            expectedLines = listOf(
                "ID\tName\tSpend",
                "4\tRUBBLE, Betty\t2000.00",
                "1\tFLINTSTONE, Fred\t1000.00",
                "\tTOTAL\t3000.00"
            )
        )
    }

    ...
    private fun check(
        inputLines: List<String>,
        expectedLines: List<String>
    ) {
        val output = StringWriter()
        generate(
            StringReader(inputLines.joinToString("\n")),
            output
        )
        assertEquals(expectedLines.joinToString("\n"), output.toString())
    }
}
```

Мы фактически не рассматривали в *главе 10* тесты, но что, если мы сделаем это сейчас — в свете одержимости ваших авторов действиями и вычислениями (см. *главу 7*)? В частности, посмотрите на функцию `check`.

Очевидно, что `check` не является вычислением (см. *разд. «Вычисления» главы 7*), потому что выполняет создание исключения вместо возврата значения. Но что, если мы взглянем на это с другой стороны (пример 20.3)?

Пример 20.3
[io-to-data.2:src/test/java/travelator/marketing/HighValueCustomersReportTests.kt]

```
private fun check(
    inputLines: List<String>,
    expectedLines: List<String>
) {
    val output = StringWriter()
    val reader = StringReader(inputLines.joinToString("\n"))
    generate(reader, output)
    val outputLines = output.toString().lines()

    assertEquals(expectedLines, outputLines)
}
```

По сути, это один и тот же код, но теперь заметно, что у нас есть вычисление, берущее `inputLines` и получающее `outputLines`, прежде чем мы перейдем к утверждению. Несмотря на то что `generate` — это действие, основанное на побочных эффектах записи в его параметры и чтения из них, мы можем преобразовать его в вычисление, ограничив область его побочных эффектов локальными переменными.

Если мы остановимся на мгновение и прислушаемся, то сможем разобрать, что тесты разговаривают с нами. Они говорят: «Смотрите, формирование отчета — это, по сути, вычисление: оно преобразует `List<String>` в `List<String>`. Мы знаем, что это так, потому что именно это мы и проверяем».

Тем самым тесты сообщают нам, что фундаментальной сигнатурой `generate` является `generate(lines: List<String>): List<String>`. Если бы *это* была сигнатура, то ей не пришлось бы объявлять, что она выдает `IOException`, потому что весь ввод/вывод будет происходить вне функции. Ввод/вывод должен где-то происходить, но, как и в случае с другими действиями, чем ближе мы можем переместить его к точкам входа в нашу систему, тем больше можем выполнять приятных простых вычислений.

Должны ли мы провести рефакторинг для достижения этой цели? Вы правы, это был риторический вопрос.

Ввод/вывод данных

В качестве первого этапа нашего рефакторинга давайте попробуем отключить параметр `generate` от параметра `reader`. Вот существующий код (пример 20.4).

Пример 20.4
[io-to-data.3:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
@Throws(IOException::class)
fun generate(reader: Reader, writer: Writer) {
    val valuableCustomers = reader
        .readLines()
        .toValuableCustomers()
        .sortedBy(CustomerData::score)
    writer.appendLine("ID\tName\tSpend")
    for (customerData in valuableCustomers) {
        writer.appendLine(customerData.outputLine)
    }
    writer.append(valuableCustomers.summarised())
}
```

Можно преобразовать `generate` для считывания из `List`, выполнив команду **Ввести параметр** (Introduce parameter) для выражения `reader.readLines()`. Назовите этот параметр `lines`. Поскольку выражение является единственным использованием существующего параметра `reader`, IntelliJ удалит для нас `reader` (пример 20.5).

Пример 20.5
[io-to-data.4:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
@Throws(IOException::class)
fun generate(writer: Writer, lines: List<String>) {
    val valuableCustomers = lines
        .toValuableCustomers()
        .sortedBy(CustomerData::score)
    writer.appendLine("ID\tName\tSpend")
    for (customerData in valuableCustomers) {
        writer.appendLine(customerData.outputLine)
    }
    writer.append(valuableCustomers.summarised())
}
```

Рефакторинг переместил `readLines()` к вызывающим абонентам. Перед вами результат в тесте (пример 20.6).

Пример 20.6
[io-to-data.4:src/test/java/travelator/marketing/HighValueCustomersReportTests.kt]

```
private fun check(
    inputLines: List<String>,
```

```
    expectedLines: List<String>
) {
    val output = StringWriter()
    val reader = StringReader(inputLines.joinToString("\n"))
    generate(output, reader.readLines())
    val outputLines = output.toString().lines()

    assertEquals(expectedLines, outputLines)
}
```

Теперь этот код кричит о том, о чем все это время шептал тест. Нам пришлось создать `StringReader` из списка строк только для того, чтобы разобрать строки обратно в `generate`.

Сейчас, когда они находятся в одном и том же месте теста, мы можем исключить их, чтобы удалить `Reader` (пример 20.7).

Пример 20.7
[io-to-data.5:src/test/java/travelator/marketing/HighValueCustomersReportTests.kt]

```
private fun check(
    inputLines: List<String>,
    expectedLines: List<String>
) {
    val output = StringWriter()
    generate(output, inputLines)
    val outputLines = output.toString().lines()
    assertEquals(expectedLines, outputLines)
}
```

Здесь мы считываем данные из `List`. Давайте вернемся назад и посмотрим, как вернуть также и `List` — вместо того чтобы изменять `Writer`. Перед вами код (пример 20.8).

Пример 20.8
[io-to-data.5:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
writer.appendLine("ID\tName\tSpend")
for (customerData in valuableCustomers) {
    writer.appendLine(customerData.outputLine)
}
writer.append(valuableCustomers.summarised())
```

Вместо того чтобы усиленно размышлять о том, как мы хотим изменить `Writer`, давайте подумаем о данных, которые мы хотим записать, и создадим их (пример 20.9).

Пример 20.9
[io-to-data.6:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
val resultLines = listOf("ID\tName\tSpend") +
    valuableCustomers.map(CustomerData::outputLine) +
    valuableCustomers.summarised()
```

Тогда мы можем записать это одним фрагментом в `writer` (пример 20.10).

Пример 20.10
[io-to-data.6:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
@Throws(IOException::class)
fun generate(writer: Writer, lines: List<String>) {
    val valuableCustomers = lines
        .toValuableCustomers()
        .sortedBy(CustomerData::score)
    val resultLines = listOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
    writer.append(resultLines.joinToString("\n"))
}
```

Наша функция теперь состоит из двух инструкций, составляющих вычисление, и конечного действия, принимающего результат вычисления. Если теперь применить к строкам вычисления опцию **Извлечь функцию** (Extract function), сделать их общедоступными и присвоить название `generate`, то мы получим следующее (пример 20.11).

Пример 20.11
[io-to-data.7:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
@Throws(IOException::class)
fun generate(writer: Writer, lines: List<String>) {
    val resultLines = generate(lines)
    writer.append(resultLines.joinToString("\n"))
}

fun generate(lines: List<String>): List<String> {
    val valuableCustomers = lines
        .toValuableCustomers()
        .sortedBy(CustomerData::score)
    val resultLines = listOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
    return resultLines
}
```

Замена встроенным выражением обеих встроенных `resultLines` даст следующий код (пример 20.12).

Пример 20.12
[io-to-data.8:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
@Throws(IOException::class)
fun generate(writer: Writer, lines: List<String>) {
    writer.append(generate(lines).joinToString("\n"))
}

fun generate(lines: List<String>): List<String> {
    val valuableCustomers = lines
        .toValuableCustomers()
        .sortedBy(CustomerData::score)
    return listOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
}
```

Нам потребуется и еще одна замена встроенным выражением — на этот раз из старой функции `generate`. Мы заменяем ей вызов в клиентском коде, оставляя ее в тесте (пример 20.13).

Пример 20.13
[io-to-data.9:src/test/java/travelator/marketing/HighValueCustomersReportTests.kt]

```
private fun check(
    inputLines: List<String>,
    expectedLines: List<String>
) {
    val output = StringWriter()
    output.append(generate(inputLines).joinToString("\n"))
    val outputLines = output.toString().lines()

    assertEquals(expectedLines, outputLines)
}
```

Проведенный нами рефакторинг переместил часть действия `generate` на уровень выше, оставив на ее месте приятные биты чистого вычисления. Другой способ взглянуть на это заключается в том, чтобы наш первоначальный `Writer` стал бы накопительным объектом, который мы заменили бы преобразованием, как показано в *главе 14*. В любом случае наши тесты на самом деле не хотели проверять действие, поэтому у них снова есть избыточный ввод/вывод, который мы можем упростить до той формы, к которой стремились (пример 20.14).

Пример 20.14
[io-to-data.10:src/test/java/travelator/marketing/HighValueCustomersReportTests.kt]

```
private fun check(
    inputLines: List<String>,
    expectedLines: List<String>
) {
    assertEquals(expectedLines, generate(inputLines))
}
```

Давайте подведем итоги для нашей новой функции `generate` (пример 20.15).

Пример 20.15
[io-to-data.11:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun generate(lines: List<String>): List<String> {
    val valuableCustomers = lines
        .toValuableCustomers()
        .sortedBy(CustomerData::score)
    return listOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
}

private fun List<String>.toValuableCustomers() = withoutHeader()
    .map(String::toCustomerData)
    .filter { it.score >= 10 }

private fun List<String>.withoutHeader() = drop(1)
```

Сейчас функция `generate` делает намного меньше, и неясно, стоит ли использовать функцию `toValuableCustomers()`. Взглянув на нее по-новому, мы увидим, что она работает на смешанных уровнях, конвертируя и фильтруя. Давайте попытаемся заменить ее встроенным выражением (пример 20.16).

Пример 20.16
[io-to-data.12:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun generate(lines: List<String>): List<String> {
    val valuableCustomers = lines
        .withoutHeader()
        .map(String::toCustomerData)
        .filter { it.score >= 10 }
        .sortedBy(CustomerData::score)
    return listOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
}
```

Так лучше. Локальная переменная `valuableCustomers` выполняет полезную работу, объясняя нам, что означает выражение, а операции со списком разъясняют существующую реализацию. Все это относится к тому случаю, когда однострочная функция (см. *главу 9*), вероятно, ухудшила бы ситуацию, поэтому мы оставим ее в двух частях. Мы также продолжим сопротивляться искушению сделать ее функцией-расширением `List<String>.toReport()` — по крайней мере, сейчас.

Рефакторинг для удобства чтения

Рефакторинг для удобства чтения часто происходит именно так. Мы можем извлечь функцию, чтобы сделать что-то более читаемым в контексте, но, когда контекст меняется, извлеченная нами функция усложняет ситуацию. Время — это тоже контекст. Иногда то, что, по нашему мнению, было достаточно выразительным, когда мы писали код, оказывается менее выразительным, когда мы перечитываем его позже, или теперь может быть улучшено, потому что мы усвоили новые средства выражения.

Эффективное написание

Мы очень довольны этим рефакторингом. Он упростил наши тесты и рабочий код, и мы перешли от смешивания ввода/вывода и логики к более простым вычислениям без побочных эффектов.

Какое-то время в производстве тоже все было в порядке, но с ослаблением ограничений на поездки из-за COVID-19 наш `Travelator` добился впечатляющего успеха, в котором мы все были уверены. Однако со временем милые люди из отдела маркетинга начали жаловаться на то, что формирование отчетов терпит неудачу с `OutOfMemoryError`, и попросили нас разобраться в этом.

Еще о стимулах к работе...

Помимо нехватки памяти, у нас, как мы помним, были две другие проблемы с ошибками в этом коде. Оба раза входной файл оказывался искаженным, но маркетологи сидят за соседней дверью, и они просто звонили нам, чтобы мы им помогли, когда это происходило. Получив помощь, они угощали нас пирожными, так что на тот момент у нас вряд ли был лучший стимул справляться с обработкой ошибок (в отличие от описанных в *главе 21*). Научившись быстро исправлять ошибку `OutOfMemoryError`, мы попробовали немало вкусностей...

До сих пор мы не надоедали вам подробностями, но есть метод `main`, вызывающий наш отчет. Он предназначен для вызова с перенаправлением оболочки, чтения из файла, передаваемого по каналу в качестве стандартного ввода, и записи в файл, собранный из стандартного вывода. Таким образом, нашему процессу нет надобности считывать имена файлов из командной строки (пример 20.17).

Пример 20.17
[io-to-data.0:src/main/java/travelator/marketing/HighValueCustomersMain.kt]

```
fun main() {
    InputStreamReader(System.`in`).use { reader ->
        OutputStreamWriter(System.out).use { writer ->
```

```
            generate(reader, writer)
        }
    }
}
```

Когда мы провели рефакторинг `generate` для работы с `List`, а не с `Reader` и `Writer`, IntelliJ автоматически обновил `main` для вывода (пример 20.18).

Пример 20.18
[lo-to-data.9:src/main/java/travelator/marketing/HighValueCustomersMain.kt]

```
fun main() {
    System.`in`.reader().use { reader ->
        System.out.writer().use { writer ->
            writer.append(
                generate(
                    reader.readLines()
                ).joinToString("\n")
            )
        }
    }
}
```

Ах, вот в чем наша проблема. Мы считываем весь ввод в память (`readLines()`), обрабатываем его, а затем создаем весь вывод снова в памяти (`joinToString()`), прежде чем записывать его обратно. Вот памяти и не хватает...

Иногда мы сталкиваемся с подобными проблемами при функциональной декомпозиции. В исходном коде `Reader` и `Writer` не было этой проблемы, так что мы сами навлекли ее на себя в стремлении к хорошему стилю. Мы могли бы быстро отменить наши изменения и пойти к маркетологам за оставшимися вкусностями, но есть возможность найти и более функциональное решение.

Давайте вернемся к `generate` и посмотрим, какая у нас есть свобода действий (пример 20.19).

Пример 20.19
[lo-to-data.12:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun generate(lines: List<String>): List<String> {
    val valuableCustomers = lines
        .withoutHeader()
        .map(String::toCustomerData)
        .filter { it.score >= 10 }
        .sortedBy(CustomerData::score)
    return listOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
}
```

Сосредоточившись сейчас на результатах, можно заметить, что мы строим `List` строк вывода. Затем `main` берет каждый элемент `String` из результата и создает один гигантский результат с помощью `joinToString()`. На этом этапе обе отдельные выходные строки и их объединение будут занимать память. Чтобы избежать нехватки памяти, нам нужно отложить создание промежуточных коллекций, и, как показано в *главе 13*, последовательности `Sequence` предназначены именно для этого.

Мы можем преобразовать `generate` для возврата `Sequence` методично или быстро. На этот раз мы выберем быстрый вариант и просто заменим `listOf` на `sequenceOf` в выражении `return` (пример 20.20).

Пример 20.20
[io-to-data.13:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun generate(lines: List<String>): Sequence<String> {
    val valuableCustomers = lines
        .withoutHeader()
        .map(String::toCustomerData)
        .filter { it.score >= 10 }
        .sortedBy(CustomerData::score)
    return sequenceOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
}
```

Теперь мы будем создавать выходные строки только по одной за раз, когда `Sequence` повторяется. От отдельной строки можно быстро избавиться, а не ждать, пока будет записан весь файл целиком.

Тесты должны измениться, чтобы преобразовать возвращенные `Sequence` в `List` (пример 20.21).

Пример 20.21
[io-to-data.13:src/test/java/travelator/marketing/HighValueCustomersReportTests.kt]

```
private fun check(
    inputLines: List<String>,
    expectedLines: List<String>
) {
    assertEquals(
        expectedLines,
        generate(inputLines).toList()
    )
}
```

Интересно, однако, что `main` не меняется (пример 20.22).

Пример 20.22
[io-to-data.13:src/main/java/travelator/marketing/HighValueCustomersMain.kt]

```
fun main() {
    System.`in`.reader().use { reader ->
        System.out.writer().use { writer ->
            writer.append(
                generate(
                    reader.readLines()
                ).joinToString("\n")
            )
        }
    }
}
```

Необходимо *перекомпилировать* `main`, потому что сейчас `generate` возвращает `Sequence`, а не `List`, но ее *источник* не должен быть изменен. Все потому что функции-расширения `joinToString()` определены для `Iterable` и `Sequence`, и обе возвращают `String`.

Возможно, `main` не *нужно* менять, но пока `main` не *изменится*, мы все равно создаем одну большую строку из всех выходных данных, прежде чем записать ее за одну операцию. Чтобы избежать этого, нам нужно снова получить императив и записать каждую выходную строку по отдельности, как было сделано в нашем исходном `generate` (пример 20.23).

Пример 20.23
[io-to-data.14:src/main/java/travelator/marketing/HighValueCustomersMain.kt]

```
fun main() {
    System.`in`.reader().use { reader ->
        System.out.writer().use { writer ->
            generate(
                reader.readLines()
            ).forEach { line ->
                writer.appendLine(line)
            }
        }
    }
}
```

Педантичный читатель (не волнуйтесь, вы среди друзей) заметит, что такое поведение немного отличается от версии `joinToString("\n")`. Но мы точно уверены, что завершающая новая строка ничего не нарушит, поэтому продолжаем.

Мы всегда можем притвориться, что не зацикливаемся, скрыв итерацию внутри функции-расширения `Writer::appendLines`, которую, как мы предполагали, определит стандартная библиотека Kotlin, но, похоже, это не так (пример 20.24).

Пример 20.24
[io-to-data.15:src/main/java/travelator/marketing/HighValueCustomersMain.kt]

```
fun main() {
    System.`in`.reader().use { reader ->
        System.out.writer().use { writer ->
            writer.appendLines(
                generate(reader.readLines())
            )
        }
    }
}

fun Writer.appendLines(lines: Sequence<CharSequence>): Writer {
    return this.also {
        lines.forEach(this::appendLine)
    }
}
```

Обратите внимание, что, хотя определение `Writer::appendLines` является однострочным выражением, в *главе 9* мы договорились использовать длинную форму, где функции — это действия, и `appendLines` — это определенно такой случай.

Получив этот результат, мы понимаем, что можно было бы отложить кризис памяти, просто перебирая исходный результат `List` в `main` и записывая каждую строку по отдельности, как мы сейчас делаем с `Sequence`. Однако наше решение задействует еще меньше памяти, поэтому мы зафиксируем его, получив за счет небольших изменений большой запас памяти и отработав наши вкусности. Есть еще что-нибудь с кремом?

Эффективное считывание

Было бы небрежностью с нашей стороны не закончить работу и сделать вид, что нам нужно экономить память и при чтении тоже. Давайте рассмотрим `generate` снова (пример 20.25).

Пример 20.25
[io-to-data.15:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun generate(lines: List<String>): Sequence<String> {
    val valuableCustomers = lines
        .withoutHeader()
        .map(String::toCustomerData)
        .filter { it.score >= 10 }
        .sortedBy(CustomerData::score)
```

```
    return sequenceOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
}
```

Конвейер операций, который строит `valuableCustomers`, будет создавать промежуточные `List` — по одному для каждого уровня, и каждый раз будет занимать память. Каждая строка во входных данных будет находиться в памяти одновременно вместе с объектом `CustomerData` для каждой строки.

Мы можем избежать промежуточных коллекций, считывая данные из `Sequence`, хотя это само по себе создаст несколько проблем. Можно увидеть их, если изменить код в `generate`, чтобы преобразовать `lines` в `Sequence` и исправить методы, которые принимали `List` (пример 20.26).

Пример 20.26
[io-to-data.16:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun generate(lines: List<String>): Sequence<String> {
    val valuableCustomers: Sequence<CustomerData> = lines
        .asSequence()
        .withoutHeader()
        .map(String::toCustomerData)
        .filter { it.score >= 10 }
        .sortedBy(CustomerData::score)
    return sequenceOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
}

private fun Sequence<String>.withoutHeader() = drop(1)

private fun Sequence<CustomerData>.summarised(): String =
    sumByDouble { it.spend }.let { total ->
        "\tTOTAL\t${total.toMoneyString()}"
    }
```

Этот код проходит модульные тесты. Мы закончили? Еще один риторический вопрос?

Давайте перейдем к сути и признаемся: проблема в том, что в итоге мы повторяем `valuableCustomers` дважды. Впервые *до* возврата из `generate` в `sumByDouble`, и снова *после* возврата, когда наши вызывающие абоненты повторяют возвращенную `Sequence` для вывода отчета.

Если мы дважды перебираем `Sequence`, то и проделываем всю работу по созданию `Sequence` дважды. В нашем случае два раза выполняем: удаление заголовка, маппинг, фильтрацию и сортировку. Хуже того, когда мы пытаемся использовать код

в работе, передавая Sequence для считывания стандартных входных данных, то не сможем повторить это дважды, поскольку будем получать IllegalStateException. В *главе 13* было показано, что экземпляры Sequence отличаются способами выражения в системе типов и также несут скрытое состояние. Повтор действия над Sequence выглядит как повтор над List, но это изменит саму Sequence, поглощая ее содержимое.

Мы можем показать, что злоупотребляем этой Sequence, добавив вызов .constrainOnce() (пример 20.27).

Пример 20.27
[io-to-data.17:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
val valuableCustomers: Sequence<CustomerData> = lines
    .asSequence()
    .constrainOnce()
    .withoutHeader()
    .map(String::toCustomerData)
    .filter { it.score >= 10 }
    .sortedBy(CustomerData::score)
```

Такое решение приведет к сбою наших тестов с IllegalStateException. Самое простое здесь — исправить Sequence при помощи вызова .toList() (пример 20.28).

Пример 20.28
[io-to-data.18:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
val valuableCustomers: List<CustomerData> = lines
    .asSequence()
    .constrainOnce()
    .withoutHeader()
    .map(String::toCustomerData)
    .filter { it.score >= 10 }
    .sortedBy(CustomerData::score)
    .toList()
```

Это завершает последовательность (следовательно, в итоге считывает весь файл) в приведенной инструкции, но, по крайней мере, мы запускаем конвейер только один раз, и память для каждой строки может быть очищена, как только строка будет проанализирована с помощью toCustomerData. На самом деле нам все равно придется прочитать весь ввод этой функции, потому что Sequence.sortedBy должна для выполнения сортировки прочитать каждый элемент — она может возвращать Sequence, но она не будет отложенной.

Теперь у нас есть возможность повторить рефакторинг **Ввести параметр**, который мы выполняли в начале этой главы. Тогда мы преобразовали параметр Reader в List. Сейчас же мы преобразуем List в Sequence. Параметр, который мы вводим, — это выражение lines.asSequence().constrainOnce() (пример 20.29).

Пример 20.29
[io-to-data.19:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun generate(lines: Sequence<String>): Sequence<String> {
    val valuableCustomers = lines
        .withoutHeader()
        .map(String::toCustomerData)
        .filter { it.score >= 10 }
        .sortedBy(CustomerData::score)
        .toList()
    return sequenceOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
}

private fun List<CustomerData>.summarised(): String =
    sumByDouble { it.spend }.let { total ->
        "\tTOTAL\t${total.toMoneyString()}"
    }
```

Рефакторинг приводит к преобразованию `List` в `Sequence` прямо в тестах (пример 20.30).

Пример 20.30
[io-to-data.19:src/test/java/travelator/marketing/HighValueCustomersReportTests.kt]

```
private fun check(
    inputLines: List<String>,
    expectedLines: List<String>
) {
    assertEquals(
        expectedLines,
        generate(
            inputLines.asSequence().constrainOnce()
        ).toList()
    )
}
```

Это также подтягивает ее в `main` (пример 20.31).

Пример 20.31
[io-to-data.19:src/main/java/travelator/marketing/HighValueCustomersMain.kt]

```
fun main() {
    System.`in`.reader().use { reader ->
        System.out.writer().use { writer ->
```

```
            writer.appendLines(
                generate(
                    reader.readLines().asSequence().constrainOnce()
                )
            )
        }
    }
}
```

Вот, где мы действительно можем сэкономить память. Вместо того чтобы читать все строки сразу и преобразовывать в `Sequence`, мы можем получить `Sequence` из `Reader` с помощью `buffered().lineSequence()` (пример 20.32).

Пример 20.32
[io-to-data.20:src/main/java/travelator/marketing/HighValueCustomersMain.kt]

```
fun main() {
    System.`in`.reader().use { reader ->
        System.out.writer().use { writer ->
            writer.appendLines(
                generate(
                    reader.buffered().lineSequence()
                )
            )
        }
    }
}
```

Теперь `generate` будет вытягивать строки в память одну за другой по мере выполнения своего конвейера. Сейчас мы действительно довольно эффективно используем память и работаем приятно быстро. Можем ли мы устоять перед небольшим последним исправлением? Насколько приятнее было бы читать `main` с большим количеством функций-расширений (пример 20.33)?

Пример 20.33
[io-to-data.21:src/main/java/travelator/marketing/HighValueCustomersMain.kt]

```
fun main() {
    System.`in`.reader().use { reader ->
        System.out.writer().use { writer ->
            reader
                .asLineSequence()
                .toHighValueCustomerReport()
                .writeTo(writer)
        }
    }
}
```

Это, наконец, отвечает на заданный в конце *главы 10* вопрос: да, в конечном итоге мы получаем формирование отчетов в качестве функции-расширения. Нам нравится, когда детали плана сводятся воедино (пример 20.34).

Пример 20.34
[io-to-data.21:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun Sequence<String>.toHighValueCustomerReport(): Sequence<String> {
    val valuableCustomers = this
        .withoutHeader()
        .map(String::toCustomerData)
        .filter { it.score >= 10 }
        .sortedBy(CustomerData::score)
        .toList()
    return sequenceOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
}
```

Двигаемся дальше

Рассмотренный здесь рефакторинг был продиктован желанием упростить наш код. Перемещение ввода/вывода к точке входа нашей программы поможет внутренним процессам стать скорее вычислениями, чем действиями. Они также могут отказаться от ответственности за ошибки ввода/вывода. Это было хорошо и правильно, но вычисления принимают и возвращают значения, а формирование значения всего содержимого больших файлов иногда слишком сложно даже для современных компьютеров.

Чтобы решить эту проблему, мы прибегли к преобразованию *списков* `List` в *последовательности* `Sequence`. Последовательности имеют состояние и не являются значениями, но мы с осторожностью можем обрабатывать их как отложенные значения. Отложенные в том смысле, что они не требуют или не возвращают все свое содержимое заранее, но могут считывать или предоставлять их по требованию. Они не так просты, как списки, но их совместимый API в Kotlin позволяет использовать лучшее из обоих миров.

Исходная версия `generate` — от `Reader` к `Writer` — зависела от ошибок ввода/вывода, в то время как версия от `List` к `List` переместила их все к их вызывающим абонентам. Версия `Sequence` находится в промежуточном положении. Она не зависит от ошибок ввода/вывода, потому что они скрыты при помощи абстракций `Sequence`, оборачивающих `Reader` и `Writer`. Это не значит, что ошибки не могут произойти, — просто `generate` не несет за них ответственности. Мы сделаем перерыв, чтобы проверить, не осталось ли у наших коллег по маркетингу еще каких-нибудь лакомств, прежде чем обратиться к этой теме в *главе 21 «От исключений к значениям»*.

ГЛАВА 21

От исключений к значениям

В главе 19 мы рассмотрели стратегии обработки ошибок в Kotlin и способы преобразования исключений в Java в более функциональные методы. Однако правда заключается в том, что большая часть кода игнорирует ошибки в надежде, что они не произойдут. Можем ли мы это так оставить?

Какие-то новички из отдела маркетинга взялись за отладку электронной таблицы, которую мы в последний раз видели в *главе 20*, — ту, что формирует рейтинг особо значимых клиентов. Мы точно не знаем, как там у них все это делается, но они продолжают экспортировать файлы, которые нарушают наш синтаксический анализ, а затем просят нас объяснить, что такое трассировка стека. В результате между нами и ими возникает некоторая неловкость, которую не могут сгладить даже предлагаемые нам сласти. Может ли быть еще какой-нибудь стимул?

Ну да, может. Они попросили нас написать программу, с помощью которой отдел маркетинга мог бы сохранять файлы на сервере и которая автоматически формировала бы сводные данные без участия человека в цикле, который интерпретирует трассировки стека. Что ж, похоже, нам придется найти способ правильно сообщать об ошибках.

Определение того, что может пойти не так

Вот код в том виде, в каком мы его оставили (пример 21.1).

Пример 21.1 [exceptions-to-values.0:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun Sequence<String>.toHighValueCustomerReport(): Sequence<String> {
    val valuableCustomers = this
        .withoutHeader()
        .map(String::toCustomerData)
        .filter { it.score >= 10 }
        .sortedBy(CustomerData::score)
        .toList()
    return sequenceOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
}
```

```
private fun List<CustomerData>.summarised(): String =
    sumByDouble { it.spend }.let { total ->
        "\tTOTAL\t${total.toMoneyString()}"
    }

private fun Sequence<String>.withoutHeader() = drop(1)

internal fun String.toCustomerData(): CustomerData =
    split("\t").let { parts ->
        CustomerData(
            id = parts[0],
            givenName = parts[1],
            familyName = parts[2],
            score = parts[3].toInt(),
            spend = if (parts.size == 4) 0.0 else parts[4].toDouble()
        )
    }

private val CustomerData.outputLine: String
    get() = "$id\t$marketingName\t${spend.toMoneyString()}"

private fun Double.toMoneyString() = this.formattedAs("%#.2f")

private fun Any?.formattedAs(format: String) = String.format(format, this)

private val CustomerData.marketingName: String
    get() = "${familyName.toUpperCase()}, $givenName"
```

Если требуется провести тщательную работу по обработке ошибок, то первое, что мы должны сделать, это установить, что может пойти не так. Как показано в *главе 19*, в Kotlin нет проверяемых исключений, которые могли бы дать нам подсказки, но эти исключения так плохо используются в большей части Java-кода, что в этом отношении между языками нет большой разницы. Если в коде не предусмотрена система сообщений о возможном сбое, нам придется полагаться на экспертизу, интуицию и опыт, чтобы разобраться в этом вопросе. Так, опыт подсказывает нам, что причиной происходящих сбоев в основном являются отсутствующие поля, поэтому можем сосредоточиться на этом аспекте кода, но все равно должны проявлять должную осмотрительность во всех прочих его аспектах. Давайте пройдемся по функциям снизу вверх, ища потенциальные ошибки.

`CustomerData.marketingName` выглядит вполне прилично (пример 21.2).

Пример 21.2 [exceptions-to-values.0:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
private val CustomerData.marketingName: String
    get() = "${familyName.toUpperCase()}, $givenName"
```

Если бы `CustomerData` была реализована на Java, мы могли бы найти `familyName`, сведя решение к `null` и, следовательно, выдав ошибку при попытке выполнить `toUpperCase()`. Но в Kotlin это невозможно, а значит, не произойдет. Как и во всем коде, функция зависит от подклассов `Error` (таких как `OutOfMemoryError`), но в целом она должна быть безопасна. С этого момента мы будем считать выдачу `Error` экстраординарным вариантом и не будем учитывать ее в нашем анализе.

Теперь посмотрим на `formattedAs` (пример 21.3).

Пример 21.3 [exceptions-to-values.0:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
private fun Any?.formattedAs(format: String) = String.format(format, this)
```

Функция `String.format(format, this)` реализована как `java.lang.String::format` и оформлена так, чтобы выдать `IllegalFormatException`, если `format` несовместим с другими входными данными. Это *частично применяемая функция* (**https://oreil.ly/ErpGo**) — та, которая возвращает результат только для некоторых из возможных значений ее параметров. Она может возвращать результат для всех значений `Double`, но только тогда, когда мы используем очень специфические значения `format`. К счастью, мы «кормим» ее только одним конкретным `format` — со значением `%#.2f`, которое, как мы знаем, работает. Так что это ее единственный абонент `Double.toMoneyString()`, который не должен завершиться ошибкой. И если код дает сбой, это происходит потому, что наш анализ неверен (или его предположения больше не соответствуют действительности), а ошибки времени выполнения — разумный способ сигнализировать об этой ошибке программиста.

Идем далее (пример 21.4).

Пример 21.4 [exceptions-to-values.0:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
private val CustomerData.outputLine: String
    get() = "$id\t$marketingName\t${spend.toMoneyString()}"
```

Это вызывает только тот код, который, как мы только что выяснили, не должен завершаться сбоем, поэтому благодаря транзитивному свойству сбоя он также должен быть безопасным.

Обратите внимание, что до сих пор было легко, потому что все эти функции являются вычислениями (см. *разд. «Вычисления» главы 7*). Они не зависят от какого-либо внешнего состояния, поэтому мы можем рассуждать о вычислениях, просто взглянув на них.

Пока все совсем неплохо, и глянем теперь на `String.toCustomerData()`(пример 21.5).

Пример 21.5 [exceptions-to-values.0:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
internal fun String.toCustomerData(): CustomerData =
    split("\t").let { parts ->
        CustomerData(
            id = parts[0],
            givenName = parts[1],
            familyName = parts[2],
            score = parts[3].toInt(),
            spend = if (parts.size == 4) 0.0 else parts[4].toDouble()
        )
    }
```

Хорошо, это еще одна частично применяемая функция — меньшая часть значений получателя `String` приведет к возвращению результата. К счастью, почти все получаемые на практике результаты корректны, поэтому обработка ошибок только сейчас становится приоритетом. Но что может пойти не так?

Начнем с верхней части функции: `String.split` может вести себя странно, если мы передадим ей пустой разделитель, но мы не станем этого делать. Тогда нам может не хватить частей (`parts`) — следовательно, `parts[n]` выдаст `IndexOutOfBoundsException`. Наконец, `parts[3]` может не быть `Int` или `parts[4]` может не быть `Double` — оба будут выдавать `NumberFormatException`.

Установив, что `toCustomerData` может завершиться неудачей, если будет передана `String`, которая не соответствует нашей спецификации формата, что мы должны с этим делать? Пока еще все варианты сбоя кода сводятся к возникновению исключения и прерыванию программы с недружественным сообщением об ошибке. И вызову нас в отдел маркетинга. Что приводит к двум следующим вопросам: «Должны ли мы прервать работу?» и «Как мы можем улучшить сообщение об ошибке, чтобы в отделе маркетинга могли его интерпретировать?».

Как показано в *главе 19*, мы не должны использовать исключения для прерывания при предсказуемых ошибках. Отсутствие проверяемых исключений в Kotlin (и отсутствие их применения в Java) означает, что если мы это сделаем, то потеряем возможность сообщить, что код подвержен сбоям. Затем абоненты, вызывающие код, должны делать то, что мы делаем в настоящее время, — анализировать каждую строку кода в реализации. Но даже после этого реализация может измениться, незаметно аннулируя полученные данные.

Если мы не должны выдавать исключение, то изменение с самыми небольшими затратами (при условии, что все наши абоненты относятся к Kotlin) — это вернуть `null` при сбое. Тогда клиентский код будет вынужден учитывать случай `null` и действовать соответствующим образом. Например, так (пример 21.6).

Пример 21.6 [exceptions-to-values.1:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
internal fun String.toCustomerData(): CustomerData? =
    split("\t").let { parts ->
        if (parts.size < 4)
            null
        else
            CustomerData(
                id = parts[0],
                givenName = parts[1],
                familyName = parts[2],
                score = parts[3].toInt(),
                spend = if (parts.size == 4) 0.0 else parts[4].toDouble()
            )
    }
```

Мы могли бы просто обернуть всю реализацию в блок `try` и вернуть `null` из `catch`, но здесь мы действовали скорее активно, чем реактивно. Это означает, что код все равно выдаст ошибку, если соответствующие поля не могут быть преобразованы в `Int` или `Double`. Мы еще вернемся к этому.

Произведенное изменение нарушает `toHighValueCustomerReport`, который теперь вынужден учитывать возможность сбоя (пример 21.7).

Пример 21.7 [exceptions-to-values.1:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun Sequence<String>.toHighValueCustomerReport(): Sequence<String> {
    val valuableCustomers = this
        .withoutHeader()
        .map(String::toCustomerData)
        .filter { it.score >= 10 } ❶
        .sortedBy(CustomerData::score)
        .toList()
    return sequenceOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
}
```

❶ Не компилируется, потому что `it` является обнуляемым.

Далее, если мы хотим просто игнорировать плохо сформированные входные строки, то можем снова запустить все с помощью `filterNotNull` (пример 21.8).

Пример 21.8 [exceptions-to-values.2:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun Sequence<String>.toHighValueCustomerReport(): Sequence<String> {
    val valuableCustomers = this
        .withoutHeader()
        .map(String::toCustomerData)
        .filterNotNull()
        .filter { it.score >= 10 }
        .sortedBy(CustomerData::score)
        .toList()
    return sequenceOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
}
```

У нас нет никаких тестов, подтверждающих это, и мы действительно должны написать некоторые из них, но пока будем действовать без подстраховки, потому что это исследовательское решение. Отсюда мы можем использовать `null` — чтобы представить другие варианты, в которых `toCustomerData` точно даст сбой (пример 21.9).

Пример 21.9 [exceptions-to-values.3:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
internal fun String.toCustomerData(): CustomerData? =
    split("\t").let { parts ->
        if (parts.size < 4)
            return null
        val score = parts[3].toIntOrNull() ?:
            return null
        val spend = if (parts.size == 4) 0.0 else parts[4].toDoubleOrNull() ?:
            return null
        CustomerData(
            id = parts[0],
            givenName = parts[1],
            familyName = parts[2],
            score = score,
            spend = spend
        )
    }
```

Обратите внимание, что стандартная библиотека Kotlin помогла нам, предоставив функции `String::toSome thingOrNull` только с этим соглашением об обработке ошибок. Теперь наш код представляет все разумные ошибки со значением `null`. Мы

можем вернуться к `toHighValueCustomerReport` и решить, что с ними делать, а не притворяться, что они еще не произошли (пишется `filterNotNull`).

Мы могли бы прервать работу при первой ошибке, но, похоже, стоит приложить дополнительные усилия, чтобы собрать все проблемные строки и каким-то образом сообщить о них. *Каким-то образом* — это немного расплывчато, но, как ни странно, у нас для этого есть тип — в нашем случае: `(String) -> Unit`. Другими словами, мы можем передать «что делать» функции, которая принимает ошибочную строку и не влияет на результат. Мы ссылаемся на эту технику в *разд. «Обработка ошибок до появления методов, основанных на исключениях» главы 19*. Чтобы проиллюстрировать это, давайте добавим тест (пример 21.10).

Пример 21.10 [exceptions-to-values.4:src/test/java/travelator/marketing/HighValueCustomersReportTests.kt]

```
@Test
fun `calls back on parsing error`() {
    val lines = listOf(
        "ID\tFirstName\tLastName\tScore\tSpend",
        "INVALID LINE",
        "1\tFred\tFlintstone\t11\t1000.00",
    )

    val errorCollector = mutableListOf<String>()
    val result = lines
        .asSequence()
        .constrainOnce()
        .toHighValueCustomerReport { badLine -> ❶
            errorCollector += badLine
        }
        .toList()

    assertEquals(
        listOf(
            "ID\tName\tSpend",
            "1\tFLINTSTONE, Fred\t1000.00",
            "\tTOTAL\t1000.00"
        ),
        result
    )
    assertEquals(
        listOf("INVALID LINE"),
        errorCollector
    )
}
```

❶ Эта лямбда реализует `onErrorLine` в следующем примере.

И реализуем его с помощью самого простого способа, который должен сработать (пример 21.11).

Пример 21.11 [exceptions-to-values.4:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun Sequence<String>.toHighValueCustomerReport(
    onErrorLine: (String) -> Unit = {}
): Sequence<String> {
    val valuableCustomers = this
        .withoutHeader()
        .map { line ->
            val customerData = line.toCustomerData()
            if (customerData == null)
                onErrorLine(line)
            customerData
        }
        .filterNotNull()
        .filter { it.score >= 10 }
        .sortedBy(CustomerData::score)
        .toList()
    return sequenceOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
}
```

Такой вариант все еще отфильтровывает строки ошибок, но только после передачи их абоненту `onErrorLine`, который может решить, что делать. В `main` мы воспользуемся им для вывода ошибок в `System.err`, а затем прервем выполнение (пример 21.12).

Пример 21.12 [exceptions-to-values.4:src/main/java/travelator/marketing/HighValueCustomersMain.kt]

```
fun main() {
    System.`in`.reader().use { reader ->
        System.out.writer().use { writer ->
            val errorLines = mutableListOf<String>()
            val reportLines = reader
                .asLineSequence()
                .toHighValueCustomerReport {
                    errorLines += it
                }
            if (errorLines.isNotEmpty()) {
                System.err.writer().use { error ->
                    error.appendLine("Lines with errors")
                    errorLines.asSequence().writeTo(error)
                }
```

```
                exitProcess(-1)
            } else {
                reportLines.writeTo(writer)
            }
        }
    }
}
```

Это одно из немногих мест в нашей книге, где мы вернулись к мутирующему `List`. Почему здесь? Например, мы могли бы изменить `toHighValueCustomerReport`, чтобы она возвращала `Pair<Sequence<String>, List<String>>`, где вторая из пар — это ошибки. Основное преимущество такой схемы заключается в том, что она позволяет вызывающему абоненту прервать выполнение на ранней стадии, выдав исключение в `onErrorLine`. Для максимальной гибкости мы могли бы даже создать стратегию обработки ошибок с сигнатурой `(String) -> CustomerData?`, чтобы абонент мог предоставить замену, позволяющую восстанавливаться от ошибок в любой отдельной строке.

В *главе 20* мы изо всех сил постарались преобразовать `toHighValueCustomerReport` из действия в вычисление. Затем мы немного смягчили требования к чистоте, выполнив чтение и запись `Sequence`. Здесь мы ввели функцию обработки ошибок, возвращающую `Unit`, — верный признак того, что мы ввели действие. При условии, что область действия ограничена обработкой ошибок, а любые побочные эффекты, как и в `main`, ограничены локальными переменными, это еще один разумный компромисс. Таково целесообразное решение для обработки ошибок, которое является гибким и хорошо взаимодействующим, но никак не чистым.

Представление ошибок

Теперь, когда мы сообщаем, *что* наш парсинг может завершиться сбоем (возвращая обнуляемый тип) и *где* произошел сбой (с обратным вызовом, передающим строку), можно ли дать более полную информацию: *почему* произошел сбой?

Возврат типа результата, а не типа с обнуляемым значением позволяет нам указать, какие существуют режимы сбоя, и предоставить подробную информацию, когда они происходят. Давайте изменим `String.toCustomerData()`, чтобы она возвращала `Result`, а не обнуляемое значение (пример 21.13).

Пример 21.13 [exceptions-to-values.5:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
internal fun String.toCustomerData(): Result<CustomerData, ParseFailure> =
    split("\t").let { parts ->
        if (parts.size < 4)
            return Failure(NotEnoughFieldsFailure(this))
        val score = parts[3].toIntOrNull() ?:
            return Failure(ScoreIsNotAnIntFailure(this))
```

```
        val spend = if (parts.size == 4) 0.0 else parts[4].toDoubleOrNull() ?:
            return Failure(SpendIsNotADoubleFailure(this))
        Success(
            CustomerData(
                id = parts[0],
                givenName = parts[1],
                familyName = parts[2],
                score = score,
                spend = spend
            )
        )
    }
```

Мы создаем запечатанный класс, как делали в *главе 19*, чтобы объяснить, почему парсинг не удался (пример 21.14).

Пример 21.14 [exceptions-to-values.5:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
sealed class ParseFailure(open val line: String)
data class NotEnoughFieldsFailure(override val line: String) :
    ParseFailure(line)
data class ScoreIsNotAnIntFailure(override val line: String) :
    ParseFailure(line)
data class SpendIsNotADoubleFailure(override val line: String) :
    ParseFailure(line)
```

Честно говоря, в такой ситуации это излишне (один класс данных содержит строку с ошибкой и строку с причиной), но мы демонстрируем мастерство в обработке ошибок. Можно исправить абонентов `toCustomerData`, вызвав `onErrorLine` с данными, хранящимися в `ParseFailure`, а затем выдавая `null` при появлении `Error`. Это проходит текущие тесты (пример 21.15).

Пример 21.15 [exceptions-to-values.5:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun Sequence<String>.toHighValueCustomerReport(
    onErrorLine: (String) -> Unit = {}
): Sequence<String> {
    val valuableCustomers = this
        .withoutHeader()
        .map { line ->
            line.toCustomerData().recover {
                onErrorLine(line)
                null
            }
        }
```

```
        .filterNotNull()
        .filter { it.score >= 10 }
        .sortedBy(CustomerData::score)
        .toList()
    return sequenceOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
}
```

Однако чего мы действительно хотим, так это выявить `ParseFailure`. Давайте сначала изменим тест, чтобы собирать `ParseFailure` вместо строк с ошибками (пример 21.16).

Пример 21.16 [exceptions-to-values.6:src/test/java/travelator/marketing/HighValueCustomersReportTests.kt]

```
val errorCollector = mutableListOf<ParseFailure>()
val result = lines
    .asSequence()
    .constrainOnce()
    .toHighValueCustomerReport { badLine ->
        errorCollector += badLine
    }
    .toList()
assertEquals(
    listOf(NotEnoughFieldsFailure("INVALID LINE")),
    errorCollector
)
```

Теперь мы можем изменить `onErrorLine` для получения данных сбоя (пример 21.17).

Пример 21.17 [exceptions-to-values.6:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun Sequence<String>.toHighValueCustomerReport(
    onErrorLine: (ParseFailure) -> Unit = {}
): Sequence<String> {
    val valuableCustomers = this
        .withoutHeader()
        .map { line ->
            line.toCustomerData().recover {
                onErrorLine(it)
                null
            }
        }
        .filterNotNull()
        .filter { it.score >= 10 }
```

```
        .sortedBy(CustomerData::score)
        .toList()
    return sequenceOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
}
```

Это позволяет `main` сообщить о причине и строке (пример 21.18).

Пример 21.18 [exceptions-to-values.6:src/main/java/travelator/marketing/HighValueCustomersMain.kt]

```
if (errorLines.isNotEmpty()) {
    System.err.writer().use { error ->
        error.appendLine("Lines with errors")
        errorLines.asSequence().map { parseFailure ->
            "${parseFailure::class.simpleName} in ${parseFailure.line}"
        }.writeTo(error)
    }
    exitProcess(-1)
} else {
    reportLines.writeTo(writer)
}
```

Возможно, мы не так, как положено, использовали тип `ParseFailure` среды выполнения для обработки ошибок, но задействовали его имя в сообщении об ошибке. Так что мы, по крайней мере, получаем какое-то значение из нашей маленькой запечатанной иерархии классов. Если выдаваемых сообщений об ошибках недостаточно, чтобы помочь отделу маркетинга исправить ввод данных, мы можем применить выражение `when` в запечатанном классе, чтобы различать типы сбоев, как показано в *разд. «Слои» главы 19*.

Сейчас все компилируется, и наши тесты проходят — т. е. в этой маленькой части мира пока все хорошо. Если бы у нас было больше клиентского кода, вызывающего этот API, или наши изменения должны были проходить через большее количество слоев кода, мы могли бы выбрать более сложную стратегию рефакторинга, чем изменение кода в одном файле и исправление неполадок. Однако обычно это не стоит затраченных усилий — ведь можно получить компиляцию кода и прохождение тестов максимум за пару минут. Если мы поймем, что откусили больше, чем можем проглотить, легко вернуться и применить более взвешенный подход.

Теперь, когда тесты пройдены, мы должны вернуться и убедиться, что все настолько аккуратно и выразительно, насколько это возможно. В частности, мы сделали самое быстрое, что могли, чтобы все снова заработало в `toHighValueCustomerReport` (пример 21.19).

Пример 21.19 [exceptions-to-values.6:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
fun Sequence<String>.toHighValueCustomerReport(
    onErrorLine: (ParseFailure) -> Unit = {}
): Sequence<String> {
    val valuableCustomers = this
        .withoutHeader()
        .map { line ->
            line.toCustomerData().recover {
                onErrorLine(it)
                null
            }
        }
        .filterNotNull()
        .filter { it.score >= 10 }
        .sortedBy(CustomerData::score)
        .toList()
    return sequenceOf("ID\tName\tSpend") +
        valuableCustomers.map(CustomerData::outputLine) +
        valuableCustomers.summarised()
}
```

Есть что-то малоприятное в получении значений `null` из блока `recover` и пропуске их с помощью `filterNotNull`. Такой вариант не сообщает напрямую, как он работает, и мешает движению по «счастливому пути». Мы хотели бы иметь возможность найти более приятную формулировку выражения `valuableCustomers`, но правда в том, что все остальное в глазах ваших авторов еще хуже. Если вы все-таки найдете хороший простой способ, пожалуйста, дайте нам знать.

Аналогичным образом ранние возвращения в `toCustomerData` выглядят слегка некрасиво (пример 21.20).

Пример 21.20 [exceptions-to-values.6:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
internal fun String.toCustomerData(): Result<CustomerData, ParseFailure> =
    split("\t").let { parts ->
        if (parts.size < 4)
            return Failure(NotEnoughFieldsFailure(this))
        val score = parts[3].toIntOrNull() ?:
            return Failure(ScoreIsNotAnIntFailure(this))
        val spend = if (parts.size == 4) 0.0 else parts[4].toDoubleOrNull() ?:
            return Failure(SpendIsNotADoubleFailure(this))
        Success(
            CustomerData(
                id = parts[0],
```

```
                givenName = parts[1],
                familyName = parts[2],
                score = score,
                spend = spend
            )
        )
    }
```

«Правильная» функциональная обработка ошибок не применяла бы ранний возврат, но использовала бы цепочку `flatMap`. Нервные читатели, возможно, пожелают отвести взгляд (пример 21.21).

Пример 21.21 [exceptions-to-values.7:src/main/java/travelator/marketing/HighValueCustomersReport.kt]

```
internal fun String.toCustomerData(): Result<CustomerData, ParseFailure> =
    split("\t").let { parts ->
        parts
            .takeUnless { it.size < 4 }
            .asResultOr { NotEnoughFieldsFailure(this) }
            .flatMap { parts ->
                parts[3].toIntOrNull()
                    .asResultOr { ScoreIsNotAnIntFailure(this) }
                    .flatMap { score: Int ->
                        (if (parts.size == 4) 0.0
                        else parts[4].toDoubleOrNull())
                            .asResultOr { SpendIsNotADoubleFailure(this) }
                            .flatMap { spend ->
                                Success(
                                    CustomerData(
                                        id = parts[0],
                                        givenName = parts[1],
                                        familyName = parts[2],
                                        score = score,
                                        spend = spend
                                    )
                                )
                            }
                    }
            }
    }
```

Вашим авторам нравится однострочное выражение даже больше, чем большинству людей, но не в том случае, если это `Result`. Очевидно, мы могли бы упростить наш фрагмент, введя больше функций (например, `asResultOr ... flatMap` выглядит как понятие, пытающееся вырваться наружу). Некоторые другие библиотеки результа-

тов позволили бы нам злоупотреблять сопрограммами или исключениями, чтобы получить тот же эффект, что и предыдущие ранние возвраты, но без лучшей языковой поддержки, позволяющей избежать отступа для каждой инструкции, — сущность Kotlin благоприятствует в этих случаях ранним возвратам. Мы не рассматривали такой пример в этой книге, но тот факт, что лямбды могут быть скомпилированы заменой встроенными выражениями и поэтому поддерживают возврат из их заключающей функции, побуждает нас в подобных ситуациях использовать императивный код. Тогда для нас подойдет раннее возвращение.

Наконец, возвращаясь к `main` во время нашей последней проверки перед регистрацией (пример 21.22).

Пример 21.22 [exceptions-to-values.6:src/main/java/travelator/marketing/HighValueCustomersMain.kt]

```
fun main() {
    System.`in`.reader().use { reader ->
        System.out.writer().use { writer ->
            val errorLines = mutableListOf<ParseFailure>()
            val reportLines = reader
                .asLineSequence()
                .toHighValueCustomerReport {
                    errorLines += it
                }
            if (errorLines.isNotEmpty()) {
                System.err.writer().use { error ->
                    error.appendLine("Lines with errors")
                    errorLines.asSequence().map { parseFailure ->
                        "${parseFailure::class.simpleName} in ${parseFailure.line}"
                    }.writeTo(error)
                }
                exitProcess(-1)
            } else {
                reportLines.writeTo(writer)
            }
        }
    }
}
```

Три уровня вложенных `use` запутывают нынешнюю структуру, и `exitProcess` из глубоких недр функции тоже немного сомнительна. Мы можем определить наши собственные перегрузки `using`, чтобы решить первую проблему и передать выходящий код для решения второй (пример использования данных, а не потока управления для устранения ошибок). Мы также можем извлечь функцию-расширение для вывода ошибок (пример 21.23).

Пример 21.23 [exceptions-to-values.8:src/main/java/travelator/marketing/HighValueCustomersMain.kt]

```
fun main() {
    val statusCode = using(
        System.`in`.reader(),
        System.out.writer(),
        System.err.writer()
    ) { reader, writer, error ->
        val errorLines = mutableListOf<ParseFailure>()
        val reportLines = reader
            .asLineSequence()
            .toHighValueCustomerReport {
                errorLines += it
            }
        if (errorLines.isEmpty()) {
            reportLines.writeTo(writer)
            0
        } else {
            errorLines.writeTo(error)
            -1
        }
    }
    exitProcess(statusCode)
}

inline fun <A : Closeable, B : Closeable, C : Closeable, R> using(
    a: A,
    b: B,
    c: C,
    block: (A, B, C) -> R
): R =
    a.use {
        b.use {
            c.use {
                block(a, b, c)
            }
        }
    }

private fun List<ParseFailure>.writeTo(error: OutputStreamWriter) {
    error.appendLine("Lines with errors")
    asSequence().map { parseFailure ->
        "${parseFailure::class.simpleName} in ${parseFailure.line}"
    }.writeTo(error)
}
```

А как насчет ввода/вывода?

Сделанного к этому моменту почти достаточно. Однако прежде чем мы пойдем дальше, нам следует подумать об ошибках ввода/вывода. Поскольку мы ввели `List`, а затем `Sequence`, нашему коду формирования отчетов не нужно беспокоиться о сбое записи, потому что вызывающий код отвечает за перебор строк результатов и фактическое ее выполнение. Функция `main` в этом случае делает разумное предположение, что `System.out` всегда будет там. Но когда мы реализуем автоматический процесс, который мотивировал этот рефакторинг, нам придется иметь дело с возможностью того, что файл или сетевой сокет могут исчезнуть, даже если они были открыты, когда мы начали процедуру.

Существует аналогичная ситуация со считыванием. Сейчас мы перебираем каждую `String` в `Sequence`. В тестовом коде они находятся в памяти, но в рабочей среде они извлекаются из файла (с помощью `System.in`). Таким образом, наши операции `Sequence` подвержены сбою из-за `IOExceptions`, о которых формирование отчетов блаженно не знает.

В этих случаях `toHighValueCustomerReport()` мало что может или должна выполнять. Практического способа восстановления от ошибок ввода/вывода после того, как мы начали здесь считывать, не существует — разумно прервать всю операцию целиком. К счастью, теперь ответственность полностью лежит на вызывающем абоненте (в нашем случае `main`). `toHighValueCustomerReport` сообщает об известных ей ошибках (сбой парсинга) и о том, как они представлены (подклассы `ParseFailure`) с помощью параметра `onErrorLine`. `IOException` находятся за пределами ее зоны ответственности. Функция `main` передает `Sequence`, поддерживающую ввод/вывод, в `toHighValueCustomerReport`. Следовательно `main` должна знать, что `toHighValueCustomerReport` может дать сбой с `IOException`, и соответствующим образом с ним справиться. Давайте добавим следующий код (пример 21.24).

Пример 21.24 [exceptions-to-values.9:src/main/java/travelator/marketing/HighValueCustomersMain.kt]

```
fun main() {
    val statusCode = try {
        using(
            System.`in`.reader(),
            System.out.writer(),
            System.err.writer()
        ) { reader, writer, error ->
            val errorLines = mutableListOf<ParseFailure>()
            val reportLines = reader
                .asLineSequence()
                .toHighValueCustomerReport {
                    errorLines += it
                }
            if (errorLines.isEmpty()) {
                reportLines.writeTo(writer)
                0
```

```
            } else {
                errorLines.writeTo(error)
                -1
            }
        }
    } catch (x: IOException) {
        System.err.println("IO error processing report ${x.message}")
        -1
    }
    exitProcess(statusCode)
}
```

Возможно, этот код излишен для нашего приложения, но он показывает схему перехвата и обработки ожидаемых нами исключений (вывод относительно дружественного сообщения для `IOException`), позволяя всем остальным просочиться и выйти из приложения. Если следовать стратегии из *главы 19*, то неожиданные исключения — это либо неустранимые ошибки среды, либо ошибки программиста. В обоих случаях поведение JVM по умолчанию, заключающееся в завершении процесса после вывода трассировки стека, дает нам хороший шанс диагностировать проблему. Когда мы преобразуем это в автоматическое задание сервера, то аналогичным образом обработаем ожидаемые ошибки в нашей функции обработчика верхнего уровня. Мы могли бы прерваться на `IOException` или повторить все взаимодействие, если посчитали бы, что проблема может быть временной. Повторная попытка не поможет с ошибками парсинга, поэтому придется регистрировать их и/или отправлять уведомления куда-нибудь. Неожиданным ошибкам в функциях обработчика обычно разрешается просачиваться в обобщенный код обработки исключений, который будет регистрировать их и отправлять статус ошибки внутреннего сервера, прежде чем возвращать поток в его пул.

Двигаемся дальше

Очень часто в инженерном деле нам приходится идти на компромиссы. В частности, попытки упростить один процесс часто усложняют другой. Ввод/вывод усложняет наше программное обеспечение в двух моментах. Это действие, поэтому мы не можем просто игнорировать его, когда оно произойдет или произойдет ли оно вообще, пока мы осуществляем рефакторинг. И оно подвержено ошибкам, с которыми приходится иметь дело, если мы хотим создать надежную систему. Ошибки могут быть простыми сбоями при чтении или записи из-за окружающей среды или из-за того, что считываемые данные не соответствуют нашим ожиданиям, — например, когда маркетинговый файл неправильно отформатирован.

Как действия, так и ошибки портят вызывающих их абонентов, и решение в обоих случаях одно и то же: переместите код ближе к точкам входа, чтобы он меньше загрязнял вашу систему. Таким образом, это та область, где вместо того, чтобы идти на компромисс, мы можем убить двух зайцев одним выстрелом. Перемещая ввод/вывод за пределы наших систем, мы можем уменьшить количество вариантов усложнения нашего кода как действиями, так и ошибками.

ГЛАВА 22

От классов к функциям

Объектно-ориентированные программисты умеют решать проблемы путем создания типов. Функциональные программисты склонны дополнять существующие типы функциями. Как далеко мы можем зайти, не определяя новые типы?

В *главе 15 «От инкапсулированных коллекций к псевдонимам типов»* мы рассмотрели преимущества работы с необработанными коллекциями, а в *главе 16 «От интерфейсов к функциям»* — возможность использования встроенных типов функций вместо создания новых. В этой главе мы применим полученные знания, чтобы написать немного кода Kotlin с нуля.

Даже в дни REST API и webhooks большая часть автоматического делового общения осуществляется в форме табличных текстовых данных, обмен которыми происходит по протоколу безопасной передачи файлов (SFTP, Secure File Transfer Protocol). Наш `Travelator` при этом должен уметь импортировать данные о местоположении кемпинга, достопримечательностях, неоплаченных счетах и т. д., которые предоставляются ему в обычных строках и столбцах, с разными разделителями столбцов, а также с заголовками столбцов и без них. В *главе 20* шла речь о созданном одной из команд своем собственном анализаторе (парсере). В других главах мы используем проверенную и надежную библиотеку Apache Commons CSV library (**https:// oreil.ly/jnI4h**). Честно говоря, в большинстве случаев мы бы все равно использовали Commons CSV, потому что она работает прямо из коробки, легко настраивается для особых случаев и очень хорошо совместима с Kotlin.

Однако сегодня мы увидим, как будет выглядеть чистый парсер Kotlin. А закончив, сравним то, что придумали, с функциональностью Commons CSV, чтобы разобраться, как сущности Java и Kotlin приводят к различным API и реализациям.

Приемочный тест

Как вы, возможно, поняли из чтения предыдущих глав, разработчики `Travelator` — экстремальные программисты (поясним: экстремальное программирование — это разработка через тестирование). То есть сначала мы пишем тест кода, начиная с высокоуровневого приемочного теста. Поскольку мы работаем над программой чтения таблиц, то с помощью метода заглушек создаем класс `TableReaderAcceptanceTests` и проверяем, что он выполняется (пример 22.1).

Пример 22.1
[table-reader.1:src/test/java/travelator/tablereader/TableReaderAcceptanceTests.kt]

```
class TableReaderAcceptanceTests {
    @Test
    fun test() {
    }
}
```

Он действительно запускается (и даже проходит!). Так что теперь мы можем приступить к правильному кодированию.

Часть работы приемочного теста состоит в том, чтобы помочь нам решить, как должен выглядеть наш интерфейс. Проанализировав в свое время несколько файлов, мы узнали то, что почти всегда хотим сделать: прочитать файл и вернуть список значений некоторого доменного типа, по одному для каждой строки (без заголовка). Давайте набросаем такой тест с `Measurement` в качестве доменного типа (пример 22.2).

Пример 22.2
[table-reader.2:src/test/java/travelator/tablereader/TableReaderAcceptanceTests.kt]

```
class TableReaderAcceptanceTests {
    data class Measurement(
        val t: Double,
        val x: Double,
        val y: Double,
    )

    @Test
    fun `acceptance test`() {
        val input = listOf(
            "time,x,y",
            "0.0, 1, 1",
            "0.1,1.1,1.2",
            "0.2,1.2,1.4",
        )
        val expected = listOf(
            Measurement(0.0, 1.0, 1.0),
            Measurement(0.1, 1.1, 1.2),
            Measurement(0.2, 1.2, 1.4)
        )
        assertEquals(
            expected,
            someFunction(input)
        )
    }
```

```
    private fun someFunction(input: List<String>): List<Measurement> {
        TODO("Not yet implemented")
    }
}
```

Здесь `Measurement` является типом значения, представляющим данные, которые мы хотим извлечь из каждой строки таблицы. В Java мы, вероятно, начали бы с создания класса `TableReader`, но из теста видно, что чтение таблицы является просто вычислением — *маппингом* (сопоставлением) входных строк со списком нужных нам данных (см. разд *«Вычисления» главы 7*). Следовательно, мы станем по умолчанию использовать функцию верхнего уровня `someFunction` до тех пор, пока не будем вынуждены сделать что-то более сложное.

Мы можем представить себе всевозможные волшебные способы, которыми API мог бы реализовать `someFunction`, но если у него нет каких-либо специальных знаний о типе `Measurement` (и в библиотеках нет информации о *наших* типах), то нам нужно будет объяснить ему, как сопоставить некоторое представление строки с `Measurement`.

Мы здесь применили термин *маппинг*. Может быть, в этом решение? Что, если бы `someFunction` выглядела так (пример 22.3)?

Пример 22.3
[table-reader.3:src/test/java/travelator/tablereader/TableReaderAcceptanceTests.kt]

```
private fun someFunction(input: List<String>): List<Measurement> =
    readTable(input) ❶
        .map { record -> ❷ ❸
        Measurement(
            record["time"].toDouble(), ❹
            record["x"].toDouble(), ❹
            record["y"].toDouble(), ❹
        )
    }
```

❶ `readTable` является точкой входа нашего API для чтения таблиц.

❷ Она возвращает что-то, содержащее реализацию `map`.

❸ `record` является нашим представлением строки в таблице.

❹ Мы можем индексировать в `record` по имени поля, получая `String`, которую можем преобразовать в другие типы.

Код не компилируется, потому что у нас еще нет `readTable`, но если нажать на ошибке комбинацию клавиш <Alt>+<Enter>, IntelliJ создаст для нас функцию (пример 22.4).

Пример 22.4
[table-reader.3:src/test/java/travelator/tablereader/TableReaderAcceptanceTests.kt]

```
private fun readTable(input: List<String>): Any {
    TODO("Not yet implemented")
}
```

Мы не дали IntelliJ достаточно подсказок о типе возвращаемого `readTable`, поэтому она выбирает `Any` — следовательно, `someFunction` все еще не может быть скомпилирована. К какому типу мы могли бы вернуться, чтобы исправить это? Что ж, если мы вернем `List` из `readTable`, тогда `map` станет операцией для `List`. И если `List` содержит `Map<String, String>`, переменная `record` примет значение `Map<String, String>`, а значит, мы можем вызвать `record["time"]` и т. д.

Единственная проблема заключается в том, что `Map.get` возвращает обнуляемое значение. Мы уже достаточно близко — давайте примем это во внимание в `someFunction`, вызывая ошибки, если `get` возвращает `null` (пример 22.5).

Пример 22.5
[table-reader.4:src/test/java/travelator/tablereader/TableReaderAcceptanceTests.kt]

```
private fun someFunction(input: List<String>): List<Measurement> =
    readTable(input).map { record ->
        Measurement(
            record["time"]?.toDoubleOrNull() ?: error("in time"),
            record["x"]?.toDoubleOrNull() ?: error("in x"),
            record["y"]?.toDoubleOrNull() ?: error("in y"),
        )
    }

fun readTable(input: List<String>): List<Map<String, String>> {
    TODO("Not yet implemented")
}
```

Это компилируется, хотя очевидно, что `TODO` не проходит тест. (Вы можете спросить, почему мы здесь так бесцеремонно обращаемся с ошибками — в отличие подходов, представленных в *главе 21*. Ответ заключается в том, что это всего лишь тестовый код — API `Map.get` заставляет нас задуматься о том, что делать в случае ошибок, и наш тест выбирает сброс.)

Мы «примерили шляпу» наших клиентов, чтобы написать приемочные тесты. Эти тесты показали, что можно как минимум использовать функцию с сигнатурой `readTable` для преобразования строки в список `Measurement`. Теперь, когда у нас есть правдоподобный API, мы можем так переместить определение `readTable` (пример 22.6).

Пример 22.6 [table-reader.5:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTable(input: List<String>): List<Map<String, String>> {
    TODO("Not yet implemented")
}
```

Наконец, на этом первом этапе мы можем применить встроенное выражение `someFunction`, чтобы провести наш приемочный тест (пример 22.7).

Пример 22.7 [table-reader.5:src/test/java/travelator/tablereader/TableReaderAcceptanceTests.kt]

```
@Disabled
@Test
fun `acceptance test`() {
    val input = listOf(
        "time,x,y",
        "0.0, 1, 1",
        "0.1,1.1,1.2",
        "0.2,1.2,1.4",
    )
    val expected = listOf(
        Measurement(0.0, 1.0, 1.0),
        Measurement(0.1, 1.1, 1.2),
        Measurement(0.2, 1.2, 1.4)
    )
    assertEquals(
        expected,
        readTable(input).map { record ->
            Measurement(
                t = record["time"]?.toDoubleOrNull() ?: error("in time"),
                x = record["x"]?.toDoubleOrNull() ?: error("in x"),
                y = record["y"]?.toDoubleOrNull() ?: error("in y"),
            )
        }
    )
}
```

Обратите внимание, что мы отключили тест, потому что должно пройти некоторое время до того, как нам предстоит его запустить. Это нормально для приемочных тестов. Мы не ожидаем, что они пройдут быстро, поэтому сообщите нам об их окончании. К этому моменту наш код выполнил свою работу, помогая нам набросать простой API, который теперь можно реализовать.

Прежде чем мы продолжим, давайте поразмышляем над тем фактом, что нам удалось определить интерфейс для нашего анализатора без определения каких-либо

новых типов, задействовав вместо этого `List` и `Map` из `String`. Используя стандартные типы, мы опираемся на уверенность, что у нас есть удобные API-интерфейсы Kotlin для предоставления списка (`List`), из которого мы считываем, и для интерпретации `List` из `Map`, которые мы возвращаем.

Модульное тестирование

Теперь, когда у нас есть интерфейс для реализации, мы можем оставить приемочный тест и написать минимальный модульный тест. Что значит минимальный? Нам нравится начинать с пустоты — что должно произойти, если мы считаем пустой файл (пример 22.8)?

Пример 22.8 [table-reader.6:src/test/java/travelator/tablereader/TableReaderTests.kt]

```
class TableReaderTests {
    @Test
    fun `empty list returns empty list`() {
        val input: List<String> = emptyList()
        val expectedResult: List<Map<String, String>> = emptyList()
        assertEquals(
            expectedResult,
            readTable(input)
        )
    }
}
```

Самый простой способ заставить этот тест пройти — жестко запрограммировать результат в `readTable` (пример 22.9).

Пример 22.9 [table-reader.7:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTable(input: List<String>): List<Map<String, String>> {
    return emptyList()
}
```

В таком случае тест проходит. Этот вариант может показаться тривиальным, но всегда полезно провести тест на пустой ввод. Чем сложнее наш алгоритм, тем больше вероятность того, что он потерпит в таком случае неудачу. Однако синтаксический анализатор, который всегда возвращает пустой результат, — это плохой анализатор, так что давайте продолжим. Следуя подходу «Разработка через тестирование на примере» (Test-Driven Development By Example, TDD), сначала нужно добавить неудачный тест, чтобы дать повод изменить реализацию. Мы решили добавить случай чтения таблицы без заголовка и строк данных.

Почему так, а не заголовок и одна строка данных? Честно говоря, это просто первое, что пришло в голову. Можно было бы, конечно, задействовать строку заголов-

ка. Но наш выбор заставляет нас решить, как назвать столбцы, и мы принимаем решение использовать представление `String` индекса `"0"` для первого столбца, `"1"` — для второго и т. д. Это, похоже, самый простой способ создания ключа `String` (пример 22.10).

Пример 22.10 [table-reader.8:src/test/java/travelator/tablereader/TableReaderTests.kt]

```
@Test
fun `empty list returns empty list`() {
    assertEquals(
        emptyList<Map<String, String>>(),
        readTable(emptyList())
    )
}

@Test
fun `one line of input with default field names`() {
    assertEquals(
        listOf(
            mapOf("0" to "field0", "1" to "field1")
        ),
        readTable(listOf(
            "field0,field1"
        ))
    )
}
```

Вместо этого мы *могли бы* сделать так, чтобы `readTable` возвращала `<Map<Int, String>>` , когда строка заголовка отсутствует. Если у вас есть немного свободного времени, стоит пойти по этому пути, чтобы посмотреть, куда он ведет.

Вернемся к текущему затруднительному положению. У нас есть неудачный тест, и мы можем действовать умно, а можем — быстро. Мы выбираем быстрый вариант, чтобы сразу пройти тест, снова жестко запрограммировав результат (пример 22.11).

Пример 22.11 [table-reader.8:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTable(lines: List<String>): List<Map<String, String>> {
    return if (lines.isEmpty())
        emptyList()
    else listOf(
        mapOf("0" to "field0", "1" to "field1")
    )
}
```

Теперь, когда наши тесты успешно проходят, мы можем упростить реализацию, заметив, что нам нужна строка в выходных данных для каждой строки во входных

данных. Это можно реализовать при помощи `Iterable::map`, что позволит нам удалить выражение `if` (пример 22.12).

Пример 22.12 [single-expressions.0:src/main/java/travelator/EmailAddress.kt]

```
fun readTable(lines: List<String>): List<Map<String, String>> {
    return lines.map {
        mapOf("0" to "field0", "1" to "field1")
    }
}
```

Код продолжает проходить тесты и теперь будет работать для большего количества строк (идентичных данных)! Однако это всего лишь ступенька, позволяющая нам извлечь лямбду как функцию (пример 22.13).

Пример 22.13 [table-reader.10:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTable(lines: List<String>): List<Map<String, String>> {
    return lines.map(::parseLine)
}

private fun parseLine(line: String) = mapOf("0" to "field0", "1" to "field1")
```

Далее мы начнем удалять жестко закодированные значения, разбивая пары на `keys` и `values` (пример 22.14).

Пример 22.14 [table-reader.11:src/main/java/travelator/tablereader/table-reading.kt]

```
private fun parseLine(line: String): Map<String, String> {
    val keys = listOf("0", "1")
    val values = listOf("field0", "field1")
    return keys.zip(values).toMap()
}
```

Мы все еще решительно жульничаем, но теперь можем видеть закономерность в `keys` и формировать их из `values` (пример 22.15).

Пример 22.15 [table-reader.12:src/main/java/travelator/tablereader/table-reading.kt]

```
private fun parseLine(line: String): Map<String, String> {
    val values = listOf("field0", "field1")
    val keys = values.indices.map(Int::toString)
    return keys.zip(values).toMap()
}
```

Для `values` мы можем разделить строку запятыми (пример 22.16).

Пример 22.16 [table-reader.13:src/main/java/travelator/tablereader/table-reading.kt]

```
private fun parseLine(line: String): Map<String, String> {
    val values = line.split(",")
    val keys = values.indices.map(Int::toString)
    return keys.zip(values).toMap()
}
```

Успех! — мы удалили жестко закодированные ключи и значения, а тесты все еще проходят. Поскольку мы использовали `lines.map` в `readTable`, то считаем, что функция будет работать для любого количества строк, но было бы неплохо провести тест, чтобы подтвердить это.

Помечаем для себя, что нужно потом добавить такую возможность, но сейчас мы хотели бы сначала взглянуть на кое-что, вызывающее беспокойство. Если вам столько же лет, сколько вашим авторам (или вы моложе и лучше одарены), и у вас, возможно, развилось в отношении кода шестое чувство, то оно может взывать к вам при взгляде на `split`. Что произойдет, если мы попытаемся разделить пустую строку? Если уж на то пошло, что должно возвращать `readTable`, когда подается пустая строка?

Размышляя над этим, мы приходим к выводу, что пустая строка должна давать пустую `Map`. Это кажется понятным, поэтому мы пишем тест, чтобы документировать наше решение и проверить, работает ли оно (пример 22.17).

Пример 22.17 [table-reader.14:src/test/java/travelator/tablereader/TableReaderTests.kt]

```
@Test
fun `empty line returns empty map`() {
    assertEquals(
        listOf(
            emptyMap<String, String>()
        ),
        readTable(listOf(
            ""
        ))
    )
}
```

Ага!

```
org.opentest4j.AssertionFailedError:
Expected :[{}]
Actual :[{0=}]
```

После небольшого исследования мы обнаруживаем, что вызов `split` для пустой `String` возвращает `List` одной пустой `String`. Может быть, такой вариант имеет

смысл при других обстоятельствах... Возможно, но это портит наш алгоритм, поэтому мы должны обойти его с помощью специального случая в `parseLine` (пример 22.18).

Пример 22.18 [table-reader.14:src/main/java/travelator/tablereader/table-reading.kt]

```
private fun parseLine(line: String): Map<String, String> {
    val values = if (line.isEmpty()) emptyList() else line.split(",")
    val keys = values.indices.map(Int::toString)
    return keys.zip(values).toMap()
}
```

Это позволяет пройти тесты, но «мутит воду» в функции `parseLine`. Так что мы извлекаем «мутную» строку в функцию, назвав ее `splitFields` (пример 22.19).

Пример 22.19 [table-reader.15:src/main/java/travelator/tablereader/table-reading.kt]

```
private fun parseLine(line: String): Map<String, String> {
    val values = splitFields(line)
    val keys = values.indices.map(Int::toString)
    return keys.zip(values).toMap()
}

private fun splitFields(line: String): List<String> =
    if (line.isEmpty()) emptyList() else line.split(",")
```

Если сделать `splitFields` функцией-расширением и ввести параметр `separators`, мы получим функцию `split`, которую действительно всегда хотели получить (пример 22.20).

Пример 22.20 [table-reader.16:src/main/java/travelator/tablereader/table-reading.kt]

```
private fun parseLine(line: String): Map<String, String> {
    val values = line.splitFields(",")
    val keys = values.indices.map(Int::toString)
    return keys.zip(values).toMap()
}

private fun String.splitFields(separators: String): List<String> =
    if (isEmpty()) emptyList() else split(separators)
```

До сих пор мы получали код, работающий с пустым вводом, а затем с вводом одной строки. Если бы мы написали императивное решение, возможно, теперь пришлось бы добавить цикл для обработки большего количества входных данных, но `map` поддерживает нас, потому что всегда вернет столько элементов, сколько мы ей дадим. Мы верим, что `readTable` должна работать для всех известных программи-

стам чисел: 0, 1 и далее до бесконечности (ладно, до $2^{31} - 1$, а не до фактической бесконечности).

Говорят: «Доверяй, но проверяй», поэтому мы добавляем тест (пример 22.21).

Пример 22.21
[table-reader.17:src/test/java/travelator/tablereader/TableReaderTests.kt]

```
@Test
fun `two lines of input with default field names`() {
    assertEquals(
        listOf(
            mapOf("0" to "row0field0", "1" to "row0field1"),
            mapOf("0" to "row1field0", "1" to "row1field1")
        ),
        readTable(listOf(
            "row0field0,row0field1",
            "row1field0,row1field1"
        ))
    )
}
```

Тест проходит, и мы, решив, что `(0, 1, 2)` достаточно близко к `(0, 1, 2147483647)`, на этом и закончим. Самое время остановиться, приготовить свежий кофе и с удовольствием употребить его, прежде чем вернуться к работе.

Заголовки

Готовы снова отправиться в путь? Хорошо, как насчет строки заголовка?

Для начала — как наш API должен знать, что его следует ожидать? Мы могли бы добавить флаг к `readTable`, чтобы сообщить ему, что у наших данных есть заголовок, или можем добавить другую функцию. Как правило, мы предпочитаем отдельные функции для разных задач, поэтому давайте добавим функцию с именем `readTableWithHeader`.

Как и в случае с `readTable`, сначала мы добавляем тест, который вызывает требующуюся нам функцию (пример 22.22).

Пример 22.22
[table-reader.18:src/test/java/travelator/tablereader/TableReaderTests.kt]

```
@Test
fun `takes headers from header line`() {
    assertEquals(
        listOf(
            mapOf("H0" to "field0", "H1" to "field1")
        ),
```

```
        readTableWithHeader(
            listOf(
                "H0,H1",
                "field0,field1"
            )
        )
    )
}
```

Нажмите комбинацию клавиш <Alt>+<Enter> на ошибке компиляции в `readTableWithHeader`, и IntelliJ создаст ее для нас. Затем мы можем назвать параметры и делегировать их нашей нынешней исходной функции (пример 22.23).

Пример 22.23 [table-reader.18:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(lines: List<String>): List<Map<String, String>> {
    return readTable(lines)
}

fun readTable(lines: List<String>): List<Map<String, String>> {
    return lines.map(::parseLine)
}
```

Код компилируется, но не проходит тесты, как мы и ожидали:

```
org.opentest4j.AssertionFailedError:
Expected :[{H0=field0, H1=field1}]
Actual   :[{0=H0, 1=H1}, {0=field0, 1=field1}]
```

Чтобы тесты прошли успешно, мы могли бы жестко запрограммировать результат, как и раньше, но на этот раз собираемся изменить код, чтобы освободить место для функциональности. Когда мы говорим *освободить место*, то имеем в виду код, который выполняет текущую задачу (используя имена полей `Int::toString`) и который мы можем *дополнять*, а не изменять для поддержки новой функциональности. Тогда в соответствии с *принципом открытости/закрытости* (the open-closed principle — см. **https://oreil.ly/MwO5l**) новая функция будет скорее *дополнением*, чем модификацией.

В настоящее время информация о названии поля скрыта в `parseLine` (пример 22.24).

Пример 22.24 [table-reader.18:src/main/java/travelator/tablereader/table-reading.kt]

```
private fun parseLine(line: String): Map<String, String> {
    val values = line.splitFields(",")
    val keys = values.indices.map(Int::toString)
    return keys.zip(values).toMap()
}
```

Мы собираемся вытащить его отсюда в такое место, где сможем использовать строку заголовка для его предоставления.

`Int::toString` — это текущий маппинг от индекса к ключу. Давайте подготовимся к тому, чтобы сделать его настраиваемым, введя переменную с именем `headerProvider` (пример 22.25).

Пример 22.25 [table-reader.19:src/main/java/travelator/tablereader/table-reading.kt]

```
private fun parseLine(line: String): Map<String, String> {
    val values = line.splitFields(",")
    val headerProvider: (Int) -> String = Int::toString
    val keys = values.indices.map(headerProvider)
    return keys.zip(values).toMap()
}
```

Код все еще проходит наши тесты, за исключением новой `takes headers from header line`, которая все еще дает сбой. На самом деле нам не следует проводить рефакторинг с неудачным тестом, потому что при каждом запуске тестов придется проверять, действительно ли любой сбой соответствует ожидаемому. Поэтому мы отключили (`@Disabled`) его на текущий момент, чтобы запускать тесты во время рефакторинга только для завершенных функций.

Опция **Ввести параметр** (Introduce Parameter) в строке `headerProvider` с присвоением ему имени `headerProvider` позволит нам поддерживать различные модели поведения (пример 22.26).

Пример 22.26 [table-reader.20:src/main/java/travelator/tablereader/table-reading.kt]

```
private fun parseLine(
    line: String,
    headerProvider: (Int) -> String
): Map<String, String> {
    val values = line.splitFields(",")
    val keys = values.indices.map(headerProvider)
    return keys.zip(values).toMap()
}
```

К сожалению, IntelliJ в настоящее время не может заставить этот рефакторинг работать, разрушая `readTable` (пример 22.27).

Пример 22.27 [table-reader.20:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(lines: List<String>): List<Map<String, String>> {
    return readTable(lines)
}
```

```
fun readTable(lines: List<String>): List<Map<String, String>> {
    return lines.map(::parseLine) ❶
}
```

❶ Мы могли бы использовать ссылку на функцию, если бы у `parseLine` был всего один параметр. Но теперь ему нужны два аргумента, но `map` поддерживает только один.

Выполнение команды **Заменить ссылку на функцию на лямбду** (Replace function reference with lambda) *до* рефакторинга заставило бы все работать, но мы снова потерпим неудачу, расширив лямбду сейчас и добавив `Int::toString` в качестве `headerProvider` для повторной компиляции (пример 22.28).

Пример 22.28 [table-reader.21:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(lines: List<String>): List<Map<String, String>> {
    return readTable(lines)
}

fun readTable(lines: List<String>): List<Map<String, String>> {
    return lines.map { parseLine(it, Int::toString) }
}
```

Все тесты до сих пор проходят, так что мы полностью уверены, что ничего не сломали.

К чему мы с этим идем? Наш план заключается в том, чтобы новая `readTableWithHeader` считывала строку заголовка для создания `headerProvider` и передачи в `parseLine`. Между `readTableWithHeader` и `parseLine` находится вызов старой функции `readTable`, следовательно, ей тоже нужен параметр `header Provider`, чтобы она могла передавать значение. Так что снова применим опцию **Ввести параметр** вместе с **Ввести значение по умолчанию** (Introduce Default Value), но на этот раз к `Int::toString` в `readTable` (пример 22.29).

Пример 22.29 [table-reader.22:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(lines: List<String>): List<Map<String, String>> {
    return readTable(lines)
}

fun readTable(
    lines: List<String>,
    headerProvider: KFunction1<Int, String> = Int::toString ❶
): List<Map<String, String>> {
    return lines.map { parseLine(it, headerProvider) }
}
```

❶ Не компилируется: `Unresolved reference: KFunction1`.

Трудно сказать, почему IntelliJ (на момент подготовки книги) иногда использует при рефакторинге типы функций, а иногда типы `KFunctionN`. Было бы неплохо, если бы она работала последовательно или, по крайней мере, генерировала скомпилированный код. Мы исправим это, переведя `KFunction1` в `(Int) -> String` вручную, затаив небольшую обиду за этот второй неудачный рефакторинг подряд (пример 22.30).

Пример 22.30 [table-reader.23:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(lines: List<String>): List<Map<String, String>> {
    return readTable(lines)
}

fun readTable(
    lines: List<String>,
    headerProvider: (Int) -> String = Int::toString
): List<Map<String, String>> {
    return lines.map { parseLine(it, headerProvider) }
}
```

И положительный момент: поскольку параметр `headerProvider` имеет значение по умолчанию, наши тесты остаются неизменными и продолжают удачно выполняться.

Теперь мы в состоянии проанализировать строку заголовка. Функции `readTableWithHeader` потребуется считать заголовок, создать `headerProvider` (помните: `(Int) -> String`), а затем передать его `readTable`. Функции нужно разделить строки в заголовке (`Iterable.first()`) и в остальной части (`Iterable.drop(1)`). Но `Iterable.first` завершится неудачей, если в ней нет строк, поэтому мы делаем пометку: необходимо добавить тест для этого случая. Что касается преобразования строки заголовка в `header Provider`, то мы притворимся, что у нас есть функция для этого, которая называется `headerProviderFrom(String)` (пример 22.31).

Пример 22.31 [table-reader.24:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(lines: List<String>): List<Map<String, String>> {
    return readTable(
        lines.drop(1),
        headerProviderFrom(lines.first())
    )
}
```

Применение комбинации клавиш <Alt>+<Enter> к вызову новой функции позволит создать ее, и мы получим (пример 22.32).

Пример 22.32 [table-reader.24:src/main/java/travelator/tablereader/table-reading.kt]

```
fun headerProviderFrom(header: String): (Int) -> String {
    TODO("Not yet implemented")
}
```

Это функция, которая должна возвращать тип функции. Мы можем реализовать возвращаемое значение с помощью лямбды, которая принимает индекс `Int` и возвращает `String`. Нам нужно вернуть `String` — поле заголовка по этому индексу. Здесь можно снова применить `splitFields` (пример 22.33).

Пример 22.33 [table-reader.25:src/main/java/travelator/tablereader/table-reading.kt]

```
private fun headerProviderFrom(header: String): (Int) -> String {
    val headers = header.splitFields(",")
    return { index -> headers[index] }
}
```

Мы позаботились о том, чтобы разделить `header` снаружи лямбды. В противном случае это произойдет для каждой последующей строки таблицы. Тесты все еще проходят успешно, и если мы правы, то и тест для `readTableWithHeader`, который мы отключили ранее, пройдет удачно. Давайте включим его, убрав `@Disabled` (пример 22.34).

Пример 22.34 [table-reader.26:src/test/java/travelator/tablereader/TableReaderTests.kt]

```
@Test
fun `takes headers from header line`() {
    assertEquals(
        listOf(
            mapOf("H0" to "field0", "H1" to "field1")
        ),
        readTableWithHeader(
            listOf(
                "H0,H1",
                "field0,field1"
            )
        )
    )
}
```

Тест проходит успешно, ура! Мы готовы сказать, что уже закончили. Пока не посмотрим на наш список дел и не вспомним свое предположение: `readTableWithHeader` должна завершиться сбоем при пустом вводе. Итак, мы пишем тест, утверждающий желаемое поведение, которое заключается в возврате пустого `List` (пример 22.35).

Пример 22.35 [table-reader.26:src/test/java/travelator/tablereader/TableReaderTests.kt]

```
@Test
fun `readTableWithHeader on empty list returns empty list`() {
```

```
    assertEquals(
        emptyList<String>(),
        readTableWithHeader(
            emptyList()
        )
    )
}
```

Как мы и опасались, тест неудачен и выдает:

```
java.util.NoSuchElementException: List is empty
```

потому что `readTableWithHeader` пытается вызвать `lines.first()` для пустого `List` (пример 22.36).

Пример 22.36 [table-reader.25:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(lines: List<String>): List<Map<String, String>> {
    return readTable(
        lines.drop(1),
        headerProviderFrom(lines.first())
    )
}
```

Наше раздражение из-за того, что мы не закончили, смягчается правотой в утверждении, что проблема существует! Самое простое решение — разделить нашу функцию на два определения и с помощью `when` выбирать между ними. Такой вариант проходит все тесты и очищает наш список дел. Итак, вот наш общедоступный API (пример 22.37).

Пример 22.37 [table-reader.26:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(
    lines: List<String>
): List<Map<String, String>> =
    when {
        lines.isEmpty() -> emptyList()
        else -> readTable(
            lines.drop(1),
            headerProviderFrom(lines.first())
        )
    }

fun readTable(
    lines: List<String>,
    headerProvider: (Int) -> String = Int::toString
): List<Map<String, String>> =
    lines.map { parseLine(it, headerProvider) }
```

Как мило. Наши клиенты теперь могут считывать данные как со строкой заголовка, так и без нее. Но подождите! Глядя на код, мы понимаем, что если они захотят указать свои собственные имена полей для `read Table`, то смогут сделать это, переопределив значение по умолчанию `headerProvider` в `readTable`. У нас есть бесплатная дополнительная функция! Давайте напишем тест, чтобы продемонстрировать ее (пример 22.38).

Пример 22.38
[table-reader.27:src/test/java/travelator/tablereader/TableReaderTests.kt]

```
@Test
fun `can specify header names when there is no header row`() {
    val headers = listOf("apple", "banana")
    assertEquals(
        listOf(
            mapOf(
                "apple" to "field0",
                "banana" to "field1",
            )
        ),
        readTable(
            listOf("field0,field1"),
            headers::get
        )
    )
}
```

Посмотрите, как легко преобразовать `List<String>` в нашу функцию поставщика заголовков `(Int) -> String` при помощи ссылочного метода `headers::get`! Это интересный способ просмотра коллекций (рис. 22.1).

Тип	По типу функции	С помощью
`List<T>`	`(index: Int) -> T`	`List.get(index)`
`Set<T>`	`(item: T) -> Boolean`	`Set.contains(item)`
`Map<K, V>`	`(key: K) -> V?`	`Map.get(key)`

Рис. 22.1. Способы просмотра коллекций

Если нам удастся выразить зависимость как один из этих типов функций, то наши клиенты и наши тесты смогут использовать стандартные коллекции для обеспечения реализации.

Теперь, когда чтение таблицы с заголовком было успешно внедрено, мы можем попробовать запустить наш приемочной тест. Он был таким (пример 22.39).

Пример 22.39
[table-reader.26:src/test/java/travelator/tablereader/TableReaderAcceptanceTests.kt]

```
@Disabled
@Test
fun `acceptance test`() {
    val input = listOf(
        "time,x,y",
        "0.0, 1, 1",
        "0.1,1.1,1.2",
        "0.2,1.2,1.4",
    )
    val expected = listOf(
        Measurement(0.0, 1.0, 1.0),
        Measurement(0.1, 1.1, 1.2),
        Measurement(0.2, 1.2, 1.4)
    )
    assertEquals(
        expected,
        readTable(input).map { record ->
            Measurement(
                t = record["time"]?.toDoubleOrNull() ?: error("in time"),
                x = record["x"]?.toDoubleOrNull() ?: error("in x"),
                y = record["y"]?.toDoubleOrNull() ?: error("in y"),
            )
        }
    )
}
```

Функция, которую в момент написания теста мы думали назвать `readTable`, теперь называется `readTableWithHeader`, следовательно, мы вносим изменения и запускаем тест (пример 22.40).

Пример 22.40
[table-reader.27:src/test/java/travelator/tablereader/TableReaderAcceptanceTests.kt]

```
assertEquals(
    expected,
    readTableWithHeader(input).map { record ->
        Measurement(
            t = record["time"]?.toDoubleOrNull() ?: error("in time"),
            x = record["x"]?.toDoubleOrNull() ?: error("in x"),
            y = record["y"]?.toDoubleOrNull() ?: error("in y"),
        )
    }
```

Код проходит тест, и мы испытываем небольшой прилив дофамина, чтобы сохранить код и выпить чашечку кофе.

Различные разделители полей

Вернувшись от кофемашины, мы делаем быстрый обзор различных мест в `Travelator`, в которых считываются таблицы. Интересно, что у нас есть только один вариант использования, который считывает классические разделенные запятыми переменные (в кавычках), но в некоторых случаях нам необходимо в качестве разделителя полей иметь в виду точку с запятой. Помнится, в каком-то задании экспорта французского SQL Server используются точки с запятой, а затем файл сохраняется с расширением CSV. Далее мы рассмотрим их считывание, но попробуем найти интерфейс, который будет работать с более сложными правилами применения экранирования и кавычек. Чтобы добавить гибкости, нам нужно определить абстракцию, как мы это сделали ранее с `headerProvider`. В чем здесь заключается абстракция?

Глядя на код, мы видим, что парсинг заголовка и тела вызывает `splitFields` (пример 22.41).

Пример 22.41 [table-reader.28:src/main/java/travelator/tablereader/table-reading.kt]

```
private fun headerProviderFrom(header: String): (Int) -> String {
    val headers = header.splitFields(",")
    return { index -> headers[index] }
}

private fun parseLine(
    line: String,
    headerProvider: (Int) -> String
): Map<String, String> {
    val values = line.splitFields(",")
    val keys = values.indices.map(headerProvider)
    return keys.zip(values).toMap()
}

private fun String.splitFields(separators: String): List<String> =
    if (isEmpty()) emptyList() else split(separators)
```

Ни синтаксический анализ заголовка, ни синтаксический анализ тела на самом деле не хотят зависеть от деталей того, как должно происходить разделение, поэтому давайте абстрагируемся от этого за функцией `(String) -> List<String>`. Почему именно эта сигнатура, а не просто параметризация символа?

Это интересный вопрос, спасибо, что задали его. Введя параметр `separators` в `parseLine` и `headerProviderFrom` и в конечном счете в их абоненты `readTable` и

`readTableWithHeader`, мы пойдем по самому простому из возможных путей. Однако мы получим гораздо больше гибкости от использования типа функции, потому что можем скрыть все детали разделения, кавычек и экранирования за этой сигнатурой. В Java до появления лямбда преимущество гибкости не стоило бы затрат на внедрение и реализацию интерфейса SAM (Single Abstract Method, Единый абстрактный метод) — по крайней мере, до тех пор, пока нам действительно не понадобился весь этот контроль. С лямбдами в Java уравнение кажется более сбалансированным, но, вероятно, неестественным для большинства Java-программистов. В Kotlin, разработанном с самого начала с использованием типов функций как части языка, мы применяем их еще охотнее. Как только нам нужно параметризовать какой-либо аспект нашего кода, естественно спросить, будет ли функция предоставлять большее значение, чем, скажем, простое значение.

Давайте начнем с `parseLine`. Чтобы извлечь текущую реализацию разделения, мы можем выбрать `line.splitFields(",")` и применить опцию **Ввести функциональный параметр** (Introduce Functional Parameter), выбрав имя параметра `splitter` (пример 22.42).

Пример 22.42 [table-reader.29:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTable(
    lines: List<String>,
    headerProvider: (Int) -> String = Int::toString
): List<Map<String, String>> =
    lines.map {
        parseLine(it, headerProvider) { line -> ❶
            line.splitFields(",")
        }
    }

...

private fun parseLine(
    line: String,
    headerProvider: (Int) -> String,
    splitter: (String) -> List<String>, ❷
): Map<String, String> {
    val values = splitter(line)
    val keys = values.indices.map(headerProvider)
    return keys.zip(values).toMap()
}
```

❶ Эта лямбда...

❷ ...реализует разделитель.

Мы могли бы продолжить этот процесс, извлекая лямбду-разделитель на верхний уровень. Однако наша жизнь станет немного проще, если у нас будет глобальное

значение для разделителя, поэтому мы выбираем лямбду в `readTable` и применяем опцию **Ввести переменную** с названием `splitOnComma` (пример 22.43).

Пример 22.43 [table-reader.30:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTable(
    lines: List<String>,
    headerProvider: (Int) -> String = Int::toString
): List<Map<String, String>> =
    lines.map {
        val splitOnComma: (String) -> List<String> = { line ->
            line.splitFields(",")
        }
        parseLine(it, headerProvider, splitOnComma)
    }
```

Теперь мы можем вырезать `val` из функции и переместить его на верхний уровень (пример 22.44). Такое ощущение, что для этого должен существовать автоматический рефакторинг, но на момент подготовки книги таковой не работал.

Пример 22.44 [table-reader.31:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTable(
    lines: List<String>,
    headerProvider: (Int) -> String = Int::toString
): List<Map<String, String>> =
    lines.map {
        parseLine(it, headerProvider, splitOnComma)
    }

val splitOnComma: (String) -> List<String> = { line ->
    line.splitFields(",")
}
```

Теперь `splitOnComma` является глобальным свойством, и мы можем с удобством использовать его по умолчанию. Мы выбираем ссылку на него в `readTable` и применяем опцию **Ввести параметр** в паре с **Ввести значение по умолчанию**, назвав новый параметр `splitter` (пример 22.45).

Пример 22.45 [table-reader.32:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTable(
    lines: List<String>,
    headerProvider: (Int) -> String = Int::toString,
    splitter: (String) -> List<String> = splitOnComma
```

```
): List<Map<String, String>> =
    lines.map {
        parseLine(it, headerProvider, splitter)
    }

val splitOnComma: (String) -> List<String> = { line ->
    line.splitFields(",")
}
```

Из-за значения по умолчанию нам не пришлось менять ни одного из клиентов, и тесты продолжают проходить успешно. В том виде, в каком она есть, `readTable` использует поддерживаемый `splitter`, но `head erProviderFrom` этого не может (пример 22.46).

Пример 22.46 [table-reader.32:src/main/java/travelator/tablereader/table-reading.kt]

```
private fun headerProviderFrom(header: String): (Int) -> String {
    val headers = header.splitFields(",")
    return { index -> headers[index] }
}
```

Введем функциональный параметр для `header.splitFields(...)` (пример 22.47).

Пример 22.47 [table-reader.33:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(
    lines: List<String>
): List<Map<String, String>> =
    when {
        lines.isEmpty() -> emptyList()
        else -> readTable(
            lines.drop(1),
            headerProviderFrom(lines.first()) { header -> ❶
                header.splitFields(",")
            }
        )
    }
...

val splitOnComma: (String) -> List<String> = { line ->
    line.splitFields(",")
}

private fun headerProviderFrom(
    header: String,
    splitter: (String) -> List<String> ❷
```

```
): (Int) -> String {
    val headers = splitter(header)
    return { index -> headers[index] }
}
```

❶ Эта лямбда...

❷ ...реализует разделитель.

Теперь лямбда в `readTableWithHeader` представляет собой тот же код, что и `splitOnComma`, так что мы будем использовать последнюю (пример 22.48).

Пример 22.48 [table-reader.34:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(
    lines: List<String>
): List<Map<String, String>> =
    when {
        lines.isEmpty() -> emptyList()
        else -> readTable(
            lines.drop(1),
            headerProviderFrom(lines.first(), splitOnComma)
        )
    }
...

val splitOnComma: (String) -> List<String> = { line ->
    line.splitFields(",")
}
```

Вы можете увидеть здесь шаблон: мы создаем параметр из ссылки `splitOnComma`, опять же с настройкой по умолчанию, чтобы избежать нарушения существующих клиентов (пример 22.49).

Пример 22.49 [table-reader.35:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(
    lines: List<String>,
    splitter: (String) -> List<String> = splitOnComma
): List<Map<String, String>> =
    when {
        lines.isEmpty() -> emptyList()
        else -> readTable(
            lines.drop(1),
            headerProviderFrom(lines.first(), splitter)
        )
    }
```

Наконец, в `readTableWithHeader` мы вызываем `readTable` без предоставления `splitter`, так что она будет использовать значение по умолчанию (`splitOnComma`). Нам это не нужно, поэтому мы передаем параметр вниз. Заголовок и тело должны использовать один и тот же разделитель, так что мы передаем его из `readTableWithHeader` внутренней `readTable` (пример 22.50).

Пример 22.50 [table-reader.36:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(
    lines: List<String>,
    splitter: (String) -> List<String> = splitOnComma
): List<Map<String, String>> =
    when {
        lines.isEmpty() -> emptyList()
        else -> readTable(
            lines.drop(1),
            headerProviderFrom(lines.first(), splitter),
            splitter ❶
        )
    }

fun readTable(
    lines: List<String>,
    headerProvider: (Int) -> String = Int::toString,
    splitter: (String) -> List<String> = splitOnComma
): List<Map<String, String>> =
    lines.map {
        parseLine(it, headerProvider, splitter)
    }
```

❶ Передача `splitter`.

Некоторые разработчики, ориентированные на тестирование, могут настаивать на неудачном тестировании, чтобы показать необходимость этого последнего шага. Мы, безусловно, должны написать тест, чтобы продемонстрировать использование разделителя, но прежде, чем мы его напишем, давайте сделаем его более удобным для создания. Так выглядит `splitOnComma` (пример 22.51).

Пример 22.51 [table-reader.36:src/main/java/travelator/tablereader/table-reading.kt]

```
val splitOnComma: (String) -> List<String> = { line ->
    line.splitFields(",")
}
```

Было бы неплохо иметь возможность создавать разделители без необходимости каждый раз определять лямбду. Тогда наши французские клиенты могли бы вызвать `readTable` с, например, `splitter = splitOn(";")`. Функция `splitOn` получит раз-

делители и вернет значение типа функции `(String) -> List<String>`. Мы могли бы попытаться извлечь эту функцию из нашей текущей лямбды `splitOnComma`, но рефакторинг утомителен, поэтому давайте просто определим функцию и вызовем ее (пример 22.52).

Пример 22.52 [table-reader.37:src/main/java/travelator/tablereader/table-reading.kt]

```
fun splitOn(
    separators: String
): (String) -> List<String> = { line: String ->
    line.splitFields(separators)
}

val splitOnComma: (String) -> List<String> = splitOn(",")
val splitOnTab: (String) -> List<String> = splitOn("\t")
```

Как можно видеть, мы воспользовались возможностью определить `splitOnTab` тоже, так что для использования ее в новом тесте мы пообещали себе, что напишем его (пример 22.53).

Пример 22.53 [table-reader.38:src/test/java/travelator/tablereader/TableReaderTests.kt]

```
@Test
fun `can specify splitter`() {
    assertEquals(
        listOf(
            mapOf(
                "header1" to "field0",
                "header2" to "field1",
            )
        ),
        readTableWithHeader(
            listOf(
                "header1\theader2",
                "field0\tfield1"
            ),
            splitOnTab
        )
    )
}
```

Тест проходит успешно, давая нам как уверенность, так и документирование. Давайте проверим это и сделаем перерыв на несколько минут, прежде чем вернуться, чтобы подвести итоги.

Последовательности

Итак, у нас есть основы парсинга таблиц, и мы не ввели никаких новых типов, кроме тех, что используются в стандартной среде выполнения Kotlin. Так часто происходит при более функциональном подходе. Сущность Kotlin заключается в том, чтобы использовать богатые абстракции, предоставляемые стандартной библиотекой, где программы Java с большей вероятностью будут определять новые типы. Как показано в *главах 6* и *15*, одна из причин различия заключается в том, что Kotlin позволяет нам обрабатывать коллекции как значения, что делает их более безопасными для компоновки, чем мутирующие объекты Java. Мы можем определить API, который принимает и возвращает типы коллекций, не беспокоясь о псевдонимах.

Типы значений могут создавать API-интерфейсы, состоящие из предсказуемых вычислений, но также способны создавать свои собственные проблемы. Наш наивный API страдает от той же проблемы, с которой мы столкнулись в *главе 20*, — он работает с `List<String>`, загруженным в память, и выдает так же в память `List<Map<String, String>>`.

Даже без учета затратности структур данных, объем памяти `readTable` в два раза превышает количество байтов входных данных, что (вероятно) в два раза превышает размер содержащего данные файла в кодировке UTF-8. Для обработки больших файлов было бы неплохо работать в рамках последовательностей, а не в рамках списков, поскольку при необходимости последовательности могут одновременно сохранять в памяти только один элемент на каждом шаге конвейера.

Как показано в *главе 13*, можно очень легко преобразовать `Sequence` в `List` и обратно (с некоторыми оговорками) — следовательно, мы можем выполнить функции `Sequence`, передав их существующему API `List`. Однако это не уменьшило бы объем занимаемой памяти, поэтому вместо этого мы напишем версии `Sequence` и передадим им версии `List`. Если мы примем умное решение, то сможем провести тестирование с помощью удобного API `List` и, таким образом, получим два набора тестов по цене одного.

`readTable` сейчас выглядит так (пример 22.54).

Пример 22.54 [table-reader.39:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTable(
    lines: List<String>,
    headerProvider: (Int) -> String = Int::toString,
    splitter: (String) -> List<String> = splitOnComma
): List<Map<String, String>> =
    lines.map {
        parseLine(it, headerProvider, splitter)
    }
```

Мы можем опробовать наш план, преобразуя `readTable` в `Sequence` и из него в середине конвейера (пример 22.55).

Пример 22.55 [table-reader.40:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTable(
    lines: List<String>,
    headerProvider: (Int) -> String = Int::toString,
    splitter: (String) -> List<String> = splitOnComma
): List<Map<String, String>> =
    lines
        .asSequence()
        .map {
            parseLine(it, headerProvider, splitter)
        }
        .toList()
```

Код проходит тесты, и все они направлены через эту функцию, что обнадеживает. Теперь мы можем извлечь внутреннюю работу в функцию, принимающую и возвращающую `Sequence` (пример 22.56). Такое извлечение описано в *разд. «Извлечение части конвейера» главы 13*.

Пример 22.56 [table-reader.41:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTable(
    lines: List<String>,
    headerProvider: (Int) -> String = Int::toString,
    splitter: (String) -> List<String> = splitOnComma
): List<Map<String, String>> =
    readTable(
        lines.asSequence(),
        headerProvider,
        splitter
    ).toList()

fun readTable(
    lines: Sequence<String>,
    headerProvider: (Int) -> String = Int::toString,
    splitter: (String) -> List<String> = splitOnComma
) = lines.map {
        parseLine(it, headerProvider, splitter)
    }
```

Это даст нам версию `Sequence` из `readTable`, которую вызывает версия `List`, а версия `List` хорошо протестирована. Теперь перейдем к внешней `readTableWithHeader`. Она выглядит так (пример 22.57).

Пример 22.57 [table-reader.42:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(
    lines: List<String>,
    splitter: (String) -> List<String> = splitOnComma
): List<Map<String, String>> =
    when {
        lines.isEmpty() -> emptyList()
        else -> readTable(
            lines.drop(1),
            headerProviderFrom(lines.first(), splitter),
            splitter
        )
    }
```

В настоящий момент `readTableWithHeader` делегирует `List`-версию функции `readTable`. Если мы хотим создать ее `Sequence`-версию (а мы хотим), она должна вызывать `Sequence`-версию функции `readTable`, так что здесь мы заменяем встроенным выражением ее вызов и получаем (пример 22.58).

Пример 22.58 [table-reader.43:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(
    lines: List<String>,
    splitter: (String) -> List<String> = splitOnComma
): List<Map<String, String>> =
    when {
        lines.isEmpty() -> emptyList()
        else -> readTable(
            lines.drop(1).asSequence(),
            headerProviderFrom(lines.first(), splitter),
            splitter
        ).toList()
    }
```

Теперь можно вручную создать `linesAsSequence` в качестве переменной и применить ее вместо `lines`. Это почти работает (пример 22.59).

Пример 22.59 [table-reader.44:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(
    lines: List<String>,
    splitter: (String) -> List<String> = splitOnComma
): List<Map<String, String>> {
    val linesAsSequence = lines.asSequence()
```

```
    return when {
        linesAsSequence.isEmpty() -> emptySequence() ❶
        else -> {
            readTable(
                linesAsSequence.drop(1),
                headerProviderFrom(linesAsSequence.first(), splitter),
                splitter
            )
        }
    }.toList()
}
```

❶ Не компилируется, потому что нет `Sequence<T>.isEmpty()`.

Как мы можем определить, является ли `Sequence` пустой? `linesAsSequence.firstOrNull() == null` делает свое дело (пример 22.60).

Пример 22.60 [table-reader.45:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(
    lines: List<String>,
    splitter: (String) -> List<String> = splitOnComma
): List<Map<String, String>> {
    val linesAsSequence = lines.asSequence()
    return when {
        linesAsSequence.firstOrNull() == null -> emptySequence()
        else -> {
            readTable(
                linesAsSequence.drop(1),
                headerProviderFrom(linesAsSequence.first(), splitter),
                splitter
            )
        }
    }.toList()
}
```

Этот вариант проходит тесты, так что мы можем снова извлечь выражение между `return` и `.toList()` в качестве искомой функции. После указанного извлечения и наведения порядка мы имеем `Sequence`-версию `readTableWithHeader` (пример 22.61).

Пример 22.61 [table-reader.46:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(
    lines: List<String>,
    splitter: (String) -> List<String> = splitOnComma
): List<Map<String, String>> =
    readTableWithHeader(
        lines.asSequence(),
```

```
        splitter
    ).toList()

fun readTableWithHeader(
    lines: Sequence<String>,
    splitter: (String) -> List<String> = splitOnComma
) = when {
    lines.firstOrNull() == null -> emptySequence()
    else -> {
        readTable(
            lines.drop(1),
            headerProviderFrom(lines.first(), splitter),
            splitter
        )
    }
}
```

Сейчас у нас есть две версии `readTable` и `readTableWithHeader` — версии `List` и `Sequence` для каждой. Учитывая, как легко преобразовать аргумент `List` в `Sequence` и результат `Sequence` в `List`, возможно, варианты с `List` не окупаются?

Давайте просто перенесем их определения в тесты, пока у нас нет никаких рабочих применений. Тогда тесты смогут использовать их, чтобы оставаться простыми, а рабочий код останется минимальным.

Итак, вот весь открытый интерфейс нашего анализатора таблиц (пример 22.62).

Пример 22.62 [table-reader.47:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(
    lines: Sequence<String>,
    splitter: (String) -> List<String> = splitOnComma
): Sequence<Map<String, String>> =
    when {
        lines.firstOrNull() == null -> emptySequence()
        else -> readTable(
            lines.drop(1),
            headerProviderFrom(lines.first(), splitter),
            splitter
        )
    }

fun readTable(
    lines: Sequence<String>,
    headerProvider: (Int) -> String = Int::toString,
    splitter: (String) -> List<String> = splitOnComma
): Sequence<Map<String, String>> =
    lines.map {
        parseLine(it, headerProvider, splitter)
    }
```

```
val splitOnComma: (String) -> List<String> = splitOn(",")
val splitOnTab: (String) -> List<String> = splitOn("\t")

fun splitOn(
    separators: String
) = { line: String ->
    line.splitFields(separators)
}
```

Это поддерживается тремя служебными функциями (пример 22.63).

Пример 22.63 [table-reader.47:src/main/java/travelator/tablereader/table-reading.kt]

```
private fun headerProviderFrom(
    header: String,
    splitter: (String) -> List<String>
): (Int) -> String {
    val headers = splitter(header)
    return { index -> headers[index] }
}

private fun parseLine(
    line: String,
    headerProvider: (Int) -> String,
    splitter: (String) -> List<String>,
): Map<String, String> {
    val values = splitter(line)
    val keys = values.indices.map(headerProvider)
    return keys.zip(values).toMap()
}
// Необходимо, потому что String.split возвращает список пустой строки
// при вызове для пустой строки.
private fun String.splitFields(separators: String): List<String> =
    if (isEmpty()) emptyList() else split(separators)
```

Из этого кода не совсем ясно, зачем нам нужна `splitFields`, поэтому мы добавили комментарий. Часто это удобнее сделать в ретроспективе, когда мы пытаемся понять код, к которому возвращаемся, а не когда его только что написали. Кроме того, мы считаем, что наш код сам по себе довольно понятен. Но иногда мы в этом ошибаемся. И если нам потребуется больше, чем беглый взгляд, для понимания того, что происходит, то при повторном прочтении кода мы можем воспользоваться возможностью добавить больше комментариев или, что еще лучше, провести рефакторинг, чтобы код стал более выразительным.

Считывание из файла

Абстрактно наше создание кажется прекрасным интерфейсом, но когда мы сгоряча пытаемся его сразу использовать, то натыкаемся на препятствие. Давайте проиллюстрируем проблему с помощью теста. Вот вызов `Sequence`-версии `readTableWithHeader` (пример 22.64).

Пример 22.64 [table-reader.48:src/test/java/travelator/tablereader/TableReaderTests.kt]

```
@Test
fun `read from reader`() {
    val fileContents = """
        H0,H1
        row0field0,row0field1
        row1field0,row1field1
    """.trimIndent()
    StringReader(fileContents).useLines { lines ->
        val result = readTableWithHeader(lines).toList()
        assertEquals(
            listOf(
                mapOf("H0" to "row0field0", "H1" to "row0field1"),
                mapOf("H0" to "row1field0", "H1" to "row1field1")
            ),
            result
        )
    }
}
```

Понимаете, почему это дает сбой? А что, если мы скажем, что сбой происходит с ошибкой `java.lang.IllegalStateException: This sequence can be consumed only once`?

Да, снова множественные итерации `Sequence` (см. соответствующий раздел *главы 13*) кусаются, потому что мы не тестировали оба типа — те, которые могут и не могут использоваться дважды — в качестве входных данных (пример 22.65).

Пример 22.65 [table-reader.47:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(
    lines: Sequence<String>,
    splitter: (String) -> List<String> = splitOnComma
): Sequence<Map<String, String>> =
    when {
        lines.firstOrNull() == null -> emptySequence()
        else -> readTable(
            lines.drop(1),
```

```
            headerProviderFrom(lines.first(), splitter),
            splitter
        )
    }
```

Следовательно, `lines.firstOrNull()` потребит последовательность, и при чтении из `Reader` мы не сможем просто вернуться и начать все сначала, чтобы оценить `lines.drop(1)` и `lines.first()`. Все наши модульные тесты начинались со списка `List` всех строк файла. Эти последовательности *могут* применяться повторно, потому что хранятся в памяти.

Чтобы использовать интерфейс `Sequence` для данных в файлах, нам придется либо загрузить все это в память, либо найти способ извлечь первую и оставшуюся части `Sequence`, не пытаясь считать ее дважды. Учитывая, что мы ввели `Sequence` специально для того, чтобы избежать одновременной загрузки всех данных в память, выбираем последнее. Все, что нам нужно сделать, это проверить, есть ли в `Sequence` какие-либо элементы, не потребляя их. Знаете как?

Ах, это был вопрос с подвохом. Чтобы выполнить такую проверку, нам *необходимо* вызвать `iterator()` для `Sequence`, которая как раз и является тем, что ее потребляет. Мы не можем увидеть, является ли `Sequence` пустой, а позже снова ее использовать. Однако иногда в логике, когда нет возможности сделать желаемое отдельно, мы можем сделать вместе это и еще что-то. И сейчас нам требуется не просто узнать, пуста ли `Sequence`. Мы хотим разделить ее на «голову» и «хвост», если это не так. Можно достичь этой более широкой цели, разрушив `Sequence` с помощью функции, подобной этой (пример 22.66).

Пример 22.66 [table-reader.49:src/main/java/travelator/tablereader/table-reading.kt]

```
fun <T> Sequence<T>.destruct()
    : Pair<T, Sequence<T>>? {
    val iterator = this.iterator()
    return when {
        iterator.hasNext() ->
            iterator.next() to iterator.asSequence()
        else -> null
    }
}
```

Функция `destruct` вернет `null`, если `Sequence` пуста. В противном случае она вернет `Pair`, состоящую из «головы» и «хвоста» (где «хвост» может быть пустой `Sequence`). Она потребляет оригинал (вызывая `iterator()`), но предоставляет новую `Sequence` для продолжения обработки. Теперь можно использовать ее для рефракторинга `readTableWithHeader` (пример 22.67).

Пример 22.67 [table-reader.48:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(
    lines: Sequence<String>,
    splitter: (String) -> List<String> = splitOnComma
): Sequence<Map<String, String>> =
    when {
        lines.firstOrNull() == null -> emptySequence()
        else -> readTable(
            lines.drop(1),
            headerProviderFrom(lines.first(), splitter),
            splitter
        )
    }
```

Это, конечно, нетривиальная перестановка, но мы можем преобразовать ее в следующий код (пример 22.68).

Пример 22.68 [table-reader.49:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(
    lines: Sequence<String>,
    splitter: (String) -> List<String> = splitOnComma
): Sequence<Map<String, String>> {
    val firstAndRest = lines.destruct()
    return when {
        firstAndRest == null -> emptySequence()
        else -> readTable(
            firstAndRest.second,
            headerProviderFrom(firstAndRest.first, splitter),
            splitter
        )
    }
}
```

Новая форма проходит все тесты, потому что она не потребляет `lines` более одного раза. Если это кажется немного неуклюжим, мы можем объединить `?.let`, деструктурирование и оператор Элвиса, чтобы получить однострочное выражение, которое можно счесть как минимум приемлемо кратким. Результатом является следующий публичный API (пример 22.69).

Пример 22.69 [table-reader.50:src/main/java/travelator/tablereader/table-reading.kt]

```
fun readTableWithHeader(
    lines: Sequence<String>,
    splitter: (String) -> List<String> = splitOnComma
```

```
): Sequence<Map<String, String>> =
    lines.destruct()?.let { (first, rest) ->
        readTable(
            rest,
            headerProviderFrom(first, splitter),
            splitter
        )
    } ?: emptySequence()

fun readTable(
    lines: Sequence<String>,
    headerProvider: (Int) -> String = Int::toString,
    splitter: (String) -> List<String> = splitOnComma
): Sequence<Map<String, String>> =
    lines.map {
        parseLine(it, headerProvider, splitter)
    }

val splitOnComma: (String) -> List<String> = splitOn(",")
val splitOnTab: (String) -> List<String> = splitOn("\t")

fun splitOn(
    separators: String
) = { line: String ->
    line.splitFields(separators)
}
```

Мы почти закончили, честно.

Последний шаг, теперь, когда API сформировался вокруг двух функций, заключается в том, чтобы воспользоваться возможностью сделать наши тесты более выразительными (пример 22.70).

Пример 22.70
[table-reader.52:src/test/java/travelator/tablereader/TableReaderTests.kt]

```
class TableReaderTests {
    @Test
    fun `empty input returns empty`() {
        checkReadTable(
            lines = emptyList(),
            shouldReturn = emptyList()
        )
    }

    @Test
    fun `one line of input with default field names`() {
```

```
        checkReadTable(
            lines = listOf("field0,field1"),
            shouldReturn = listOf(
                mapOf("0" to "field0", "1" to "field1")
            )
        )
    }

    ...
    @Test
    fun `can specify header names when there is no header row`() {
        val headers = listOf("apple", "banana")
        checkReadTable(
            lines = listOf("field0,field1"),
            withHeaderProvider = headers::get,
            shouldReturn = listOf(
                mapOf(
                    "apple" to "field0",
                    "banana" to "field1",
                )
            )
        )
    }

    @Test
    fun `readTableWithHeader takes headers from header line`() {
        checkReadTableWithHeader(
            lines = listOf(
                "H0,H1",
                "field0,field1"
            ),
            shouldReturn = listOf(
                mapOf("H0" to "field0", "H1" to "field1")
            )
        )
    }

    ...
}

private fun checkReadTable(
    lines: List<String>,
    withHeaderProvider: (Int) -> String = Int::toString,
    shouldReturn: List<Map<String, String>>,
) {
    assertEquals(
        shouldReturn,
```

```
        readTable(
            lines.asSequence().constrainOnce(),
            headerProvider = withHeaderProvider,
            splitter = splitOnComma
        ).toList()
    )
}

private fun checkReadTableWithHeader(
    lines: List<String>,
    withSplitter: (String) -> List<String> = splitOnComma,
    shouldReturn: List<Map<String, String>>,
) {
    assertEquals(
        shouldReturn,
        readTableWithHeader(
            lines.asSequence().constrainOnce(),
            splitter = withSplitter
        ).toList()
    )
}
```

Это важный шаг. Как показано в *главе 17*, поиск шаблонов в тестах и выражение их в функциях (например, `checkReadTable`) помогают читателям тестов увидеть, что делает код, а нам могут помочь найти пробелы в нашем тестовом покрытии. Например, как ведет себя наш анализатор, когда полей больше, чем заголовков, или наоборот? Тесты, которые мы пишем для получения быстрой обратной связи во время тестирования реализации, вряд ли будут оптимально эффективными для информирования об API, поиска проблем или выявления регрессий, если мы вернемся к реализации и изменим ее. И если мы используем *разработку через тестирование* (Test-Driven Development, TDD) в качестве метода проектирования, то не должны забывать убедиться, что окончательные тесты подходят для определения правильности, добавления документирования и предотвращения регрессии.

Сравнение с Commons CSV

Мы начали эту главу с мысли о том, что в большинстве реальных ситуаций мы бы обратились к Apache Commons CSV — вместо того, чтобы запускать наш собственный синтаксический парсер (анализатор). Прежде чем мы ее закончим, давайте сравним наш API с его эквивалентом в Commons.

Наиболее распространенным вариантом использования анализатора таблиц является чтение файла с известными столбцами и перевод каждой строки в некоторый класс данных. Вот как мы это делаем с помощью нашего синтаксического анализатора (пример 22.71).

Пример 22.71
[table-reader.53:src/test/java/travelator/tablereader/CsvExampleTests.kt]

```
@Test
fun example() {
    reader.useLines { lines ->
        val measurements: Sequence<Measurement> =
            readTableWithHeader(lines, splitOnComma)
                .map { record ->
                    Measurement(
                        t = record["time"]?.toDoubleOrNull()
                            ?: error("in time"),
                        x = record["x"]?.toDoubleOrNull()
                            ?: error("in x"),
                        y = record["y"]?.toDoubleOrNull()
                            ?: error("in y"),
                    )
                }
        assertEquals(
            expected,
            measurements.toList()
        )
    }
}
```

В реальном коде, вероятно, потребуется больше обработки ошибок (мы показали, как это делается, в *главе 21*), но здесь мы рассматриваем лишь его базовый вариант. Мы применяем функцию-расширение Kotlin `Reader.useLines` для создания `Sequence<String>`, которую наш анализатор преобразует в `Sequence<Map<String, String>>`. Можно применить `map` ко всем `Map`, индексируя по имени поля для извлечения нужных нам данных и преобразования их в тип (`Measurement`), который фактически нужен. Эта конструкция появилась не случайно — такие решения мы приняли в самом начале, хотя в то время использовали `List`, а не `Sequence`.

Перед вами версия Commons CSV (пример 22.72).

Пример 22.72
[table-reader.53:src/test/java/travelator/tablereader/CsvExampleTests.kt]

```
@Test
fun `commons csv`() {
    reader.use { reader ->
        val parser = CSVParser.parse(
            reader,
            CSVFormat.DEFAULT.withFirstRecordAsHeader()
        )
```

```
        val measurements: Sequence<Measurement> = parser
            .asSequence()
            .map { record ->
                Measurement(
                    t = record["time"]?.toDoubleOrNull()
                        ?: error("in time"),
                    x = record["x"]?.toDoubleOrNull()
                        ?: error("in x"),
                    y = record["y"]?.toDoubleOrNull()
                        ?: error("in y"),
                )
            }
        assertEquals(
            expected,
            measurements.toList()
        )
    }
}
```

У этого кода тоже есть точка входа статической функции `CSVParser.parse`, которая также принимает конфигурацию относительно формата таблицы (здесь `CSVFormat.DEFAULT.withFirstRecordAsHeader()`, а в нашем случае `splitOnComma`). У нас есть две функции для различения файлов с заголовками или без них. Apache API преобразует это в `CSVFormat`.

Однако метод `parse` из Commons принимает `Reader`, а не нашу `Sequence<String>`. Это позволяет ему обрабатывать разделители записей, отличные от новой строки, и справляться с наличием новых строк в середине полей, но приводит к увеличению методов `parse`. Также есть варианты, принимающие `Path`, `File`, `InputStream`, `String` и `URL`. Разработчики, вероятно, сочли это необходимым, потому что Java предоставляет слишком мало поддержки для преобразования между этими типами источников и безопасного удаления из них. `CSVParser`, возвращенный статическим методом `parse`, содержит много кода для управления ресурсами. Наш API делегирует их для работы `Sequence` и таких функций жизненного цикла Kotlin, как `use` и `useLines`.

Что касается строк, вы должны вчитаться в пример кода, чтобы увидеть, как `CSVParser` реализует `Iterable<CSVRecord>`. Это разумный выбор конструкции, поскольку он позволяет разработчикам Java задействовать инструкцию `for` при реализации цикла для записей, а разработчикам Kotlin преобразовывать в `Sequence` с помощью `.asSequence`. Фактически удобство применения Kotlin обусловлено конструкцией стандартной библиотеки Kotlin, основанной на той же `Iterable`-абстракции, которую используют разработчики Apache.

Продвигаемся дальше. Код для создания индивидуального `Measurement` выглядит одинаково в обоих примерах (пример 22.73).

Пример 22.73
[table-reader.53:src/test/java/travelator/tablereader/CsvExampleTests.kt]

```
.map { record ->
    Measurement(
        t = record["time"]?.toDoubleOrNull()
            ?: error("in time"),
        x = record["x"]?.toDoubleOrNull()
            ?: error("in x"),
        y = record["y"]?.toDoubleOrNull()
            ?: error("in y"),
    )
}
```

Хотя типом `record` в нашем анализаторе является `Map<String, String>`, в случае с Commons — это `CSVRecord`. `CSVRecord` содержит метод `get(String)`, с помощью которого `record["time"]` и последующие решаются. В нем также содержатся методы: `get(int)` для извлечения полей по индексам, где мы можем применить `Map.values.get(Int)`; `size()` вместо `Map.size()`; `isSet(String)` в качестве замены `Map.hasKey(String)`.

В принципе, сущность `CSVRecord` заключается в том, чтобы воспроизвести интерфейс `Map` вручную, а не просто в *существовании* `Map`. Почему? Потому что, как мы обсуждали в *главе 6*, интерфейс Java `Map` является мутитрующим, а мутация не имеет смысла в контексте чтения полей из файла, поскольку мутации, безусловно, не будут записаны обратно в исходный код. При программировании на Java нам приходится создавать новые типы для решения проблем, тогда как в Kotlin мы можем выражать себя в стандартных типах, а затем наслаждаться богатством Kotlin API для этих типов.

Одной из областей, в которой выделяется библиотека Commons CSV library Excels™, является предоставление готовых параметров парсинга по умолчанию. Они выражаются в виде констант в классе `CSVFormat`. Мы встречали `CSVFormat.DEFAULT`, но есть многие другие, включая `CSVFormat.EXCEL`. Вооружившись `CSVFormat`, вы можете передать его методу `CSVParser.parse`, как показано ранее, или применить его напрямую. Например, так: `CSVFormat.EXCEL.parse(reader)`. Можем ли мы предоставить эту возможность без определения новых типов в нашем API? Как насчет применения `splitOnComma`, как если бы это была наша конфигурация (пример 22.74).

Пример 22.74
[table-reader.54:src/test/java/travelator/tablereader/CsvExampleTests.kt]

```
@Test
fun `configuration example`() {
    reader.use { reader ->
        val measurements = splitOnComma.readTableWithHeader(reader)
```

```
            .map { record ->
                Measurement(
                    t = record["time"]?.toDoubleOrNull()
                        ?: error("in time"),
                    x = record["x"]?.toDoubleOrNull()
                        ?: error("in x"),
                    y = record["y"]?.toDoubleOrNull()
                        ?: error("in y"),
                )
            }
        assertEquals(
            expected,
            measurements.toList()
        )
    }
}
```

Мы можем достичь этого, определив `splitOnComma.readTableWithHeader(reader)` как функцию-расширение для типа функции (пример 22.75).

Пример 22.75 [table-reader.54:src/main/java/travelator/tablereader/table-reading.kt]

```
fun ((String) -> List<String>).readTableWithHeader(
    reader: StringReader
): Sequence<Map<String, String>> =
    readTableWithHeader(reader.buffered().lineSequence(), this)
```

На самом деле `CSVFormat` представляет собой целый пакет стратегий для обхода правил, показывающих, что делать с пустыми строками и т. д., а не только как разделить строку. Когда наш анализатор расширит эти возможности, мы, вероятно, захотим создать класс данных для их сбора. До этого момента мы могли продвигаться вперед, используя только встроенные типы и функции языка Kotlin.

Но есть еще одна полезная функция, которую предоставляет интерфейс Commons, чего нет в нашем интерфейсе, и которую нам придется создать для реализации. Commons CSV содержит `CSVParser.getHeaderNames` для предоставления доступа к информации заголовка. Можем ли мы добавить это средство, не изменяя наш текущий API или, по крайней мере, не требуя изменений в нашем клиентском коде?

Для многих входных данных мы могли бы просто вызвать `Map.keys` на первом из выходных `Sequence`, но это не сработает, если в таблице нет строк данных, а есть только заголовок. Чтобы вернуть информацию заголовка и проанализированные записи, мы могли бы вернуть `Pair<List<String>,Sequence<Map<String, String>>`, но это вынудит наших нынешних клиентов отказаться от первого из пары. Вместо этого мы можем вернуть тип `Table`, который вводит `Sequence<Map<String, String>>`, но также содержит свойство заголовка. Таким образом, все наши текущие абоненты остают-

ся неизменными, но мы можем получить доступ к `headers`, когда это требуется (пример 22.76).

Пример 22.76 [table-reader.55:src/test/java/travelator/tablereader/TableReaderTests.kt]

```
@Test
fun `Table contains headers`() {
    val result: Table = readTableWithHeader(
        listOf(
            "H0,H1",
            "field0,field1"
        ).asSequence()
    )
    assertEquals(
        listOf("H0", "H1"),
        result.headers
    )
}

@Test
fun `Table contains empty headers for empty input`() {
    assertEquals(
        emptyList<String>(),
        readTableWithHeader(emptySequence()).headers
    )
}
```

Мы избавим вас от шагов рефакторинга, сразу представив полученную реализацию (пример 22.77).

Пример 22.77 [table-reader.55:src/main/java/travelator/tablereader/table-reading.kt]

```
class Table(
    val headers: List<String>,
    val records: Sequence<Map<String, String>>
) : Sequence<Map<String, String>> by records

fun readTableWithHeader(
    lines: Sequence<String>,
    splitter: (String) -> List<String> = splitOnComma
): Table =
    lines.destruct()?.let { (first, rest) ->
        tableOf(splitter, first, rest)
    } ?: Table(emptyList(), emptySequence())
```

```
private fun tableOf(
    splitter: (String) -> List<String>,
    first: String,
    rest: Sequence<String>
): Table {
    val headers = splitter(first)
    val sequence = readTable(
        lines = rest,
        headerProvider = headers::get,
        splitter = splitter
    )
    return Table(headers, sequence)
}
```

Двигаемся дальше

На этом, заключительном этапе нашего путешествия мы позволили себе роскошь написать Kotlin с нуля, а не выполнять рефакторинг существующего кода Java. Даже теперь мы начали с тестов, а затем просто скопировали тестовые данные в нашу реализацию и осуществили рефакторинг оттуда. Мы не можем написать весь код таким образом, но он действительно хорошо работает, когда представляет собой просто вычисления. И чем больше в нем просто вычислений, тем лучше работает и наш код.

Мы видели возможности повторного использования встроенных типов в *главе 15 «От инкапсулированных коллекций к псевдонимам типов»* и в *главе 16 «От интерфейсов к функциям»*. А также определение API в качестве функций-расширений в *главе 10 «От функций к функциям-расширениям»*.

В нашем примере оба типа коллекций и функций прекрасно сочетались, и нам даже удалось определить функцию-расширение для типа функции! Там, где нам пришлось бы определять новые классы для инкапсуляции мутирующих коллекций Java и методов для управления этими коллекциями, мы передали немутирующие коллекции Kotlin между нашими функциями и написали расширения для конкретных приложений этих типов коллекций. Там, где нам нужно было бы определять интерфейсы на Java, мы использовали типы функций Kotlin.

Опять же, не все проблемы могут или должны решаться таким образом, но ваши авторы обнаружили, что, хотя трудно заставить Java двигаться в этом направлении, функции Kotlin объединяются, чтобы активно поощрять такой стиль. Мы не должны зацикливаться на том, чтобы не определять новые типы, но и не должны бросаться решать каждую проблему с помощью нового класса.

ГЛАВА 23

Продолжение путешествия

Мы добрались до завершения книги. Спасибо, что присоединились к путешествию. Вашим авторам выпала честь работать со многими великими разработчиками и учиться у них, и теперь вы в этом списке. Даже если вы пропустили пару глав или отключились в середине сложного рефакторинга, хорошо, что нам все-таки было с кем поговорить. Мы больше не можем работать над улучшением `Travelator`*, но чему мы научились в наших путешествиях?*

Когда O'Reilly спросили нас, не хотели бы мы написать книгу о Kotlin, нам пришлось задуматься о том, *что* мы хотим написать, и о том, *что* может захотеть прочитать достаточное количество людей. Мы знали, что проделали долгий путь, осваивая этот язык, и что нам комфортно в месте назначения, но мы также знали, что наша отправная точка не была таковой для типичного Java-разработчика. Мы видели, что в большинстве существующих книг Kotlin подается так, как просто еще один синтаксис Java, который может дать тот же результат при меньшем количестве текста, но не требует изменения подхода. Наш опыт не подтверждал это мнение — мы обнаружили, что для Kotlin требуется более функциональное мышление, чем для Java. Однако книги по функциональному программированию на Kotlin просят читателя оставить позади все, что он знает об объектном программировании, и присоединиться к новому культу. Нас это тоже не устраивало. Классы и объекты — это гуманный способ выражения поведения, особенно по сравнению со многими функциональными средствами. Зачем выбрасывать инструменты из ящика, когда там достаточно места для новых? Разве мы не можем просто получить больше инструментов и выбрать для работы наиболее подходящий?

О сущностях

Эти размышления привели Нэта к представлению о том, что языки программирования имеют сущность, которая влияет на конструктивные особенности программ, которые мы на них пишем. Сущность языка облегчает применение одних стилей разработки, а другие делает трудными или рискованными.

Сущность Kotlin отличается от сущности Java. Сущность Java благоприятствует мутирующим (изменяющимся) объектам и рефлексии за счет компонуемости и безопасности типов. По сравнению с Java Kotlin предпочитает преобразование неизменяемых значений и автономных функций и содержит систему типов, которая является ненавязчивой и полезной. Хотя преобразовать Java в Kotlin с помощью

IntelliJ легко, в итоге мы получаем Java в синтаксисе Kotlin вместо того, чтобы воспользоваться всеми преимуществами, которые мог бы предложить новый язык, если бы изменили и свое мышление.

Java и Kotlin могут сосуществовать в одной и той же кодовой базе, и граница их взаимодействия практически не нарушается, но при передаче информации из строго типизированного мира Kotlin в более свободно типизированный мир Java возникают некоторые риски. При этом мы обнаруживаем, что можем с некоторой долей осторожности преобразовать характерный для Java код в код, характерный для Kotlin, небольшими и безопасными шагами, используя, где это возможно, автоматизированные инструменты рефакторинга и — в качестве последнего средства — «ручное» редактирования кода. Мы также можем одновременно поддерживать соглашения обоих языков во время преобразования кода Java в код Kotlin.

Функциональное мышление

Как мы показали в наших уроках истории (см. *главу 1*), сущность Java была сформирована в 90-х годах прошлого века — тогда мы верили, что объектно-ориентированное программирование — это мифическая «серебряная пуля». Когда оказалось, что ООП не решает всех наших проблем, основные языки программирования и даже сама Java начали перенимать идеи функционального программирования. В эти времена Kotlin родился из Java, и подобно тому, как наши дети лучше подготовлены к будущему, чем мы, Kotlin больше подходит для современного программирования, чем Java.

Что мы подразумеваем под *функциональным мышлением*?

Программное обеспечение в конечном счете ограничено нашей способностью понимать его. Наше понимание, в свою очередь, ограничено сложностью созданного нами программного обеспечения, и большая часть этой сложности возникает из-за путаницы в том, *когда* что-то происходит. Функциональные программисты поняли, что самый простой способ укротить эту сложность — просто сделать так, чтобы события *происходили* намного реже. Они называют происходящее *эффектом* — изменением, которое можно наблюдать в некоторой области.

Изменение переменной или коллекции *внутри* функции — это эффект, но если эта переменная не является общедоступной *вне* функции, она не *влияет* ни на какой другой код. Когда область действия эффекта локальна для функции, нам не нужно учитывать это при рассуждении о том, что делает наша система. Как только мы изменяем общее состояние (возможно, параметр функции, или глобальную переменную, или файл, или сетевой сокет), локальный эффект становится эффектом в любой области, которая может видеть общедоступное выражение, и это быстро усложняет и затрудняет понимание.

Недостаточно того, что функция *фактически* не изменяет общее состояние. Если существует вероятность того, что функция *может* изменить общее состояние, мы должны изучить источник функции и рекурсивно каждую функцию, которую она вызывает, чтобы понять, что делает наша система. Каждый фрагмент глобального

мутирующего состояния делает каждую функцию подозрительной. Аналогично если мы программируем в среде, в которой каждая функция может выполнять запись в базу данных, то теряем способность предсказывать, когда могут произойти такие записи, и планировать этот процесс соответствующим образом.

Получается, что функциональные программисты укрощают сложность, уменьшая мутации. Иногда они программируют на языках (таких как Clojure и Haskell), которые обеспечивают контроль над мутациями. В ряде случаев они работают на основе соглашений. Если мы примем эти соглашения на более широком спектре языков, то получим больше возможностей для работы с нашим кодом. Kotlin предпочитает не применять контроль над эффектами, но язык и его среда выполнения имеют некоторые встроенные соглашения, которые подталкивают нас в правильном направлении. По сравнению с Java у нас есть, например, немутирующее объявление `val`, а не необязательный модификатор `final`, просмотр предназначенных только для чтения коллекций и краткие классы данных, поощряющие копирование при записи, а не мутацию. Многие главы этой книги описывают более тонкие соглашения с той же целью: *глава 5 «От объектов JavaBeans к значениям»*, *глава 6 «От коллекций Java к коллекциям Kotlin»*, *глава 7 «От действий к вычислениям»*, *глава 14 «От накопления объектов к преобразованиям»* и *глава 20 «От выполнения ввода/вывода к передаче данных»*.

Конечно, функциональное программирование — это гораздо больше, чем просто отказ от изменения общего состояния. Но если просто сосредоточиться на решении проблем без мутации (или там, где речь идет о мутации, минимизировать ее масштабы), наши системы станет легче понимать и изменять. Усердная ориентация на простой принцип программирования «Не повторяйся» (DRY, Don't repeat yourself — см. **https://oreil.ly/HSaLs**), также известный как «Один и только один раз» (Once and only once — см. **https://oreil.ly/5HKxy**), имеет глубокие последствия. Однако совокупная ориентация на принципы «Не изменяйте общее состояние» (Don't mutate shared state) и «Один и только один раз» вызывает опасение, что если мы не будем осторожны, применение правил может увеличить сложность быстрее, чем уменьшить ее. Нам нужно изучить методы, которые позволяют управлять мутациями (и устранять дублирование, облегчать тестирование и т. д.), не делая наш код более трудным для понимания, и научиться распознавать эти методы такими, какие они есть, когда они нам встречаются. Эти методы, как правило, различаются в разных языках, средах и доменах и представляют собой основу искусства нашей профессии.

Исследуя функциональные методы, вы столкнетесь со множеством антиобъектных настроений. Похоже, это коренится в представлении о том, что ООП строится на изменяемых объектах, но не стоит выплескивать из ванны ребенка, передающего сообщения, вместе с изменяющейся водой. Хотя существует возможность использовать ООП для управления общим мутирующим состоянием, на практике в наши дни мы обычно используем объекты для инкапсуляции неизменяемого состояния или для представления служб и их зависимостей. В *главе 16 «От интерфейсов к функциям»* мы видели, что для инкапсуляции данных мы можем использовать как функции с закрытием, так и классы со свойствами. Оба эти метода также могут

скрывать детали кода и позволять клиенту работать с различными реализациями. Нам нужны такие методы для создания гибких, надежных и тестируемых систем. Там, где в Java мы традиционно используем подклассы как инструмент, Kotlin с его запечатанными по умолчанию классами поощряет более композиционный стиль. Вместо переопределения защищенного метода мы получаем свойство с функциональным типом, представляющее стратегию или соглашение. Стоит отдавать предпочтение этому стилю, но не стесняться определять иерархии классов и подклассов там, где они упрощают реализацию. Аналогичным образом, функции-расширения, описанные в *главе 10 «От функций к функциям-расширениям»*, очень хороши, и они могут творить чудеса, облегчая соединения между различными проблемами в наших кодовых базах, но они не заменяют полиморфные методы, когда они нам бывают нужны.

В конце концов, одной из привлекательных сторон программирования является сочетание человеческого и математического. Объекты и классы, по крайней мере для ваших авторов, являются более человечным способом моделирования мира, и это часто представляется прекрасной отправной точкой. Когда нам нужна строгость (что случается часто, но не так часто, как могли бы подумать люди, не обладающие широким воображением), функциональное программирование всегда рядом с нами. Мы не видим причин выбирать тот или иной лагерь, когда у нас может быть в каждом по палатке и возможность перемещаться между ними, и Kotlin позволяет нам делать это лучше, чем любой другой известный нам язык.

Простая конструкция

Если сложность является ограничивающим фактором в нашем программном обеспечении, а функциональное мышление — это инструмент для снижения сложности, как это согласуется с другими принципами — в частности, с *Правилами простого проектирования Кента Бека* (см. его книгу «Extreme Programming Explained: Embrace Change»)? Они хорошо служили нам в течение двух десятилетий и говорят о том, что простая конструкция:

- проходит тесты;
- раскрывает намерение;
- не содержит дублирования;
- имеет наименьшее количество элементов.

Из них «раскрывает намерение» наиболее открыто для интерпретации, так что давайте потянем за эту нить.

Намерение — это «цель или план» — оно подразумевает изменение. А следовательно, и действие. Подчеркивая в нашем коде различие между действиями и вычислениями, мы показываем места, где мы ожидаем, что что-то произойдет, а где — нет, на какие аспекты могут повлиять другие элементы, а на какие — нет. Когда большая часть нашего кода выполнена в виде вычислений, мы можем явно указать, какие функции являются действиями, что лучше раскрывает наши намерения.

Как показано в *главе 7 «От действий к вычислениям»* и в *главе 20 «От выполнения ввода/вывода к передаче данных»*, наш основной метод отделения вычислений от действий заключается в перемещении действий в точки входа наших взаимодействий, чтобы они захламляли наименьшее количество кода. Это не просто и не панацея, но мы ощущаем, что такой подход действительно позволяет создавать более простые проекты и менее сложный код.

Функциональное программирование и текстуальные рассуждения

Закончив эту книгу, мы, к своему удивлению, поняли, что не включили в нее никаких схем проектирования программного обеспечения.

Отчасти, честно говоря, этому способствовала лень. Достаточно сложно управлять несколькими версиями кода примера, когда он проходит через рефакторинг, не беспокоясь о других представлениях. Но у нас также вошло в привычку пытаться выражать свои желания лишь на том языке программирования, который у нас есть. Если мы сможем достичь достаточного понимания просто в необработанном тексте, то в нашей повседневной работе нам не придется переключать контексты для просмотра схемы, которая может синхронизироваться или не синхронизироваться с кодом.

Когда мы писали об объектно-ориентированном проектировании, то полагались на схемы, чтобы показать динамическую структуру и поведение программного обеспечения, а также то, как изменения в исходном коде влияют на его динамическое поведение. Но в объектно-ориентированном программном обеспечении эта динамическая структура — граф объектов и способов передачи сообщений между ними — в значительной степени неявна. Это затрудняет связь того, что вы видите в исходном коде, с тем, что произойдет во время выполнения, поэтому визуализация является жизненно важной частью объектно-ориентированного программирования. В 80–90-х годах прошлого века светила разработки программного обеспечения создавали различные виды схем для визуализации объектно-ориентированного программного обеспечения. В середине 1990-х годов разработчики самых популярных обозначений: Гради Буч, Ивар Якобсон и Джеймс Рамбо — объединенными усилиями создали *Унифицированный язык моделирования* (Unified Modeling Language, UML).

Сообщество функционального программирования не уделяет такого внимания диаграммам и визуализации. Целью функционального программирования является *алгебраическое рассуждение* — представление поведения программы путем манипулирования ее текстовыми выражениями. Ссылочная прозрачность и статические типы позволяют нам рассуждать о наших программах исключительно с помощью синтаксиса исходного кода. Это приводит к гораздо более тесному соответствию между исходным кодом и средой выполнения. По мере того как наш код становится более функциональным, мы обнаруживаем, что можем *читать* поведение нашей системы, не задумываясь о механизмах, которые не сразу видны в исходном коде и должны быть визуализированы для понимания.

Рефакторинг

Наряду с прагматичным функциональным программированием, рефакторинг является еще одним ключевым принципом этой книги. Рефакторинг играет важную роль в нашей профессиональной деятельности, потому что, если мы недостаточно знаем о конечной форме нашей системы, чтобы правильно спроектировать ее с самого начала, нам придется постепенно преобразовывать то, что у нас есть, в то, что нам нужно. Ваши авторы, по крайней мере, никогда не знали достаточно о конечной форме системы, чтобы правильно спроектировать ее с первого раза. Даже те приложения, в которых мы начинали с подробных требований, к моменту их ввода в эксплуатацию сильно отличались от исходных спецификаций.

На поздних стадиях проекта и под давлением графика нет времени учиться рефакторингу своего кода. Вместо этого мы используем любую возможность, чтобы практиковать рефакторинг. Как показано в *главе 22 «От классов к функциям»*, даже при написании кода с нуля мы часто жестко кодируем значения, чтобы тестирование прошло успешно, а затем проводим рефакторинг, чтобы устранить дублирование между тестовыми вариантами и рабочим кодом. Мы всегда ищем новые способы быстрого прохождения тестов, а затем реорганизуем наш способ в код, который выглядит так, как мы это планировали. Иногда мы используем новый автоматизированный рефакторинг, встроенный в IntelliJ. В других случаях находим способ объединить существующие способы рефакторинга для достижения наших целей.

Когда масштаб изменений невелик, можно обойтись ручным редактированием определения, а затем и его соответствующих применений, а иногда, что более полезно, наоборот. Однако работа над кодом становится утомительной, и он больше подвержен ошибкам, когда изменение затрагивает многие файлы. Поэтому практика использования инструментов рефракторинга для достижения даже небольших изменений поможет нам, когда мы столкнемся с более серьезными задачами рефакторинга. Там, где у нас есть многоступенчатый рефакторинг или где приходится вручную применять изменения в нескольких местах, «Рефракторинг с помощью Expand-and-Contract» (см. соответствующую врезку в *разд. «Рефракторинг от необязательных типов к обнуляемым» главы 4*) позволяет нам поддерживать построение и работу системы на протяжении всего процесса. Это жизненно важно, когда изменение может занять несколько дней или даже недель, потому что так можно постоянно объединять свою работу с другими изменениями в системе. Когда вы впустую потратили месяц работы из-за того, что слияние в стиле «большого взрыва» в конце концов оказалось невозможным, тогда начинаете ценить значимость этой техники и хотите практиковать ее, даже когда в этом нет острой необходимости.

Мы надеемся, что рефакторинг, описанный в этой книге, расширит ваши возможности. Вашим авторам посчастливилось работать со специалистами-практиками мирового класса, людьми, которые не одобрят, если по вашей вине во время рефракторинга возникнет ошибка компиляции. Преобразования, которые мы показали, могут быть неоптимальными (и даже если бы они были такими, самое современное

оборудование изменится с изменением инструментария и языка), но они являются подлинными и отражают то, как мы пишем и рефакторируем код.

Рефакторинг и функциональное мышление

Как мы видели на протяжении всего нашего путешествия, существует взаимосвязь между функциональным мышлением и рефакторингом. Рефакторинг — это перестроение нашего кода. И там, где этот код представляет действия (см. *разд. «Действия» главы 7*) — код, который зависит от того, когда вы его запускаете, — перестроение может измениться при выполнении действий, и, следовательно, изменится функционирование программного обеспечения. Напротив, вычисления (см. *разд. «Вычисления» главы 7*) можно безопасно переставлять, но в конечном счете они бессильны (без чтения и записи наш код представляет собой простой набор символов). Функциональное мышление побуждает нас распознавать действия и управлять ими и тем самым делает рефакторинг намного безопаснее.

Ваши авторы усвоили это на собственном горьком опыте. Мы учились проводить рефакторинг во времена мутирующих объектов и получали ошибки, когда не могли предсказать последствия. Это могло бы заставить нас отказаться от рефакторинга, но мы все еще не были достаточно умны, чтобы правильно спроектировать наши системы изначально. Вместо этого мы обнаружили, что определенный стиль программирования — объектная ориентация, но с неизменяемыми объектами, — оказался выразительным и понятным, поддающимся рефакторингу и безопасным. Когда мы внедрили этот стиль в наш Java-код, он часто работал вопреки здравому смыслу, но, несмотря на это, был намного более продуктивным, чем альтернативы. Открыв для себя Kotlin, мы поняли, что это лучшее место для нас. Теперь мы можем использовать современный язык, где функциональное мышление является частью конструкции, объекты по-прежнему хорошо поддерживаются, а применение инструментов рефакторинга не является запоздалой мыслью.

Как выразился Кент Бек: «Сделайте изменение легким, а затем внесите легкое изменение». Постоянно проводите рефакторинг, чтобы каждое изменение, которое вам нужно внести, было бы легким изменением. Рефакторинг — это фундаментальная практика для решения присущей нашему программному обеспечению сложности.

Счастливого пути.

Предметный указатель

М

О

П

Р

С

Т

Ф

Ч, Э

Об авторах

Нэт Прайс и **Дункан Макгрегор** имеют на двоих более чем пятидесятилетний профессиональный опыт в разработке программного обеспечения. Они написали много программ для различных отраслей промышленности. Они также много писали о разработке программного обеспечения, создавали программное обеспечение, чтобы помочь себе писать о разработке программного обеспечения, выступали с докладами о создании программного обеспечения, проводили семинары по разработке программного обеспечения на конференциях и помогали организовывать конференции по созданию программного обеспечения.

Большая часть программного обеспечения, которое они создали, была написана на Java. Не все, но многое. Они достаточно взрослые, чтобы помнить, когда Java была «глотком свежего воздуха» по сравнению с C++. И теперь они считают, что Kotlin — это «глоток свежего воздуха» по сравнению с Java. Поэтому они написали книгу о разработке программного обеспечения с помощью Kotlin (и создали некоторое программное обеспечение, которое помогло им ее написать).

Об изображении на обложке

На обложке этой книги изображена западноафриканская генетта (Genetta thierryi), также известная как ложная генетта. Генетты — это маленькие, похожие на кошек существа, обитающие в лесах, саваннах и кустарниковых зарослях от Гамбии до Камеруна. Их скорость и маневренность затрудняют наблюдение за ними в природе, но западноафриканские генетты были замечены в тропических лесах Сьерра-Леоне, Ганы и Кот-д'Ивуара, в степях Сенегала и в лесах Гвинеи-Бисау.

Западноафриканские генетты обычно светло-коричневые с ржавыми или черными пятнами на теле и полосами на спине. Большие треугольные уши и круглые глаза украшают их длинную и угловатую мордочку. Их хвосты такие же длинные, как и тело, и окаймлены темными кольцами. Разные виды генетт имеют схожий окрас, вырастают от 30 до 80 сантиметров и весят от 2 до 18 килограммов. Западноафриканские генетты находятся на меньшем конце этого диапазона как по размеру, так и по весу. Поскольку их весьма трудно изучать, мы мало что знаем об их поведении или питании. Другие виды генетт ведут ночной образ жизни, а днем спят в норах или дуплах деревьев. Ночью они охотятся на мелких млекопитающих и рептилий или добывают яйца, фрукты, насекомых и корни.

Западноафриканские генетты встречаются редко, но Международный союз охраны природы (МСОП) классифицирует их как виды, вызывающие «наименьшее беспокойство», из-за разнообразия мест обитаний и относительно широкого ареала, который они занимают в Западной Африке. Однако в прошлом на них охотились ради мяса или шкуры или отлавливали на продажу в качестве домашних животных.

Многие представители животного мира, изображенные на обложках книг издательства O'Reilly, находятся под угрозой исчезновения. Все они чрезвычайно важны для нашей планеты.

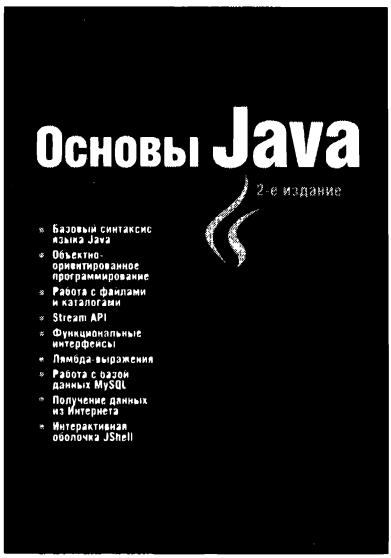
Иллюстрация на обложке выполнена Карен Монтгомери на основе черно-белой гравюры из книги «Animate Creation» Джона Вуда.

www.bhv.ru

Отдел оптовых поставок:
e-mail: opt@bhv.ru

Прохоренок Н.

Основы Java, 2-е изд.



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

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

Прохоренок Николай Анатольевич, профессиональный программист, автор книг «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Python 3. Самое необходимое», «Python 3 и PyQt 5. Разработка приложений», «OpenCV и Java. Обработка изображений и компьютерное зрение» и др.

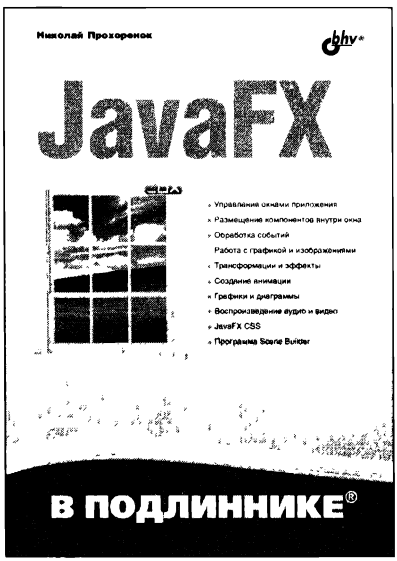
www.bhv.ru

Прохоренок Н.

JavaFX

Отдел оптовых поставок:
e-mail: opt@bhv.ru

Разработка оконных приложений на языке Java



- Управление окнами приложения
- Размещение компонентов внутри окна
- Обработка событий
- Работа с графикой и изображениями
- Трансформации и эффекты
- Создание анимации
- Графики и диаграммы
- Воспроизведение аудио и видео
- JavaFX CSS
- Программа Scene Builder

Описываются базовые возможности библиотеки JavaFX, позволяющей создавать приложения с графическим интерфейсом на языке Java. Книга ориентирована на тех, кто уже знаком с языком программирования Java и хотел бы научиться разрабатывать оконные приложения, насыщенные графикой, анимацией и интерактивными элементами.

Рассматриваются способы обработки событий, управление свойствами окна, создание формы с помощью программы Scene Builder, а также все основные компоненты (кнопки, текстовые поля, списки, таблицы, меню и др.) и варианты их размещения внутри окна.

Книга содержит большое количество практических примеров, помогающих начать разрабатывать приложения с графическим интерфейсом самостоятельно. Весь материал тщательно подобран, хорошо структурирован и компактно изложен, что позволяет использовать книгу как удобный справочник.

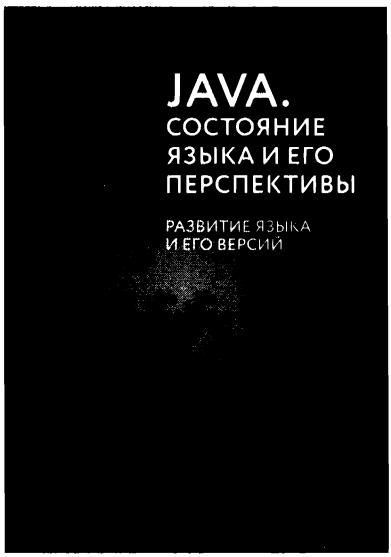
Прохоренок Николай Анатольевич, профессиональный программист, имеющий большой практический опыт создания и продвижения сайтов, анализа и обработки данных, автор книг «Основы Java», «OpenCV и Java. Обработка изображений и компьютерное зрение», «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Python 3. Самое необходимое», «Python 3 и PyQt 5. Разработка приложений» и др., многие из которых выдержали несколько переизданий и стали бестселлерами.

www.bhv.ru

Урванов Ф.

Java. Состояние языка и его перспективы

Отдел оптовых поставок:
e-mail: opt@bhv.ru



Книга о современном состоянии языка Java, векторе его развития, а также о грамотном программировании в духе паттернов GoF. Книга дает базовое представление о фреймворке Spring, контейнерах Docker, принципах ООП, затрагивая, в частности, переход к облачным решениям, обращение с IDE. Также освещены темы из enterprise-разработки: файловый ввод/вывод NIO.2, многопоточность, локализация, интеграция и оптимизация производительности. В книге отражено состояние языка по состоянию на версию Java 17 с разбором некоторых аспектов Java 18.

Будет интересна специалистам, возвращающимся к работе с Java после перерыва, бэкенд-разработчикам, читателям, готовящимся к сертификационным экзаменам.

Урванов Федор Владиславович, обладает более чем тринадцатилетним опытом профессионального программирования. Обладает сертификатами 1Z0-803 (Oracle Certified Associate, Java SE 7 Programmer) и 1Z0-804 (Oracle Certified Professional, Java SE 7 Programmer), 7 лет ведет блог по разработке программного обеспечения https://urvanov.ru, имеет богатый опыт коммерческой разработки на Java, участвовал в разработке программного обеспечения для нескольких крупнейших банков России.

O'REILLY®

От Java к Kotlin

Расстояние от острова Ява до острова Котлин составляет 9892 километра по прямой. Чтобы преодолеть его, не пользуясь воздушным транспортом, понадобится как минимум неделя. Если вы — опытный Java-разработчик, который решил перейти на Kotlin, вам потребуется примерно столько же времени. Вы обнаружите, что в Kotlin все делается по-другому, он требует иных подходов к программированию. Этот язык более функционален, у него больше возможностей, поэтому перенос кода ваших проектов с Java на Kotlin — отличная идея.

Ваши гиды Дункан и Нэт уже проделали этот путь в 2015 году, и с тех пор помогли многим командам и отдельным разработчикам повторить его. Путешествуйте вместе с ними по проверенному маршруту, включающему такие промежуточные остановки, как переход от объектов JavaBeans к значениям, от коллекций Java к коллекциям Kotlin и от классов Java к классам Kotlin. Авторы подробно объясняют ключевые концепции, а затем показывают, как постепенно и безопасно перенести код Java в идиоматичный Kotlin, сохраняя при этом совместимость. В результате код становится проще, выразительнее и удобнее для редактирования. К концу путешествия вы будете уверены в том, что сможете перевести все свои Java-проекты на платформу Kotlin, освоите Kotlin с нуля и научитесь управлять смешанной языковой базой кода по мере ее развития.

Дункан Макгрегор и **Нэт Прайс** — опытные разработчики, консультанты и преподаватели. Ранние последователи как Java, так и Kotlin, они научились сочетать методы объектно-ориентированного и функционального программирования, а также рефакторинга между этими языками. Их успешные семинары на KotlinConf доказали ценность обучения через рефакторинг и привели к написанию этой книги.

Взять знакомый всем старый добрый код Java и наблюдать за тем, как он постепенно превращается в лаконичный, четкий, выразительный и легкий в обслуживании код на языке Kotlin — замечательный способ выучить язык. Воспользуйтесь опытом, которым делятся Макгрегор и Прайс.

— **Венкат Субраманиам,** Ph.D., писатель и основатель компании Agile Developer, Inc.

Самый быстрый способ перенести свои навыки и умения из Java в Kotlin. Обязательно к прочтению любому профессиональному разработчику на Java.

— **Дон и Дэвид Гриффитс,** авторы книг «React. Сборник рецептов» и «Head First. Kotlin»

ISBN 978-5-9775-6841-8

9 785977 568418

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

bhv®