Kotlin в действии

Дмитрий Жемеров
Светлана Исакова

ДМК
MANNING

Жемеров Дмитрий, Исакова Светлана

Kotlin в действии

Kotlin in Action

DMITRY JEMEROV
AND SVETLANA ISAKOVA

MANNING
SHELTER ISLAND

Kotlin в действии

ЖЕМЕРОВ ДМИТРИЙ
ИСАКОВА СВЕТЛАНА

ДМК Пресс
издательство
Москва, 2018

УДК 004.438Kotlin
ББК 32.973.26-018.1
Ж53

Ж53 Жемеров Д., Исакова С.
Kotlin в действии. / пер. с англ. Киселев А. Н. – М.: ДМК Пресс, 2018. – 402 с.: ил.

ISBN 978-5-97060-497-7

Язык Kotlin предлагает выразительный синтаксис, мощную и понятную систему типов, великолепную поддержку и бесшовную совместимость с существующим кодом на Java, богатый выбор библиотек и фреймворков. Kotlin может компилироваться в байт-код Java, поэтому его можно использовать везде, где используется Java, включая Android. А благодаря эффективному компилятору и маленькой стандартной библиотеке Kotlin практически не привносит накладных расходов.

Данная книга научит вас пользоваться языком Kotlin для создания высококачественных приложений. Написанная создателями языка – разработчиками в компании JetBrains, – эта книга охватывает такие темы, как создание предметно-ориентированных языков, функциональное программирование в JVM, совместное использование Java и Kotlin и др.

Издание предназначено разработчикам, владеющим языком Java и желающим познакомиться и начать эффективно работать с Kotlin.

ISBN 978-1-61729-329-0 (англ.)
ISBN 978-5-97060-497-7 (рус.)

Оглавление

Предисловие

Впервые оказавшись в JetBrains весной 2010 года, я был абсолютно уверен, что мир не нужен еще один язык программирования общего назначения. Я полагал, что существующие JVM-языки достаточно хороши, да и кто в здравом уме станет создавать новый язык? Примерно после часа обсуждения проблем разработки крупномасштабных программных продуктов мое мнение изменилось, и я набросал на доске первые идеи, которые позже стали частью языка Kotlin. Вскоре после этого я присоединился к JetBrains для проектирования языка и работы над компилятором.

Сегодня, спустя шесть лет, мы приближаемся к выпуску второй версии. Сейчас в команде работает более 30 человек, а у языка появились тысячи активных пользователей, и у нас осталось еще множество потрясающих идей, которые с трудом укладываются в моей голове. Но не волнуйтесь: прежде чем стать частью языка, эти идеи подвергнутся тщательной проверке. Мы хотим, чтобы в будущем описание языка Kotlin по-прежнему могло уместиться в одну не слишком большую книгу.

Изучение языка программирования – это захватывающее и часто очень полезное занятие. Если это ваш первый язык, с ним вы откроете целый новый мир программирования. Если нет, он заставит вас по-другому думать о знакомых вещах и глубже осознать их на более высоком уровне абстракции. Эта книга предназначена в основном для последней категории читателей, уже знакомых с Java.

Проектирование языка с нуля – это сама по себе сложная задача, но обеспечение взаимодействия с другим языком – это совсем другая история, полная злых огров и мрачных подземелий. (Если не верите мне – спросите Бьярне Страуструпа (Bjarne Stroustrup), создателя C++.) Совместимость с Java (то есть возможность смешивать код на Java и Kotlin и вызывать один из другого) стала одним из краеугольных камней Kotlin, и эта книга уделяет большое внимание данному аспекту. Взаимодействие с Java очень важно для постепенного внедрения Kotlin в существующие проекты на Java. Даже создавая проект с нуля, важно учитывать, как язык вписывается в общую картину платформы со всем многообразием библиотек, написанных на Java.

В данный момент, когда я пишу эти строки, разрабатываются две новые платформы: Kotlin уже запускается на виртуальных машинах JavaScript, обеспечивая поддержку разработки всех уровней веб-приложения, и скоро его можно будет компилировать прямо в машинный код и запускать без виртуальной машины. Хотя эта книга ориентирована на JVM, многое из того, что вы узнаете, может быть использовано в других средах выполнения.

Авторы присоединились к команде Kotlin с самых первых дней, поэтому они хорошо знакомы с языком и его внутренним устройством. Благодаря опыту выступления на конференциях и проведению семинаров и курсов о Kotlin авторы смогли сформировать хорошие объяснения, предвосхищающие самые распространенные вопросы и возможные подводные камни. Книга объясняет высокоуровневые понятия языка во всех необходимых подробностях.

Надеюсь, вы хорошо проведете время с книгой, изучая наш язык. Как я часто говорю в сообщениях нашего сообщества: «Хорошего Kotlin!»

Андрей Бреслав,
ведущий разработчик языка Kotlin в JetBrains

Вступление

Идея создания Kotlin зародилась в JetBrains в 2010 году. К тому времени компания уже была признанным производителем инструментов разработки для множества языков, включая Java, C#, JavaScript, Python, Ruby и PHP. IntelliJ IDEA – наш флагманский продукт, интегрированная среда разработки (IDE) для Java – также включала плагины для Groovy и Scala.

Опыт разработки инструментария для такого разнообразного набора языков позволил нам получить уникальное понимание процесса проектирования языков целом и взглянуть на него под другим углом. И все же интегрированные среды разработки на платформе IntelliJ, включая IntelliJ IDEA, по-прежнему разрабатывались на Java. Мы немного завидовали нашим коллегам из команды .NET, которые вели разработку на C# – современном, мощном, быстро развивающемся языке. Но у нас не было языка, который мы могли бы использовать вместо Java.

Какими должны быть требования к такому языку? Первое и самое очевидное – статическая типизация. Мы не знаем другого способа разрабатывать проекты с миллионами строк кода на протяжении многих лет, не сходя при этом с ума. Второе – полная совместимость с существующим кодом на Java. Этот код является чрезвычайно ценным активом компании JetBrains, и мы не могли себе позволить потерять или обесценить его из-за проблем совместимости. В-третьих, мы не хотели идти на компромиссы с точки зрения качества инструментария. Производительность разработчиков – самая главная ценность компании JetBrains, а для её достижения нужен хороший инструментарий. Наконец, нам был нужен язык, простой в изучении и применении.

Когда мы видим в своей компании неудовлетворенную потребность, мы знаем, что есть и другие компании, оказавшиеся в подобной ситуации, и что наше решение найдет много пользователей за пределами JetBrains. Учитывая это, мы решили начать проект разработки нового языка: Kotlin. Как это часто бывает, проект занял больше времени, чем мы ожидали, и Kotlin 1.0 вышел больше чем через пять лет после того, как в репозитории появился первый фрагмент его реализации; но теперь мы уверены, что язык нашел своих пользователей и будет использоваться в дальнейшем.

Язык Kotlin назван в честь острова Котлин неподалеку от Санкт-Петербурга в России, где живет большинство разработчиков Kotlin. Выбрав для названия языка имя острова, мы последовали прецеденту, созданному языками Java и Ceylon, но решили найти что-то ближе к нашему дому.

По мере приближения к выпуску первой версии языка мы поняли, что нам не помешала бы книга про Kotlin, написанная людьми, которые участвовали в принятии проектных решений и могут уверенно объяснить, почему Kotlin устроен так, а не иначе. Данная книга – результат совместных усилий этих людей, и мы надеемся, что она поможет вам узнать и понять язык Kotlin. Удачи вам, и программируйте с удовольствием!

Благодарности

Прежде всего мы хотим поблагодарить Сергея Дмитриева и Максима Шафирова за веру в идею нового языка и решение вложить средства JetBrains в его разработку. Без них не было бы ни языка, ни этой книги.

Мы хотели бы особо поблагодарить Андрея Бреслава – главного виновника, что язык спроектирован так, что писать про него одно удовольствие (впрочем, как и программировать на нем). Несмотря на занятость управлением постоянно растущей командой Kotlin, Андрей смог сделать много полезных замечаний, что мы очень высоко ценим. Более того, вы можете быть уверены, что книга получила одобрение от ведущего разработчика языка в виде предисловия, которое он любезно написал.

Мы благодарны команде издательства Manning, которая помогла нам написать книгу и сделать ее легко читаемой и хорошо структурированной. В частности, мы хотим поблагодарить редактора-консультанта Дэна Махари (Dan Maharry), который всегда стремился найти время для обсуждения, несмотря на наш напряженный график, а также Майкла Стивенса (Michael Stephens), Хелен Стергиус (Helen Stergius), Кевина Салливана (Kevin Sullivan), Тиффани Тейлор (Tiffany Taylor), Элизабет Мартин (Elizabeth Martin) и Марию Тюдор (Marija Tudor). Отзывы наших технических редакторов Брента Уотсона (Brent Watson) и Игоря Войды (Igor Wojda) оказались просто бесценны, так же как замечания рецензентов, читавших рукопись в процессе работы: Алессандро Кампеи (Alessandro Campeis), Амита Ламба (Amit Lamba), Анджело Косты (Angelo Costa), Бориса Василе (Boris Vasile), Брендана Грейнджера (Brendan Grainger), Кальвина Фернандеса (Calvin Fernandes), Кристофера Бейли (Christopher Bailey), Кристофера Борца (Christopher Bortz), Конора Редмонда (Conor Redmond), Дилана Скотта (Dylan Scott), Филипа Правика (Filip Pravica), Джейсона Ли (Jason Lee), Джастина Ли (Justin Lee), Кевина Орра (Kevin Orr), Николаса Франкеля (Nicolas Frankel), Павла Гайды (Paweł Gajda), Рональда Тишлера (Ronald Tischliar) и Тима Лаверса (Tim Laver).

Также благодарим всех, кто отправил свои предложения в рамках MEAP (Manning Early Access Program – программы раннего доступа Manning) на форуме книги; ваши комментарии помогли улучшить текст книги.

Мы благодарны всем участникам команды Kotlin, которым приходилось выслушивать ежедневные заявления вроде: «Еще один раздел закончен!» – на протяжении всего периода написания этой книги. Мы хотим поблагодарить наших коллег, которые помогли составить план книги и оставляли отзывы о её ранних вариантах, особенно Илью Рыженкова, Хади Харири (Hadi Hariri), Михаила Глухих и Илью Горбунова. Мы также хотим поблагодарить друзей, которые не только поддерживали нас, но и читали текст книги и оставляли отзывы о ней (иногда на горнолыжных курортах во время отпуска): Льва Серебрякова, Павла Николаева и Алису Афонину.

Наконец, мы хотели бы поблагодарить наши семьи и котов за то, что они делают этот мир лучше.

Об этой книге

Книга «*Kotlin в действии*» расскажет о языке Kotlin и как писать на нем приложения для виртуальной машины Java и Android. Она начинается с обзора основных особенностей языка Kotlin, постепенно раскрывая наиболее отличительные аспекты, такие как поддержка создания высокоуровневых абстракций и предметно-ориентированных языков (Domain-Specific Languages, DSL). Книга уделяет большое внимание интеграции Kotlin с существующими проектами на языке Java и поможет вам внедрить Kotlin в текущую рабочую среду.

Книга описывает версию языка Kotlin 1.0. Версия Kotlin 1.1 разрабатывалась параллельно с написанием книги, и, когда это было возможно, мы упоминали об изменениях в версии 1.1. Но поскольку на момент написания книги новая версия еще не была готова, мы не могли полностью охватить все нововведения. За более подробной информацией о новых возможностях и изменениях обращайтесь к документации по адресу: `https://kotlinlang.org`.

Кому адресована эта книга

Книга «*Kotlin в действии*» адресована в первую очередь разработчикам с опытом программирования на языке Java. Kotlin во многом основан на понятиях и приёмах языка Java, и с помощью этой книги вы быстро освоите его, используя имеющиеся знания. Если вы только начали изучать Java или владеете другими языками программирования, такими как C# или JavaScript, вам может понадобиться обратиться к другим источникам информации, чтобы понять наиболее сложные аспекты взаимодействия Kotlin с JVM, но вы все равно сможете изучить Kotlin, читая эту книгу. Мы описываем язык Kotlin в целом, не привязываясь к конкретной предметной области, поэтому книга должна быть одинаково полезна и для разработчиков серверных приложений, и для разработчиков на Android, и для всех, кто создает проекты для JVM.

Как организована эта книга

Книга делится на две части. *Часть 1* объясняет, как начать использовать Kotlin вместе с существующими библиотеками и API:

- *глава 1* рассказывает о ключевых целях, ценностях и областях применения языка и показывает различные способы запуска кода на Kotlin;
- *глава 2* демонстрирует важные элементы любой программы на языке Kotlin, включая управляющие структуры, переменные и функции;

- *в главе 3* подробно рассматриваются объявления функций в Kotlin, а также вводятся понятия функций-расширений (extension functions) и свойств-расширений (extension properties);
- *глава 4* посвящена объявлению классов и знакомит с понятиями классов данных (data classes) и объектов-компаньонов (companion objects);
- *глава 5* знакомит с лямбда-выражениями и демонстрирует ряд примеров их использования в стандартной библиотеке Kotlin;
- *глава 6* описывает систему типов Kotlin, обращая особое внимание на работу с типами, допускающими значения null, и с коллекциями.

Часть 2 научит вас создавать собственные API и абстракции на языке Kotlin и охватывает некоторые более продвинутые особенности языка:

- *глава 7* рассказывает о соглашениях, придающих особый смысл методам и свойствам с определенными именами, и вводит понятие делегируемых свойств (delegated properties);
- *глава 8* показывает, как объявлять функции высшего порядка – функции, принимающие другие функции или возвращающие их. Здесь также вводится понятие встраиваемых функций (inline functions);
- *глава 9* глубоко погружается в тему обобщенных типов (generics), начиная с базового синтаксиса и переходя к более продвинутым темам, таким как овеществляемые типовые параметры (reified type parameters) и вариантность;
- *глава 10* посвящена использованию аннотаций и механизма рефлексии (reflection) и организована вокруг JKid – простой библиотеки сериализации в формат JSON, которая интенсивно использует эти понятия;
- *глава 11* вводит понятие предметно-ориентированного языка (DSL), описывает инструменты Kotlin для их создания и демонстрирует множество примеров DSL.

В книге есть три приложения. Приложение A объясняет, как выполнять сборку проектов на Kotlin с помощью Gradle, Maven и Ant. Приложение B фокусируется на документирующих комментариях и создании документации с описанием API модулей. Приложение C является руководством по экосистеме Kotlin и поиску актуальной информации в Интернете.

Книгу лучше читать последовательно, от начала до конца, но также можно обращаться к отдельным главам, посвященным интересующим вам конкретным темам, и переходить по перекрестным ссылкам для уточнения незнакомых понятий.

Соглашения об оформлении программного кода и загружаемые ресурсы

В книге приняты следующие соглашения:

- *Курсивом* обозначаются новые термины.
- `Моноширинным шрифтом` выделены фрагменты кода, имена классов и функций и другие идентификаторы.
- Примеры кода сопровождаются многочисленными примечаниями, подчеркивающими важные понятия.

Многие листинги кода в книге показаны вместе с его выводом. В таких случаях строки кода, производящие вывод, начинаются с префикса >>>, как показано ниже:

```
>>> println("Hello World")
Hello World
```

Некоторые примеры являются полноценными программами, тогда как другие – лишь фрагменты, демонстрирующие определенные понятия и могущие содержать сокращения (обозначенные как ...) или синтаксические ошибки (описанные в тексте книги или самих примерах). Выполняемые примеры можно загрузить в виде zip-архива на сайте издательства www.manning.com/books/kotlin-in-action. Примеры из книги также выгружены в онлайн-окружение http://try.kotlinlang.org, где вы сможете опробовать любой пример, сделав всего несколько щелчков в окне браузера.

Авторы онлайн

Покупка книги «*Kotlin в действии*» дает право свободного доступа к закрытому веб-форуму издательства Manning Publication, где можно высказать свои замечания о книге, задать технические вопросы и получить помощь от авторов и других пользователей. Чтобы получить доступ к форуму, перейдите по ссылке www.manning.com/books/kotlin-in-action. На этой странице описывается, как попасть на форум после регистрации, какая помощь доступна и какие правила поведения действуют на форуме.

Издательство Manning обязуется предоставить читателям площадку для содержательного диалога не только между читателями, но и между читателями и авторами. Но авторы не обязаны выделять определённое количество времени для участия, т. к. их вклад в работу форума является добровольным (и неоплачиваемым). Мы предлагаем читателям задавать авторам действительно непростые вопросы, чтобы их интерес не угасал!

Прочие онлайн-ресурсы

Kotlin имеет активное сообщество, поэтому имеющие вопросы или желающие пообщаться с другими пользователями Kotlin могут воспользоваться следующими ресурсами:

- официальный форум Kotlin – https://discuss.kotlinlang.org;
- чат Slack – http://kotlinlang.slack.com (вы можете получить приглашение по ссылке http://kotlinslackin.herokuapp.com/);
- вопросы и ответы с тегом Kotlin на Stack Overflow – http://stackoverflow.com/questions/tagged/kotlin;
- Kotlin на форуме Reddit – http://www.reddit.com/r/Kotlin.

Об авторах

Дмитрий Жемеров работает в компании JetBrains с 2003 года и принимал участие в разработке многих продуктов, в том числе IntelliJ IDEA, PyCharm и WebStorm. Был одним из первых участников команды Kotlin, создал начальную версию генератора байт-кода JVM из кода Kotlin и сделал много презентаций о языке Kotlin на различных встречах по всему миру. Сейчас возглавляет команду, работающую над плагином Kotlin IntelliJ IDEA.

Светлана Исакова вошла в состав команды Kotlin в 2011 году. Работала над механизмом вывода типов (type inference) и подсистемой компилятора по разрешению перегруженных имен. Сейчас она – технический евангелист, рассказывает о Kotlin на конференциях и разрабатывает онлайн-курс по языку Kotlin.

Об изображении на обложке

Иллюстрация на обложке книги «*Kotlin в действии*» называется «Одежда русской женщины на Валдае в 1764 году». Город Валдай находится в Новгородской области, между Москвой и Санкт-Петербургом. Иллюстрация взята из работы Томаса Джефферис (Thomas Jefferys) «Коллекция платьев разных народов, древних и современных», опубликованной в Лондоне между 1757 и 1772 г.. На титульной странице говорится, что это – раскрашенная вручную гравюра, обработанная гуммиарабиком для повышения яркости. Томаса Джефферис (1719–1771) называли географом короля Георга III. Он был английским картографом и ведущим поставщиком карт своего времени. Он гравировал и печатал карты для правительства и других официальных органов, выпускал широкий спектр коммерческих карт и атласов, особенно Северной Америки. Работа картографом пробудила интерес к местным традиционным нарядам в землях, которые он исследовал и картографировал; эти наряды великолепно представлены в четырехтомном сборнике.

Очарование дальними странами и путешествия для удовольствия были относительно новым явлением в восемнадцатом веке, и такие коллекции, как эта, были популярны и показывали туристам и любителям книг о путешествиях жителей других стран. Разнообразие рисунков в книгах Джефферис красноречиво говорит об уникальности и индивидуальности народов мира много веков назад. С тех пор стиль одежды сильно изменился, и разнообразие, характеризующее различные области и страны, исчезло. Сейчас часто трудно отличить даже жителей одного континента от дру-

гого. Возможно, с оптимистической точки зрения, мы обменяли культурное и визуальное разнообразие на более разнообразную частную жизнь или более разнообразную и интересную интеллектуальную и техническую деятельность.

В наше время, когда трудно отличить одну компьютерную книгу от другой, издательство Manning с инициативой и находчивостью наделяет книги обложками, изображающими богатое разнообразие жизненного уклада народов многовековой давности, давая новую жизнь рисункам Джефферис.

Часть 1

Введение в Kotlin

Цель этой части книги – помочь начать продуктивно писать код на языке Kotlin, используя существующие API. Глава 1 познакомит вас с языком Kotlin в общих чертах. В главах 2–4 вы узнаете, как в Kotlin реализованы основные понятия языка Java – операторы, функции, классы и типы, – и как Kotlin обогащает их, делая программирование более приятным. Вы сможете положиться на имеющиеся знания языка Java, а также на вспомогательные инструменты, входящие в состав интегрированной среды разработки, и конвертер кода на Java в код на Kotlin, чтобы быстро начать писать код. В главе 5 вы узнаете, как лямбда-выражения помогают эффективно решать некоторые из распространенных задач программирования, такие как работа с коллекциями. Наконец, в главе 6 вы познакомитесь с одной из ключевых особенностей Kotlin – поддержкой операций со значениями `null`.

Глава 1

Kotlin: что это и зачем

В этой главе:

- общий обзор языка Kotlin;
- основные особенности;
- возможности разработки для Android и серверных приложений;
- отличие Kotlin от других языков;
- написание и выполнение кода на языке Kotlin.

Что же такое Kotlin? Это новый язык программирования для платформы Java. Kotlin – лаконичный, безопасный и прагматичный язык, совместимый с Java. Его можно использовать практически везде, где применяется Java: для разработки серверных приложений, приложений для Android и многого другого. Kotlin прекрасно работает со всеми существующими библиотекам и фреймворками, написанными на Java, не уступая последнему в производительности. В этой главе мы подробно рассмотрим основные черты языка Kotlin.

1.1. Знакомство с Kotlin

Начнем с небольшого примера для демонстрации языка Kotlin. В этом примере определяется класс `Person`, создается коллекция его экземпляров, выполняется поиск самого старого и выводится результат. Даже в этом маленьком фрагменте кода можно заметить множество интересных особенностей языка Kotlin; мы выделили некоторые из них, чтобы вам проще было отыскать их в книге в будущем. Код пояснен довольно кратко, но не беспокойтесь, если что-то останется непонятным. Позже мы подробно всё обсудим.

Желающие опробовать этот пример могут воспользоваться онлайн-полигоном по адресу: `http://try.kotl.in`. Введите пример, щелкните на кнопке **Run** (Запустить), и код будет выполнен.

Листинг 1.1. Первое знакомство с Kotlin

```
data class Person(val name: String,                    <— Класс «данных»
                  val age: Int? = null)                <— Тип, допускающий значение null (Int?);
                                                          значение параметра по умолчанию

fun main(args: Array<String>) {                        <— Функция верхнего уровня
  val persons = listOf(Person("Alice"),
                       Person("Bob", age = 29))        <— Именованный аргумент

  val oldest = persons.maxBy { it.age ?: 0 }           <— Лямбда-выражение; оператор «Элвис»
  println("The oldest is: $oldest")                    <— Строка-шаблон
}

// The oldest is: Person(name=Bon, age=29)             <— Часть вывода автоматически сгенерирована
                                                          методом toString
```

Здесь объявляется простой класс данных с двумя свойствами: `name` и `age`. Свойству `age` по умолчанию присваивается значение `null` (если оно не задано). При создании списка людей возраст Алисы не указывается, поэтому он принимает значение `null`. Затем, чтобы отыскать самого старого человека в списке, вызывается функция `maxBy`. Лямбда-выражение, которое передается функции, принимает один параметр с именем `it` по умолчанию. *Оператор «Элвис»* (`?:`) возвращает ноль, если возраст имеет значение `null`. Поскольку возраст Алисы не указан, оператор «Элвис» заменит его нулем, поэтому Боб получит приз как самый старый человек.

Вам понравилось? Читайте дальше, чтобы узнать больше и стать экспертом в языке Kotlin. Мы надеемся, что скоро вы увидите такой код в своих проектах, а не только в этой книге.

1.2. Основные черты языка Kotlin

Возможно, у вас уже есть некоторое представление о языке Kotlin. Давайте подробнее рассмотрим его ключевые особенности. Для начала определим типы приложений, которые можно создавать с его помощью.

1.2.1. Целевые платформы: серверные приложения, Android и везде, где запускается Java

Основная цель языка Kotlin – предложить более компактную, производительную и безопасную альтернативу языку Java, пригодную для использования везде, где сегодня применяется Java. Java – чрезвычайно популярный язык, который используется в самых разных окружениях, начиная от смарт-карт (технология Java Card) до крупнейших вычислительных центров таких компаний, как Google, Twitter и LinkedIn. В большинстве таких окружений применение Kotlin способно помочь разработчикам достигать своих целей меньшим объемом кода и избегая многих неприятностей.

Наиболее типичные области применения Kotlin:

- разработка кода, работающего на стороне сервера (как правило, серверной части веб-приложений);
- создание приложений, работающих на устройствах Android.

Но Kotlin работает также в других областях. Например, код на Kotlin можно выполнять на устройствах с iOS, используя технологию Intel Multi-OS Engine (https://software.intel.com/en-us/multi-os-engine). На Kotlin можно писать и настольные приложения, используя его совместно с TornadoFX (https://github.com/edvin/tornadofx) и JavaFX[1] .

Помимо Java, код на Kotlin можно скомпилировать в код на JavaScript и выполнять его в браузере. Но на момент написания этой книги поддержка JavaScript находилась в стадии исследования и прототипирования, поэтому она осталась за рамками данной книги. В будущих версиях языка также рассматривается возможность поддержки других платформ.

Как видите, область применения Kotlin достаточно обширна. Kotlin не ограничивается одной предметной областью или одним типом проблем, с которыми сегодня сталкиваются разработчики программного обеспечения. Вместо этого он предлагает всестороннее повышение продуктивности при решении любых задач, возникающих в процессе разработки. Он также предоставляет отличный уровень интеграции с библиотеками, созданными для поддержки определенных предметных областей или парадигм программирования. Давайте рассмотрим основные качества Kotlin как языка программирования.

1.2.2. Статическая типизация

Так же, как Java, Kotlin – *статически типизированный* язык программирования. Это означает, что тип каждого выражения в программе известен во время компиляции, и компилятор может проверить, что методы и поля, к которым вы обращаетесь, действительно существуют в используемых объектах.

Этим Kotlin отличается от *динамически типизированных* (dynamically typed) языков программирования на платформе JVM, таких как Groovy и JRuby. Такие языки позволяют определять переменные и функции, способные хранить или возвращать данные любого типа, а ссылки на поля и методы определяются во время выполнения. Это позволяет писать более компактный код и дает большую гибкость в создании структур данных. Но в языках с динамической типизацией есть свои недостатки: например, опечатки в именах нельзя обнаружить во время компиляции, что влечет появление ошибок во время выполнения.

---

[1] «JavaFX: Getting Started with JavaFX», Oracle, http://mng.bz/500y.

С другой стороны, в отличие от Java, Kotlin не требует явно указывать тип каждой переменной. В большинстве случаев тип переменной может быть определен автоматически. Вот самый простой пример:

```
val x = 1
```

Вы объявляете переменную, но поскольку она инициализируется целочисленным значением, Kotlin автоматически определит её тип как `Int`. Способность компилятора определять типы из контекста называется *выведением типа* (type inference).

Ниже перечислены некоторые преимущества статической типизации:

- *Производительность* – вызов методов происходит быстрее, поскольку во время выполнения не нужно выяснять, какой метод должен быть вызван.
- *Надежность* – корректность программы проверяется компилятором, поэтому вероятность ошибок во время выполнения меньше.
- *Удобство сопровождения* – работать с незнакомым кодом проще, потому что сразу видно, с какими объектами код работает.
- *Поддержка инструментов* – статическая типизация позволяет увереннее выполнять рефакторинг, обеспечивает точное автодополнение кода и поддержку других возможностей IDE.

Благодаря поддержке выведения типов в Kotlin исчезает излишняя избыточность статически типизированного кода, поскольку больше не нужно объявлять типы явно.

Система типов в Kotlin поддерживает много знакомых понятий. Классы, интерфейсы и обобщенные типы работают практически так же, как в Java, так что большую часть своих знаний Java вы с успехом сможете применить на Kotlin. Однако есть кое-что новое.

Наиболее важным нововведением в Kotlin является поддержка типов, допускающих значения `null` (nullable types), которая позволяет писать более надежные программы за счет выявления потенциальных ошибок обращения к пустому указателю на этапе компиляции. Мы ещё вернемся к типам, допускающим значение `null`, далее в этой главе и подробно обсудим их в главе 6.

Другим новшеством в системе типов Kotlin является поддержка *функциональных типов* (function types). Чтобы понять, о чем идет речь, обратимся к основным идеям функционального программирования и посмотрим, как они поддерживаются в Kotlin.

1.2.3. Функциональное и объектно-ориентированное программирование

Как Java-разработчик вы, без сомнения, знакомы с основными понятиями объектно-ориентированного программирования, но функциональное

программирование может оказаться для вас в новинку. Ниже перечислены ключевые понятия функционального программирования:

- *Функции как полноценные объекты* – с функциями (элементами поведения) можно работать как со значениями. Их можно хранить в переменных, передавать в аргументах или возвращать из других функций.
- *Неизменяемость* – программные объекты никогда не изменяются, что гарантирует неизменность их состояния после создания.
- *Отсутствие побочных эффектов* – функции всегда возвращают один и тот же результат для тех же аргументов, не изменяют состояние других объектов и не взаимодействуют с окружающим миром.

Какие преимущества дает функциональный стиль? Во-первых, *лаконичность*. Функциональный код может быть более элегантным и компактным, по сравнению с императивными аналогами, потому что возможность работать с функциями как со значениями дает возможность создавать более мощные абстракции, позволяющие избегать дублирования в коде.

Представьте, что у вас есть два похожих фрагмента кода, решающих аналогичную задачу (например, поиск элемента в коллекции), которые отличаются в деталях (способом проверки критериев поиска). Вы легко сможете перенести общую логику в функцию, передавая отличающиеся части в виде аргументов. Эти аргументы сами будут функциями, но вы сможете описать их, используя лаконичный синтаксис анонимных функций, называемых *лямбда-выражениями*:

```
fun findAlice() = findPerson { it.name == "Alice" }
fun findBob() = findPerson { it.name == "Bob" }
```

Функция findPerson() описывает общую логику поиска

Блок кода в фигурных скобках задает свойства искомого элемента

Второе преимущество функциональной парадигмы – *безопасное многопоточное программирование*. Одним из основных источников ошибок в многопоточных программах является модификация одних и тех же данных из нескольких потоков без надлежащей синхронизации. Используя неизменяемые структуры данных и чистые функции, можно не опасаться никаких изменений и не надо придумывать сложных схем синхронизации.

Наконец, функциональное программирование *облегчает тестирование*. Функции без побочных эффектов можно проверять по отдельности, без необходимости писать много кода для настройки окружения.

В целом функциональную парадигму можно использовать в любом языке программирования, включая Java, и многие ее аспекты считаются хорошим стилем программирования. Но не все языки поддерживают соответствующий синтаксис и библиотеки, упрощающие применение этого стиля; например, такая поддержка отсутствовала в Java до версии Java 8.

Язык Kotlin изначально обладает богатым арсеналом возможностей для поддержки функционального программирования. К ним относятся:

- *функциональные типы,* позволяющие функциям принимать или возвращать другие функции;
- *лямбда-выражения,* упрощающие передачу фрагментов кода;
- *классы данных,* предоставляющие емкий синтаксис для создания неизменяемых объектов-значений;
- обширный набор средств в стандартной библиотеке для работы с объектами и коллекциями в функциональном стиле.

Kotlin позволяет программировать в функциональном стиле, но не требует этого. Когда нужно, вы можете работать с изменяемыми данными и писать функции с побочными эффектами без всяких затруднений. Работать с фреймворками, основанными на иерархиях классов и интерфейсах, так же легко, как на языке Java. В программном коде на Kotlin вы можете совмещать объектно-ориентированный и функциональный подходы, используя для каждой решаемой проблемы наиболее подходящий инструмент.

1.2.4. Бесплатный язык с открытым исходным кодом

Язык Kotlin, включая компилятор, библиотеки и все связанные с ними инструменты, – это проект с открытым исходным кодом, который может свободно применяться для любых целей. Он доступен на условиях лицензии Apache 2, разработка ведется открыто в GitHub (`http://github.com/jetbrains/kotlin`), и любой добровольный вклад приветствуется. Также на выбор есть три IDE с открытым исходным кодом, поддерживающих разработку приложений на Kotlin: IntelliJ IDEA Community Edition, Android Studio и Eclipse. (Конечно же, поддержка Kotlin имеется также в IntelliJ IDEA Ultimate.)

Теперь, когда вы получили некоторое представление о Kotlin, пришла пора узнать, как использовать его преимущества для конкретных практических приложений.

1.3. Приложения на Kotlin

Как мы уже упоминали ранее, основные области применения Kotlin – это создание серверной части приложений и разработка для Android. Рассмотрим эти области по очереди и разберемся, почему Kotlin так хорошо для них подходят.

1.3.1. Kotlin на сервере

Серверное программирование – довольно широкое понятие. Оно охватывает все следующие типы приложений и многое другое:

- веб-приложения, возвращающие браузеру страницы HTML;
- серверные части мобильных приложений, открывающие доступ к своему JSON API по протоколу HTTP;
- микрослужбы, взаимодействующие с другими микрослужбами посредством RPC.

Разработчики много лет создавали эти типы приложений на Java и накопили обширный набор фреймворков и технологий, облегчающих их создание. Подобные приложения, как правило, не разрабатываются изолированно и не пишутся с нуля. Почти всегда есть готовая система, которую нужно расширять, улучшать или заменять, и новый код должен интегрироваться с существующими частями системы, которые могли быть написаны много лет назад.

Большим преимуществом Kotlin в подобной среде является его взаимодействие с существующим Java-кодом. Kotlin отлично подходит и для создания новых компонентов, и для переноса кода существующей службы на Kotlin. Вы не столкнетесь с затруднениями, когда коду на Kotlin понадобится унаследовать Java-классы или отметить специальными аннотациями методы и поля класса. Преимущество же заключается в том, что код системы станет более компактным, надежным и простым в обслуживании.

В то же время Kotlin предлагает ряд новых приемов для создания таких систем. Например, его поддержка шаблона «Строитель» (Builder) позволяет создавать произвольные графы объектов, используя очень лаконичный синтаксис, не отказываясь при этом от полного набора абстракций и инструментов повторного использования кода в языке.

Один из простейших примеров использования этой особенности – библиотека создания разметки HTML: лаконичное и полностью типобезопасное решение, которое может полностью заменить сторонний язык шаблонов. Например:

```
fun renderPersonList(persons: Collection<Person>) =
  createHTML().table {
    for (person in persons) {
      tr {
        td { +person.name }
        td { +person.age }
      }
    }
  }
}
```

Обычный цикл

Функции, выполняющие отображение в теги HTML

Вы легко сможете объединить функции, выводящие теги HTML, с обычными конструкциями языка Kotlin. Вам больше не нужно изучать отдельный язык шаблонов со своим синтаксисом только затем, чтобы использовать цикл при создании HTML-страницы.

Другой пример использования ясности Kotlin и лаконичности предметно-ориентированных языков – фреймворки хранения данных. Например, фреймворк Exposed (https://github.com/jetbrains/exposed) поддерживает простой и понятный предметный язык для описания структуры базы данных SQL и выполнения запросов прямо из кода на Kotlin с полноценной проверкой типов. Вот маленький пример, показывающий возможности такого подхода:

```
object CountryTable : IdTable() {
  val name = varchar("name", 250).uniqueIndex()
  val iso = varchar("iso", 2).uniqueIndex()
}

class Country(id: EntityID) : Entity(id) {
  var name: String by CountryTable.name
  var iso: String by CountryTable.iso
}

val russia = Country.find {
  CountryTable.iso.eq("ru")
}.first()

println(russia.name)
```

Описание таблицы в базе данных

Определение класса, соответствующего сущности в базе данных

Вы можете выполнять запросы к базе данных на чистом Kotlin

Мы рассмотрим эти методы более подробно в разделе 7.5 и в главе 11.

1.3.2. Kotlin в Android

Типичное мобильное приложение значительно отличается от типичного корпоративного приложения. Оно гораздо меньше по объему, не так сильно зависит от интеграции с существующими кодовыми базами и, как правило, должно быть разработано за короткий срок, при этом поддерживая надежную работу на различных устройствах. Kotlin также хорошо справляется с проектами такого типа.

Языковые особенности Kotlin в сочетании со специальным плагином для компилятора делают разработку для Android приятным и продуктивным занятием. Часто встречающиеся задачи программирования, такие как добавление обработчиков событий в элементы управления или связывание элементов интерфейса с полями, можно решить гораздо меньшим объемом кода, а иногда и совсем без кода (компилятор сгенерирует его за вас). Библиотека Anko (https://github.com/kotlin/anko), тоже разработанная командой Kotlin, сделает вашу работу ещё приятнее за счет Kotlin-совместимых адаптеров для многих стандартных Android API.

Ниже продемонстрирован простой пример использования Anko, чтобы вы почувствовали, что значит разработка для Android на языке Kotlin. Вы

можете скопировать этот код в класс-наследник `Activity` и получить готовое Android-приложение!

```
verticalLayout {
  val name = editText()
  button("Say Hello") {
    onClick { toast("Hello, ${name.text}!") }
  }
}
```

Создание простого текстового поля

По щелчку на кнопке вывести всплывающее сообщение с содержимым текстового поля

Лаконичный синтаксис подключения обработчика событий и отображения всплывающего сообщения

Другое большое преимущество Kotlin – повышенная надежность приложений. Если у вас есть какой-либо опыт разработки приложений для Android, вы наверняка знакомы с сообщением «К сожалению, процесс остановлен». Это диалоговое окно появляется, когда приложение встречается с необработанным исключением, обычно `NullPointerException`. Система типов в Kotlin, с её точным контролем значений `null`, значительно снижает риск исключений из-за обращения к пустому указателю. Большую часть кода, который в Java приводит к исключению `NullPointerException`, в языке Kotlin попросту не удастся скомпилировать, что гарантирует исправление ошибки до того, как приложение попадет к пользователям.

В то же время, поскольку Kotlin полностью совместим с Java 6, его использование не создает каких-либо новых опасений в области совместимости. Вы сможете воспользоваться всеми новыми возможностями языка, а пользователи по-прежнему смогут запускать ваше приложение на устройствах даже с устаревшей версией Android.

С точки зрения производительности, применение Kotlin также не влечет за собой каких-либо недостатков. Код, сгенерированный компилятором Kotlin, выполняется так же эффективно, как обычный Java-код. Стандартная библиотека Kotlin довольно небольшая, поэтому вы не заметите существенного увеличения размеров скомпилированного приложения. А при использовании лямбда-выражений многие функции из стандартной библиотеки Kotlin просто будут встраивать их в место вызова. Встраивание лямбда-выражений гарантирует, что не будет создано никаких новых объектов, а приложение не будет работать медленнее из-за дополнительных пауз сборщика мусора (GC).

Познакомившись с преимуществами Котлин, давайте теперь рассмотрим философию Kotlin – основные черты, которые отличают Kotlin от других современных языков для платформы JVM.

1.4. Философия Kotlin

Говоря о Kotlin, мы подчеркиваем, что это прагматичный, лаконичный, безопасный язык, совместимый с Java. Что мы подразумеваем под каждым из этих понятий? Давайте рассмотрим по порядку.

1.4.1. Прагматичность

Под *прагматичностью* мы понимаем одну простую вещь: Kotlin является практичным языком, предназначенным для решения реальных задач. Он спроектирован с учетом многолетнего опыта создания крупномасштабных систем, а его характеристики выбирались исходя из задач, которые чаще всего приходится решать разработчикам. Более того, разработчики в компании JetBrains и в сообществе несколько лет использовали ранние версии Kotlin, и полученная от них обратная связь помогла сформировать итоговую версию языка. Это дает нам основания заявлять, что Kotlin действительно помогает решать проблемы в реальных проектах.

Kotlin не является исследовательским языком. Мы не пытаемся создать ультрасовременный язык программирования и исследовать различные инновационные идеи. Вместо этого, когда это возможно, мы полагаемся на особенности и готовые решения в других языках программирования, оказавшиеся успешными. Это уменьшает сложность языка и упрощает его изучение за счет того, что многие понятия уже знакомы.

Кроме того, Kotlin не требует применения какого-то конкретного стиля программирования или парадигмы. Приступая к изучению языка, вы сможете использовать стиль и методы, знакомые вам по работе с Java. Позже вы постепенно откроете более мощные возможности Kotlin и научитесь применять их в своем коде, делая его более лаконичным и идиоматичным.

Другой аспект прагматизма Kotlin касается инструментария. Хорошая среда разработки так же важна для программиста, как и хороший язык; следовательно, поддержка Kotlin в IDE не является чем-то малозначительным. Плагин для IntelliJ IDEA с самого начала разрабатывался параллельно с компилятором, а свойства языка всегда рассматривались через призму инструментария.

Поддержка в IDE также играет важную роль в изучении возможностей Kotlin. В большинстве случаев инструменты автоматически обнаруживают шаблонный код, который можно заменить более лаконичными конструкциями, и предлагают его исправить. Изучая особенности языка в исправлениях, предлагаемых инструментами, вы сможете научиться применять их в своем собственном коде.

1.4.2. Лаконичность

Известно, что разработчики тратят больше времени на чтение существующего кода, чем на создание нового. Представьте, что вы в составе команды участвуете в разработке большого проекта, и вам нужно добавить новую функцию или исправить ошибку. Каковы ваши первые шаги? Вы найдете область кода, которую нужно изменить, и только потом сделаете исправление. Вы должны прочесть много кода, чтобы выяснить, что нужно сделать. Этот код мог быть недавно написан вашими коллегами, кем-то,

кто уже не работает на проекте, или вами, но давным-давно. Только поняв, как работает окружающий код, вы сможете внести необходимые изменения.

Чем проще и лаконичнее код, тем быстрее вы поймете, что он делает. Конечно, хороший дизайн и выразительные имена играют свою роль. Но выбор языка и лаконичность также имеют большое значение. Язык является *лаконичным*, если его синтаксис явно выражает намерения кода, не загромождая его вспомогательными конструкциями с деталями реализации.

Создавая Kotlin, мы старались организовать синтаксис так, чтобы весь код нес определенный смысл, а не писался просто ради удовлетворения требований к структуре кода. Большинство операций, стандартных для Java, таких как определение методов чтения/записи для свойств и присваивание параметров конструктора полям объекта, реализовано в Kotlin неявно и не захламляет исходного кода.

Другой причиной ненужной избыточности кода является необходимость описания типовых задач, таких как поиск элемента в коллекции. Как во многих современных языках, в Kotlin есть богатая стандартная библиотека, позволяющая заменить эти длинные, повторяющиеся участки кода вызовами библиотечных функций. Поддержка лямбда-выражений в Kotlin позволяет передавать небольшие блоки кода в библиотечные функции и инкапсулировать всю общую логику в библиотеке, оставляя в коде только уникальную логику для решения конкретных задач.

В то же время Kotlin не пытается минимизировать количество символов в исходном коде. Например, несмотря на то что Kotlin поддерживает перегрузку операторов, пользователи не могут определять собственных операторов. Поэтому разработчики библиотек не смогут заменять имена методов загадочными последовательностями знаков препинания. Как правило, слова читать легче, чем знаки препинания, и их проще искать в документации.

Более лаконичный код требует меньше времени для написания и, что особенно важно, меньше времени для чтения. Это повышает продуктивность и позволяет разрабатывать программы значительно быстрее.

1.4.3. Безопасность

Называя язык *безопасным*, мы обычно подразумеваем, что его дизайн предотвращает появление определенных видов ошибок в программах. Конечно, это качество не абсолютно: ни один язык не защитит от всех возможных ошибок. Кроме того, за предотвращение ошибок, как правило, приходится платить. Вы должны сообщить компилятору больше информации о планируемых действиях программы, чтобы компилятор мог проверить, что код действительно соответствует этим действиям. Вследствие

этого всегда возникает компромисс между приемлемым уровнем безопасности и потерей продуктивности из-за необходимости добавления дополнительных аннотаций.

В Kotlin мы попытались достичь более высокого уровня безопасности, чем в Java, с минимальными накладными расходами. Выполнение кода в JVM уже дает многие гарантии безопасности, например: защита памяти избавляет от переполнения буфера и других проблем, связанных с некорректным использованием динамически выделяемой памяти. Будучи статически типизированным языком для JVM, Kotlin также обеспечивает безопасность типов в приложении. Это обходится дешевле, чем в Java: вам не нужно явно объявлять типы всех сущностей, поскольку во многих случаях компилятор может вывести тип автоматически.

Но Kotlin идёт ещё дальше: теперь ещё больше ошибок может быть предотвращено во время компиляции, а не во время выполнения. Важнее всего, что Kotlin пытается избавить программу от исключений `NullPointerException`. Система типов в Kotlin отслеживает значения, которые могут или и не могут принимать значение `null`, и запрещает операции, которые могут привести к возникновению `NullPointerException` во время выполнения. Дополнительные затраты при этом минимальны: чтобы указать, что значение может принимать значение `null`, требуется только один символ – вопросительный знак в конце:

```
val s: String? = null          <— Может содержать значение null
val s2: String = ""            <— Не может содержать значения null
```

Кроме того, Kotlin поддерживает множество удобных способов обработки значений, которые могут содержать `null`. Это очень помогает в устранении сбоев приложений.

Другой тип исключений, которого помогает избежать Kotlin, – это `ClassCastException`. Оно возникает во время приведения типа без предварительной проверки возможности такой операции. В Java разработчики часто не делают такую проверку, потому что имя типа приходится повторять в выражении проверки и в коде приведения типа. Однако в Kotlin проверка типа и приведение к нему объединены в одну операцию: после проверки можно обращаться к членам этого типа без дополнительного явного приведения. Поэтому нет причин не делать проверку и нет никаких шансов сделать ошибку. Вот как это работает:

```
if (value is String)           <— Проверка типа
  println(value.toUpperCase())          <— Вызов метода типа
```

1.4.4. Совместимость

В отношении совместимости часто первым возникает вопрос: «Смогу ли я использовать существующие библиотеки?» Kotlin дает однозначный

ответ: «Да, безусловно». Независимо от того, какой тип API предлагает библиотека, вы сможете работать с ними напрямую из Kotlin. Вы сможете вызывать Java-методы, наследовать Java-классы и реализовывать интерфейсы, использовать Java-аннотации в Kotlin-классах и т. д.

В отличие от некоторых других языков для JVM, Kotlin идет еще дальше по пути совместимости, позволяя также легко вызывать код Kotlin из Java. Для этого не требуется никаких трюков: классы и методы на Kotlin можно вызывать как обычные классы и методы на Java. Это дает максимальную гибкость при смешивании кода Java с кодом Kotlin на любом этапе вашего проекта. Приступая к внедрению Kotlin в свой Java-проект, попробуйте преобразовать какой-нибудь один класс из Java в Kotlin с помощью конвертера, и остальной код будет продолжать компилироваться и работать без каких-либо изменений. Этот прием работает независимо от назначения преобразованного класса.

Ещё одна область, где уделяется большое внимание совместимости, – максимальное использование существующих библиотек Java. Например, в Kotlin нет своей библиотеки коллекций. Он полностью полагается на классы стандартной библиотеки Java, расширяя их дополнительными функциями для большего удобства использования в Kotlin. (Мы рассмотрим этот механизм более подробно в разделе 3.3.) Это означает, что вам никогда не придется обертывать или конвертировать объекты при использовании Java API из Kotlin или наоборот. Все богатство возможностей предоставляется языком Kotlin без дополнительных накладных расходов во время выполнения.

Инструментарий Kotlin также обеспечивает полную поддержку многоязычных проектов. Можно скомпилировать произвольную смесь исходных файлов на Java и Kotlin с любыми зависимостями друг от друга. Поддержка IDE также распространяется на оба языка, что позволяет:

- свободно перемещаться между исходными файлами на Java и Kotlin;
- отлаживать смешанные проекты, перемещаясь по коду, написанному на разных языках;
- выполнять рефакторинг Java-методов, получая нужные изменения в коде на Kotlin, и наоборот.

Надеемся, что теперь мы убедили вас дать шанс языку Kotlin. Итак, как начать пользоваться им? В следующем разделе мы рассмотрим процесс компиляции и выполнения кода на Kotlin из командной строки и с помощью различных инструментов.

1.5. Инструментарий Kotlin

Как и Java, Kotlin – компилируемый язык. То есть, прежде чем запустить код на Kotlin, его нужно скомпилировать. Давайте обсудим процесс компи-

ляции, а затем рассмотрим различные инструменты, которые позаботятся о нём за вас. Более подробную информацию о настройке вашего окружения вы найдете в разделе «Tutorials» на сайте Kotlin (https://kotlinlang.org/docs/tutorials).

1.5.1. Компиляция кода на Kotlin

Исходный код на Kotlin обычно хранится в файлах с расширением *.kt*. Компилятор Kotlin анализирует исходный код и генерирует файлы *.class* так же, как и компилятор Java. Затем сгенерированные файлы *.class* упаковываются и выполняются с использованием процедуры, стандартной для данного типа приложения. В простейшем случае скомпилировать код из командной строки можно с помощью команды `kotlinc`, а запустить – командой `java`:

```
kotlinc <исходный файл или каталог> -include-runtime -d <имя jar-файла>
java -jar <имя jar-файла>
```

На рис. 1.1 приводится упрощенная схема процесса сборки в Kotlin.

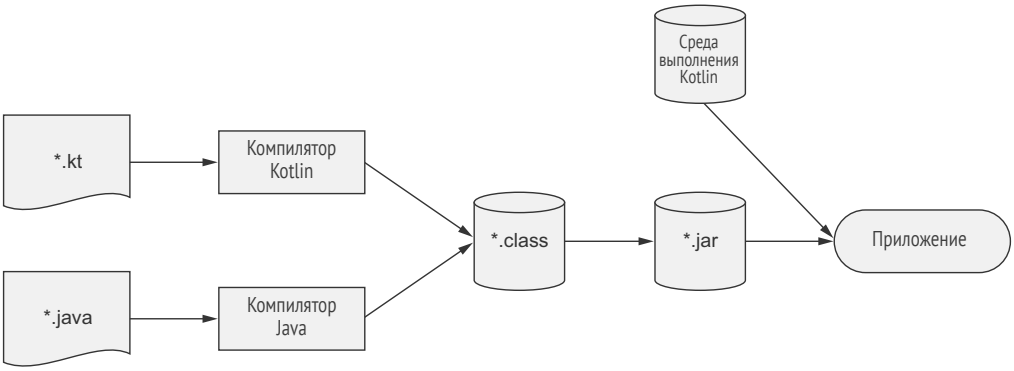


Рис. 1.1. Процесс сборки в Kotlin

Код, скомпилированный с помощью компилятора Kotlin, зависит от *библиотеки среды выполнения Kotlin*. Она содержит определения собственных классов стандартной библиотеки Kotlin, а также расширения, которые Kotlin добавляет в стандартный Java API. Библиотека среды выполнения должна распространяться вместе с приложением.

На практике для компиляции кода обычно используются специализированные системы сборки, такие как Maven, Gradle или Ant. Kotlin совместим со всеми ними, и мы обсудим эту тему в приложении А. Все эти системы сборки поддерживают многоязычные проекты, включающие код на Kotlin и Java в общую базу кода. Кроме того, Maven и Gradle сами позаботятся о подключении библиотеки времени выполнения Kotlin как зависимости вашего приложения.

1.5.2. Плагин для Intellij IDEA и Android Studio

Плагин с поддержкой Kotlin для IntelliJ IDEA разрабатывался параллельно с языком и является наиболее полной средой разработки для языка Kotlin. Это зрелая и стабильная среда, обладающая полным набором инструментов для разработки на Kotlin. Плагин Kotlin вошел в состав IntelliJ IDEA, начиная с версии 15, поэтому дополнительная установка не потребуется. Вы можете использовать бесплатную среду разработки IntelliJ IDEA Community Edition с открытым исходным кодом, или IntelliJ IDEA Ultimate. Выберите **Kotlin** в диалоге **New Project** (Новый проект) и можете начинать разработку.

Если вы используете Android Studio, можете установить плагин Kotlin с помощью диспетчера плагинов. В диалоговом окне **Settings** (Настройки) выберите вкладку **Plugins** (Плагины), затем щелкните на кнопке **Install JetBrains Plugin** (Установить плагин JetBrains) и выберите из списка **Kotlin**.

1.5.3. Интерактивная оболочка

Для быстрого опробования небольших фрагментов кода на Kotlin можно использовать интерактивную оболочку (так называемый цикл *REPL* – Read Eval Print Loop: чтение ввода, выполнение, вывод результата, повтор). В REPL можно вводить код на Kotlin строку за строкой и сразу же видеть результаты выполнения. Чтобы запустить REPL, выполните команду `kotlinc` без аргументов или воспользуйтесь соответствующим пунктом меню в плагине IntelliJ IDEA.

1.5.4. Плагин для Eclipse

Пользователи Eclipse также имеют возможность программировать на Kotlin в своей IDE. Плагин Kotlin для Eclipse поддерживает основную функциональность, такую как навигация и автодополнение кода. Плагин доступен в Eclipse Marketplace. Чтобы установить его, выберите пункт меню **Help > Eclipse Marketplace** (Справка > Eclipse Marketplace) и найдите **Kotlin** в списке.

1.5.5. Онлайн-полигон

Самый простой способ попробовать Kotlin в действии не требует никаких дополнительных установок и настроек. По адресу `http://try.kotl.in` вы найдете онлайн-полигон, где сможете писать, компилировать и запускать небольшие программы на Kotlin. На полигоне представлены примеры кода, демонстрирующие возможности Kotlin (включая все примеры из этой книги), а также ряд упражнений для изучения Kotlin в интерактивном режиме.

1.5.6. Конвертер кода из Java в Kotlin

Освоение нового языка никогда не бывает легким. К счастью, мы создали утилиту, которая поможет быстрее изучить и овладеть им, опираясь на знание языка Java. Это – автоматизированный конвертер кода на Java в код на Kotlin.

В начале изучения конвертер поможет вам запомнить точный синтаксис Kotlin. Напишите фрагмент кода на Java, вставьте его в файл с исходным кодом на Kotlin, и конвертер автоматически предложит перевести этот фрагмент на язык Kotlin. Результат не всегда будет идиоматичным, но это будет рабочий код, с которым вы сможете продвинуться ближе к решению своей задачи.

Конвертер также удобно использовать для внедрения Kotlin в существующий Java-проект. Новый класс можно сразу определить на языке Kotlin. Но если понадобится внести значительные изменения в существующий класс и у вас появится желание использовать Kotlin, конвертер придет вам на выручку. Сначала переведите определение класса на язык Kotlin, а затем сделайте необходимые изменения, используя все преимущества современного языка.

Использовать конвертер в IntelliJ IDEA очень просто. Достаточно просто скопировать фрагмент кода на Java и вставить в файл на Kotlin или выбрать пункт меню **Code** → **Convert Java File to Kotlin** (Код → Преобразовать файл Java в Kotlin), чтобы перевести весь файл на язык Kotlin. Конвертер также доступен в Eclipse и на онлайн-полигоне.

1.6. Резюме

- Kotlin – статически типизированный язык, поддерживающий автоматический вывод типов, что позволяет гарантировать корректность и производительность, сохраняя при этом исходный код лаконичным.
- Kotlin поддерживает как объектно-ориентированный, так и функциональный стиль программирования, позволяя создавать высокоуровневые абстракции с помощью функций, являющихся полноценными объектами, и упрощая разработку и тестирование многопоточных приложений благодаря поддержке неизменяемых значений.
- Язык хорошо подходит для разработки серверных частей приложений, поддерживает все существующие Java-фреймворки и предоставляет новые инструменты для решения типичных задач, таких как создание разметки HTML и операции с хранимыми данными.
- Благодаря компактной среде выполнения, специальной поддержке Android API в компиляторе и богатой библиотеке Kotlin-функций для

решения основных задач Kotlin отлично подходит для разработки под Android.

- Это бесплатный язык с открытым исходным кодом, поддерживаемый основными IDE и системами сборки.
- Kotlin – прагматичный, безопасный, лаконичный и совместимый язык, уделяющий большое внимание возможности использования проверенных решений для популярных задач, предотвращающий распространенные ошибки (такие как исключение `NullPointerException`), позволяющий писать компактный, легко читаемый код и обеспечивающий бесшовную интеграцию с Java.

Глава 2

Основы Kotlin

В этой главе объясняются:

- объявление функций, переменных, классов, перечислений и свойств;
- управляющие структуры;
- автоматическое приведение типов;
- возбуждение и перехват исключений.

В этой главе вы узнаете, как на языке Kotlin объявляются важнейшие элементы любой программы: переменные, функции и классы. Попутно вы познакомитесь с понятием свойств в Kotlin.

Вы научитесь пользоваться различными управляющими структурами Kotlin. В основном они похожи на знакомые вам структуры в языке Java, но имеют ряд важных усовершенствований.

Мы познакомим вас с идеей *автоматического приведения типов* (smart casts), когда проверка и приведение типа сочетаются в одной операции. И наконец, мы поговорим об обработке исключений. К концу этой главы вы научитесь писать действующий код на языке Kotlin, даже если он будет не вполне идиоматичным.

2.1. Основные элементы: переменные и функции

В этом разделе вы познакомитесь с основными элементами, присутствующими в каждой программе на Kotlin: переменными и функциями, и увидите, как Kotlin позволяет опускать объявления типов и поощряет использование неизменяемых данных.

2.1.1. Привет, мир!

Начнем с классического примера – программы, которая выводит на экран текст «Hello, world!» (Привет, мир!). В Kotlin для этого достаточно всего одной функции:

Листинг 2.1. «Привет, мир!» на языке Kotlin

```
fun main(args: Array<String>) {
  println("Hello, world!")
}
```

Какие особенности и нюансы синтаксиса демонстрирует этот простой пример? Взгляните на этот список:

- Объявления функций начинаются с ключевого слова `fun`. Программировать на языке Kotlin действительно весело![1]
- Тип параметра указывается после его имени. Как вы увидите позже, это относится и к объявлениям переменных.
- Функцию можно объявить на верхнем уровне в файле – её не обязательно помещать в класс.
- Массивы – это просто классы. В отличие от Java, в Kotlin нет специального синтаксиса для объявления массивов.
- Вместо `System.out.println` можно писать просто `println`. Стандартная библиотека Kotlin включает множество оберток с лаконичным синтаксисом для функций в стандартной библиотеке Java, и `println` – одна из них.
- Точку с запятой в конце строки можно опустить, как и во многих других современных языках.

Пока всё идёт хорошо! Позже мы подробнее обсудим некоторые из этих пунктов. А пока исследуем синтаксис объявления функций.

2.1.2. Функции

Вы уже знаете, как объявить функцию, которая ничего не возвращает. Но где нужно указывать тип возвращаемого значения для функций, возвращающих результат? Вы наверняка догадались, что это должно быть где-то после списка параметров:

```
fun max(a: Int, b: Int): Int {
  return if (a > b) a else b
}

>>> println(max(1, 2))
2
```

[1] fun (англ.) – веселый, забавный. – *Прим. пер.*

Объявление функции начинается с ключевого слова `fun`, за которым следует ее имя: в данном случае `max`. Далее следует список параметров в круглых скобках. Тип возвращаемого значения указывается после списка параметров и отделяется от него двоеточием.

На рис. 2.1 изображена базовая структура функции. Обратите внимание, что в Kotlin оператор `if` является выражением, возвращающим значение. Это похоже на тернарный оператор в Java: `(a > b) ? a : b`.

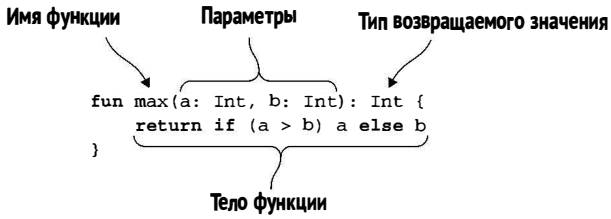


Рис. 2.1. Объявление функции в Kotlin

Выражения и инструкции

В языке Kotlin оператор `if` – это выражение, а не инструкция. Разница между выражениями и инструкциями состоит в том, что выражение имеет значение, которое можно использовать в других выражениях, в то время как инструкции всегда являются элементами верхнего уровня в охватывающем блоке и не имеют собственного значения. В Java все управляющие структуры – инструкции. В Kotlin большинство управляющих структур, кроме циклов (`for`, `do` и `do/while`), – выражения. Возможность комбинировать структуры управления с другими выражениями позволяет ёмко выражать многие распространенные шаблоны, как вы увидите в следующих главах.

С другой стороны, оператор присваивания в Java – это выражение, а в Kotlin – инструкция. Это помогает избежать частого источника ошибок – путаницы между сравнениями и присваиваниями.

Тела выражений

Функцию на рис. 2.1 можно ещё упростить. Поскольку её тело состоит из единственного выражения, то им можно заменить всё тело функции, удалив фигурные скобки и инструкцию `return`:

```
fun max(a: Int, b: Int): Int = if (a > b) a else b
```

Если тело функции заключено в фигурные скобки, мы говорим, что такая функция имеет *тело-блок* (block body). Функция, возвращающая выражение напрямую, имеет *тело-выражение* (expression body).

Совет для IntelliJ IDEA

IntelliJ IDEA поддерживает специальные операции преобразования между двумя стилями функций: **Convert to expression body** (Преобразовать в тело-выражение) и **Convert to block body** (Преобразовать в тело-блок).

Функции с телом-выражением часто встречаются в коде на Kotlin. Такой стиль применяется не только для простых однострочных функций, но также для функций, вычисляющих единственное более сложное выражение, таких как `if`, `when` или `try`. Вы увидите такие функции далее в этой главе, когда мы будем обсуждать оператор `when`.

Функцию `max` можно упростить ещё больше, опустив тип возвращаемого значения:

```
fun max(a: Int, b: Int) = if (a > b) a else b
```

Как возможны функции без объявления типа возвращаемого значения? Разве язык Kotlin как статически типизированный не требует знать тип каждого выражения на этапе компиляции? Действительно, каждая переменная и каждое выражение имеют тип, и каждая функция имеет тип возвращаемого значения. Но для функций с телом-выражением компилятор может проанализировать выражение и использовать его тип в качестве типа возвращаемого значения функции, даже когда он не указан явно. Анализ этого вида обычно называется выведением типа (type inference).

Обратите внимание, что опустить тип возвращаемого значения можно только в функциях с телом-выражением. В функциях с телом-блоком тип возвращаемого значения (если оно имеется) должен указываться явно, и обязательно должна использоваться инструкция `return`. Это осознанный выбор. В реальном мире функции часто бывают длинными и могут содержать несколько инструкций `return`; явно указанный тип возвращаемого значения и инструкция `return` помогут быстро понять, что возвращает функция. Теперь давайте рассмотрим синтаксис объявления переменных.

2.1.3. Переменные

В Java объявление переменной начинается с типа. Такой способ не поддерживается в Kotlin, поскольку он позволяет пропускать типы во многих объявлениях переменных. Поэтому в Kotlin объявление начинается с ключевого слова, а тип можно указать (или не указывать) после имени переменной. Объявим две переменные:

```
val question =
    "The Ultimate Question of Life, the Universe, and Everything"
val answer = 42
```

В этом примере объявления типов отсутствуют, но вы можете добавить их, если хотите:

```
val answer: Int = 42
```

Так же, как в функциях с телом-выражением, если тип не указан явно, компилятор проанализирует инициализирующее выражение и присвоит его тип переменной. В данном случае инициализирующее выражение 42 имеет тип `Int`, поэтому переменная получит тот же тип.

Если использовать константу с плавающей точкой, переменная получит тип `Double`:

```
val yearsToCompute = 7.5e6          <— 7.5 * 10^6 = 7500000.0
```

Подробнее числовые типы рассматриваются в разделе 6.2.

Если в объявлении переменной отсутствует инициализирующее выражение, её тип нужно указать явно:

```
val answer: Int
answer = 42
```

Компилятор не сможет определить тип, если не дать ему никакой информации о значениях, которые могут быть присвоены этой переменной.

Изменяемые и неизменяемые переменные

Есть два ключевых слова для объявления переменной:

- `val` (от value) – неизменяемая ссылка. Переменной, объявленной с ключевым словом `val`, нельзя присвоить значение после инициализации. Такие переменные соответствуют финальным переменным в Java.
- `var` (от variable) – изменяемая ссылка. Значение такой переменной можно изменить. Такое объявление соответствует обычной (не финальной) переменной в Java.

По умолчанию вы должны стремиться объявлять все переменные в Kotlin с ключевым словом `val`. Заменяйте его на `var` только при необходимости. Использование неизменяемых ссылок и объектов, а также функций без побочных эффектов приблизит ваш код к функциональному стилю. Мы немного коснулись его достоинств в главе 1 и еще вернемся к этой теме в главе 5.

Переменная, объявленная с ключевым словом `val`, должна быть инициализирована только один раз во время выполнения блока, в котором она определена. Но её можно инициализировать разными значениями в зависимости от некоторых условий, если компилятор сможет гарантировать, что выполнится только одно из инициализирующих выражений:

```
val message: String
if (canPerformOperation()) {
  message = "Success"
  // ... выполнить операцию
```

```
}
else {
  message = "Failed"
}
```

Обратите внимание: несмотря на невозможность изменить ссылку `val`, объект, на который она указывает, может быть изменяемым. Например, следующий код является вполне допустимым:

```
val languages = arrayListOf("Java")        <— Объявление неизменяемой ссылки
languages.add("Kotlin")                    <— Изменение объекта, на который она указывает
```

В главе 6 мы подробнее обсудим изменяемые и неизменяемые объекты.

Хотя ключевое слово `var` позволяет менять значение переменной, но её тип фиксирован. Например, следующий код не скомпилируется:

```
var answer = 42
answer = "no answer"       <— Ошибка: несовпадение типов
```

Попытка присвоить строковый литерал вызовет ошибку, потому что его тип (`String`) не соответствует ожидаемому (`Int`). Компилятор определяет тип переменной только по инициализирующему выражению и не принимает во внимание всех последующих операций присваивания.

Если вам нужно сохранить в переменной значение другого типа, вы должны преобразовать его вручную или привести к нужному типу. Мы обсудим преобразование простых типов в разделе 6.2.3.

Теперь, когда вы узнали, как определять переменные, перейдем к знакомству с некоторыми приемами, позволяющими ссылаться на значения этих переменных.

2.1.4. Простое форматирование строк: шаблоны

Вернемся к примеру «Hello, world!» в начале этого раздела. Вот как можно перейти к следующей стадии традиционного упражнения и поприветствовать людей по именам на Kotlin:

Листинг 2.2. Применение строковых шаблонов

```
fun main(args: Array<String>) {
  val name = if (args.size > 0) args[0] else "Kotlin"
  println("Hello, $name!")
}
```

Выведет «Hello, Kotlin!» или «Hello, Bob!», если передать аргумент со строкой «Bob»

Этот пример демонстрирует применение особенности синтаксиса Kotlin, которая называется *строковые шаблоны* (string templates). Вы объявляете в коде переменную `name`, а затем используете её в строковом литерале

ниже. Так же, как многие языки сценариев, Kotlin позволяет использовать в строковых литералах ссылки на локальные переменные, добавляя к ним в начало символ $. Эта запись равносильна конкатенации строк в Java (`"Hello, " + name + "!"`), но она более компактна и столь же эффективна[2]. И конечно, такие выражения проверяются статически, поэтому при попытке обратиться к несуществующей переменной код не будет компилироваться.

Чтобы включить в строку символ `"$"`, его нужно его экранировать: `println("\$x")` выведет `$x` и не проинтерпретирует `x` как ссылку на переменную.

Вы не ограничены простыми именами переменных, но также можете использовать более сложные выражения. Для этого достаточно заключить выражение в фигурные скобки:

```
fun main(args: Array<String>) {
  if (args.size > 0) {
    println("Hello, ${args[0]}!")
  }
}
```

Синтаксис ${} используется для подстановки первого элемента массива args

Также можно помещать двойные кавычки внутрь других двойных кавычек, пока они входят в состав выражения:

```
fun main(args: Array<String>) {
  println("Hello, ${if (args.size > 0) args[0] else "someone"}!")
}
```

Позже, в разделе 3.5, мы вернемся к строкам и подробнее поговорим о том, что можно с ними делать.

Теперь вы знаете, как объявлять переменные и функции. Давайте поднимемся на уровень выше и посмотрим на классы. На этот раз вы будете использовать конвертер кода из Java в Kotlin, который поможет приступить к работе с использованием новых возможностей языка.

2.2. Классы и свойства

Возможно, вы не новичок в объектно-ориентированном программировании и знакомы с абстракцией под названием *класс*. Основные понятия языка Kotlin в этой области будут вам знакомы, но вы обнаружите, что множество типичных задач можно решить гораздо меньшим объемом кода. Этот раздел познакомит вас с базовым синтаксисом объявления классов. Мы рассмотрим эту тему более подробно в главе 4.

Для начала рассмотрим простой JavaBean-класс `Person`, который пока имеет только одно свойство, `name`:

[2] Скомпилированный код создает объект **`StringBuilder`**, передавая ему константы и переменные.

Листинг 2.3. Простой Java-класс `Person`

```
/* Java */
public class Person {
  private final String name;

  public Person(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}
```

В Java тело конструктора часто содержит повторяющийся код: он присваивает значения параметров полям с соответствующими именами. В Kotlin эту логику можно выразить без ненужного шаблонного кода.

В разделе 1.5.6 мы познакомились с конвертером кода из Java в Kotlin: инструментом, который автоматически заменяет код на Java эквивалентным кодом на Котлин. Давайте посмотрим, как он действует, и переведем определение класса `Person` на язык Kotlin.

Листинг 2.4. Класс `Person`, преобразованный в Kotlin

```
class Person(val name: String)
```

Выглядит неплохо, не так ли? Если вам довелось попробовать другой современный язык для JVM, возможно, вы уже видели нечто подобное. Классы этого типа (содержащие только данные, без кода) часто называют *объектами-значениями* (value objects), и многие языки предлагают краткий синтаксис для их объявления.

Обратите внимание, что в ходе преобразования из Java в Kotlin пропал модификатор `public`. В Kotlin область видимости `public` принята по умолчанию, поэтому её можно не указывать.

2.2.1. Свойства

Как известно, классы предназначены для объединения данных и кода, работающего с этими данными, в одно целое. В Java данные хранятся в полях, обычно с модификатором `private`. Чтобы дать клиентам класса доступ к этим данным, нужно определить *методы доступа*: чтения и, возможно, записи. Вы видели пример этих методов в классе `Person`. Метод записи может содержать дополнительную логику для проверки переданного значения, отправки уведомления об изменении и т. д.

В Java сочетание поля и методов доступа часто называют *свойством* (property), и многие фреймворки широко используют это понятие. Свойства в языке Kotlin являются полноценной частью языка, полностью заменившей поля и методы доступа. Свойство в классе объявляется так же, как переменная: с помощью ключевых слов `val` и `var`. Свойство, объявленное как `val`, доступно только для чтения, а свойство `var` можно изменять.

Листинг 2.5. Объявление изменяемого свойства в классе

```
class Person(
  val name: String,     <— Неизменяемое свойство: для него будут созданы поле и простой метод чтения
  var isMarried: Boolean     <— Изменяемое свойство: поле, методы чтения и записи
)
```

Обычно, объявляя свойство, вы объявляете соответствующие методы доступа (метод чтения для свойства, доступного только для чтения, и методы чтения/записи для свойства, доступного для записи). По умолчанию методы доступа имеют хорошо известную реализацию: создается поле для хранения значения, а методы чтения и записи возвращают и изменяют его. Но при желании можно определить собственный метод доступа, использующий другую логику вычисления или изменения значения свойства.

Краткое объявление класса `Person` в листинге 2.5 скрывает традиционную реализацию, присутствующую в исходном коде на Java: это класс с приватными полями, которые инициализированы в конструкторе и доступны через соответствующие методы чтения. Это означает, что данный класс можно использовать и в Java, и в Kotlin, независимо от того, где он объявлен. Примеры использования выглядят идентично. Вот как можно использовать класс `Person` в коде на Java.

Листинг 2.6. Использование класса `Person` в Java

```
/* Java */
>>> Person person = new Person("Bob", true);
>>> System.out.println(person.getName());
Bob
>>> System.out.println(person.isMarried());
true
```

Обратите внимание, что этот код не зависит от того, на каком языке определен класс `Person` – Java или Kotlin. Свойство `name`, объявленное на языке Kotlin, доступно Java-коду через метод доступа с именем `getName`. В правилах именования методов доступа есть исключение: если имя свойства начинается с префикса `is`, никаких дополнительных префиксов для образования имени метода чтения не добавляется, а в имени метода

записи is заменяется на set. То есть в Java вы должны вызывать метод isMarried().

Если перевести код в листинге 2.6 на язык Kotlin, получится следующее.

Листинг 2.7. Использование класса Person в Kotlin

```
>>> val person = Person("Bob", true)  <— Конструктор вызывается без ключевого слова «new»
>>> println(person.name)
Bob                                   <— Прямое обращение к свойству, но
>>> println(person.isMarried)             при этом вызывается метод чтения
true
```

Теперь можно не вызывать метода чтения, а обращаться к свойству непосредственно. Логика та же, но код становится более лаконичным. Методы записи изменяемых свойств работают точно так же: чтобы сообщить о разводе, в Java требуется выполнить вызов person.setMarried(false), а в Kotlin достаточно записать person.isMarried = false.

> **Совет.** Синтаксис Kotlin также можно использовать для доступа к свойствам классов, объявленных в Java. К методам чтения Java-класса можно обращаться как val-свойствам Kotlin, а к парам методов чтения/записи – как к var-свойствам. Например, если Java-класс определяет методы getName и setName, к ним можно обратиться через свойство name. Если же он определяет методы isMarried и setMarried, соответствующее свойство в Kotlin будет иметь имя isMarried.

В большинстве случаев свойству соответствует поле, хранящее его значение. Но если значение можно вычислить на лету – например, на основании значений других свойств, – это можно выразить с помощью собственного метода чтения.

2.2.2. Собственные методы доступа

В этом разделе показано, как написать собственную реализацию метода доступа к свойству. Предположим, что вы определяете класс прямоугольников, который может сообщить, является ли эта фигура квадратом. Вам не надо хранить эту информацию в отдельном поле, так как всегда можно динамически проверить равенство высоты и ширины:

```
class Rectangle(val height: Int, val width: Int) {
  val isSquare: Boolean
    get() {                          <— Объявление метода чтения для свойства
      return height == width
  }
}
```

Свойству `isSquare` не нужно поле для хранения значения. Ему достаточно метода чтения с особой реализацией. Значение свойства вычисляется при каждом обращении к нему.

Обратите внимание, что не обязательно использовать полный синтаксис с фигурными скобками; также можно написать `get() = height == width`. Это не влияет на способ обращения к свойству:

```
>>> val rectangle = Rectangle(41, 43)
>>> println(rectangle.isSquare)
false
```

Если вам нужно обратиться к этому свойству из Java, вызовите метод `isSquare`, как прежде.

Вы можете спросить, что лучше: объявить функцию без параметров или свойство с собственным методом чтения. Оба варианта похожи: нет никакой разницы в реализации или производительности; они отличаются только оформлением. Но вообще, характеристика (свойство) класса должна быть объявлена свойством.

В главе 4 мы продемонстрируем больше примеров использования классов и свойств и рассмотрим синтаксис явного объявления конструктора. А пока, если вам не терпится, можете использовать конвертер из Java в Kotlin. Теперь, прежде чем перейти к обсуждению других особенностей языка, кратко обсудим, как код на Kotlin размещается на диске.

2.2.3. Размещение исходного кода на Kotlin: пакеты и каталоги

Вы знаете, что в Java все классы находятся в пакетах. В Kotlin также существует понятие пакета, похожее на аналогичное понятие в Java. Каждый файл Kotlin может иметь инструкцию `package` в начале, и все объявления (классы, функции и свойства) в файле будут помещены в этот пакет. Объявления из других файлов в том же пакете можно использовать напрямую, а объявления из других пакетов нужно импортировать. Так же, как в Java, инструкции импорта помещаются в начало файла и начинаются с ключевого слова `import`. Вот пример исходного файла, демонстрирующего синтаксис объявления пакета и инструкцию импорта.

Листинг 2.8. Объявление класса и функции в пакете

```
package geometry.shapes    <— Объявление пакета

import java.util.Random    <— Импорт класса из стандартной библиотеки Java

class Rectangle(val height: Int, val width: Int) {
  val isSquare: Boolean
```

```
    get() = height == width
}

fun createRandomRectangle(): Rectangle {
  val random = Random()
  return Rectangle(random.nextInt(), random.nextInt())
}
```

Kotlin не делает различия между импортом классов и функций, что позволяет импортировать любые объявления с помощью ключевого слова `import`. Функции верхнего уровня можно импортировать по имени.

Листинг 2.9. Импорт функции из другого пакета

```
package geometry.example

import geometry.shapes.createRandomRectangle  <— Импорт функции по имени

fun main(args: Array<String>) {
  println(createRandomRectangle().isSquare)  <— Очень редко будет выводить «true»
}
```

Кроме того, можно импортировать все объявления из определенного пакета, добавив `.*` после имени пакета. Обратите внимание, что такой *импорт со звездочкой* сделает видимыми не только классы, объявленные в пакете, но и свойства и функции верхнего уровня. Если в листинге 2.9 написать `import geometry.shapes.*` вместо явного импорта, код тоже скомпилируется без ошибок.

В Java вы должны располагать классы в структуре файлов и каталогов, соответствующей структуре пакета. Например, если у вас есть пакет `shapes` с несколькими классами, вы должны поместить каждый класс в отдельный файл с соответствующим именем и сохранить эти файлы в каталог с тем же именем `shapes`. На рис. 2.2 виден пример организации пакета `geometry` и его подпакетов. Предположим, что функция `createRandomRectangle` находится в отдельном классе `RectangleUtil`.

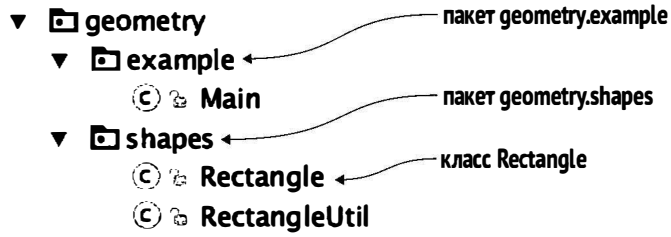


Рис. 2.2. В Java иерархия каталогов соответствует иерархии пакетов

Программируя на Kotlin, вы можете поместить несколько классов в один файл и выбрать любое имя для этого файла. Также Kotlin не накладыва-

ет никаких ограничений на расположение исходных файлов на диске; вы можете использовать любую структуру каталогов для организации своих файлов. Например, все содержимое пакета `geometry.shapes` можно поместить в файл `shapes.kt`, а сам файл сохранить в папку `geometry`, не создавая отдельной папки `shapes` (см. рис. 2.3).

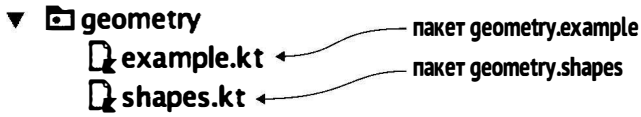


Рис. 2.3. Структура пакетов не должна соответствовать структуре каталогов

Однако в большинстве случаев хорошим тоном считается следовать структуре каталогов в Java и организовывать исходные файлы в каталогах в соответствии со структурой пакета. Придерживаться этого правила особенно важно в проектах, где Kotlin смешивается с Java, так как это позволит постепенно выполнить миграцию кода без всяких сюрпризов. Но не бойтесь помещать несколько классов в один файл, особенно если эти классы небольшие (а в Kotlin они часто бывают такими).

Теперь вы знаете, как организовывать программы. Давайте продолжим знакомство с основными понятиями и рассмотрим структуры управления в Kotlin.

2.3. Представление и обработка выбора: перечисления и конструкция «when»

В этом разделе мы поговорим о конструкции `when`. Её можно считать заменой конструкции `switch` в Java, но с более широкими возможностями и более частым применением на практике. Попутно мы покажем пример объявления перечислений в Kotlin и обсудим концепцию автоматического приведения типов (smart casts).

2.3.1. Объявление классов перечислений

Добавим воображаемых цветных картинок в эту серьезную книгу и создадим перечисление цветов.

Листинг 2.10. Объявление простого класса перечисления

```
enum class Color {
  RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
}
```

Это тот редкий случай, когда в объявлении на Kotlin используется больше ключевых слов, чем в Java: `enum class` против `enum` в Java. В языке

Kotlin enum – это так называемое *«мягкое» ключевое слово* (soft keyword): оно имеет особое значение только перед ключевым словом class, в других случаях его можно использовать как обычное имя. С другой стороны, ключевое слово class сохраняет свое специальное значение, и вам по-прежнему придется объявлять переменные с именами clazz или aClass.

Точно как в Java, перечисления – это не просто списки значений: в классах перечислений можно объявлять свойства и методы. Вот как это работает:

Листинг 2.11. Объявление класса перечисления со свойствами

```
enum class Color(
  val r: Int, val g: Int, val b: Int
) {
  RED(255, 0, 0), ORANGE(255, 165, 0),
  YELLOW(255, 255, 0), GREEN(0, 255, 0), BLUE(0, 0, 255),
  INDIGO(75, 0, 130), VIOLET(238, 130, 238);

  fun rgb() = (r * 256 + g) * 256 + b

}
>>> println(Color.BLUE.rgb())
255
```

Значения свойств определяются для каждой константы

Объявление свойств констант перечисления

Точка с запятой здесь обязательна

Определение метода класса перечисления

Константы перечислений используют тот же синтаксис объявления конструкторов и свойств, что и обычные классы. Объявляя константу перечисления, необходимо указать значения её свойств. Обратите внимание, что на этом примере вы видите единственное место в синтаксисе Kotlin, где требуется использовать точку с запятой: когда в классе перечисления определяются какие-либо методы, точка с запятой отделяет список констант от определений методов. Теперь рассмотрим некоторые интересные приёмы работы с константами перечислений.

2.3.2. Использование оператора «when» с классами перечислений

Помните, как дети используют мнемонические фразы для запоминания цветов радуги? Вот одна из них: «Каждый Охотник Желает Знать, Где Сидит Фазан!» Представьте, что вам нужно написать функцию, которая возвращает слово из фразы для каждого цвета (но вы не хотите хранить эту информацию в самом перечислении). В Java для этого можно использовать оператор switch. В Kotlin есть аналогичная конструкция: when.

Подобно if, оператор when – это выражение, возвращающее значение, поэтому вы можете написать функцию с телом-выражением, которая напрямую возвращает выражение when. Когда мы говорили о функциях в

начале главы, мы обещали привести пример многострочной функции с телом-выражением. Вот этот пример.

Листинг 2.12. Применение when для выбора правильного значения перечисления

```
fun getMnemonic(color: Color) =                    <— Сразу возвращает выражение «when»
            when (color) {
                Color.RED -> "Каждый"                  <— Возвращает соответствующую
                Color.ORANGE -> "Охотник"                  строку, если цвет совпадает
                Color.YELLOW -> "Желает"                   с константой перечисления
                Color.GREEN -> "Знать"
                Color.BLUE -> "Где"
                Color.INDIGO -> "Сидит"
                Color.VIOLET -> "Фазан"
            }
>>> println(getMnemonic(Color.BLUE))
Где
```

Код находит ветку, соответствующую заданному значению цвета. В отличие от Java, в Kotlin не нужно добавлять в каждую ветку инструкцию `break` (отсутствие `break` часто вызывает ошибки в Java). При наличии совпадения выполнится только соответствующая ветка. В одну ветку можно объединить несколько значений, разделив их запятыми.

Листинг 2.13. Объединение вариантов в одну ветку when

```
fun getWarmth(color: Color) = when(color) {
  Color.RED, Color.ORANGE, Color.YELLOW -> "теплый"
  Color.GREEN -> "нейтральный"
  Color.BLUE, Color.INDIGO, Color.VIOLET -> "холодный"
}

>>> println(getWarmth(Color.ORANGE))
теплый
```

В этих примерах использовались полные имена констант с указанием класса перечисления `Color`. Вы можете упростить код, импортировав значения констант.

Листинг 2.14. Импорт констант перечисления для использования без квалификатора

```
import ch02.colors.Color          <— Обращение к импортированным константам по именам
import ch02.colors.Color.*          <— Импорт класса Color, объявленный в другом пакете

fun getWarmth(color: Color) = when(color) {          Явный импорт констант перечисления
  RED, ORANGE, YELLOW -> "теплый"                <— для обращения к ним по именам
```

```
        GREEN -> "нейтральный"
        BLUE, INDIGO, VIOLET -> "холодный"
}
```

2.3.3. Использование оператора «when» с произвольными объектами

Оператор when в Kotlin обладает более широкими возможностями, чем switch в Java. В отличие от switch, который требует использовать константы (константы перечисления, строки или числовые литералы) в определениях вариантов, оператор when позволяет использовать любые объекты. Давайте напишем функцию, которая смешивает два цвета, если в результате такого смешивания получается цвет из нашей небольшой палитры. У нас не так много вариантов, и мы легко сможем перечислить их все.

Листинг 2.15. Использование различных объектов в ветках when

```
fun mix(c1: Color, c2: Color) =
  when (setOf(c1, c2)) {
    setOf(RED, YELLOW) -> ORANGE
    setOf(YELLOW, BLUE) -> GREEN
    setOf(BLUE, VIOLET) -> INDIGO
    else -> throw Exception("Грязный цвет")
  }
```

Перечисление пар цветов, пригодных для смешивания

Аргументом выражения «when» может быть любой объект. Он проверяется условными выражениями ветвей

Выполняется, если не соответствует ни одной из ветвей

```
>>> println(mix(BLUE, YELLOW))
GREEN
```

Если цвета c1 и c2 – это RED (красный) и YELLOW (желтый) или наоборот, в результате их смешивания получится ORANGE (оранжевый), и так далее. Для такой проверки необходимо использовать сравнение множеств. Стандартная библиотека Kotlin включает функцию setOf, которая создает множество Set с объектами, переданными в аргументах. *Множество* – это коллекция, порядок элементов которой не важен; два множества считаются равными, если содержат одинаковые элементы. То есть если множества setOf(c1, c2) и setOf(RED, YELLOW) равны, значит, c1 и c2 – это RED и YELLOW (или наоборот). А это как раз то, что нужно проверить.

Выражение when последовательно сравнивает аргумент с условиями во всех ветвях, начиная с верхней, пока не обнаружит совпадение. То есть множество setOf(c1, c2) сначала сравнивается со множеством setOf(RED, YELLOW), а затем с другими, друг за другом. Если ни одно из условий не выполнится, произойдет переход на ветку else.

Возможность использования любых условных выражений в ветвях when часто позволяет создавать лаконичный и выразительный код. В данном

примере условием является проверка равенства; далее вы увидите, что в качестве условия можно выбрать любое логическое выражение.

2.3.4. Выражение «when» без аргументов

Вы, наверное, заметили, что код в листинге 2.15 не очень эффективен. В каждом вызове функция создает несколько множеств `Set`, которые используются только для сравнения двух пар цветов. Обычно это не вызывает проблем, но если функция вызывается часто, стоит переписать код таким образом, чтобы при его выполнении не создавался мусор. Это делается с помощью выражения `when` без аргументов. Код станет менее читабельным, но это цена, которую часто приходится платить за лучшую производительность.

Листинг 2.16. Выражение `when` без аргументов

```
fun mixOptimized(c1: Color, c2: Color) =
    when {                                    ◁— Выражение «when» без аргумента
        (c1 == RED && c2 == YELLOW) ||
        (c1 == YELLOW && c2 == RED) ->
            ORANGE
        (c1 == YELLOW && c2 == BLUE) ||
        (c1 == BLUE && c2 == YELLOW) ->
            GREEN
        (c1 == BLUE && c2 == VIOLET) ||
        (c1 == VIOLET && c2 == BLUE) ->
            INDIGO
        else -> throw Exception("Dirty color")
    }
>>> println(mixOptimized(BLUE, YELLOW))
GREEN
```

В выражениях `when` без аргумента условием выбора ветки может стать любое логическое выражение. Функция `mixOptimized` делает то же, что `mix` в листинге 2.15, но она не создает дополнительных объектов, за что приходится расплачиваться читаемостью кода.

Давайте перейдем дальше и рассмотрим примеры использования выражения `when` с *автоматическим приведением типов*.

2.3.5. Автоматическое приведение типов: совмещение проверки и приведения типа

В качестве примера для этого раздела напишем функцию, которая вычисляет простые арифметические выражения, такие как `(1 + 2) + 4`. Для простоты мы реализуем только один тип операций: сложение двух чисел. Другие арифметические операции (вычитание, умножение и деление)

реализуются аналогично, и вы можете сделать это упражнение самостоятельно.

Сначала определимся, как будем представлять выражения. Их можно хранить в древовидной структуре, где каждый узел является суммой (Sum) или числом (Num). Узел Num всегда является листом, в то время как узел Sum имеет двух потомков: аргументы операции сложения. В листинге 2.17 определяется простая структура классов, используемых для представления выражений: интерфейс Expr и два класса, Num и Sum, которые его реализуют. Обратите внимание, что в интерфейсе Expr нет никаких методов; он используется как маркерный интерфейс, играющий роль общего типа для различных видов выражений. Чтобы показать, что класс реализует некоторый интерфейс, нужно после имени класса добавить двоеточие (:) и имя интерфейса:

Листинг 2.17. Иерархия классов для представления выражений

```
interface Expr
class Num(val value: Int) : Expr
class Sum(val left: Expr, val right: Expr) : Expr
```

Простой класс объектов-значений с одним свойством value, реализующий интерфейс Expr

Аргументами операции Sum могут быть любые экземпляры Expr: Num или другой объект Sum

Объект Sum хранит ссылки на аргументы left и right типа Expr; в этом маленьком примере они могут представлять экземпляры классов Num или Sum. Для хранения выражения (1 + 2) + 4, упомянутого выше, требуется создать объект Sum(Sum(Num(1), Num(2)), Num(4)). На рис. 2.4 изображено его древовидное представление.

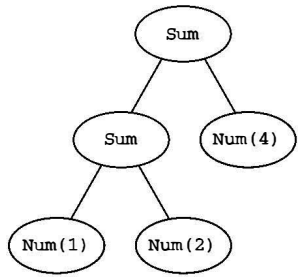


Рис. 2.4. Представление выражения Sum(Sum(Num(1), Num(2)), Num(4))

Теперь посмотрим, как вычислить значение выражения. Результатом выражения из примера должно быть число 7:

```
>>> println (eval(Sum(Sum(Num(1), Num(2)), Num (4))))
7
```

Интерфейс Expr имеет две реализации – соответственно, есть два варианта вычисления значения:

- если выражение является числом, возвращается соответствующее значение;
- если это операция сложения, после вычисления левой и правой частей выражения необходимо вернуть их сумму.

Сначала посмотрим, как такую функцию можно написать обычным для Java способом, а затем преобразуем её в стиле Kotlin. В Java для проверки

параметров часто используются последовательности операторов if, поэтому используем этот подход в коде на Kotlin.

Листинг 2.18. Вычисление выражения с помощью каскада операторов if

```
fun eval(e: Expr): Int {
  if (e is Num) {
    val n = e as Num      <— Явное приведение к типу Num здесь излишне
    return n.value
  }
  if (e is Sum) {
    return eval(e.right) + eval(e.left)  <— Переменная e уже приведена к нужному типу!
  }
  throw IllegalArgumentException("Unknown expression")
}

>>> println(eval(Sum(Sum(Num(1), Num(2)), Num(4))))
7
```

В Kotlin принадлежность переменной к определенному типу проверяется с помощью оператора is. Если вы программировали на C#, такой синтаксис должен быть вам знаком. Эта проверка подобна оператору instanceof в Java. Но в Java для получения доступа к нужным свойствам и методам необходимо выполнить явное приведение после проверки оператором instanceof. Когда исходная переменная используется несколько раз, приведенное значение часто сохраняют в отдельной переменной. Компилятор Kotlin сделает эту работу за вас. Если вы проверили переменную на соответствие определенному типу, приводить её к этому типу уже не надо; её можно использовать как значение проверенного типа. Компилятор выполнит приведение типа за вас, и мы называем это *автоматическим приведением типа* (smart cast).

```
if (e is Sum) {
    return eval(e.right) + eval(e.left)
}
```

Рис. 2.5. IDE выделяет переменные, подвергнутые автоматическому приведению типа

После проверки соответствия переменной e типу Num компилятор будет интерпретировать её как переменную типа Num. В результате вы можете обратиться к свойству value класса Num без явного приведения: e.value. То же касается свойств right и left класса Sum: в данном контексте можно просто писать e.right и e.left. IDE выделяет фоном значения, подвергнутые такому приведению типов, поэтому вы легко поймете, что значение уже было проверено. См. рис. 2.5.

Автоматическое приведение работает, только если переменная не изменялась после проверки оператором is. Автоматическое приведение применяется только к свойствам класса, объявленным с ключевым словом

val, как в данном примере, и не имеющим метода записи. В противном случае нельзя гарантировать, что каждое обращение к объекту будет возвращать одинаковое значение.

Явное приведение к конкретному типу выражается с помощью ключевого слова as:

```
val n = e as Num
```

Теперь давайте посмотрим, как можно привести функцию eval к более идиоматичному стилю Kotlin.

2.3.6. Рефакторинг: замена «if» на «when»

Чем оператор if в Kotlin отличается от оператора if в Java? Вы уже знаете разницу. В начале главы вы видели выражение if, которое используется в таком контексте, в котором в Java использовался бы тернарный оператор: if (a > b) a else b действует так же, как a > b ? a : b в Java. В Kotlin тернарный оператор отсутствует, поскольку, в отличие от Java, выражение if возвращает значение. Это означает, что функцию eval можно переписать, используя синтаксис тела-выражения без оператора return и фигурных скобок, с выражением if вместо тела функции.

Листинг 2.19. Использование выражения if, возвращающего значения

```
fun eval(e: Expr): Int =
  if (e is Num) {
    e.value
  } else if (e is Sum) {
    eval(e.right) + eval(e.left)
  } else {
    throw IllegalArgumentException("Unknown expression")
  }

>>> println(eval(Sum(Num(1), Num(2))))
3
```

Фигурные скобки необязательны, если тело функции состоит только из одного выражения if. Если одна из веток if является блоком, её результатом будет последнее выражение в блоке.

Давайте улучшим этот код, используя оператор when.

Листинг 2.20. Использование when вместо каскада выражений if

```
fun eval(e: Expr): Int =
  when (e) {
    is Num ->              <— Ветка «when» проверяет тип аргумента
      e.value              <— Используется автоматическое приведение типов
```

```
    is Sum ->                                    <— Ветка «when» проверяет тип аргумента
      eval(e.right) + eval(e.left)               <—Используется автоматическое приведение типов
    else ->
      throw IllegalArgumentException("Unknown expression")
  }
```

Выражение when не ограничивается проверкой равенства значений, как вы уже заметили. Здесь мы использовали другую форму ветвления when, проверяющую тип аргумента when. Так же, как в примере с if (листинг 2.19), проверка типа вызывает автоматическое приведение, поэтому к свойствам Num и Sum можно обращаться без дополнительных преобразований.

Сравните две последние версии Kotlin-функции eval и подумайте, как заменить последовательности операторов if в вашем собственном коде. Для реализации сложной логики в теле ветки можно использовать блочное выражение. Давайте посмотрим, как это работает.

2.3.7. Блоки в выражениях «if» и «when»

Оба выражения – if и when – позволяют определять ветви в виде блоков. Результатом такой ветви станет результат последнего выражения в блоке. Чтобы добавить журналирование в пример функции, достаточно оформить ветви в виде блоков и возвращать из них желаемое значение, как и раньше.

Листинг 2.21. Оператор when с составными операциями в ветках

```
fun evalWithLogging(e: Expr): Int =
  when (e) {
    is Num -> {
      println("num: ${e.value}")
      e.value                                    <— Это последнее выражение в блоке, функция
    }                                               вернет его значение, если e имеет тип Num
    is Sum -> {
      val left = evalWithLogging(e.left)
      val right = evalWithLogging(e.right)
      println("sum: $left + $right")
      left + right                               <— Функция вернет значение этого выражения,
    }                                               если e имеет тип Sum
    else -> throw IllegalArgumentException("Unknown expression")
  }
```

Теперь можно заглянуть в журнал и посмотреть, какие записи добавила функция evalWithLogging, чтобы понять, как выполнялись вычисления:

```
>>> println(evalWithLogging(Sum(Sum(Num(1), Num(2)), Num(4))))
num: 1
```

```
num: 2
sum: 1 + 2
num: 4
sum: 3 + 4
7
```

Правило «последнее выражение в блоке является результатом» действует всегда, когда используется блок и ожидается результат. Как будет показано в конце этой главы, то же правило относится к предложениям `try` и `catch`, а в главе 5 демонстрируется применение этого правила к лямбда-выражениям. Но, как упоминалось в разделе 2.2, это правило не выполняется для обычных функций. Функция может обладать телом-выражением, которое не может быть блоком, или телом-блоком с оператором `return` внутри.

Вы уже знаете, как Kotlin выбирает правильное значение из нескольких. Теперь настало время узнать о работе с последовательностями элементов.

2.4. Итерации: циклы «while» и «for»

Среди всех особенностей языка, обсуждавшихся в этой главе, итерации в Kotlin больше всего похожи на итерации в Java. Цикл `while` действует так же, как в Java, поэтому мы отведем ему совсем немного места в начале этого раздела. Цикл `for` существует только в одной форме, эквивалентной циклу `for-each` в Java. Он записывается в форме `for <элемент> in <элементы>`, как в C#. Чаще всего этот цикл используется для обхода элементов коллекций – как в Java. Мы также рассмотрим другие способы организации итераций.

2.4.1. Цикл «while»

В языке Kotlin есть циклы `while` и `do-while`, и их синтаксис не отличается от синтаксиса соответствующих циклов в Java:

```
while (condition) {
  /*...*/
}
```

Тело цикла выполняется, пока условие остается истинным

```
do {
  /*...*/
} while (condition)
```

Тело выполняется первый раз безусловно. После этого оно выполняется, пока условие остается истинным

Kotlin не добавляет ничего нового в эти простые циклы, поэтому не будем задерживаться на них, а перейдем к обсуждению вариантов использования цикла `for`.

2.4.2. Итерации по последовательности чисел: диапазоны и прогрессии

Как мы только что отметили, в Kotlin нет привычного для Java цикла `for`, включающего выражения инициализации переменной цикла, её изменения на каждом шаге и проверки условия выхода из цикла. Для таких случаев в Kotlin используется понятие диапазонов (ranges).

Диапазон представляет собой интервал между двумя значениями, обычно числовыми: началом и концом. Диапазоны определяются с помощью оператора `..`:

```
val oneToTen = 1..10
```

Обратите внимание, что диапазоны в Kotlin – *закрытые* или *включающие*, т. е. второе значение всегда является частью диапазона.

Самое элементарное действие, которое можно делать с диапазонами целых чисел, – выполнить последовательный обход всех значений. Если есть возможность обойти все значения диапазона, такой диапазон называется *прогрессией*.

Давайте используем диапазон целых чисел, чтобы сыграть в Fizz-Buzz. Эта игра – хороший способ скоротать долгое путешествие на машине и вспомнить забытые навыки деления. Игроки договариваются о диапазоне чисел и затем ходят по очереди, начиная с единицы и заменяя число, кратное трем, словом *fizz*, и число, кратное пяти, словом *buzz*. Если число кратно и трем, и пяти, оно заменяется парой слов *fizz buzz*.

Функция в листинге 2.22 выводит правильные ответы для чисел от 1 до 100. Обратите внимание, как проверяются условия в выражении `when` без аргументов.

Листинг 2.22. Применение оператора `when` в реализации игры Fizz-Buzz

```
fun fizzBuzz(i: Int) = when {
  i % 15 == 0 -> "FizzBuzz "
  i % 3 == 0 -> "Fizz "
  i % 5 == 0 -> "Buzz "
  else -> "$i "
}

>>> for (i in 1..100) {
...     print(fizzBuzz(i))
...   }

1 2 Fizz 4 Buzz Fizz 7 ...
```

Предположим, после часа езды вам надоели эти правила и вы решили немного усложнить их: счет должен выполняться в обратном порядке, начиная со 100, и рассматриваться должны только четные числа.

Листинг 2.23. Итерации по диапазону с шагом

```
>>> for (i in 100 downTo 1 step 2) {
  ... print(fizzBuzz(i))
  ... }
Buzz 98 Fizz 94 92 FizzBuzz 88 ...
```

Здесь выполняется обход прогрессии с *шагом*, что позволяет пропускать некоторые значения. Шаг также может быть отрицательным – в таком случае обход прогрессии будет выполняться в обратном направлении. Выражение `100 downTo 1` в данном примере – это убывающая прогрессия (с шагом –1). Оператор `step` меняет абсолютное значение шага на 2, не меняя направления (фактически он устанавливает шаг равным –2).

Как мы уже упоминали ранее, оператор `..` всегда создает закрытый диапазон, включающий конечное значение (справа от `..`). Во многих случаях удобнее использовать полузакрытые диапазоны, не включающие конечного значения. Создать такой диапазон можно с помощью функции `until`. Например, выражение `for (x in 0 until size)` эквивалентно выражению `for (x in 0..size-1)`, но выглядит проще. Позже, в разделе 3.4.3, вы узнаете больше о синтаксисе операторов `downTo`, `step` и `until` из этих примеров.

Вы увидели, как диапазоны и прогрессии помогли нам справиться с усложненными правилами игры Fizz-Buzz. Теперь давайте рассмотрим другие примеры использования цикла `for`.

2.4.3. Итерации по элементам словарей

Как отмечалось выше, цикл `for...in` чаще всего используется для обхода элементов коллекций. Этот цикл действует в точности, как в языке Java, поэтому мы не будем уделять ему много внимания, а просто посмотрим, как с его помощью организовать обход элементов словаря.

В качестве примера рассмотрим небольшую программу, которая выводит коды символов в двоичном представлении. Эти двоичные представления хранятся в словаре (такой способ выбран исключительно для иллюстрации). Нижеследующий код создает словарь, заполняет двоичными представлениями некоторых символов и затем выводит его содержимое.

Листинг 2.24. Создание словаря и обход его элементов

```
val binaryReps = TreeMap<Char, String>()   <— Словарь TreeMap хранит ключи в порядке сортировки
for (c in 'A'..'F') {                      <— Обход диапазона символов от A до F
```

```
  val binary = Integer.toBinaryString(c.toInt())
  binaryReps[c] = binary
}

for ((letter, binary) in binaryReps) {
  println("$letter = $binary")
}
```

Преобразует ASCII-код в двоичное представление

Сохраняет в словаре значение с ключом в с

Обход элементов словаря; ключ и значение присваиваются двум переменным

Оператор диапазона `..` работает не только для чисел, но и для символов. Здесь он использован для определения диапазона всех символов от *A* до *F* включительно.

Листинг 2.24 демонстрирует, как цикл `for` позволяет распаковать элемент коллекции, участвующей в итерациях (в данном случае это коллекция пар ключ/значение, то есть словарь). Результат распаковки сохраняется в двух отдельных переменных: `letter` принимает ключ, а `binary` – значение. Позже, в разделе 7.4.1, вы узнаете больше о синтаксисе распаковки.

Еще один интересный прием в листинге 2.24 – использование сокращенного синтаксиса для получения и изменения значений в словаре по ключу. Вместо вызова методов `get` и `put` можно писать `map[key]`, чтобы прочитать значение, и `map[key] = value` для его изменения. Код

```
binaryReps[c] = binary
```

эквивалентен следующему коду на Java:

```
binaryReps.put(c, binary)
```

Пример в листинге 2.24 выведет следующие результаты (мы разместили их в двух колонках вместо одной):

```
A = 1000001 D = 1000100
B = 1000010 E = 1000101
C = 1000011 F = 1000110
```

Тот же синтаксис распаковки при обходе коллекции можно применить, чтобы сохранить индекс текущего элемента. Это избавит вас от необходимости создавать отдельную переменную для хранения индекса и увеличивать её вручную:

```
val list = arrayListOf("10", "11", "1001")
for ((index, element) in list.withIndex()) {
  println("$index: $element")
}
```

Обход коллекции с сохранением индекса

Код выведет то, что вы ожидаете:

```
0: 10
1: 11
2: 1001
```

Тонкости метода `withIndex` мы обсудим в следующей главе.

Вы увидели, как ключевое слово `in` используется для обхода диапазона или коллекции. Кроме того, `in` позволяет проверить вхождение значения в диапазон или коллекцию.

2.4.4. Использование «in» для проверки вхождения в диапазон или коллекцию

Для проверки вхождения значения в диапазон можно использовать оператор `in` или его противоположность – `!in`, проверяющий отсутствие значения в диапазоне. Вот как можно использовать `in` для проверки вхождения символа в диапазон.

Листинг 2.25. Проверка вхождения в диапазон с помощью `in`

```
fun isLetter(c: Char) = c in 'a'..'z' || c in 'A'..'Z'
fun isNotDigit(c: Char) = c !in '0'..'9'

>>> println(isLetter('q'))
true
>>> println(isNotDigit('x'))
true
```

Проверить, является ли символ буквой, очень просто. Под капотом не происходит ничего замысловатого: оператор проверяет, находится ли код символа где-то между кодом первой и последней буквы. Но эта логика скрыта в реализации классов диапазонов в стандартной библиотеке:

```
c in 'a'..'z'          <— Преобразуется в a <= c && c <= z
```

Операторы `in` и `!in` также можно использовать в выражении `when`.

Листинг 2.26. Использование проверки `in` в ветках `when`

```
fun recognize(c: Char) = when (c) {
  in '0'..'9' -> "It's a digit!"          <— Проверяет вхождение значения в диапазон от 0 до 9
  in 'a'..'z', in 'A'..'Z' -> "It's a letter!"   <— Можно совместить несколько диапазонов
  else -> "I don't know..."
}

>>> println(recognize('8'))
It's a digit!
```

Но диапазоны не ограничиваются и символами. Если есть класс, который поддерживает сравнение экземпляров (за счет реализации интерфейса `java.lang.Comparable`), вы сможете создавать диапазоны из объектов

этого типа, но не можете перечислить всех объектов в таких диапазонах. Подумайте сами: сможете ли вы, к примеру, перечислить все строки между «Java» и «Kotlin»? Нет, не сможете. Но вы по-прежнему сможете убедиться в принадлежности объекта диапазону с помощью оператора `in`:

```
>>> println("Kotlin" in "Java".."Scala")
true
```

Обратите внимание, что здесь строки сравниваются по алфавиту, потому что именно так класс `String` реализует интерфейс `Comparable`.

Та же проверка `in` будет работать с коллекциями:

```
>>> println("Kotlin" in setOf("Java", "Scala"))
false
```

В разделе 7.3.2 вы увидите, как использовать диапазоны и прогрессии со своими собственными типами данных и что любые объекты можно использовать в проверках `in`.

В этой главе мы рассмотрим еще одну группу Java-инструкций: инструкции для работы с исключениями.

2.5. Исключения в Kotlin

Обработка исключений в Kotlin выполняется так же, как в Java и многих других языках. Функция может завершиться обычным способом или возбудить исключение в случае ошибки. Код, вызывающий функцию, может перехватить это исключение и обработать его; если этого не сделать, исключение продолжит свое движение вверх по стеку.

Инструкции для работы с исключениями в Kotlin в своем простейшем виде похожи на аналогичные инструкции в Java. Способ возбуждения исключения вряд ли вас удивит:

```
if (percentage !in 0..100) {
  throw IllegalArgumentException(
    "A percentage value must be between 0 and 100: $percentage")
}
```

Как и со всеми другими классами, для создания экземпляра исключения не требуется использовать ключевое слово `new`.

В отличие от Java, конструкция `throw` в Kotlin является выражением и может использоваться в таком качестве в составе других выражений:

```
val percentage =
  if (number in 0..100)
    number
  else
```

```
    throw IllegalArgumentException(                    <— «throw» – это выражение
      "A percentage value must be between 0 and 100: $number")
```

В этом примере, если условие выполнится, программа поведет себя правильно и инициализирует переменную `percentage` значением `number`. В противном случае будет возбуждено исключение и переменная не будет инициализирована. Технические детали применения конструкции `throw` в составе других выражений мы обсудим в разделе 6.2.6.

2.5.1. «try», «catch» и «finally»

Как и в Java, для обработки исключений используется выражение `try` с разделами `catch` и `finally`. Как это делается, можно увидеть в листинге 2.27 – демонстрируемая функция читает строку из заданного файла, пытается преобразовать её в число и возвращает число или `null`, если строка не является допустимым представлением числа.

Листинг 2.27. Использование `try` как в Java

```
fun readNumber(reader: BufferedReader): Int? {   <— Не требуется явно указывать, какое
  try {                                              исключение может возбудить функция
    val line = reader.readLine()
    return Integer.parseInt(line)
  }
  catch (e: NumberFormatException) {             <— Тип исключения записывается
    return null                                      справа
  }
  finally {                                      <— Блок «finally» действует так же,
    reader.close()                                   как в Java
  }
}
```

```
>>> val reader = BufferedReader(StringReader("239"))
>>> println(readNumber(reader))
239
```

Самое большое отличие от Java заключается в отсутствии конструкции `throws` в сигнатуре функции: в Java вам пришлось бы добавить `throws IOException` после объявления функции. Это необходимо, потому что исключение `IOException` является *контролируемым*. В Java такие исключения требуется обрабатывать явно. Вы должны объявить все контролируемые исключения, возбуждаемые функцией, а при вызове из другой функции – обрабатывать все её контролируемые исключения или объявить, что эта другая функция тоже может их возбуждать.

Подобно многим другим современным языкам для JVM, Kotlin не делает различий между контролируемыми и неконтролируемыми исключе-

ниями. Исключения, возбуждаемые функцией, не указываются – можно обрабатывать или не обрабатывать любые исключения. Такое проектное решение основано на практике использования контролируемых исключений в Java. Как показал опыт, правила языка Java часто требуют писать массу бессмысленного кода для повторного возбуждения или игнорирования исключений, и эти правила не всегда в состоянии защитить вас от возможных ошибок.

Например, исключение `NumberFormatException` в листинге 2.27 не является контролируемым. Поэтому компилятор Java не заставит вас перехватывать его, и вы легко сможете столкнуться с этим исключением во время выполнения. Это неудачное решение, поскольку ввод неправильных данных – довольно распространенная ситуация, которая должна быть корректно обработана. В то же время метод `BufferedReader.close` может возбудить контролируемое исключение `IOException`, которое нужно обработать. Большинство программ не сможет предпринять каких-либо осмысленных действий в ответ на неудачную попытку закрыть поток, и, следовательно, для обработки исключения, возбуждаемого методом `close`, всегда будет использоваться один и тот же код.

А что насчет конструкции `try-with-resources` из Java 7? В Kotlin нет никакого специального синтаксиса для этого; эта конструкция реализована в виде библиотечной функции. Вы поймете, как это возможно, читая раздел 8.2.5.

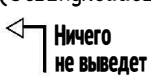
2.5.2. «try» как выражение

Для демонстрации еще одного существенного различия между Java и Kotlin немного модифицируем пример. Удалим блок `finally` (вы уже знаете, как он действует) и добавим код, выводящий число, прочитанное из файла.

Листинг 2.28. Использование try в качестве выражения

```
fun readNumber(reader: BufferedReader) {
    val number = try {
        Integer.parseInt(reader.readLine())
    } catch (e: NumberFormatException) {
        return
    }
    println(number)
}

>>> val reader = BufferedReader(StringReader("not a number"))
>>> readNumber(reader)
```



Ключевое слово `try` в языке Kotlin наряду с `if` и `when` является выражением, значение которого можно присвоить переменной. В отличие от `if`, тело выражения всегда нужно заключать в фигурные скобки. Как и в остальных случаях, если тело содержит несколько выражений, итоговым результатом станет значение последнего выражения.

В листинге 2.28 в блоке `catch` находится оператор `return`, поэтому выполнение функции прервется после выполнения блока `catch`. Если нужно, чтобы функция продолжила выполнение после выхода из блока `catch`, он тоже должен вернуть значение – значение последнего выражения в нем. Вот как это работает.

Листинг 2.29. Возврат значения из блока catch

```
fun readNumber(reader: BufferedReader) {
  val number = try {
    Integer.parseInt(reader.readLine())
  } catch (e: NumberFormatException) {
    null
  }
  println(number)
}
```

Если исключение не возникнет, будет возвращено это значение

Если исключение возникнет, будет возвращено значение null

```
>>> val reader = BufferedReader(StringReader("not a number"))
>>> readNumber(reader)
null
```

Возбудит исключение, поэтому функция выведет «null»

Если блок `try` выполнится нормально, последнее выражение в нём станет его результатом. Если возникнет исключение, результатом станет последнее выражение в соответствующем блоке `catch`. Если в листинге 2.29 возникнет исключение `NumberFormatException`, результатом станет значение `null`.

Если вам не терпится, вы уже можете начать писать программы на языке Kotlin в стиле, похожем на Java. По мере чтения этой книги вы будете постепенно менять свой привычный образ мышления и учиться использовать всю мощь нового языка.

2.6. Резюме

- Объявление функции начинается с ключевого слова `fun`. Ключевые слова `val` и `var` служат для объявления констант и изменяемых переменных соответственно.
- Строковые шаблоны помогают избежать лишних операций конкатенации строк. Добавьте к переменной префикс `$` или заключите

выражение в `${}` – и соответствующее значение будет подставлено в строку.

- В Kotlin можно очень лаконично описывать классы объектов-значений.
- Знакомый оператор `if` теперь является выражением и возвращает значение.
- Выражение `when` – более мощный аналог выражения `switch` в Java.
- Исчезла необходимость явно выполнять приведение типа после проверки типа переменной: благодаря механизму автоматического приведения типов компилятор сделает это за вас.
- Циклы `for`, `while` и `do-while` похожи на свои аналоги в Java, но цикл `for` стал более удобным, особенно при работе со словарями или коллекциями с индексом.
- Лаконичный синтаксис `1..5` создает диапазон. Диапазоны и прогрессии позволяют использовать единый синтаксис и набор абстракций для цикла `for`, а с помощью операторов `in` и `!in` можно проверить принадлежность значения диапазону.
- Обработка исключений в Kotlin выполняется почти так же, как в Java, лишь с той разницей, что Kotlin не требует перечислять контролируемые исключения, которые может возбуждать функция.

Глава 3

Определение и вызов функций

В этой главе:

- функции для работы с коллекциями, строками и регулярными выражениями;
- именованные аргументы, значения параметров по умолчанию и инфиксный синтаксис вызова;
- использование Java-библиотек в Kotlin с помощью функций-расширений и свойств-расширений;
- структурирование кода с помощью функций верхнего уровня, а также локальных функций и свойств.

Надеемся, теперь вы чувствуете себя так же комфортно с Kotlin как с Java. Вы видели, как знакомые понятия из Java транслируются в Kotlin, и как в Kotlin они часто становятся более лаконичными и читабельными.

В этой главе вы увидите, как Kotlin может улучшить один из ключевых элементов любой программы: объявление и вызов функций. Также мы рассмотрим возможности использования библиотек Java в стиле Kotlin с помощью функций-расширений – это позволяет получить все преимущества Kotlin в смешанных проектах.

Чтобы сделать обсуждение полезным и менее абстрактным, выберем нашей предметной областью коллекции, строки и регулярные выражения. Сначала давайте посмотрим, как создаются коллекции в Kotlin.

3.1. Создание коллекций в Kotlin

Прежде чем вы сможете делать интересные вещи с коллекциями, вам нужно научиться создавать их. В разделе 2.3.3 вы уже сталкивались с одним способом создания множеств: функцией `setOf`. В тот раз мы создавали множество цветов, но сейчас давайте для простоты использовать цифры:

```
val set = hashSetOf(1, 7, 53)
```

Списки и словари создаются аналогично:

```
val list = arrayListOf(1, 7, 53)
val map = hashMapOf(1 to "one", 7 to "seven", 53 to "fifty-three")
```

Обратите внимание, что `to` – это не особая конструкция, а обычная функция. Мы вернемся к ней позже в этой главе.

Сможете ли вы угадать классы объектов, созданных выше? Выполните следующий код, чтобы все увидеть своими глазами:

```
>>> println(set.javaClass)
class java.util.HashSet

>>> println(list.javaClass)
class java.util.ArrayList

>>> println(map.javaClass)
class java.util.HashMap
```

Свойство javaClass эквивалентно методу getClass() в Java

Как можно заметить, Kotlin использует стандартные классы коллекций из Java. Это хорошая новость для Java-разработчиков: в языке Kotlin нет собственных классов коллекций. Все, что вы знаете о коллекциях в Java, верно и тут.

Почему в Kotlin нет своих коллекций? Потому что использование стандартных Java-коллекций значительно упрощает взаимодействие с Java-кодом. Вам не нужно конвертировать коллекции тем или иным способом, вызывая Java-функции из Kotlin или наоборот.

Хотя коллекции в Kotlin представлены теми же классами, что и в Java, в Kotlin они обладают гораздо более широкими возможностями. Например, можно получить последний элемент в списке или найти максимальное значение в коллекции чисел:

```
>>> val strings = listOf("first", "second", "fourteenth")

>>> println(strings.last())
fourteenth

>>> val numbers = setOf(1, 14, 2)

>>> println(numbers.max())
14
```

В этой главе мы подробно рассмотрим механизм работы этого примера и узнаем, откуда берутся новые методы в Java-классах.

В следующих главах, когда речь пойдёт о лямбда-выражениях, вы познакомитесь с ещё более широким кругом возможностей коллекций – но и там мы будем продолжать использовать те же стандартные Java-классы коллекций. О том, как представлены классы коллекций Java в системе типов языка Kotlin, вы узнаете в разделе 6.3.

Прежде чем обсуждать работу магических функций `last` и `max` для работы с коллекциями Java, познакомимся с некоторыми новыми понятиями, связанными с объявлением функций.

3.2. Упрощение вызова функций

Теперь, когда вы знаете, как создать коллекцию элементов, давайте сделаем что-нибудь несложное – например, выведем её содержимое. Не волнуйтесь, если эта задача кажется слишком простой: в процессе её решения вы встретитесь со множеством важных понятий.

В Java-коллекциях есть реализация по умолчанию для метода `toString`, но её формат вывода фиксирован и не всегда подходит:

```
>>> val list = listOf(1, 2, 3)
>>> println(list)              <— Вызов метода toString()
[1, 2, 3]
```

Допустим, нам нужно, чтобы элементы отделялись друг от друга точкой с запятой, а вся коллекция заключалась в круглые скобки вместо квадратных: `(1; 2; 3)`. Для решения этой проблемы в Java-проектах используются сторонние библиотеки, такие как Guava и Apache Commons, или переопределяется логика метода внутри проекта. В Kotlin способ решения этой задачи – часть стандартной библиотеки.

В этом разделе мы сами реализуем нужную функцию. Начнем с очевидной реализации, не использующей особенностей Kotlin, которые упрощают объявление функций, а затем перепишем её в более идиоматичном стиле.

Функция `joinToString` в листинге 3.1 добавляет элементы из коллекции в объект `StringBuilder`, вставляя разделитель между ними, префикс в начало и постфикс в конец.

Листинг 3.1. Начальная реализация `joinToString`

```
fun <T> joinToString(
  collection: Collection<T>,
  separator: String,
  prefix: String,
  postfix: String
): String {

  val result = StringBuilder(prefix)

  for ((index, element) in collection.withIndex()) {
    if (index > 0) result.append(separator)
    result.append(element)
```

<— Не нужно вставлять разделитель перед первым элементом

```
    }

    result.append(postfix)
    return result.toString()
}
```

Это – обобщенная функция: она работает с коллекциями, содержащими элементы любого типа. Синтаксис использования обобщенных типов похож на Java. (Более подробное обсуждение обобщенных типов станет темой главы 9.)

Давайте убедимся, что функция работает, как задумано:

```
>>> val list = listOf(1, 2, 3)
>>> println(joinToString(list, "; ", "(", ")"))
(1; 2; 3)
```

Реализация замечательно работает, и нам почти не придётся её менять. Но давайте задумаемся: как изменить объявление функции, чтобы сделать её вызов менее многословным? Возможно, есть способ не передавать все четыре аргумента в каждый вызов функции. Посмотрим, как это сделать.

3.2.1. Именованные аргументы

Первая проблема, которую мы решим, касается читабельности вызовов функций. Например, взгляните на следующий вызов функции `joinToString`:

```
joinToString(collection, " ", " ", ".")
```

Можете ли вы сказать, каким параметрам соответствуют все эти строковые значения? Элементы будут разделяться пробелом или точкой? На эти вопросы трудно ответить, не имея сигнатуры функции перед глазами. Возможно, вы её помните или ваша IDE сможет помочь вам. Но если вы смотрите только на код вызова, нельзя сказать ничего определенного.

Эта проблема особенно часто возникает при использовании логических флагов. В качестве решения некоторые стили оформления Java-кода рекомендуют использовать перечисления вместо логического типа. Другие даже требуют указывать имена параметров прямо в комментариях, как в следующем примере:

```
/* Java */
joinToString(collection, /* separator */ " ", /* prefix */ " ",
    /* postfix */ ".");
```

Kotlin предлагает более изящное решение:

```
joinToString(collection, separator = " ", prefix = " ", postfix = ".")
```

Вызывая функции, написанные на Kotlin, можно указывать имена некоторых аргументов. Когда указано имя одного аргумента, то необходимо

указать имена всех аргументов, следующих за ним, чтобы избежать путаницы.

> **Совет.** IntelliJ IDEA может автоматически обновлять имена аргументов при изменении имен параметров вызываемой функции. Для этого используйте операции **Rename or Change Signature** (Переименовать или изменить сигнатуру) вместо изменения имен вручную.

> **Внимание.** К сожалению, вы не можете использовать именованных аргументов при вызове методов, написанных на Java, в том числе методов из JDK и фреймворка Android. Поддержка хранения имен параметров в файлах *.class* появилась только в версии Java 8, а Kotlin поддерживает совместимость с Java 6. В результате компилятор не распознает имен параметров в вызове и не сопоставляет их с определением метода.

Именованные аргументы особенно хорошо работают вместе со значениями по умолчанию, которые мы рассмотрим далее.

3.2.2. Значения параметров по умолчанию

Другая распространенная проблема Java – избыток перегруженных методов в некоторых классах. Только взгляните на класс `java.lang.Thread` и его восемь конструкторов (`http://mng.bz/4KZC`)! Перегрузка может использоваться ради обратной совместимости, для удобства пользователей или по иным причинам, но итог один – дублирование. Имена параметров и типы повторяются снова и снова, и если вы добропорядочный гражданин, вам придется повторять большую часть документации в каждой перегрузке. С другой стороны, при использовании перегруженной версии, в которой отсутствуют некоторые параметры, не всегда понятно, какие значения будут для них использованы.

В Kotlin часто можно избежать перегрузки благодаря возможности указывать значения параметров по умолчанию в объявлениях функций. Давайте воспользуемся этим приемом, чтобы усовершенствовать функцию `joinToString`. В большинстве случаев строки можно разделять запятыми и не использовать префикса и постфикса. Давайте объявим эти значения параметрами по умолчанию.

Листинг 3.2. Объявление функции `joinToString()` со значениями параметров по умолчанию

```
fun <T> joinToString(
  collection: Collection<T>,
  separator: String = ", ",
  prefix: String = "",
  postfix: String = ""
): String
```

Параметры со значениями по умолчанию

Теперь функцию можно вызвать с аргументами или опустить некоторые из них:

```
>>> joinToString(list, ", ", "", "")
1, 2, 3
>>> joinToString(list)
1, 2, 3
>>> joinToString(list, "; ")
1; 2; 3
```

При использовании обычного синтаксиса вызова аргументы должны следовать в том же порядке, что и параметры в объявлении функции, а опускать можно только аргументы в конце. При использовании именованных аргументов можно опустить аргументы из середины списка и указать только нужные, причем в любом порядке:

```
>>> joinToString(list, suffix = ";", prefix = "# ")
# 1, 2, 3;
```

Обратите внимание, что значения параметров по умолчанию определяются в вызываемой функции, а не в месте вызова. Если изменить значение по умолчанию и заново скомпилировать класс, содержащий функцию, то в вызовах, где значение параметра не указано явно, будет использовано новое значение по умолчанию.

Значения по умолчанию и Java

Поскольку в Java отсутствует понятие «значения параметров по умолчанию», вам придется явно указывать все значения при вызове функции Kotlin со значениями параметров по умолчанию из Java. Если такие функции приходится часто вызывать из Java и желательно упростить их вызов из Java-кода, отметьте их аннотацией `@JvmOverloads`. Она требует от компилятора создать перегруженные Java-методы, опуская каждый из параметров по одному, начиная с последнего.

Например, отметив аннотацией `@JvmOverloads` функцию `joinToString`, вы получите следующие перегруженные версии:

```
/* Java */
String joinToString(Collection<T> collection, String separator,
  String prefix, String postfix);

String joinToString(Collection<T> collection, String separator,
  String prefix);

String joinToString(Collection<T> collection, String separator);

String joinToString(Collection<T> collection);
```

Каждая версия использует значения по умолчанию для параметров, не указанных в сигнатуре.

До сих пор мы трудились над своей вспомогательной функцией, не обращая особого внимания на окружающий контекст. Наверняка она должна быть методом какого-то класса, не показанного в листинге с примером, верно? На самом деле в Kotlin это не обязательно.

3.2.3. Избавление от статических вспомогательных классов: свойства и функции верхнего уровня

Мы знаем, что Java как объектно-ориентированный язык требует помещать весь код в методах классов. Это хорошая идея, но в реальности почти во всех больших проектах есть много кода, который нельзя однозначно отнести к одному классу. Иногда операция работает с объектами двух разных классов, которые играют для неё одинаково важную роль. Иногда есть один основной объект, но вы не хотите усложнять его API, добавляя операцию как метод экземпляра.

В конечном итоге появляется множество классов, которые не имеют ни состояния, ни методов экземпляров и используются только как контейнеры для кучи статических методов. Идеальный пример – класс `Collections` в JDK. Чтобы обнаружить другие примеры в своем коде, ищите классы, которые содержат в имени слово `Util`.

В Kotlin не нужно создавать этих бессмысленных классов. Вместо этого можно помещать функции непосредственно на верхнем уровне файла с исходным кодом, за пределами любых классов. Такие функции всё ещё остаются членами пакета, объявленного в начале файла, и их всё ещё нужно импортировать для использования в других пакетах, но ненужный дополнительный уровень вложенности исчезает.

Давайте поместим функцию `joinToString` прямо в пакет `strings`. Создайте файл *join.kt* со следующим содержимым.

Листинг 3.3. Объявление `joinToString` как функции верхнего уровня

```
package strings
fun joinToString(...): String { ... }
```

Как это работает? Вам должно быть известно, что при компиляции файла будут созданы некоторые классы, поскольку JVM может выполнять только код в классах. Если вы работаете только с Kotlin, этого знания вам будет достаточно. Но если функцию нужно вызвать из Java, вы должны понимать, что получится в результате компиляции. Чтобы выяснить это, давайте рассмотрим Java-код, который будет компилироваться в такой же класс:

```
/* Java */
package strings;
public class JoinKt {
```

Соответствует имени файла join.kt из листинга 3.3

```
  public static String joinToString(...) { ... }
}
```

Как видите, компилятор Kotlin генерирует имя класса, соответствующее имени файла с функцией. Все функции верхнего уровня в файле компилируются в статические методы этого класса. Поэтому вызов такой функции из Java выглядит так же, как вызов любого другого статического метода:

```
/* Java */
import strings.JoinKt;
...
JoinKt.joinToString(list, ", ", "", "");
```

Изменение имени класса в файле

Чтобы изменить имя класса с Kotlin-функциями верхнего уровня, нужно добавить в файл аннотацию `@JvmName`. Поместите её в начало файла перед именем пакета:

```
@file:JvmName("StringFunctions")

package strings

fun joinToString(...): String { ... }
```

Аннотация для объявления имени класса

Выражение package следует за аннотациями уровня файла

Теперь функцию можно вызвать так:

```
/* Java */
import strings.StringFunctions;
StringFunctions.joinToString(list, ", ", "", "");
```

Синтаксис аннотаций мы подробно обсудим в главе 10.

Свойства верхнего уровня

Как и функции, свойства можно объявлять на верхнем уровне файла. Хранение отдельных фрагментов данных вне класса требуется не так часто, но все же бывает полезно.

Например, `var`-свойство можно использовать для подсчета выполненных операций:

```
var opCount = 0

fun performOperation() {
    opCount++
    // ...
}

fun reportOperationCount() {
```

Объявление свойства верхнего уровня

Изменение значения свойства

```
    println("Operation performed $opCount times")
}
```

Чтение значения свойства

Значение такого свойства будет храниться в статическом поле.

Кроме того, свойства верхнего уровня позволяют определять константы в коде:

```
val UNIX_LINE_SEPARATOR = "\n"
```

По умолчанию к свойствам верхнего уровня, как и любым другим, можно обращаться из Java через методы доступа (методы чтения `val`-свойств и пары методов чтения/записи `var`-свойств). Если нужно сделать константу доступной для Java-кода как поле с модификаторами `public static final`, то её использование можно сделать более естественным, добавив перед ней модификатор `const` (это возможно для свойств простых типов и типа `String`):

```
const val UNIX_LINE_SEPARATOR = "\n"
```

Это эквивалентно следующему коду на Java:

```
/* Java */
public static final String UNIX_LINE_SEPARATOR = "\n";
```

Мы усовершенствовали начальную версию вспомогательной функции `joinToString`. А теперь давайте узнаем, как сделать её ещё удобнее.

3.3. Добавление методов в сторонние классы: функции-расширения и свойства-расширения

Одна из главных особенностей языка Kotlin – простота интеграции с существующим кодом. Даже проекты, написанные исключительно на языке Kotlin, строятся на основе Java-библиотек, таких как JDK, фреймворк Android и другие. И когда код на Kotlin интегрируется в Java-проект, то приходится иметь дело с существующим кодом, который не был или не будет переводиться на язык Kotlin. Правда, здорово было бы использовать все прелести языка Kotlin при работе с этими API без их переписывания? Как раз для этого и созданы функции-расширения.

По сути, *функция-расширение* – очень простая штука: это функция, которая может вызываться как член класса, но определена за его пределами. Для демонстрации добавим метод получения последнего символа в строке:

```
package strings

fun String.lastChar(): Char = this.get(this.length - 1)
```

Чтобы определить такую функцию, достаточно добавить имя расширяемого класса или интерфейса перед именем функции. Имя класса назы-

вается *типом-получателем* (receiver type); значение, для которого вызывается функция-расширение, называется *объектом-получателем* (receiver object). Это проиллюстрировано на рис. 3.1.

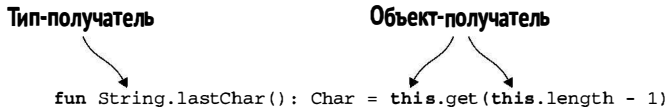


Рис. 3.1. Тип-получатель – это тип, для которого определяется расширение, а объект-получатель – это экземпляр данного типа

Функцию можно вызывать, используя тот же синтаксис, что и для обычных членов класса:

```
>>> println("Kotlin".lastChar())
n
```

В этом примере класс `String` – это тип-получатель, а строка `"Kotlin"` – объект-получатель.

В каком-то смысле мы добавили собственный метод в класс `String`. Хотя класс `String` не принадлежит нам и у нас даже может не быть его исходного кода, в него все равно можно добавить методы, необходимые в проекте. Не важно даже, написан класс `String` на Java, на Kotlin или другом JVM-языке (например, Groovy). Если он компилируется в Java-класс, мы можем добавлять в него свои расширения.

В теле функции-расширения ключевое слово `this` используется так же, как в обычном методе. И так же, как в обычном методе, его можно опустить:

```
package strings

fun String.lastChar(): Char = get(length - 1)
```

К методам объекта-получателя можно обращаться без «this»

В функции-расширении разрешено напрямую обращаться к методам и свойствам расширяемого класса точно так же, как в методах, определяемых в самом классе. Обратите внимание, что функции-расширения не позволяют нарушать правила инкапсуляции. В отличие от методов, объявленных в классе, функции-расширения не имеют доступа к закрытым или защищенным членам класса.

Далее мы будем использовать термин *метод* и для членов класса, и для функций-расширений. Например, мы можем сказать, что в теле функции-расширения можно вызвать любой метод получателя, подразумевая возможность вызова членов класса и функций-расширений. В точке вызова функции-расширения ничем не отличаются от членов класса, и зачастую не имеет значения, является ли конкретный метод членом класса или расширением.

3.3.1. Директива импорта и функции-расширения

Функция-расширение не становится автоматически доступной всему проекту; как любой другой класс или функцию, её необходимо импортировать. Такой порядок вещей помогает избежать случайных конфликтов имен. Kotlin позволяет импортировать отдельные функции, используя тот же синтаксис, что и для классов:

```
import strings.lastChar

val c = "Kotlin".lastChar()
```

Функцию-расширение также можно импортировать с помощью *:

```
import strings.*

val c = "Kotlin".lastChar()
```

Можно поменять имя импортируемого класса или функции, добавив ключевое слово `as`:

```
import strings.lastChar as last

val c = "Kotlin".last()
```

Изменение импортируемых имен полезно, когда в разных пакетах имеются функции с одинаковыми именами и их требуется использовать в одном файле. Для обычных классов и функций существует другое решение данной проблемы: можно использовать полностью квалифицированное имя класса или функции. Для функций-расширений правила синтаксиса требуют использовать короткое имя, поэтому ключевое слово `as` в инструкции импорта остаётся единственным способом разрешения конфликта.

3.3.2. Вызов функций-расширений из Java

Фактически функция-расширение – это самый обычный статический метод, которому в первом аргументе передается объект-приемник. Её вызов не предполагает создания объектов-адаптеров или других накладных расходов во время выполнения.

Благодаря этому использовать функции-расширения в Java довольно просто: нужно лишь вызвать статический метод, передав ему экземпляр объекта-приемника. По аналогии с функциями верхнего уровня, имя Java-класса с методом определяется по имени файла, где объявлена функция-расширение. Предположим, она объявлена в файле *StringUtil.kt*:

```
/* Java */
char c = StringUtilKt.lastChar("Java");
```

Эта функция-расширение объявлена как функция верхнего уровня, поэтому будет скомпилирована в статический метод. Вы можете статически импортировать метод `lastChar` в Java, что упростит вызов до `lastChar("Java")`. Этот код читается несколько хуже, чем его Kotlin-версия, но для Java такое использование идиоматично.

3.3.3. Вспомогательные функции как расширения

Теперь можно написать окончательную версию функции `joinToString`. Она будет практически совпадать с той, что имеется в стандартной библиотеке Kotlin.

Листинг 3.4. Функция `joinToString` как расширение

```
fun <T> Collection<T>.joinToString(
    separator: String = ", ",
    prefix: String = "",
    postfix: String = ""
): String {
  val result = StringBuilder(prefix)

  for ((index, element) in this.withIndex())
    if (index > 0) result.append(separator)
    result.append(element)
  }

  result.append(postfix)
  return result.toString()
}
```

Объявление функции-расширения для типа Collection<T>

Значения по умолчанию для параметров

«this» ссылается на объект-приемник: коллекцию элементов типа T

```
>>> val list = listOf(1, 2, 3)
>>> println(list.joinToString(separator = "; ",
... prefix = "(", postfix = ")"))
(1; 2; 3)
```

Мы создали расширение для коллекции элементов и определили значения по умолчанию для всех аргументов. Теперь `joinToString` можно вызывать как обычный член класса:

```
>>> val list = arrayListOf(1, 2, 3)
>>> println(list.joinToString(" "))
1 2 3
```

Поскольку функции-расширения фактически представляют собой синтаксический сахар, упрощающий вызов статических методов, в качестве типа-приемника можно указывать не просто класс, а более конкретный

тип. Допустим, нам понадобилось написать функцию `join`, которая может быть вызвана только для коллекции строк.

```
fun Collection<String>.join(
  separator: String = ", ",
  prefix: String = "",
  postfix: String = ""
) = joinToString(separator, prefix, postfix)

>>> println(listOf("one", "two", "eight").join(" "))
one two eight
```

Попытка вызвать эту функцию для списка объектов другого типа закончится ошибкой:

```
>>> listOf(1, 2, 8).join()
Error: Type mismatch: inferred type is List<Int> but Collection<String> was expected.
```

Статическая природа расширений также означает невозможность переопределения функций-расширений в подклассах. Чтобы убедиться в этом, приведем пример.

3.3.4. Функции-расширения не переопределяются

В Kotlin допускается переопределять функции-члены, но нельзя переопределить функцию-расширение. Предположим, у нас есть два класса – `View` и его подкласс `Button`, – и класс `Button` переопределяет функцию `click` суперкласса.

Листинг 3.5. Переопределение функции-члена класса

```
open class View {
  open fun click() = println("View clicked")
}

class Button: View() {                          <- Класс Button наследует View
  override fun click() = println("Button clicked")
}
```

Если объявить переменную типа `View`, в ней можно сохранить значение типа `Button`, поскольку `Button` – подтип `View`. При вызове обычного метода, такого как `click`, будет выполнен метод из класса `Button`, если он переопределен в этом классе:

```
>>> val view: View = Button()
>>> view.click()                <- Вызываемый метод определяется фактическим значением переменной «view»
Button clicked
```

Но это не работает для функций-расширений, как показано на рис. 3.2.

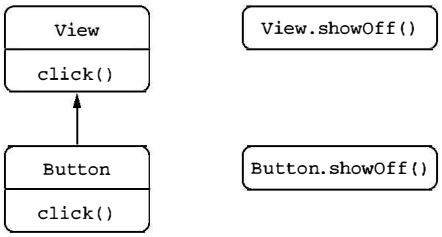


Рис. 3.2. Функции-расширения объявляются вне класса

Функции-расширения – не часть класса: они объявляются вне его. Несмотря на то что вы можете определить функции-расширения с одинаковыми именами и типами параметров для базового класса и его подклассов, вызываемая функция все равно будет зависеть от статического типа переменной, а не от типа значения этой переменной во время выполнения.

В следующем примере демонстрируются две функции-расширения `showOff`, объявленные для классов `View` и `Button`.

Листинг 3.6. Функции-расширения не переопределяются

```
fun View.showOff() = println("I'm a view!")
fun Button.showOff() = println("I'm a button!")

>>> val view: View = Button()
>>> view.showOff()
I'm a view!
```

Вызываемая функция-расширение определяется статически

Для вызова метода `showOff` переменной типа `View` компилятор выберет соответствующее расширение, руководствуясь типом, указанным в объявлении переменной, даже если фактическое значение имеет тип `Button`.

Если вспомнить, что функции-расширения компилируются в статические функции Java, принимающие объект-приемник в первом аргументе, такое поведение не покажется странным, поскольку Java определяет вызываемую функцию точно так же:

```
/* Java */
>>> View view = new Button();
>>> ExtensionsKt.showOff(view);
I'm a view!
```

Функции showOff объявлены в файле extensions.kt

Как вы можете видеть, переопределение невозможно для функций-расширений: Kotlin определяет их статически.

Примечание. Если класс имеет метод с той же сигнатурой, как у функции-расширения, приоритет всегда будет отдаваться методу. Имейте это в виду, расширяя API классов: если вы добавите в класс метод с такой же сигнатурой, как у функции-расширения, которую опреде-

лил клиент вашего класса, то после перекомпиляции код клиента изменит свою семантику и начнет ссылаться на новый метод класса.

Мы обсудили, как определять дополнительные методы для внешних классов. Теперь давайте посмотрим, как сделать то же самое со свойствами.

3.3.5. Свойства-расширения

Свойства-расширения позволяют добавлять в классы функции, к которым можно обращаться, используя синтаксис свойств, а не функций. Хотя они называются *свойствами*, свойства-расширения не могут иметь состояние из-за отсутствия места для его хранения: нельзя добавить дополнительных полей в существующие экземпляры объектов Java. Но более краткий синтаксис иногда бывает удобен.

В предыдущем разделе мы определили функцию `lastChar`. Теперь давайте преобразуем её в свойство.

Листинг 3.7. Объявление свойства-расширения

```
val String.lastChar: Char
    get() = get(length - 1)
```

Как и функции, свойства-расширения похожи на обычные свойства и отличаются только наличием имени типа-получателя в начале своего имени. Поскольку отдельного поля для хранения значения не существует, метод чтения должен определяться всегда и не может иметь реализации по умолчанию. Инициализаторы не разрешены по той же причине: негде хранить указанное значение.

Определяя то же самое свойство для класса `StringBuilder`, его можно объявить как `var`, потому что содержимое экземпляра `StringBuilder` может меняться.

Листинг 3.8. Объявление изменяемого свойства-расширения

```
var StringBuilder.lastChar: Char
  get() = get(length - 1)                  <— Метод чтения для свойства
  set(value: Char) {
    this.setCharAt(length - 1, value)      <— Метод записи для свойства
  }
```

К свойствам-расширениям можно обращаться как к обычным свойствам:

```
>>> println("Kotlin".lastChar)
n
```

```
>>> val sb = StringBuilder("Kotlin?")
>>> sb.lastChar = '!'
>>> println(sb)
Kotlin!
```

Обратите внимание: чтобы обратиться к свойству-расширению из Java, нужно явно вызывать его метод чтения: `StringUtilKt.getLastChar("Java")`.

Мы обсудили понятие расширений в целом. Теперь вернемся к теме коллекций и рассмотрим несколько библиотечных функций, помогающих в работе с ними, а также особенности языка, проявляющиеся в этих функциях.

3.4. Работа с коллекциями: переменное число аргументов, инфиксная форма записи вызова и поддержка в библиотеке

В этом разделе демонстрируются некоторые функции из стандартной библиотеки Kotlin для работы с коллекциями. Попутно мы опишем несколько связанных с ними особенностей языка:

- ключевое слово `vararg` позволяет объявить функцию, принимающую произвольное количество аргументов;
- *инфиксная* нотация поможет упростить вызовы функций с одним аргументом;
- *мультидекларации* (destructuring declarations) позволяют распаковать одно составное значение в несколько переменных.

3.4.1. Расширение API коллекций Java

Мы начали эту главу с рассуждения о том, что коллекции в Kotlin – те же классы, что и в Java, но с расширенным API. Вы видели примеры получения последнего элемента в списке и поиска максимального значения в коллекции чисел:

```
>>> val strings: List<String> = listOf("first", "second", "fourteenth")
>>> strings.last()
fourteenth

>>> val numbers: Collection<Int> = setOf(1, 14, 2)
>>> numbers.max()
14
```

Но нам интересно, как это работает: почему в Kotlin коллекции поддерживают так много всяких операций, хотя они – экземпляры библиотечных

классов Java. Теперь ответ должен быть ясен: функции `last` и `max` объявлены как функции-расширения!

Функция `last` ничуть не сложнее функции `lastChar` для класса `String`, которую мы обсудили в предыдущем разделе: она представляет собой расширение класса `List`. Для функции `max` мы покажем упрощенный вариант (настоящая библиотечная функция работает не только с числами типа `Int`, но и с любыми экземплярами типов, поддерживающих сравнение):

```
fun <T> List<T>.last(): T { /* возвращает последний элемент */ }
fun Collection<Int>.max(): Int { /* отыскивает максимальное значение в коллекции */ }
```

В стандартной библиотеке Kotlin объявлено много функций-расширений, но мы не будем перечислять их все. Возможно, вас больше интересует, как лучше исследовать стандартную библиотеку Kotlin. Вам не нужно учить её возможности наизусть: в любой момент, когда понадобится что-то сделать с коллекцией или любым другим объектом, механизм автоматического завершения кода в IDE покажет вам все функции, доступные для объекта данного типа. В списке будут как обычные методы, так и функции-расширения; вы сможете выбрать то, что вам нужно. Кроме того, в справочнике по стандартной библиотеке перечисляются все методы каждого библиотечного класса: не только члены класса, но и расширения.

В начале главы вы видели функции для создания коллекций. Общая черта этих функций – способность принимать произвольное число аргументов. В следующем разделе вы прочтете о синтаксисе объявления таких функций.

3.4.2. Функции, принимающие произвольное число аргументов

Вызывая функцию создания списка, вы можете передать ей любое количество аргументов:

```
val list = listOf(2, 3, 5, 7, 11)
```

В месте объявления этой библиотечной функции вы найдете следующее:

```
fun listOf<T>(vararg values: T): List<T> { ... }
```

Наверное, вы знакомы с возможностью передачи произвольного количества аргументов в Java путем упаковки их в массив. В Kotlin эта возможность реализована похожим образом, но синтаксис немного отличается: вместо трех точек после имени типа Kotlin использует модификатор `vararg` перед параметром.

Синтаксис вызова функций в Kotlin и Java также отличается способом передачи аргументов, уже упакованных в массив. В Java массив передается непосредственно, а Kotlin требует явно распаковать массив, чтобы каждый элемент стал отдельным аргументом вызываемой функции. Эта операция

называется вызовом с *оператором распаковки* (spread operator), а на практике это просто символ * перед соответствующим аргументом:

```
fun main(args: Array<String>) {
  val list = listOf("args: ", *args)
  println(list)
}
```

Оператор «звездочка» распаковывает содержимое массива

На этом примере видно, что оператор распаковки позволяет объединить в одном вызове значения из массива и несколько фиксированных значений. Этот способ не поддерживается в Java.

Теперь перейдем к словарям. Обсудим еще один способ улучшения читаемости вызовов функций в Kotlin: *инфиксный вызов* (infix call).

3.4.3. Работа с парами: инфиксные вызовы и мультидекларации

Для создания словаря применяется функция `mapOf`:

```
val map = mapOf(1 to "one", 7 to "seven", 53 to "fifty-three")
```

Сейчас самое время познакомиться с объяснением, которое мы обещали в начале главы. Ключевое слово `to` в этой строке – не встроенная конструкция, а специальная форма вызова метода, называемая *инфиксным вызовом*.

В инфиксном вызове имя метода помещается между именем целевого объекта и параметром, без дополнительных разделителей. Следующие два вызова эквивалентны:

```
1.to("one")
1 to "one"
```

Вызов функции to обычным способом

Вызов функции to с использованием инфиксной нотации

Инфиксную форму вызова можно применять к обычным методам и к функциям-расширениям, имеющим один обязательный параметр. Чтобы разрешить вызовы функции в инфиксной нотации, в её объявление нужно добавить модификатор `infix`. Ниже представлена упрощенная версия объявления функции `to`:

```
infix fun Any.to(other: Any) = Pair(this, other)
```

Функция `to` возвращает экземпляр класса `Pair` (из стандартной библиотеки Kotlin), который, как нетрудно догадаться, представляет пару из двух элементов. Настоящие объявления класса `Pair` и функции `to` используют обобщенные типы, но ради простоты изложения мы не будем вдаваться в эти подробности.

Обратите внимание, что значениями объекта `Pair` можно инициализировать сразу две переменные:

```
val (number, name) = 1 to "one"
```

Это называется *мультидекларацией* (destructuring declaration). На рис. 3.3 показано, как это работает с парой элементов.

Мультидекларации работают не только с парами. Например, содержимым элемента словаря можно инициализировать две переменные, `key` и `value`.

Этот прием также можно использовать в циклах – вы видели его использование для распаковки результатов вызова `withIndex` в реализации функции `joinToString`:

```
for ((index, element) in collection.withIndex()) {
    println("$index: $element")
}
```

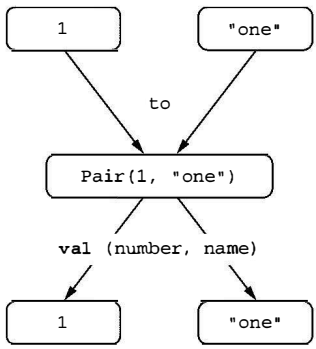


Рис. 3.3. Создание пары с помощью функции to и её распаковка с помощью мультидекларации

В разделе 7.4 будут описаны общие правила деструктуризации выражений и инициализации нескольких переменных.

Функция `to` является функцией-расширением. С её помощью можно создать пару любых элементов, т. е. эта функция расширяет обобщенный тип-приемник: можно написать `1 to "one"`, `"one" to 1`, `list to list.size()` и т. д. Давайте посмотрим на объявление функции `mapOf`:

```
fun <K, V> mapOf(vararg values: Pair<K, V>): Map<K, V>
```

Подобно `listOf`, функция `mapOf` принимает переменное количество аргументов, но на этот раз они должны быть парами ключей и значений.

Даже притом, что создание нового словаря может показаться специальной конструкцией в Kotlin, это самая обычная функция, поддерживающая краткий синтаксис. Далее мы покажем, как функции-расширения упрощают работу со строками и регулярными выражениями.

3.5. Работа со строками и регулярными выражениями

Строки в Kotlin – это те же объекты, что и в Java. Вы можете передать строку, созданную в коде на Kotlin, любому Java-методу и использовать любые методы из стандартной библиотеки Kotlin для обработки строк, полученных из Java-кода. При этом не происходит никаких преобразований и не создается никаких объектов-оберток.

Но Kotlin делает работу с обычными Java-строками более приятной, добавляя множество полезных функций-расширений. Кроме того, он скрывает некоторые методы, вызывающие путаницу, добавляя расширения с более однозначными именами. В качестве первого примера различий в API посмотрим, как в Kotlin выполняется разбиение строк.

3.5.1. Разбиение строк

Вы наверняка знакомы с методом `split` класса `String`. Он известен всем, но иногда на форуме Stack Overflow (http://stackoverflow.com) можно встретить жалобы на него: «Метод `split` в Java не работает с точкой». Многие заблуждаются, полагая, что вызов `"12.345-6.A".split(".")` вернет массив `[12, 345-6, A]`. Но в Java он вернет пустой массив! Это происходит потому, что `split` ожидает получить регулярное выражение и разбивает строку на несколько частей согласно этому выражению. В этом контексте точка `(.)` – регулярное выражение, соответствующее любому символу.

Kotlin скрывает этот метод, вызывающий путаницу, предоставляя сразу несколько перегруженных расширений с именем `split` и различными аргументами. Перегруженная версия, принимающая регулярное выражение, требует значения типа `Regex`, а не `String`. Это однозначно определяет, как будет интерпретироваться строка, переданная методу: как обычный текст или как регулярное выражение.

Вот как можно разделить строку с точкой или тире:

```
>>> println("12.345-6.A".split("\\.|-".toRegex()))
[12, 345, 6, A]
```

Явная передача регулярного выражения

Kotlin использует точно такой же синтаксис регулярных выражений, как Java. Здесь шаблон соответствует точке (мы экранировали её, чтобы показать, что имеем в виду саму точку, а не произвольный символ) или тире. Набор методов для работы с регулярными выражениями тоже аналогичен набору методов в стандартной библиотеке Java, но он более идиоматичен. Например, в Kotlin для преобразования строки в регулярное выражение используется функция-расширение `toRegex`.

Но для такого простого случая не нужно регулярное выражение. Другая перегруженная версия `split` в Kotlin принимает произвольное число разделителей в виде обычных строк:

```
>>> println("12.345-6.A".split(".", "-"))
[12, 345, 6, A]
```

Передача нескольких разделителей

Обратите внимание, что если передать аргументы символьного типа и написать `"12.345-6.A"1.split('.', '-')`, то получится аналогичный результат. Этот метод заменяет похожий Java-метод, принимающий только один аргумент с символом-разделителем.

3.5.2. Регулярные выражения и строки в тройных кавычках

Рассмотрим еще один пример с двумя различными реализациями: в первой используется расширение класса `String`, а во второй – регулярные выражения. Наша задача – разбить полный путь к файлу на компоненты: каталог, имя файла и расширение. Стандартная библиотека Kotlin содер-

жит функции для получения подстроки перед первым (или после последнего) появлением заданного разделителя. Вот как их можно использовать для решения этой задачи (см. также рис. 3.4).

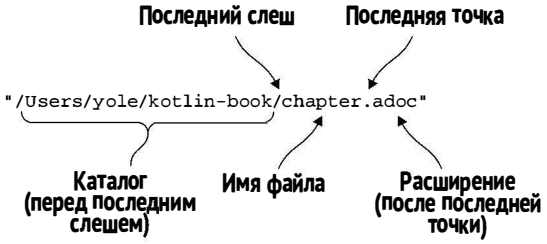


Рис. 3.4. Разбиение пути файла на компоненты: каталог, имя файла и расширение с помощью функций substringBeforeLast и substringAfterLast

Листинг 3.9. Использование расширений класса `String` для работы с путями к файлам

```
fun parsePath(path: String) {
  val directory = path.substringBeforeLast("/")
  val fullName = path.substringAfterLast("/")

  val fileName = fullName.substringBeforeLast(".")
  val extension = fullName.substringAfterLast(".")

  println("Dir: $directory, name: $fileName, ext: $extension")
}
>>> parsePath("/Users/yole/kotlin-book/chapter.adoc")
Dir: /Users/yole/kotlin-book, name: chapter, ext: adoc
```

Подстрока перед последней косой чертой в пути к файлу `path` – это каталог, где находится файл, подстрока после последней точки – это расширение файла, а его имя находится между ними.

Kotlin упрощает разбор строк без использования регулярных выражений (они мощный инструмент, но их бывает трудно понять после написания). Впрочем, стандартная библиотека Kotlin поможет и тем, кто желает использовать регулярные выражения. Вот как ту же задачу можно решить с помощью регулярных выражений:

Листинг 3.10. Использование регулярных выражений для разбора пути к файлу

```
fun parsePath(path: String) {
  val regex = """(.+)/(.+)\.(.+)""".toRegex()
  val matchResult = regex.matchEntire(path)
  if (matchResult != null) {
    val (directory, filename, extension) = matchResult.destructured
    println("Dir: $directory, name: $filename, ext: $extension")
```

```
  }
}
```

В этом примере регулярное выражение записано *в тройных кавычках*. В такой строке не нужно экранировать символов, включая обратную косую черту, поэтому литерал точки можно описать как `\.` вместо `\\.` в обычных строковых литералах (см. рис. 3.5).

Рис. 3.5. Регулярное выражение для разделения пути к файлу на каталог, имя файла и расширение

Это регулярное выражение разбивает путь на три группы, разделенные символом слеша и точкой. Шаблон `.` соответствует любому символу, начиная с начала строки, поэтому первой группе `(.+)` соответствует подстрока до последнего слеша. Эта подстрока включает все предыдущие слеши, потому что они соответствуют шаблону «любой символ». Соответственно, второй группе соответствует подстрока до последней точки, а третьей – оставшаяся часть.

Теперь обсудим реализацию функции `parsePath` из предыдущего примера. Она создает регулярное выражение и сопоставляет его с путем к файлу. Если совпадение найдено (результат не `null`), его свойство `destructured` присваивается соответствующим переменным. Это тот же синтаксис мультидеклараций, который был использован в случае инициализации двух переменных значениями свойств объекта `Pair`; подробнее об этом рассказывается в разделе 7.4.

3.5.3. Многострочные литералы в тройных кавычках

Тройные кавычки нужны не только для того, чтобы избавиться от необходимости экранировать символы. Такой строковый литерал может содержать любые символы, включая переносы строк. Это создает простой способ встраивания в программу текста, содержащего переносы строк. Для примера создадим ASCII-рисунок:

```
val kotlinLogo = """| //
                   .|//
                   .|// \"""
>>> println(kotlinLogo.trimMargin("."))
| //
|//
|// \
```

Многострочный литерал включает все символы между тройными кавычками, в том числе и отступы, использованные для форматирования кода. Для лучшего представления такой строки можно обрезать отступ (то есть левое поле). Для этого добавьте в строку префиксы, отмечающие конец отступа, а затем вызовите функцию `trimMargin`, которая удалит префикс и отступы в каждой строке. В этом примере в качестве такого префикса используются точки.

Строки в тройных кавычках могут содержать переносы строк, но в них нельзя использовать специальных символов, таких как `\n`. С другой стороны, отпадает необходимость экранировать символ `\`, поэтому путь в стиле Windows `"C:\\Users\\yole\\kotlin-book"` можно записать как `"""C:\Users\yole\kotlin-book"""`.

В многострочных литералах тоже можно использовать шаблоны. Поскольку многострочные литералы не поддерживают экранированных последовательностей, единственный способ включить в литерал знак доллара – это использовать встроенное выражение. Выглядит это так: `val price = """${'$'}99.9"""`.

Одна из областей, где многострочные литералы могут оказаться полезными (кроме игр, использующих ASCII-графику), – это тесты. В тестах часто приходится выполнять операцию, создающую многострочный литерал (например, фрагмент веб-страницы), и сравнивать его с ожидаемым результатом. Многострочные литералы позволяют хранить ожидаемые результаты в коде тестов. Больше нет необходимости заботиться об экранировании или загружать текст из внешних файлов – достаточно заключить ожидаемую разметку HTML или другой результат в тройные кавычки. А для лучшего форматирования используйте уже упоминавшуюся функцию `trimMargin` – еще один пример функции-расширения.

> **Примечание.** Теперь вы знаете, что функции-расширения – это мощное средство наращивания API существующих библиотек и их адаптации к идиомам нового языка – так называемый шаблон «Прокачай мою библиотеку»[1]. Действительно, большая часть стандартной библиотеки Kotlin состоит из функций-расширений для стандартных классов Java. Библиотека Anko (`https://github.com/kotlin/anko`), также разработанная компанией JetBrains, включает функции-расширения, которые делают Android API более дружелюбным по отношению к языку Kotlin. Кроме того, существует множество разработанных сообществом библиотек с удобными для работы с Kotlin обертками, таких как Spring.

Теперь, когда вы знаете, как Kotlin улучшает API используемых вам библиотек, давайте снова вернемся к нашему коду. Далее мы рассмотрим новые способы использования функций-расширений, а также обсудим новое понятие: *локальные функции* (local functions).

---

[1] Martin Odersky, «Pimp My Library», Artima Developer, October 9, 2006, http://mng.bz/86Qh.

3.6. Чистим код: локальные функции и расширения

Многие разработчики считают, что одно из самых важных качеств хорошего кода – отсутствие дублирования. Есть даже специальное название этого принципа: «не повторяйся» (Don't Repeat Yourself, DRY). В программах на Java следовать этому принципу не всегда просто. Во многих случаях можно разбить большой метод на меньшие фрагменты, а затем повторно их использовать (для рефакторинга можно использовать операцию **Extract Method** (Извлечь метод) в IDE). Но, это делает код более трудным для понимания, потому что в конечном итоге в классе окажется множество мелких методов без четкой взаимосвязи между ними. Можно пойти ещё дальше и сгруппировать извлеченные методы во внутреннем классе, сохранив структуру кода, но такой подход требует значительного объема шаблонного кода.

Kotlin предлагает более чистое решение: функции, извлеченные из основной функции, можно сделать вложенными. В результате получается требуемая структура, без лишних синтаксических накладных расходов.

Давайте посмотрим, как использовать локальные функции для решения такой распространенной проблемы, как дублирование кода. Функция в листинге 3.11 сохраняет информацию о пользователе в базе данных, и ей нужно убедится, что объект, представляющий пользователя, содержит допустимые данные.

Листинг 3.11. Функция с повторяющимся кодом

```
class User(val id: Int, val name: String, val address: String)

fun saveUser(user: User) {
  if (user.name.isEmpty()) {
    throw IllegalArgumentException(
      "Can't save user ${user.id}: empty Name")
  }

  if (user.address.isEmpty()) {
    throw IllegalArgumentException(
      "Can't save user ${user.id}: empty Address")
  }

  // Сохранение информации о пользователе в базе данных
}

>>> saveUser(User(1, "", ""))
java.lang.IllegalArgumentException: Can't save user 1: empty Name
```

Объем повторяющегося кода здесь невелик, и вы вряд ли захотите создавать отдельный метод в классе для обработки единственного особого случая проверки. Но, поместив код проверки в локальную функцию, можно избежать дублирования и сохранить четкую структуру кода (листинг 3.12).

Листинг 3.12. Извлечение локальной функции для предотвращения дублирования

```
class User(val id: Int, val name: String, val address: String)

fun saveUser(user: User) {
  fun validate(user: User,                    <-- Объявление локальной функции
               value: String,                     для проверки произвольного поля
               fieldName: String) {
    if (value.isEmpty()) {
      throw IllegalArgumentException(
        "Can't save user ${user.id}: empty $fieldName")
    }
  }

  validate(user, user.name, "Name")           <-- Вызов функции для проверки
  validate(user, user.address, "Address")         конкретных полей

  // Сохранение информации о пользователе в базе данных
}
```

Такой код выглядит лучше. Логика проверки не дублируется, и если по мере развития проекта понадобится добавить другие поля в класс `User`, то можно легко выполнять дополнительные проверки. Но передача объекта `User` в функцию проверки выглядит немного некрасиво. Самое интересное, что в этом нет никакой необходимости, потому что локальные функции имеют доступ ко всем параметрам и переменным охватывающей функции. Воспользуемся этим обстоятельством и избавимся от дополнительного параметра `User`.

Листинг 3.13. Доступ к параметрам внешней функции из локальной функции

```
class User(val id: Int, val name: String, val address: String)

fun saveUser(user: User) {
  fun validate(value: String, fieldName: String) {   <-- Теперь не нужно дублировать
    if (value.isEmpty()) {                                параметра user в функции saveUser
      throw IllegalArgumentException(
        "Can't save user ${user.id}: " +              <-- Можно напрямую обращаться к
        "empty $fieldName")                               параметрам внешней функции
    }
```

```
  }

  validate(user.name, "Name")
  validate(user.address, "Address")

  // Сохранение информации о пользователе в базе данных
}
```

Можно ещё улучшить этот пример, если перенести логику проверки в функцию-расширение класса `User`.

Листинг 3.14. Перемещение логики в функцию-расширение

```
class User(val id: Int, val name: String, val address: String)

fun User.validateBeforeSave() {
  fun validate(value: String, fieldName: String) {
    if (value.isEmpty()) {
      throw IllegalArgumentException(
        "Can't save user $id: empty $fieldName")
    }
  }
  validate(name, "Name")
  validate(address, "Address")
}

fun saveUser(user: User) {
  user.validateBeforeSave()

  // Сохранение пользователя в базу данных
}
```

К свойствам класса User можно обращаться напрямую

Вызов функции-расширения

Извлечение фрагмента кода в функцию-расширение оказывается на удивление полезным приемом рефакторинга. Даже если класс `User` – часть кода вашего проекта, а не библиотечный класс, вы можете посчитать излишним оформлять эту логику в виде метода класса, потому что она нигде больше не используется. Если следовать этому подходу, прикладной интерфейс класса будет содержать только необходимые методы, используемые повсеместно, не разрастется и останется легкодоступным для понимания. С другой стороны, функции, которые обращаются к одному объекту и которым не нужен доступ к приватным данным, могут напрямую обращаться к нужным свойствам, как показано в листинге 3.14.

Функции-расширения также могут объявляться как локальные функции, поэтому можно пойти дальше и сделать функцию `User.validateBeforeSave` локальной функцией в `saveUser`. Но глубоко вложенные локальные функ-

ции трудно читать; поэтому в общем случае мы рекомендуем не использовать более одного уровня вложенности.

Узнав, что можно делать с функциями, в следующей главе мы рассмотрим действия с классами.

3.7. Резюме

- В языке Kotlin нет собственных классов коллекций; вместо этого он добавляет новые методы в классы коллекций Java, предоставляя богатый API.
- Определение значений параметров по умолчанию снижает необходимость перегрузки функций, а синтаксис именованных аргументов делает вызов функций с несколькими аргументами более читабельным.
- Функции и свойства можно объявлять не только как члены класса, но и непосредственно на верхнем уровне файла, что позволяет определять код с более гибкой структурой.
- Функции-расширения и свойства-расширения дают возможность расширять API любых классов, в том числе классов во внешних библиотеках, без модификации их исходного кода и без дополнительных накладных расходов во время выполнения.
- Инфиксная нотация обеспечивает более чистый синтаксис вызова методов с одним аргументом, похожий на синтаксис операторов.
- Kotlin поддерживает большое количество функций для работы со строками и с регулярными выражениями.
- Строки в тройных кавычках упрощают запись выражений, которые в Java потребовали бы использования неуклюжих символов экранирования и множества операций конкатенации.
- Локальные функции помогают лучше структурировать код и избавиться от дублирования.

Глава 4

Классы, объекты и интерфейсы

В этой главе:

- классы и интерфейсы;
- нетривиальные свойства и конструкторы;
- классы данных;
- делегирование;
- ключевое слово `object`.

Эта глава позволит вам лучше понять особенности работы с классами в Kotlin. Во второй главе вы познакомились с базовым синтаксисом объявления класса, узнали, как объявлять методы и свойства, как использовать основные конструкторы (разве они не чудо?) и как работать с перечислениями. Но есть кое-что, чего вы ещё не видели.

Классы и интерфейсы в Kotlin немного отличаются от своих аналогов в Java: например, интерфейсы могут содержать объявления свойств. В отличие от Java, объявления в Kotlin по умолчанию получают модификаторы `final` и `public`. Кроме того, вложенные классы по умолчанию не становятся внутренними: они не содержат неявной ссылки на внешний класс.

Что касается конструкторов, то лаконичный синтаксис основного конструктора (primary constructor) отлично подходит для большинства случаев; но существует и более полный синтаксис, позволяющий объявлять конструкторы с нетривиальной логикой инициализации. Это же относится к свойствам: краткий синтаксис хорош, но есть и возможность определять собственные реализации методов доступа.

Компилятор Kotlin может сам генерировать полезные методы, позволяя избавляться от лишнего кода. Если класс объявляется как *класс данных* (data class), компилятор добавит в него несколько стандартных методов. Также нет необходимости делегировать методы вручную, потому что шаблон делегирования поддерживается в Kotlin на уровне языка.

Эта глава также описывает новое ключевое слово `object`, которое объявляет класс и одновременно создает его экземпляр. Это ключевое слово используется для объявления объектов-одиночек (singleton), объектов-компаньонов (companion objects) и объектов-выражений (аналог анонимных классов в Java). Начнем с разговора о классах и интерфейсах и тонкостях определения иерархий классов в Kotlin.

4.1. Создание иерархий классов

В этом разделе сравниваются способы создания иерархий классов в Kotlin и Java. Вы познакомитесь с понятием видимости и модификаторами доступа в Kotlin, которые тоже есть в Java, но отличаются некоторыми умолчаниями. Вы также узнаете о новом модификаторе `sealed`, который ограничивает возможность создания подклассов.

4.1.1. Интерфейсы в Kotlin

Начнем с определения и реализации интерфейсов. Интерфейсы в Kotlin напоминают Java 8: они могут содержать определения абстрактных и реализации конкретных методов (подобно методам по умолчанию в Java 8), но они не могут иметь состояний.

Интерфейсы объявляются в Kotlin с помощью ключевого слова `interface`.

Листинг 4.1. Объявление простого интерфейса

```
interface Clickable {
    fun click()
}
```

Это объявление интерфейса с единственным абстрактным методом `click`. Все конкретные классы, реализующие этот интерфейс, должны реализовать данный метод, например:

Листинг 4.2. Реализация простого интерфейса

```
class Button : Clickable {
    override fun click() = println("I was clicked")
}

>>> Button().click()
I was clicked
```

Вместо ключевых слов `extends` и `implements`, используемых в Java, в языке Kotlin используется двоеточие после имени класса. Как в Java, класс может реализовать столько интерфейсов, сколько потребуется, но наследовать только один класс.

Модификатор `override`, похожий на аннотацию `@Override` в Java, используется для определения методов и свойств, которые переопределяют соответствующие методы и свойства суперкласса или интерфейса. В отличие от Java, в языке Kotlin *применение модификатора* `override` *обязательно*. Это избавит вас от случайного переопределения метода, если он будет добавлен в базовый класс после написания реализации: ваш код не будет компилироваться, пока вы явно не добавите модификатор `override` в объявление метода или не переименуете его.

Метод интерфейса может иметь реализацию по умолчанию. В отличие от Java 8, где эти реализации должны объявляться с ключевым словом `default`, в языке Kotlin нет специальной аннотации для таких методов: достаточно просто написать тело метода. Давайте изменим интерфейс `Clickable`, добавив метод с реализацией по умолчанию.

Листинг 4.3. Определение метода с телом в интерфейсе

```
interface Clickable {
    fun click()
    fun showOff() = println("I'm clickable!")
}
```

Обычное объявление метода

Метод с реализацией по умолчанию

При реализации этого интерфейса вам придется определить только реализацию метода `click`. Вы можете переопределить поведение метода `showOff` – или оставить поведение по умолчанию.

Теперь предположим, что другой интерфейс также определяет метод `showOff` со следующей реализацией.

Листинг 4.4. Другой интерфейс, реализующий тот же метод

```
interface Focusable {
    fun setFocus(b: Boolean) =
      println("I ${if (b) "got" else "lost"} focus.")

    fun showOff() = println("I'm focusable!")
}
```

Что случится, если потребуется реализовать оба интерфейса в одном классе? Оба включают метод `showOff` с реализацией по умолчанию; какая реализация будет выбрана? Ни одна из них. Если вы явно не реализуете своего метода `showOff`, компилятор сообщит об ошибке:

```
The class 'Button' must
override public open fun showOff() because it inherits
many implementations of it.
```

Компилятор Kotlin вынуждает вас предоставить собственную реализацию.

Листинг 4.5. Вызов метода интерфейса с реализацией по умолчанию

```
class Button : Clickable, Focusable {
    override fun click() = println("I was clicked")

    override fun showOff() {
      super<Clickable>.showOff()
      super<Focusable>.showOff()
    }
}
```

Вы должны явно реализовать метод, если наследуется несколько его реализаций

Ключевое слово «super» с именем супертипа в угловых скобках определяет родителя, чей метод будет вызван

Теперь класс `Button` реализует два интерфейса. В реализации метода `showOff` вызываются обе реализации, унаследованные от супертипов. Для вызова унаследованной реализации используется то же ключевое слово, что и в Java: `super`. Но синтаксис выбора конкретной реализации отличается. Если в Java имя базового типа указывается перед ключевым словом, как в выражении `Clickable.super.showOff()`, то в Kotlin имя базового типа должно указываться в угловых скобках: `super<Clickable>.showOff()`.

Если требуется вызвать лишь одну унаследованную реализацию, это можно оформить так:

```
override fun showOff() = super<Clickable>.showOff()
```

Попробуйте создать экземпляр этого класса и убедиться, что все наследуемые методы доступны для вызова.

```
fun main(args: Array<String>) {
    val button = Button()
    button.showOff()
    button.setFocus(true)
    button.click()
}
```

I'm clickable!
I'm focusable!

I got focus.

I was clicked.

Реализация метода `setFocus` объявлена в интерфейсе `Focusable` и автоматически наследуется классом `Button`.

Реализация интерфейсов с телами методов в Java

Версия Kotlin 1.0 разрабатывалась с прицелом на совместимость с Java 6, которая не поддерживает реализацию методов по умолчанию в интерфейсах. Поэтому каждый интерфейс с методами по умолчанию компилируется как сочетание обычного интерфейса и класса с реализацией в виде статического метода. Интерфейс содержит только объявления, а класс – все реализациями методов. Поэтому, если понадобится реализовать такой интерфейс в Java-классе, вам придется добавить собственные реализации всех методов, в том числе тех, для которых в Kotlin определено тело метода.

Теперь, когда вы узнали, как Kotlin позволяет реализовать методы в интерфейсах, посмотрим на вторую часть этой истории: переопределение членов базовых классов.

4.1.2. Модификаторы open, final и abstract: по умолчанию final

Как вы знаете, Java позволяет создавать подклассы любого класса и переопределять любые методы, если они не отмечены ключевым словом `final`. Часто это бывает удобно, но иногда может вызывать проблемы.

Проблема с так называемыми *хрупкими базовыми классами* (fragile base class) возникает при модификации базового класса, которая может вызвать некорректное поведение подклассов (потому что изменившийся код базового класса больше не соответствует ожиданиям подклассов). Если для класса не указаны точные правила наследования – какие методы должны переопределяться и как, – клиенты рискуют переопределить методы таким способом, какого автора базовый класс не предусматривал. Поскольку невозможно проанализировать все подклассы, базовый класс считается «хрупким» в том смысле, что любое изменение в нем может привести к неожиданным изменениям поведения в подклассах.

Для решения этой проблемы Joshua Bloch (Джошуа Блох) в своей книге «Effective Java» (Addison-Wesley, 2008)[1], одной из самых известных книг о стиле программирования на Java, рекомендует «проектировать и документировать наследование или запрещать его». Это означает, что все классы и методы, которые не предназначены для переопределения в подклассах, должны снабжаться модификатором `final`.

Kotlin тоже придерживается этой философии. В Java все классы и методы открыты по умолчанию, а в Kotlin они по умолчанию отмечены модификатором `final`.

Если вы хотите разрешить наследовать свой класс, объявите его открытым, добавив модификатор `open`. Вам также понадобится добавить модификатор `open` ко всем свойствам и методам, которые могут быть переопределены.

Листинг 4.6. Объявление открытого класса с открытым методом

```
open class RichButton : Clickable {      <— Это открытый класс: другие могут наследовать его

    fun disable() {}                     <— Это закрытая функция: ее невозможно переопределить в подклассе

    open fun animate() {}                <— Это открытая функция: ее можно переопределить в подклассе

    override fun click() {}              <— Переопределение открытой функции также является открытым
}
```

[1] Блох Д., Java. Эффективное программирование, ISBN: 978-5-85582-347-9, Лори (2013). – *Прим. ред.*

Обратите внимание, что если переопределить метод базового класса или интерфейса, эта версия метода также будет открытой. Чтобы запретить переопределение вашей реализации в подклассах, добавьте в объявление переопределяющей версии модификатор `final`.

Листинг 4.7. Запрет переопределения

```
open class RichButton : Clickable {
    final override fun click() {}
}
```

Ключевое слово «final» здесь не лишнее, потому что модификатор «override» без «final» означает, что метод останется открытым

Открытые классы и умное приведение типов

Одно из основных преимуществ классов, закрытых по умолчанию: они позволяют выполнять автоматическое приведение типов в большем количестве сценариев. Как уже упоминалось в разделе 2.3.5, автоматическое приведение возможно только для переменных, которые нельзя изменить после проверки типа. Для класса это означает, что автоматическое приведение типа может применяться только к неизменяемым свойствам с модификатором `val` со стандартным методом доступа. Из этого следует, что свойство должно быть `final` — иначе подкласс сможет переопределить свойство и определить собственный метод доступа, нарушая основное требование автоматического приведения типов. Поскольку по умолчанию свойства получают модификатор `final`, автоматическое приведение работает с большинством свойств в вашем коде, что повышает его выразительность.

Как в Java, в Kotlin класс можно объявить абстрактным, добавив ключевое слово `abstract`, и создать экземпляр такого класса будет невозможно. Абстрактный класс обычно содержит абстрактные методы без реализации, которые должны быть переопределены в подклассах. Абстрактные методы всегда открыты, поэтому не требуется явно использовать модификатор `open`. Вот пример такого класса.

Листинг 4.8. Объявление абстрактного класса

```
abstract class Animated {

    abstract fun animate()

    open fun stopAnimating() {
    }

    fun animateTwice() {
    }
}
```

Это абстрактный класс: нельзя создать его экземпляр

Это абстрактная функция: она не имеет реализации и должна быть переопределена в подклассах

Конкретные функции в абстрактных классах по умолчанию закрыты, но их можно сделать открытыми

В табл. 4.1 перечислены модификаторы доступа, поддерживаемые в языке Kotlin. Комментарии в таблице также относятся к модификаторам классов; в интерфейсах ключевые слова `final`, `open` и `abstract` не используются. Все методы в интерфейсе по умолчанию снабжены модификатором `open`; вы не сможете объявить их закрытыми (`final`). В отсутствие реализации метод будет считаться абстрактным, а ключевое слово `abstract` можно опустить.

Таблица 4.1. Значение модификаторов доступа в классе

Модификатор	Соответствующий член	Комментарии
`final`	Не может быть переопределен	Применяется к членам класса по умолчанию
`open`	Может быть переопределен	Должен указываться явно
`abstract`	Должен быть переопределен	Используется только в абстрактных классах; абстрактные методы не могут иметь реализацию
`override`	Переопределяет метод суперкласса или интерфейса	По умолчанию переопределяющий метод открыт, если только не объявлен как `final`

Обсудив модификаторы, управляющие наследованием, перейдем к другому типу модификаторов: модификаторам видимости.

4.1.3. Модификаторы видимости: по умолчанию public

Модификаторы видимости помогают контролировать доступность объявлений в коде. Ограничивая видимость деталей реализации класса, можно гарантировать возможность их изменения без риска поломки кода, зависящего от класса.

По сути, модификаторы видимости в Kotlin похожи на аналогичные модификаторы в Java. Здесь используются те же ключевые слова: `public`, `protected` и `private`. Отличается лишь видимость по умолчанию: отсутствие модификатора в объявлении предполагает модификатор `public`.

Область видимости в Java по умолчанию ограничивается рамками пакета. В Kotlin такой модификатор видимости отсутствует: пакеты используются только для организации кода в пространства имен, но не для управления видимостью.

В качестве альтернативы Kotlin предлагает новый модификатор видимости `internal`, обозначающий «видимость в границах модуля». *Модуль* – это набор файлов, компилируемых вместе. Это может быть модуль IntelliJ IDEA, проект Eclipse, Maven или Gradle, или набор файлов, компилируемых заданием Ant.

Преимущество видимости `internal` в том, что она обеспечивает настоящую инкапсуляцию деталей реализации модуля. В Java инкапсуляцию легко нарушить: автору стороннего кода достаточно определить классы в тех

же пакетах, что и в вашем коде, – и он получит доступ к вашим объявлениям из области видимости пакета.

Ещё одно отличие: Kotlin позволяет применять модификатор `private` к объявлениям верхнего уровня, в том числе к классам, функциям и свойствам. Такие объявления видны только в файле, где они определены. Это ещё один полезный способ сокрытия деталей реализации подсистемы. В табл. 4.2 перечислены все модификаторы видимости.

Таблица 4.2. Модификаторы видимости в Kotlin

Модификатор	Член класса	Объявление верхнего уровня
`public` (по умолчанию)	Доступен повсюду	Доступно повсюду
`internal`	Доступен только в модуле	Доступно в модуле
`protected`	Доступен в подклассах	–
`private`	Доступен в классе	Доступно в файле

Рассмотрим пример. Каждая строка в функции `giveSpeech` при выполнении нарушала бы правила видимости. Она компилируется с ошибкой.

```
internal open class TalkativeButton : Focusable {
    private fun yell() = println("Hey!")
    protected fun whisper() = println("Let's talk!")
}
fun TalkativeButton.giveSpeech() {
    yell()
    whisper()
}
```

Ошибка: «публичный» член класса раскрывает «внутренний» тип-приемник «TalkativeButton»

Ошибка: функция «yell» недоступна; в классе «TalkativeButton» она объявлена с модификатором «private»

Ошибка: функция «whisper» недоступна; в классе «TalkativeButton» она объявлена с модификатором «protected»

Kotlin запрещает ссылаться из публичной функции `giveSpeech` на тип `TalkativeButton` с более узкой областью видимости (в данном случае `internal`). Это лишь частный случай общего правила, которое требует, чтобы все классы в списке базовых и параметризованных типов класса или в сигнатурах методов имели такую же или более широкую область видимости, как сам класс или метод. Это правило гарантирует, что у вас всегда будет доступ ко всем типам, нужным для вызова функции или наследования класса.

Чтобы исправить проблему в примере выше, можно объявить функцию как `internal` или сделать класс публичным, убрав модификатор `internal`.

Обратите внимание на разницу в поведении модификатора `protected` в Java и в Kotlin. В Java член класса с модификатором `protected` доступен во всем пакете – но Kotlin такого не позволяет. В Kotlin правила видимости проще: член класса с модификатором `protected` доступен только в самом классе и его подклассах. Также заметьте, что функции-расширения

класса не могут обращаться к его членам с модификаторами `protected` или `private`.

> **Модификаторы видимости Kotlin и Java**
>
> Когда код на Kotlin компилируется в байт-код Java, модификаторы `public`, `protected` и `private` сохраняются. Вы можете использовать такие объявления Kotlin в коде на Java, как если бы они изначально были объявлены с такой же областью видимости в Java. Единственное исключение – класс с модификатором `private`: он будет скомпилирован с областью видимости пакета (в Java нельзя сделать класс приватным).
>
> Но что произойдет с модификатором `internal`? В Java отсутствует прямой аналог. Область видимости пакета – совершенно иное дело: обычно модуль состоит из нескольких пакетов, и разные модули могут содержать объявления из одного пакета. Поэтому модификатор `internal` в байт-коде превратится в `public`.
>
> Такое соответствие между объявлениями в Kotlin и их аналогами в Java (точнее, их представлением на уровне байт-кода) объясняет, почему из Java-кода иногда можно вызвать что-то, что нельзя вызвать из Kotlin. Например, в Java-коде можно получить доступ к классу с модификатором `internal` или объявлению верхнего уровня из другого модуля, или получить доступ к защищенному (`protected`) члену класса в том же пакете (как разрешают правила доступа Java).
>
> Но имейте в виду, что компилятор добавляет к именам членов класса с модификатором `internal` специальные суффиксы. Технически такие элементы можно использовать в коде Java, но это будет выглядеть некрасиво. В Kotlin это помогает избежать неожиданных конфликтов при переопределении, когда наследуемый класс находится в другом модуле, а также предотвращает случайное использование внутренних классов.

Еще одно отличие в правилах видимости между Kotlin и Java: в Kotlin внешний класс не видит приватных членов внутренних (вложенных) классов. Давайте поговорим о внутренних и вложенных классах Kotlin и рассмотрим пример в следующей главе.

4.1.4. Внутренние и вложенные классы: по умолчанию вложенные

И в Java, и в Kotlin можно объявить один класс внутри другого класса. Это полезно для сокрытия вспомогательного класса или размещения кода ближе к месту его использования. Разница в том, что в Kotlin вложенные классы не имеют доступа к экземпляру внешнего класса, если не запросить его явно. Давайте посмотрим на примере, почему это важно.

Представьте, что нам нужно определить видимый элемент (`View`), состояние которого может быть сериализовано. Сериализовать такой элемент часто очень сложно, но можно скопировать все необходимые дан-

ные в другой вспомогательный класс. Для этого мы объявим интерфейс State, наследующий Serializable. В интерфейсе View имеются методы getCurrentState и restoreState для сохранения состояния элемента.

Листинг 4.9. Объявление видимого элемента с сериализуемым состоянием

```
interface State: Serializable

interface View {
    fun getCurrentState(): State
    fun restoreState(state: State) {}
}
```

Было бы удобно определить класс, который будет хранить состояние кнопки в классе Button. Давайте посмотрим, как это можно сделать в Java (а аналогичный код Kotlin покажем чуть позже).

Листинг 4.10. Реализация интерфейса View в Java с помощью внутреннего класса

```
/* Java */
public class Button implements View {
    @Override
    public State getCurrentState() {
      return new ButtonState();
    }

    @Override
    public void restoreState(State state) { /*...*/ }

    public class ButtonState implements State { /*...*/ }
}
```

Мы определили класс ButtonState, который реализует интерфейс State и хранит состояние Button. Теперь в методе getCurrentState создаем новый экземпляр этого класса. В реальном коде мы бы инициализировали ButtonState со всеми необходимыми данными.

Что не так с этим кодом? Почему при попытке сериализовать состояние кнопки возникает исключение java.io.NotSerializableException: Button? На первый взгляд это может показаться странным: сериализуемая переменная имеет тип ButtonState, а не Button.

Все станет на свои места, если вспомнить, что когда в Java один класс объявляется внутри другого, он по умолчанию становится внутренним классом. В данном случае класс ButtonState будет неявно хранить ссылку на внешний класс Button. Это объясняет, почему ButtonState не может

быть сериализован: класс Button не сериализуется, а ссылка на него препятствует сериализации ButtonState.

Чтобы устранить эту проблему, нужно сделать класс ButtonState статическим, добавив модификатор static. Объявление вложенного класса статическим удаляет неявную ссылку на внешний класс.

В Kotlin поведение внутренних классов по умолчанию противоположно, как показано далее.

Листинг 4.11. Реализация интерфейса View в Kotlin с помощью вложенного класса

```
class Button : View {
    override fun getCurrentState(): State = ButtonState()

    override fun restoreState(state: State) { /*...*/ }

    class ButtonState : State { /*...*/ }
}
```

Это аналог статического вложенного класса в Java

В Kotlin вложенный класс без модификаторов — это полный аналог статического вложенного класса в Java. Чтобы превратить его во внутренний класс со ссылкой на внешний класс, нужно добавить модификатор inner. В табл. 4.3 описаны различия между Java и Kotlin, а разница между вложенными и внутренними классами проиллюстрирована на рис. 4.1.

Таблица 4.3. Соответствие между внутренними и вложенными классами в Java и Kotlin

Класс A, объявленный внутри другого класса B	В Java	В Kotlin
Вложенный класс (не содержит ссылки на внешний класс)	`static class A`	`class A`
Внутренний класс (содержит ссылку на внешний класс)	`class A`	`inner class A`

Рис. 4.1. Во вложенных классах, в отличие от внутренних, отсутствует ссылка на внешний класс

Синтаксис обращения к внешнему классу из внутреннего в Kotlin также отличается от Java. Чтобы получить доступ к классу Outer из класса Inner, нужно написать this@Outer.

```
class Outer {
    inner class Inner {
      fun getOuterReference(): Outer = this@Outer
    }
}
```

Теперь вы знаете разницу между внутренними и вложенными классами в Java и Kotlin. Давайте рассмотрим еще один случай, когда вложенные классы могут пригодиться в Kotlin: создание иерархии с ограниченным числом классов.

4.1.5. Запечатанные классы: определение жестко заданных иерархий

Вспомните пример иерархии классов для представления выражений из раздела 2.3.5. Суперкласс `Expr` имеет два подкласса: `Num`, представляющий число, и `Sum`, представляющий сумму двух выражений. Все возможные подклассы удобно обрабатывать с помощью выражения `when`. Но при этом необходимо предусмотреть ветку `else`, чтобы определить развитие событий, если не будет выбрана ни одна из других веток:

Листинг 4.12. Реализация выражения как интерфейса

```
interface Expr
class Num(val value: Int) : Expr
class Sum(val left: Expr, val right: Expr) : Expr

fun eval(e: Expr): Int =
    when (e) {
      is Num -> e.value
      is Sum -> eval(e.right) + eval(e.left)
      else ->
        throw IllegalArgumentException("Unknown expression")
    }
```

Необходимо также проверять ветку «else»

При вычислении выражения с использованием конструкции `when` компилятор Kotlin вынуждает добавить ветку, выполняемую по умолчанию. В этом примере невозможно вернуть что-либо осмысленное, поэтому генерируется исключение.

Необходимость добавления ветки, выполняемой по умолчанию, может вызывать неудобства. Более того, если позднее добавить новый подкласс, компилятор не поймет, что что-то изменилось. Если забыть добавить новую ветку, будет выбрана ветка по умолчанию, что может привести к трудно диагностируемым ошибкам.

Kotlin решает эту проблему с помощью запечатанных (`sealed`) классов. Достаточно добавить модификатор `sealed` в объявление суперкласса, и он ограничит возможность создания подклассов. Все прямые подклассы должны быть вложены в суперкласс:

Листинг 4.13. Выражения в виде запечатанных классов

```
sealed class Expr {                          <— Представление выражений запечатанными классами...
  class Num(val value: Int) : Expr()
  class Sum(val left: Expr, val right: Expr) : Expr()   <— ...и перечислить все возможные подклассы в виде вложенных классов.
}

fun eval(e: Expr): Int =
    when (e) {
      is Expr.Num -> e.value                 <— Выражение «when» охватывает все возможные варианты, поэтому ветка «else» не нужна.
      is Expr.Sum -> eval(e.right) + eval(e.left)
    }
```

При обработке всех подклассов запечатанного класса в выражении when нет необходимости в ветке по умолчанию. Обратите внимание: модификатор sealed означает, что класс по умолчанию открыт, добавлять модификатор open не требуется. Поведение запечатанных классов показано на рис. 4.2.

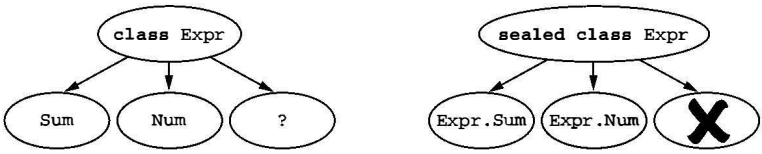


Рис. 4.2. Запечатанный класс не может иметь наследников, объявленных вне класса

Когда выражение when используется с запечатанными классами, при добавлении нового подкласса выражение when, возвращающее значение, не скомпилируется, а сообщение об ошибке укажет, какой код нужно изменить.

Внутренне класс Expr получит приватный конструктор, который можно вызвать только внутри класса. Вы не сможете объявить запечатанный интерфейс. Почему? Если бы такая возможность была, компилятор Kotlin не смог бы гарантировать, что кто-то не реализует этого интерфейса в коде на Java.

> **Примечание.** В Kotlin 1.0 модификатор sealed накладывает серьезные ограничения. Например, все подклассы должны быть вложенными, и подкласс не может быть классом данных (классы данных описаны далее в этой главе). Kotlin 1.1 ослабляет эти ограничения и позволяет определять подклассы запечатанных классов в любом месте в том же файле.

Как вы помните, в Kotlin двоеточие используется для перечисления наследуемого класса и реализуемых интерфейсов. Давайте внимательнее посмотрим на объявление подкласса:

```
class Num(val value: Int) : Expr()
```

Этот простой пример должен быть понятен во всём, кроме назначения скобок после имени класса `Expr`. Мы поговорим о них в следующем разделе, который охватывает инициализацию классов в Kotlin.

4.2. Объявление классов с нетривиальными конструкторами или свойствами

Как известно, в Java-классе можно объявить один или несколько конструкторов. Kotlin поддерживает всё то же самое, кроме одного: он различает *основной* конструктор (который, как правило, является главным лаконичным средством инициализации класса и объявляется вне тела класса) и *вторичный* конструктор (который объявляется в теле класса). Kotlin также позволяет вставлять дополнительную логику инициализации в соответствующие блоки. Сначала продемонстрируем синтаксис объявления главного конструктора и блоков инициализации, а затем объясним, как объявить несколько конструкторов. После этого поговорим о свойствах.

4.2.1. Инициализация классов: основной конструктор и блоки инициализации

В главе 2 вы узнали, как объявить простой класс:

```
class User(val nickname: String)
```

Как правило, все объявления, относящиеся к классу, находятся внутри фигурных скобок. Вы можете спросить, почему объявление самого класса не имеет фигурных скобок, а единственное объявление заключено в круглые скобки? Этот блок кода, окруженный круглыми скобками, называется *основным конструктором* (primary constructor). Он преследует две цели: определение параметров конструктора и определение свойств, которые инициализируются этими параметрами. Давайте разберемся, что здесь происходит, и посмотрим, как выглядит явный код, который делает то же самое:

```
class User constructor(_nickname: String) {
  val nickname: String

  init {
    nickname = _nickname
  }
}
```

В этом примере используются два новых ключевых слова Kotlin: `constructor` и `init`. С ключевого слова `constructor` начинается объявление основного или вторичного конструктора. Ключевое слово `init` обозначает

начало *блока инициализации*. Такие блоки содержат код инициализации, который выполняется при создании каждого экземпляра класса, и предназначены для использования вместе с первичными конструкторами. Блоки инициализации приходится использовать, поскольку синтаксис первичного конструктора ограничен и он не может содержать кода инициализации. При желании можно объявить несколько блоков инициализации в одном классе.

Подчеркивание в параметре конструктора `_nickname` поможет отличить имя свойства от имени параметра конструктора. Можно использовать одно имя, но в этом случае, чтобы избежать двусмысленности, присваивание надо будет оформить так, как это делается в Java: `this.nickname = nickname`.

В этом примере инициализирующий код можно совместить с объявлением свойства `nickname`, поэтому его не нужно помещать в блок инициализации. В отсутствие аннотаций и модификаторов видимости основного конструктора ключевое слово `constructor` также можно опустить. После применения этих изменений получается следующее:

```
class User(_nickname: String) {
    val nickname = _nickname
}
```

Основной конструктор с одним параметром

Свойство инициализируется значением параметра

Это ещё один способ объявления того же класса. Обратите внимание, что к параметрам основного конструктора ещё можно обращаться при установке значений свойств в блоках инициализации.

В двух предыдущих примерах свойство объявлялось с ключевым словом `val` в теле класса. Если свойство инициализируется соответствующим параметром конструктора, код можно упростить, поставив ключевое слово `val` перед параметром. Такое определение параметра заменит объявление свойства в теле класса:

```
class User(val nickname: String)
```

«val» означает, что для параметра должно быть создано соответствующее свойство

Все вышеперечисленные объявления класса `User` эквивалентны, но последнее имеет самый лаконичный синтаксис.

Параметрам конструктора и параметрам функций можно назначать значения по умолчанию:

```
class User(val nickname: String,
           val isSubscribed: Boolean = true)
```

Значение по умолчанию для параметра конструктора

Чтобы создать экземпляр класса, нужно вызвать конструктор напрямую, без ключевого слова `new`.

```
>>> val alice = User("Alice")
>>> println(alice.isSubscribed)
```

Использует значение параметра isSubscribed по умолчанию, равное «true»

```
true
>>> val bob = User("Bob", false)
>>> println(bob.isSubscribed)
false
>>> val carol = User("Carol", isSubscribed = false)
>>> println(carol.isSubscribed)
false
```

Значения параметров можно передавать в порядке определения

Можно явно указывать имена некоторых аргументов конструктора

Похоже, что Алиса подписалась на рассылку автоматически, в то время как Боб внимательно прочел условия и поменял значение параметра по умолчанию.

> **Примечание.** Если все параметры конструктора будут иметь значения по умолчанию, компилятор сгенерирует дополнительный конструктор без параметров, использующий все значения по умолчанию. Это упрощает использование в Kotlin библиотек, которые создают экземпляры классов с помощью конструкторов без параметров.

Если класс имеет суперкласс, основной конструктор также должен инициализировать свойства, унаследованные от суперкласса. Сделать это можно, перечислив параметры конструктора суперкласса после имени его типа в списке базовых классов:

```
open class User(val nickname: String) { ... }

class TwitterUser(nickname: String) : User(nickname) { ... }
```

Если вообще не объявить никакого конструктора, компилятор добавит конструктор по умолчанию, который ничего не делает:

```
open class Button
```

Будет сгенерирован конструктор по умолчанию без аргументов

Если вы захотите унаследовать класс `Button` в другом классе, не объявляя своих конструкторов, вы должны будете явно вызвать конструктор суперкласса, даже если тот не имеет параметров:

```
class RadioButton: Button()
```

Вот зачем нужны пустые круглые скобки после имени суперкласса. Обратите внимание на отличие от интерфейсов: интерфейсы не имеют конструктора, поэтому при реализации интерфейса никогда не приходится добавлять круглые скобки после его имени в списке супертипов.

Если вы хотите получить гарантии того, что никакой другой код не сможет создавать экземпляров вашего класса, сделайте конструктор приватным с помощью ключевого слова `private`:

```
class Secretive private constructor() {}
```

Конструктор этого класса приватный

Поскольку класс `Secretive` имеет только приватный конструктор, код снаружи класса не сможет создать его экземпляра. Ниже мы поговорим об объектах-компаньонах, которые способны вызывать такие конструкторы.

> **Альтернатива приватным конструкторам**
>
> В Java можно использовать конструктор с модификатором `private`, чтобы запретить создание экземпляров класса — подобные классы служат контейнерами статических вспомогательных методов или реализуют шаблон «Одиночка» (Singleton). В Kotlin для этих же целей используются встроенные механизмы языка. В качестве статических вспомогательных методов используются функции верхнего уровня (которые вы видели в разделе 3.2.3). Для создания «одиночки» используется объявление объекта, которое вы увидите в разделе 4.4.1.

В большинстве сценариев конструктор класса выглядит очень просто: он либо не имеет параметров, либо присваивает их значения соответствующим свойствам. Вот почему в языке Kotlin такой лаконичный синтаксис определения основных конструкторов: он отлично подходит для большинства случаев. Но в жизни всё бывает сложнее, поэтому Kotlin позволяет определить столько конструкторов, сколько потребуется. Давайте посмотрим, как это работает.

4.2.2. Вторичные конструкторы: различные способы инициализации суперкласса

В целом классы с несколькими конструкторами встречаются в Kotlin значительно реже, чем в Java. В большинстве ситуаций, когда в Java нужны перегруженные версии конструктора, в языке Котлин можно воспользоваться значениями параметров по умолчанию и синтаксисом именованных аргументов.

> **Совет.** Не объявляйте несколько дополнительных конструкторов только для определения значений аргументов по умолчанию. Вместо этого указывайте значения по умолчанию напрямую.

Но иногда бывает нужно несколько конструкторов. Наиболее частая ситуация: необходимо расширить класс фреймворка, поддерживающий несколько конструкторов для инициализации класса различными способами. Представьте класс `View`, объявленный в Java, который имеет два конструктора (разработчики для Android без труда узнают это определение). Вот как выглядит аналогичное объявление в Kotlin:

```
open class View {
    constructor(ctx: Context) {
      // некоторый код
    }

    constructor(ctx: Context, attr: AttributeSet) {
      // некоторый код
    }
}
```

Вторичные конструкторы

Этот класс не имеет основного конструктора (это видно по отсутствию круглых скобок после имени в определении класса), зато у него два вторичных конструктора. Вторичный конструктор объявляется с помощью ключевого слова `constructor`. Вы можете объявить столько вторичных конструкторов, сколько нужно.

Чтобы расширить этот класс, объявите те же конструкторы:

```
class MyButton : View {
    constructor(ctx: Context)
      : super(ctx) {
      // ...
    }

    constructor(ctx: Context, attr: AttributeSet)
      : super(ctx, attr) {
      // ...
    }
}
```

Вызов конструкторов суперкласса

Здесь определяются два конструктора, каждый из которых вызывает соответствующий конструктор суперкласса с помощью ключевого слова `super()`. Это проиллюстрировано на рис. 4.3, стрелка указывает, какому конструктору делегируется выполнение.

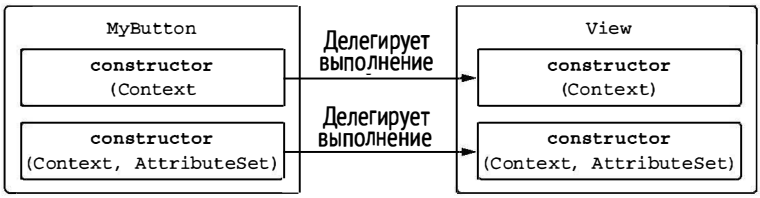


Рис. 4.3. Вызов различных конструкторов суперкласса

Как в Java, в Kotlin есть возможность вызвать один конструктор класса из другого с помощью ключевого слова `this()`. Вот как это работает:

```
class MyButton : View {
    constructor(ctx: Context): this(ctx, MY_STYLE) {
      // ...
```

Делегирует выполнение другому конструктору класса

```
    }

    constructor(ctx: Context, attr: AttributeSet): super(ctx, attr) {
      // ...
    }
}
```

Мы изменили класс объекта `MyButton` так, что один его конструктор делегирует выполнение другому конструктору этого же класса (с помощью `this`), передавая значение параметра по умолчанию, как показано на рис. 4.4. Второй конструктор по-прежнему вызывает `super()`.

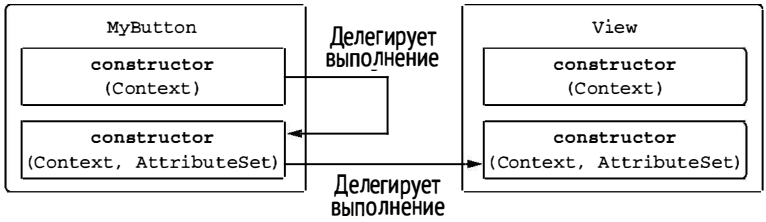


Рис. 4.4. Делегирование выполнения конструктору того же класса

Если класс не имеет основного конструктора, каждый вторичный конструктор должен либо инициализировать базовый класс, либо делегировать выполнение другому конструктору. Согласно предыдущим рисункам, каждый вторичный конструктор должен начинать цепочку вызовов, оканчивающуюся в конструкторе базового класса.

Совместимость с Java – основная причина применения вторичных конструкторов. Но возможна ещё одна ситуация: когда требуется предоставить несколько способов создания экземпляров класса с разными списками параметров. Мы рассмотрим такой пример в разделе 4.4.2.

Мы обсудили порядок определения нетривиальных конструкторов. Теперь давайте обратим внимание на нетривиальные свойства.

4.2.3. Реализация свойств, объявленных в интерфейсах

Интерфейсы в Kotlin могут включать объявления абстрактных свойств. Вот пример определения интерфейса с таким объявлением:

```
interface User {
    val nickname: String
}
```

Классы, реализующие интерфейс `User`, должны предоставить способ получить значение свойства `nickname`. Интерфейс не определяет, как будет доступно значение – как поле или через метод доступа. Поскольку сам интерфейс не имеет состояния, то только реализующие его классы смогут хранить соответствующее значение.

Давайте рассмотрим несколько возможных реализаций этого интерфейса: `PrivateUser`, хранящий только имя, `SubscribingUser`, который хранит адрес электронной почты для регистрации, и `FacebookUser`, готовый поделиться своим идентификатором в Facebook.

Каждый из этих классов реализует абстрактное свойство интерфейса по-своему.

Листинг 4.14. Реализация свойства интерфейса

```
class PrivateUser(override val nickname: String) : User

class SubscribingUser(val email: String) : User {
    override val nickname: String
      get() = email.substringBefore('@') )
    }

class FacebookUser(val accountId: Int) : User {
    override val nickname = getFacebookName(accountId)
}
```

Свойство основного конструктора

Собственный метод чтения

Инициализация свойства

```
>>> println(PrivateUser("test@kotlinlang.org").nickname)
test@kotlinlang.org
>>> println(SubscribingUser("test@kotlinlang.org").nickname)
test
```

Для описания класса `PrivateUser` можно использовать лаконичный синтаксис объявления свойства непосредственно в основном конструкторе. Ключевое слово `override` перед ним означает, что это свойство реализует абстрактное свойство интерфейса `User`.

В классе `SubscribingUser` свойство `nickname` реализуется с помощью метода доступа. У этого свойства отсутствует поле для хранения значения – есть только метод чтения, определяющий имя по адресу электронной почты.

В классе `FacebookUser` значение присваивают свойству `nickname` в коде инициализации. Для доступа к нему используют функцию `getFacebookName`, которая вернёт имя пользователя Facebook по заданному идентификатору (предполагается, что он определен где-то в другом месте). Вызов такой функции — дорогостоящая операция: она должна подключаться к Facebook для получения данных. Вот почему она вызывается лишь один раз, на этапе инициализации.

Обратите внимание на различные реализации свойства `nickname` в классах `SubscribingUser` и `FacebookUser`. Хотя они выглядят похоже, в первом случае определен собственный метод чтения, возвращающий `substringBefore`, а во втором для свойства в классе `FacebookUser` предусмотрено

поле, которое хранит значение, вычисленное во время инициализации класса.

Кроме абстрактных свойств, интерфейс может содержать свойства с методами чтения и записи — при условии, что они не обращаются к полю в памяти (такое поле потребовало бы хранения состояния в интерфейсе, что невозможно). Рассмотрим пример:

```
interface User {
    val email: String
    val nickname: String
        get() = email.substringBefore('@')
}
```

Свойство не имеет поля для хранения значения: результат вычисляется при каждой попытке доступа

Этот интерфейс определяет абстрактное свойство `email`, а также свойство `nickname` с методом доступа. Первое свойство должно быть переопределено в подклассах, а второе может быть унаследовано.

В отличие от свойств, реализованных в интерфейсах, свойства, реализованные в классах, имеют полный доступ к полям, хранящим их значения. Давайте посмотрим, как обращаться к ним из методов доступа.

4.2.4. Обращение к полю из методов доступа

Вы уже видели несколько примеров двух видов свойств: одни просто хранят значения, а другие имеют собственные методы доступа, вычисляющие значения при каждом обращении. Теперь посмотрим, как комбинировать эти два вида, и реализуем свойство, хранящее значение и обладающее дополнительной логикой, выполняющейся при чтении и изменении. Для этого нам потребуются поле для хранения значения и методы доступа.

Представим, что нам нужно организовать журналирование любых изменений в данных, хранящихся в свойстве. Для этого объявим изменяемое поле и выполним дополнительный код при каждом обращении к нему.

Листинг 4.15. Доступ к полю из метода записи

```
class User(val name: String) {
    var address: String = "unspecified"
      set(value: String) {
        println("""
          Address was changed for $name:
          "$field" -> "$value".""".trimIndent())
        field = value
      }
}
```

Чтение значения из поля

Изменение значения поля

```
>>> val user = User("Alice")
>>> user.address = "Elsenheimerstrasse 47, 80687 Muenchen"
```

```
Address was changed for Alice:
"unspecified" -> "Elsenheimerstrasse 47, 80687 Muenchen".
```

Изменить значение свойства можно как обычно: выполнив присваивание `user.address = "новое значение"`, которое за кулисами вызовет метод записи. В этом примере мы переопределили метод записи, чтобы добавить логику журналирования (для простоты мы просто выводим сообщение).

В теле метода записи для доступа к значению поля используется специальный идентификатор `field`. В методе чтения можно только прочитать значение, а в методе записи – прочитать и изменить.

Обратите внимание, что для изменяемого свойства можно переопределить только один из методов доступа. Реализация метода чтения в листинге 4.15 тривиальна и просто возвращает значение поля, так что его не нужно переопределять.

Вы можете спросить: в чем разница между свойствами, имеющими и не имеющими поля для хранения значений? Способ обращения к свойству не зависит от наличия у него отдельного поля. Компилятор сгенерирует такое поле для свойства, если вы будете явно ссылаться на него или использовать реализацию методов доступа по умолчанию. Если вы определите собственные методы доступа, не использующие идентификатора `field` (в методе чтения, если свойство объявлено как `val`, или в обоих методах, если это изменяемое свойство), тогда отдельное поле не будет сгенерировано.

Иногда нет необходимости менять стандартную реализацию методов доступа, но требуется изменить их видимость. Давайте узнаем, как это сделать.

4.2.5. Изменение видимости методов доступа

По умолчанию методы доступа имеют ту же видимость, что и свойство. Но вы можете изменить её, добавив модификатор видимости перед ключевыми словами `get` и `set`. Рассмотрим этот приём на примере.

Листинг 4.16. Объявление свойства с приватным методом записи

```
class LengthCounter {
    var counter: Int = 0
      private set                    <┐ Значение этого свойства нельзя
                                     │ изменить вне класса
    fun addWord(word: String) {
      counter += word.length
    }
}
```

Этот класс вычисляет общую длину слов, добавляемых в него. Свойство, хранящее общую длину, объявлено как `public`, потому что является частью API класса, доступного клиентам. Но мы должны гарантировать, что оно будет изменяться только внутри класса, потому что в противном случае внешний код сможет изменить его и сохранить неверное значение. Поэтому мы позволяем компилятору сгенерировать метод чтения с видимостью по умолчанию и изменяем видимость метода записи на `private`.

Вот как можно использовать этот класс:

```
>>> val lengthCounter = LengthCounter()
>>> lengthCounter.addWord("Hi!")
>>> println(lengthCounter.counter)
3
```

Создаем экземпляр `LengthCounter` и добавляем в него слово `"Hi!"` длиной в 3 символа. Теперь свойство `counter` хранит значение 3.

> **Еще о свойствах**
>
> Далее в этой книге мы продолжим обсуждение свойств. Ниже представлен небольшой указатель:
>
> - Модификатор `lateinit` перед свойством, которое не может принимать значения `null`, обеспечивает инициализацию свойства после вызова конструктора, что часто используют некоторые фреймворки. Эта особенность рассматривается в главе 6.
> - Свойства с отложенной инициализацией — часть более общего класса *делегированных свойств* (delegated properties), о которых рассказывается в главе 7.
> - Для совместимости с Java-фреймворками можно использовать аннотации, имитирующие особенности Java в языке Kotlin. Например, аннотация `@JvmField` перед свойством открывает доступ к полю с модификатором `public` без методов доступа. Подробнее об аннотациях рассказывается в главе 10.
> - Модификатор `const` делает работу с аннотациями удобнее, потому что позволяет использовать свойство простого типа или типа `String` в качестве аргумента аннотации. Подробнее — в главе 10.

На этом мы завершим обсуждение нетривиальных конструкторов и свойств в Kotlin. Далее вы узнаете, как понятие классов данных (data classes) помогает сделать классы объектов-значений более дружелюбными.

4.3. Методы, сгенерированные компилятором: классы данных и делегирование

Платформа Java определяет методы, которые должны присутствовать во многих классах и обычно реализуются платформой автоматически – на-

пример, `equals`, `hashCode` и `toString`. К счастью, Java IDE могут генерировать эти методы самостоятельно, часто избавляя от необходимости писать их вручную. Но в таком случае код вашего проекта будет полон повторяющихся шаблонных фрагментов. Компилятор Kotlin помогает решить эту проблему: он может автоматически генерировать код за кулисами, не загромождая файлов с исходным кодом.

Вы уже знаете, как этот механизм работает в отношении тривиальных конструкторов и методов доступа к свойствам. Давайте рассмотрим примеры, когда компилятор Kotlin генерирует типичные методы для простых классов данных и значительно упрощает делегирование.

4.3.1. Универсальные методы объектов

Как в Java, все классы в Kotlin имеют несколько методов, которые иногда приходится переопределять: `equals`, `hashCode` и `toString`. Давайте посмотрим, что это за методы и как Kotlin помогает автоматически сгенерировать их реализации. В качестве отправной точки возьмем простой класс `Client`, который хранит имя клиента и почтовый индекс.

Листинг 4.17. Первоначальное определение класса `Client`

```
class Client(val name: String, val postalCode: Int)
```

Взглянем на строковое представление его экземпляров.

Строковое представление: метод toString()

Все классы в Kotlin, как в Java, позволяют получить строковое представление объектов. В основном эта функция используется для отладки и журналирования, но её можно применить и в других контекстах. По умолчанию строковое представление объекта выглядит как `Client@5e9f23b4` – не очень информативно. Чтобы изменить его, необходимо переопределить метод `toString`.

Листинг 4.18. Реализация метода `toString()` в классе `Client`

```
class Client(val name: String, val postalCode: Int) {
    override fun toString() = "Client(name=$name, postalCode=$postalCode)"
}
```

Вот как теперь выглядит строковое представление экземпляра:

```
>>> val client1 = Client("Alice", 342562)
>>> println(client1)
Client(name=Alice, postalCode=342562)
```

Не правда ли, так гораздо содержательнее?

Равенство объектов: метод equals()

Все вычисления, связанные с классом Client, происходят вне его. Этот класс просто хранит данные, он должен быть простым и понятным. Однако у вас могут быть свои требования к поведению этого класса. К примеру, вы хотите, чтобы объекты считались равными, если содержат одни и те же данные:

```
>>> val client1 = Client("Alice", 342562)
>>> val client2 = Client("Alice", 342562)
>>> println(client1 == client2)
false
```

В Kotlin оператор == проверяет равенство объектов, а не ссылок. Он компилируется в вызов метода «equals»

Как видите, объекты не равны. Это значит, что для класса Client нужно переопределить метод equals.

> **Оператор == и проверка на равенство**
>
> Для сравнения простых и ссылочных типов в Java можно использовать оператор ==. Когда он применяется к простым типам, сравниваются значения, а для ссылочных типов сравниваются указатели. В результате в Java появились широко распространенная практика всегда вызывать метод equals и распространенная проблема: это часто забывают сделать.
>
> В Kotlin оператор == представляет способ сравнения двух объектов по умолчанию: он сравнивает их, вызывая за кулисами метод equals. То есть, если вы переопределили метод equals в своем классе, можете спокойно сравнить экземпляры с помощью оператора ==. Для сравнения ссылок можно использовать оператор ===, который работает точно как оператор == в Java, выполняя сравнение указателей.

Посмотрим, как изменится класс Client.

Листинг 4.19. Реализация метода equals() в классе Client

```
class Client(val name: String, val postalCode: Int) {
    override fun equals(other: Any?): Boolean {
      if (other == null || other !is Client)
        return false
      return name == other.name &&
        postalCode == other.postalCode
    }
    override fun toString() = "Client(name=$name, postalCode=$postalCode)"
}
```

«Any» – это аналог java.lang.Object: суперкласс всех классов в Kotlin. Знак вопроса в «Any?» означает, что аргумент «other» может иметь значение null

Убедиться, что «other» имеет тип Client

Вернуть результат сравнения свойств

Напомним, что оператор is является аналогом instanceof в Java и проверяет, имеет ли значение слева тип, указанный справа. Подобно оператору !in, который возвращает инвертированный результат проверки in (об

этом мы рассказывали в разделе 2.4.4), оператор `!is` выражает отрицание проверки `is`. Такие операторы делают код более читабельным. В главе 6 мы обсудим типы, которые могут принимать значение `null`, и расскажем, почему выражение `other == null || other !is Client` можно упростить до `other !is Client`.

Модификатор `override`, ставший в Kotlin обязательным, защищает от ошибочного объявления `fun equals(other: Client)`, которое добавит новый метод `equals` вместо переопределения имеющегося. После переопределения метода `equals` можно ожидать, что экземпляры с одинаковыми значениями свойств будут равны. Действительно, операция сравнения `client1 == client2` в предыдущем примере теперь возвращает `true`. Но если вы захотите проделать с этими объектами некоторые более сложные операции, ничего не получится. Это частый вопрос на собеседовании: что не так и в чем проблема? Вероятно, вы ответите, что проблема в отсутствии метода `hashCode`. Это действительно так, и сейчас мы обсудим, почему это важно.

Контейнеры, применяющие хэш-функции: метод `hashCode()`

Вместе с `equals` всегда должен переопределяться метод `hashCode`. В этом разделе мы объясним, почему.

Давайте создадим множество с одним элементом: клиентом по имени Alice. Затем создадим новый экземпляр, содержащий такие же данные, и проверим, присутствует ли он в этом множестве. Вы, наверное, ожидаете, что проверка вернет `true`, потому что свойства экземпляров равны, но на самом деле она вернет `false`.

```
>>> val processed = hashSetOf(Client("Alice", 342562))
>>> println(processed.contains(Client("Alice", 342562)))
false
```

Причина в том, что в классе `Client` отсутствует метод `hashCode`. Следовательно, нарушается основной контракт метода `hashCode`: если два объекта равны, они должны иметь одинаковый хэш-код. Множество из примера является экземпляром `HashSet`. Сравнение значений в `HashSet` оптимизировано: сначала сравниваются их хэш-коды, и только если они равны, сравниваются фактические значения. Хэш-коды двух экземпляров класса `Client` в предыдущем примере не совпадают, поэтому множество решает, что не содержит второго объекта, хотя метод `equals` будет возвращать `true`. Следовательно, если правило не соблюдается, `HashSet` не сможет корректно работать с такими объектами.

Чтобы исправить проблему, добавим реализацию метода `hashCode`.

Листинг 4.20. Реализация метода hashCode() в классе Client

```
class Client(val name: String, val postalCode: Int) {
```

```
    ...
    override fun hashCode(): Int = name.hashCode() * 31 + postalCode
}
```

Теперь у нас есть класс, который всегда будет работать правильно, – но обратите внимание, сколько кода пришлось для этого написать! К счастью, компилятор Kotlin может помочь, создав все эти методы автоматически. Давайте посмотрим, как можно это сделать.

4.3.2. Классы данных: автоматическая генерация универсальных методов

Чтобы класс стал максимально удобным для хранения данных, следует переопределить следующие методы: `toString`, `equals` и `hashCode`. Как правило, эти методы имеют тривиальную реализацию, и IDE (такие как IntelliJ IDEA) способны создавать их автоматически и проверять, что их реализация корректна и последовательна.

Самое замечательное, что в Kotlin вам не придется создавать всех этих методов. Добавьте в объявление класса модификатор `data` – и все необходимые методы появятся автоматически.

Листинг 4.21. Класс `Client` как класс данных

```
data class Client(val name: String, val postalCode: Int)
```

Как просто, не правда ли? Теперь у вас есть класс, переопределяющий стандартные Java-методы:

- `equals` для сравнения экземпляров;
- `hashCode` для использования экземпляров в качестве ключей в контейнерах на основе хэш-функций, таких как `HashMap`;
- `toString` для создания строкового представления, показывающего все поля в порядке их объявления.

Методы `equals` и `hashCode` учитывают все свойства, объявленные в основном конструкторе. Сгенерированный метод `equals` проверяет равенство значений всех свойств. Метод `hashCode` возвращает значение, зависящее от хэш-кодов всех свойств. Обратите внимание, что свойства, не объявленные в основном конструкторе, не принимают участия в проверках равенства и вычислении хэш-кода.

И это только часть полезных методов, которые автоматически создаются для класса данных. В следующим разделе описан ещё один, а в разделе 7.4 вы познакомитесь с оставшимися.

Классы данных и неизменяемые значения: метод `copy()`

Обратите внимание: даже притом, что свойства класса данных не обязательно должны объявляться с модификатором `val` (можно использовать

var), мы настоятельно рекомендуется использовать свойства, доступные только для чтения, чтобы сделать экземпляры класса *неизменяемыми*. Это необходимо, чтобы экземпляры можно было использовать в качестве ключей в HashMap или аналогичном контейнере – иначе контейнер может оказаться в некорректном состоянии, если объект, используемый в качестве ключа, изменится после добавления в контейнер. Кроме того, неизменяемые объекты существенно упрощают понимание кода, особенно многопоточного: после создания объект останется в исходном состоянии, и вам не придется беспокоиться о других потоках, способных изменить объект, пока ваш код будет с ним работать.

Чтобы ещё упростить использование классов данных в качестве неизменяемых объектов, компилятор Kotlin генерирует для них метод, который позволяет *копировать* экземпляры, изменяя значения некоторых свойств. Как правило, создание копии – хорошая альтернатива модификации экземпляра на месте: копия имеет собственный жизненный цикл и не влияет на код, ссылающийся на исходный экземпляр. Вот как выглядел бы метод копирования, реализуй вы его без компилятора:

```
class Client(val name: String, val postalCode: Int) {
    ...
    fun copy(name: String = this.name,
             postalCode: Int = this.postalCode) =
        Client(name, postalCode)
}
```

А так его можно использовать:

```
>>> val bob = Client("Bob", 973293)
>>> println(bob.copy(postalCode = 382555))
Client(name=Bob, postalCode=382555)
```

Как видите, модификатор data делает классы объектов-значений удобнее в использовании. Теперь давайте поговорим о другой особенности Kotlin, которая позволяет избавиться от шаблонного кода, сгенерированного IDE в делегировании.

4.3.3. Делегирование в классах. Ключевое слово by

Частая проблема при проектировании крупных объектно-ориентированных систем – нестабильность из-за наследования реализации. Когда вы наследуете класс и переопределяете некоторые его методы, ваш код становится зависимым от деталей реализации наследуемого класса. Если система продолжает развиваться – меняется реализация базового класса или в него добавляют новые методы, – прежние предположения о его поведении могут оказаться неверными, и ваш код перестанет вести себя корректно.

Эта проблема нашла свое отражение в дизайне языка Kotlin – все классы по умолчанию получают модификатор `final`. Это гарантирует, что вы сможете наследовать только те классы, для которых такая возможность предусмотрена. Работая над таким классом, вы будете видеть, что он открыт, и учитывать совместимость любых изменений с производными классами.

Но часто бывает нужно добавить поведение в другой класс, даже если он не предназначен для наследования. Для этого применяется шаблон «Декоратор». Он создает новый класс с тем же интерфейсом, что у оригинального класса, и сохраняет экземпляр оригинального класса в поле нового класса. Методы, поведение которых должно остаться неизменным, просто передают вызовы оригинальному экземпляру класса.

Недостаток такого подхода – большой объем шаблонного кода (его так много, что некоторые IDE, такие как IntelliJ IDEA, поддерживают специальную возможность создания такого кода за вас). Например, вот сколько кода понадобится декоратору, чтобы реализовать простой интерфейс `Collection` – даже притом, что он не изменяет поведения исходного класса.

```
class DelegatingCollection<T> : Collection<T> {
    private val innerList = arrayListOf<T>()

    override val size: Int get() = innerList.size
    override fun isEmpty(): Boolean = innerList.isEmpty()
    override fun contains(element: T): Boolean = innerList.contains(element)
    override fun iterator(): Iterator<T> = innerList.iterator()
    override fun containsAll(elements: Collection<T>): Boolean =
        innerList.containsAll(elements)
}
```

К счастью, при использовании Kotlin писать столько кода не нужно, потому что он предоставляет полноценную поддержку делегирования на уровне языка. Всякий раз, реализуя интерфейс, вы можете *делегировать* реализацию другому объекту, добавив ключевое слово `by`. Вот как выглядит предыдущий пример с использованием этой особенности:

```
class DelegatingCollection<T>(
    innerList: Collection<T> = ArrayList<T>()
) : Collection<T> by innerList {}
```

Все реализации методов в классе исчезли. Компилятор сам сгенерирует их, и фактическая реализация будет похожа на ту, что вы видели в примере с `DelegatingCollection`. Поскольку такой код содержит мало интересного, нет смысла писать его вручную, если компилятор может делать всё то же самое автоматически.

Теперь, если вам понадобится изменить поведение некоторых методов, вы сможете переопределить их и вместо сгенерированных методов вызывать ваш код. Вы можете пропустить методы, реализация которых

по умолчанию устраивает вас, делегируя выполнение декорируемому экземпляру.

Давайте посмотрим, как применить эту технику для реализации коллекции, которая подсчитывает количество попыток добавления элементов в неё. Например, если вам нужно избавляться от дублирующихся элементов, такая коллекция позволит вам измерить, насколько эффективно вы это делаете, сравнивая количество попыток добавления элемента с размером коллекции.

Листинг 4.22. Делегирование

```
class CountingSet<T>(
    val innerSet: MutableCollection<T> = HashSet<T>()
) : MutableCollection<T> by innerSet {

    var objectsAdded = 0

    override fun add(element: T): Boolean {
      objectsAdded++
      return innerSet.add(element)
    }

    override fun addAll(c: Collection<T>): Boolean {
      objectsAdded += c.size
      return innerSet.addAll(c)
    }
}
```

Делегирование реализации MutableCollection объекту в поле innerSet

Собственная реализация вместо делегирования

```
>>> val cset = CountingSet<Int>()
>>> cset.addAll(listOf(1, 1, 2))
>>> println("${cset.objectsAdded} objects were added, ${cset.size} remain")
3 objects were added, 2 remain
```

Как видите, здесь переопределяются методы `add` и `addAll`, которые увеличивают значение счетчика, а реализация остальных методов интерфейса `MutableCollection` делегируется декорируемому контейнеру.

Самое важное, что этот прием не создает зависимости от особенностей реализации основной коллекции. Например, нет необходимости беспокоиться, как коллекция реализует метод `addAll` – путем вызова метода `add` в цикле или используя другую реализацию, оптимизированную для конкретного случая. Когда клиентский код обращается к вашему классу, вы полностью контролируете происходящее, можете опираться на документацию API декорируемой коллекции для реализации своих операций и рассчитывать, что всё продолжит работать.

Мы закончили обсуждение способности компилятора Kotlin генерировать полезные методы для классов. Давайте перейдем к заключительной части в повествовании о классах Kotlin: к ключевому слову `object` и различным ситуациям, когда оно вступает в игру.

4.4. Ключевое слово object: совместное объявление класса и его экземпляра

Ключевое слово `object` используется в языке Kotlin в разных случаях, которые объединены общей идеей: это ключевое слово одновременно объявляет класс и создает его экземпляр (другими словами, объект). Рассмотрим различные ситуации, когда оно используется:

- *объявление объекта* как способ реализации шаблона «Одиночка»;
- реализация *объекта-компаньона*, содержащего лишь фабричные методы, а также методы, связанные с классом, но не требующие обращения к его экземпляру. К членам такого объекта можно обращаться просто по имени класса;
- запись *объекта-выражения*, которое используется вместо анонимного внутреннего класса Java.

Теперь обсудим все это подробнее.

4.4.1. Объявление объекта: простая реализация шаблона «Одиночка»

Часто в объектно-ориентированных системах встречается класс, который должен существовать только в одном экземпляре. В Java это реализуется с помощью шаблона «Одиночка»: вы определяете класс с приватным конструктором и статическим полем, содержащим единственный существующий экземпляр этого класса.

Kotlin обеспечивает поддержку такого решения на уровне языка, предлагая синтаксис *объявления объекта*. Объявление объекта сочетает в себе *объявление класса* и *единственного экземпляра* этого класса. Например, можно объявить объект, который представляет фонд заработной платы организации. Наверняка у вас нет нескольких фондов, поэтому использование одного объекта представляется разумным:

```
object Payroll {
    val allEmployees = arrayListOf<Person>()

    fun calculateSalary() {
      for (person in allEmployees) {
        ...
      }
```

```
    }
}
```

Объявление объекта начинается с ключевого слова `object` и фактически определяет класс с переменной этого класса в одном выражении.

По аналогии с классом объявление объекта может содержать определения свойств, методов, блоков инициализации и т. д. Единственное, что не допускается, – конструкторы, основные или вторичные. В отличие от экземпляров обычных классов, объявления объектов создаются непосредственно в точке определения, а не через вызов конструктора из других мест в коде. Следовательно, определять конструктор в объявлении объекта не имеет смысла.

Как и обычные переменные, объявления объектов позволяют вызывать методы и обращаться к свойствам с помощью имени объекта слева от символа `.`:

```
Payroll.allEmployees.add(Person(...))
Payroll.calculateSalary()
```

Объявления объектов также могут наследовать классы и интерфейсы. Это бывает полезно, когда используемый фреймворк требует реализации интерфейса, но в вашей реализации нет нужного состояния. Например, рассмотрим интерфейс `java.util.Comparator` в Java. Экземпляр `Comparator` принимает два объекта и возвращает целое число, указывающее, какой из объектов больше. Такой компаратор никогда не хранит данных, поэтому для каждого конкретного способа сравнения объектов достаточно одного компаратора. Это идеальный случай для использования объявления объекта.

В качестве конкретного примера давайте реализуем компаратор, сравнивающий пути к файлам без учета регистра.

Листинг 4.23. Реализация интерфейса `Comparator` с помощью объявления объекта

```
object CaseInsensitiveFileComparator : Comparator<File> {
    override fun compare(file1: File, file2: File): Int {
      return file1.path.compareTo(file2.path,
        ignoreCase = true)
    }
}

>>> println(CaseInsensitiveFileComparator.compare(
... File("/User"), File("/user")))
0
```

Объект-одиночку можно использовать в любом контексте, где используется обычный объект (экземпляр класса), – например, передать этот объект в качестве аргумента функции, принимающей экземпляр `Comparator`:

```
>>> val files = listOf(File("/Z"), File("/a"))
>>> println(files.sortedWith(CaseInsensitiveFileComparator))
[/a, /Z]
```

> **Объекты-одиночки и внедрение зависимостей**
>
> Как и шаблон «Одиночка», объявления объектов не всегда пригодны для использования в больших программных системах. Они прекрасно подходят для небольших модулей с малым количеством зависимостей или вообще без них, но не для крупных компонентов, взаимодействующих с другими частями системы. Основная причина — в отсутствии контроля над созданием объектов и невозможности управлять параметрами конструкторов.
>
> Это означает, что в модульных тестах или в разных конфигурациях системы нет возможности заменить реализацию объекта или других классов, зависящих от него. Если вам нужна такая функциональность, наряду с обычными классами Kotlin вы должны использовать библиотеку для внедрения зависимостей (например, Guice, https://github.com/google/guice), как в Java.

Объекты также можно объявлять в классах. Такие объекты существуют в единственном числе – у вас не будет отдельного объекта для каждого экземпляра содержащего его класса. Например, логично разместить компаратор, сравнивающий объекты определенного класса, внутри этого класса.

Листинг 4.24. Реализация интерфейса Comparator как вложенного объекта

```
data class Person(val name: String) {
    object NameComparator : Comparator<Person> {
      override fun compare(p1: Person, p2: Person): Int =
        p1.name.compareTo(p2.name)
    }
}
>>> val persons = listOf(Person("Bob"), Person("Alice"))
>>> println(persons.sortedWith(Person.NameComparator))
[Person(name=Alice), Person(name=Bob)]
```

> **Объекты-одиночки Kotlin в Java**
>
> Объявление объекта в Kotlin компилируется в класс со статическим полем, хранящим его единственный экземпляр, который всегда называется INSTANCE. Реализуя шаблон «Одиночка» в Java, вы наверняка сделали бы то же самое вручную. Поэтому, чтобы использовать объект Kotlin из Java-кода, нужно обращаться к статическому полю экземпляра:
>
> ```
> /* Java */
> CaseInsensitiveFileComparator.INSTANCE.compare(file1, file2);
> ```
>
> В этом примере поле INSTANCE имеет тип CaseInsensitiveFileComparator.

А сейчас рассмотрим особую категорию объектов, хранящихся внутри класса: *объекты-компаньоны*.

4.4.2. Объекты-компаньоны: место для фабричных методов и статических членов класса

Классы в Kotlin не могут иметь статических членов; ключевое слово `static`, имеющееся в Java, не является частью языка Kotlin. В качестве замены Kotlin используются функции уровня пакета (которые во многих ситуациях могут заменить статические методы Java) и объявления объектов (которые заменяют статические методы Java в других случаях, наряду со статическими полями). В большинстве случаев рекомендуется использовать функции верхнего уровня. Но, как видно на рис. 4.5, такие функции не имеют доступа к приватным членам класса. Поэтому, чтобы написать функцию, которую можно вызывать без экземпляра класса, но с доступом к внутреннему устройству класса, вы можете сделать её членом объявления объекта внутри класса. Примером такой функции может служить фабричный метод.

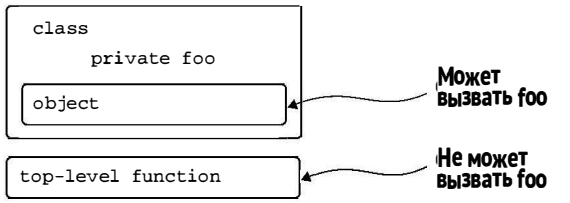


Рис. 4.5. Функции верхнего уровня не имеют доступа к приватным членам класса

Один из объектов в классе можно отметить специальным ключевым словом: `companion`. Сделав это, вы сможете обращаться к методам и свойствам такого объекта непосредственно через имя содержащего его класса, без явного указания имени объекта. В итоге синтаксис выглядит почти так же, как вызов статического метода в Java. Вот простой пример такого синтаксиса:

```
class A {
    companion object {
      fun bar() {
        println("Companion object called")
      }
    }
}

>>> A.bar()
Companion object called
```

Помните, мы обещали показать хороший пример вызова приватного конструктора? Это объект-компаньон. Он имеет доступ ко всем приватным членам класса и идеально подходит для реализации шаблона «Фабрика».

Давайте рассмотрим пример с объявлением двух конструкторов, а затем изменим его, чтобы он использовал фабричные методы в объекте-компаньоне. За основу возьмем листинг 4.14 с классами `FacebookUser` и `SubscribingUser`. Ранее эти сущности представляли разные классы, реализующие общий интерфейс `User`. Теперь мы решили использовать только один класс, но разные способы его создания.

Листинг 4.25. Определение класса с несколькими вторичными конструкторами

```
class User {
    val nickname: String

    constructor(email: String) {
      nickname = email.substringBefore('@')
    }

    constructor(facebookAccountId: Int) {
      nickname = getFacebookName(facebookAccountId)
    }
}
```

Вторичные конструкторы

Альтернативный способ реализации той же логики, который может быть полезен по многим причинам, — использовать фабричные методы для создания экземпляров класса. Экземпляр `User` создается вызовом фабричного метода, а не с помощью различных конструкторов.

Листинг 4.26. Замещение вторичных конструкторов фабричными методами

```
class User private constructor(val nickname: String) {
    companion object {
      fun newSubscribingUser(email: String) =
        User(email.substringBefore('@'))

      fun newFacebookUser(accountId: Int) =
        User(getFacebookName(accountId))
    }
}
```

Основной конструктор объявлен приватным

Объявление объекта-компаньона

Фабричный метод создает нового пользователя на основе идентификатора в Facebook

Вы можете вызвать объект-компаньон через имя класса:

```
>>> val subscribingUser = User.newSubscribingUser("bob@gmail.com")
>>> val facebookUser = User.newFacebookUser(4)
>>> println(subscribingUser.nickname)
bob
```

Фабричные методы очень полезны. Как показано в примере, им можно давать говорящие имена. Кроме того, фабричный метод может возвращать подклассы того класса, в котором объявлен метод (как в данном примере, где `SubscribingUser` и `FacebookUser` – классы). Также можно запретить создание новых объектов, когда это нужно. Например, можно проверять соответствие адреса электронной почты уникальному экземпляру `User` и возвращать существующий экземпляр, а не создавать новый, если адрес электронной почты уже присутствует в кэше. Но если такие классы понадобится расширять, то использование нескольких конструкторов может оказаться лучшим решением, поскольку объекты-компаньоны не могут переопределяться в подклассах.

4.4.3. Объекты-компаньоны как обычные объекты

Объект-компаньон – это обычный объект, объявленный в классе. Он может иметь имя, реализовать интерфейс или обладать функциями-расширениями и свойствами-расширениями. В этом разделе мы рассмотрим ещё один пример.

Предположим, вы работаете над веб-службой для расчета заработной платы предприятия и вам нужно сериализовать и десериализовать объекты в формате JSON. Можно поместить логику сериализации в объект-компаньон.

Листинг 4.27. Объявление именованного объекта-компаньона

```
class Person(val name: String) {
    companion object Loader {
      fun fromJSON(jsonText: String): Person = ...
    }
}

>>> person = Person.Loader.fromJSON("{name: 'Dmitry'}")
>>> person.name
Dmitry
>>> person2 = Person.fromJSON("{name: 'Brent'}")
>>> person2.name
Brent
```

Для вызова метода fromJSON можно использовать оба подхода

В большинстве случаев можно ссылаться на объект-компаньон через имя содержащего его класса, поэтому не нужно беспокоиться о выборе имени для него самого, если нет необходимости (как в листинге 4.27: `companion object Loader`). Если имя объекта-компаньона не указано, по умолчанию выбирается имя `Companion`. Вы увидите некоторые примеры использования этого имени ниже, когда мы будет говорить о расширении объектов-компаньонов.

Объекты-компаньоны и статические члены в Kotlin

Объект-компаньон класса компилируется так же, как обычный объект: статическое поле класса ссылается на собственный экземпляр. Если объект не имеет имени, к нему можно обратиться из Java-кода через ссылку `Companion`:

```
/* Java */
Person.Companion.fromJSON("...");
```

Если у объекта есть имя, вместо `Companion` нужно использовать его.

Но вам может понадобиться работать с Java-кодом, который требует, чтобы методы вашего класса были статическими. Для этого добавьте перед соответствующим методом аннотацию `@JvmStatic`. Если понадобится объявить статическое поле, используйте аннотацию `@JvmField` перед свойством верхнего уровня или свойством, объявленным в объекте. Эти аннотации нужны только для совместимости и, строго говоря, не являются частью ядра языка. Мы подробно рассмотрим их в главе 10.

Также обратите внимание, что Kotlin можете обращаться к статическим методам и полям, объявленным в Java-классах, используя тот же синтаксис, что в Java.

Реализация интерфейсов в объектах-компаньонах

Как любые другие объявления объектов, объекты-компаньоны могут реализовать интерфейсы. Как вы скоро узнаете, имя содержащего объект класса можно использовать непосредственно, в качестве экземпляра объекта, реализующего интерфейс.

Предположим, в вашей системе много типов объектов, включая `Person`. Вы хотели бы обеспечить единый способ создания объектов всех типов. Допустим, у вас есть интерфейс `JSONFactory` для объектов, которые можно десериализовать из формата JSON, и все объекты в вашей системе должны создаваться с помощью этой фабрики. Реализуйте этот интерфейс в классе `Person`.

Листинг 4.28. Реализация интерфейса в объекте-компаньоне

```
interface JSONFactory<T> {
    fun fromJSON(jsonText: String): T
}

class Person(val name: String) {
    companion object : JSONFactory<Person> {
      override fun fromJSON(jsonText: String): Person = ...
    }
}
```

Объект-компаньон, реализующий интерфейс

Далее, если у вас есть функция, использующая абстрактную фабрику для загрузки сущностей, передайте ей объект `Person`.

```
fun loadFromJSON<T>(factory: JSONFactory<T>): T {
    ...
}

loadFromJSON(Person)
```

Передача объекта-компаньона в функцию

Обратите внимание, что класс `Person` используется как экземпляр `JSONFactory`.

Расширение объектов-компаньонов

Как вы видели в разделе 3.3, функции-расширения позволяют определять, какие методы могут вызываться для экземпляров классов, определенных в другом месте в коде. Но что, если понадобится создать функцию, которую можно вызывать для самого класса так же, как методы объектов-компаньонов или статические методы в Java? Если у класса есть объект-компаньон, это можно сделать путем определения функции-расширения для него. Говоря более конкретно, если у класса `C` есть объект-компаньон и вы определили функцию-расширение `func` для объекта `C.Companion`, её можно вызвать как `C.func`.

К примеру, требуется обеспечить четкое разделение обязанностей в вашем классе `Person`. Сам класс будет частью модуля основной бизнес-логики, но вы не хотите связывать этот модуль с каким-либо конкретным форматом данных. Вследствие этого функция десериализации должна быть определена в модуле, отвечающем за взаимодействие между клиентом и сервером. Добиться этого можно с помощью функции-расширения. Обратите внимание, как имя по умолчанию (`Companion`) используется для ссылки на объект-компаньон, который объявлен без имени:

Листинг 4.29. Определение функции-расширения для объекта-компаньона

```
// модуль реализации бизнес-логики
class Person(val firstName: String, val lastName: String) {
    companion object {
    }
}
```

Объявление пустого объекта-компаньона

```
// модуль реализации взаимодействий между клиентом и сервером
fun Person.Companion.fromJSON(json: String): Person {
    ...
}

val p = Person.fromJSON(json)
```

Объявление функции-расширения

Функцию `fromJSON` можно вызывать, как если бы она была определена как метод объекта-компаньона, но на самом деле она определена вне его, как функция-расширение. Как всегда бывает с функциями-расширениями, она выглядит как член класса, но не является им. Но учтите, что в вашем классе нужно объявить объект-компаньон, пусть даже пустой, чтобы иметь возможность создать расширение для него.

Вы видели, как полезны объекты-компаньоны. Теперь давайте перейдем к следующей особенности языка Kotlin, реализуемой с помощью того же ключевого слова `object`: объектам-выражениям.

4.4.4. Объекты-выражения: другой способ реализации анонимных внутренних классов

Ключевое слово `object` можно использовать не только для объявления именованных объектов-одиночек, но и для создания *анонимных объектов*. Анонимные объекты заменяют анонимные внутренние классы в Java. Например, вот как перевести обычный пример использования анонимных внутренних классов в Java – реализацию обработчика событий – на язык Kotlin:

Листинг 4.30. Реализация обработчика событий с помощью анонимного объекта

```
window.addMouseListener(
    object : MouseAdapter() {
      override fun mouseClicked(e: MouseEvent) {
        // ...
      }

      override fun mouseEntered(e: MouseEvent) {
        // ...
      }
    }
)
```

Объявление анонимного объекта, наследующего MouseAdapter

Переопределение методов MouseAdapter

Синтаксис ничем не отличается от объявления объекта, за исключением указания его имени (здесь оно отсутствует). Объект-выражение объявляет класс и создает экземпляр этого класса, но не присваивает имени ни классу, ни экземпляру. Как правило, в этом нет необходимости, поскольку объект используется в качестве параметра вызова функции. Если объекту потребуется дать имя, его можно сохранить в переменной:

```
val listener = object : MouseAdapter() {
  override fun mouseClicked(e: MouseEvent) { ... }
  override fun mouseEntered(e: MouseEvent) { ... }
}
```

В отличие от анонимных внутренних классов Java, которые могут наследовать только один класс или реализовать только один интерфейс, анонимный объект Kotlin может реализовывать несколько интерфейсов или вовсе ни одного.

Примечание. В отличие от объявления объекта, анонимные объекты — не «одиночки». При каждом выполнении объекта-выражения создается новый экземпляр объекта.

Как анонимные классы в Java, код в объекте-выражении может обращаться к переменным в функциях, где он был создан. Но, в отличие от Java, это не ограничено переменными с модификатором `final`; объект-выражение может также изменять значения переменных. Например, посмотрим, как с помощью обработчика событий подсчитать количество щелчков мышью в окне.

Листинг 4.31. Доступ к локальным переменным из анонимного объекта

```
fun countClicks(window: Window) {
    var clickCount = 0                                  <-- Объявление локальной переменной

    window.addMouseListener(object : MouseAdapter() {
      override fun mouseClicked(e: MouseEvent) {
        clickCount++                                    <-- Изменение значения переменной
      }
    })
    // ...
}
```

Примечание. Объекты-выражения полезны, когда в анонимном объекте нужно переопределить несколько методов. Если же требуется реализовать только один метод интерфейса (такого как `Runnable`), то можно рассчитывать на поддержку в Kotlin преобразований для интерфейсов с одним абстрактным методом (SAM conversion) – преобразование литерала функции в реализацию интерфейса с одним абстрактным методом – и написать свою реализацию в виде литерала функции (лямбда-выражения). Мы обсудим лямбда-выражения и SAM-преобразования более подробно в главе 5.

Мы закончили обсуждение классов, интерфейсов и объектов. В следующей главе перейдем к одному из самых интересных разделов языка Kotlin: лямбда-выражениям и функциональному программированию.

4.5. Резюме

- Интерфейсы в Kotlin похожи на интерфейсы в Java, но могут включать свойства и реализации методов по умолчанию (это доступно в Java только с версии 8).

- Всем объявлениям по умолчанию присваиваются модификаторы `final` и `public`.
- Модификатор `open` отменяет действие модификатора `final`.
- Объявления с модификатором `internal` видны только в том же модуле.
- Вложенные классы по умолчанию не становятся внутренними. Чтобы сохранить ссылку на внешний класс, используйте модификатор `inner`.
- У запечатанного класса с модификатором `sealed` подклассы могут определяться только внутри него (Kotlin 1.1 позволяет располагать такие объявления в любом месте в том же файле).
- Блоки инициализации и вторичные конструкторы дают дополнительную гибкость инициализации экземпляров классов.
- Идентификатор `field` используется в методах доступа для ссылки на соответствующее поле, хранящее значение.
- Для классов данных компилятор автоматически генерирует методы `equals`, `hashCode`, `toString`, `copy` и др.
- Поддержка делегирования позволяет избавиться от множества делегирующих методов в вашем коде.
- Объявление объекта – это способ создания «одиночки» в Kotlin.
- Объекты-компаньоны (наряду с функциями и свойствами верхнего уровня) используются вместо статических методов и полей Java.
- Объекты-компаньоны, как и другие объекты, могут реализовать интерфейсы; у них могут быть функции- и свойства-расширения.
- Объекты-выражения в Kotlin заменяют анонимные внутренние классы в Java. Вдобавок они могут реализовать несколько интерфейсов и изменять значения переменных в области видимости, где были созданы.

Глава 5

Лямбда-выражения

В этой главе:

- лямбда-выражения и ссылки на члены класса;
- работа с коллекциями в функциональном стиле;
- последовательности: откладывание операций с коллекциями;
- функциональные интерфейсы Java в Kotlin;
- использование лямбда-выражений с получателями.

Лямбда-выражения – это небольшие фрагменты кода, которые можно передавать другим функциям. Благодаря поддержке лямбда-выражений вы легко сможете выделить общий код в библиотечные функции, и стандартная библиотека Kotlin активно этим пользуется. Чаще всего лямбда-выражения применяются для работы с коллекциями, и в этой главе приведено множество примеров замены типичных шаблонов обработки коллекций на лямбда-выражения, которые передаются функциям стандартной библиотеки. Вы также увидите, как использовать лямбда-выражения с Java-библиотеками – даже с теми, которые не были изначально спроектированы для работы с ними. Наконец, мы рассмотрим лямбда-выражения с получателями – особый вид лямбда-выражений, тело которых выполняется в другом контексте, нежели окружающий код.

5.1. Лямбда-выражения и ссылки на члены класса

Появление лямбда-выражений в Java 8 стало одним из самых долгожданных изменений в языке. Почему это так важно? В этом разделе вы узнаете, почему лямбда-выражения так полезны и как синтаксис лямбда-выражений выглядит в Kotlin.

5.1.1. Введение в лямбда-выражения: фрагменты кода как параметры функций

Очень часто приходится решать задачу передачи и хранения некоторого поведения в коде. Например, нередко требуется выразить такие идеи, как «Когда произойдет событие, запустить этот обработчик» или «Применить эту операцию ко всем элементам в структуре данных». В старых версиях Java такие задачи решались с помощью анонимных внутренних классов. Это вполне действенный прием, но он требует громоздких синтаксических конструкций.

Функциональное программирование предлагает другой подход к решению этой проблемы: возможность использования функций в качестве значений. Вместо объявления класса и передачи его экземпляра в функцию можно передать функцию непосредственно. С лямбда-выражениями код становится более компактным. Вам даже не нужно объявлять функцию – вместо этого можно просто передать блок кода в параметре функции.

Рассмотрим пример. Представьте, что нам нужно определить реакцию на нажатие кнопки. Мы добавляем обработчик, который отвечает за обработку события щелчка и реализует соответствующий интерфейс `OnClickListener` с единственным методом `onClick`.

Листинг 5.1. Реализация обработчика событий с помощью анонимного внутреннего класса

```
/* Java */
button.setOnClickListener(new OnClickListener() {
  @Override
  public void onClick(View view) {
    /* действия по щелчку */
  }
});
```

Избыточность, сопутствующая объявлению анонимного внутреннего класса, при многократном повторении начинает раздражать. Способ, позволяющий выразить только то, должно быть сделано при нажатии, помогает избавиться от лишнего кода. В Kotlin, как и в Java 8, с этой целью можно использовать лямбда-выражение.

Листинг 5.2. Реализация обработчика событий с помощью лямбда-выражения

```
button.setOnClickListener { /* действия по щелчку */ }
```

Этот код на Kotlin делает то же самое, что анонимный класс в Java, но он лаконичнее и читабельнее. Мы обсудим особенности данного примера ниже.

Вы увидели, как лямбда-выражения могут использоваться вместо анонимных объектов с единственным методом. Давайте рассмотрим еще одну классическую область применения лямбда-выражений – операции с коллекциями.

5.1.2. Лямбда-выражения и коллекции

Один из главных принципов хорошего стиля программирования – избегать любого дублирования кода. Большинство задач, которые мы решаем, работая с коллекциями, следует одному из хорошо известных шаблонов, а значит, код, реализующий эти шаблоны, должен размещаться в библиотеке. Но без лямбда-выражений сложно создать хорошую и удобную библиотеку для работы с коллекциями. Конечно, если вы писали свой код на Java до версии 8, у вас, скорее всего, выработалась привычка делать все самостоятельно. С приходом Kotlin эта привычка больше не нужна!

Рассмотрим пример. Здесь мы будем использовать класс `Person`, который хранит информацию о человеке: его имя и возраст.

```
data class Person(val name: String, val age: Int)
```

Предположим, что нужно найти самого старого человека в представленном списке людей. Те, кто не имеют опыта работы с лямбда-выражениями, наверняка поспешат выполнить поиск вручную: добавят две промежуточные переменные – одну для хранения максимального возраста и другую для хранения найденного имени человека этого возраста, – а затем выполнят перебор списка, обновляя эти переменные.

Листинг 5.3. Поиск в коллекции вручную

```
fun findTheOldest(people: List<Person>) {
  var maxAge = 0                              <— Хранит максимальный возраст
  var theOldest: Person? = null               <— Хранит самого старого человека
  for (person in people) {
    if (person.age > maxAge) {                <— Если следующий старше предыдущего,
      maxAge = person.age                        максимальный возраст изменится
      theOldest = person
    }
  }
  println(theOldest)
}
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> findTheOldest(people)
Person(name=Bob, age=31)
```

При наличии опыта такие циклы можно строчить довольно быстро. Но здесь слишком много кода и легко допустить ошибку: например, ошибиться в сравнении и найти минимальный элемент вместо максимального.

Kotlin предлагает способ лучше: воспользоваться библиотечными функциями, как показано далее.

Листинг 5.4. Поиск в коллекции с помощью лямбда-выражения

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.maxBy { it.age })
Person(name=Bob, age=31)
```

Найдет элемент коллекции с максимальным значением свойства age

Функцию `maxBy` можно вызвать для любой коллекции. Она принимает один аргумент: функцию, определяющую значения, которые должны сравниваться во время поиска наибольшего элемента. Код в фигурных скобках – `{ it.age }` – это лямбда-выражение, реализующее требуемую логику. В качестве аргумента оно получает элемент коллекции (доступный по ссылке `it`) и возвращает значение для сравнения. В данном примере элемент коллекции – это объект `Person`, а значение для сравнения – возраст, хранящийся в свойстве `age`.

Если лямбда-выражение делегирует свою работу функции или свойству, его можно заменить ссылкой на метод.

Листинг 5.5. Поиск с использованием ссылки на член

```
people.maxBy(Person::age)
```

Этот код делает то же самое, что и код в листинге 5.3. Подробности будут раскрыты в разделе 5.1.5.

Большинство действий, которые обычно производятся с коллекциями в Java до версии 8, лучше выразить с помощью библиотечных функций, принимающих лямбда-выражения или ссылки на члены класса. Код получится короче и проще для понимания. Чтобы помочь вам освоить этот прием, рассмотрим синтаксис лямбда-выражений.

5.1.3. Синтаксис лямбда-выражений

Как уже упоминалось, лямбда-выражение представляет небольшой фрагмент поведения, которое можно передать как значение. Его можно объявить отдельно и сохранить в переменной. Но чаще оно объявляется непосредственно при передаче в функцию. Рисунок 5.1 демонстрирует синтаксис объявления лямбда-выражения.

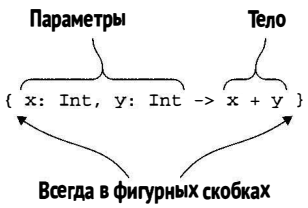


Рис. 5.1. Синтаксис лямбда-выражений

Лямбда-выражения в Kotlin всегда окружены фигурными скобками. Обратите внимание на отсутствие круглых скобок вокруг аргументов. Список аргументов отделяется от тела лямбда-выражения стрелкой.

Лямбда-выражение можно сохранить в переменной, а затем обращаться к ней как к обычной функции (вызывая с соответствующими аргументами):

```
>>> val sum = { x: Int, y: Int -> x + y }
>>> println(sum(1, 2))          <┐ Вызов лямбда-выражения,
3                               <┘ хранящегося в переменной
```

При желании лямбда-выражение можно вызывать напрямую:

```
>>> { println(42) }()
42
```

Но такой синтаксис неудобно читать, и он не имеет особого смысла (эквивалентно непосредственному выполнению тела лямбда-выражения). Если нужно заключить фрагмент кода в блок, используйте библиотечную функцию `run`, которая выполнит переданное ей лямбда-выражение:

```
>>> run { println(42) }      <— Выполняет код лямбда-выражения
42
```

В разделе 8.2 вы узнаете, почему такие выражения не требуют накладных расходов во время выполнения и почему они так же эффективны, как встроенные конструкции языка. А пока давайте вернемся к листингу 5.4, где выполняется поиск самого старого человека в списке:

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.maxBy { it.age })
Person(name=Bob, age=31)
```

Если переписать этот код без всяких синтаксических сокращений, получится следующее:

```
people.maxBy({ p: Person -> p.age })
```

Суть его понятна: фрагмент кода в фигурных скобках – это лямбда-выражение, которое передается функции в качестве аргумента. Лямбда-выражение принимает единственный аргумент типа `Person` и возвращает возраст.

Но этот код избыточен. Во-первых, в нем слишком много знаков препинания, что затрудняет чтение. Во-вторых, тип легко вывести из контекста, поэтому его можно опустить. И наконец, в данном случае не обязательно присваивать имя аргументу лямбда-выражения.

Приступим к усовершенствованию, начав с круглых скобок. Синтаксис языка Kotlin позволяет вынести лямбда-выражение за круглые скобки, если оно является последним аргументом вызываемой функции. В этом примере лямбда-выражение – единственный аргумент, поэтому его можно поместить после круглых скобок:

```
people.maxBy() { p: Person -> p.age }
```

Когда лямбда-выражение является единственным аргументом функции, также можно избавиться от пустых круглых скобок:

```
people.maxBy { p: Person -> p.age }
```

Все три синтаксические формы означают одно и то же, но последняя – самая читабельная. Повторим: если лямбда-выражение – единственный аргумент, его определенно стоит писать без круглых скобок. Если есть несколько аргументов, то можно подчеркнуть, что лямбда-выражение является аргументом, оставив его внутри круглых скобок, или же поместить его за ними – допустимы оба варианта. Если требуется передать несколько лямбда-выражений, вы сможете вынести за скобки только одно – последнее, – поэтому обычно лучше передавать их, используя обычный синтаксис.

Чтобы узнать, как эти варианты выглядят в более сложных вызовах, вернемся к функции `joinToString`, часто применявшейся в главе 3. Она также определена в стандартной библиотеке Kotlin, но версия в стандартной библиотеке принимает функцию в дополнительном параметре. Эта функция может использоваться для преобразования элемента в строку иным способом, чем при помощи вызова `toString`. Вот как можно воспользоваться ею, чтобы напечатать только имена.

Листинг 5.6. Передача лямбда-выражения в именованном аргументе

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> val names = people.joinToString(separator = " ",
...         transform = { p: Person -> p.name })
>>> println(names)
Alice Bob
```

А вот как можно переписать этот вызов, поместив лямбда-выражение за скобками.

Листинг 5.7. Передача лямбда-выражения за скобками

```
people.joinToString(" ") { p: Person -> p.name }
```

В листинге 5.6 лямбда-выражение передается в именованном аргументе, чтобы прояснить цель лямбда-выражения. Листинг 5.7 короче, но тем, кто не знаком с вызываемой функцией, будет сложнее понять, для чего используется лямбда-выражение.

Совет для пользователей IntelliJ IDEA. Преобразовать одну синтаксическую форму в другую можно с помощью действий **Move lambda expression out of parentheses** (Вынести лямбда-выражение за скобки) и **Move lambda expression into parentheses** (Внести лямбда-выражение внутрь скобок).

Ещё упростим синтаксис, избавившись от типа параметра.

Листинг 5.8. Удаление типа параметра

```
people.maxBy { p: Person -> p.age }   <— Тип параметра указан явно
people.maxBy { p -> p.age }           <— Тип параметра выводится из контекста
```

Так же, как в случае с локальными переменными, если тип параметра лямбда-выражения можно вывести, его не нужно указывать явно. В функции `maxBy` тип параметра всегда совпадает с типом элемента коллекции. Компилятор знает, что функция `maxBy` вызывается для коллекции элементов типа `Person`, поэтому может понять, что параметр лямбда-выражения тоже будет иметь тип `Person`.

Бывают случаи, когда компилятор не в состоянии вывести тип параметра лямбда-выражения, но мы не будем обсуждать их здесь. Вот простое правило: всегда опускайте тип, но если компилятор пожалуется, укажите его.

Вы можете указать типы только для некоторых аргументов, оставив для других только имена. Это может понадобиться, если компилятор не в состоянии определить один из типов или когда явное указание типа улучшает читабельность.

Последнее упрощение, которое можно сделать в этом примере, – замена имени параметра именем по умолчанию: `it`. Это имя доступно, если в контексте ожидается лямбда-выражение только с одним аргументом и его тип можно вывести автоматически.

Листинг 5.9. Использование имени параметра по умолчанию

```
people.maxBy { it.age }   <— «it» – автоматически сгенерированное имя параметра
```

Имя по умолчанию создается только тогда, когда имя аргумента не указано явно.

> **Примечание.** Соглашение об имени по умолчанию `it` помогает сократить объем кода, но им не следует злоупотреблять. В частности, в случае вложенных лямбда-выражений лучше объявлять параметры каждого лямбда-выражения явно – иначе будет трудно понять, к какому значению относится `it`. Также полезно объявлять параметры явно, если значение или тип параметра трудно понять из контекста.

Если лямбда-выражение хранится в переменной, то компилятор не имеет контекста, из которого можно вывести тип параметра. Поэтому его следует указать явно:

```
>>> val getAge = { p: Person -> p.age }
>>> people.maxBy(getAge)
```

До сих пор вы видели примеры лямбда-выражений, состоящих из одного выражения или инструкции. Но лямбда-выражения могут содержать несколько выражений! В таком случае их результат – последнее выражение:

```
>>> val sum = { x: Int, y: Int ->
...     println("Computing the sum of $x and $y...")
...     x + y
...   }
>>> println(sum(1, 2))
Computing the sum of 1 and 2...
3
```

Далее поговорим о понятии, которое часто идет бок о бок с лямбда-выражениями: захват переменных из контекста.

5.1.4. Доступ к переменным из контекста

Как известно, когда анонимный внутренний класс объявляется внутри функции, он может ссылаться на параметры и локальные переменные этой функции. В лямбда-выражениях можно делать то же самое. Если лямбда-выражение определено в функции, оно может обращаться к её параметрам и локальным переменным, объявленным перед лямбда-выражением.

Для демонстрации возьмем функцию `forEach` из стандартной библиотеки. Это одна из главных функций для работы с коллекциями, и она выполняет заданное лямбда-выражение для каждого элемента в коллекции. Функция `forEach` лаконичнее обычного цикла `for`, но это её единственное преимущество, поэтому не нужно спешить преобразовывать все циклы в лямбда-выражения.

Следующий листинг принимает список сообщений и выводит каждое с заданным префиксом.

Листинг 5.10. Использование параметров функции в лямбда-выражении

```
fun printMessagesWithPrefix(messages: Collection<String>, prefix: String) {
    messages.forEach {
      println("$prefix $it")
    }
}
```

Принимает в качестве аргумента лямбда-выражение, определяющее, что делать с каждым элементом

Обращение к параметру «prefix» из лямбда-выражения

```
>>> val errors = listOf("403 Forbidden", "404 Not Found")
>>> printMessagesWithPrefix(errors, "Error:")
Error: 403 Forbidden
Error: 404 Not Found
```

Одно важное отличие Kotlin от Java состоит в том, что Kotlin не ограничивается доступом только к финальным переменным. Вы можете изменять переменные внутри лямбда-выражений. Следующий листинг подсчитывает количество клиентских и серверных ошибок в данном наборе кодов ответа.

Листинг 5.11. Изменение локальных переменных внутри лямбда-выражения

```
fun printProblemCounts(responses: Collection<String>) {
    var clientErrors = 0
    var serverErrors = 0
    responses.forEach {
      if (it.startsWith("4")) {
        clientErrors++
      } else if (it.startsWith("5")) {
        serverErrors++
      }
    }
    println("$clientErrors client errors, $serverErrors server errors")
}
```

Объявление переменных, к которым будет обращаться лямбда-выражение

Изменение переменных внутри лямбда-выражения

```
>>> val responses = listOf("200 OK", "418 I'm a teapot",
...         "500 Internal Server Error")
>>> printProblemCounts(responses)
1 client errors, 1 server errors
```

Kotlin, в отличие от Java, позволяет обращаться к обычным, не финальным переменным (без модификатора `final`) и даже изменять их внутри лямбда-выражения. Про внешние переменные `prefix`, `clientErrors` и `serverErrors` в этих примерах, к которым обращается лямбда-выражение, говорят, что они *захватываются* лямбда-выражением.

Обратите внимание, что по умолчанию время жизни локальной переменной ограничено временем жизни функции, в которой она объявлена. Но если она захвачена лямбда-выражением, использующий её код может быть сохранен и выполнен позже. Вы можете спросить: как это работает? Когда захватывается финальная переменная, её значение сохраняется вместе с использующим её кодом лямбда-выражения. Значения обычных переменных заключаются в специальную обертку, которая позволяет менять переменную, а ссылка на обертку сохраняется вместе с лямбда-выражением.

Важно отметить, что если лямбда-выражение используется в качестве обработчика событий или просто выполняется асинхронно, модификация локальных переменных произойдет только при выполнении лямбда-вы-

ражения. Например, следующий код демонстрирует неправильный способ подсчета нажатий кнопки:

```
fun tryToCountButtonClicks(button: Button): Int {
    var clicks = 0
    button.onClick { clicks++ }
    return clicks
}
```

Эта функция всегда будет возвращать 0. Даже если обработчик `onClick` будет изменять значение переменной `clicks`, вы не увидите изменений, поскольку обработчик `onClick` будет вызываться после выхода из функции. Для правильной работы количество нажатий необходимо сохранять не в локальную переменную, а в месте, доступном за пределами функции, – например, в свойстве класса.

Мы обсудили синтаксис объявления лямбда-выражений и как они захватывают переменные. Теперь поговорим о ссылках на члены класса, с помощью которых легко можно передавать ссылки на существующие функции.

Захват изменяемых переменных: детали реализации

Java позволяет захватывать только финальные переменные. Если вы хотите захватить изменяемую переменную, то можете использовать один из следующих приемов: либо создать экземпляр класса-обертки, хранящего ссылку, которая может быть изменена. Если вы примените эту технику в Kotlin, код будет выглядеть следующим образом:

```
class Ref<T>(var value: T)
>>> val counter = Ref(0)
>>> val inc = { counter.value++ }
```

Класс, имитирующий захват изменяемой переменной

Формально захватывается неизменяемая переменная, но реальное значение сохраняется в поле и может быть изменено

В реальном коде такие обертки вам не понадобятся. Вместо этого можно менять значение переменной напрямую.

```
var counter = 0
val inc = { counter++ }
```

Как это работает? Собственно, первый пример объясняет, что происходит при работе второго примера. Всякий раз при захвате финальной переменной (`val`) её значение копируется, точно как в Java. При захвате изменяемой переменной (`var`) её значение сохраняется как экземпляр класса `Ref`. Переменная `Ref` – финальная и может быть захвачена, в то время как фактическое значение хранится в поле и может быть изменено внутри лямбда-выражения.

5.1.5. Ссылки на члены класса

Теперь вы знаете, как лямбда-выражения позволяют передать блок кода в вызов функции. Но что, если код, который нужно передать, уже определен как функция? Конечно, можно передать лямбда-выражение, вызывающее эту функцию, но это избыточное решение. Можно ли передать функцию напрямую?

В Kotlin, как и в Java 8, это можно сделать, преобразовав функцию в значение с помощью оператора `::`.

```
val getAge = Person::age
```

Рис. 5.2. Синтаксис ссылки на член класса

Это выражение называется *ссылкой на член класса* (member reference) и обеспечивает короткий синтаксис создания значения функции, вызывающего ровно один метод или обращающегося к свойству. Двойное двоеточие отделяет имя класса от имени члена класса, на который нужно сослаться (метод или свойство), как показано на рис. 5.2.

Это более краткая форма записи следующего лямбда-выражения:

```
val getAge = { person: Person -> person.age }
```

Обратите внимание: независимо от того, на что указывает ссылка – на функцию или свойство, – вы не должны ставить круглые скобки после имени члена класса при создании ссылки.

Ссылка на член класса – того же типа, что и лямбда-выражение, вызывающее эту функцию, поэтому их можно взаимно заменять:

```
people.maxBy(Person::age)
```

Также можно создать ссылку на функцию верхнего уровня (и не являющуюся членом класса):

```
fun salute() = println("Salute!")
>>> run(::salute)                    <— Ссылка на функцию верхнего уровня
Salute!
```

В этом случае имя класса не указывается и ссылка начинается с `::`. Ссылка на функцию `::salute` передается как аргумент библиотечной функции `run`, которая вызывает соответствующую функцию.

Иногда удобно использовать ссылку на функцию вместо лямбда-выражения, делегирующего свою работу функции, принимающей несколько параметров:

```
val action = { person: Person, message: String ->   <— Это лямбда-выражение делегирует
    sendEmail(person, message)                          работу функции sendEmail
}
val nextAction = ::sendEmail   <— Вместо него можно использовать
                                  ссылку на функцию
```

Вы можете сохранить или отложить операцию создания экземпляра класса с помощью *ссылки на конструктор*. Чтобы сформировать ссылку на конструктор, нужно указать имя класса после двойного двоеточия:

```
data class Person(val name: String, val age: Int)

>>> val createPerson = ::Person
>>> val p = createPerson("Alice", 29)
>>> println(p)
Person(name=Alice, age=29)
```

Операция создания экземпляра Person сохраняется в переменную

Обратите внимание, что ссылку можно получить и на функцию-расширение:

```
fun Person.isAdult() = age >= 21
val predicate = Person::isAdult
```

Хотя функция `isAdult` не является членом класса `Person`, её можно вызывать через ссылку, точно как при обращении через метод экземпляра: `person.isAdult`.

Связанные ссылки

В Kotlin 1.0 всегда требуется передавать экземпляр класса при обращении к его методу или свойству по ссылке. В Kotlin 1.1 планируется добавить поддержку связанных ссылок, позволяющих использовать специальный синтаксис для захвата ссылки на метод конкретного экземпляра класса:

```
>>> val p = Person("Dmitry", 34)
>>> val personsAgeFunction = Person::age
>>> println(personsAgeFunction(p))
34
>>> val dmitrysAgeFunction = p::age
>>> println(dmitrysAgeFunction())
34
```

Связанная ссылка, доступная в Kotlin 1.1

Обратите внимание, что функция `personsAgeFunction` принимает один аргумент (и возвращает возраст конкретного человека), тогда как `dmitrysAgeFunction` – это функция без аргументов (возвращает возраст конкретного человека). До Kotlin 1.1 вместо использования связанной ссылки на метод `p::age` нужно было написать лямбда-выражение `{ p.age }`.

В следующем разделе мы познакомимся со множеством библиотечных функций, которые отлично работают с лямбда-выражениями, а также ссылками на члены класса.

5.2. Функциональный API для работы с коллекциями

Функциональный стиль дает много преимуществ при работе с коллекциями. В стандартной библиотеке есть функции для решения большинства задач, позволяющие упростить ваш код. В этом разделе мы познакомимся с некоторыми функциями для работы с коллекциями из стандартной библиотеки Kotlin. Начнем с самых основных, таких как `filter` и `map`, и узнаем, на каких идеях они основаны. Мы также расскажем о других полезных функциях и обьясним, как не злоупотреблять ими и писать ясный и понятный код.

Обратите внимание, что ни одна из этих функций не была придумана разработчиками языка Kotlin. Эти или похожие функции доступны во всех языках с поддержкой лямбда-выражений, включая C#, Groovy и Scala. Если вы уже знакомы с этими понятиями, можете быстро просмотреть следующие примеры и пропустить объяснения.

5.2.1. Основы: filter и map

Функции `filter` и `map` – основа работы с коллекциями. С их помощью можно выразить многие операции по сбору данных.

Для каждой функции мы покажем один пример с цифрами и один с использованием знакомого вам класса `Person`:

```
data class Person(val name: String, val age: Int)
```

Функция `filter` выполняет обход коллекции, отбирая только те элементы, для которых лямбда-выражение вернет `true`:

```
>>> val list = listOf(1, 2, 3, 4)
>>> println(list.filter { it % 2 == 0 })
[2, 4]
```


В результате получится новая коллекция, содержащая только элементы, удовлетворяющие предикату, как показано на рис. 5.3.

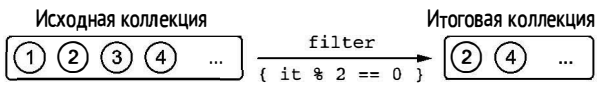


Рис. 5.3. Функция filter отбирает только элементы, удовлетворяющие предикату

С помощью `filter` можно найти в списке всех людей старше 30:

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.filter { it.age > 30 })
[Person(name=Bob, age=31)]
```

Функция `filter` сможет удалить из коллекции ненужные элементы, но не сможет изменить их. Для преобразования элементов вам понадобится функция `map`.

Функция `map` применяет заданную функцию к каждому элементу коллекции, объединяя результаты в новую коллекцию. Например, вот как можно преобразовать список чисел в список их квадратов:

```
>>> val list = listOf(1, 2, 3, 4)
>>> println(list.map { it * it })
[1, 4, 9, 16]
```

В результате получится новая коллекция, содержащая такое же количество элементов, но каждый элемент будет преобразован согласно заданному предикату (см. рис. 5.4).

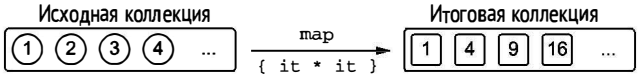


Рис. 5.4. Функция `map` применяет лямбда-выражение к каждому элементу коллекции

Чтобы просто вывести список имен, можно преобразовать исходный список, используя функцию `map`:

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.map { it.name })
[Alice, Bob]
```

Обратите внимание, что этот пример можно элегантно переписать, используя ссылку на член класса:

```
people.map(Person::name)
```

Вызовы функций можно объединять в цепочки. Например, давайте выведем имена всех, кто старше 30:

```
>>> people.filter { it.age > 30 }.map(Person::name)
[Bob]
```

Теперь допустим, что вам нужны имена самых взрослых людей в группе. Как это сделать? Можно найти максимальный возраст в группе и вернуть список всех с тем же возрастом. Такой код легко написать с помощью лямбда-выражений:

```
people.filter { it.age == people.maxBy(Person::age).age }
```

Но обратите внимание: этот код повторяет процесс поиска максимального возраста для каждого человека – следовательно, если в коллекции хранится список из 100 человек, поиск максимального возраста будет выполнен 100 раз!

Следующее решение не имеет этого недостатка и рассчитывает максимальный возраст только один раз:

```
val maxAge = people.maxBy(Person::age).age
people.filter { it.age == maxAge }
```

Не повторяйте расчетов без необходимости! Код с лямбда-выражением, который кажется простым, иногда может скрывать сложность используемых им операций. Всегда думайте о том, что происходит в коде, который вы пишете.

К словарям также можно применить функции отбора и преобразования:

```
>>> val numbers = mapOf(0 to "zero", 1 to "one")
>>> println(numbers.mapValues { it.value.toUpperCase() })
{0=ZERO, 1=ONE}
```

Обратите внимание: существуют отдельные функции для обработки ключей и значений. Функции `filterKeys` и `mapKeys` отбирают и преобразуют ключи словаря соответственно, тогда как `filterValues` и `mapValues` отбирают и преобразуют значения.

5.2.2. Применение предикатов к коллекциям: функции «all», «any», «count» и «find»

Ещё одна распространенная задача – проверка всех элементов коллекции на соответствие определенному условию (или, например, проверка наличия хотя бы одного такого элемента). В Kotlin эта задача решается с помощью функций `all` и `any`. Функция `count` проверяет, сколько элементов удовлетворяет предикату, а функция `find` возвращает первый подходящий элемент.

Чтобы продемонстрировать работу этих функций, определим предикат `canBeInClub27`, возвращающий `true`, если возраст человека 27 лет или меньше:

```
val canBeInClub27 = { p: Person -> p.age <= 27 }
```

Если вас интересует, все ли элементы удовлетворяют этому предикату, воспользуйтесь функцией `all`:

```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.all(canBeInClub27))
false
```

Когда нужно найти хотя бы один подходящий элемент, используйте функцию `any`.

```
>>> println(people.any(canBeInClub27))
true
```

Обратите внимание, что выражение `!all` («не все») с условием можно заменить выражением `any` с противоположным условием, и наоборот. Чтобы сделать код более понятным, выбирайте функцию, не требующую знака отрицания перед ней:

```
>>> val list = listOf(1, 2, 3)
>>> println(!list.all { it == 3 })
true
>>> println(list.any { it != 3 })
true
```

Символ отрицания ! трудно заметить, поэтому в данном случае лучше использовать «any»

Условие в этом аргументе поменялось на противоположное

Первая проверка гарантирует, что не все элементы равны 3. То есть она проверяет наличие хотя бы одного элемента, не равного 3, – именно это проверяется с помощью выражения `any` во второй строке.

Если требуется узнать, сколько элементов удовлетворяет предикату, используйте `count`:

```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.count(canBeInClub27))
1
```

Выбор правильной функции: count или size

Можно легко забыть о методе `count` и реализовать подсчет с помощью фильтрации коллекции и получения её размера:

```
>>> println(people.filter(canBeInClub27).size)
1
```

Но в этом случае для хранения всех элементов, удовлетворяющих предикату, будет создана промежуточная коллекция. С другой стороны, метод `count` только подсчитывает количество подходящих элементов, а не сами элементы, поэтому он более эффективен.

Всегда пытайтесь подобрать операцию, наиболее подходящую вашим потребностям.

Чтобы найти элемент, удовлетворяющий предикату, используйте функцию `find`:

```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.find(canBeInClub27))
Person(name=Alice, age=27)
```

Она возвращает первый найденный элемент, если их несколько, или `null`, если ни один не удовлетворяет предикату. Синоним функции `find` – `firstOrNull`, которую также можно использовать, если она лучше выражает вашу идею.

5.2.3. Группировка значений в списке с функцией groupBy

Представьте, что вам нужно разделить элементы коллекции на разные группы по некоторому критерию – например, разбить список людей на группы по возрасту. Было бы удобно передать этот критерий непосредственно в качестве параметра. Функция groupBy может сделать это для вас:

```
>>> val people = listOf(Person("Alice", 31),
...     Person("Bob", 29), Person("Carol", 31))
>>> println(people.groupBy { it.age })
```

Результатом этой операции будет словарь с ключами, определяющими признак для группировки (в данном случае возраст), – см. рис. 5.5.

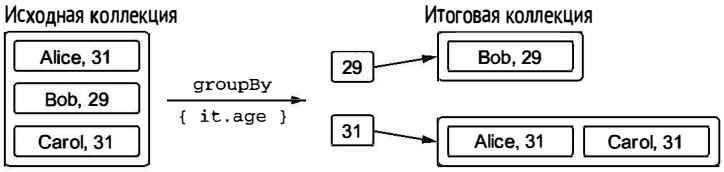


Рис. 5.5. Результат применения функции groupBy

Данный пример даст следующий результат:

```
{29=[Person(name=Bob, age=29)],
31=[Person(name=Alice, age=31), Person(name=Carol, age=31)]}
```

Каждая группа сохраняется в виде списка, поэтому результат будет иметь тип Map<Int, List<Person>>. Вы можете изменять этот словарь, используя функции mapKeys и mapValues.

В качестве другого примера посмотрим, как группировать строки по первому символу, используя ссылку на метод:

```
>>> val list = listOf("a", "ab", "b")
>>> println(list.groupBy(String::first))
{a=[a, ab], b=[b]}
```

Обратите внимание, что функция first является не членом класса String, а функцией-расширением. Тем не менее к ней можно обращаться через ссылку на метод.

5.2.4. Обработка элементов вложенных коллекций: функции flatMap и flatten

Теперь оставим людей в покое и переключимся на книги. Предположим, у нас есть хранилище книг, представленных классом Book:

```
class Book(val title: String, val authors: List<String>)
```

Каждая книга написана одним или несколькими авторами. Вот как можно найти множество всех авторов в библиотеке:

```
books.flatMap { it.authors }.toSet()      <— Множество всех авторов книг в коллекции «books»
```

Функция `flatMap` сначала преобразует (или *отображает* – *map*) каждый элемент в коллекцию, согласно функции, переданной в аргументе, а затем собирает (или *уплощает* – *flattens*) несколько списков в один. Пример со строками хорошо иллюстрирует эту идею (см. рис. 5.6):

```
>>> val strings = listOf("abc", "def")
>>> println(strings.flatMap { it.toList() })
[a, b, c, d, e, f]
```

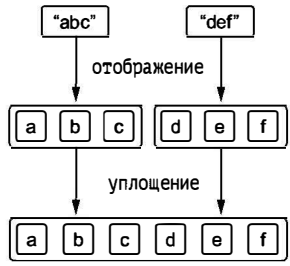


Рис. 5.6. Результат применения функции flatMap

Если применить функцию `toList` к строке, она превратит её в список символов. Если использовать функцию `map` вместе с функцией `toList`, получится список списков символов, как во втором ряду на рис. 5.6. Функция `flatMap` делает следующий шаг и возвращает список из всех элементов.

Вернемся к авторам:

```
>>> val books = listOf(Book("Thursday Next", listOf("Jasper Fforde")),
...                    Book("Mort", listOf("Terry Pratchett")),
...                    Book("Good Omens", listOf("Terry Pratchett",
...                                              "Neil Gaiman")))
>>> println(books.flatMap { it.authors }.toSet())
[Jasper Fforde, Terry Pratchett, Neil Gaiman]
```

Каждая книга может быть написана несколькими авторами, список которых хранится в свойстве `book.authors`. Функция `flatMap` объединяет авторов всех книг в один плоский список. Функция `toSet` удаляет дубликаты из получившейся коллекции: так, в этом примере Терри Пратчетт (Terry Pratchett) появится в выводе программы только один раз.

Вспомните о функции `flatMap`, когда нужно будет объединить коллекцию коллекций элементов. Но если потребуется просто плоская коллекция, без преобразований, используйте функцию `flatten`: `listOfLists.flatten()`.

Мы показали только некоторые функции для работы с коллекциями, имеющиеся в стандартной библиотеке Kotlin, – кроме них, есть множество других. Мы не будем рассматривать их все из-за нехватки места, а также потому, что показывать длинный список функций скучно. Вообще, при написании кода, работающего с коллекциями, мы советуем думать о том, как действие может быть выражено в виде общего преобразования, и искать библиотечную функцию, выполняющую такое преобразование. Вполне вероятно, что вы найдете такую функцию и сможете использовать её для решения вашей проблемы гораздо быстрее, чем если будете писать её вручную.

Теперь давайте внимательно исследуем производительность кода, объединяющего несколько операций над коллекциями. В следующем разделе вы увидите различные способы реализации подобных операций.

5.3. Отложенные операции над коллекциями: последовательности

В предыдущем разделе вы видели несколько примеров составления цепочек из вызовов функций для работы с коллекциями – таких как `map` и `filter`. Эти функции *немедленно* создают промежуточные коллекции, т. е. результат, полученный на каждом промежуточном шаге, сразу же сохраняется во временном списке. *Последовательности* (sequences) дают альтернативный способ реализации таких вычислений, позволяющий избежать создания временных промежуточных объектов.

Рассмотрим пример.

```
people.map(Person::name).filter { it.startsWith("A") }
```

Справочник по стандартной библиотеке Kotlin говорит, что `filter` и `map` возвращают список. Это значит, что данная цепочка вызовов создаст два списка: один – для хранения результатов функции `filter` и другой – для результатов функции `map`. Это не проблема, если в исходном списке всего пара элементов, но в случае со списком из миллиона элементов это может существенно снизить эффективность операции.

Для повышения эффективности нужно реализовать операцию с применением *последовательностей* вместо коллекций:

```
people.asSequence()
  .map(Person::name)
  .filter { it.startsWith("A") }
  .toList()
```

Преобразует исходную коллекцию в последовательность

Последовательность реализует тот же API, что и коллекции

Преобразование получившейся последовательности обратно в список

Эта операция вернет тот же результат, что и предыдущий пример: список имен, начинающихся с буквы *A*. Но второй пример не создает промежуточных коллекций для хранения элементов, а следовательно, для большого количества элементов производительность будет заметно лучше.

Точка входа для выполнения отложенных операций в Kotlin – интерфейс `Sequence`. Он представляет собой простую последовательность элементов, которые могут перечисляться один за другим. Интерфейс `Sequence` определяет только один метод – `iterator`, который используется для получения значений последовательности.

Особенность интерфейса `Sequence` – способ реализации операций. Элементы последовательности вычисляются «лениво». Благодаря этому по-

следовательности можно использовать для эффективного выполнения цепочек операций над элементами, не создавая коллекций для хранения промежуточных результатов.

Любую коллекцию можно преобразовать в последовательность, вызвав функцию-расширение `asSequence`. Обратное преобразование выполняется вызовом функции `toList`.

Зачем нужно преобразовывать последовательность обратно в коллекцию? Не проще ли всегда использовать последовательности вместо коллекций, раз они настолько лучше? Иногда они действительно удобнее. Если нужно выполнить только обход элементов, можно использовать последовательность напрямую. Но если потребуется использовать другие методы (например, доступ к элементам по индексу), последовательность нужно преобразовать в список.

> **Примечание.** Как правило, последовательности применяются всякий раз, когда требуется выполнить цепочку операций над *большой* коллекцией. В разделе 8.2 мы обсудим, почему немедленные операции над обычными коллекциями так эффективны в Kotlin, несмотря на создание промежуточных коллекций. Но когда коллекция содержит большое количество элементов и промежуточная перестановка элементов требует много затрат, предпочтительнее использовать отложенные вычисления.

Поскольку операции над последовательностями выполняются в отложенной манере («лениво»), для их фактического выполнения нужно либо перебрать элементы последовательности, либо преобразовать её в коллекцию. Причины этого – в следующем разделе.

5.3.1. Выполнение операций над последовательностями: промежуточная и завершающая операции

Операции над последовательностями делятся на две категории: промежуточные и завершающие. *Промежуточная операция* возвращает другую последовательность, которая знает, как преобразовать элементы исходной последовательности. *Завершающая операция* возвращает результат, который может быть коллекцией, элементом, числом или любым другим объектом, полученным в ходе преобразований исходной коллекции (см. рис. 5.7).

Рис. 5.7. Промежуточные и завершающая операции над последовательностями

Выполнение промежуточных операций всегда откладывается. Взгляните на пример, в котором отсутствует завершающая операция:

```
>>> listOf(1, 2, 3, 4).asSequence()
...                    .map { print("map($it) "); it * it }
...                    .filter { print("filter($it) "); it % 2 == 0 }
```

Этот код ничего не выведет в консоль. Это значит, что преобразования `map` и `filter` отложены и будут применены, только когда потребуется вернуть результат (т. е. во время выполнения завершающей операции):

```
>>> listOf(1, 2, 3, 4).asSequence()
...                    .map { print("map($it) "); it * it }
...                    .filter { print("filter($it) "); it % 2 == 0 }
...                    .toList()
map(1) filter(1) map(2) filter(4) map(3) filter(9) map(4) filter(16)
```

Завершающая операция заставляет выполниться все отложенные вычисления.

Еще одна важная деталь, которую нужно отметить в этом примере, – порядок выполнения вычислений. При реализации «в лоб» к каждому элементу сначала применяется функция `map`, а затем вызывается функция `filter` для каждого элемента получившейся последовательности. Так функции `map` и `filter` работают с коллекциями, но не с последовательностями. Для последовательности все операции применяются к каждому элементу поочередно: сначала обрабатывается первый элемент (преобразуется, а затем фильтруется), затем второй и т. д.

Такой подход означает, что некоторые элементы могут вовсе не подвергнуться преобразованию, если результат будет вычислен прежде, чем до них дойдет очередь. Давайте посмотрим пример с операциями `find` и `map`. Сначала вычислим квадрат числа, а затем найдем первый элемент больше 3:

```
>>> println(listOf(1, 2, 3, 4).asSequence()
                              .map { it * it }.find { it > 3 })
4
```

Если те же операции применить к коллекции вместо последовательности, сначала выполнится функция `map`, которая преобразует каждый элемент исходной коллекции в квадрат. А затем, на втором шаге, в промежуточной коллекции будет найден элемент, удовлетворяющий предикату. Отложенные вычисления позволяют пропустить обработку некоторых элементов. Рисунок 5.8 иллюстрирует разницу между немедленным (при работе с коллекциями) и отложенным (при работе с последовательностями) выполнениями этих операций.

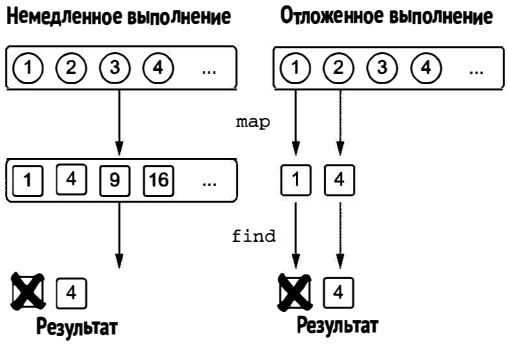


Рис. 5.8. Немедленный способ выполнит каждую операцию для всей коллекции; отложенный будет обрабатывать элементы один за другим

В первом случае, когда при работе с коллекциями исходный список превращается в другой список, преобразование применяется к каждому элементу, в том числе к 3 и 4. После этого выполняется поиск первого элемента, удовлетворяющего предикату: квадрата числа 2.

Во втором случае вызов find обрабатывает элементы по одному. Из исходной последовательности извлекается число, преобразуется с помощью map, а затем проверяется на соответствие предикату в find. По достижении числа 2 обнаружится, что его квадрат больше 3 и оно будет возвращено как результат операции find. Программе не придется проверять 3 и 4, поскольку результат будет до того, как до них дойдет очередь.

Порядок выполнения операций над коллекцией также влияет на производительность. Представьте, что у нас есть коллекция людей, и мы хотим вывести их имена, но только если они меньше определенной длины. Для этого нам потребуется отобразить каждый объект `Person` в имя человека, а затем выбрать достаточно короткие имена. В этом случае операции `map` и `filter` можно применить в любом порядке. Оба подхода дают одинаковый результат, но отличаются количеством преобразований (см. рис. 5.9).

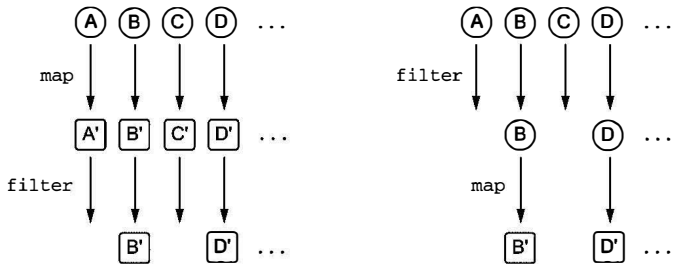


Рис. 5.9. Применение функции filter первой помогает уменьшить число необходимых преобразований

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31),
... Person("Charles", 31), Person("Dan", 21))
>>> println(people.asSequence().map(Person::name)          <— Сначала «map», затем «filter»
```

```
... .filter { it.length < 4 }.toList())
[Bob, Dan]
>>> println(people.asSequence().filter { it.name.length < 4 }
... .map(Person::name).toList())          <— «map» выполнится после «filter»
[Bob, Dan]
```

Если map выполнится первой, будет преобразован каждый элемент. Если же сначала применить filter, неподходящие элементы отфильтруются раньше и не будут преобразованы.

> **Потоки и последовательности**
>
> Знакомые с потоками (streams) в Java 8 без труда узнают их в последовательностях. Kotlin предлагает собственную версию этой идеи, потому что потоки Java 8 недоступны на платформах, основанных на старых версиях Java (таких как Android). Если вы ориентируетесь на Java 8, потоки дадут вам одно большое преимущество, которое в настоящее время не реализовано для коллекций и последовательностей в Kotlin, – возможность запуска потоковой операции (например, map или filter) параллельно на нескольких процессорах. Вы можете выбирать между потоками и последовательностями в зависимости от версии Java и ваших особых требований.

5.3.2. Создание последовательностей

В предыдущих примерах для создания последовательностей использовался один и тот же способ: вызывалась функция asSequence() коллекции. Однако последовательность также можно создать вызовом функции generateSequence. Она вычисляет следующий элемент последовательности на основании предыдущего. Например, вот как можно использовать generateSequence для подсчета суммы всех натуральных чисел до 100.

Листинг 5.12. Создание и использование последовательности натуральных чисел

```
>>> val naturalNumbers = generateSequence(0) { it + 1 }
>>> val numbersTo100 = naturalNumbers.takeWhile { it <= 100 }
>>> println(numbersTo100.sum())   <— Все отложенные операции выполнятся
5050                                  при обращении к «sum»
```

Обратите внимание, что naturalNumbers и numbersTo100 в данном примере – последовательности с отложенным выполнением операций. Реальные числа в этих последовательностях не будут вычислены до вызова завершающей операции (в данном случае sum).

Другой распространенный вариант использования – это последовательность родителей. Если у элемента есть родитель того же типа (например, человек или Java-файл), вас могут заинтересовать свойства всех его пред-

ков в последовательности. Следующий пример проверяет, находится ли файл в скрытом каталоге, создав последовательность родительских каталогов и проверив соответствующий атрибут у каждого каталога.

Листинг 5.13. Создание и применение последовательности родительских каталогов

```
fun File.isInsideHiddenDirectory() =
        generateSequence(this) { it.parentFile }.any { it.isHidden }

>>> val file = File("/Users/svtk/.HiddenDir/a.txt")
>>> println(file.isInsideHiddenDirectory())
true
```

Еще раз напомним, что вы создаете последовательность, предоставляя первый элемент и способ получения каждого последующего элемента. Заменив `any` на `find`, вы получите желаемый каталог. Обратите внимание, что последовательности позволяют остановить обход родителей, как только нужный каталог будет найден.

Мы подробно обсудили распространенные случаи применения лямбда-выражений для упрощения манипуляций с коллекциями. Теперь затронем не менее важную тему: использование лямбда-выражений с существующим Java API.

5.4. Использование функциональных интерфейсов Java

Использовать лямбда-выражения с библиотеками Kotlin приятно, но большая часть API, с которыми вам доведется работать, написана на Java, а не на Kotlin. Замечательно, что лямбда-выражения Kotlin полностью совместимы с Java API. В этом разделе вы точно узнаете, как это работает.

В начале главы вы видели пример передачи лямбда-выражения в Java-метод:

```
button.setOnClickListener { /* actions on click */ }
```

Передача лямбда-выражения в качестве аргумента

Класс `Button` позволяет подключить новый обработчик с помощью метода `setOnClickListener`, принимающего аргумент типа `OnClickListener`:

```
/* Java */
public class Button {
    public void setOnClickListener(OnClickListener l) { ... }
}
```

В интерфейсе `OnClickListener` объявлен только один метод – `onClick`:

```
/* Java */
public interface OnClickListener {
```

```
    void onClick(View v);
}
```

В Java (до версии 8) вы должны создать новый экземпляр анонимного класса и передать его методу `setOnClickListener`:

```
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
      ...
    }
}
```

В Kotlin вместо этого можно передать лямбда-выражение:

```
button.setOnClickListener { view -> ... }
```

Лямбда-выражение, реализующее интерфейс `OnClickListener`, принимает один параметр типа `View`, как и метод `onClick`. Это соответствие показано на рис. 5.10.

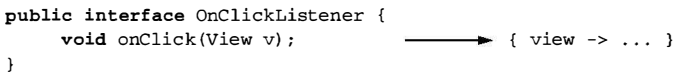


Рис. 5.10. Параметры лямбда-выражения соответствуют параметрам метода

Это возможно потому, что интерфейс `OnClickListener` имеет только один абстрактный метод. Такие интерфейсы называются *функциональными интерфейсами*, или SAM-интерфейсами (*Single Abstract Method* – с единственным абстрактным методом). В Java API полным-полно функциональных интерфейсов (таких как `Runnable` и `Callable`), и методов для работы с ними. Kotlin позволяет использовать лямбда-выражения в вызовах методов Java, принимающих в качестве параметров функциональные интерфейсы, что гарантирует чистоту и идиоматичность кода на Kotlin.

> **Примечание.** В отличие от Java, в Kotlin есть настоящие типы функций. Поэтому функции в Kotlin, принимающие лямбда-выражения, должны использовать в качестве параметров типы функций, а не типы функциональных интерфейсов. Kotlin не поддерживает автоматического преобразования лямбда-выражений в объекты, реализующие интерфейсы. Мы обсудим использование типов функций в объявлениях функций в разделе 8.1.

Давайте посмотрим, что происходит, когда методу, ожидающему аргумента с типом функционального интерфейса, передается лямбда-выражение.

5.4.1. Передача лямбда-выражения в Java-метод

Вы можете передать лямбда-выражение в любой метод Java, принимающий функциональный интерфейс. К примеру, рассмотрим следующий метод, принимающий параметр типа `Runnable`:

```
/* Java */
void postponeComputation(int delay, Runnable computation);
```

В Kotlin можно вызвать этот метод, передав в аргументе лямбда-выражение. Компилятор автоматически преобразует его в экземпляр `Runnable`:

```
postponeComputation(1000) { println(42) }
```

Обратите внимание: говоря «экземпляр `Runnable`», мы имеем в виду «экземпляр анонимного класса, реализующего интерфейс `Runnable`». Компилятор создаст его за вас, а в качестве тела единственного абстрактного метода (`run` в данном случае) использует лямбда-выражение.

Тот же эффект можно получить, создав анонимный объект, явно реализующий интерфейс `Runnable`:

```
postponeComputation(1000, object : Runnable {
    override fun run() {
      println(42)
    }
})
```

Передача объекта-выражения в качестве реализации функционального интерфейса

Но есть одно отличие: при явном объявлении объекта в каждом вызове создается новый экземпляр. С лямбда-выражениями ситуация иная: если лямбда-выражение не захватывает переменных из функции, где оно определено, соответствующий экземпляр анонимного класса повторно используется между вызовами:

```
postponeComputation(1000) { println(42) }
```

В программе будет создан только один экземпляр интерфейса Runnable

Ниже приводится эквивалентная реализация с явным объявлением объекта, которая сохраняет экземпляр `Runnable` в переменную и использует её в каждом вызове:

```
val runnable = Runnable { println(42) }
fun handleComputation() {
    postponeComputation(1000, runnable)
}
```

Компилируется в глобальную переменную; в программе существует только один экземпляр

В каждый вызов метода handleComputation будет передаваться один и тот же экземпляр

Когда лямбда-выражение захватывает переменные из окружающего контекста, становится невозможно повторно использовать один и тот же экземпляр в каждом вызове. В этом случае компилятор создает новый объект для каждого вызова, сохраняя в нем значения захваченных переменных. Например, каждый вызов следующей функции использует новый экземпляр `Runnable`, хранящий значение `id` в своем поле:

```
fun handleComputation(id: String) {
    postponeComputation(1000) { println(id) }
}
```

Лямбда-выражение захватывает переменную «id»

Для каждого вызова handleComputation создается новый экземпляр Runnable

Подробности реализации лямбда-выражений

В Kotlin 1.0 каждое лямбда-выражение компилируется в анонимный класс, если только не является встраиваемым. Поддержку генерации байт-кода Java 8 планируется добавить в более поздних версиях Kotlin. Как только она будет реализована, компилятору не придется создавать отдельные файлы *.class* для каждого лямбда-выражения.

Если лямбда-выражение захватывает переменные, в анонимном классе будет предусмотрено поле для каждой захваченной переменной, и новый экземпляр этого класса будет создаваться при каждом вызове. В противном случае будет создан один экземпляр. Название класса формируется путем добавления суффикса к имени функции, в которой объявлено лямбда-выражение: например, `HandleComputation$1`.

Вот что вы увидите, декомпилировав код предыдущего лямбда-выражения:

```
class HandleComputation$1(val id: String) : Runnable {
    override fun run() {
       println(id)
    }
}
fun handleComputation(id: String) {
   postponeComputation(1000, HandleComputation$1(id))
}
```

За кулисами вместо лямбда-выражения будет создан экземпляр специального класса

Как видите, компилятор генерирует поле и параметр конструктора для каждой захваченной переменной.

Обратите внимание, что наше обсуждение создания анонимного класса и его экземпляра для лямбда-выражения актуально лишь для Java-методов, принимающих функциональные интерфейсы, и неприменимо для работы с коллекциями с использованием методов-расширений Kotlin. Если передать лямбда-выражение в функцию на Kotlin, отмеченную ключевым словом `inline`, анонимный класс не будет создан. Большинство библиотечных функций отмечено ключевым словом `inline`. Подробности реализации этого ключевого слова ищите в разделе 8.2.

Как видите, в большинстве случаев преобразование лямбда-выражения в экземпляр функционального интерфейса происходит автоматически, без каких-либо усилий с вашей стороны. Но иногда необходимо явно выполнить преобразование. Давайте узнаем, как это сделать.

5.4.2. SAM-конструкторы: явное преобразование лямбда-выражений в функциональные интерфейсы

SAM-конструктор – это функция, сгенерированная компилятором, которая позволяет явно выполнить преобразование лямбда-выражения в

экземпляр функционального интерфейса. Его можно использовать в контекстах, где компилятор не применяет преобразования автоматически. Например, если есть метод, который возвращает экземпляр функционального интерфейса, вы не сможете вернуть лямбда-выражение напрямую: его нужно завернуть в вызов SAM-конструктора. Вот простая демонстрация.

Листинг 5.14. Применение SAM-конструктора к возвращаемому значению

```
fun createAllDoneRunnable(): Runnable {
    return Runnable { println("All done!") }
}

>>> createAllDoneRunnable().run()
All done!
```

Имя SAM-конструктора совпадает с именем соответствующего функционального интерфейса. SAM-конструктор принимает один аргумент – лямбда-выражение, которое будет использовано как тело единственного абстрактного метода в функциональном интерфейсе, – и возвращает экземпляр класса, реализующего данный интерфейс.

Помимо создания возвращаемых значений, SAM-конструкторы используются, чтобы сохранить в переменной экземпляр функционального интерфейса, созданный из лямбда-выражения. Предположим, вы хотите использовать один обработчик событий для нескольких кнопок, как в следующем листинге (в Android-приложении этот код может быть частью метода `Activity.onCreate`).

Листинг 5.15. Использование SAM-конструктора для повторного использования обработчика событий

```
val listener = OnClickListener { view ->
    val text = when (view.id) {
      R.id.button1 -> "First button"
      R.id.button2 -> "Second button"
      else -> "Unknown button"
    }
    toast(text)
}
button1.setOnClickListener(listener)
button2.setOnClickListener(listener)
```

Экземпляр в переменной `listener` проверяет, какая кнопка породила событие, и ведет себя соответственно. Можно определить обработчик событий, используя объявление объекта, реализующего интерфейс `OnClickListener`, но SAM-конструктор – более лаконичный вариант.

Лямбда-выражения и добавление/удаление обработчиков событий

Обратите внимание, что в лямбда-выражении, в отличие от анонимного объекта, нет ссылки `this`: не существует способа сослаться на анонимный экземпляр класса, в который преобразуется лямбда-выражение. С точки зрения компилятора лямбда-выражение – это блок кода, а не объект, и вы не можете относиться к нему как к объекту. Ссылка `this` в лямбда-выражении относится к окружающему классу.

Если обработчик события должен отменить свою подписку на события во время обработки, вы не сможете воспользоваться для этого лямбда-выражением. Вместо этого используйте для реализации обработчика анонимный объект. В анонимном объекте ключевое слово `this` ссылается на экземпляр этого объекта, и вы можете передать его API для удаления обработчика.

Также, хотя SAM-преобразование в вызовах методов, как правило, происходит автоматически, иногда компилятор не может выбрать правильную перегруженную версию при передаче лямбда-выражения в аргументе перегруженному методу. В таких случаях явное применение SAM-конструктора – хороший способ устранения ошибки компиляции.

Чтобы завершить обсуждение синтаксиса лямбда-выражений и их применения, рассмотрим лямбда-выражения с получателями и то, как они используются для определения удобных библиотечных функций, которые выглядят как встроенные конструкции языка.

5.5. Лямбда-выражения с получателями: функции «with» и «apply»

Этот раздел посвящен `with` и `apply` – очень удобным функциям из стандартной библиотеки Kotlin, которым можно найти массу применений, даже не зная, как они объявлены. Позже, в разделе 11.2.1, вы увидите, как объявлять аналогичные функции для собственных нужд. А пока информация в этом разделе поможет вам познакомиться с уникальной особенностью лямбда-выражений в Kotlin, недоступной в Java: возможностью вызова методов другого объекта в теле лямбда-выражений без дополнительных квалификаторов. Такие конструкции называются *лямбда-выражениями с получателями*. Для начала познакомимся с функцией `with`, которая использует лямбда-выражение с получателем.

5.5.1. Функция «with»

Во многих языках есть специальные инструкции, помогающие выполнить несколько операций над одним и тем же объектом, не повторяя его имени. В Kotlin есть похожая возможность, но она реализована как библиотечная функция `with`, а не как специальная языковая конструкция.

Чтобы понять, где это может пригодиться, рассмотрим следующий пример, для которого потом выполним рефакторинг с использованием функции `with`.

Листинг 5.16. Генерация алфавита

```
fun alphabet(): String {
    val result = StringBuilder()
    for (letter in 'A'..'Z') {
      result.append(letter)
    }
    result.append("\nNow I know the alphabet!")
    return result.toString()
}
>>> println(alphabet())
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Now I know the alphabet!
```

В этом примере вызывается несколько методов объекта в переменной `result`, и его имя повторяется при каждом вызове. Это не слишком плохо, но что, если используемое выражение будет больше или будет повторяться чаще?

Вот как можно переписать код с помощью функции `with`.

Листинг 5.17. Использование функции `with` для генерации алфавита

```
fun alphabet(): String {
    val stringBuilder = StringBuilder()
    return with(stringBuilder) {
      for (letter in 'A'..'Z') {
        this.append(letter)
      }
      append("\nNow I know the alphabet!")
      this.toString()
    }
}
```

Определяется получатель, методы которого будут вызываться

Вызов метода получателя с помощью ссылки «this»

Вызов метода без ссылки «this»

Возврат значения из лямбда-выражения

Структура `with` выглядит как особая конструкция, но это лишь функция, которая принимает два аргумента – в данном случае объект `stringBuilder` и лямбда-выражение. Здесь используется соглашение о передаче лямбда-выражения за круглыми скобками, поэтому весь вызов выглядит как встроенная конструкция языка. То же самое можно было бы записать как `with(stringBuilder, { ... })`, но такой код труднее читать.

Функция `with` преобразует первый аргумент в *получатель* лямбда-выражения во втором аргументе. Вы можете явно обращаться к этому получателю через ссылку `this`. С другой стороны, можно опустить ссылку `this` и

обращаться к методам или свойствам текущего объекта без дополнительных квалификаторов.

В листинге 5.17 ссылка `this` указывает на экземпляр `stringBuilder`, переданный в первом аргументе. К методам `stringBuilder` можно обращаться явно, через ссылку `this`, как в выражении `this.append(letter)`, или напрямую, как в выражении `append("\nNow...")`.

Лямбда-выражения с получателем и функции-расширения

Мы уже видели похожую идею со ссылкой `this`, указывающей на получатель функции. В теле функции-расширения ссылка `this` указывает на экземпляр типа, который расширяет функция, и её можно опускать, обращаясь к членам получателя напрямую.

Обратите внимание, что функция-расширение – это в некотором смысле функция с получателем. Можете воспользоваться следующей аналогией:

Обычная функция	Функция-расширение
Обычное лямбда-выражение	Лямбда-выражение с получателем

Лямбда-выражение – это способ определения поведения, похожий на обычную функцию. Лямбда-выражение с получателем – это способ определения поведения, аналогичный функции-расширению.

А теперь реорганизуем функцию `alphabet` и избавимся от дополнительной переменной `stringBuilder`.

Листинг 5.18. Применение функции `with` и тела-выражения для генерации алфавита

```
fun alphabet() = with(StringBuilder()) {
    for (letter in 'A'..'Z') {
      append(letter)
    }
    append("\nNow I know the alphabet!")
    toString()
}
```

Теперь эта функция возвращает только выражение, поэтому она была переписана с использованием синтаксиса тела-выражения. Мы создали новый экземпляр `StringBuilder` и напрямую передали его в аргументе, а теперь используем его без явной ссылки `this` внутри лямбда-выражения.

Конфликтующие имена методов

Что произойдет, если в объекте, переданном в функцию `with`, есть метод с таким же именем, как в классе, в котором применяется функция `with`? В этом случае для указания на метод можно явно использовать ссылку `this`.

Представьте, что функция `alphabet` – метод класса `OuterClass`. Если понадобится обратиться к методу `toString` внешнего класса вместо объявленного в классе `StringBuilder`, это можно сделать с помощью следующего синтаксиса:

```
this@OuterClass.toString()
```

Функция `with` возвращает результат выполнения лямбда-выражения – результат последнего выражения в теле лямбда-функции. Но иногда нужно, чтобы результатом вызова стал сам объект-получатель, а не результат выполнения лямбда-выражения. Здесь вам пригодится библиотечная функция `apply`.

5.5.2. Функция «apply»

Функция `apply` работает почти так же, как `with`, – разница лишь в том, что `apply` всегда возвращает объект, переданный в аргументе (другими словами, объект-получатель). Давайте ещё раз реорганизуем функцию `alphabet`, применив на этот раз функцию `apply`.

Листинг 5.19. Использование функции `apply` для генерации алфавита

```
fun alphabet() = StringBuilder().apply {
    for (letter in 'A'..'Z') {
      append(letter)
    }
    append("\nNow I know the alphabet!")
}.toString()
```

Функция `apply` объявлена как функция-расширение. Её получателем становится получатель лямбда-выражения, переданного в аргументе. Результатом вызова `apply` станет экземпляр `StringBuilder`, поэтому нужно вызвать в конце метод `toString`, чтобы превратить его в строку `String`.

Один из многих случаев, где это может пригодиться, – создание экземпляра, у которого нужно сразу инициализировать некоторые свойства. В Java это обычно выполняется с помощью отдельного объекта `Builder`, а в Kotlin можно использовать функцию `apply` с любым объектом без какой-либо специальной поддержки со стороны библиотеки, где этот объект определен.

Чтобы увидеть, как работает функция apply в таких случаях, рассмотрим пример, создающий Android-компонент TextView с некоторыми измененными атрибутами.

Листинг 5.20. Применение функции apply для инициализации экземпляра TextView

```
fun createViewWithCustomAttributes(context: Context) =
    TextView(context).apply {
      text = "Sample Text"
      textSize = 20.0
      setPadding(10, 0, 0, 0)
    }
```

Функция apply позволяет использовать компактный синтаксис тела-выражения. Здесь мы создаем новый экземпляр TextView и немедленно передаем его функции apply. В лямбда-выражении, переданном в функцию apply, экземпляр TextView становится получателем, благодаря чему появляется возможность вызывать его методы и менять свойства. После выполнения лямбда-выражения функция apply вернет уже инициализированный экземпляр – это и станет результатом вызова функции createViewWithCustomAttributes.

Функции with и apply – наиболее типичные примеры использования лямбда-выражений с получателями. Более конкретные функции тоже могут использовать этот шаблон. Например, функцию append можно упростить с помощью стандартной библиотечной функции buildString, которая позаботится о создании экземпляра StringBuilder и вызовет метод toString. Она ожидает получить лямбда-выражение с получателем, а получателем всегда будет экземпляр StringBuilder.

Листинг 5.21. Использование функции buildString для генерации алфавита

```
fun alphabet() = buildString {
    for (letter in 'A'..'Z') {
      append(letter)
    }
    append("\nNow I know the alphabet!")
}
```

Функция buildString – элегантное решение задачи создания строки с помощью класса StringBuilder.

Еще более интересные примеры вы увидите в главе 11, когда мы начнем обсуждать предметно-ориентированные языки (Domain Specific Languages, DSL). Лямбда-выражения с получателями прекрасно подходят для создания DSL; мы покажем, как применить их для этой цели и как определить собственные функции, вызывающие лямбда-выражения с получателями.

5.6. Резюме

- Лямбда-выражения позволяют передавать фрагменты кода в функции.
- Kotlin дает возможность передавать лямбда-выражения в функции за скобками и ссылаться на единственный аргумент лямбда-выражения как на `it`.
- Код внутри лямбда-выражения может читать и изменять локальные переменные функции, в которой оно вызывается.
- Есть возможность ссылаться на методы, конструкторы и свойства, добавляя к их именам префикс `::` и передавая такие ссылки в функции вместо лямбда-выражений.
- Большинство операций над коллекциями могут выполняться без организации итераций вручную, с помощью функций `filter`, `map`, `all`, `any` и т. д.
- Последовательности позволяют объединить несколько операций с коллекцией, не создавая дополнительных коллекций для хранения промежуточных результатов.
- Лямбда-выражения можно передавать в методы, принимающие в параметрах функциональные интерфейсы Java (интерфейсы с одним абстрактным методом, также известные как SAM-интерфейсы).
- Лямбда-выражения с получателем – это особый вид анонимных функций, которые могут непосредственно вызывать методы специального объекта-получателя.
- Функция стандартной библиотеки `with` позволяет вызвать несколько методов одного объекта, не повторяя имени ссылки на него. Функция `apply` позволяет создать и инициализировать любой объект в стиле шаблона «Строитель».

Глава 6

Система типов Kotlin

В этой главе:

- типы с поддержкой пустых значений и работа с `null`;
- простые типы и их соответствие типам Java;
- коллекции в Kotlin и их связь с Java.

На данный момент вы познакомились с большей частью синтаксиса Kotlin, вышли за рамки написания Kotlin-кода, эквивалентного коду на Java, и готовы воспользоваться некоторыми особенностями Kotlin, которые повысят вашу эффективность и сделают код более компактным и читаемым.

Давайте немного сбавим темп и подробнее рассмотрим одну из самых важных частей языка Kotlin: его систему типов. В сравнении с Java система типов в Kotlin имеет особенности, необходимые для повышения надежности кода, такие как *типы с поддержкой пустого значения* и *коллекции, доступные только для чтения*. Система типов в Kotlin также избавлена от некоторых особенностей системы типов в Java, которые оказались ненужными или проблематичными (например, поддержка специального синтаксиса для массивов). Давайте углубимся в детали.

6.1. Поддержка значения null

Поддержка значения `null` – это особенность системы типов Kotlin, которая помогает избежать исключения `NullPointerException`. Как пользователь программ вы наверняка видели такие сообщения: «An error has occurred: java.lang.NullPointerException» (Произошла ошибка: java.lang.NullPointerException) без каких-либо дополнительных подробностей. Другой вариант – сообщение вида: «Unfortunately, the application X has stopped» (К сожалению, приложение X перестало работать), которое часто скрывает истинную причину – исключение `NullPointerException`. Такие ошибки сильно портят жизнь пользователям и разработчикам.

Многие современные языки, в том числе Kotlin, преобразуют такие ошибки времени выполнения в ошибки времени компиляции. Поддерживая значение `null` как часть системы типов, компилятор может обнаружить множество потенциальных ошибок во время компиляции, уменьшая вероятность появления исключений во время выполнения.

В этом разделе мы обсудим типы Kotlin, поддерживающие `null`: как в Kotlin отмечаются элементы, способные принимать значение `null`, и какие инструменты он предоставляет для работы с такими значениями. Также мы рассмотрим нюансы смешения кода Kotlin и Java с точки зрения допустимости значения `null`.

6.1.1. Типы с поддержкой значения null

Первое и самое важное различие между системами типов в Kotlin и Java – это явная поддержка в Kotlin типов, *допускающих значение* `null`. Что это значит? То, что есть способ указать, каким переменным или свойствам в программе разрешено иметь значение `null`. Если переменная может иметь значение `null`, вызов её метода небезопасен, поскольку есть риск получить исключение `NullPointerException`. Kotlin запрещает такие вызовы, предотвращая этим множество потенциальных ошибок. Чтобы увидеть, как это работает на практике, рассмотрим следующую функцию на Java:

```
/* Java */
int strLen(String s) {
  return s.length();
}
```

Безопасна ли эта функция? Если вызвать функцию с аргументом `null`, она возбудит исключение `NullPointerException`. Стоит ли добавить в функцию проверку на `null`? Это зависит от предполагаемого использования функции.

Попробуем переписать её на языке Kotlin. Первый вопрос, на который мы должны ответить: может ли функция вызываться с аргументом `null`? Мы имеем в виду не только непосредственную передачу `null`, как в вызове `strLen(null)`, но и передачу любой переменной или другого выражения, которое во время выполнения может иметь значение `null`.

Если такая возможность не предполагается, тогда функцию можно объявить так:

```
fun strLen(s: String) = s.length
```

Вызов `strLen` с аргументом, который может иметь значение `null`, не допускается и будет расценен как ошибка компиляции:

```
>>> strLen(null)
ERROR: Null can not be a value of a non-null type String
```

Тип параметра объявлен как `String`, а в Kotlin это означает, что он всегда должен содержать экземпляр `String`. Компилятор не позволит передать аргумент, способный содержать значение `null`. Это гарантирует, что функция `strLen` никогда не вызовет исключение, `NullPointerException` во время выполнения.

Чтобы разрешить вызов этой функции с любыми аргументами, в том числе и `null`, мы должны указать это явно, добавив вопросительный знак после имени типа:

```
fun strLenSafe(s: String?) = ...
```

Вопросительный знак можно добавить после любого типа, чтобы указать, что переменные этого типа могут хранить значение `null`: `String?`, `Int?`, `MyCustomType?` и т. д. (см. рис. 6.1).

Рис. 6.1. Если тип допускает, переменная может хранить null

Повторим еще раз: тип без знака вопроса означает, что переменные этого типа не могут хранить значения `null`. То есть все обычные типы по умолчанию не поддерживают `null`, если не указать это явно.

Как только у вас появится значение типа с поддержкой значения `null`, вы сразу обнаружите, что набор допустимых операций с этим значением заметно сузился. Например, вы больше не сможете вызывать его методы:

```
>> fun strLenSafe(s: String?) = s.length()
ERROR: only safe (?.) or non-null asserted (!!.) calls are allowed
on a nullable receiver of type kotlin.String?
```

Вы также не сможете присвоить такое значение переменной, тип которой не поддерживает `null`:

```
>>> val x: String? = null
>>> var y: String = x
ERROR: Type mismatch: inferred type is String? but String was expected
```

Вы не сможете передать значение типа с поддержкой `null` в функцию, параметр который никогда не может быть `null`:

```
>>> strLen(x)
ERROR: Type mismatch: inferred type is String? but String was expected
```

Так что же с этим делать? Самое главное – сравнить его с `null`. Как только вы выполните это сравнение, компилятор запомнит его результат и будет считать, что переменная не может иметь значения `null` в области действия, где было произведено вышеупомянутое сравнение. Например, такой код вполне допустим.

Листинг 6.1. Работа со значением null с помощью проверки if

```
fun strLenSafe(s: String?): Int =
    if (s != null) s.length else 0
>>> val x: String? = null
>>> println(strLenSafe(x))
0
>>> println(strLenSafe("abc"))
3
```

После добавления проверки на null код начал компилироваться

Если бы проверка if была единственным инструментом для работы со значениями null, ваш код довольно быстро превратился бы в многоэтажные наслоения таких проверок. К счастью, Kotlin предоставляет ещё несколько инструментов, помогающих справиться со значением null в более лаконичной манере. Но, прежде чем приступить к их изучению, поговорим немного о смысле допустимости значения null и о том, какие бывают типы переменных.

6.1.2. Зачем нужны типы

Начнем с самых общих вопросов: что такое типы и зачем они переменным? Статья в Википедии о типах (https://ru.wikipedia.org/wiki/Тип_данных) дает очень хороший ответ на первый вопрос: «Тип данных – множество значений и операций на этих значениях».

Попробуем применить это определение к некоторым из типов Java, начиная с типа double. Как известно, тип double представляет 64-разрядное число с плавающей точкой. Над этими значениями можно выполнять стандартные математические операции. Все эти функции в равной степени применимы к любому значению типа double. Поэтому если в программе есть переменная типа double, можно быть уверенным, что любая операция, разрешенная компилятором, будет успешно выполнена со значением данного типа.

Теперь проведем аналогию с переменной типа String. В Java такая переменная может хранить значение одного из двух видов: экземпляр класса String или null. Эти виды значений совершенно непохожи друг на друга: даже Java-оператор instanceof скажет вам, что null – это не String. Операции, допустимые для значений этих видов, тоже совершенно различны: настоящий экземпляр String позволяет вызывать любые строковые методы, в то время как к значению null может применяться лишь ограниченный набор операций.

Это означает, что система типов в Java не справляется со своей задачей. Хотя переменная объявлена с типом String, мы не сможем узнать, какие операции допустимы, не выполнив дополнительную проверку. Часто такие проверки опускаются, поскольку из общего потока данных в програм-

ме известно, что значение не может быть null в некоторой точке. Но иногда вы можете ошибиться – и тогда ваша программа аварийно завершит работу с исключением NullPointerException.

Другие способы борьбы с NullPointerException

В Java есть инструменты, способные помочь решить проблему NullPointerException. Например, некоторые разработчики используют аннотации (например, @Nullable и @NotNull), чтобы указать на допустимость или недопустимость значения null. Существуют инструменты (например, встроенный анализ кода в IntelliJ IDEA), способные использовать эти аннотации для обнаружения места, где может возникнуть NullPointerException. Но такие инструменты не являются частью стандартного процесса компиляции в Java, поэтому их постоянное применение трудно гарантировать. Также трудно снабдить аннотациями всю кодовую базу, включая библиотеки, используемые в проекте, чтобы можно было обнаружить все возможные места возникновения ошибок. Наш собственный опыт в JetBrains показывает, что даже широкое применение аннотаций, определяющих допустимость null, не решает проблему NullPointerException полностью.

Другой подход к решению этой проблемы заключается в запрете использования значения null и в применении особых оберток (таких как тип Optional, появившийся в Java 8) для представления значений, которые могут быть или не быть определены. Этот подход имеет несколько недостатков: код становится более громоздким, дополнительные обертки отрицательно сказываются на производительности, и они не применяются последовательно во всей экосистеме. Даже если вы используете Optional везде в своем коде, вам все равно придется иметь дело со значением null, возвращаемым из методов JDK, фреймворка Android и других сторонних библиотек.

Типы Kotlin с поддержкой null предоставляют комплексное решение этой проблемы. Разграничение типов на поддерживающие и не поддерживающие null дает четкое понимание того, какие операции можно выполнять со значениями, а какие операции могут привести к исключению во время выполнения и поэтому запрещены.

Примечание. Экземпляры типов – с поддержкой или без поддержки null – во время выполнения суть одни и те же. Тип с поддержкой значения null – это не обертка вокруг обычного типа. Все проверки выполняются во время компиляции. Это означает, что типы с поддержкой null в Kotlin не расходуют дополнительных системных ресурсов во время выполнения.

Теперь посмотрим, как работать с типами, поддерживающими null, и почему работа с ними не раздражает. Начнем со специального оператора безопасного доступа к значениям, которые содержат null.

6.1.3. Оператор безопасного вызова: «?.»

Один из самых полезных инструментов в арсенале Kotlin – это оператор безопасного вызова: `?.` Он сочетает в одной операции проверку на `null` и вызов метода. Например, выражение `s?.toUpperCase()` эквивалентно следующему, более громоздкому: `if (s != null) s.toUpperCase() else null`.

Другими словами, если значение, метод которого вы пытаетесь вызвать, не является `null`, то вызов будет выполнен обычным образом. Если значение равно `null`, вызов игнорируется, и в качестве результата возвращается `null`. Работа оператора показана на рис. 6.2.

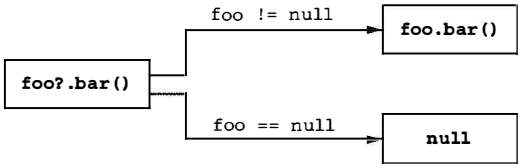


Рис. 6.2. Оператор безопасного вызова выполняет методы лишь для значений, отличных от null

Обратите внимание, что результатом такого вызова может быть `null`. Хотя метод `String.toUpperCase` возвращает значение типа `String`, выражение `s?.toUpperCase()` для `s` с поддержкой `null` получит тип `String?`:

```
fun printAllCaps(s: String?) {
    val allCaps: String? = s?.toUpperCase()
    println(allCaps)
}
```

Переменная allCaps может хранить null

```
>>> printAllCaps("abc")
ABC
>>> printAllCaps(null)
null
```

Оператор безопасного вызова также можно использовать не только для вызова методов, но и для доступа к свойствам. В следующем примере показан простой класс Kotlin со свойством, способным принимать значение `null`, и продемонстрировано использование оператора безопасного вызова для доступа к этому свойству.

Листинг 6.2. Применение оператора безопасного вызова для доступа к свойствам, способным принимать значение null

```
class Employee(val name: String, val manager: Employee?)

fun managerName(employee: Employee): String? = employee.manager?.name

>>> val ceo = Employee("Da Boss", null)
>>> val developer = Employee("Bob Smith", ceo)
```

```
>>> println(managerName(developer))
Da Boss
>>> println(managerName(ceo))
null
```

Если в программе есть иерархия объектов, различные свойства которых способны принимать значение `null`, бывает удобно использовать несколько безопасных вызовов в одном выражении. Представьте, что вы храните информацию о человеке, его компании и адресе компании, используя различные классы. И компания, и её адрес могут быть опущены. С помощью оператора `?.` можно в одном вызове без дополнительных проверок определить, из какой страны этот человек.

Листинг 6.3. Объединение нескольких операторов безопасного вызова

```
class Address(val streetAddress: String, val zipCode: Int,
        val city: String, val country: String)

class Company(val name: String, val address: Address?)

class Person(val name: String, val company: Company?)

fun Person.countryName(): String {
    val country = this.company?.address?.country
    return if (country != null) country else "Unknown"
}
>>> val person = Person("Dmitry", null)
>>> println(person.countryName())
Unknown
```

Объединение нескольких операторов безопасного вызова

Последовательность вызовов с проверками на `null` в Java – обычное явление, и сейчас вы увидели, как Kotlin делает её компактнее. Но в листинге 6.3 есть ненужные повторения: мы сравниваем результат с `null` и возвращаем само значение либо что-то ещё, если оно равно `null`. Давайте посмотрим, поможет ли Kotlin избавиться от этого повторения.

6.1.4. Оператор «Элвис»: «?:»

В языке Kotlin есть удобный оператор для замены `null` значениями по умолчанию. Он называется *оператор «Элвис»* (или *оператор объединения со значением null*, если вы предпочитаете официальные названия). Выглядит он так: `?:` (если наклонить голову влево, можно увидеть изображение Элвиса). Вот как он используется:

```
fun foo(s: String?) {
    val t: String = s ?: ""
}
```

Если «s» хранит null, вернуть пустую строку

Оператор принимает два значения и возвращает первое, если оно не равно null, а в противном случае – второе. На рис. 6.3 показано, как это работает.

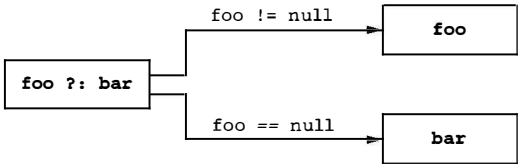


Рис. 6.3. Оператор «Элвис» подставляет конкретное значение вместо null

Оператор «Элвис» часто используется вместе с оператором безопасного вызова, чтобы вернуть значение, отличное от null, когда объект, метод которого вызывается, сам может вернуть null. Вот как можно применить этот шаблон, чтобы упростить код в листинге 6.1.

Листинг 6.4. Применение оператора «Элвис» для работы с null

```
fun strLenSafe(s: String?): Int = s?.length ?: 0

>>> println(strLenSafe("abc"))
3
>>> println(strLenSafe(null))
0
```

Функцию countryName из листинга 6.3 тоже можно уместить в одну строку:

```
fun Person.countryName() =
    company?.address?.country ?: "Unknown"
```

Особенно удобным оператор «Элвис» делает то обстоятельство, что такие операции, как return и throw, действуют как выражения и, следовательно, могут находиться справа от оператора. То есть если значение слева окажется равным null, функция немедленно вернет управление или возбудит исключение. Это полезно для проверки предусловий в функции.

Давайте посмотрим, как применить этот оператор, чтобы реализовать функцию печати почтовой наклейки с адресом компании человека, которого мы упоминали выше. В листинге 6.5 повторяются объявления всех классов, но в Kotlin они настолько лаконичны, что это не представляет неудобств.

Листинг 6.5. Использование оператора throw с оператором «Элвис»

```
class Address(val streetAddress: String, val zipCode: Int,
```

```
    val city: String, val country: String)

class Company(val name: String, val address: Address?)

class Person(val name: String, val company: Company?)

fun printShippingLabel(person: Person) {
    val address = person.company?.address
      ?: throw IllegalArgumentException("No address")
    with (address) {
      println(streetAddress)
      println("$zipCode $city, $country")
    }
}
```

Вызовет исключение в отсутствие адреса

Переменная «address» хранит непустое значение

```
>>> val address = Address("Elsestr. 47", 80687, "Munich", "Germany")
>>> val jetbrains = Company("JetBrains", address)
>>> val person = Person("Dmitry", jetbrains)

>>> printShippingLabel(person)
Elsestr. 47
80687 Munich, Germany

>>> printShippingLabel(Person("Alexey", null))
java.lang.IllegalArgumentException: No address
```

Если все правильно, функция `printShippingLabel` напечатает наклейку. Если адрес отсутствует, она не просто сгенерирует исключение `NullPointerException` с номером строки, но вернет осмысленное сообщение об ошибке. Если адрес присутствует, текст наклейки будет состоять из адреса, почтового индекса, города и страны. Обратите внимание, как здесь используется функция `with`, которую мы видели в предыдущей главе: она помогает избежать повторения переменной `address` четыре раза подряд.

Теперь, когда вы знаете, как Kotlin выполняет проверки на `null`, давайте поговорим о безопасной версии оператора `instanceof` в Kotlin: *операторе безопасного приведения*, который часто появляется вместе с безопасными вызовами и оператором «Элвис».

6.1.5. Безопасное приведение типов: оператор «as?»

В главе 2 вы уже видели оператор `as`, используемый в Kotlin для приведения типов. Как и в Java, оператор `as` возбудит `ClassCastException`, если значение нельзя привести к указанному типу. Конечно, можно совместить его с проверкой `is`, чтобы убедиться, что значение имеет правильный тип. Но разве в таком безопасном и лаконичном языке, как Kotlin, нет лучшего решения? Безусловно, есть.

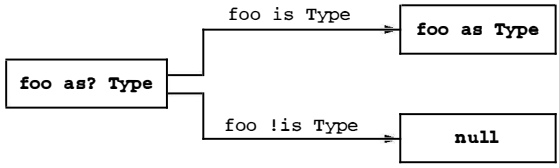


Рис. 6.4. Оператор безопасного приведения пытается привести значение к заданному типу и возвращает null, если фактический тип отличается от ожидаемого

Оператор as? пытается привести значение указанного типа и возвращает null, если значение нельзя привести к указанному типу, как показано на рис. 6.4.

Оператор безопасного приведения часто используется в сочетании с оператором «Элвис». Например, это может пригодиться для реализации метода equals.

Листинг 6.6. Использование оператора безопасного приведения для реализации метода equals

```
class Person(val firstName: String, val lastName: String) {
    override fun equals(o: Any?): Boolean {
      val otherPerson = o as? Person ?: return false

      return otherPerson.firstName == firstName &&
         otherPerson.lastName == lastName
    }

    override fun hashCode(): Int =
      firstName.hashCode() * 37 + lastName.hashCode()
}

>>> val p1 = Person("Dmitry", "Jemerov")
>>> val p2 = Person("Dmitry", "Jemerov")
>>> println(p1 == p2)
true
>>> println(p1.equals(42))
false
```

Проверит тип и вернет false, если указанный тип недопустим

После безопасного приведения типа переменная otherPerson преобразуется к типу Person

Оператор == вызывает метод «equals»

Этот шаблон позволяет легко убедиться, что параметр обладает правильным типом, выполнить его приведение, вернуть false, если значение имеет несовместимый тип, – и все в одном выражении. Автоматическое приведение типов также применимо в данном контексте: после проверки типа и отсеивания значения null компилятор знает, что переменная otherPerson имеет тип Person, и позволяет использовать её соответствующим образом.

Операторы безопасного вызова, безопасного приведения и оператор «Элвис» очень удобны и часто встречаются в коде на Kotlin. Но иногда бы-

вает нужно просто сообщить компилятору, что значение на самом деле не может быть равно null. Давайте посмотрим, как можно это сделать.

6.1.6. Проверка на null: утверждение «!!»

Данное утверждение – самый простой и незатейливый способ, который Kotlin предоставляет для работы со значением null. Оно записывается как двойной восклицательный знак и преобразует любое значение к типу, не поддерживающему значения null. Если значение равно null, во время выполнения возникнет исключение. Логика работы утверждения показана на рис. 6.5.

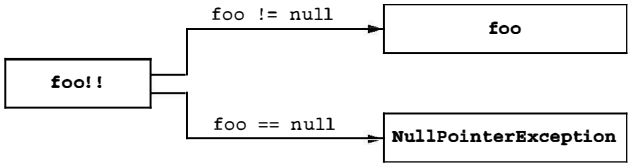


Рис. 6.5. Используя утверждение, что значение не равно null, можно явно возбудить исключение, если значение окажется равным null

Вот простой пример функции, использующей утверждение для преобразования аргумента из типа с поддержкой null в тип без поддержки null.

Листинг 6.7. Использование утверждения о невозможности значения null

```
fun ignoreNulls(s: String?) {
    val sNotNull: String = s!!
    println(sNotNull.length)
}
```

Исключение указывает на эту строку

```
>>> ignoreNulls(null)
Exception in thread "main" kotlin.KotlinNullPointerException
  at <...>.ignoreNulls(07_NotnullAssertions.kt:2)
```

Что произойдет, если передать этой функции значение null в аргументе s? У Kotlin не останется выбора: во время выполнения он возбудит исключение (особый вид исключения NullPointerException). Но обратите внимание, что исключение ссылается на строку с утверждением, а не на инструкцию вызова. По сути, вы говорите компилятору: «Я знаю, что значение не равно null, и я готов получить исключение, если выяснится, что я ошибаюсь».

> **Примечание.** Вы можете заметить, что двойной восклицательный знак выглядит грубовато – как будто вы кричите на компилятор. Так и было задумано. Разработчики языка Kotlin пытаются подтолкнуть вас в сторону лучших решений, без заявлений, которые не могут быть проверены компилятором.

Но бывают ситуации, когда такие утверждения (что значение не равно null) оптимально решают проблему: когда вы делаете проверку на null в одной функции и используете это значение в другой, компилятор не может понять, что такое использование безопасно. Если вы уверены, что проверка всегда выполняется в другой функции, у вас может появиться желание отказаться от её повторения перед использованием значения. В таком случае вы можете применить утверждение, заявив, что значение не может быть равно null.

На практике это происходит с классами обработчиков, определяемых во многих фреймворках для создания пользовательских интерфейсов – например, таких как Swing. Класс-обработчик имеет отдельные методы для обновления состояния (его включения или отключения) и выполнения действия. Проверки, выполненные в методе update, гарантируют, что метод execute не будет вызван в отсутствие необходимых условий, но компилятор не способен понять это.

Давайте рассмотрим пример реализации обработчика в Swing с использованием утверждения !!. Объект класса CopyRowAction должен скопировать значение из строки, выбранной в списке, в буфер обмена. Мы опустим все ненужные детали, оставив только код, проверяющий факт выбора строки (т. е. что действие может быть выполнено) и получающий выбранную строку. Интерфейс Action API предполагает, что метод actionPerformed можно вызвать, только если метод isEnabled вернет true.

Листинг 6.8. Использование утверждения !! в обработчике Swing

```
class CopyRowAction(val list: JList<String>) : AbstractAction() {
    override fun isEnabled(): Boolean =
      list.selectedValue != null

    override fun actionPerformed(e: ActionEvent) {
      val value = list.selectedValue!!
      // copy value to clipboard
    }
}
```

Метод actionPerformed будет вызван, только если isEnabled вернет «true»

Обратите внимание: если вы не хотите использовать утверждение !! в этой ситуации, то можете написать val value = list.selectedValue ?: return, чтобы получить значение, отличное от null. Если list.selectedValue вернет null, это приведет к досрочному выходу из функции и value никогда не получит значения null. Хотя проверка на null с помощью оператора «Элвис» здесь лишняя, она может оказаться хорошей страховкой, если в будущем метод isEnabled усложнится.

Ещё один нюанс, который стоит держать в уме: когда вы используете утверждение !! и в результате получаете исключение, трассировка стека

покажет только номер строки, в которой оно возникло, а не конкретное выражение. Чтобы выяснить, какое значение оказалось равно `null`, лучше избегать использования нескольких утверждений `!!` в одной строке:

```
person.company!!.address!!.country      <— Не пишите такого кода!
```

Если вы получите исключение в этой строке, то не сможете сказать, какое поле получило значение `null`: `company` или `address`.

До сих пор мы в основном обсуждали, как *обращаться* к значениям типов с поддержкой значения `null`. Но что делать, если требуется передать аргумент, который может оказаться значением `null`, в функцию, которая ожидает значения, не равного `null`? Компилятор не позволит сделать это без проверки, поскольку это небезопасно. В языке Kotlin нет специального механизма для этого случая, но в стандартной библиотеке есть одна функция, которая может вам помочь: она называется `let`.

6.1.7. Функция let

Функция `let` облегчает работу с выражениями, допускающими значение `null`. Вместе с оператором безопасного вызова она позволяет вычислить выражение, проверить результат на `null` и сохранить его в переменной – и все это в одном коротком выражении.

Чаще всего она используется для передачи аргумента, который может оказаться равным `null`, в функцию, которая ожидает параметра, не равного `null`. Допустим, функция `sendEmailTo` принимает один параметр типа `String` и отправляет электронное письмо на этот адрес. Эта функция написана на Kotlin и требует параметра, который не равен `null`:

```
fun sendEmailTo(email: String) { /*...*/ }
```

Вы не сможете передать в эту функцию значение типа с поддержкой `null`:

```
>>> val email: String? = ...
>>> sendEmailTo(email)
ERROR: Type mismatch: inferred type is String? but String was expected
```

Вы должны явно проверить, что значение не равно `null`:

```
if (email != null) sendEmailTo(email)
```

Но можно пойти другим путем: применить функцию `let`, объединив её с оператором безопасного вызова. Функция `let` просто превращает объект вызова в параметр лямбда-выражения. Объединяя её с синтаксисом безопасного вызова, вы фактически преобразуете объект из типа с поддержкой `null` в тип без поддержки `null` (см. рис. 6.6).

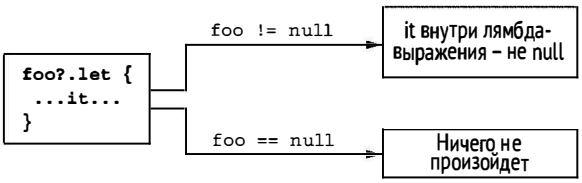


Рис. 6.6. Безопасный вызов функции let выполнит лямбда-выражение, только если выражение не равно null

Функция `let` будет вызвана, только если значение адреса электронной почты не равно `null`, поэтому его можно использовать как аргумент лямбда-выражения, не равный `null`:

```
email?.let { email -> sendEmailTo(email) }
```

С использованием краткого синтаксиса – с автоматически созданным именем `it` – результат станет ещё лаконичнее: `email?.let { sendEmailTo(it) }`. Ниже показан полноценный пример, демонстрирующий этот шаблон.

Листинг 6.9. Применение функции `let` для вызова функции, не принимающей `null`

```
fun sendEmailTo(email: String) {
    println("Sending email to $email")
}

>>> var email: String? = "yole@example.com"
>>> email?.let { sendEmailTo(it) }
Sending email to yole@example.com
>>> email = null
>>> email?.let { sendEmailTo(it) }
```

Обратите внимание, что нотация `let` особенно удобна, когда нужно использовать значение большого выражения, не равного `null`. В этом случае вам не придется создавать отдельную переменную. Сравните следующий фрагмент с явной проверкой:

```
val person: Person? = getTheBestPersonInTheWorld()
if (person != null) sendEmailTo(person.email)
```

с аналогичной реализацией без дополнительной переменной:

```
getTheBestPersonInTheWorld()?.let { sendEmailTo(it.email) }
```

Эта функция всегда возвращает `null`, поэтому лямбда-выражение никогда не будет выполнено.

```
fun getTheBestPersonInTheWorld(): Person? = null
```

Чтобы проверить несколько значений на `null`, можно использовать вложенные вызовы `let`. Но в большинстве случаев такой код становится слишком трудным для понимания. Обычно для проверки сразу всех выражений проще использовать простой оператор `if`.

Еще одна распространенная ситуация – непустые свойства, которые тем не менее невозможно инициализировать в конструкторе значением, отличным от `null`. Давайте посмотрим, как Kotlin помогает справиться с этой ситуацией.

6.1.8. Свойства с отложенной инициализацией

Многие фреймворки инициализируют объекты в специальных методах, которые вызываются после создания экземпляра. Например, в Android инициализация осуществляется в методе `onCreate`. JUnit требует помещать логику инициализации в методы с аннотацией `@Before`.

Но вы не можете оставить свойства, не поддерживающего значения `null`, без инициализации в конструкторе, присваивая ему значение только в специальном методе. Kotlin требует инициализации всех свойств в конструкторе, и если свойство не может иметь значения `null`, вы обязаны указать значение для инициализации. Если нет возможности предоставить такое значение, приходится использовать тип с поддержкой `null`. Но в этом случае при каждом обращении к свойству придется использовать проверку на `null` или утверждение `!!`.

Листинг 6.10. Применение утверждений `!!` для доступа к полю с поддержкой `null`

```
class MyService {
    fun performAction(): String = "foo"
}

class MyTest {
    private var myService: MyService? = null   <-- Объявление свойства с типом, поддерживающим null, чтобы инициализировать его значением null

    @Before fun setUp() {
      myService = MyService()                  <-- Настоящее значение присваивается в методе setUp
    }

    @Test fun testAction() {
      Assert.assertEquals("foo",
        myService!!.performAction())           <-- Из-за такого объявления приходится использовать !! или ?
    }
}
```

Это неудобно, особенно когда приходится обращаться к свойству много раз. Чтобы решить эту проблему, можно объявить, что поле `myService`

поддерживает *отложенную инициализацию*. Это делается с помощью модификатора `lateinit`.

Листинг 6.11. Применение свойства с «ленивой» инициализацией

```
class MyService {
    fun performAction(): String = "foo"
}

class MyTest {
    private lateinit var myService: MyService      <- Объявление свойства с типом, не поддерживающим null, без инициализации

    @Before fun setUp() {
      myService = MyService()                      <- Инициализация в методе setup такая же, как раньше
    }

    @Test fun testAction() {
      Assert.assertEquals("foo",
        myService.performAction())                 <- Обращение к свойству без лишних проверок на null
    }
}
```

Обратите внимание, что поля с отложенной инициализацией всегда объявляются как `var`, потому что их значения изменяются за пределами конструктора, в то время как свойства `val` компилируются в финальные поля, которые должны инициализироваться в конструкторе. Поля с отложенной инициализацией больше не требуется инициализировать в конструкторе, даже если их типы поддерживают `null`. Попытка обратиться к такому свойству прежде, чем оно будет инициализировано, возбудит исключение «lateinit property myService has not been initialized» (свойство myService с модификатором lateinit не было инициализировано). Оно четко сообщит, что произошло, и понять его гораздо легче, чем обычное исключение `NullPointerException`.

> **Примечание.** Свойства с модификатором `lateinit` широко используются для внедрения зависимостей. Значения таких свойств устанавливаются извне специализированным фреймворком. Для совместимости с широким спектром фреймворков Java язык Kotlin генерирует поле с такой же видимостью, как и свойство с модификатором `lateinit`. Если свойство объявлено как `public`, поле тоже получит модификатор `public`.

Теперь посмотрим, как можно расширить набор инструментов Kotlin для работы с `null`, определяя функции-расширения для типов с поддержкой `null`.

6.1.9. Расширение типов с поддержкой null

Определение функций-расширений для типов с поддержкой `null` – ещё один мощный способ справляться с `null`. Вместо того чтобы проверять переменную на неравенство `null` перед вызовом метода, можно разрешить вызовы функций, где в роли получателя выступает `null`, и иметь дело с `null` в этой функции. Это возможно только для функций-расширений – обычные вызовы методов класса направляются экземпляру класса и, следовательно, не могут быть выполнены, если экземпляр равен `null`.

В качестве примера рассмотрим функции `isEmpty` и `isBlank`, определенные как расширения класса `String` в стандартной библиотеке Kotlin. Первая проверяет, пуста ли строка (т. е. строка `""`), а вторая проверяет, что она пуста или состоит только из символов пробела. Обычно эти функции используются, чтобы убедиться, что строка не тривиальна, и затем сделать с ней что-то осмысленное. Вам может показаться, что было бы полезно обрабатывать значения `null` так же, как пустые строки или строки, состоящие из пробелов. И это действительно возможно: функции `isEmptyOrNull` и `isBlankOrNull` могут вызываться с получателем типа `String?`.

Листинг 6.12. Вызов функции-расширения с получателем, тип которого поддерживает `null`

```
fun verifyUserInput(input: String?) {
    if (input.isNullOrBlank()) {
      println("Please fill in the required fields")
    }
}

>>> verifyUserInput(" ")
Please fill in the required fields
>>> verifyUserInput(null)
Please fill in the required fields
```

Не требуется использовать оператора безопасного вызова

Если вызвать isNullOrBlank с null в качестве получателя, это не приведет к исключению

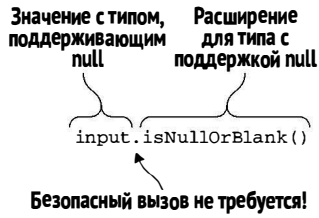


Рис. 6.7. Расширения для типов с поддержкой null могут вызываться без синтаксиса безопасного вызова

Функцию-расширение, объявленную допускающей использование `null` в качестве получателя, можно вызывать без оператора безопасного вызова (см. рис. 6.7). Функция сама обработает возможные значения `null`.

Функция `isNullOrBlank` выполняет явную проверку на `null`, возвращая значение `true`, а иначе вызывает `isBlank`, которую можно применять только к строкам, которые не могут быть `null`:

```
fun String?.isNullOrBlank(): Boolean =
    this == null || this.isBlank()
```

Расширение для типа String с поддержкой null

Во втором обращении к «this» применяется автоматическое приведение типов

Когда функция-расширение объявляется для типа с поддержкой null (имя которого оканчивается на ?), это означает, что её можно вызывать для значений null; в этом случае ссылка this в теле функции может оказаться равной null, поэтому её следует проверять явно. В Java ссылка this никогда не может быть null, поскольку она ссылается на экземпляр класса, которому принадлежит метод. В Kotlin это изменилось: в функции-расширении для типа с поддержкой null ссылка this может оказаться равной null.

Обратите внимание, что ранее рассмотренная функция let тоже может быть вызвана для получателя null, но сама функция не будет выполнять никаких проверок. Если попытаться вызвать её для типа с поддержкой null без оператора безопасного вызова, аргумент лямбда-выражения тоже будет поддерживать null:

```
>>> val person: Person? = ...
>>> person.let { sendEmailTo(it) }
ERROR: Type mismatch: inferred type is Person? but Person was expected
```

Небезопасный вызов, поэтому «it» может оказаться равным null

Следовательно, если вы хотите проверять аргументы на неравенство значению null с помощью функции let, вы должны использовать оператор безопасного вызова ?., как было показано выше: person?.let { sendEmailTo(it) }.

> **Примечание.** Создавая собственную функцию-расширение, вам стоит задуматься о том, нужно ли определять её как расширение типа с поддержкой null. По умолчанию имеет смысл определять как расширение типа без поддержки null. Вы можете спокойно изменить её позже (существующий код не сломается), если выяснится, что функция в основном используется для значений, которые могут быть равны null, и может адекватно обработать такие значения.

В этом разделе вы увидели кое-что неожиданное. Разыменование переменной без дополнительной проверки, как в s.isNullOrBlank(), не означает, что переменная не может хранить значения null: используемая функция может расширять тип, допускающий такие значения. Далее давайте обсудим другой случай, который тоже может вас удивить: параметр типа может поддерживать значение null даже без вопросительного знака в конце.

6.1.10. Типовые параметры с поддержкой null

По умолчанию все типовые параметры функций и классов в Kotlin автоматически поддерживают null. Любой тип, в том числе с поддержкой null, может быть использован как типовой параметр – в этом случае объявления, использующие типовые параметры, должны предусматривать поддержку null, даже если имя параметра T не оканчивается вопросительным знаком. Рассмотрим следующий пример.

Листинг 6.13. Работа с параметром типа, допускающим значение `null`

```
fun <T> printHashCode(t: T) {
    println(t?.hashCode())
}

>>> printHashCode(null)
null
```

Обязательно должен использоваться безопасный вызов, поскольку «t» может хранить null

Параметр «T» определяется как тип «Any?»

Для вызова функции `printHashCode` в листинге 6.13 компилятор определит параметр `T` как тип `Any?` с поддержкой `null`. Соответственно, параметр `t` может оказаться равным `null` даже притом, что в имени типа `T` отсутствует вопросительный знак.

Чтобы запретить параметру принимать значение `null`, необходимо определить соответствующую верхнюю границу. Это не позволит передать `null` в качестве аргумента.

Листинг 6.14. Объявление верхней границы, не допускающей `null`, для параметра типа

```
fun <T: Any> printHashCode(t: T) {
    println(t.hashCode())
}

>>> printHashCode(null)
Error: Type parameter bound for 'T' is not satisfied
>>> printHashCode(42)
42
```

Теперь «T» не поддерживает null

Этот код не скомпилируется: нельзя передать null, поскольку это запрещено

Про обобщенные типы в Kotlin мы еще поговорим в главе 9, и особенно детально – в разделе 9.1.4.

Обратите внимание, что типовые параметры – это единственное исключение из правила «вопросительный знак в конце имени типа разрешает значение `null`, а типы с именами без знака вопроса не допускают `null`». В следующем разделе демонстрируется другой частный случай допустимости значения `null`: типы, пришедшие из кода Java.

6.1.11. Допустимость значения null и Java

Выше мы рассмотрели инструменты для работы с `null` в мире Kotlin. Но Kotlin гордится своей совместимостью с Java, а мы знаем, что система типов в Java не поддерживает управления допустимостью значений `null`. Что же происходит, когда вы объединяете Kotlin и Java? Сохранится ли безопасность или же вам потребуется проверять каждое значение на `null`? И есть ли лучшее решение? Давайте выясним это.

Во-первых, как уже упоминалось, код на Java может содержать информацию о допустимости значений null, выраженную с помощью аннотаций. Если эта информация присутствует в коде, Kotlin воспользуется ею. То есть объявление @Nullable String в Java будет представлено в Kotlin как String?, а объявление @NotNull String – как String (см. рис. 6.8).

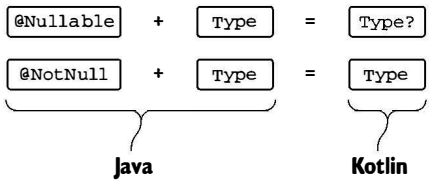


Рис. 6.8. Типы Java, отмеченные аннотациями, представлены в Kotlin как типы, поддерживающие или не поддерживающие значение null

Kotlin распознает множество разных аннотаций, описывающих допустимость значения null, в том числе аннотации из стандарта JSR-305 (javax.annotation), из Android (android.support.annotation) и те, что поддерживаются инструментами компании JetBrains (org.jetbrains.annotations). Интересно, а что происходит, когда аннотация отсутствует? В этом случае тип Java становится *платформенным типом* в Kotlin.

Платформенные типы

Платформенный тип – это тип, для которого Kotlin не может найти информацию о допустимости null. С ним можно работать как с типом, допускающим null, или как с типом, не допускающим null (см. рис. 6.9).

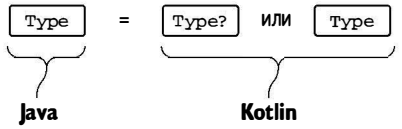


Рис. 6.9. Типы Java представлены в Kotlin как платформенные типы, которые могут поддерживать или не поддерживать значение null

Это означает, что, точно как в Java, вы несете полную ответственность за операции с этим типом. Компилятор разрешит вам делать всё. Он также не станет напоминать об избыточных безопасных операциях над такими значениями (как он делает обычно, встречая безопасные операции над значениями, которые не могут быть null). Если известно, что значение может быть равно null, вы можете сравнить его с null перед использованием. Если известно, что оно не может быть равно null, вы сможете использовать его напрямую. Но, как и в Java, если вы ошибетесь, то получите исключение NullPointerException во время использования.

Предположим, что класс Person объявлен в Java.

Листинг 6.15. Java-класс без аннотаций, определяющих допустимость `null`

```
/* Java */
public class Person {
    private final String name;

    public Person(String name) {
      this.name = name;
    }

    public String getName() {
      return name;
    }
}
```

Может ли метод `getName` вернуть значение `null`? Компилятор Kotlin ничего не знает о допустимости `null` для типа `String` в данном случае, поэтому вам придется справиться с этим самостоятельно. Если вы уверены, что имя не будет равно `null`, можете разыменовать его в обычном порядке (как в Java) без дополнительных проверок, но будьте готовы столкнуться с исключением.

Листинг 6.16. Обращение к Java-классу без дополнительных проверок на `null`

```
fun yellAt(person: Person) {
    println(person.name.toUpperCase() + "!!!")
}
```

Получатель метода toUpperCase() – поле person.name – равен null, поэтому будет возбуждено исключение

```
>>> yellAt(Person(null))
java.lang.IllegalArgumentException: Parameter specified as non-null
is null: method toUpperCase, parameter $receiver
```

Обратите внимание, что вместо обычного исключения `NullPointerException` вы получите более подробное сообщение об ошибке: что метод `toUpperCase` не может вызываться для получателя, равного `null`.

На самом деле для общедоступных функций Kotlin компилятор генерирует проверки для каждого параметра (и для получателя в том числе), который не должен принимать значения `null`. Поэтому попытка вызвать такую функцию с неверными аргументами сразу закончится исключением. Обратите внимание, что проверка значения выполняется сразу же при вызове функции, а не во время использования параметра. Это гарантирует обнаружение некорректных вызовов на ранних стадиях и избавляет от непонятных исключений, возникающих после того, как значение `null` прошло через несколько функций в разных слоях кода.

Другой вариант – интерпретация типа значения, возвращаемого методом `getName()`, как допускающего значение `null`, и обращение к нему с помощью безопасных вызовов.

Листинг 6.17. Обращение к классу Java с проверками на `null`

```
fun yellAtSafe(person: Person) {
    println((person.name ?: "Anyone").toUpperCase() + "!!!")
}

>>> yellAtSafe(Person(null))
ANYONE!!!
```

В этом примере значения `null` обрабатываются должным образом, и во время выполнения никаких исключений не возникнет.

Будьте осторожны при работе с Java API. Большинство библиотек не использует аннотаций, поэтому, если вы будете интерпретировать все типы как не поддерживающие `null`, это может привести к ошибкам. Во избежание ошибок читайте документацию с описанием методов Java, которые используете (а если нужно, то и их реализацию), выясните, когда они могут возвращать `null`, и добавьте проверку для таких методов.

> **Зачем нужны платформенные типы?**
>
> Разве не безопаснее для Kotlin считать, что все значения, поступающие из Java, могут оказаться равными `null`? Такое возможно, но требует большого количества лишних проверок на `null` для значений, которые на самом деле никогда не могут быть `null`, но компилятор Kotlin не имеет информации для того, чтобы это понять.
>
> Особенно плохой была бы ситуация с обобщенными типами – например, каждая коллекция `ArrayList<String>`, приходящая из Java, превращалась бы в Kotlin в `ArrayList<String?>?`. В результате вам пришлось бы выполнять проверку на `null` при каждом обращении или применить приведение типа, что повлияло бы на безопасность. Написание таких проверок очень раздражает, поэтому создатели Kotlin пришли к прагматичному решению: позволить разработчикам самим брать на себя ответственность за правильность обработки значений, поступающих из Java.

Вы не можете объявить переменную платформенного типа в Kotlin – эти типы могут прийти только из кода на Java. Но вы можете встретить их в сообщениях об ошибках или в IDE:

```
>>> val i: Int = person.name
ERROR: Type mismatch: inferred type is String! but Int was expected
```

С помощью нотации `String!` компилятор Kotlin обозначает платформенные типы, пришедшие из кода на Java. Вы не можете использовать это-

го синтаксиса в своем коде, и чаще всего этот восклицательный знак не связан с источником проблемы, поэтому его можно игнорировать. Он просто подчеркивает, что допустимость значения `null` не была установлена.

Как мы уже сказали, платформенные типы можно интерпретировать как угодно – как допускающие или как не допускающие `null`, – поэтому следующие объявления допустимы:

```
>>> val s: String? = person.name
>>> val s1: String = person.name
```

Получатель метода toUpperCase() – поле person.name – равен null, поэтому будет возбуждено исключение...

...или как не допускающее

Как и при вызове методов, в этом случае вы должны быть уверены, что понимаете, где может появиться `null`. При попытке присвоить пришедшее из Java значение `null` переменной Kotlin, которая не может хранить `null`, вы понимаете, где получите исключение в месте вызова.

Мы обсудили, как типы Java выглядят с точки зрения Kotlin. Теперь давайте поговорим о некоторых подводных камнях создания смешанных иерархий Kotlin и Java.

Наследование

Переопределяя Java-метод в коде на Kotlin, вы можете выбрать, будут ли параметры и возвращаемое значение поддерживать `null` или нет. К примеру, давайте взглянем на интерфейс `StringProcessor` в Java.

Листинг 6.18. Java-интерфейс с параметром типа String

```
/* Java */
interface StringProcessor {
    void process(String value);
}
```

Компилятор Kotlin допускает следующие реализации.

Листинг 6.19. Реализация Java-интерфейса с поддержкой и без поддержки `null`

```
class StringPrinter : StringProcessor {
    override fun process(value: String) {
      println(value)
    }
}

class NullableStringPrinter : StringProcessor {
    override fun process(value: String?) {
      if (value != null) {
        println(value)
```

```
        }
    }
}
```

Обратите внимание, что при реализации методов классов или интерфейсов Java важно правильно понимать, когда можно получить `null`. Поскольку методы могут быть вызваны из кода не на Kotlin, компилятор сгенерирует проверки, что значение не может быть равно `null`, для каждого параметра, объявление которого запрещает присваивание `null`. Если Java-код передаст в метод значение `null`, то сработает проверка, и вы получите исключение, даже если ваша реализация вообще не обращается к значению параметра.

Давайте подведем итоги нашей дискуссии о допустимости значений `null`. Мы обсудили типы с поддержкой и без поддержки `null`, а также способы работы с ними: операторы безопасности (безопасный вызов `?.`, оператор «Элвис» `?:` и оператор безопасного приведения `as?`), а также оператор небезопасного разыменования (утверждение `!!`). Вы увидели, как библиотечная функция `let` может выполнить лаконичную проверку на `null` и как функции-расширения для типов с поддержкой `null` помогают переместить проверку на `null` в функцию. Мы также обсудили платформенные типы, представляющие типы Java в Kotlin.

Теперь, рассмотрев тему допустимости значений `null`, поговорим о представлении простых типов в Kotlin. Знание особенностей поддержки `null` имеет большое значение для понимания того, как Kotlin работает с классами-обёртками Java.

6.2. Примитивные и другие базовые типы

В данном разделе описываются основные типы, используемые в программах: `Int`, `Boolean`, `Any` и другие. В отличие от Java, язык Kotlin не делит типы на простые и обертки. Вскоре вы узнаете причину этого дизайн-решения и то, какие механизмы работают за кулисами, и также увидите соответствие между типами Kotlin и такими типами Java, как `Object` и `Void`.

6.2.1. Примитивные типы: Int, Boolean и другие

Как известно, Java различает примитивные и ссылочные типы. Переменная *примитивного типа* (например, `int`) непосредственно содержит свое значение. Переменная *ссылочного типа* (например, `String`) содержит ссылку на область памяти, где хранится объект.

Значения примитивных типов можно хранить и передавать более эффективно, но такие значения не имеют методов, и их нельзя сохранять в коллекциях. Java предоставляет специальные типы-обертки (такие как `java.lang.Integer`), которые инкапсулируют примитивные типы в ситуа-

циях, когда требуется объект. То есть, чтобы определить коллекцию целых чисел, вместо `Collection<int>` следует написать `Collection<Integer>`.

Kotlin не различает примитивных типов и типов-оберток. Вы всегда используете один и тот же тип (например, `Int`):

```
val i: Int = 1
val list: List<Int> = listOf(1, 2, 3)
```

Это удобно. Более того, такой подход позволяет вызывать методы на значениях числовых типов. Для примера рассмотрим нижеследующий фрагмент – в нём используется функция `coerceIn` из стандартной библиотеки, ограничивающая значение указанным диапазоном:

```
fun showProgress(progress: Int) {
    val percent = progress.coerceIn(0, 100)
    println("We're ${percent}% done!")
}

>>> showProgress(146)
We're 100% done!
```

Если примитивные и ссылочные типы совпадают, значит ли это, что Kotlin представляет все числа как объекты? Будет ли это крайне неэффективно? Действительно, будет, поэтому Kotlin так не делает.

Во время выполнения числовые типы представлены наиболее эффективным способом. В большинстве случаев – для переменных, свойств, параметров и возвращаемых значений – тип `Int` в Kotlin компилируется в примитивный тип `int` из Java. Единственный случай, когда это невозможно, – обобщенные классы, например коллекции. Примитивный тип, указанный в качестве аргумента типа обобщенного класса, компилируется в соответствующий тип-обертку в Java. Поэтому если в качестве аргумента типа коллекции указан `Int`, то коллекция будет хранить экземпляры соответствующего типа-обертки `java.lang.Integer`.

Ниже приводится полный список типов, соответствующих примитивным типам Java:

- *целочисленные типы* – `Byte`, `Short`, `Int`, `Long`;
- *числовые типы с плавающей точкой* – `Float`, `Double`;
- *символьный тип* – `Char`;
- *логический тип* – `Boolean`.

Такие Kotlin-типы, как `Int`, за кулисами могут компилироваться в соответствующие примитивные типы Java, поскольку значения обоих типов не могут хранить ссылку на `null`. Обратное преобразование работает аналогично: при использовании Java-объявлений в Kotlin примитивные типы становятся типами, не допускающими `null` (а не платформенными типа-

ми), поскольку они не могут содержать значения null. Теперь рассмотрим те же типы, но уже допускающие значение null.

6.2.2. Примитивные типы с поддержкой null: Int?, Boolean? и прочие

Kotlin-типы с поддержкой null не могут быть представлены в Java как примитивные типы, поскольку в Java значение null может храниться только в переменных ссылочных типов. Это означает, что всякий раз, когда в коде на Kotlin используется версия простого типа, допускающая значение null, она компилируется в соответствующий тип-обертку.

Чтобы увидеть такие типы в действии, давайте вернемся к примеру в начале этой книги и вспомним объявление класса Person. Класс описывает человека, чье имя всегда известно и чей возраст может быть указан или нет. Добавим функцию, которая проверяет, является ли один человек старше другого.

Листинг 6.20. Работа с простыми типами, допускающими значение null

```
data class Person(val name: String,
        val age: Int? = null) {

    fun isOlderThan(other: Person): Boolean? {
      if (age == null || other.age == null)
        return null
      return age > other.age
    }
}

>>> println(Person("Sam", 35).isOlderThan(Person("Amy", 42)))
false
>>> println(Person("Sam", 35).isOlderThan(Person("Jane")))
null
```

Обратите внимание, что здесь применяются обычные правила работы со значением null. Нельзя просто взять и сравнить два значения типа Int?, поскольку одно из них может оказаться null. Вместо этого вам нужно убедиться, что оба значения не равны null, и после этого компилятор позволит работать с ними как обычно.

Значение свойства age, объявленного в классе Person, хранится как java.lang.Integer. Но эта деталь имеет значение только тогда, когда вы работаете с этим классом в Java. Чтобы выбрать правильный тип в Kotlin, нужно понять только то, допустимо ли присваивать значение null переменной или свойству.

Как уже упоминалось, обобщенные классы – это ещё одна ситуация, когда на сцену выходят оберточные типы. Если в качестве аргумента типа указан простой тип, Kotlin будет использовать обертку для данного типа. Например, следующее объявление создаст список значений типа `Integer`, даже если вы не указывали, что тип допускает значение `null`, и не использовали самого значения `null`:

```
val listOfInts = listOf(1, 2, 3)
```

Это происходит из-за способа реализации обобщенных классов в виртуальной машине Java. JVM не поддерживает использования примитивных типов в качестве аргументов типа, поэтому обобщенные классы (как в Java, так и в Kotlin) всегда должны использовать типы-обертки. Следовательно, когда требуется обеспечить эффективное хранение больших коллекций простых типов, то приходится использовать сторонние библиотеки (такие как Trove4J, http://trove.starlight-systems.com), поддерживающие такие коллекции, или хранить их в массивах. Мы подробно обсудим массивы в конце этой главы.

А теперь посмотрим, как преобразовывать значения между различными простыми типами.

6.2.3. Числовые преобразования

Одно важное отличие Kotlin от Java – способ обработки числовых преобразований. Kotlin не выполняет автоматического преобразования чисел из одного типа в другой, даже когда другой тип охватывает более широкий диапазон значений. Например, в Kotlin следующий код не будет компилироваться:

```
val i = 1
val l: Long = i          ◁— Ошибка: несоответствие типов
```

Вместо этого нужно применить явное преобразование:

```
val i = 1
val l: Long = i.toLong()
```

Функции преобразования определены для каждого простого типа (кроме `Boolean`): `toByte()`, `toShort()`, `toChar()` и т. д. Функции поддерживают преобразование в обоих направлениях: расширение меньшего типа к большему, как `Int.toLong()`, и усечение большего типа до меньшего, как `Long.toInt()`.

Kotlin делает преобразование явным, чтобы избежать неприятных неожиданностей, особенно при сравнении обернутых значений. Метод `equals` для двух обернутых значений проверяет тип обертки, а не только хранящееся в ней значение. Так, выражение `new Integer(42).equals(new`

Long(42)) в Java вернет false. Если бы Kotlin поддерживал неявные преобразования, вы могли бы написать что-то вроде этого:

```
val x = 1                    <— Переменная типа Int
val list = listOf(1L, 2L, 3L)      <— Список значений типа Long
x in list                    <— Если бы Kotlin поддерживал неявные преобразования типов, это выражение вернуло бы false
```

Вопреки всем ожиданиям эта функция вернула бы false. Поэтому строка x in list в этом примере не скомпилируется. Kotlin требует явного преобразования типов, поэтому сравнивать можно только значения одного типа:

```
>>> val x = 1
>>> println(x.toLong() in listOf(1L, 2L, 3L))
true
```

Если вы одновременно используете различные числовые типы в коде и хотите избежать неожиданного поведения – явно преобразуйте переменные.

Литералы примитивных типов

В дополнение к обычным десятичным числам Kotlin поддерживает следующие способы записи числовых литералов в исходном коде:

- Литералы типа Long обозначаются суффиксом L: 123L.
- В литералах типа Double используется стандартное представление чисел с плавающей точкой: 0.12, 2.0, 1.2e10, 1.2 e-10.
- Литералы типа float обозначаются суффиксом F или f: 123.4f, .456F, 1e3f.
- Литералы шестнадцатеричных чисел обозначаются префиксом 0x или 0X (например, 0xCAFEBABE или 0xbcdL).
- Двоичные литералы обозначаются префиксом 0b или 0B (например, 0b000000101).

Обратите внимание, что символ подчеркивания в числовых литералах начал поддерживаться только с версии Kotlin 1.1.

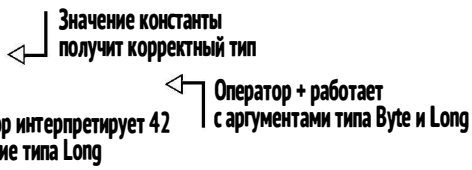
Для символьных литералов в основном используется тот же синтаксис, что и в Java. Символ записывается в одиночных кавычках, а если нужно, то можно использовать экранирование. Вот примеры допустимых символьных литералов в Kotlin: '1', '\t' (символ табуляции), '\u0009' (символ табуляции представлен как экранированная последовательность Юникода).

Обратите внимание, что при записи числового литерала обычно не нужно использовать функции преобразования. Допускается использовать специальный синтаксис для явного обозначения типа константы, 42L или 42.0f. Но даже если его не использовать, компилятор автоматически применит к числовому литералу необходимое преобразование для инициа-

лизации переменной известного типа или передачи его в качестве аргумента функции. Кроме того, арифметические операторы перегружены для всех соответствующих числовых типов. Например, следующий код работает правильно без каких-либо явных преобразований:

```
fun foo(l: Long) = println(l)

>>> val b: Byte = 1
>>> val l = b + 1L
>>> foo(42)
42
```



Обратите внимание, что при переполнении арифметические операторы в Kotlin действуют так же, как в Java, – Kotlin не привносит накладных расходов для проверки переполнения.

> **Преобразование строк**
>
> Стандартная библиотека Kotlin включает набор функций-расширений для преобразования строк в простые типы (`toInt`, `toByte`, `toBoolean` и т. д.):
>
> ```
> >>> println("42".toInt())
> 42
> ```
>
> Каждая из этих функций пытается проанализировать содержимое строки на соответствие нужному типу и вызывает `NumberFormatException`, если анализ завершается неудачей.

Прежде чем перейти к другим типам, отметим еще три специальных типа: `Any`, `Unit` и `Nothing`.

6.2.4. Корневые типы Any и Any?

Подобно тому, как тип `Object` является корнем иерархии классов в Java, тип `Any` – это супертип всех типов в Kotlin, не поддерживающих `null`. Но в Java тип `Object` – это супертип для всех ссылочных типов, а примитивные типы не являются частью его иерархии. Это означает, что когда требуется экземпляр `Object`, то для представления значений примитивных типов нужно использовать типы-обертки, такие как `java.lang.Integer`. В Kotlin тип `Any` – супертип для всех типов, в том числе для примитивных, таких как `Int`.

Так же как в Java, присваивание примитивного значения переменной типа `Any` вызывает автоматическую упаковку значения:

```
val answer: Any = 42
```

Обратите внимание, что тип Any не поддерживает значения null, поэтому переменная типа Any не может хранить null. Если нужна переменная, способная хранить любое допустимое значение в Kotlin, в том числе null, используйте тип Any?.

На уровне реализации тип Any соответствует типу java.lang.Object. Тип Object, используемый в параметрах и возвращаемых значениях методов Java, рассматривается в Kotlin как тип Any. (Точнее, он рассматривается как платформенный тип, поскольку допустимость значения null неизвестна.) Если функция Kotlin использует тип Any, в байт-коде этот тип компилируется в Java-тип Object.

Как было показано в главе 4, все классы в Kotlin имеют три метода: toString, equals и hashCode. Эти методы наследуются от класса Any. Другие методы, объявленные в классе java.lang.Object (например, wait и notify), недоступны в классе Any, но их можно вызвать, если вручную привести значение к типу java.lang.Object.

6.2.5. Тип Unit: тип «отсутствующего» значения

Тип Unit играет в Kotlin ту же роль, что и void в Java. Он может использоваться в качестве типа возвращаемого значения функции, которая не возвращает ничего интересного:

```
fun f(): Unit { ... }
```

Синтаксически это определение равноценно следующему, где отсутствует объявление типа тела функции:

```
fun f() { ... }      <┐ Явное объявление типа
                      │ Unit опущено
```

В большинстве случаев вы не заметите разницы между типами Unit и void. Если ваша Kotlin-функция объявлена как возвращающая тип Unit и она не переопределяет обобщенную функцию, она будет скомпилирована в старую добрую функцию void. Если вы переопределяете её в Java, то переопределяющая функция просто должна возвращать void.

Тогда чем тип Unit в Kotlin отличается от типа void в Java? В отличие от void, тип Unit – это полноценный тип, который может использоваться как аргумент типа. Существует только один экземпляр данного типа – он тоже называется Unit и возвращается *неявно*. Это полезно при переопределении функции с обобщенным параметром, чтобы заставить её возвращать значение типа Unit:

```
interface Processor<T> {
    fun process(): T
}

class NoResultProcessor : Processor<Unit> {
```

```
    override fun process() {
      // сделать что-то
    }
}
```

Возвращает значение типа Unit, но в объявлении это не указано

Не требуется писать инструкцию return

Сигнатура интерфейса требует, чтобы функция `process` возвращала значение, а поскольку тип `Unit` не имеет значения, вернуть его из метода не проблема. Но вам не нужно явно писать инструкцию `return` в функции `NoResultProcessor.process`, потому что `return Unit` неявно добавляется компилятором.

Сравните это с Java, где ни одно из решений проблемы указания отсутствия аргумента типа не выглядит так элегантно, как в Kotlin. Один из вариантов – использовать отдельные интерфейсы (такие как `Callable` и `Runnable`) для представления элементов, возвращающих и не возвращающих значения. Другой заключается в использовании специального типа `java.lang.Void` в качестве параметра типа. Если вы выбрали второй вариант, вам всё равно нужно добавить инструкцию `return null` для возвращения единственно возможного значения этого типа, поскольку если тип возвращаемого значения не `void`, то вы всегда должны использовать оператор `return`.

Вас может удивить, почему мы выбрали другое имя для `Unit` и не назвали его `Void`. Имя `Unit` традиционно используется в функциональных языках и означает «единственный экземпляр», а это именно то, что отличает тип `Unit` в Kotlin от `void` в Java. Мы могли бы использовать обычное имя `Void`, но в Kotlin есть тип `Nothing`, выполняющий совершенно другую функцию. Наличие двух типов с именами `Void` и `Nothing` («пустота» и «ничто») сбивало бы с толку, поскольку значения этих слов очень похожи. Так что же это за тип – `Nothing`? Давайте выясним.

6.2.6. Тип Nothing: функция, которая не завершается

Для некоторых функций в Kotlin понятие возвращаемого значения просто не имеет смысла, поскольку они никогда не возвращают управления. Например, во многих библиотеках для тестирования есть функция `fail`, которая генерирует исключение с указанным сообщением и заставляет текущий тест завершиться неудачей. Функция с бесконечным циклом также никогда не завершится.

При анализе кода, вызывающего такую функцию, полезно знать, что она не возвращает управления. Чтобы выразить это, в Kotlin используется специальный тип возвращаемого значения `Nothing`:

```
fun fail(message: String): Nothing {
    throw IllegalStateException(message)
}
```

```
>>> fail("Error occurred")
java.lang.IllegalStateException: Error occurred
```

Тип `Nothing` не имеет значений, поэтому его имеет смысл использовать только в качестве типа возвращаемого значения функции или аргумента типа для обозначения типа возвращаемого значения обобщенной функции. Во всех остальных случаях объявление переменной, в которой нельзя сохранить значение, не имеет смысла.

Обратите внимание, что функции, возвращающие `Nothing`, могут использоваться справа от оператора «Элвис» для проверки предусловий:

```
val address = company.address ?: fail("No address")
println(address.city)
```

Этот пример показывает, почему наличие `Nothing` в системе типов крайне полезно. Компилятор знает, что функция с таким типом не вернет управления, и использует эту информацию при анализе кода вызова функции. В предыдущем примере компилятор сообщит, что тип поля `address` не допускает значений `null` – потому что ветка, где значение равно `null`, всегда возбуждает исключение.

Мы закончили обсуждение основных типов в Kotlin: примитивных типов, `Any`, `Unit` и `Nothing`. Теперь рассмотрим типы коллекций и чем они отличаются от своих аналогов в Java.

6.3. Массивы и коллекции

Вы увидели примеры использования различных API для работы с коллекциями и знаете, что Kotlin использует библиотеку коллекций Java и дополняет её новыми возможностями через функции-расширения. Но впереди – рассказ о поддержке коллекций в языке Kotlin и соответствии между коллекциями в Java и Kotlin. Пришло время узнать подробности.

6.3.1. Коллекции и допустимость значения null

Ранее в этой главе мы обсудили типы с поддержкой `null`, но при этом лишь вкратце затронули допустимость `null` для аргументов типов. Но для согласованной системы типов это ключевой пункт: не менее важно знать, может ли коллекция хранить значения `null`, чем знать, может ли переменная хранить `null`. К счастью, Kotlin позволяет указать допустимость `null` в аргументах типов. Как имя типа переменной может заканчиваться символом `?`, свидетельствующим о допустимости значения `null`, так и аргумент типа может быть помечен таким образом. Чтобы понять суть, рассмотрим функцию, которая читает строки из файла и пытается представить каждую строку как число.

Листинг 6.21. Создание коллекции, которая может хранить значения `null`

```
fun readNumbers(reader: BufferedReader): List<Int?> {
    val result = ArrayList<Int?>()
    for (line in reader.lineSequence()) {
      try {
        val number = line.toInt()ч
        result.add(number)
      }
      catch(e: NumberFormatException) {
        result.add(null)
      }
    }
    return result
}
```

Создание списка значений типа Int с поддержкой null

Добавление в список целочисленного значения (не равного null)

Добавление значения null в список, поскольку текущая строка не может быть преобразована в число

Список типа `List<Int?>` может хранить значения типа `Int?` – другими словами, `Int` или `null`. Если строка может быть преобразована в число, мы добавляем в список `result` целое число, а в противном случае – `null`. Заметьте, что начиная с Kotlin 1.1 этот пример можно сократить, используя функцию `String.toIntOrNull`, – она возвращает `null`, если строковое значение не может быть преобразовано в число.

Обратите внимание, что допустимость значения `null` для самой переменной отличается от допустимости значения `null` для типа, который используется в качестве аргумента типа. Разница между списком, поддерживающим элементы типа `Int` и `null`, и списком элементов `Int`, который сам может оказаться пустой ссылкой `null`, показана на рис. 6.10.

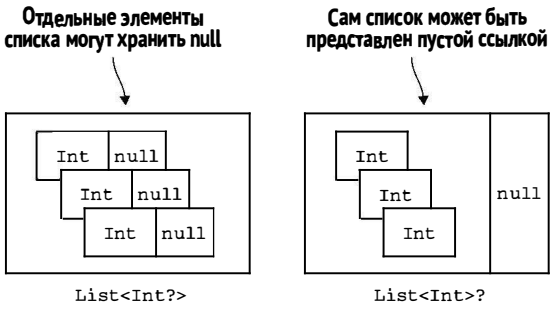


Рис. 6.10. Помните, для каких объектов допускается значение `null` – для элементов или для самой коллекции

В первом случае сам список не может оказаться пустой ссылкой, но его элементы могут хранить `null`. Переменная второго типа может содержать значение `null` вместо ссылки на экземпляр списка, но элементы в списке гарантированно не будут хранить `null`.

В другом контексте вы можете захотеть объявить переменную, содержащую пустую ссылку на список с элементами, которые могут иметь значе-

ния null. В Kotlin такой тип записывается с двумя вопросительными знаками: List<Int?>?. При этом вам придется выполнять проверки на null два раза: и при использовании значения переменной, и при использовании значения каждого элемента в списке.

Чтобы понять, как работать со списком элементов, способных хранить null, напишем функцию для сложения всех корректных чисел и подсчета некорректных чисел отдельно.

Листинг 6.22. Работа с коллекцией, которая может хранить значения null

```
fun addValidNumbers(numbers: List<Int?>) {
    var sumOfValidNumbers = 0
    var invalidNumbers = 0
    for (number in numbers) {
      if (number != null) {
        sumOfValidNumbers += number
      } else {
        invalidNumbers++
      }
    }
    println("Sum of valid numbers: $sumOfValidNumbers")
    println("Invalid numbers: $invalidNumbers")
}
```

Чтение из списка значения, которое может оказаться равным null

Проверка значения на null

```
>>> val reader = BufferedReader(StringReader("1\nabc\n42"))
>>> val numbers = readNumbers(reader)
>>> addValidNumbers(numbers)
Sum of valid numbers: 43
Invalid numbers: 1
```

Здесь нет ничего особенного. Обращаясь к элементу списка, вы получаете значение типа Int?, и прежде чем использовать его в арифметических операциях, его необходимо проверить на null.

Обработка коллекции значений, которые могут быть равны null, и последующая фильтрация таких элементов – очень распространенная операция. Поэтому для её выполнения в Kotlin есть стандартная функция filterNotNull. Вот как использовать её для упрощения предыдущего примера.

Листинг 6.23. Применение функции filterNotNull к коллекции, которая может хранить значения null

```
fun addValidNumbers(numbers: List<Int?>) {
    val validNumbers = numbers.filterNotNull()
    println("Sum of valid numbers: ${validNumbers.sum()}")
```

```
    println("Invalid numbers: ${numbers.size - validNumbers.size}")
}
```

Конечно, фильтрация тоже влияет на тип коллекции. Коллекция `validNumbers` имеет тип `List<Int>`, потому что фильтрация гарантирует, что коллекция не будет содержать значений `null`.

Теперь, зная, как Kotlin различает коллекции, которые могут или не могут содержать элементы `null`, рассмотрим другие важные отличия языка Kotlin: коллекции с доступом только для чтения и изменяемые коллекции.

6.3.2. Изменяемые и неизменяемые коллекции

Важная черта, отличающая коллекции в Kotlin от коллекций в Java, – разделение интерфейсов, открывающих доступ к данным в коллекции только для чтения и для изменения. Это разделение начинается с базового интерфейса коллекций – `kotlin.collections.Collection`. С помощью этого интерфейса можно выполнить обход элементов коллекции, узнать её размер, проверить наличие определенного элемента и выполнить другие операции чтения данных из коллекции. Но в этом интерфейсе отсутствуют методы добавления или удаления элементов.

Чтобы получить возможность изменения данных в коллекции, используйте интерфейс `kotlin.collections.MutableCollection`. Он наследует интерфейс `kotlin.collections.Collection`, добавляя к нему методы для добавления и удаления элементов, очистки коллекции и т. д. На рис. 6.11 показаны основные методы, присутствующие в этих двух интерфейсах.

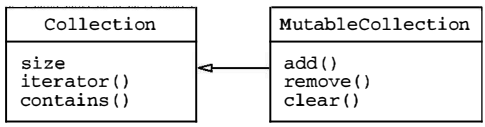


Рис. 6.11. Интерфейс MutableCollection наследует Collection и добавляет методы для изменения содержимого коллекции

Как правило, вы везде должны использовать интерфейсы с доступом только для чтения. Применяйте изменяемые варианты, только если собираетесь изменять коллекцию.

Такое разделение интерфейсов позволяет быстрее понять, что происходит с данными в программе. Если функция принимает параметр типа `Collection`, но не `MutableCollection`, можно быть уверенным, что она не изменяет коллекцию и будет только читать данные из нее. Но если функция требует передачи аргумента `MutableCollection`, можно предположить, что она собирается изменять данные. Если у вас есть коллекция, которая хранит внутреннее состояние вашего компонента, то вам может понадобиться сделать копию этой коллекции перед передачей в такую функцию (этот шаблон обычно называют *защитным копированием*).

Например, в следующем примере видно, что функция `copyElements` будет изменять целевую коллекцию, но не исходную.

Листинг 6.24. Применение интерфейсов для чтения и изменения значений коллекции

```
fun <T> copyElements(source: Collection<T>,
        target: MutableCollection<T>) {
    for (item in source) {
      target.add(item)
    }
}
```

Цикл по всем элементам исходной коллекции

Добавление элементов в изменяемую целевую коллекцию

```
>>> val source: Collection<Int> = arrayListOf(3, 5, 7)
>>> val target: MutableCollection<Int> = arrayListOf(1)
>>> copyElements(source, target)
>>> println(target)
[1, 3, 5, 7]
```

Вы не сможете передать в аргументе `target` переменную с типом коллекции, доступной только для чтения, даже если она ссылается на изменяемую коллекцию:

```
>>> val source: Collection<Int> = arrayListOf(3, 5, 7)
>>> val target: Collection<Int> = arrayListOf(1)
>>> copyElements(source, target)
Error: Type mismatch: inferred type is Collection<Int>
but MutableCollection<Int> was expected
```

Ошибочный аргумент «target»

Самое главное, что нужно помнить при работе с интерфейсами коллекций, доступных только для чтения, – они *необязательно неизменяемы*[1]. Если вы работаете с переменной-коллекцией, интерфейс которой дает доступ только для чтения, она может оказаться лишь одной из нескольких ссылок на одну и ту же коллекцию. Другие ссылки могут иметь тип изменяемого интерфейса, как показано на рис. 6.12.

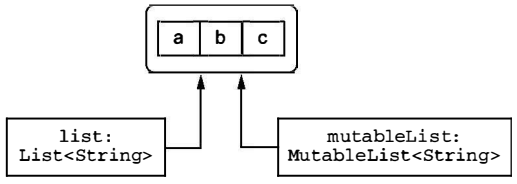


Рис. 6.12. Две разные ссылки: одна – с доступом только для чтения и другая, разрешающая изменение, – указывают на одну и ту же коллекцию

Вызывая код, хранящий другую ссылку на вашу коллекцию, или запуская его параллельно, вы все ещё можете столкнуться с ситуацией, когда коллекция меняется под воздействием другого кода, пока вы работаете с ней. Это

[1] Позже планируется добавить неизменяемые коллекции в стандартную библиотеку Kotlin.

приводит к появлению исключения `ConcurrentModificationException` и другим проблемам. Поэтому важно понимать, что *коллекции с доступом только для чтения не всегда потокобезопасны*. Если вы работаете с данными в многопоточной среде, убедитесь, что ваш код правильно синхронизирует доступ к данным или использует структуры данных, поддерживающие одновременный доступ.

Так как же происходит разделение на коллекции только для чтения и коллекции с доступом для чтения/записи? Разве мы не говорили ранее, что коллекции в Kotlin ничем не отличаются от коллекций в Java? Нет ли здесь противоречия? Давайте выясним, что происходит на самом деле.

6.3.3. Коллекции Kotlin и язык Java

Действительно, любая коллекция в Kotlin – экземпляр соответствующего интерфейса коллекции Java. При переходе между Kotlin и Java никакого преобразования не происходит, и нет необходимости ни в создании оберток, ни в копировании данных. Но для каждого интерфейса Java-коллекций в Kotlin существуют два представления: только для чтения и для чтения/записи, как можно видеть на рис. 6.13.

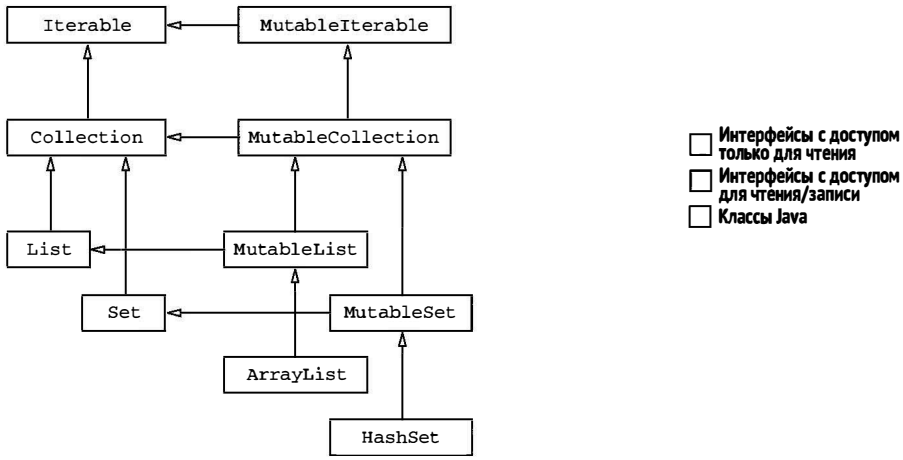


Рис. 6.13. Иерархия интерфейсов коллекций Kotlin. Java-классы ArrayList и HashSet наследуют интерфейсы изменяемых коллекций в Kotlin

Все интерфейсы коллекций на рис. 6.13 объявлены в Kotlin. Базовая структура интерфейсов для чтения и изменения коллекций в Kotlin выстроена параллельно иерархии интерфейсов Java-коллекций в пакете `java.util`. Кроме того, каждый интерфейс, меняющий коллекцию, наследует соответствующий интерфейс только для чтения. Интерфейсы коллекций с доступом для чтения/записи непосредственно соответствуют Java-интерфейсам в пакете `java.util`, тогда как интерфейсы только для чтения лишены любых методов, которые могли бы изменить коллекцию.

Для демонстрации того, как стандартные классы Java выглядят с точки зрения Kotlin, на рис. 6.13 также показаны коллекции `java.util.ArrayList` и `java.util.HashSet`. Язык Kotlin рассматривает их так, словно они наследуют интерфейсы `MutableList` и `MutableSet` соответственно. Здесь не представлены другие реализации из Java-библиотеки (`LinkedList`, `SortedSet` и т. д.), но с точки зрения Kotlin они имеют схожие супертипы. Таким образом, вы получаете не только совместимость, но так же четкое разделение интерфейсов с доступом только для чтения и для чтения/записи.

В дополнение к коллекциям в Kotlin есть класс `Map` (который не наследует ни `Collection`, ни `Iterable`) в двух различных вариантах: `Map` и `MutableMap`. В табл. 6.1 показаны функции, которые можно использовать для создания коллекций различных типов.

Таблица 6.1. Функции для создания коллекций

Тип коллекции	Только для чтения	Изменяемая коллекция
List	`listOf`	`mutableListOf, arrayListOf`
Set	`setOf`	`mutableSetOf, hashSetOf, linkedSetOf, sortedSetOf`
Map	`mapOf`	`mutableMapOf, hashMapOf, linkedMapOf, sortedMapOf`

Обратите внимание, что функции `setOf()` и `mapOf()` возвращают экземпляры классов из стандартной библиотеки Java (по крайней мере, в Kotlin 1.0), которые на самом деле изменяемы[2]. Но не стоит полагаться на это: вполне возможно, что в будущих версиях Kotlin функции `setOf` и `mapOf` будут использовать по-настоящему неизменяемые реализации классов.

Когда нужно вызвать метод Java и передать ему коллекцию, это можно сделать непосредственно, без каких-либо промежуточных шагов. Например, если у вас есть метод Java, принимающий экземпляр `java.util.Collection`, вы можете передать в нем любой объект `Collection` или `MutableCollection`.

Это сильно влияет на изменяемость коллекций. Поскольку в Java нет различий между коллекциями только для чтения и для чтения/записи, Java-код *сможет изменить коллекцию*, даже если в Kotlin она объявлена как доступная только для чтения. Компилятор Kotlin не в состоянии полностью проанализировать, что происходит с коллекцией на стороне Java-кода, и не имеет никакой возможности отклонить передачу коллекции, доступной только для чтения, в модифицирующий её код на Java. Например, следующие два фрагмента образуют компилируемую многоязыковую программу Java/Kotlin:

[2] Обертывание коллекций типом `Collection.unmodifiable` приводит к накладным расходам, поэтому не используется.

```
/* Java */
// CollectionUtils.java
public class CollectionUtils {
    public static List<String> uppercaseAll(List<String> items) {
      for (int i = 0; i < items.size(); i++) {
        items.set(i, items.get(i).toUpperCase());
      }
      return items;
    }
}
```

```
// Kotlin
// collections.kt
fun printInUppercase(list: List<String>) {
    println(CollectionUtils.uppercaseAll(list))
    println(list.first())
}
```

Объявление параметра как доступного только для чтения

Вызов Java-функции, которая изменяет коллекцию

Показывает, что коллекция изменилась

```
>>> val list = listOf("a", "b", "c")
>>> printInUppercase(list)
[A, B, C]
A
```

Поэтому, если вы пишете функцию Kotlin, которая принимает коллекцию и передает её Java-коду, *только вы отвечаете за объявление правильного типа параметра* в зависимости от того, изменяет вызываемый Java-код коллекцию или нет.

Обратите внимание, что этот нюанс также касается коллекций, элементы которых не могут содержать `null`. Если передать такую коллекцию в Java-метод, он вполне может поместить в неё значение `null` – Kotlin не может запретить или просто обнаружить такую ситуацию без ущерба для производительности. Поэтому, передавая коллекции в Java-код, который может изменить их, принимайте меры предосторожности – тогда типы Kotlin правильно отразят все возможности изменения коллекции.

Теперь внимательнее посмотрим, как Kotlin работает с коллекциями, объявленными в Java-коде.

6.3.4. Коллекции как платформенные типы

Вернувшись к обсуждению значения `null` в начале этой главы, вы наверняка вспомните, что типы, объявленные в Java-коде, рассматриваются в Kotlin как *платформенные типы*. Для платформенных типов Kotlin не имеет информации о поддержке значения `null`, поэтому компилятор позволяет обращаться с ними как с поддерживающими или не поддерживающими `null`. Таким же образом типы переменных и коллекций, объявленные в Java, в языке Kotlin рассматриваются как платформенные типы.

Коллекция платформенного типа фактически представляет собой коллекцию с неизвестным статусом изменяемости – код на Kotlin может считать её доступной для чтения/записи или только для чтения. Обычно это неважно, поскольку все операции, которые вы можете захотеть выполнить, просто работают.

Разница становится важной при переопределении или реализации метода Java, в сигнатуре которого есть тип коллекции. Как и в случае с поддержкой `null` для платформенных типов, здесь только вы решаете, какой тип Kotlin использовать для представления типа Java в переопределяемом или реализуемом методе.

В этом случае вам нужно принять несколько решений, каждое из которых отразится на типе параметра в Kotlin:

- Может ли сама коллекция оказаться пустой ссылкой `null`?
- Может ли хотя бы один из ее элементов оказаться значением `null`?
- Будет ли ваш метод изменять коллекцию?

Чтобы понять разницу, рассмотрим следующие случаи. В первом примере интерфейс Java представляет объект, обрабатывающий текст в файле.

Листинг 6.25. Интерфейс Java с коллекцией в качестве параметра

```
/* Java */
interface FileContentProcessor {
    void processContents(File path,
      byte[] binaryContents,
      List<String> textContents);
}
```

В Kotlin-реализации этого интерфейса нужно принять во внимание следующие соображения:

- Ссылка на список может оказаться пустой, поскольку существуют двоичные файлы, которые нельзя представить в виде текста.
- Элементы в списке не могут хранить `null`, поскольку строки в файле никогда не имеют значения `null`.
- Список доступен только для чтения, поскольку представляет неизменяемое содержимое файла.

Вот как выглядит реализация:

Листинг 6.26. Реализация интерфейса `FileContentProcessor` в Kotlin

```
class FileIndexer : FileContentProcessor {
    override fun processContents(path: File,
      binaryContents: ByteArray?,
```

```
      textContents: List<String>?) {

      // ...
    }
}
```

Сравните её с другим интерфейсом. Здесь реализация интерфейса предусматривает преобразование некоторых данных из текстовой формы в список объектов (с добавлением новых объектов в выходной список), сообщает об ошибках, обнаруженных при анализе, и добавляет текст сообщений в отдельный список.

Листинг 6.27. Другой интерфейс Java с коллекцией в качестве параметра

```
/* Java */
interface DataParser<T> {
    void parseData(String input,
      List<T> output,
      List<String> errors);
}
```

Здесь нужно учесть другие соображения:

- Список `List<String>` не может быть пустой ссылкой, поскольку вызывающий код всегда должен получать сообщения об ошибках.
- Среди элементов списка может оказаться значение `null`, поскольку не все элементы в выходном списке будут иметь связанные с ними сообщения об ошибках.
- Список `List<String>` будет изменяемым, поскольку реализация должна добавлять в него элементы.

Вот как можно реализовать такой интерфейс в Kotlin.

Листинг 6.28. Реализация интерфейса DataParser на языке Kotlin

```
class PersonParser : DataParser<Person> {
    override fun parseData(input: String,
      output: MutableList<Person>,
      errors: MutableList<String?>) {

      // ...
    }
}
```

Обратите внимание, как один и тот же Java-тип `List<String>` представлен в Kotlin двумя различными типами: `List<String>?` (список строк, который может быть представлен пустой ссылкой) в одном случае и

MutableList<String?> (изменяемый список строк с возможностью хранения null) в другом. Чтобы сделать правильный выбор, нужно точно знать контракт, которому должен следовать интерфейс или класс Java. Обычно это легко понять из назначения вашей реализации.

Теперь, когда мы разобрались с коллекциями, пришло время взглянуть на массивы. Как уже говорилось, по умолчанию стоит предпочитать коллекции, а не массивы. Но поскольку многие интерфейсы Java API по-прежнему используют массивы, то полезно знать, как работать с ними в Kotlin.

6.3.5. Массивы объектов и примитивных типов

Поскольку массив – это часть сигнатуры Java-функции main, то синтаксис объявления массивов в Kotlin встречается в каждом примере. Напомним, как он выглядит:

Листинг 6.29. Использование массивов

```
fun main(args: Array<String>) {
    for (i in args.indices) {
      println("Argument $i is: ${args[i]}")
    }
}
```

Использование свойства-расширения array.indices для обхода диапазона индексов

Обращение к элементу по индексу array[index]

Массив в Kotlin – это класс с параметром типа, где тип элемента определяется аргументом типа.

Создать массив в Kotlin можно следующими способами:

- Функция arrayOf создает массив с элементами, соответствующими аргументам функции.
- Функция arrayOfNulls создает массив заданного размера, где все элементы равны null. Конечно, эту функцию можно использовать лишь для создания массивов, допускающих хранение null в элементах.
- Конструктор класса Array принимает размер массива и лямбда-выражение, после чего инициализирует каждый элемент с помощью этого лямбда-выражения. Так можно инициализировать массив, который не поддерживает значения null в элементах, не передавая всех элементов непосредственно.

В качестве простого примера воспользуемся функцией Array и создадим массив строк от "a" до "z".

Листинг 6.30. Создание массива строк

```
>>> val letters = Array<String>(26) { i -> ('a' + i).toString() }
```

```
>>> println(letters.joinToString(""))
abcdefghijklmnopqrstuvwxyz
```

Лямбда-выражение принимает индекс элемента массива и возвращает значение, которое будет помещено в массив с этим индексом. Здесь значение вычисляется путем сложения индекса с кодом символа `"a"` и преобразованием результата в строку. Тип элемента массива показан для ясности – в реальном коде его можно опустить, поскольку компилятор определит его самостоятельно.

Одна из самых распространенных причин создания массивов в Kotlin – необходимость вызова метода Java, принимающего массив, или функции Kotlin с параметром типа `vararg`. Чаще всего в таких случаях данные уже хранятся в коллекции, и вам просто нужно преобразовать их в массив. Сделать это можно с помощью метода `toTypedArray`.

Листинг 6.31. Передача коллекции в метод, принимающий `vararg`

```
>>> val strings = listOf("a", "b", "c")
>>> println("%s/%s/%s".format(*strings.toTypedArray()))
a/b/c
```

Для передачи массива в метод, ожидающий vararg, применяется оператор развертывания (*)

Как и с другими типами, *аргумент типа в типе массива всегда становится ссылочным типом*. Поэтому объявление `Array<Int>`, например, превратится в массив оберток для целых чисел (Java-тип `java.lang.Integer[]`). Если вам нужно создать массив значений примитивного типа без оберток, используйте один из специализированных классов для представления массивов примитивных типов.

Для этой цели в Kotlin есть ряд отдельных классов, по одному для каждого примитивного типа. Например, класс, соответствующий массиву значений типа `Int`, называется `IntArray`. Также существуют классы `ByteArray`, `CharArray`, `CharArray` и другие. Все эти типы компилируются в обычные массивы примитивных Java-типов, таких как `int[]`, `byte[]`, `char[]` и т. д. Следовательно, значения в таких массивах не заворачиваются в объекты и хранятся наиболее эффективным способом.

Создать массив примитивного типа можно следующими способами:

- Конструктор типа принимает параметр `size` и возвращает массив, инициализированный значениями по умолчанию для данного типа (обычно нулями).
- Фабричная функция (`intArrayOf` – для массива `IntArray` и аналогичные для остальных типов) принимает переменное число аргументов и создает массив из этих аргументов.
- Другой конструктор принимает значение размера и лямбда-выражение для инициализации каждого элемента.

Вот как можно воспользоваться первыми двумя способами для создания массива целых чисел, состоящего из пяти нулей:

```
val fiveZeros = IntArray(5)
val fiveZerosToo = intArrayOf(0, 0, 0, 0, 0)
```

А вот как использовать конструктор, принимающий лямбда-выражение:

```
>>> val squares = IntArray(5) { i -> (i+1) * (i+1) }
>>> println(squares.joinToString())
1, 4, 9, 16, 25
```

Кроме того, существующие массив или коллекцию, хранящие обёртки для значений примитивного типа, можно преобразовать в массив примитивного типа с помощью соответствующей функции – например, `toIntArray`.

Теперь посмотрим, что можно сделать с массивами. Кроме основных операций (получения длины массива, чтения и изменения элементов), стандартная библиотека Kotlin поддерживает для массивов тот же набор функций-расширений, что и для коллекций. Все функции, которые вы видели в главе 5 (`filter`, `map` и т. д.), тоже применимы к массивам, включая массивы примитивных типов. (Заметим, что эти функции возвращают списки, а не массивы.)

Давайте посмотрим, как переписать листинг 6.29, используя функцию `forEachIndexed` и лямбда-выражение, которое вызывается для каждого элемента массива и получает два аргумента: индекс элемента и сам элемент.

Листинг 6.32. Применение функции `forEachIndexed` к массиву

```
fun main(args: Array<String>) {
    args.forEachIndexed { index, element ->
      println("Argument $index is: $element")
    }
}
```

Теперь вы знаете, как использовать массивы в своих программах. Работать с ними в Kotlin так же просто, как с коллекциями.

6.4. Резюме

- Управление поддержкой `null` в языке Kotlin помогает выявить возможные исключения `NullPointerException` на этапе компиляции.
- Для работы со значением `null` в Kotlin есть специальные инструменты: оператор безопасного вызова (`?.`), оператор «Элвис» (`?:`), утверждение, что значение не равно `null` (`!!`), и функция `let`.

- Оператор `as?` позволяет привести значения к типу и обрабатывать случаи, когда оно имеет несовместимый тип.
- Типы, пришедшие из Java, интерпретируются в Kotlin как платформенные типы, что позволяет разработчику относиться к ним как к типам с поддержкой или без поддержки `null`.
- Типы, представляющие обычные числа (например, `Int`), выглядят и функционируют как рядовые классы, но обычно компилируются в простые типы Java.
- Простые типы с поддержкой `null` (такие как `Int?`) соответствуют оберткам простых типов в Java (таким как `java.lang.Integer`).
- Тип `Any` – это супертип всех других типов и аналог типа `Object` в Java. А тип `Unit` – аналог `void`.
- Тип `Nothing` используется в качестве типа возвращаемого значения для функций, которые в обычном режиме не завершаются.
- Для представления коллекций Kotlin использует стандартные классы Java, но делит их на доступные только для чтения и для чтения/записи.
- При наследовании Java-классов и реализации Java-интерфейсов в Kotlin нужно обращать пристальное внимание на возможность изменения и допустимость значения `null`.
- Класс `Array` в Kotlin выглядит как обычный обобщенный класс, но компилируется в Java-массив.
- Массивы простых типов представлены специальными классами, такими как `IntArray`.

Часть 2

Непростой Kotlin

Теперь вы должны иметь достаточно полное представление о приемах использования существующих API из Kotlin. В этой части книги вы узнаете, как создавать свои API на Kotlin. Имейте в виду, что эта тема касается не только разработчиков библиотек: всякий раз, когда вам в вашей программе потребуются два взаимодействующих класса, как минимум один из них должен будет предоставить свой API другому.

В главе 7 вы познакомитесь с соглашениями, которые используются в Kotlin при реализации перегруженных операторов, и с другими абстракциями, такими как делегированные свойства. В главе 8 во всех деталях рассматриваются лямбда-выражения; в ней вы увидите, как объявлять свои функции, принимающие лямбда-выражения в параметрах. Затем вы познакомитесь с особенностями реализации в Kotlin более продвинутых понятий Java, таких как обобщенные типы (глава 9), и научитесь применять аннотации и механизм рефлексии (глава 10). Также в главе 10 вы рассмотрите довольно большой проект на языке Kotlin: JKid – библиотеку сериализации/десериализации формата JSON. И в заключение, в главе 11, мы увидим один из самых драгоценных бриллиантов в короне Kotlin: его поддержку создания предметно-ориентированных языков.

Глава 7

Перегрузка операторов и другие соглашения

В этой главе объясняются:

- перегрузка операторов;
- соглашения: функции со специальными именами для поддержки различных операций;
- делегирование свойств.

Как известно, некоторые особенности языка Java тесно связаны с определенными классами в стандартной библиотеке. Например, объекты, реализующие интерфейс `java.lang.Iterable`, можно использовать в циклах `for`, а объекты, реализующие интерфейс `java.lang.AutoCloseable`, можно использовать в инструкциях `try-with-resources`.

В Kotlin есть похожие особенности: некоторые конструкции языка вызывают функции, определяемые в вашем коде. Но в Kotlin они связаны не с определенными типами, а с функциями, имеющими специальные имена. Например, если ваш класс определяет метод со специальным именем `plus`, то к экземплярам этого класса может применяться оператор +. Такой подход в Kotlin называется *соглашениями*. В этой главе мы познакомимся с разными соглашениями, принятыми в языке Kotlin, и посмотрим, как они используются на практике.

Вместо опоры на типы, как это принято в Java, в Kotlin действует принцип следования соглашениям, потому что это позволяет разработчикам адаптировать существующие Java-классы к требованиям возможностей языка Kotlin. Множество интерфейсов, реализованных Java-классами, фиксировано, и Kotlin не может изменять существующих классов, чтобы добавить в них реализацию дополнительных интерфейсов. С другой стороны, благодаря механизму функций-расширений есть возможность

добавлять новые методы в классы. Вы можете определить любой метод, описываемый соглашениями, как расширение и тем самым адаптировать любой существующий Java-класс без изменения его кода.

В этой главе в роли примера будет простой класс `Point`, представляющий точку на экране. Подобные классы доступны в большинстве фреймворков, предназначенных для разработки пользовательского интерфейса, и вы легко сможете перенести изменения, продемонстрированные ниже, в свое программное окружение:

```
data class Point(val x: Int, val y: Int)
```

Начнем с определения арифметических операторов в классе `Point`.

7.1. Перегрузка арифметических операторов

Арифметические операторы – самый простой пример использования соглашений в Kotlin. В Java полный набор арифметических операций поддерживается только для примитивных типов, а кроме того, оператор + можно использовать со значениями типа `String`. Но эти операции могли бы стать удобным подспорьем в других случаях. Например, при работе с числами типа `BigInteger` намного элегантнее использовать + для сложения, чем явно вызывать метод `add`. Для добавления элемента в коллекцию удобно было использовать оператор +=. Kotlin позволяет реализовать это, и в данном разделе мы покажем, как.

7.1.1. Перегрузка бинарных арифметических операций

Для начала реализуем операцию сложения двух точек. Она вычисляет суммы координат X и Y точки. Вот как выглядит эта реализация.

Листинг 7.1. Определение оператора `plus`

```
data class Point(val x: Int, val y: Int) {
    operator fun plus(other: Point): Point {
        return Point(x + other.x, y + other.y)
    }
}
```

Определение функции с именем «plus», реализующей оператор

Складывает координаты и возвращает новую точку

```
>>> val p1 = Point(10, 20)
>>> val p2 = Point(30, 40)
>>> println(p1 + p2)
Point(x=40, y=60)
```

Вызов функции «plus» путем использования оператора +

Обратите внимание на ключевое слово `operator` в объявлении функции `plus`. Все функции, реализующие перегрузку операторов, обязательно должны отмечаться этим ключевым словом. Оно явно сообщает, что вы

намереваетесь использовать эту функцию в соответствии с соглашениями и неслучайно выбрали такое имя.

После объявления функции `plus` с модификатором `operator` становится возможно складывать объекты, используя оператор +. За кулисами на место этого оператора компилятор будет вставлять вызов функции `plus`, как показано на рис. 7.1.

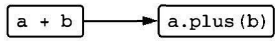


Рис. 7.1. Оператор + транслируется в вызов функции plus

Объявить оператор можно не только как функцию-член, но также как функцию-расширение.

Листинг 7.2. Определение оператора в виде функции-расширения

```
operator fun Point.plus(other: Point): Point {
    return Point(x + other.x, y + other.y)
}
```

При этом реализация осталась прежней. В следующих примерах мы будем использовать синтаксис функций-расширений – это более распространенный шаблон реализации соглашений для классов во внешних библиотеках, и этот синтаксис с успехом можно применять и для своих классов.

Определение и использование перегруженных операторов в Kotlin выглядят проще, чем в других языках, потому что в нём отсутствует возможность определять свои, нестандартные операторы. Kotlin ограничивает набор операторов, доступных для перегрузки. Каждому такому оператору соответствует имя функции, которую нужно определить в своем классе. В табл. 7.1 перечислены все бинарные операторы, доступные для перегрузки, и соответствующие им имена функций.

Таблица 7.1. Бинарные арифметические операторы, доступные для перегрузки

Выражение	Имя функции
`a * b`	`times`
`a / b`	`div`
`a % b`	`mod`
`a + b`	`plus`
`a - b`	`minus`

Операторы для любых типов следуют тем же правилам приоритета, что и для стандартных числовых типов. Например, в выражении `a + b * c` умножение всегда будет выполняться перед сложением, даже если это ваши собственные версии операторов. Операторы *, / и % имеют одинаковый приоритет, который выше приоритета операторов + и -.

Функции-операторы и Java

Перегруженные операторы Kotlin вполне доступны в Java-коде: так как каждый перегруженный оператор определяется как функция, их можно вызывать как обычные функции, используя полные имена. Вызывая Java-код из Kotlin, можно использовать синтаксис операторов для любых методов с именами, совпадающими с соглашениями в Kotlin. Так как в Java отсутствует синтаксис, позволяющий отметить функции-операторы, требование использовать модификатор `operator` к Java-коду не применяется, и в учет принимаются только имена и количество параметров. Если Java-класс определяет метод с требуемым поведением, но под другим именем, вы можете создать функцию-расширение с правильным именем, делегирующую выполнение операции существующему Java-методу.

Определяя оператор, необязательно использовать одинаковые типы для операндов. Например, давайте определим оператор, позволяющий масштабировать точку в определенное число раз. Он может пригодиться для представления точки в разных системах координат.

Листинг 7.3. Определение оператора с операндами разных типов

```
operator fun Point.times(scale: Double): Point {
    return Point((x * scale).toInt(), (y * scale).toInt())
}

>>> val p = Point(10, 20)
>>> println(p * 1.5)
Point(x=15, y=30)
```

Обратите внимание, что операторы в Kotlin не поддерживают *коммутативность* (перемену операндов местами) по умолчанию. Если необходимо дать пользователям возможность использовать выражения вида `1.5 * p` в дополнение к `p * 1.5`, следует определить отдельный оператор: `operator fun Double.times(p: Point): Point`.

Тип значения, возвращаемого функцией-оператором, также может отличаться от типов операндов. Например, можно определить оператор, создающий строку путем повторения заданного символа указанное число раз.

Листинг 7.4. Определение оператора с отличающимся типом результата

```
operator fun Char.times(count: Int): String {
    return toString().repeat(count)
}

>>> println('a' * 3)
aaa
```

Этот оператор принимает левый операнд типа `Char`, правый операнд типа `Int` и возвращает результат типа `String`. Такое сочетание типов операндов и результата вполне допустимо.

Заметьте также: подобно любым другим функциям, функции-операторы могут иметь перегруженные версии. Можно определить несколько методов с одним именем, отличающихся только типами параметров.

Отсутствие специальных операторов для поразрядных операций

В Kotlin отсутствуют любые поразрядные (битовые) операторы для стандартных числовых типов – и вследствие этого отсутствует возможность определять их для своих типов. Для этих целей используются обычные функции, поддерживающие инфиксный синтаксис вызова. Вы также можете определить одноименные функции для работы со своими типами.

Вот полный список функций, поддерживаемых в Kotlin для выполнения поразрядных операций:

- `shl` – сдвиг влево со знаком;
- `shr` – сдвиг вправо со знаком;
- `ushr` – сдвиг вправо без знака;
- `and` – поразрядное «И»;
- `or` – поразрядное «ИЛИ»;
- `xor` – поразрядное «ИСКЛЮЧАЮЩЕЕ ИЛИ»;
- `inv` – поразрядная инверсия.

Следующий пример демонстрирует использование некоторых из этих функций:

```
>>> println(0x0F and 0xF0)
0
>>> println(0x0F or 0xF0)
255
>>> println(0x1 shl 4)
16
```

Теперь перейдем к обсуждению операторов, которые совмещают в себе два действия: арифметическую операцию и присваивание (таких как +=).

7.1.2. Перегрузка составных операторов присваивания

Обычно, когда определяется оператор `plus`, Kotlin начинает поддерживать не только операцию +, но и +=. Операторы +=, -= и другие называют *составными операторами присваивания*. Например:

```
>>> var point = Point(1, 2)
>>> point += Point(3, 4)
>>> println(point)
Point(x=4, y=6)
```

Это выражение действует точно так же, как `point = point + Point(3, 4)`. Конечно, такое возможно только тогда, когда переменная изменяема.

В некоторых случаях имеет смысл определить операцию +=, которая могла бы изменить объект, на который ссылается переменная, участвующая в операции, а не менять переменную так, чтобы она ссылалась на другой объект. Один из таких случаев – добавление нового элемента в изменяемую коллекцию:

```
>>> val numbers = ArrayList<Int>()
>>> numbers += 42
>>> println(numbers[0])
42
```

Если определить функцию с именем `plusAssign`, возвращающую значение типа `Unit`, Kotlin будет вызывать её, встретив оператор +=. Другие составные бинарные операторы выглядят аналогично: `minusAssign`, `timesAssign` и т. д.

В стандартной библиотеке Kotlin определяется функция `plusAssign` для изменяемых коллекций, и предыдущий пример использует её:

```
operator fun <T> MutableCollection<T>.plusAssign(element: T) {
    this.add(element)
}
```

Теоретически, встретив оператор +=, компилятор может вызвать любую из функций: `plus` и `plusAssign` (см. рис. 7.2). Если определены и применимы обе функции, компилятор сообщит об ошибке. Исправить проблему проще всего заменой оператора обычным вызовом функции. Также можно заменить `var` на `val`, чтобы операция `plusAssign` оказалась недопустимой в текущем контексте. Но лучше всего изначально проектировать классы непротиворечивыми: старайтесь не добавлять сразу обе функции, `plus` и `plusAssign`. Если класс неизменяемый (как `Point` в одном из примеров выше), добавляйте в него только операции, возвращающие новые значения (например, `plus`). Если вы проектируете изменяемый класс (например, построитель), добавляйте только `plusAssign` и другие подобные операции.

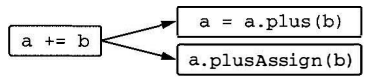
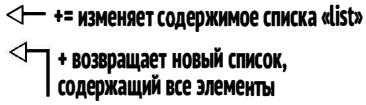


Рис. 7.2. Оператор += может быть преобразован в вызов функции plus или plusAssign

Стандартная библиотека Kotlin поддерживает оба подхода для коллекций. Операторы + и - всегда возвращают новую коллекцию. Операторы += и -= работают с изменяемыми коллекциями, модифицируя их на месте, а для неизменяемых коллекций возвращают копию с модификациями. (То есть операторы += и -= будут работать с неизменяемыми коллекциями, только если переменная со ссылкой объявлена как `var`.) В качестве операндов этих операторов можно использовать отдельные элементы или другие коллекции с элементами соответствующего типа:

```
>>> val list = arrayListOf(1, 2)
>>> list += 3
>>> val newList = list + listOf(4, 5)
>>> println(list)
[1, 2, 3]
>>> println(newList)
[1, 2, 3, 4, 5]
```



До сих пор мы обсуждали перегрузку бинарных операторов – операторов, применяемых к двум значениям (например, `a + b`). Однако в Kotlin поддерживается возможность перегрузки унарных операторов, которые применяются к единственному значению (например, `-a`).

7.1.3. Перегрузка унарных операторов

Процедура перегрузки унарного оператора ничем не отличается от описанной выше: объявляется функция (член или расширение) с предопределенным именем, которая затем отмечается модификатором `operator`. Рассмотрим это на примере.

Листинг 7.5. Определение унарного оператора

```
operator fun Point.unaryMinus(): Point {
    return Point(-x, -y)
}
```


```
>>> val p = Point(10, 20)
>>> println(-p)
Point(x=-10, y=-20)
```

Функции, используемые для перегрузки унарных операторов, не принимают никаких аргументов. Как показано на рис. 7.3, оператор унарного плюса действует аналогично. В табл. 7.2 перечислены все унарные операторы, которые можно перегрузить.

Рис. 7.3. Унарный + транслируется в вызов функции unaryPlus

Таблица 7.2. Унарные арифметические операторы, доступные для перегрузки

Выражение	Имя функции
`+a`	`unaryPlus`
`-a`	`unaryMinus`
`!a`	`not`
`++a, a++`	`inc`
`--a, a--`	`dec`

Когда определяются функции `inc` и `dec` для перегрузки операторов инкремента и декремента, компилятор автоматически поддерживает ту же семантику пред- и постинкремента, что и для обычных числовых типов. Взгляните на следующий пример – перегружающий оператор ++ для класса `BigDecimal`.

Листинг 7.6. Определение оператора инкремента

```
operator fun BigDecimal.inc() = this + BigDecimal.ONE

>>> var bd = BigDecimal.ZERO
>>> println(bd++)
0
>>> println(++bd)
2
```

Увеличит значение переменной после первого вызова println

Увеличит значение переменной перед вторым вызовом println

Постфиксная операция ++ сначала вернет текущее значение переменной `bd`, а затем увеличит его, в то время как префиксная операция работает с точностью до наоборот. На экране появятся те же значения, как если бы использовалась переменная типа `Int`, и для этого не требуется дополнительной поддержки.

7.2. Перегрузка операторов сравнения

По аналогии с арифметическими операторами Kotlin дает возможность использовать операторы сравнения (==, !=, >, < и другие) с любыми объектами, а не только с простыми типами. Вместо вызова `equals` или `compareTo`, как в Java, вы можете использовать непосредственные операторы сравнения, которые выглядят короче и понятнее. В этом разделе мы познакомимся с соглашениями, используемыми для поддержки этих операторов.

7.2.1. Операторы равенства: «equals»

Мы уже затрагивали тему равенства в разделе 4.3.1. Там мы узнали, что в языке Kotlin оператор == транслируется в вызов метода `equals`. Это лишь одно из применений принципа соглашений, о котором мы говорим.

Оператор != также транслируется в вызов `equals`, но с очевидным различием: результат вызова инвертируется. Обратите внимание: в отличие от всех других операторов, == и != можно использовать с операндами, способными иметь значение `null`, потому что за кулисами эти операторы проверяют на равенство значению `null`. Сравнение `a == b` проверит операнды на равенство `null`, а затем, если эта проверка дала отрицательный результат, вызовется

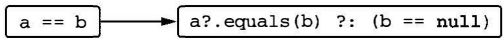


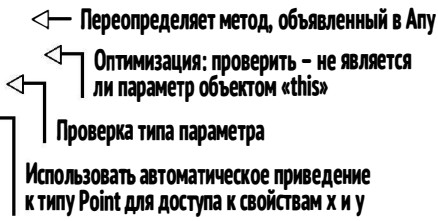
Рис. 7.4. Проверка на равенство == транслируется в вызов equals и проверку на равенство null

`a.equals(b)` (см. рис. 7.4). Иначе результат может принять значение `true`, только если оба аргумента являются пустыми (`null`) ссылками.

Для класса `Point` компилятор автоматически сгенерирует реализацию `equals`, потому что класс снабжен модификатором `data` (подробности описаны в разделе 4.3.2). Но если бы потребовалось написать её вручную, она могла бы выглядеть как в листинге 7.7.

Листинг 7.7. Реализация метода `equals`

```
class Point(val x: Int, val y: Int) {
    override fun equals(obj: Any?): Boolean {
        if (obj === this) return true
        if (obj !is Point) return false
        return obj.x == x && obj.y == y
    }
}
```




```
>>> println(Point(10, 20) == Point(10, 20))
true
>>> println(Point(10, 20) != Point(5, 5))
true
>>> println(null == Point(1, 2))
false
```

Здесь используется оператор *строгого равенства*, или *идентичности* (`===`), чтобы проверить равенство параметра текущему объекту. Оператор идентичности действует в точности как оператор `==` в Java: он сравнивает ссылки своих аргументов (или значения, если аргументы – это значения простого типа). Использование этого оператора – распространенная оптимизация реализаций `equals`. Обратите внимание, что оператор `===` недоступен для перегрузки.

Функция `equals` отмечена модификатором `override`, потому что (в отличие от других соглашений) реализация метода имеется в классе `Any` (и проверка равенства поддерживается в Kotlin для всех объектов). Это также объясняет, почему не нужно добавлять модификатор `operator`: базовый метод в классе `Any` уже отмечен этим модификатором, который автоматически применяется ко всем методам, которые реализуют или перекрывают его. Также отметьте, что `equals` нельзя реализовать как расширение, потому что реализация наследуется из класса `Any` и всегда имеет приоритет перед расширением.

Этот пример демонстрирует, что использование оператора `!=` также транслируется в вызов метода `equals`. Компилятор автоматически инвертирует возвращаемое значение, поэтому для гарантии правильной работы от вас не требуется ничего. А теперь перейдем к другим операторам сравнения.

7.2.2. Операторы отношения: compareTo

Классы в Java могут реализовать интерфейс `Comparable`, чтобы дать возможность использовать их в алгоритмах сравнения значений, таких как поиск максимального значения или сортировка. Метод `compareTo` этого интерфейса помогает определить, какой из двух объектов больше. Но в Java отсутствует краткий синтаксис вызова этого метода. Только значения простых типов могут сравниваться с использованием < и >, а для всех остальных типов приходится явно писать `element1.compareTo(element2)`.

Kotlin поддерживает тот же интерфейс `Comparable`. Но метод `compareTo` этого интерфейса может вызываться по соглашениям и используется операторами сравнения (<, >, <= и >=), которые автоматически транслируются в вызовы `compareTo` (см. рис. 7.5). Значение, возвращаемое методом `compareTo`, должно иметь тип `Int`. Выражение `p1 < p2` эквивалентно выражению `p1.compareTo(p2) < 0`. Другие операторы сравнения работают аналогично. Поскольку нет очевидно правильного способа сравнения точек, воспользуемся старым добрым классом `Person`, чтобы показать реализацию метода. Реализация будет использовать алгоритм, используемый в телефонных справочниках (сначала сравниваются фамилии, а затем, если они равны, сравниваются имена).

Рис. 7.5. Сравнение двух объектов транслируется в сравнение с нулем результата, возвращаемого методом compareTo

Листинг 7.8. Реализация метода `compareTo`

```
class Person(
    val firstName: String, val lastName: String
) : Comparable<Person> {

    override fun compareTo(other: Person): Int {
        return compareValuesBy(this, other,
            Person::lastName, Person::firstName)
    }
}
```

Вызывает заданные функции в указанном порядке и сравнивает возвращаемые ими результаты

```
>>> val p1 = Person("Alice", "Smith")
>>> val p2 = Person("Bob", "Johnson")
>>> println(p1 < p2)
false
```

Мы реализовали интерфейс `Comparable` так, что объекты `Person` могут сравниваться не только с помощью кода на Kotlin, но также и функций на Java (например, функций, используемых для сортировки коллекций). Как в случае с методом `equals`, модификатор `operator` уже применен к функ-

ции базового интерфейса. Поэтому нам не понадобилось повторять это ключевое слово при переопределении функции.

Обратите внимание: чтобы упростить реализацию `compareTo`, мы использовали функцию `compareValuesBy` из стандартной библиотеки Kotlin. Она принимает список функций обратного вызова, результаты которых подлежат сравнению. Каждая из этих функций по очереди вызывается для обоих объектов, сравниваются полученные значения, и возвращается результат. Если значения первой пары отличаются, возвращается результат их сравнения. Если они равны, вызывается следующая функция, и сравниваются её результаты для каждого из объектов – или, если не осталось других функций, возвращается 0. В качестве функций обратного вызова можно передавать лямбда-выражения или, как в данном примере, ссылки на свойства.

Также имейте в виду, что прямое сравнение полей вручную могло бы работать быстрее, но пришлось бы написать больше кода. Как обычно, предпочтение следует отдавать более компактному коду, а о производительности беспокоиться, только если сравнение планируется выполнять очень часто.

Все Java-классы, реализующие интерфейс `Comparable`, можно сравнивать в коде на Kotlin с использованием краткого синтаксиса операторов:

```
>>> println("abc" < "bac")
true
```

Для этого не требуется добавлять никаких расширений.

7.3. Соглашения для коллекций и диапазонов

К наиболее распространенным операциям для работы с коллекциями относятся операции извлечения и изменения значений элементов по их индексам, а также операция проверки вхождения в коллекцию. Все такие операции поддерживают синтаксис операторов: чтобы прочитать или изменить значение элемента по индексу, используется синтаксис `a[b]` (называется *оператором индекса*). Для проверки вхождения элемента в коллекцию или в диапазон, а также для итераций по коллекциям можно использовать оператор `in`. Эти операции можно добавить в свои классы, действующие подобно коллекциям. Давайте посмотрим, какие соглашения используются для поддержки этих операций.

7.3.1. Обращение к элементам по индексам: «get» и «set»

Как известно, к элементам словарей в Kotlin можно обращаться так же, как к элементам массивов в Java, – с применением квадратных скобок:

```
val value = map[key]
```

Тот же оператор можно использовать для изменения значений по ключам в изменяемом словаре:

```
mutableMap[key] = newValue
```

Теперь посмотрим, как это работает. Оператор индекса в Kotlin – ещё один пример соглашений. Операция чтения элемента с использованием оператора индекса транслируется в вызов метода-оператора `get`, а операция записи – в вызов `set`. Эти методы уже определены в интерфейсах `Map` и `MutableMap`. Ниже – о том, как добавить аналогичные методы в свой класс.

Мы можем реализовать возможность обращения к координатам точки с использованием квадратных скобок: `p[0]` – для доступа к координате X и `p[1]` – для доступа к координате Y. Вот как реализовать и использовать такую возможность.

Листинг 7.9. Реализация соглашения get

```
operator fun Point.get(index: Int): Int {
    return when(index) {
        0 -> x
        1 -> y
        else ->
            throw IndexOutOfBoundsException("Invalid coordinate $index")
    }
}

>>> val p = Point(10, 20)
>>> println(p[1])
20
```

Определение функции-оператора с именем «get»

Вернуть координату, соответствующую заданному индексу

Как видите, достаточно определить функцию `get` и отметить её модификатором `operator`. После этого выражения вида `p[1]`, где `p` – это объект типа `Point`, будут транслироваться в вызовы метода `get`, как показано на рис. 7.6.

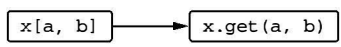


Рис. 7.6. Операция доступа с применением квадратных скобок транслируется в вызов функции get

Обратите внимание, что в функции `get` можно объявить параметр не только типа `Int`, но и любого другого типа. Например, когда оператор индекса применяется к словарю, тип параметра должен совпадать с типом ключей словаря, который может быть любым произвольным типом. Также можно определить метод `get` с несколькими параметрами. Например, в реализации класса, представляющего двумерный массив или матрицу, можно определить метод `operator fun get(rowIndex: Int, colIndex: Int)` и вызывать его как `matrix[row, col]`. Если доступ к элементам коллекции возможен с применением клю-

чей разных типов, то допускается также определить перегруженные версии метода get с разными типами параметров.

Аналогично можно определить функцию, позволяющую изменять значение элемента по индексу с применением синтаксиса квадратных скобок. Класс Point – неизменяемый, поэтому нет смысла добавлять в него такой метод. Давайте лучше определим другой класс, представляющий изменяемую точку, и используем его в качестве примера.

Листинг 7.10. Реализация соглашения set

```
data class MutablePoint(var x: Int, var y: Int)

operator fun MutablePoint.set(index: Int, value: Int) {
    when(index) {
        0 -> x = value
        1 -> y = value
        else ->
            throw IndexOutOfBoundsException("Invalid coordinate $index")
    }
}

>>> val p = MutablePoint(10, 20)
>>> p[1] = 42
>>> println(p)
MutablePoint(x=10, y=42)
```

Определение функции-оператора с именем «set»

Изменить координату, соответствующую заданному индексу

Рис. 7.7. Операция присваивания с применением квадратных скобок транслируется в вызов функции set

Этот пример так же прост, как предыдущий: чтобы дать возможность использовать оператор индекса в операции присваивания, достаточно определить функцию с именем set. Последний параметр в set получает значение, указанное справа от оператора присваивания, а другие аргументы соответствуют индексам внутри квадратных скобок (рис. 7.7).

7.3.2. Соглашение «in»

Еще один оператор, поддерживаемый коллекциями, – оператор in. Он используется для проверки вхождения объекта в коллекцию. Соответствующая функция называется contains. Реализуем её, чтобы дать возможность использовать оператор in для проверки вхождения точки в границы прямоугольника.

Листинг 7.11. Реализация соглашения in

```
data class Rectangle(val upperLeft: Point, val lowerRight: Point)
```

```
operator fun Rectangle.contains(p: Point): Boolean {
    return p.x in upperLeft.x until lowerRight.x &&
           p.y in upperLeft.y until lowerRight.y
}

>>> val rect = Rectangle(Point(10, 20), Point(50, 50))
>>> println(Point(20, 30) in rect)
true
>>> println(Point(5, 5) in rect)
false
```

Создает диапазон и проверяет принадлежность ему координаты «x»

Использует функцию «until» для создания открытого диапазона

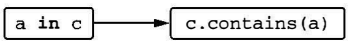


Рис. 7.8. Оператор in транслируется в вызов функции contains

Метод `contains` вызывается для объекта справа от `in`, а объект слева передается этому методу в аргументе (см. рис. 7.8).

В реализации метода `Rectangle.contains` мы задействовали функцию `until` из стандартной библиотеки и с её помощью конструируем открытый диапазон, который затем используем для проверки принадлежности точки этому диапазону.

Открытый диапазон – это диапазон, не включающий конечного значения. Например, если сконструировать обычный (закрытый) диапазон `10..20`, он будет включать все значения от 10 до 20, в том числе и 20. Открытый диапазон от 10 до 20 включает число от 10 до 19, но не включает 20. Класс прямоугольника обычно определяется так, что правая нижняя точка не считается частью прямоугольника, поэтому мы использовали открытые диапазоны.

7.3.3. Соглашение rangeTo

Для создания диапазона используется синтаксис `..`: например, `1..10` перечисляет все числа от 1 до 10. Мы уже познакомились с диапазонами в разделе 2.4.2, а теперь пришла пора обсудить соглашение, помогающее создавать их. Оператор `..` представляет собой краткую форму вызова функции `rangeTo` (см. рис. 7.9).

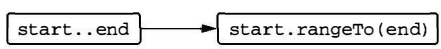


Рис. 7.9. Оператор .. транслируется в вызов функции rangeTo

Функция `rangeTo` возвращает диапазон. Вы можете реализовать поддержку этого оператора в своем классе. Но если класс реализует интерфейс `Comparable`, в этом нет необходимости: в таком случае диапазоны будут создаваться средствами стандартной библиотеки Kotlin. Библиотека включает функцию `rangeTo`, которая может быть вызвана для любого элемента, поддерживающего операцию сравнения:

```
operator fun <T: Comparable<T>> T.rangeTo(that: T): ClosedRange<T>
```

Эта функция возвращает диапазон, что дает возможность проверить разные элементы на принадлежность ему. Для примера сконструируем

диапазон дат, используя класс `LocalDate` (объявлен в стандартной библиотеке Java 8).

Листинг 7.12. Операции с диапазонами дат

```
>>> val now = LocalDate.now()
>>> val vacation = now..now.plusDays(10)
>>> println(now.plusWeeks(1) in vacation)
true
```

Создает 10-дневный диапазон, начиная от текущей даты

Проверяет принадлежность конкретной даты этому диапазону

Компилятор преобразует выражение `now..now.plusDays(10)` в `now.rangeTo(now.plusDays(10))`. Функция `rangeTo` не является членом класса `LocalDate` – это функция-расширение для `Comparable`, как было показано ранее.

Оператор `rangeTo` имеет более низкий приоритет, чем арифметические операторы, поэтому мы рекомендуем пользоваться круглыми скобками, чтобы избежать путаницы:

```
>>> val n = 9
>>> println(0..(n + 1))
0..10
```

Также можно записать как 0..n + 1, но круглые скобки делают код более ясным

Также отметьте, что выражение `0..n.forEach {}` не будет компилироваться. Чтобы исправить эту проблему, диапазон нужно заключить в круглые скобки:

```
>>> (0..n).forEach { print(it) }
0123456789
```

Заключите диапазон в круглые скобки, чтобы получить возможность вызвать его метод

Теперь обсудим соглашения, используемые для итерации по коллекциям и диапазонам.

7.3.4. Соглашение «iterator» для цикла «for»

Как рассказывалось в главе 2, циклы `for` в Kotlin используют тот же оператор `in`, что применяется для проверки принадлежности диапазону. Но в данном контексте он предназначен для выполнения итераций. То есть такие инструкции, как `for (x in list) { ... }`, транслируются в вызов `list.iterator()`, который повторно вызывает методы `hasNext` и `next`, – в точности как в Java.

Обратите внимание, что в Kotlin этот метод – соглашение, то есть метод `iterator` можно определить как расширение. Это объясняет возможность итераций по обычным строкам Java: стандартная библиотека определяет функцию-расширение `iterator` в `CharSequence`, суперклассе, который наследует класс `String`:

```
operator fun CharSequence.iterator(): CharIterator

>>> for (c in "abc") {}
```

Эта библиотечная функция позволяет выполнять итерацию по строке

Вы можете определять метод `iterator` в своих классах. Например, следующий метод позволяет выполнять итерации по датам.

Листинг 7.13. Реализация итератора по диапазону дат

```
operator fun ClosedRange<LocalDate>.iterator(): Iterator<LocalDate> =
    object : Iterator<LocalDate> {
        var current = start

        override fun hasNext() =
            current <= endInclusive

        override fun next() = current.apply {
            current = plusDays(1)
        }
    }
```

Этот объект реализует интерфейс Iterator для поддержки итераций по элементам LocalDate

Обратите внимание, что для дат используется соглашение compareTo

Возвращает текущую дату как результат перед ее изменением

Увеличивает текущую дату на один день

```
>>> val newYear = LocalDate.ofYearDay(2017, 1)
>>> val daysOff = newYear.minusDays(1)..newYear
>>> for (dayOff in daysOff) { println(dayOff) }
2016-12-31
2017-01-01
```

Выполняет итерации по daysOff, когда доступна соответствующая функция iterator

Обратите внимание, как определяется метод `iterator` для нестандартного типа диапазона: аргумент имеет тип `LocalDate`. Библиотечная функция `rangeTo`, представленная в предыдущем разделе, возвращает экземпляр `ClosedRange`, а расширение `iterator` в `ClosedRange<LocalDate>` позволяет использовать экземпляр диапазона в цикле `for`.

7.4. Мультидекларации и функции component

Обсуждая классы данных в разделе 4.3.2, мы упоминали, что раскроем некоторые из их дополнительных особенностей позднее. Теперь, познакомившись с идеей соглашений, мы готовы рассмотреть *мультидекларации* (destructuring declarations), которые позволяют распаковать единое составное значение и использовать его для инициализации нескольких переменных.

Вот как это работает:

```
>>> val p = Point(10, 20)
>>> val (x, y) = p
>>> println(x)
10
```

Объявляются переменные x и y и инициализируются компонентами объекта p

```
>>> println(y)
20
```

Мультидекларация похожа на обычное объявление переменной, но содержит несколько переменных, заключенных в круглые скобки.

Скрытая от ваших глаз работа мультидеклараций также основана на принципе соглашений. Для инициализации каждой переменной в мультидекларации вызывается функция с именем componentN, где *N* – номер позиции переменной в объявлении. Иными словами, предыдущий пример транслируется в код, изображенный на рис. 7.10.

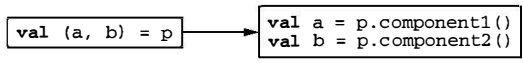


Рис. 7.10. Мультидекларация транслируется в вызовы функций componentN

Для класса данных компилятор сгенерирует функции componentN для каждого свойства, объявленного в основном конструкторе. Следующий пример демонстрирует, как можно объявить такие функции вручную в других классах, не являющихся классами данных:

```
class Point(val x: Int, val y: Int) {
    operator fun component1() = x
    operator fun component2() = y
}
```

Мультидекларации часто оказываются удобным способом возврата нескольких значений из функций. Для этого объявите класс данных для хранения возвращаемых значений и используйте его в качестве типа значения, возвращаемого функцией. Синтаксис мультидеклараций позволяет легко распаковать и использовать значения после вызова функции. Для демонстрации напишем простую функцию, разбивающую имя файла на отдельные имя и расширение.

Листинг 7.14. Использование мультидекларации для возврата из функции нескольких значений

```
data class NameComponents(val name: String,
                          val extension: String)

fun splitFilename(fullName: String): NameComponents {
    val result = fullName.split('.', limit = 2)
    return NameComponents(result[0], result[1])
}

>>> val (name, ext) = splitFilename("example.kt")
>>> println(name)
example
```

Объявление класса данных для хранения значений

Возврат экземпляра класса данных из функции

Использует синтаксис мультидеклараций для извлечения значений

```
>>> println(ext)
kt
```

Этот пример можно усовершенствовать, приняв во внимание, что функции componentN имеются в массивах и коллекциях. Это может пригодиться при работе с коллекциями, размер которых известен заранее. Именно такой случай – функция split, возвращающая список с двумя элементами.

Листинг 7.15. Использование мультидекларации с коллекцией

```
data class NameComponents(
    val name: String,
    val extension: String)

fun splitFilename(fullName: String): NameComponents {
    val (name, extension) = fullName.split('.', limit = 2)
    return NameComponents(name, extension)
}
```

Конечно, невозможно объявить бесконечное количество таких функций componentN, чтобы можно было использовать этот синтаксис для работы с произвольными коллекциями. Но обычно в этом нет никакой необходимости. Стандартная библиотека позволяет использовать этот синтаксис для извлечения первых пяти элементов из контейнера.

Ещё более простой способ возврата нескольких значений из функций дают классы Pair и Triple из стандартной библиотеки. В этом случае код получается менее выразительным, потому что эти классы не позволяют узнать смысл возвращаемого объекта, зато более лаконичным, потому что отпадает необходимость объявлять свой класс данных.

7.4.1. Мультидекларации и циклы

Мультидекларации могут использоваться не только как инструкции верхнего уровня в функциях, но и в других местах, где допускается объявление переменных – например, в циклах. Один из интересных приемов – перечисление элементов словаря в цикле for. Вот маленький пример использования этого синтаксиса для вывода всех элементов заданного словаря.

Листинг 7.16. Использование мультидекларации для обхода элементов словаря

```
fun printEntries(map: Map<String, String>) {
    for ((key, value) in map) {
        println("$key -> $value")
    }
```

```
}

>>> val map = mapOf("Oracle" to "Java", "JetBrains" to "Kotlin")
>>> printEntries(map)
Oracle -> Java
JetBrains -> Kotlin
```

В этом простом примере используются два соглашения Kotlin: одно – для организации итерации по содержимому объекта, а другое – для поддержки мультидеклараций. Стандартная библиотека Kotlin включает функцию-расширение `iterator` для словарей, которая возвращает итератор для обхода элементов словаря. То есть, в отличие от Java, в Kotlin есть возможность выполнять итерации по словарю непосредственно. В `Map.Entry` имеются также функции-расширения `component1` и `component2`, которые возвращают ключ и значение соответственно. В результате предыдущий цикл транслируется в следующий эквивалентный код:

```
for (entry in map.entries) {
    val key = entry.component1()
    val value = entry.component2()
    // ...
}
```

Этот пример ещё раз иллюстрирует важность функций-расширений для соглашений.

7.5. Повторное использование логики обращения к свойству: делегирование свойств

В заключение главы рассмотрим еще одну особенность, опирающуюся на соглашения, одну из самых необычных и мощных в языке Kotlin: *делегирование свойств*. Эта особенность позволяет без труда реализовать свойства с логикой сложнее, чем хранение данных в соответствующих полях, без дублирования кода в каждом методе доступа. Например, свойства могут хранить свои значения в базе данных, в сеансе браузера, в словаре и так далее.

В основе этой особенности лежит *делегирование*: шаблон проектирования, согласно которому объект не сам выполняет требуемое задание, а делегирует его другому вспомогательному объекту. Такой вспомогательный объект называется *делегатом*. Вы уже видели этот шаблон в разделе 4.3.3, когда мы обсуждали делегирование классов. Но в данном случае данный шаблон применяется к свойствам, которые могут делегировать логику доступа методам вспомогательного объекта. Вы можете реализовать этот шаблон вручную (как будет показано чуть ниже) или использовать лучшее решение: воспользоваться встроенной поддержкой в языке Kotlin. Для на-

чала познакомимся с общей идеей, а затем рассмотрим конкретные примеры.

7.5.1. Делегирование свойств: основы

В общем случае синтаксис делегирования свойств выглядит так:

```
class Foo {
    var p: Type by Delegate()
}
```

Свойство p делегирует логику своих методов доступа другому объекту: в данном случае новому экземпляру класса `Delegate`. Экземпляр создается выражением, следующим за ключевым словом `by`, и может быть чем угодно, удовлетворяющим требованиям соглашения для делегирования свойств.

Компилятор создаст скрытое вспомогательное свойство, инициализированное экземпляром объекта-делегата, которому делегируется логика работы свойства p. Для простоты назовем это свойство `delegate`:

```
class Foo {
    private val delegate = Delegate()

    var p: Type
        set(value: Type) = delegate.setValue(..., value)
        get() = delegate.getValue(...)
}
```

Это вспомогательное свойство, сгенерированное компилятором

Методы доступа, сгенерированные для свойства «p», вызывают getValue и setValue объекта «delegate»

В соответствии с соглашением класс `Delegate` должен иметь методы `getValue` и `setValue` (последний требуется только для изменяемых свойств). Как обычно, они могут быть членами или расширениями. Чтобы упростить рассуждения, опустим их параметры; точные их сигнатуры будут показаны ниже в этой главе. В простейшем случае класс `Delegate` мог бы выглядеть примерно так:

```
class Delegate {
    operator fun getValue(...) { ... }

    operator fun setValue(..., value: Type) { ... }
}
```

Метод getValue реализует логику метода чтения

Метод setValue реализует логику метода записи

```
class Foo {
    var p: Type by Delegate()
}

>>> val foo = Foo()
>>> val oldValue = foo.p
>>> foo.p = newValue
```

Ключевое слово «by» связывает свойство с объектом-делегатом

Обращение к свойству foo.p приводит к вызову delegate.getValue(...) за кулисами

Операция изменения значения свойства вызывает delegate.setValue(..., newValue)

Свойство `foo.p` можно использовать как обычно, но за кулисами операции с ним будут вызывать методы вспомогательного свойства типа `Delegate`. Чтобы понять, как этот механизм может пригодиться на практике, рассмотрим пример, демонстрирующий мощь делегирования свойств: поддержку отложенной инициализации в библиотеке. А затем покажем, как определять свои делегаты и в каких случаях это может пригодиться.

7.5.2. Использование делегирования свойств: отложенная инициализация и «by lazy()»

Отложенная инициализация (lazy initialization) – распространенный шаблон, позволяющий отложить создание объекта до момента, когда он действительно потребуется. Это может пригодиться, когда процесс инициализации потребляет значительные ресурсы или данные в объекте могут не требоваться.

Для примера рассмотрим класс `Person`, включающий список указанных пользователем адресов электронной почты. Адреса хранятся в базе данных, и для их извлечения требуется определенное время. Хотелось бы сделать так, чтобы адреса извлекались из базы данных только при первом обращении к свойству и только один раз. Допустим, что у нас есть функция `loadEmails`, возвращающая адреса из базы данных:

```
class Email { /*...*/ }
fun loadEmails(person: Person): List<Email> {
    println("Load emails for ${person.name}")
    return listOf(/*...*/)
}
```

Вот как можно реализовать отложенную загрузку, используя дополнительное свойство `_emails`, изначально хранящее `null`, и список адресов после загрузки.

Листинг 7.17. Реализация отложенной инициализации с использованием вспомогательного свойства

```
class Person(val name: String) {
    private var _emails: List<Email>? = null

    val emails: List<Email>
        get() {
            if (_emails == null) {
                _emails = loadEmails(this)
            }
```

Свойство «_emails», хранящее данные и которому делегируется логика работы свойства «emails»

Загрузка данных при первом обращении

```
            return _emails!!          <┐ Если данные уже загружены,
        }                             <┘ вернуть их
}

>>> val p = Person("Alice")
>>> p.emails                          <┐ Загрузка адресов при
Load emails for Alice                 <┘ первом обращении
>>> p.emails
```

Здесь используется приём на основе так называемого *теневого свойства* (backing property). У нас имеются свойство `_emails`, хранящее значение, и свойство `emails`, открывающее доступ к нему для чтения. Мы вынуждены использовать два свойства, потому что они имеют два разных типа: `_emails` может хранить значение `null`, а `emails` – нет. Этот приём используется очень часто, поэтому его стоит освоить.

Но код получился тяжелым для чтения – просто представьте, что вам потребовалось реализовать несколько свойств с отложенной инициализацией. Более того, этот приём не всегда работает правильно: реализация не будет безопасной в контексте многопоточного выполнения. Однако Kotlin предлагает более удачное решение.

Код станет намного проще, если использовать делегированные свойства, инкапсулирующие теневые свойства для хранения значений и логику, гарантирующую инициализацию свойств только один раз. В данном случае можно использовать делегата, возвращаемого функцией `lazy` из стандартной библиотеки.

Листинг 7.18. Реализация отложенной инициализации с использованием делегирования свойства

```
class Person(val name: String) {
    val emails by lazy { loadEmails(this) }
}
```

Функция `lazy` возвращает объект, имеющий метод `getValue` с соответствующей сигнатурой, – то есть её можно использовать с ключевым словом `by` для создания делегированного свойства. В аргументе функции `lazy` передается лямбда-выражение, которое она вызывает для инициализации значения. Функция `lazy` по умолчанию пригодна для использования в многопоточном окружении, и если потребуется, ей можно передать дополнительные параметры, чтобы сообщить, какую блокировку использовать или вообще игнорировать средства синхронизации, если класс никогда не будет использоваться в многопоточной среде.

В следующем разделе мы углубимся в детали механизма делегирования свойств и обсудим соглашения, действующие в этой области.

7.5.3. Реализация делегирования свойств

Чтобы понять, как реализуется делегирование свойств, рассмотрим еще один пример: уведомление обработчиков событий, когда свойство объекта изменяет свое значение. Это может пригодиться в самых разных ситуациях: например, когда объект представляет элемент пользовательского интерфейса и требуется автоматически обновлять изображение на экране при изменении содержимого объекта. Для решения подобных задач в Java есть стандартный механизм: классы `PropertyChangeSupport` и `PropertyChangeEvent`. Давайте сначала посмотрим, как можно использовать их в Kotlin без делегирования свойств, а затем проведем рефакторинг кода и применим поддержку делегирования.

Класс `PropertyChangeSupport` управляет списком обработчиков и передает им события `PropertyChangeEvent`. Чтоб задействовать этот механизм, нужно сохранить экземпляр этого класса в поле класса компонента JavaBean и делегировать ему обработку изменения свойства.

Чтобы не добавлять это поле в каждый класс, можно создать маленький вспомогательный класс, который хранит экземпляр `PropertyChangeSupport` со списком обработчиков событий изменения свойства. Все ваши классы могут наследовать этот вспомогательный класс, чтобы получить доступ к `changeSupport`.

Листинг 7.19. Вспомогательный класс для использования `PropertyChangeSupport`

```
open class PropertyChangeAware {
    protected val changeSupport = PropertyChangeSupport(this)

    fun addPropertyChangeListener(listener: PropertyChangeListener) {
        changeSupport.addPropertyChangeListener(listener)
    }

    fun removePropertyChangeListener(listener: PropertyChangeListener) {
        changeSupport.removePropertyChangeListener(listener)
    }
}
```

Теперь напишем класс `Person`. Определим одно неизменяемое свойство (имя, которое обычно не меняется) и два изменяемых свойства: возраст и размер зарплаты. Класс будет уведомлять обработчиков при изменении возраста или размера зарплаты.

Листинг 7.20. Реализация передачи уведомлений об изменении свойств вручную

```
class Person(
        val name: String, age: Int, salary: Int
```

```
) : PropertyChangeAware() {

    var age: Int = age
        set(newValue) {
            val oldValue = field
            field = newValue
            changeSupport.firePropertyChange(
                "age", oldValue, newValue)
        }

        var salary: Int = salary
            set(newValue) {
                val oldValue = field
                field = newValue
                changeSupport.firePropertyChange(
                    "salary", oldValue, newValue)
            }
}

>>> val p = Person("Dmitry", 34, 2000)
>>> p.addPropertyChangeListener(
...     PropertyChangeListener { event ->
...         println("Property ${event.propertyName} changed " +
...                 "from ${event.oldValue} to ${event.newValue}")
...     }
... )
>>> p.age = 35
Property age changed from 34 to 35
>>> p.salary = 2100
Property salary changed from 2000 to 2100
```

Идентификатор «field» даёт доступ к полю, соответствующему свойству

Уведомляет обработчиков об изменении свойства

Подключает обработчик события изменения свойства

Обратите внимание, что для доступа к полям, соответствующим свойствам `age` и `salary`, в этом примере используется идентификатор `field`, который обсуждался в разделе 4.2.4.

В методах записи присутствует масса повторяющегося кода. Давайте попробуем извлечь класс, который будет хранить значение свойства и посылать все необходимые уведомления.

Листинг 7.21. Реализация передачи уведомлений об изменении свойств с применением вспомогательного класса

```
class ObservableProperty(
    val propName: String, var propValue: Int,
    val changeSupport: PropertyChangeSupport
) {
    fun getValue(): Int = propValue
```

```
    fun setValue(newValue: Int) {
        val oldValue = propValue
        propValue = newValue
        changeSupport.firePropertyChange(propName, oldValue, newValue)
    }
}

class Person(
    val name: String, age: Int, salary: Int
) : PropertyChangeAware() {

    val _age = ObservableProperty("age", age, changeSupport)
    var age: Int
        get() = _age.getValue()
        set(value) { _age.setValue(value) }

    val _salary = ObservableProperty("salary", salary, changeSupport)
    var salary: Int
        get() = _salary.getValue()
        set(value) { _salary.setValue(value) }
}
```

Теперь мы ещё ближе к пониманию механизма делегирования свойств в Kotlin. Мы создали класс, хранящий значение свойства и автоматически рассылающий уведомления при его изменении. Это избавило нас от массы повторяющегося кода, но теперь мы вынуждены создавать экземпляр `ObservableProperty` для каждого свойства и делегировать ему операции чтения и записи. Поддержка делегирования свойств в Kotlin позволяет избавиться и от этого шаблонного кода. Но прежде чем узнать, как это делается, изменим сигнатуры методов `ObservableProperty`, чтобы привести их в соответствие с соглашениями.

Листинг 7.22. ObservableProperty как объект-делегат для свойства

```
class ObservableProperty(
    var propValue: Int, val changeSupport: PropertyChangeSupport
) {
    operator fun getValue(p: Person, prop: KProperty<*>): Int = propValue

    operator fun setValue(p: Person, prop: KProperty<*>, newValue: Int) {
        val oldValue = propValue
        propValue = newValue
        changeSupport.firePropertyChange(prop.name, oldValue, newValue)
    }
}
```

По сравнению с предыдущей версией в коде появились следующие изменения:

- функции `getValue` и `setValue` теперь отмечены модификатором `operator`, как того требуют соглашения;
- в функции было добавлено по два параметра: один – для приёма экземпляра, свойство которого требуется читать или изменять, а второй – для представления самого свойства. Свойство представлено объектом типа `KProperty`. Мы подробно рассмотрим его в разделе 10.2, а пока просто имейте в виду, что обратиться к свойству по его имени можно в виде `KProperty.name`;
- мы убрали свойство `name`, потому что теперь вы можете получить доступ через `KProperty`.

Теперь можно использовать все волшебство механизма делегирования свойств в Kotlin. Хотите увидеть, насколько короче стал код?

Листинг 7.23. Использование делегирования свойств для передачи извещений об изменении

```
class Person(
    val name: String, age: Int, salary: Int
) : PropertyChangeAware() {

    var age: Int by ObservableProperty(age, changeSupport)
    var salary: Int by ObservableProperty(salary, changeSupport)
}
```

Ключевое слово `by` заставляет компилятор Kotlin делать всё то, что мы делали в предыдущем разделе вручную. Сравните этот код с предыдущей версией класса `Person`: код, сгенерированный компилятором, очень похож на неё. Объект справа от `by` называется *делегатом*. Kotlin автоматически сохраняет делегата в скрытом свойстве и вызывает методы `getValue` и `setValue` делегата при попытке прочитать или изменить основное свойство.

Вместо реализации логики наблюдения за свойством вручную можно воспользоваться стандартной библиотекой Kotlin. Оказывается, стандартная библиотека уже содержит класс, похожий на `ObservableProperty`. Но класс в стандартной библиотеке никак не связан с классом `PropertyChangeSupport`, использовавшимся выше, поэтому требуется передать лямбда-выражение, определяющее, как должны передаваться уведомления об изменении значения свойства. Вот как это сделать.

Листинг 7.24. Использование `Delegates.observable` для отправки уведомлений об изменении свойства

```
class Person(
    val name: String, age: Int, salary: Int
```

```
) : PropertyChangeAware() {

    private val observer = {
        prop: KProperty<*>, oldValue: Int, newValue: Int ->
        changeSupport.firePropertyChange(prop.name, oldValue, newValue)
    }

    var age: Int by Delegates.observable(age, observer)
    var salary: Int by Delegates.observable(salary, observer)
}
```

Выражение справа от by не обязательно должно создавать новый экземпляр. Это может быть вызов функции, другое свойство или любое выражение, при условии, что значением этого выражения является объект с методами getValue и setValue, принимающими параметры требуемых типов. Как и другие соглашения, getValue и setValue могут быть методами, объявленными в самом объекте, или функциями-расширениями.

Обратите внимание: чтобы сделать примеры максимально простыми, мы показали только делегирование свойств типа Int. Механизм делегирования свойств полностью универсален и с успехом применяется к свойствам любых типов.

7.5.4. Правила трансляции делегированных свойств

Ещё раз перечислим правила делегирования свойств. Допустим, у вас имеется класс, делегирующий свойство:

```
class C {
    var prop: Type by MyDelegate()
}

val c = C()
```

Экземпляр MyDelegate будет хранить скрытое свойство (назовем его <delegate>). Кроме того, для представления свойства компилятор будет использовать объект типа KProperty (назовем его <property>).

В результате компилятор сгенерирует следующий код:

```
class C {
    private val <delegate> = MyDelegate()

    var prop: Type
        get() = <delegate>.getValue(this, <property>)
        set(value: Type) = <delegate>.setValue(this, <property>, value)
}
```

То есть внутри каждого метода доступа свойства компилятор вызовет соответствующие методы getValue и setValue, как показано на рис. 7.11.

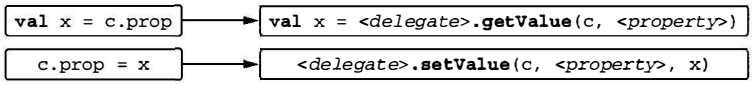


Рис. 7.11. При обращении к свойству вызываются функции getValue и setValue объекта <delegate>

Механика делегирования свойств чрезвычайно проста и одновременно способна поддерживать множество интересных сценариев. Можно определить, где должно храниться значение (в словаре, в базе данных или в сеансовом cookie), а также что должно произойти в процессе обращения к свойству (можно добавить проверку допустимости, послать уведомление об изменении и так далее). Всё это можно реализовать с минимумом кода. Давайте исследуем ещё один приём делегирования свойств в стандартной библиотеке, а затем посмотрим, как использовать его в своих фреймворках.

7.5.5. Сохранение значений свойств в словаре

Кроме всего прочего, делегирование свойств широко применяется в объектах с динамически определяемым набором атрибутов. Такие объекты иногда называют *расширяемыми объектами* (expando objects). Например, представьте систему управления контактами, которая позволяет сохранять произвольную информацию о ваших контактах. Каждый контакт в такой системе имеет несколько обязательных свойств (как минимум имя), которые обрабатываются специальным образом, а также произвольное количество дополнительных атрибутов, разных для разных контактов (например, день рождения младшего ребенка).

Для хранения таких атрибутов можно использовать словарь и реализовать свойства для доступа к информации, требующей специальной обработки. Например:

Листинг 7.25. Определение свойства, хранящего свое значение в словаре

```
class Person {
    private val _attributes = hashMapOf<String, String>()

    fun setAttribute(attrName: String, value: String) {
        _attributes[attrName] = value
    }

    val name: String
        get() = _attributes["name"]!!   <-- Извлечение атрибута из словаря вручную
}

>>> val p = Person()
>>> val data = mapOf("name" to "Dmitry", "company" to "JetBrains")
```

```
>>> for ((attrName, value) in data)
... p.setAttribute(attrName, value)
>>> println(p.name)
Dmitry
```

Здесь мы использовали обобщенный API для загрузки данных в объект (в реальном проекте данные могут извлекаться из строки в формате JSON или откуда-то ещё), а затем конкретный API для доступа к значению одного из свойств. Задействовать механизм делегирования свойств в этом коде очень просто: достаточно указать имя словаря вслед за ключевым словом by.

Листинг 7.26. Делегированное свойство, хранящее свое значение в словаре

```
class Person {
    private val _attributes = hashMapOf<String, String>()

    fun setAttribute(attrName: String, value: String) {
        _attributes[attrName] = value
    }

    val name: String by _attributes    <-- Использовать словарь в роли объекта-делегата
}
```

Это возможно благодаря тому, что в стандартной библиотеке определены функции-расширения `getValue` и `setValue` для стандартных интерфейсов `Map` и `MutableMap`. Имя свойства автоматически используется как ключ для доступа к значению в словаре. Как показано в листинге 7.25, ссылка на `p.name` фактически транслируется в вызов `_attributes.getValue(p, prop)`, который, в свою очередь, возвращает результат выражения `_attributes[prop.name]`.

7.5.6. Делегирование свойств в фреймворках

Возможность изменить способ хранения и изменения свойств объекта чрезвычайно удобна для разработчиков фреймворков. В разделе 1.3.1 мы показали пример использования делегированных свойств в фреймворке для работы с базой данных. В этом разделе вы увидите похожий пример и узнаете, как он работает.

Представим, что в нашей базе данных есть таблица `Users` с двумя столбцами: `name`, строкового типа, и `age`, целочисленного типа. Мы можем определить Kotlin-классы `Users` и `User`, а затем загружать все записи из базы данных и изменять их в коде на Kotlin посредством экземпляров класса `User`.

Листинг 7.27. Доступ к столбцам базы данных с использованием делегированных свойств

```
object Users : IdTable() {                          <— Объект, соответствующий таблице в базе данных
    val name = varchar("name", length = 50).index()  | Свойства, соответствующие
    val age = integer("age")                         | столбцам в этой таблице
}

class User(id: EntityID) : Entity(id) {   <— Каждый экземпляр User соответствует
                                             конкретной записи в таблице
    var name: String by Users.name        <— Значение «name» – это имя конкретного
    var age: Int by Users.age                пользователя, хранящееся в базе данных
}
```

Объект `Users` описывает таблицу базы данных. Он объявлен как `object`, потому что описывает таблицу в целом и должен присутствовать в единственном экземпляре. Свойства объекта представляют столбцы таблицы.

Класс `Entity`, родитель класса `User`, содержит отображение столбцов базы данных на их значения для конкретной записи. Свойства в классе `User` получают значения столбцов `name` и `age` в базе данных для данного пользователя.

Пользоваться фреймворком особенно удобно, потому что извлечение соответствующего значения из отображения в классе `Entity` происходит автоматически, а операция изменения отмечает объект как изменившийся. Потом, если потребуется, его можно легко сохранить в базе данных. Вы можете написать в своем коде `user.age += 1`, и соответствующая запись в базе данных автоматически изменится.

Теперь вы знаете достаточно, чтобы понять, как реализовать фреймворк с таким API. Все атрибуты в `User` (`name`, `age`) реализованы как свойства, делегированные объекту таблицы (`Users.name`, `Users.age`):

```
class User(id: EntityID) : Entity(id) {
    var name: String by Users.name        <— Users.name – делегат
    var age: Int by Users.age                для свойства «name»
}
```

Теперь посмотрим на явно заданные типы столбцов:

```
object Users : IdTable() {
    val name: Column<String> = varchar("name", 50).index()
    val age: Column<Int> = integer("age")
}
```

Для класса `Column` фреймворк определяет методы `getValue` и `setValue`, удовлетворяющие соглашениям для делегатов в Kotlin:

```
operator fun <T> Column<T>.getValue(o: Entity, desc: KProperty<*>): T {
    // извлекает значение из базы данных
}
```

```
operator fun <T> Column<T>.setValue(o: Entity, desc: KProperty<*>, value: T) {
    // изменяет значение в базе данных
}
```

Свойство `Column` (`Users.name`) можно использовать в роли делегата для делегированного свойства (`name`). В этом случае выражение `user.age += 1` в вашем коде компилятор превратит в примерно такую инструкцию: `user.ageDelegate.setValue(user.ageDelegate.getValue() + 1)` (мы опустили параметры с экземплярами свойства и объекта). Методы `getValue` и `setValue` позаботятся об извлечении и изменении информации в базе данных.

Полную реализацию классов из этого примера можно найти в исходном коде фреймворка Exposed (https://github.com/JetBrains/Exposed). Мы вновь вернемся к этому фреймворку в главе 11, когда займемся исследованием приемов проектирования предметно-ориентированного языка (DSL).

7.6. Резюме

- Kotlin поддерживает перегрузку некоторых стандартных арифметических операторов путем определения функций с соответствующими именами, но не позволяет определять новые, нестандартные операторы.
- Операторы сравнения транслируются в вызовы методов `equals` и `compareTo`.
- Определив функции с именами `get`, `set` и `contains`, можно обеспечить поддержку операторов `[]` и `in`, чтобы сделать свой класс более похожим на коллекции в Kotlin.
- Создание диапазонов и итерации по коллекциям также осуществляются посредством соглашений.
- Мультидекларации позволяют инициализировать сразу несколько переменных, распаковывая единственный объект, что можно использовать для возврата нескольких значений из функций. Эта возможность автоматически поддерживается для классов данных, но вы можете также реализовать её в своем классе, определив функции с именами `componentN`.
- Делегирование свойств дает возможность повторно использовать логику хранения значений свойств, инициализации, чтения и изменения. Это очень мощный механизм для разработки фреймворков.
- Функция `lazy` из стандартной библиотеки позволяет просто реализовать отложенную инициализацию свойств.
- Функция `Delegates.observable` позволяет наблюдать за изменениями свойств.
- Делегированные свойства, использующие словари в роли делегатов, дают гибкую возможность создавать объекты с переменным набором атрибутов.

Глава 8

Функции высшего порядка: лямбда-выражения как параметры и возвращаемые значения

В этой главе:

- типы функций;
- функции высшего порядка и их применение для структурирования кода;
- встраиваемые (inline) функции;
- нелокальные возвраты и метки;
- анонимные функции.

В главе 5 мы познакомились с лямбда-выражениями, где исследовали это понятие в целом, и познакомились с функциями в стандартной библиотеке, использующими лямбда-выражения. Лямбда-выражения – замечательный инструмент создания абстракций, и их возможности не ограничиваются только коллекциями и другими классами из стандартной библиотеки. В этой главе вы узнаете, как создавать свои *функции высшего порядка* (higher-order functions), принимающие лямбда-выражения в аргументах и возвращающие их. Вы увидите, как такие функции помогают упростить код, избавиться от повторяющихся фрагментов кода и создавать ясные абстракции. Вы также познакомитесь со *встраиваемыми* (inline) функциями – мощной особенностью языка Kotlin, устраняющей накладные расходы, связанные с использованием лямбда-выражений, и обеспечивающей гибкое управление потоком выполнения в лямбда-выражениях.

8.1. Объявление функций высшего порядка

Ключевая идея этой главы – новое понятие: *функции высшего порядка*. Функциями высшего порядка называют функции, которые принимают другие функции в аргументах и/или возвращают их. В Kotlin функции могут быть представлены как обычные значения, в виде лямбда-выражений или ссылок на функции. То есть функция высшего порядка – это любая функция, которая принимает аргумент с лямбда-выражением или ссылкой на функцию и/или возвращает их. Например, функция `filter` из стандартной библиотеки принимает аргумент с функцией-предикатом и, соответственно, является функцией высшего порядка:

```
list.filter { x > 0 }
```

В главе 5 мы познакомились со множеством функций высшего порядка, объявленных в стандартной библиотеке Kotlin: `map`, `with` и другими. Теперь узнаем, как объявлять такие функции в своем коде. Для этого сначала рассмотрим типы функций.

8.1.1. Типы функций

Чтобы объявить функцию, принимающую лямбда-выражение в аргументе, нужно узнать, как объявить тип соответствующего параметра. Но перед этим рассмотрим более простой случай и сохраним лямбда-выражение в локальной переменной. Вы уже видели, как сделать это без объявления типа, полагаясь на механизм автоматического определения типов в Kotlin:

```
val sum = { x: Int, y: Int -> x + y }
val action = { println(42) }
```

В данном случае компилятор определит, что обе переменные – `sum` и `action` – имеют тип функции. Давайте посмотрим, как выглядит явное объявление типов этих переменных:

```
val sum: (Int, Int) -> Int = { x, y -> x + y }
val action: () -> Unit = { println(42) }
```

Функция, принимающая два параметра типа Int и возвращающая значение типа Int

Функция, не имеющая аргументов и ничего не возвращающая

Чтобы объявить тип функции, поместите типы параметров в круглые скобки, после которых добавьте оператор стрелки и тип значения, возвращаемого функцией (см. рис. 8.1).

Как вы помните, тип `Unit` указывает, что функция не возвращает осмысленного значения. Тип возвращаемого значения `Unit` можно опустить, объявляя обычную функцию, но в объявлениях типов функций всегда требуется явно указывать

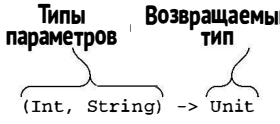


Рис. 8.1. Синтаксис объявления типа функции в Kotlin

тип возвращаемого значения, поэтому тип `Unit` нельзя опустить в данном контексте.

Обратите внимание, что в теле лямбда-выражения `{ x, y -> x + y }` можно опустить типы параметров `x` и `y`. Поскольку они указаны в объявлении типа функции, их не нужно повторять в самом лямбда-выражении.

Точно как в любой другой функции, тип возвращаемого значения в объявлении типа функции можно отметить как допускающий значение `null`:

```
var canReturnNull: (Int, Int) -> Int? = { null }
```

Также можно определить переменную, которая может принимать значение `null` и относиться к типу функции. Чтобы указать, что именно переменная, а не функция способна принимать значение `null`, нужно заключить определение типа функции в круглые скобки и добавить знак вопроса в конце:

```
var funOrNull: ((Int, Int) -> Int)? = null
```

Обратите внимание на тонкое отличие этого примера от предыдущего. Если опустить круглые скобки, получится тип функции, способной возвращать значение `null`, а не тип переменной, способной принимать значение `null`.

Имена параметров в типах функций

В типах функций допускается указывать имена параметров:

```
fun performRequest(
    url: String,
    callback: (code: Int, content: String) -> Unit
) {
    /*...*/
}

>>> val url = "http://kotl.in"
>>> performRequest(url) { code, content -> /*...*/ }
>>> performRequest(url) { code, page -> /*...*/ }
```

Определение типа функции может включать именованные параметры

Имена, указанные в определении, можно использовать как имена аргументов лямбда-выражений...

...или изменять их

Имена параметров не влияют на работу механизма контроля типов. Объявляя лямбда-выражение, вы не обязаны использовать те же имена параметров, что указаны в объявлении типа функции. Но имена улучшают читаемость и могут использоваться в IDE для автодополнения кода.

8.1.2. Вызов функций, переданных в аргументах

Теперь, узнав, как определять функции высшего порядка, обсудим особенности их реализации. В первом простейшем примере используется то

же объявление типа, что и в лямбда-выражении sum, которое было показано выше. Функция выполняет произвольную операцию с двумя числами, 2 и 3, и выводит результат.

Листинг 8.1. Определение простой функции высшего порядка

```
fun twoAndThree(operation: (Int, Int) -> Int) {
    val result = operation(2, 3)
    println("The result is $result")
}

>>> twoAndThree { a, b -> a + b }
The result is 5
>>> twoAndThree { a, b -> a * b }
The result is 6
```

Объявление параметра с типом функции

Вызов параметра с типом функции

Вызов обычной функции: указываются имя функции и список аргументов в круглых скобках.

Как более интересный пример определим свою реализацию функции filter из стандартной библиотеки. Для простоты ограничимся поддержкой только типа String, но обобщенная версия, способная работать с коллекциями любых элементов, выглядит похоже. Объявление функции показано на рис. 8.2.

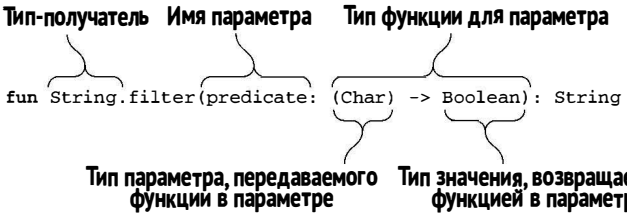


Рис. 8.2. Объявление функции filter с параметром-предикатом

Функция filter принимает предикат как параметр. Тип predicate – это функция, которая получает параметр из одного символа и возвращает логический результат: true, если символ удовлетворяет требованиям предиката и может быть включен в результирующую строку, и false в противном случае. Вот как такая функция может быть реализована.

Листинг 8.2. Реализация простой версии функции filter

```
fun String.filter(predicate: (Char) -> Boolean): String {
    val sb = StringBuilder()
    for (index in 0 until length) {
        val element = get(index)
        if (predicate(element)) sb.append(element)
    }
```

Вызов функции, переданной как аргумент для параметра «predicate»

```
    }
    return sb.toString()
}

>>> println("ab1c".filter { it in 'a'..'z' })
abc
```

Передается лямбда-выражение в аргументе для параметра «predicate»

Функция `filter` реализовывается очень просто: она проверяет каждый символ на соответствие предикату и в случае успеха добавляет в объект `StringBuilder`, содержащий результат.

Совет для пользователей IntelliJ IDEA

IntelliJ IDEA поддерживает пошаговое выполнение кода лямбда-выражения в отладчике. Если попробовать по шагам пройти предыдущий пример, можно увидеть, как выполняется тело функции `filter` и переданное ей лямбда-выражение по мере того, как функция обрабатывает каждый элемент во входной строке.

8.1.3. Использование типов функций в коде на Java

За кулисами кода типы функций объявляются как обычные интерфейсы: переменная, имеющая тип функции, – это реализация интерфейса `FunctionN`. В стандартной библиотеке Kotlin определено несколько интерфейсов с разным числом аргументов функции: `Function0<R>` (эта функция не принимает аргументов), `Function1<P1, R>` (принимает один аргумент) и так далее. Каждый интерфейс определяет единственный метод `invoke`, который вызывает функцию. Переменная, имеющая тип функции, – это экземпляр класса, реализующего соответствующий интерфейс `FunctionN`, метод `invoke` которого содержит тело лямбда-выражения.

Kotlin-функции, использующие типы функций, легко могут вызываться из кода на Java. В Java 8 лямбда-выражения автоматически преобразуются в значения типов функций:

```
/* Объявление в Kotlin */
fun processTheAnswer(f: (Int) -> Int) {
    println(f(42))
}

/* Java */
>>> processTheAnswer(number -> number + 1);
43
```

В более старых версиях Java можно передать экземпляр анонимного класса, реализующего метод `invoke` из соответствующего интерфейса:

```
/* Java */
>>> processTheAnswer(
```

```
...     new Function1<Integer, Integer>() {
...         @Override
...         public Integer invoke(Integer number) {
...             System.out.println(number);
...             return number + 1;
...         }
...     });
43
```

Использование Kotlin-типа функции из Java (ниже версии Java 8)

В Java легко использовать функции-расширения из стандартной библиотеки Kotlin, принимающие лямбда-выражения в аргументах. Но имейте в виду, что код получается не таким читабельным, как в Kotlin, – вы должны будете явно передать объект-получатель в первом аргументе:

```
/* Java */
>>> List<String> strings = new ArrayList();
>>> strings.add("42");
>>> CollectionsKt.forEach(strings, s -> {
...     System.out.println(s);
...     return Unit.INSTANCE;
... });
```

В Java-коде можно использовать функции из стандартной библиотеки Kotlin

Вы должны явно вернуть значение типа Unit

В Java ваша функция или лямбда-выражение может ничего не возвращать. Но в Kotlin такие функции возвращают значение типа `Unit`, и вы должны явно вернуть его. Нельзя передать лямбда-выражение, возвращающее `void`, в аргументе с типом функции, возвращающей `Unit` (как `(String) -> Unit` в предыдущем примере).

8.1.4. Значения по умолчанию и пустые значения для параметров типов функций

Объявляя параметр в типе функции, можно также указать его значение по умолчанию. Чтобы увидеть, где это может пригодиться, вернемся к функции `joinToString`, о которой мы говорили в главе 3. Вот реализация, на которой мы остановились.

Листинг 8.3. `joinToString` с жестко зашитым преобразованием `toString`

```
fun <T> Collection<T>.joinToString(
        separator: String = ", ",
        prefix: String = "",
        postfix: String = ""
): String {
    val result = StringBuilder(prefix)

    for ((index, element) in this.withIndex()) {
        if (index > 0) result.append(separator)
```

```
        result.append(element)
    }

    result.append(postfix)
    return result.toString()
}
```

Преобразует объект в строку с использованием реализации toString по умолчанию

Это гибкая реализация, но она не позволяет контролировать один важный аспект: преобразование в строки отдельных элементов коллекции. Код использует вызов `StringBuilder.append(o: Any?)`, который всегда преобразует объекты в строки с помощью метода `toString`. Это решение пригодно для большинства, но не для всех ситуаций. Теперь вы знаете, что можно передать лямбда-выражение, указывающее, как значения преобразовываются в строки. Но требование обязательной передачи такого лямбда-выражения слишком обременительно, потому что в большинстве случаев годится поведение по умолчанию. Для решения этого затруднения можно объявить параметр типа функции и определить его значение по умолчанию как лямбда-выражение.

Листинг 8.4. Объявление параметра с типом функции и значением по умолчанию

```
fun <T> Collection<T>.joinToString(
        separator: String = ", ",
        prefix: String = "",
        postfix: String = "",
        transform: (T) -> String = { it.toString() }
): String {
    val result = StringBuilder(prefix)

    for ((index, element) in this.withIndex()) {
        if (index > 0) result.append(separator)
        result.append(transform(element))
    }

    result.append(postfix)
    return result.toString()
}
```

Объявление параметра с типом функции и лямбда-выражением в качестве значения по умолчанию

Вызов функции, переданной в аргументе для параметра «transform»

```
>>> val letters = listOf("Alpha", "Beta")
>>> println(letters.joinToString())
Alpha, Beta
>>> println(letters.joinToString { it.toLowerCase() })
alpha, beta
>>> println(letters.joinToString(separator = "! ", postfix = "! ",
...     transform = { it.toUpperCase() }))
ALPHA! BETA!
```

Используется функция преобразования по умолчанию

Передается аргумент с лямбда-выражением

Использование синтаксиса именованных аргументов для передачи нескольких аргументов, включая лямбда-выражение

Обратите внимание, что это обобщенная функция: она имеет параметр типа T, обозначающий тип элемента в коллекции. Лямбда-выражение transform получит аргумент этого типа.

Чтобы объявить значение по умолчанию для типа функции, не требуется использовать специального синтаксиса – достаточно просто добавить лямбда-выражение после знака =. Примеры демонстрируют разные способы вызова функции: без лямбда-выражения (чтобы использовать преобразование по умолчанию с помощью toString()), с передачей за пределами круглых скобок и с передачей в именованном аргументе.

Альтернативный подход состоит в том, чтобы объявить параметр типа функции способным принимать значение null. Имейте в виду, что функцию, переданную в таком параметре, нельзя вызвать непосредственно: Kotlin откажется компилировать такой код, потому что обнаружит возможность появления исключения, вызванного обращением к пустому указателю. Одно из возможных решений – явно проверить на равенство значению null:

```
fun foo(callback: (() -> Unit)?) {
    // ...
    if (callback != null) {
        callback()
    }
}
```

Другое, более короткое решение основано на том, что тип функции – это реализация интерфейса с методом invoke. Обычный метод, каковым является invoke, можно вызвать, использовав синтаксис безопасного вызова: callback?.invoke().

Вот как можно использовать этот прием в измененной версии joinToString.

Листинг 8.5. Использование параметра с типом функции, способного принимать значение null

```
fun <T> Collection<T>.joinToString(
        separator: String = ", ",
        prefix: String = "",
        postfix: String = "",
        transform: ((T) -> String)? = null
): String {
    val result = StringBuilder(prefix)

    for ((index, element) in this.withIndex()) {
        if (index > 0) result.append(separator)
        val str = transform?.invoke(element)
```

Объявление параметра с типом функции, способного принимать значение null

Использование специального синтаксиса для безопасного вызова функции

```
            ?: element.toString()
        result.append(str)
    }

    result.append(postfix)
    return result.toString()
}
```

Использование оператора «Элвис» для обработки случая, когда функция для вызова не указана

Узнав, как писать функции, принимающие другие функции в аргументах, перейдем к следующей разновидности функций высшего порядка: к функциям, возвращающим другие функции.

8.1.5. Возврат функций из функций

Вернуть функцию из другой функции бывает нужно реже, чем передать функцию в аргументе, но такая возможность всё же может пригодиться. Представьте фрагмент программы, логика работы которого сильно зависит от состояния программы или от некоторых других условий, – например, расчет стоимости доставки в зависимости от выбранного транспорта. Для этого вы можете определить функцию, которая выбирает соответствующий вариант логики и возвращает его как ещё одну функцию. Вот как это выглядит в коде.

Листинг 8.6. Определение функции, возвращающей другую функцию

```
enum class Delivery { STANDARD, EXPEDITED }

class Order(val itemCount: Int)

fun getShippingCostCalculator(
        delivery: Delivery): (Order) -> Double {
    if (delivery == Delivery.EXPEDITED) {
        return { order -> 6 + 2.1 * order.itemCount }
    }

    return { order -> 1.2 * order.itemCount }
}

>>> val calculator =
...     getShippingCostCalculator(Delivery.EXPEDITED)
>>> println("Shipping costs ${calculator(Order(3))}")
Shipping costs 12.3
```

Объявление функции, возвращающей другую функцию

Возврат лямбда-выражения из функции

Сохранение полученной функции в переменной

Вызов полученной функции

Чтобы объявить функцию, возвращающую другую функцию, нужно указать, что возвращаемое значение имеет тип функции. Функция `getShippingCostCalculator` в листинге 8.6 возвращает функцию, которая принимает значение типа `Order` и возвращает `Double`. Чтобы вернуть функцию,

нужно записать выражение `return` и добавить лямбда-выражение, ссылку на член или другое выражение с типом функции – например, локальную переменную.

Рассмотрим ещё один пример, где пригодится приём возврата функций из других функций. Допустим, вы работаете над графическим интерфейсом приложения, управляющего контактами, и нужно определить, какие контакты отобразить на экране, ориентируясь на состояние пользовательского интерфейса. Представьте, что пользовательский интерфейс позволяет вводить строку и затем отображать контакты с именами, начинающимися с этой строки; он также позволяет скрывать контакты, для которых отсутствуют номера телефонов. Определим класс `ContactListFilters` для хранения параметров, управляющих отображением.

```
class ContactListFilters {
    var prefix: String = ""
    var onlyWithPhoneNumber: Boolean = false
}
```

Когда пользователь вводит символ `D`, чтобы просмотреть контакты, в которых имена начинаются на букву *D*, изменяется значение поля `prefix`. Мы опустили код, выполняющий необходимые изменения. (Было бы слишком расточительно приводить в книге полный код реализации пользовательского интерфейса, поэтому мы покажем упрощенный пример.)

Чтобы избежать тесной связи между логикой отображения списка контактов и фильтрацией, можно определить функцию, создающую предикат. Этот предикат должен проверить `prefix`, а также присутствие номера телефона, если требуется.

Листинг 8.7. Использование функции, возвращающей другую функцию

```
data class Person(
    val firstName: String,
    val lastName: String,
    val phoneNumber: String?
)

class ContactListFilters {
    var prefix: String = ""
    var onlyWithPhoneNumber: Boolean = false

    fun getPredicate(): (Person) -> Boolean {          <-- Объявление функции, возвращающей другую функцию
        val startsWithPrefix = { p: Person ->
            p.firstName.startsWith(prefix) || p.lastName.startsWith(prefix)
        }
        if (!onlyWithPhoneNumber) {
            return startsWithPrefix                    <-- Возврат переменной с типом функции
```

```
        }
        return { startsWithPrefix(it)
                    && it.phoneNumber != null }   <-- Возврат лямбда-выражения из этой функции
    }
}

>>> val contacts = listOf(Person("Dmitry", "Jemerov", "123-4567"),
...                       Person("Svetlana", "Isakova", null))
>>> val contactListFilters = ContactListFilters()
>>> with (contactListFilters) {
>>>     prefix = "Dm"
>>>     onlyWithPhoneNumber = true
>>> }
>>> println(contacts.filter(                          <-- Передача функции, полученной от getPredicate, в виде аргумента методу «filter»
...     contactListFilters.getPredicate()))
[Person(firstName=Dmitry, lastName=Jemerov, phoneNumber=123-4567)]
```

Метод `getPredicate` возвращает функцию, которая затем передается как аргумент функции `filter`. Типы функций в Kotlin позволяют выполнять такие манипуляции так же легко и просто, как со значениями других типов, например строками.

Функции высшего порядка – чрезвычайно мощный инструмент для структурирования кода и устранения повторяющихся фрагментов. Давайте посмотрим, как лямбда-выражения помогают убрать повторяющуюся логику.

8.1.6. Устранение повторяющихся фрагментов с помощью лямбда-выражений

Типы функций и лямбда-выражения вместе – это великолепный инструмент для создания кода многократного использования. Многие виды повторений кода, которые обычно можно предотвратить только за счет использования мудреных конструкций, теперь легко устранить с помощью лаконичных лямбда-выражений.

Рассмотрим пример анализа посещений веб-сайта. Класс `SiteVisit` хранит путь каждого посещения, его продолжительность и тип операционной системы пользователя. Типы операционных систем представлены в виде перечисления.

Листинг 8.8. Определение данных, описывающих посещение сайта

```
data class SiteVisit(
    val path: String,
    val duration: Double,
    val os: OS
```

```
)

enum class OS { WINDOWS, LINUX, MAC, IOS, ANDROID }

val log = listOf(
    SiteVisit("/", 34.0, OS.WINDOWS),
    SiteVisit("/", 22.0, OS.MAC),
    SiteVisit("/login", 12.0, OS.WINDOWS),
    SiteVisit("/signup", 8.0, OS.IOS),
    SiteVisit("/", 16.3, OS.ANDROID)
)
```

Допустим, нам нужно показать среднюю продолжительность визитов с компьютеров, действующих под управлением ОС Windows. Эту задачу можно решить с помощью функции `average`.

Листинг 8.9. Анализ данных, описывающих посещение сайта, с применением жестко заданных фильтров

```
val averageWindowsDuration = log
    .filter { it.os == OS.WINDOWS }
    .map(SiteVisit::duration)
    .average()

>>> println(averageWindowsDuration)
23.0
```

Теперь представим, что нам нужно вычислить ту же статистику для пользователей Mac. Чтобы избежать повторения кода, тип платформы можно выделить в параметр.

Листинг 8.10. Устранение повторений с помощью обычной функции

```
fun List<SiteVisit>.averageDurationFor(os: OS) =
    filter { it.os == os }.map(SiteVisit::duration).average()

>>> println(log.averageDurationFor(OS.WINDOWS))
23.0
>>> println(log.averageDurationFor(OS.MAC))
22.0
```

Повторяющийся код выделен в функцию

Обратите внимание, как оформление этой функции в виде расширения улучшает читаемость кода. Её можно даже объявить как локальную функцию-расширение, если она применима только в локальном контексте.

Но это не самое удачное решение. Представьте, что мы определяем среднюю продолжительность посещения пользователей мобильных платформ (в настоящее время есть две такие платформы: iOS и Android).

Листинг 8.11. Анализ данных, описывающих посещение сайта, с применением сложного, жестко заданного фильтра

```
val averageMobileDuration = log
    .filter { it.os in setOf(OS.IOS, OS.ANDROID) }
    .map(SiteVisit::duration)
    .average()

>>> println(averageMobileDuration)
12.15
```

Теперь простого параметра, представляющего тип платформы, оказалось недостаточно. Кроме того, в какой-то момент понадобится проанализировать журнал с использованием ещё более сложных условий – например, «средняя продолжительность посещения страницы регистрации пользователями iOS». Эта проблема решается с помощью лямбда-выражений. Мы можем использовать типы функций для выделения требуемых условий в параметр.

Листинг 8.12. Устранение повторений с помощью функции высшего порядка

```
fun List<SiteVisit>.averageDurationFor(predicate: (SiteVisit) -> Boolean) =
    filter(predicate).map(SiteVisit::duration).average()

>>> println(log.averageDurationFor {
...     it.os in setOf(OS.ANDROID, OS.IOS) })
12.15
>>> println(log.averageDurationFor {
...     it.os == OS.IOS && it.path == "/signup" })
8.0
```

Типы функций помогают избавиться от повторяющихся фрагментов кода. Если вы испытываете желание скопировать и вставить какой-то фрагмент кода, знайте, что этого повторения почти наверняка можно избежать. Лямбда-выражения позволяют извлекать не только повторяющиеся данные, но также поведение.

> **Примечание.** Применение типов функций и лямбда-выражений помогает упростить некоторые хорошо известные шаблоны проектирования. Возьмем шаблон «Стратегия» (Strategy). Без лямбда-выражений он требует объявить интерфейс с несколькими реализациями, по одной для каждой стратегии. При наличии в языке поддержки типов функций их можно использовать для описания стратегии и передавать разные лямбда-выражения в виде стратегий.

Мы обсудили порядок создания функций высшего порядка, а теперь посмотрим, как они влияют на производительность. Станет ли наш код мед-

леннее, если мы повсюду начнем использовать функции высшего порядка вместо старых добрых циклов и условных операторов? В следующем разделе мы объясним, почему ответ на этот вопрос не всегда утвердительный и как в некоторых ситуациях может помочь ключевое слово `inline`.

8.2. Встраиваемые функции: устранение накладных расходов лямбда-выражений

Вы могли заметить, что в Kotlin краткий синтаксис передачи лямбда-выражений в аргументах напоминает синтаксис обычных инструкций `if` и `for`, и видели примеры в главе 5, когда мы обсуждали функции `with` и `apply`. Но что можно сказать о производительности? Разве мы не создаем неприятных неожиданностей, определяя функции, которые похожи на обычные инструкции Java, но выполняются намного медленнее?

В главе 5 мы объяснили, что лямбда-выражения обычно компилируются в анонимные классы. Но это означает, что каждый раз, когда используется лямбда-выражение, создается дополнительный класс; и если лямбда-выражение хранит какие-то переменные, для каждого вызова создается новый объект. Это влечет дополнительные накладные расходы, ухудшающие эффективность реализации с лямбда-выражениями по сравнению с функцией, которая выполняет тот же код непосредственно.

Может ли компилятор сгенерировать код, не уступающий по эффективности инструкциям Java и всё ещё позволяющий извлекать повторяющийся код в библиотечные функции? Действительно, компилятор Kotlin поддерживает такую возможность. Если отметить функцию модификатором `inline`, компилятор не будет генерировать вызов функции в месте её использования, а просто вставит код её реализации. Давайте разберемся, как это работает, и рассмотрим конкретные примеры.

8.2.1. Как работает встраивание функций

Когда функция объявляется с модификатором `inline`, её тело становится встраиваемым – иными словами, оно подставляется вместо обычного вызова функции. Давайте посмотрим, какой код получается в результате.

Функция в листинге 8.13 гарантирует доступность общего ресурса только в одном потоке выполнения. Функция захватывает (блокирует) объект `Lock`, выполняет заданный блок кода и затем освобождает блокировку.

Листинг 8.13. Определение встраиваемой функции

```
inline fun <T> synchronized(lock: Lock, action: () -> T): T {
    lock.lock()
    try {
        return action()
```

```
    }
    finally {
        lock.unlock()
    }
}

val l = Lock()
synchronized(l) {
    // ...
}
```

Синтаксис вызова этой функции выглядит в точности как инструкция `synchronized` в Java. Разница в том, что в Java инструкция `synchronized` может использоваться с любым объектом, тогда как эта функция требует передачи экземпляра блокировки `Lock`. Определение выше – лишь пример; стандартная библиотека Kotlin определяет ещё одну версию функции `synchronized`, принимающую произвольный объект.

Но явное использование блокировок для синхронизации позволяет писать более надежный и понятный код. В разделе 8.2.5 мы познакомимся с функцией `withLock` из стандартной библиотеки Kotlin, которую желательно использовать всегда, когда требуется выполнить некоторую операцию под защитой блокировки.

Так как мы объявили функцию `synchronized` встроенной (`inline`), её тело будет вставлено в каждом месте вызова, подобно инструкции `synchronized` в Java. Рассмотрим следующий пример использования `synchronized()`:

```
fun foo(l: Lock) {
    println("Before sync")
    synchronized(l) {
        println("Action")
    }
    println("After sync")
}
```

На рис. 8.3 приводится эквивалентный код, который будет скомпилирован в тот же байт-код.

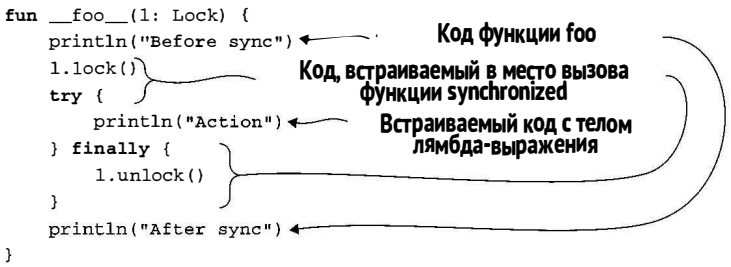


Рис. 8.3. Скомпилированная версия функции foo

Обратите внимание, что встраивание применяется не только к реализации функции `synchronized`, но и к телу лямбда-выражения. Байт-код, сгенерированный для лямбда-выражения, становится частью определения вызывающей функции и не заключается в анонимный класс, реализующий интерфейс функции.

Заметьте также, что для встраиваемых функций сохранилась возможность передавать параметры с типом функции из переменных:

```
class LockOwner(val lock: Lock) {
    fun runUnderLock(body: () -> Unit) {
        synchronized(lock, body)          <┐ Переменная с типом функции передается
    }                                       │ как аргумент, но не как лямбда-выражение
}
```

В данном случае код лямбда-выражения недоступен в точке вызова функции, поэтому его нельзя встроить. Встроится только тело функции `synchronized`, а лямбда-выражение будет вызвано как обычно. Функция `runUnderLock` будет скомпилирована в байт-код, который выглядит как функция ниже:

```
class LockOwner(val lock: Lock) {
    fun __runUnderLock__(body: () -> Unit) {     <┐ Эта функция компилируется в такой
        lock.lock()                                │ же байт-код, что и настоящая функция
        try {                                      │ runUnderLock
            body()                  <┐ Функция body не встраивается,
        }                            │ потому что лямбда-выражение
        finally {                    │ отсутствует в точке вызова
            lock.unlock()
        }
    }
}
```

Если использовать встраиваемую функцию в двух разных местах с разными лямбда-выражениями, каждый вызов встроится независимо. Код встраиваемой функции скопируется в оба места, но с разными лямбда-выражениями.

8.2.2. Ограничения встраиваемых функций

Из-за особенностей встраивания не всякая функция, использующая лямбда-выражения, может быть встраиваемой. Когда функция объявлена встраиваемой, тело лямбда-выражения, переданное в аргументе, встраивается непосредственно в конечный код. Это обстоятельство ограничивает возможные варианты использования соответствующего параметра в теле функции. Если функция, переданная в качестве параметра, вызывается, её код легко можно встроить в точку вызова. Но если параметр сохраняется где-то для последующего использования, код лямбда-выражения невоз-

можно встроить, потому что должен существовать объект, содержащий этот код.

В общем случае параметр можно встроить, если его вызывают непосредственно или передают как аргумент другой встраиваемой функции. Иначе компилятор запретит встраивание параметра с сообщением об ошибке: «Illegal usage of inline-parameter» (недопустимое использование встраиваемого параметра).

Например, некоторые функции, работающие с последовательностями, возвращают экземпляры классов, которые представляют операции с последовательностями и принимают лямбда-выражение в параметре конструктора. Например, ниже приводится определение функции `Sequence.map`:

```
fun <T, R> Sequence<T>.map(transform: (T) -> R): Sequence<R> {
    return TransformingSequence(this, transform)
}
```

Функция `map` не вызывает функцию в параметре `transform`. Она передает её конструктору класса, который сохранит эту функцию в своем свойстве. Чтобы обеспечить такую возможность, лямбда-выражение в аргументе `transform` должно быть скомпилировано в стандартное, невстраиваемое представление – анонимный класс, реализующий интерфейс функции.

Если имеется функция, принимающая несколько лямбда-выражений в аргументах, можно выбрать для встраивания только некоторые из них. Это имеет смысл, когда одно из лямбда-выражений содержит много кода или используется так, что не допускает встраивания. Соответствующие параметры можно объявить невстраиваемыми, добавив модификатор `noinline`:

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) {
    // ...
}
```

Обратите внимание, что компилятор полностью поддерживает встраивание функций, объявленных в других модулях или реализованных в сторонних библиотеках. Кроме того, большинство встраиваемых функций может вызываться из кода на Java, но такие вызовы будут не встраиваться, а компилироваться как вызовы обычных функций.

Далее, в разделе 9.2.4, вы увидите, как ещё уместно использовать модификатор `noinline` (что, впрочем, обусловлено ограничениями совместимости с Java).

8.2.3. Встраивание операций с коллекциями

Теперь обсудим производительность функций из стандартной библиотеки Kotlin, работающих с коллекциями. Большинство таких функций принимает лямбда-выражения в аргументах. Будут ли аналогичные опе-

рации, реализованные непосредственно, работать эффективнее функций из стандартной библиотеки?

Например, сравним две реализации списков людей в листингах 8.14 и 8.15.

Листинг 8.14. Фильтрация списка с применением лямбда-выражения

```
data class Person(val name: String, val age: Int)

val people = listOf(Person("Alice", 29), Person("Bob", 31))

>>> println(people.filter { it.age < 30 })
[Person(name=Alice, age=29)]
```

Код в листинге 8.14 можно переписать без использования лямбда-выражения, как видно в листинге 8.15.

Листинг 8.15. Фильтрация списка вручную

```
>>> val result = mutableListOf<Person>()
>>> for (person in people) {
>>>     if (person.age < 30) result.add(person)
>>> }
>>> println(result)
[Person(name=Alice, age=29)]
```

Функция `filter` в Kotlin объявлена как встраиваемая. Это означает, что код функции `filter` вместе с кодом заданного лямбда-выражения будет встраиваться в точку вызова `filter`. В результате для первой версии, использующей функцию `first`, будет сгенерирован примерно такой же байт-код, как для второй. Вы можете уверенно использовать идиоматичные операции с коллекциями, а поддержка встраиваемых функций в Kotlin избавит от беспокойств о производительности.

Попробуем теперь применить две операции, `filter` и `map`, друг за другом.

```
>>> println(people.filter { it.age > 30 }
...                .map(Person::name))
[Bob]
```

В этом примере используются лямбда-выражение и ссылка на член. Снова обе функции – `filter` и `map` – объявлены встраиваемыми, поэтому их тела будут встроены в конечный код без создания дополнительных классов. Но этот код создает промежуточную коллекцию, где хранится результат фильтрации списка. Код, сгенерированный из функции `filter`, добавляет элементы в эту коллекцию, а код, сгенерированный из функции `map`, читает их.

Если число обрабатываемых элементов велико и накладные расходы на создание промежуточной коллекции становятся слишком ощутимыми, можно воспользоваться последовательностью, добавив в цепочку вызов `asSequence`. Но в этом случае лямбда-выражения не будут встраиваться (как было показано в предыдущем разделе). Каждая промежуточная последовательность представлена объектом, хранящим лямбда-выражение в своем поле, и заключительная операция заставляет выполниться всю цепочку вызовов в последовательности. Поэтому, хотя операции в последовательности откладываются, вы не должны стремиться вставлять `asSequence` в каждую цепочку операций с коллекциями в своём коде. Этот приём дает преимущество только при работе с большими коллекциями, а маленькие коллекции можно обрабатывать как обычно.

8.2.4. Когда следует объявлять функции встраиваемыми

После знакомства с преимуществами ключевого слова `inline` у вас может появиться желание чаще использовать его для ускорения работы вашего кода. Но оказывается, что это не лучшее решение. Ключевое слово `inline` способно повысить производительность только функций, принимающих лямбда-выражения, а в остальных случаях требуется потратить время и оценить выигрыш.

Для вызовов обычных функций виртуальная машина JVM уже использует оптимизированный механизм встраивания. Она анализирует выполнение кода и выполняет встраивание, если это дает преимущество. Это происходит автоматически, когда байт-код транслируется в машинный код. В байт-коде реализация каждой функции присутствует в единственном экземпляре, и её не требуется копировать во все точки вызова, как в случае со встраиваемыми функциями в языке Kotlin. Более того, трассировка стека выглядит яснее, если функция вызывается непосредственно.

С другой стороны, встраивание функций с лямбда-выражениями в аргументах дает определенные выгоды. Во-первых, сокращаются накладные расходы. Время экономится не только на вызовах, но на создании дополнительных классов для лямбда-выражений и их экземпляров. Во-вторых, в настоящее время JVM не в состоянии выполнить встраивание всех вызовов с лямбда-выражениями. Наконец, встраивание позволяет использовать особенности, которые не поддерживаются для обычных лямбда-выражений, такие как нелокальные возвраты, которые мы обсудим далее в этой главе.

Кроме того, принимая решение об использовании модификатора `inline`, не нужно забывать о размере кода. Если функция, которую вы собираетесь объявить встраиваемой, достаточно велика, копирование её байт-кода во все точки вызова может существенно увеличить общий размер байт-кода. В этом случае попробуйте выделить код, не связан-

ный с лямбда-выражением в аргументе, в отдельную, невстраиваемую функцию. Загляните в стандартную библиотеку Kotlin и убедитесь, что все встраиваемые функции в ней имеют очень маленький размер.

Далее мы посмотрим, как функции высшего порядка помогают усовершенствовать код.

8.2.5. Использование встраиваемых лямбда-выражений для управления ресурсами

Управление ресурсами – захват ресурса перед операцией и его освобождение после – одна из областей, где применение лямбда-выражений помогает избавиться от повторяющегося кода. Ресурсом в данном случае может считаться что угодно: файл, блокировка, транзакция в базе данных и так далее. Стандартная реализация такого шаблона заключается в использовании инструкции `try/finally`, в которой ресурс захватывается перед блоком `try` и освобождается в блоке `finally`.

Выше в этом разделе был пример того, как можно заключить логику инструкции `try/finally` в функцию и передать в эту функцию код использования ресурса в виде лямбда-выражения. В этом примере демонстрировалась реализация функции `synchronized` с тем же синтаксисом, что и инструкция `synchronized` в Java: она принимает аргумент с объектом блокировки. В стандартной библиотеке Kotlin есть ещё одна функция – `withLock`, предлагающая более идиоматичный синтаксис для решения той же задачи: она – функция-расширение для интерфейса `Lock`. Ниже – пример её использования:

```
val l: Lock = ...
l.withLock {
    // операции с ресурсом под защитой данной блокировки
}
```

Выполняет заданную операцию под защитой блокировки

Вот как функция `withLock` определена в стандартной библиотеке Kotlin:

```
fun <T> Lock.withLock(action: () -> T): T {
    lock()
    try {
        return action()
    } finally {
        unlock()
    }
}
```

Идиома работы с блокировкой выделена в отдельную функцию

Файлы – это ещё одна распространенная разновидность ресурсов, для работы с которыми полезен этот шаблон. В Java 7 даже был добавлен специальный синтаксис для этого шаблона: инструкция try-with-resources.

В следующем листинге демонстрируется Java-метод, использующий эту инструкцию для чтения первой строки из файла.

Листинг 8.16. Использование try-with-resources в Java

```
/* Java */
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br =
                 new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

В Kotlin отсутствует эквивалентный синтаксис, потому что ту же задачу можно почти бесшовно решить с помощью функции, имеющей параметр с типом функции (то есть принимающей лямбда-выражение в аргументе). Эта функция входит в состав стандартной библиотеки Kotlin и называется `use`. Ниже показано, как её использовать, чтобы переписать пример в листинге 8.16 на языке Kotlin.

Листинг 8.17. Использование функции use для управления ресурсом

```
fun readFirstLineFromFile(path: String): String {
    BufferedReader(FileReader(path)).use { br ->
        return br.readLine()
    }
}
```

Создается экземпляр BufferedReader, вызывается функция «use», и ей передается лямбда-выражение, выполняющее операцию с файлом

Возвращается строка, прочитанная из файла

Функция `use` реализована как функция-расширение, которая применяется к закрываемым ресурсам; она принимает аргумент с лямбда-выражением. Функция вызывает лямбда-выражение и гарантирует закрытие ресурса вне зависимости от того, как выполнится лямбда-выражение – успешно или с исключением. Конечно, функция `use` объявлена как встраиваемая, поэтому она не ухудшает производительность.

Обратите внимание, что в теле лямбда-выражения используется нелокальный возврат, чтобы вернуть значение из функции `readFirstLineFromFile`. Давайте подробнее обсудим применение выражений `return` в лямбда-выражениях.

8.3. Порядок выполнения функций высшего порядка

Используя лямбда-выражения взамен императивных конструкций (таких как циклы), легко можно столкнуться с проблемой выражений `return`. Инструкция `return` в середине цикла не вызывает недоразумений. Но что

случится, если такой цикл преобразовать в вызов функции, например `filter`? Как будет работать `return` в этом случае? Давайте выясним это на примерах.

8.3.1. Инструкции «return» в лямбда-выражениях: выход из вмещающей функции

Сравним два разных способа итераций по элементам коллекции. Взглянув на код в листинге 8.18, можно сразу же сказать, что при встрече с именем `Alice` функция `lookForAlice` сразу же вернет управление вызывающему коду.

Листинг 8.18. Использование `return` в обычном цикле

```
data class Person(val name: String, val age: Int)

val people = listOf(Person("Alice", 29), Person("Bob", 31))

fun lookForAlice(people: List<Person>) {
    for (person in people) {
        if (person.name == "Alice") {
            println("Found!")
            return
        }
    }

    println("Alice is not found")    <- Функция выведет этот текст, если имя «Alice» не будет встречено в коллекции «people»
}

>>> lookForAlice(people)
Found!
```

Можно ли безбоязненно использовать этот код в вызове функции `forEach`? Как будет работать инструкция `return` в этом случае? Да, вы можете уверенно использовать этот код в вызове `forEach`, как показано ниже.

Листинг 8.19. Использование `return` в лямбда-выражении, передаваемом в вызов `forEach`

```
fun lookForAlice(people: List<Person>) {
    people.forEach {
        if (it.name == "Alice") {
            println("Found!")
            return                  <- Выполнит выход из функции, так же как в листинге 8.18
        }
```

```
    }

    println("Alice is not found")
}
```

Ключевое слово `return` в лямбда-выражении *производит выход из функции, в которой вызывается это лямбда-выражение*, а не из него самого. То есть инструкция `return` производит *нелокальный возврат*, потому что возвращает результат из внешнего блока, а не из блока, содержащего `return`.

Чтобы понять логику этого правила, представьте, что ключевое слово `return` используется в цикле `for` или в блоке `synchronized` внутри Java-метода. Очевидно, что оно должно произвести выход из функции, а не из цикла или блока. Язык Kotlin переносит эту логику на функции, принимающие лямбда-выражения в аргументах.

Обратите внимание, что выход из внешней функции выполняется *только тогда, когда функция, принимающая лямбда-выражение, является встраиваемой*. Тело функции `forEach` в листинге 8.19 встраивается вместе с телом лямбда-выражения, поэтому выражение `return` производит выход из вмещающей функции. Использование выражения `return` в лямбда-выражениях, передаваемых невстраиваемым функциям, недопустимо. Невстраиваемая функция может сохранить переданное ей лямбда-выражение в переменной, чтобы выполнить его позже, когда она уже завершится, то есть когда лямбда-выражение и его инструкция `return` окажутся в другом контексте.

8.3.2. Возврат из лямбда-выражений: возврат с помощью меток

Лямбда-выражения поддерживают также *локальный* возврат. Локальный возврат в лямбда-выражении напоминает своим действием выражение `break` в цикле `for`. Он прерывает работу лямбда-выражения и продолжает выполнение с инструкции, следующей сразу за вызовом лямбда-выражения. Чтобы отличить локальный возврат от нелокального, используются *метки*. Можно отметить лямбда-выражение, из которого требуется произвести выход, и затем сослаться на метку в выражении `return`.

Листинг 8.20. Локальный возврат с использованием метки

```
fun lookForAlice(people: List<Person>) {
    people.forEach label@{
        if (it.name == "Alice") return@label
    }
    println("Alice might be somewhere")
}
```

Метка для лямбда-выражения

return@label ссылается на эту метку

Эта строка выводится всегда

```
>>> lookForAlice(people)
Alice might be somewhere
```

Выражение «this» с меткой

Те же правила использования меток относятся к выражению `this`. В главе 5 мы обсудили лямбда-выражения с получателями – лямбда-выражениями, содержащими неявный объект контекста, доступный по ссылке `this` внутри лямбда-выражения (в главе 11 мы объясним, как писать собственные функции, принимающие лямбда-выражения с получателями в качестве аргументов). Если добавить метку к лямбда-выражению с получателем, появится возможность обратиться к неявному получателю с использованием выражения `this` и соответствующей метки:

```
>>> println(StringBuilder().apply sb@{
...     listOf(1, 2, 3).apply {
...         this@sb.append(this.toString())
...     }
... })
[1, 2, 3]
```

Этот неявный приемник лямбда-выражения доступен по ссылке this@sb

Ссылка «this» указывает на неявный приемник, ближайший в текущей области видимости

Доступны могут быть все неявные приемники, но самые внешние – только посредством явных меток

По аналогии с метками для выражений `return` метку для `this` можно задать явно или воспользоваться именем вмещающей функции.

Чтобы отметить лямбда-выражение, добавьте имя метки (может быть любым допустимым идентификатором) со следующим за ним символом @ перед открывающей фигурной скобкой лямбда-выражения. Чтобы произвести выход из лямбда-выражения, добавьте символ @ и имя метки сразу после ключевого слова `return`, как это показано на рис. 8.4.

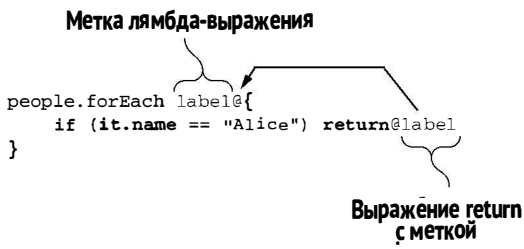


Рис. 8.4. Возврат из лямбда-выражения с использованием символа «@» и метки

Также в роли метки может использоваться имя функции, принимающей лямбда-выражение.

Листинг 8.21. Использование имени функции в роли метки для выражения `return`

```
fun lookForAlice(people: List<Person>) {
    people.forEach {
        if (it.name == "Alice") return@forEach
    }
```

return@forEach произведет выход из лямбда-выражения

```
    println("Alice might be somewhere")
}
```

Обратите внимание, что если явно снабдить лямбда-выражение собственной меткой, использование имени функции в качестве метки не даст желаемого результата. Лямбда-выражение не может иметь несколько меток.

Синтаксис нелокального возврата избыточно многословен и может вводить в заблуждение, если лямбда-выражение содержит несколько выражений `return`. Исправить недостаток можно применением альтернативного синтаксиса передачи блоков кода: *анонимных функций*.

8.3.3. Анонимные функции: по умолчанию возврат выполняется локально

Анонимная функция – другой способ оформления блока кода для передачи в другую функцию. Начнем с примера.

Листинг 8.22. Использование `return` в анонимных функциях

```
fun lookForAlice(people: List<Person>) {
    people.forEach(fun (person) {
        if (person.name == "Alice") return
        println("${person.name} is not Alice")
    })
}
```

Использует анонимную функцию вместо лямбда-выражения

«return» относится к ближайшей функции: анонимной функции

```
>>> lookForAlice(people)
Bob is not Alice
```

Как видите, анонимная функция напоминает обычную функцию, отличаясь только отсутствием имени и типа в объявлении параметра. Вот ещё один пример.

Листинг 8.23. Использование анонимной функции в вызове `filter`

```
people.filter(fun (person): Boolean {
    return person.age < 30
})
```

Анонимные функции следуют тем же правилам, что и обычные функции, и должны определять тип возвращаемого значения. Анонимные функции с телом-блоком, как в листинге 8.23, должны явно возвращать значение заданного типа. В функции с телом-выражением инструкцию `return` можно опустить.

Листинг 8.24. Использование анонимной функции с телом-выражением

```
people.filter(fun (person) = person.age < 30)
```

Внутри анонимной функции выражение `return` без метки выполнит выход из анонимной функции, а не из внешней. Здесь действует простое правило: `return` *производит выход из ближайшей функции, объявленной с помощью ключевого слова* `fun`. Лямбда-выражения определяются без ключевого слова `fun`, поэтому `return` в лямбда-выражениях производит выход из внешней функции. Анонимные функции объявляются с помощью `fun`; поэтому в предыдущем примере ближайшая функция для `return` – это анонимная функция. Следовательно, выражение `return` производит возврат из анонимной, а не из вмещающей функции. Разница видна на рис. 8.5.

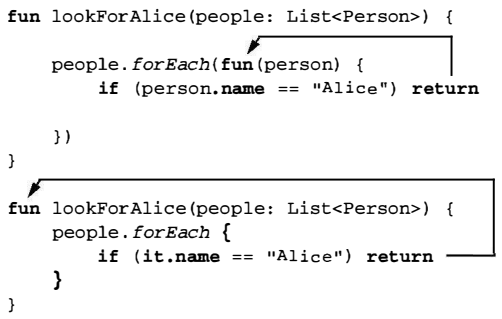


Рис. 8.5. Выражение return производит выход из ближайшей функции, объявленной с помощью ключевого слова fun

Обратите внимание: несмотря на то что анонимная функция выглядит как объявление обычной функции, в действительности она – лишь другая синтаксическая форма лямбда-выражения. Правила реализации лямбда-выражений и их встраивания во встраиваемых функциях точно так же действуют для анонимных функций.

8.4. Резюме

- Типы функций позволяют объявлять переменные, параметры или возвращаемые значения, хранящие ссылки на функции.
- Функции высшего порядка принимают другие функции в аргументах и/или возвращают их. Такие функции создаются с применением типов функций в роли типов параметров и/или возвращаемых значений.
- Когда производится компиляция вызова встраиваемой функции, её байт-код вместе с байт-кодом переданного ей лямбда-выражения

вставляется непосредственно в код, в точку вызова. Благодаря этому вызов происходит без накладных расходов, как если бы аналогичный код был написан явно.

- Функции высшего порядка упрощают повторное использование кода и позволяют писать мощные обобщенные библиотеки.
- Встраиваемые функции дают возможность производить *нелокальный возврат*, когда выражение `return` в лямбда-выражении производит выход из вмещающей функции.
- Анонимные функции – это альтернативный синтаксис оформления лямбда-выражений с иными правилами для выражения `return`. Их можно использовать, когда потребуется написать блок кода с несколькими точками выхода.

Глава 9

Обобщенные типы

В этой главе:

- объявление обобщенных функций и классов;
- стирание типов и овеществляемые параметры типов;
- определение вариантности в месте объявления и в месте использования.

Вы видели в этой книге несколько примеров использования обобщенных типов. Они были понятны без пояснений, поскольку основные идеи объявления и использования обобщенных классов и функций в Kotlin напоминают Java. В этой главе мы рассмотрим их подробнее. Мы глубже погрузимся в тему обобщенных типов и исследуем новые понятия, введенные в Kotlin: овеществляемые типовые параметры и определение вариантности в месте объявления. Они могут оказаться новыми для вас, но не волнуйтесь – глава раскрывает эту тему во всех подробностях.

Овеществляемые типовые параметры (reified type parameters) позволяют ссылаться во время выполнения на конкретные типы, используемые как типовые аргументы в вызовах встраиваемых функций. (Для обычных классов или функций это невозможно, потому что типовые аргументы стираются во время выполнения.)

Определение вариантности в месте объявления (declaration-site variance) позволяет указать, является ли обобщенный тип, имеющий типовой аргумент, подтипом или супертипом другого обобщенного типа с тем же базовым типом и другим типовым аргументом. Например, таким способом можно указать на возможность передачи типа `List<Int>` в функцию, ожидающую `List<Any>`. *Определение вариантности в месте использования* (use-site variance) преследует ту же цель для конкретного использования обобщенного типа и, следовательно, решает те же задачи, что и метасимволы (wildcards) в Java.

9.1. Параметры обобщенных типов

Поддержка обобщенных типов дает возможность определять *параметризованные типы*. Когда создается экземпляр такого типа, параметр типа заменяется конкретным типом, который называется *типовым аргументом*. Например, если в программе есть переменная типа `List`, то полезно знать, элементы какого типа хранит этот список. Типовой параметр позволяет указать именно это – то есть вместо «эта переменная хранит список» можно сказать «эта переменная хранит список строк». Синтаксис в Kotlin, говорящий «список строк», выглядит как в Java: `List<String>`. Также можно объявить несколько типовых параметров для класса. Например, класс `Map` имеет типовые параметры для ключей и значений: `class Map<K, V>`, – и его можно инициализировать конкретными аргументами: `Map<String, Person>`. Итак, синтаксис выглядит в точности как в Java.

Типовые аргументы, как и типы вообще, могут выводиться компилятором Kotlin автоматически:

```
val authors = listOf("Dmitry", "Svetlana")
```

Поскольку оба значения, переданных в вызов функции `listOf`, это строки, то компилятор сообразит, что в этой инструкции создается список строк `List<String>`. С другой стороны, если понадобится создать пустой список, типовой аргумент вывести неоткуда, поэтому вам придется указать его явно. В случае создания списка у вас есть возможность выбора: определить тип в объявлении переменной или задать типовой аргумент для вызываемой функции, создающей список. Вот как это делается:

```
val readers: MutableList<String> = mutableListOf()

val readers = mutableListOf<String>()
```

Эти объявления эквивалентны. Обратите внимание, что функции создания коллекций рассматриваются в разделе 6.3.

> **Примечание.** В отличие от Java, Kotlin всегда требует, чтобы типовые аргументы указывались явно или могли быть выведены компилятором. Поддержка обобщенных типов появилась только в версии Java 1.5, поэтому в этом языке необходимо было обеспечить совместимость с существующим кодом, написанным для более старых версий, и дать возможность использовать обобщенные типы без типовых аргументов – так называемые *необработанные типы* (raw type). Например, в Java можно объявить переменную типа `List`, не указывая типа элементов списка. Но Kotlin изначально реализует обобщенные типы и не поддерживает необработанных типов, поэтому типовые аргументы должны определяться всегда.

9.1.1. Обобщенные функции и свойства

Если вы собираетесь написать функцию, работающую со списком, и хотите, чтобы она работала с любыми списками (обобщенными), а не только со списками элементов конкретного типа, вы должны определить *обобщенную функцию*. Обобщенная функция имеет свои типовые параметры. Такие типовые параметры замещаются конкретными типовыми аргументами во всех вызовах функции.

Большая часть библиотечных функций, работающих с коллекциями, – обобщенные. Например, рассмотрим объявление функции `slice` (рис. 9.1). Эта функция возвращает список, содержащий только элементы с индексами, входящими в указанный диапазон.

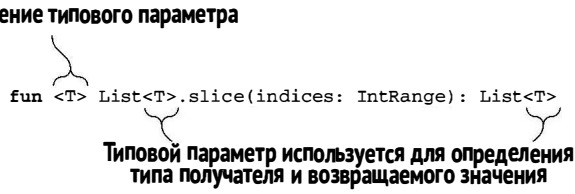


Рис. 9.1. Обобщенная функция slice имеет типовой параметр T

Параметр типа `T` функции используется для определения типа получателя и возвращаемого значения; для обоих тип определен как `List<T>`. Вызывая эту функцию для конкретного списка, можно явно передать типовой аргумент. Но это требуется далеко не всегда, потому что компилятор автоматически определяет типы, как показано далее.

Листинг 9.1. Вызов обобщенной функции

```
>>> val letters = ('a'..'z').toList()
>>> println(letters.slice<Char>(0..2))    <— Типовой аргумент задан явно
[a, b, c]
>>> println(letters.slice(10..13))        <— Компилятор сам определит, что в данном случае под T подразумевается тип Char
[k, l, m, n]
```

В обоих случаях функция будет определена как имеющая тип `List<Char>`. Компилятор подставит выведенный тип `Char` на место `T` в объявлении функции.

В разделе 8.1 мы показали объявление функции `filter`, которая принимает параметр с типом функции `(T) -> Boolean`. Давайте посмотрим, как применить её к переменным `readers` и `authors` из предыдущих примеров.

```
val authors = listOf("Dmitry", "Svetlana")
val readers = mutableListOf<String>(/* ... */)

fun <T> List<T>.filter(predicate: (T) -> Boolean): List<T>

>>> readers.filter { it !in authors }
```

В данном случае автоматический параметр `it` лямбда-выражения получит тип `String`. Компилятор способен сам вывести его: параметр с лямбда-выражением в объявлении функции имеет обобщенный тип `T` (это тип параметра функции в `(T) -> Boolean`). Компилятор понимает, что `T` – это `String`, потому что знает, что функция должна быть вызвана для `List<T>`, а `readers` (получатель) имеет фактический тип `List<String>`.

Типовые параметры можно объявлять для методов классов или интерфейсов, функций верхнего уровня и функций-расширений. В последнем случае типовой параметр можно использовать для определения типа получателя и параметров, как в листингах 9.1 и 9.2: типовой параметр `T` – часть типа получателя `List<T>`, а также используется в параметре с типом функции `(T) -> Boolean`.

Используя тот же синтаксис, можно объявить обобщенные свойства-расширения. Например, ниже определяется свойство-расширение, которое возвращает предпоследний элемент в списке:

```
val <T> List<T>.penultimate: T
    get() = this[size - 2]
```

Это обобщенное свойство-расширение доступно для списков любого типа

```
>>> println(listOf(1, 2, 3, 4).penultimate)
3
```

В данном вызове параметр типа T определяется как Int

Нельзя объявить обобщенное свойство, не являющееся расширением

Обычные свойства (не являющиеся расширениями) не могут иметь типовых параметров – нельзя сохранить несколько значений разных типов в свойстве класса, а поэтому не имеет смысла объявлять свойства обобщенного типа, не являющиеся расширениями. Если попробовать сделать это, то компилятор сообщит об ошибке:

```
>>> val <T> x: T = TODO()
ERROR: type parameter of a property must be used in its receiver type
```

Теперь посмотрим, как объявлять обобщенные классы.

9.1.2. Объявление обобщенных классов

Как в Java, в Kotlin имеется возможность объявлять обобщенные классы или интерфейсы, поместив угловые скобки после имени класса и указав в них типовые параметры. После этого типовые параметры можно использовать в теле класса, как любые другие типы. Давайте посмотрим, как можно объявить стандартный Java-интерфейс `List` на языке Kotlin. Для простоты мы опустим большую часть методов:

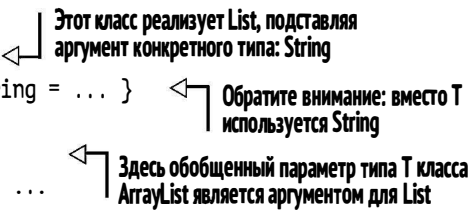
```
interface List<T> {
    operator fun get(index: Int): T
    // ...
}
```

Далее в этой главе, когда мы будем обсуждать тему вариантности, мы усовершенствуем этот пример и посмотрим, как интерфейс `List` объявлен в стандартной библиотеке Kotlin.

Если ваш класс наследует обобщенный класс (или реализует обобщенный интерфейс), вы должны указать типовой аргумент для обобщенного параметра базового типа. Это может быть или конкретный тип, или другой типовой параметр:

```
class StringList: List<String> {
    override fun get(index: Int): String = ... }

class ArrayList<T> : List<T> {
    override fun get(index: Int): T = ...
}
```



Класс `StringList` представляет коллекцию, которая может хранить только элементы типа `String`, поэтому в его определении используется аргумент типа `String`. Все функции в подклассе будут подставлять этот конкретный тип вместо `T`, то есть функция в классе `StringList` будет иметь сигнатуру `fun get(Int): String`, а не `fun get(Int): T`.

Класс `ArrayList` определяет свой типовой параметр `T` и использует его как типовой аргумент для суперкласса. Обратите внимание, что `T` в `ArrayList<T>` – это не то же самое, что в `List<T>`, – это другой типовой параметр, и он необязательно должен иметь то же имя.

В качестве типового аргумента для класса может даже использоваться сам класс. Классы, реализующие интерфейс `Comparable`, – классический пример этого шаблона. Любой элемент, поддерживающий сравнение, должен определять, как он будет сравниваться с объектами того же типа:

```
interface Comparable<T> {
    fun compareTo(other: T): Int
}

class String : Comparable<String> {
    override fun compareTo(other: String): Int = /* ... */
}
```

Класс `String` реализует обобщенный интерфейс `Comparable`, подставляя тип `String` для типового параметра `T`.

До сих пор мы не видели отличий обобщенных типов в Kotlin от их сородичей в Java. Но мы поговорим об этом далее, в разделах 9.2 и 9.3. А пока

обсудим ещё одно понятие, действующее в Kotlin так же, как в Java, и позволяющее писать удобные функции для работы с элементами, поддерживающими сравнение.

9.1.3. Ограничения типовых параметров

Ограничения типовых параметров позволяют ограничивать круг допустимых типов, которые можно использовать в качестве типовых аргументов для классов и функций. Например, рассмотрим функцию, вычисляющую сумму элементов списка. Она может использоваться со списками `List<Int>` или `List<Double>`, но не со списками, например, `List<String>`. Чтобы выразить это требование, можно определить ограничение для типового параметра, указав, что типовой параметр функции `sum` должен быть числом.

Когда какой-то тип определяется как *верхняя граница* для типового параметра, в качестве соответствующих типовых аргументов должен указываться либо именно этот тип, либо его подтипы. (Пока будем считать термин «подтип» синонимом термина «подкласс». В разделе 9.3.2 мы проведем границу между ними.)

Чтобы определить ограничение, нужно добавить двоеточие после имени типового параметра, а затем указать тип, который должен стать верхней границей, как показано на рис. 9.2. Чтобы выразить то же самое в Java, следует использовать ключевое слово `extends`: `<T extends Number> T sum(List<T> list)`.

Рис. 9.2. Для определения ограничения после имени типового параметра указывается верхняя граница

Следующий вызов функции допустим, потому что фактический типовой аргумент (`Int`) наследует `Number`:

```
>>> println(listOf(1, 2, 3).sum())
6
```

После определения верхней границы типового параметра `T` значения типа `T` можно использовать как значения типа верхней границы. Например, можно вызывать методы, объявленные в классе, который используется в качестве верхней границы:

```
fun <T : Number> oneHalf(value: T): Double {
    return value.toDouble() / 2.0
}
```

Тип Number служит верхней границей для типового параметра

Вызов метода, объявленного в классе Number

```
>>> println(oneHalf(3))
1.5
```

Теперь напишем обобщенную функцию, возвращающую наибольший из двух элементов. Поскольку определить максимальное значение можно только среди элементов, поддерживающих сравнение, это обстоятельство необходимо отразить в сигнатуре функции. Вот как это можно сделать.

Листинг 9.3. Объявление функции с ограничением для типового параметра

```
fun <T: Comparable<T>> max(first: T, second: T): T {
    return if (first > second) first else second
}

>>> println(max("kotlin", "java"))
kotlin
```

Аргументы этой функции должны быть элементами, поддерживающими сравнение

Строки сравниваются по алфавиту

Если попытаться вызвать `max` с элементами несовместимых типов, код просто не будет компилироваться:

```
>>> println(max("kotlin", 42))
ERROR: Type parameter bound for T is not satisfied:
 inferred type Any is not a subtype of Comparable<Any>
```

Верхняя граница для `T` – обобщенный тип `Comparable<T>`. Как вы видели выше, класс `String` реализует `Comparable<String>`, что делает `String` допустимым типовым аргументом для функции `max`.

Напомним, что, согласно соглашениям об операторах в Kotlin, сокращенная форма `first > second` компилируется в `first.compareTo(second) > 0`. Такое сравнение возможно, потому что тип элемента `first` (то есть `T`) наследует `Comparable<T>`, а значит, `first` можно сравнить с другим элементом типа `T`.

В тех редких случаях, когда требуется определить несколько ограничений для типового параметра, можно использовать несколько иной синтаксис. Например, обобщенный метод в листинге 9.4 гарантирует, что заданная последовательность `CharSequence` имеет точку в конце. Этот приём работает со стандартным классом `StringBuilder` и с классом `java.nio.CharBuffer`.

Листинг 9.4. Определение нескольких ограничений для типового параметра

```
fun <T> ensureTrailingPeriod(seq: T)
        where T : CharSequence, T : Appendable {
    if (!seq.endsWith('.')) {
        seq.append('.')
    }
}

>>> val helloWorld = StringBuilder("Hello World")
```

Список ограничений для типового параметра

Вызов функции-расширения, объявленной для интерфейса CharSequence

Вызов метода из интерфейса Appendable

```
>>> ensureTrailingPeriod(helloWorld)
>>> println(helloWorld)
Hello World.
```

В данном случае мы указали, что тип, определенный в качестве типового аргумента, должен реализовать два интерфейса: `CharSequence` и `Appendable`. Это означает, что со значениями данного типа можно использовать операции обращения к данным (`endsWith`) и их изменения (`append`).

Далее обсудим ещё один случай использования ограничений типовых параметров: когда требуется объявить, что типовой параметр не должен поддерживать значения `null`.

9.1.4. Ограничение поддержки null в типовом параметре

Когда объявляется обобщенный класс или функция, вместо его типового параметра может быть подставлен любой типовой аргумент, включая типы с поддержкой значения `null`. То есть типовой параметр без верхней границы на деле имеет верхнюю границу `Any?`. Рассмотрим следующий пример:

```
class Processor<T> {
    fun process(value: T) {
        value?.hashCode()
    }
}
```

«value» может иметь значение null, поэтому приходится использовать безопасный вызов

В функции `process` параметр `value` может принимать значение `null`, даже если его тип `T` не отмечен знаком вопроса. Это объясняется тем, что конкретные экземпляры класса `Processor` могут использовать тип с поддержкой значения `null` для `T`:

```
val nullableStringProcessor = Processor<String?>()
nullableStringProcessor.process(null)
```

На место T подставляется String? – тип, поддерживающий значение null

Этот код, передающий «null» в аргументе «value», прекрасно компилируется

Чтобы гарантировать замену типового параметра только типами, не поддерживающими значения `null`, необходимо объявить ограничение. Если это единственное требование к типу, то используйте в качестве верхней границы тип `Any`:

```
class Processor<T : Any> {
    fun process(value: T) {
        value.hashCode()
    }
}
```

Верхняя граница не допускает использования типов с поддержкой значения «null»

«value» типа T теперь не может иметь значения «null»

Ограничение `<T : Any>` гарантирует, что тип `T` всегда будет представлять тип без поддержки значения `null`. Код `Processor<String?>` будет

отвергнут компилятором, потому что аргумент типа `String?` не является подтипом `Any` (это подтип типа `Any?`, менее конкретного типа):

```
>>> val nullableStringProcessor = Processor<String?>()
Error: Type argument is not within its bounds: should be subtype of 'Any'
```

Обратите внимание, что такое ограничение можно наложить указанием в верхней границе любого типа, не поддерживающего `null`, не только типа `Any`.

Мы охватили основы обобщенных типов, наиболее схожие с Java. Теперь перейдем к теме, которая может показаться знакомой разработчикам на Java: поведение обобщенных типов во время выполнения.

9.2. Обобщенные типы во время выполнения: стирание и овеществление параметров типов

Как вы уже наверняка знаете, обобщенные типы реализуются в JVM через механизм *стирания типа* (type erasure) – то есть типовые аргументы экземпляров обобщенных классов не сохраняются во время выполнения. В этом разделе мы обсудим практические следствия стирания типов для Kotlin и то, как обойти эти ограничения, объявляя функции встраиваемыми (`inline`). Чтобы типовые аргументы не стирались (или, говоря терминами языка Kotlin, овеществлялись), функцию можно объявить встраиваемой. Мы подробно обсудим овеществляемые типовые параметры и рассмотрим примеры, когда это может пригодиться.

9.2.1. Обобщенные типы во время выполнения: проверка и приведение типов

Как и в Java, обобщенные типы в Kotlin *стираются* во время выполнения. То есть экземпляр обобщенного класса не хранит информацию о типовых аргументах, использованных для создания этого экземпляра. Например, если создать список `List<String>` и добавить в него несколько строк, во время выполнения вы увидите, что этот список имеет тип `List` – нет никакой возможности определить тип элементов списка. (Конечно, можно извлечь элемент списка и проверить его тип, но такая проверка не гарантирует, что другие элементы будут того же типа.)

Посмотрим, что происходит со следующими двумя списками во время выполнения (см. рис. 9.3):

```
val list1: List<String> = listOf("a", "b")
val list2: List<Int> = listOf(1, 2, 3)
```

Хотя компилятор видит два различных типа, во время выполнения списки выглядят совершенно одинаковыми. Несмотря на это, вы как програм-

мист можете гарантировать, что `List<String>` содержит только строки, а `List<Int>` – только целые числа, потому что компилятор знает типовые аргументы и может гарантировать сохранение в каждом списке только элементов совместимых типов. (Конечно, можно обмануть компилятор, применив приведение типов или используя необработанные типы Java в операциях доступа к спискам, но для этого потребуется приложить дополнительные усилия.)

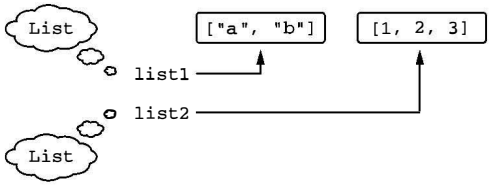


Рис. 9.3. Во время выполнения невозможно определить, что list1 и list2 объявлены как список строк и список целых чисел, – обе переменные имеют тип List

Теперь поговорим об ограничениях, которые появляются в связи со стиранием информации о типах. Так как типовые аргументы не сохраняются, вы не сможете проверить их – например, вы не сможете убедиться, что конкретный список состоит из строк, а не каких-то других объектов. Как следствие типы с типовыми аргументами нельзя использовать в проверках `is` – например, следующий код не компилируется:

```
>>> if (value is List<String>) { ... }
ERROR: Cannot check for instance of erased type
```

Во время выполнения возможно определить, что значение имеет тип `List`, но нельзя определить, что список хранит строки, объекты класса `Person` или что-то ещё, – эта информация стерта. Обратите внимание, что в стирании информации об обобщенном типе есть свои преимущества: поскольку нужно хранить меньше информации о типе, уменьшается объем памяти, занимаемый программой.

Как отмечалось выше, Kotlin не позволяет использовать обобщенных типов без типовых аргументов. Соответственно, возникает вопрос: как проверить, что значение является списком, а не множеством или каким-то другим объектом? Это можно сделать с помощью специального синтаксиса *проекций*:

```
if (value is List<*>) { ... }
```

Фактически вы должны указать * для каждого параметра типа, присутствующего в объявлении типа. Далее в этой главе мы подробно обсудим синтаксис проекций со звездочкой (и объясним, почему проекции называются *проекциями*), а пока просто рассматривайте такое объявление как тип с неизвестными аргументами (или как аналог `List<?>` в Java). Преды-

дущий пример проверяет, является ли `value` списком `List`, но не делает никаких предположений о типе элементов.

Обратите внимание, что в операторах приведения типов `as` и `as?` по-прежнему можно использовать обобщенные типы. Но такое приведение завершится удачно, если класс имеет совместимый базовый тип и несовместимые типовые аргументы, потому что когда производится приведение (во время выполнения), типовые аргументы неизвестны. По этой причине компилятор выводит предупреждение «unchecked cast» (неконтролируемое приведение) для каждой попытки такого приведения типов. Но это всего лишь предупреждение, поэтому вы сможете использовать значение как имеющее требуемый тип (см. листинг 9.5).

Листинг 9.5. Использование приведения с обобщенным типом

```
fun printSum(c: Collection<*>) {
    val intList = c as? List<Int>
            ?: throw IllegalArgumentException("List is expected")
    println(intList.sum())
}

>>> printSum(listOf(1, 2, 3))
6
```

Этот код породит предупреждение: Unchecked cast: List<*> to List<Int>

Все работает, как ожидается

Этот код прекрасно скомпилируется: компилятор выведет предупреждение, но признает этот код допустимым. Если вызвать функцию `printSum` со списком целых чисел или множеством, вы получите ожидаемое поведение: она выведет сумму в первом случае и возбудит исключение `IllegalArgumentException` во втором. Но если передать функции значение несовместимого типа, она возбудит исключение `ClassCastException`:

```
>>> printSum(setOf(1, 2, 3))
IllegalArgumentException: List is expected
>>> printSum(listOf("a", "b", "c"))
ClassCastException: String cannot be cast to Number
```

Множество – не список, поэтому возбуждается исключение

Приведение выполнено успешно, но потом возбуждается другое исключение

Обсудим исключение, которое возбуждается, когда функция `printSum` получает список строк. Она не возбудила исключения `IllegalArgumentException`, потому что невозможно убедиться, что типовой аргумент имел вид `List<Int>`. Из-за этого операция приведения типа не заметила ошибок, и была вызвана функция `sum`. Во время её выполнения возникло исключение, потому что `sum` пытается извлекать из списка значения типа `Number` и складывать их. Попытка использовать тип `String` взамен `Number` привела к появлению исключения `ClassCastException` во время выполнения.

Обратите внимание, что компилятор Kotlin достаточно интеллектуален, чтобы использовать в проверке `is` информацию о типе, доступную во время компиляции.

Листинг 9.6. Проверка типа с использованием известного типового аргумента

```
fun printSum(c: Collection<Int>) {
    if (c is List<Int>) {                    <- Такая проверка допустима
        println(c.sum())
    }
}

>>> printSum(listOf(1, 2, 3))
6
```

Проверка типа в листинге 9.6 возможна потому, что наличие в коллекции с (список или другая разновидность коллекций) целых чисел известно на этапе компиляции.

Компилятор Kotlin всегда старается сообщить, какие проверки опасны (отвергая проверки `is` и выводя предупреждения для приведений `as`), а какие допустимы. Вам остается только осознать смысл предупреждений и самостоятельно определить безопасность выполняемых операций.

Как уже упоминалось, в Kotlin есть специальная конструкция, чтобы использовать конкретные типовые аргументы в теле функции, – но такое допускается только внутри встраиваемых функций. Давайте познакомимся с этой особенностью.

9.2.2. Объявление функций с овеществляемыми типовыми параметрами

Как обсуждалось выше, обобщенные типы стираются во время выполнения. В результате вы получаете экземпляр обобщенного класса, для которого нельзя определить типового аргумента, использовавшегося при создании экземпляра. То же относится к типовым аргументам для функций. В теле обобщенной функции нельзя ссылаться на типовой аргумент, с которым она была вызвана:

```
>>> fun <T> isA(value: Any) = value is T
Error: Cannot check for instance of erased type: T
```

Это общее правило. Но есть одно исключение, позволяющее преодолеть данное ограничение: встраиваемые (`inline`) функции. Типовые параметры встраиваемых функций могут овеществляться – то есть во время выполнения можно ссылаться на фактические типовые аргументы.

Мы подробно обсудили встраиваемые функции в разделе 8.2. Тем не менее напомним: если добавить в объявление функции ключевое слово

inline, компилятор будет подставлять фактическую реализацию функции во все точки, где она вызывается. Объявление функции встраиваемой может способствовать увеличению производительности, особенно если функция принимает аргументы с лямбда-выражениями: код лямбда-выражений также может встраиваться, поэтому отпадает необходимость создавать анонимные классы. В этом разделе демонстрируется другой пример пользы от встраиваемых функций: их типовые аргументы могут овеществляться.

Если объявить предыдущую функцию isA с модификатором inline и отметить типовой параметр как овеществляемый (reified), появится возможность проверить, является ли value экземпляром типа T.

Листинг 9.7. Объявление функции с овеществляемым типовым параметром

```
inline fun <reified T> isA(value: Any) = value is T
```

Теперь этот код компилируется

```
>>> println(isA<String>("abc"))
true
>>> println(isA<String>(123))
false
```

Рассмотрим менее очевидные примеры использования овеществляемых типовых параметров. Один из них – функция filterIsInstance в стандартной библиотеке. Эта функция принимает коллекцию, выбирает из неё экземпляры заданного класса и возвращает только их. Вот как её можно использовать.

Листинг 9.8. Использование функции filterIsInstance из стандартной библиотеки

```
>>> val items = listOf("one", 2, "three")
>>> println(items.filterIsInstance<String>())
[one, three]
```

В этом коде мы указываем для функции типовой аргумент <String> и этим заявляем, что нас интересуют только строки. Соответственно, возвращаемое значение функции получает тип List<String>. В данном случае типовой аргумент известен во время выполнения и filterIsInstance использует его для проверки значений в списке.

Вот как выглядит упрощенная версия объявления функции filterIsInstance в стандартной библиотеке Kotlin.

Листинг 9.9. Упрощенная версия объявления функции filterIsInstance

```
inline fun <reified T>
        Iterable<*>.filterIsInstance(): List<T> {
```

Ключевое слово «reified» объявляет, что этот типовой параметр не будет стерт во время выполнения

```
    val destination = mutableListOf<T>()
    for (element in this) {
        if (element is T) {                    <- Функция может проверить принадлежность
            destination.add(element)              элемента к типу, указанному в типовом аргументе
        }
    }
    return destination
}
```

Почему овеществление возможно только для встраиваемых функций

Как работает этот механизм? Почему во встраиваемых функциях допускается проверка `element is T`, а в обычных классах или функциях – нет? Как было сказано в разделе 8.2, для встраиваемых функций компилятор вставляет байт-код с их реализацией в точки вызова. Каждый раз, когда в программе встречается вызов функции с овеществляемым типовым параметром, компилятор точно знает, какой тип используется в качестве типового аргумента для данного конкретного вызова. Соответственно, компилятор может сгенерировать байт-код, который ссылается на конкретный класс, указанный в типовом аргументе. В результате для вызова `filterIsInstance<String>`, показанного в листинге 9.8, генерируется код, эквивалентный следующему:

```
for (element in this) {
    if (element is String) {          <- Ссылается на
        destination.add(element)         конкретный класс
    }
}
```

Так как байт-код ссылается на конкретный класс, а не на типовой параметр, типовой аргумент не стирается во время выполнения.

Обратите внимание, что встраиваемые функции с овеществляемыми типовыми параметрами не могут быть вызваны из Java. Обычные встраиваемые функции можно вызвать из Java, но только как функции без встраивания. Напротив, функции с овеществляемыми параметрами типов требуют дополнительной обработки для подстановки значения типовых аргументов в байт-код, и поэтому они всегда должны быть встраиваемыми. А это делает невозможным их вызов таким способом, как это делает код на Java.

Встраиваемая функция может иметь несколько овеществляемых типовых параметров, а также неовеществляемые типовые параметры в дополнение к овеществляемым. Обратите внимание, что функция `filterIsInstance` объявлена встраиваемой, хотя она не принимает лямбда-выражений в аргументах. В разделе 8.2.4 отмечалось, что встраивание даёт преимущества по производительности, когда функция принимает параметры типов функций, а соответствующие аргументы – лямбда-выражения –

встраиваются вместе с функцией. Но в данном случае функция объявлена встраиваемой не ради этого, а чтобы дать возможность использовать овеществляемый типовой параметр.

Чтобы гарантировать высокую производительность, вы должны следить за размерами встраиваемых функций. Если функция становится слишком большой, то лучше выделить из неё фрагмент, не зависящий от овеществляемых типовых параметров, и превратить его в отдельную, невстраиваемую функцию.

9.2.3. Замена ссылок на классы овеществляемыми типовыми параметрами

Один из типичных примеров использования овеществляемых параметров функций – создание адаптеров для API, принимающих параметры типа `java.lang.Class`. Примером такого API может служить `ServiceLoader` из JDK, который принимает `java.lang.Class`, представляющий интерфейс или абстрактный класс, и возвращает экземпляр служебного класса, который реализует этот интерфейс. Посмотрим, как овеществляемые типовые параметры могут упростить работу с такими API.

Вот как выполняется загрузка службы с применением стандартного Java API `ServiceLoader`:

```
val serviceImpl = ServiceLoader.load(Service::class.java)
```

Синтаксис `::class.java` показывает, как получить `java.lang.Class`, соответствующий Kotlin-классу. Это точный эквивалент `Service.class` в Java. Мы подробно обсудим эту тему в разделе 10.2, когда будем обсуждать механизм рефлексии (отражения).

А теперь перепишем этот пример, использовав функцию с овеществляемым типовым параметром:

```
val serviceImpl = loadService<Service>()
```

Так намного короче, правда? Класс загружаемой службы теперь определяется как типовой аргумент для функции `loadService`. Типовой аргумент читается проще – ведь он короче, чем синтаксис `::class.java`.

Давайте посмотрим, как определена эта функция `loadService`:

```
inline fun <reified T> loadService() {          <— Параметр типа объявлен овеществляемым
    return ServiceLoader.load(T::class.java)    <— Обращение к классу с типом T::class
}
```

К овеществляемым типовым параметрам можно применять тот же синтаксис `::class.java`, что и для обычных классов. Этот синтаксис вернет `java.lang.Class`, соответствующий классу, указанному в типовом параметре, который затем можно использовать как обычно.

Упрощение функции startActivity в Android

Если вы разрабатываете приложения для Android, следующий пример покажется вам знакомым: он запускает экземпляр `Activity`. Вместо передачи класса в виде `java.lang.Class` вполне можно использовать овеществляемый типовой параметр:

```
inline fun <reified T : Activity>
        Context.startActivity() {
    val intent = Intent(this, T::class.java)
    startActivity(intent)
}

startActivity<DetailActivity>()
```

Типовой параметр объявлен овеществляемым

Обращение к классу с типом T::class

Вызов метода для отображения Activity

9.2.4. Ограничения овеществляемых типовых параметров

Несмотря на удобства, овеществляемые типовые параметры имеют определенные ограничения. Некоторые проистекают из самой идеи, другие обусловлены текущей реализацией и будут ослаблены в будущих версиях Kotlin.

В частности, овеществляемые типовые параметры можно использовать:

- в операциях проверки и приведения типа (`is`, `!is`, `as`, `as?`);
- совместно с механизмом рефлексии, как будет обсуждаться в главе 10 (`::class`);
- для получения соответствующего `java.lang.Class` (`::class.java`);
- как типовой аргумент для вызова других функций.

Ниже перечислено то, чего *нельзя* делать:

- создавать новые экземпляры класса, указанного в типовом параметре;
- вызывать методы объекта-компаньона для класса в типовом параметре;
- использовать неовеществляемый типовой параметр в качестве типового аргумента при вызове функции с овеществляемым типовым параметром;
- объявлять овеществляемыми типовые параметры для классов, свойств и невстраиваемых функций.

У последнего ограничения есть интересное следствие: поскольку овеществляемые типовые параметры могут использоваться только со встраиваемыми функциями, их применение означает, что функция вместе со всеми лямбда-выражениями, которые она принимает, становится встраиваемой. Если лямбда-выражения не могут быть встроены из-за особен-

ностей их использования во встраиваемой функции, или если вы сами не хотите делать их встраиваемыми, можно использовать модификатор `noinline`, чтобы явно объявить их невстраиваемыми. (См. раздел 8.2.2.)

Теперь, обсудив особенности работы обобщенных типов, рассмотрим поближе самые часто используемые обобщенные типы, присутствующие в любой программе на языке Kotlin: коллекции и их подклассы. Мы будем использовать их как отправную точку в обсуждении понятий подтипов и вариантности.

9.3. Вариантность: обобщенные типы и подтипы

Термин *вариантность* (variance) описывает, как связаны между собой типы с одним базовым типом и разными типовыми аргументами: например, `List<String>` и `List<Any>`. Сначала мы обсудим, почему эта связь важна, а затем посмотрим, как она выражается в языке Kotlin. Понимание вариантности пригодится, когда вы начнете писать свои обобщенные классы и функции: это поможет вам создавать API, который не ограничивает пользователей и соответствует их ожиданиям в отношении безопасности типов.

9.3.1. Зачем нужна вариантность: передача аргумента в функцию

Представим, что у нас есть функция, принимающая аргумент `List<Any>`. Безопасно ли передать такой функции переменную типа `List<String>`? Как известно, можно передать строку в функцию, ожидающую получить аргумент `Any`, потому что класс `String` наследует `Any`. Но когда `Any` и `String` выступают в роли типовых аргументов для интерфейса `List`, ситуация не так очевидна.

Например, рассмотрим функцию, которая выводит содержимое списка.

```
fun printContents(list: List<Any>) {
    println(list.joinToString())
}

>>> printContents(listOf("abc", "bac"))
abc, bac
```

Всё выглядит хорошо. Функция интерпретирует каждый элемент как значение типа `Any`, и так как любая строка – это значение типа `Any`, то общая безопасность не нарушается.

А теперь взгляните на другую функцию, которая изменяет список (и поэтому принимает `MutableList` в параметре):

```
fun addAnswer(list: MutableList<Any>) {
    list.add(42)
}
```

Может ли произойти какая-либо неприятность, если передать этой функции список строк?

```
>>> val strings = mutableListOf("abc", "bac")
>>> addAnswer(strings)
>>> println(strings.maxBy { it.length })
ClassCastException: Integer cannot be cast to String
```

Если эта строка скомпилируется ...

... вы получите исключение во время выполнения

Мы объявили переменную `strings` типа `MutableList<String>`. Затем попробовали передать её в функцию. Если бы компилятор принял эту операцию, мы смогли бы добавить целое число в список строк, и это привело бы к исключению во время выполнения (при попытке обратиться к содержимому списка как к строкам). Поэтому данный вызов не компилируется. Этот пример показывает, что не всегда безопасно передавать `MutableList<String>` в аргументе, когда ожидается `MutableList<Any>`, – компилятор Kotlin справедливо отвергает такие попытки.

Теперь вы можете сами ответить на вопрос о безопасности передачи списка строк в функцию, ожидающую получить список объектов `Any`. Это небезопасно, если функция добавляет или заменяет элементы в списке, потому что можно нарушить совместимость типов. В других случаях это безопасно (причины этого мы подробно обсудим далее). В Kotlin такими ситуациями легко управлять, выбирая правильный интерфейс в зависимости от разновидности списка – изменяемый/неизменяемый. Если функция принимает неизменяемый список, ей можно передать список `List` с более конкретным типом элементов. Если список изменяемый, этого делать нельзя.

Далее в этом разделе мы обобщим данный вопрос, распространив его не только на `List`, но и на любой обобщенный класс. Вы также увидите, почему интерфейсы `List` и `MutableList` отличаются в отношении их типовых аргументов. Но прежде обсудим понятия *тип* и *подтип*.

9.3.2. Классы, типы и подтипы

Как обсуждалось в разделе 6.1.2, тип переменной определяет её возможные значения. Иногда мы используем термины *тип* и *класс* как эквивалентные, но на самом деле они неодинаковы. Настал момент познакомиться с их отличиями.

В простейшем случае с необобщенным классом имя класса можно непосредственно использовать как название типа. Например, записав `var x: String`, мы объявим переменную, способную хранить экземпляры класса `String`. Но обратите внимание, что то же самое имя класса можно исполь-

зовать для объявления типа, способного принимать значение `null`: `var x: String?`. Это означает, что в Kotlin каждый класс можно использовать для конструирования по меньшей мере двух типов.

Ситуация становится ещё более запутанной, когда в игру вступают обобщенные классы. Чтобы получить действительный тип, нужно подставить аргумент с конкретным типом для типового параметра в классе. `List` – это не тип (это класс), но все следующие варианты подстановки – корректные типы: `List<Int>`, `List<String?>`, `List<List<String>>` и так далее. Каждый обобщенный класс потенциально способен породить бесконечное количество типов.

Чтобы мы могли перейти к обсуждению отношений между типами, необходимо познакомиться с понятием *подтипа*. Тип B – это подтип типа A, если значение типа B можно использовать везде, где ожидается значение типа A. Например, `Int` – это подтип `Number`, но `Int` не является подтипом `String`. Это также иллюстрация того, что тип одновременно является подтип самого себя. (См. рис. 9.4.)

Смысл термина *супертип* противоположен термину *подтип*. Если A – это подтипом B, тогда B – это супертип для A.

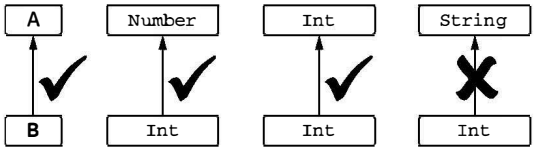


Рис. 9.4. B – подтип A, если его можно использовать везде, где ожидается A

Почему так важно знать, считается ли один тип подтипом другого? Каждый раз, когда вы присваиваете значение переменной или передаете аргумент в вызов функции, компилятор проверяет наличие отношения тип–подтип. Обратимся к примеру.

Листинг 9.10. Проверка отношения тип–подтип

```
fun test(i: Int) {
    val n: Number = i
    fun f(s: String) { /*...*/ }
    f(i)
}
```

Скомпилируется, потому что Int является подтипом Number

Не скомпилируется, потому что Int не является подтипом String

Сохранение значения в переменной допускается только тогда, когда тип значения – это подтип типа переменной. Например, тип `Int` инициализатора – параметра `i` – это подтип типа `Number` переменной. То есть такое объявление переменной `n` считается допустимым. Передача выражения в функцию допускается, только когда тип выражения – это подтип

для типа параметра функции. В примере выше тип `Int` аргумента `i` – это не подтип типа `String` параметра функции `f`, поэтому её вызов не компилируется.

В простых случаях подтип означает то же самое, что подкласс. Например, класс `Int` – это подкласс `Number` и, соответственно, тип `Int` – это подтип типа `Number`. Если класс реализует интерфейс, его тип становится подтипом этого интерфейса: `String` – подтип `CharSequence`.

Но типы, поддерживающие значение `null`, демонстрируют случаи, когда подтип – не то же самое, что подкласс. (См. рис. 9.5.)

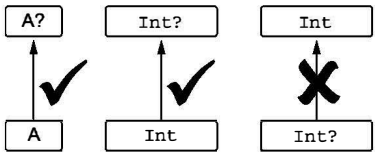


Рис. 9.5. Тип A, не поддерживающий значения null, является подтипом типа A?, поддерживающего null, но не наоборот

Тип, не поддерживающий значения `null`, – это подтип его версии, поддерживающей `null`, но они оба соответствуют одному классу. В переменной с типом, поддерживающим значение `null`, всегда можно сохранить значение типа, не поддерживающего `null`, но не наоборот (`null` – недопустимое значение для переменной с типом, не поддерживающим `null`):

```
val s: String = "abc"
val t: String? = s
```

Это присваивание допустимо, потому что String является подтипом для String?

Разница между подклассами и подтипами становится особенно важной, когда речь заходит об обобщенных типах. Вопрос о безопасности передачи переменной типа `List<String>` в функцию, ожидающую получить значение типа `List<String>`, теперь можно переформулировать в терминах подтипов: является ли `List<String>` подтипом `List<Any>`? Вы видели, почему небезопасно считать `MutableList<String>` подтипом `MutableList<Any>`. Очевидно, что обратное утверждение тоже верно: `MutableList<Any>` – это не подтип `MutableList<String>`.

Обобщенный класс – например, `MutableList` – называют *инвариантным* по типовому параметру, если для любых разных типов A и B `MutableList<A>` не является подтипом или супертипом `MutableList<B>`. В Java все классы инвариантны (хотя конкретные декларации, использующие эти классы, могут не быть инвариантными, как вы вскоре увидите).

В предыдущем разделе вы видели класс, для которого правила подтипов отличаются: `List`. Интерфейс `List` в Kotlin – это коллекция, доступная только для чтения. Если A – это подтип подтипа B, тогда `List<A>` – это подтип `List<B>`. Такие классы и интерфейсы называют *ковариантными*. Идея ковариантности подробно обсуждается в следующем разделе, где объяс-

няется, когда появляется возможность объявлять классы и интерфейсы как ковариантные.

9.3.3. Ковариантность: направление отношения тип–подтип сохраняется

Ковариантный класс – это обобщенный класс (в качестве примера будем использовать `Producer<T>`), для которого верно следующее: `Producer<A>` – это подтип `Producer<B>`, если `A` – подтип `B`. Мы называем это *сохранением направления отношения тип–подтип*. Например, `Producer<Cat>` – это подтип `Producer<Animal>`, потому что `Cat` – подтип `Animal`.

В Kotlin, чтобы объявить класс ковариантным по некоторому типовому параметру, нужно добавить ключевое слово `out` перед именем типового параметра:

```
interface Producer<out T> {
    fun produce(): T
}
```

Этот класс объявлен ковариантным по типовому параметру T

Объявление типового параметра класса ковариантным разрешает передавать значения этого класса в аргументах и возвращать их из функций, когда типовой аргумент неточно соответствует типу параметра в определении функции. Например, представьте функцию, которая заботится о питании группы животных, представленной классом `Herd`[1]. Типовой параметр класса `Herd` идентифицирует тип животных в стаде.

Листинг 9.11. Определение инвариантного класса, аналогичного коллекции

```
open class Animal {
    fun feed() { ... }
}

class Herd<T : Animal> {
    val size: Int get() = ...
    operator fun get(i: Int): T { ... }
}

fun feedAll(animals: Herd<Animal>) {
    for (i in 0 until animals.size) {
        animals[i].feed()
    }
}
```

Типовой параметр не объявлен ковариантным

Предположим, что пользователь вашего кода завёл стадо кошек и ему нужно позаботиться о них.

[1] Herd (англ.) – стадо. – *Прим. перев.*

Листинг 9.12. Использование инвариантного класса, аналогичного коллекции

```
class Cat : Animal() {                          <┐ Cat – это
    fun cleanLitter() { ... }                     | Animal
}

fun takeCareOfCats(cats: Herd<Cat>) {
    for (i in 0 until cats.size) {
        cats[i].cleanLitter()
        // feedAll(cats)                        <┐ Ошибка: передан тип Herd<Cat>, тогда
    }                                             | как ожидается тип Herd<Animal>
}
```

К сожалению, кошки останутся голодными: если попробовать передать стадо в функцию `feedAll`, компилятор сообщит о несоответствии типов. Поскольку мы не использовали модификатор вариантности в параметре типа `T` в классе `Herd`, стадо кошек не считается подклассом стада животных. Чтобы преодолеть проблему, можно выполнить явное приведение типа, но это излишне многословный, чреватый ошибками и практически всегда неправильный способ преодоления проблем совместимости типов.

Поскольку класс `Herd` имеет API, напоминающий `List`, и не позволяет своим клиентам добавлять или изменять животных в группе, его можно объявить ковариантным и изменить вызывающий код.

Листинг 9.13. Использование ковариантного класса, аналогичного коллекции

```
class Herd<out T : Animal> {                    <┐ Теперь параметр T является
    ...                                           | ковариантным
}

fun takeCareOfCats(cats: Herd<Cat>) {
    for (i in 0 until cats.size) {
        cats[i].cleanLitter()
    }
    feedAll(cats)                               <┐ Нет необходимости
}                                                 | выполнять приведение типа
```

Не каждый класс можно объявить ковариантным: это может быть небезопасно. Объявление класса ковариантным по определенному типовому параметру ограничивает возможные способы использования этого параметра в классе. Чтобы гарантировать безопасность типов, он может использоваться только в так называемых *исходящих* (*out*) позициях: то есть класс может производить значения типа `T`, но не потреблять их.

Использование типового параметра в объявлениях членов класса можно разделить на *входящие* (*in*) и *исходящие* (*out*) позиции. Рассмотрим класс, объявляющий параметр типа `T` и содержащий функцию, которая исполь-

зует T. Мы говорим, что если T используется как тип возвращаемого значения функции, то он находится в исходящей позиции. В этом случае функция *производит* значения типа T. Если T используется как тип параметра функции, он находится во входящей позиции. Такая функция потребляет значения типа T. Обе позиции показаны на рис. 9.6.

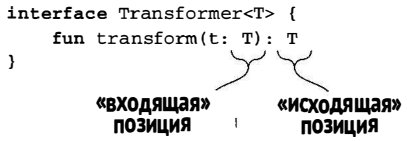


Рис. 9.6. Тип параметра функции называют входящей позицией, а тип возвращаемого значения – исходящей

Ключевое слово `out` в параметре типа класса требует, чтобы все методы, использующие T, указывали его только в исходящей позиции. Это ключевое слово ограничивает возможные варианты использования T, чтобы гарантировать безопасность соответствующего отношения с подтипом.

Для примера рассмотрим класс `Herd`. Он использует параметр типа T только в одном месте: в определении типа возвращаемого значения метода `get`.

```
class Herd<out T : Animal> {
    val size: Int get() = ...
    operator fun get(i: Int): T { ... }
}
```

Использует T как тип возвращаемого значения

Это исходящая позиция, что означает, что объявить класс ковариантным безопасно. Любой код, вызывающий метод `get` объекта типа `Herd<Animal>`, будет прекрасно работать, если метод вернет `Cat`, потому что `Cat` – это подтип `Animal`.

Повторим ещё раз, что ключевое слово `out` перед параметром типа T означает следующее:

- сохраняется отношение подтипа (`Producer<Cat>` – это подтип `Producer<Animal>`);
- T может использоваться только в исходящих позициях.

Теперь рассмотрим интерфейс `List<T>`. Значения типа `List` в языке Kotlin доступны только для чтения, потому что этот тип имеет метод `get`, возвращающий элемент типа T, но не определяет никаких методов, сохраняющих в списке значения типа T. То есть этот интерфейс также ковариантный.

```
interface List<out T> : Collection<T> {
    operator fun get(index: Int): T
    // ...
}
```

Интерфейс неизменяемого типа определяет только методы, возвращающие T (то есть T находится в «исходящей» позиции)

Обратите внимание, что параметр типа можно использовать не только как тип параметров или возвращаемого значения, но также как аргумент типа в других типах. Например, интерфейс `List` содержит метод `subList`, возвращающий `List<T>`.

```
interface List<out T> : Collection<T> {
    fun subList(fromIndex: Int, toIndex: Int): List<T>   <-- Здесь T также находится в «исходящей» позиции
    // ...
}
```

В данном случае тип `T` используется в функции `subList` в исходящей позиции. Мы не будем углубляться в детали: если вам интересен точный алгоритм определения вида позиции – исходящий или входящий, – загляните в документацию языка Kotlin.

Обратите внимание, что у вас не получится объявить `MutableList<T>` ковариантным по типовому параметру, потому что он включает методы, принимающие значения типа `T` в виде параметров и возвращающие такие значения (то есть тип `T` присутствует в обеих позициях).

```
interface MutableList<T>                          <-- MutableList нельзя объявить ковариантным по T ...
        : List<T>, MutableCollection<T> {
    override fun add(element: T): Boolean         <-- ... потому что T используется во «входящей» позиции
}
```

Компилятор требует соблюдать это ограничение и выводит сообщение об ошибке, если попробовать объявить класс ковариантным: `Type parameter T is declared as 'out' but occurs in 'in' position` (Типовой параметр T объявлен как `'out'`, но используется во `'входящей'` позиции).

Обратите внимание, что параметры конструктора не находятся ни во входящей, ни в исходящей позиции. Даже если параметр типа объявить как `out`, вы все ещё сможете использовать его в объявлениях параметров конструктора:

```
class Herd<out T: Animal>(vararg animals: T) { ... }
```

Вариантность защищает от ошибок, когда экземпляры класса используются как экземпляры более обобщенного типа: вы просто не сможете вызвать потенциально опасных методов. Конструктор – это не метод, который можно вызвать позднее (после создания экземпляра), поэтому он не представляет потенциальной опасности.

Однако если параметры конструктора объявлены с помощью ключевых слов `val` и/или `var`, то вместе с ними объявляются методы чтения и записи для изменяемых свойств. Поэтому параметр типа оказывается в исходящей позиции для неизменяемых свойств и в обеих позициях для изменяемых:

```
class Herd<T: Animal>(var leadAnimal: T, vararg animals: T) { ... }
```

В данном случае T нельзя объявить исходящим, потому что класс содержит метод записи для свойства leadAnimal – то есть использует тип T во входящей позиции.

Отметьте, что правила позиции охватывают только видимое извне (public, protected и internal) API класса. Параметры приватных методов не находятся ни в исходящей, ни во входящей позициях. Правила вариантности защищают класс от неправильного использования внешними клиентами и не применяются к внутренней реализации класса:

```
class Herd<out T: Animal>(private var leadAnimal: T, vararg animals: T) { ... }
```

Теперь мы можем безопасно объявить Herd ковариантным по T, потому что свойство leadAnimal объявлено приватным.

Вы можете спросить: что происходит с классами или интерфейсами, когда параметр типа используется только во входящей позиции? В этом случае действует обратное отношение – подробнее о нём читайте в следующем разделе.

9.3.4. Контравариантность: направление отношения тип–подтип изменяется на противоположное

Понятие *контравариантности* можно рассматривать как обратное понятию ковариантности: для контравариантного класса отношение тип–подтип действует в противоположном направлении относительно отношения между классами, использованными как типовые аргументы. Начнем с примера: интерфейса Comparator. Этот интерфейс определяет один метод, compare, который сравнивает два объекта:

```
interface Comparator<in T> {
    fun compare(e1: T, e2: T): Int { ... }
}
```

T используется во «входящей» позиции

Как видите, метод этого интерфейса только потребляет значения типа T. То есть T используется только во входящей позиции, и поэтому его имени в объявлении предшествует ключевое слово in.

Реализация интерфейса Comparator для данного типа может сравнивать значения любых его подтипов. Например, при реализации Comparator<Any> появляется возможность сравнивать значения любого конкретного типа.

```
>>> val anyComparator = Comparator<Any> {
...     e1, e2 -> e1.hashCode() - e2.hashCode()
... }
>>> val strings: List<String> = ...
>>> strings.sortedWith(anyComparator)
```

Реализацию Comparator можно использовать для сравнения любых объектов конкретного типа, таких как строки

Функция sortedWith ожидает получить Comparator<String> (реализацию, способную сравнивать строки), но ей без всякой опаски можно передать реализацию, сравнивающую более общие типы. Для сравнения объектов конкретного типа можно использовать реализацию Comparator для данного типа или для любого его супертипа. Это означает, что Comparator<Any> – это подтип Comparator<String>, тогда как Any – это супертип для String. Отношение вида тип–подтип между реализациями Comparator для двух разных типов направлено противоположно отношению между самими этими типами.

Теперь вы готовы познакомиться с полным определением контравариантности. Класс, *контравариантный* по типовому параметру, – это обобщенный класс (возьмем для примера Consumer<T>), для которого выполняется следующее: Consumer<A> – это подтип Consumer<B>, если B – это подтип A. Типовые аргументы A и B меняются местами, поэтому мы говорим, что отношение тип–подтип обратно направленное. Например, Consumer<Animal> – это подтип Consumer<Cat>.

На рис. 9.7 видны различия между отношениями вида тип–подтип для классов, ковариантных и контравариантных по типовому параметру. Как можно заметить, что для класса Producer отношение тип–подтип повторяет соответствующее отношение для типовых аргументов, в то время как для класса Consumer отношения противоположны.

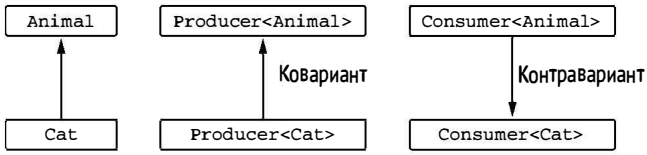


Рис. 9.7. Для ковариантного типа Producer<T> направление отношения тип–подтип сохраняется, но для контравариантного типа Consumer<T> направление отношения тип–подтип изменяется на противоположное

Ключевое слово in означает, что значения соответствующего типа *передаются в* методы данного класса (считаются входящими значениями) и потребляются этими методами. Подобно случаю ковариантности, ограничение на использование типового параметра делает возможным конкретное отношение тип–подтип. Ключевое слово in перед именем типового параметра T сообщает, что направление отношения тип–подтип меняется на противоположное и T может использоваться только во входящих позициях. В табл. 9.1 перечислены основные различия между возможными видами вариантности.

Таблица 9.1. Ковариантные, контравариантные и инвариантные классы

Ковариантные	Контравариантные	Инвариантные
`Producer<out T>`	`Consumer<in T>`	`MutableList<T>`
Направление отношения тип-подтип для классов сохраняется: `Producer<Cat>` – подтип `Producer<Animal>`.	Направление отношения тип-подтип меняется на обратное: `Consumer<Animal>` – подтип `Consumer<Cat>`.	Нет отношения тип-подтип
`T` только в исходящих позициях	`T` только во входящих позициях	`T` в любых позициях

Класс или интерфейс может быть ковариантным по одному параметру типа и контравариантным по другому. Классический пример – интерфейс `Function`. Следующее объявление демонстрирует функцию с одним параметром:

```
interface Function1<in P, out R> {
    operator fun invoke(p: P): R
}
```

Нотация `(P) -> R` – ещё одна легко читаемая форма, эквивалентная `Function1<P, R>`. Как видите, `P` (тип параметра) использован только во входящей позиции и отмечен ключевым словом `in`, тогда как `R` (тип возвращаемого значения) использован только в исходящей позиции и отмечен ключевым словом `out`. Это означает, что понятие подтипа для типов функций обратно первому типовому аргументу и совпадает со вторым. Например, функции высшего порядка, перечисляющей всех кошек в группе, можно передать лямбда-выражение, принимающее любых животных.

```
fun enumerateCats(f: (Cat) -> Number) { ... }
fun Animal.getIndex(): Int = ...

>>> enumerateCats(Animal::getIndex)
```

Это допустимый код в Kotlin. Animal – супертип для Cat, а Int – подтип Number

На рис. 9.8 изображены направления отношений тип–подтип в предыдущем примере.

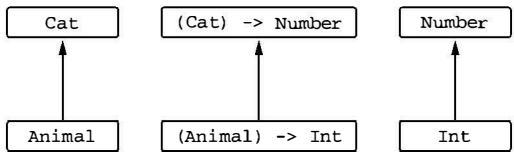


Рис. 9.8. Функция (T) -> R контравариантна по своему аргументу и ковариантна по типу возвращаемого значения

Обратите внимание, что до сих пор во всех примерах вариантность класса указывалась непосредственно в его объявлении и применялась везде, где использовался класс. Java не поддерживает этого и предлагает использовать метасимволы (wildcards) для определения вариантности использования класса. Давайте посмотрим, чем отличаются эти два подхода и как можно использовать второй подход в Kotlin.

9.3.5. Определение вариантности в месте использования: определение вариантности для вхождений типов

Поддержка модификаторов вариантности в объявлениях классов – это очень удобно, потому что указанные модификаторы применяются везде, где используется класс. Это называют *определением вариантности в месте объявления*. Знакомые с метасимволами типов в Java (`? extends` и `? super`) легко поймут, что Java обрабатывает вариантность иначе. Всякий раз, когда в Java используется тип с параметром типа, есть возможность указать, что в параметре типа можно передавать подтипы или супертипы. Это называется *определением вариантности в месте использования*.

Определение вариантности в месте объявления в Kotlin и метасимволы типов в Java

Поддержка определения вариантности в месте объявления позволяет писать более компактный код, потому что модификаторы вариантности указываются только один раз и клиентам вашего класса не приходится задумываться о них. В Java, чтобы создать API в соответствии с ожиданиями пользователей, разработчик библиотеки должен использовать метасимволы все время: `Function<? super T, ? extends R>`. Если заглянуть в исходный код стандартной библиотеки Java 8, можно увидеть, что метасимволы в ней используются везде, где используется интерфейс `Function`. Например, вот как объявлен метод `Stream.map`:

```
/* Java */
public interface Stream<T> {
    <R> Stream<R> map(Function<? super T, ? extends R> mapper);
}
```

Определение вариантности в месте объявления делает код компактнее и элегантнее.

Kotlin тоже поддерживает возможность определения вариантности в месте использования, позволяя указывать вариантность для конкретного вхождения типового параметра, даже если он не был объявлен в классе как ковариантный или контравариантный. Давайте посмотрим, как это работает.

Вы уже видели, что многие интерфейсы, такие как `MutableList`, в общем случае не считаются ни ковариантными, ни контравариантными, потому что они могут производить и потреблять значения с типами, указанными в их типовых параметрах. Но нередко переменная такого типа в конкретной функции используется только в одной из этих ролей: либо как производитель, либо как потребитель. Например, взгляните на следующую простую функцию.

Листинг 9.14. Функция копирования данных с инвариантными типовыми параметрами

```
fun <T> copyData(source: MutableList<T>,
                 destination: MutableList<T>) {
    for (item in source) {
        destination.add(item)
    }
}
```

Эта функция копирует элементы из одной коллекции в другую. Хотя обе коллекции имеют инвариантный тип, исходная коллекция (`source`) используется только для чтения, а коллекция назначения (`destination`) – только для записи. Для такой операции типы элементов коллекций могут не совпадать. Например, вполне корректно копировать коллекцию строк в коллекцию объектов `Any`.

Чтобы эта функция работала со списками разных типов, можно ввести второй параметр обобщенного типа.

Листинг 9.15. Функция копирования данных с двумя типовыми параметрами

```
fun <T: R, R> copyData(source: MutableList<T>,
                       destination: MutableList<R>) {
    for (item in source) {
        destination.add(item)
    }
}
```

Тип элементов в исходной коллекции должен быть подтипом типа элементов в коллекции назначения

```
>>> val ints = mutableListOf(1, 2, 3)
>>> val anyItems = mutableListOf<Any>()
>>> copyData(ints, anyItems)
>>> println(anyItems)
[1, 2, 3]
```

Этот вызов допустим, потому что Int является подтипом Any

Мы объявили два параметра обобщенных типов, представляющих типы элементов в исходном и целевом списках. Чтобы копирование было возможно, элементы в исходном списке должны иметь тип, являющийся под-

типом типа элементов в списке назначения, подобно тому, как тип Int является подтипом Any в листинге 9.15.

Но Kotlin предлагает более элегантный способ выразить такую зависимость. Когда функция вызывает методы, имеющие типовой параметр только во входящей (или только в исходящей) позиции, мы можем воспользоваться этим и добавить модификаторы вариантности к конкретным случаям использования типового параметра в определении функции.

Листинг 9.16. Функция копирования данных с выходной проекцией типового параметра

```
fun <T> copyData(source: MutableList<out T>,
                 destination: MutableList<T>) {
    for (item in source) {
        destination.add(item)
    }
}
```

В место использования типа можно добавить модификатор «out»: это исключит возможность использования методов с типом T в позиции «in»

Мы можем добавить модификатор вариантности в место использования типового параметра в объявлении: в тип параметра (как в листинге 9.16), тип локальной переменной, тип возвращаемого значения и так далее. Такое определение называется *проекцией типа* (type projection): оно говорит, что source – это не обычный список MutableList, а его проекция (с ограниченными возможностями). В данном случае допускается только вызов методов, возвращающих обобщенный параметр типа (или, строго говоря, использующих его только в исходящей (out) позиции). Компилятор отвергнет вызовы методов, в которых данный типовой параметр используется как аргумент (во входящей (in) позиции):

```
>>> val list: MutableList<out Number> = ...
>>> list.add(42)
Error: Out-projected type 'MutableList<out Number>' prohibits
the use of 'fun add(element: E): Boolean'
```

Не удивляйтесь, если у вас не получится вызвать какие-то методы при использовании проекции типа. Чтобы получить возможность вызывать их, нужно использовать обычные типы вместо проекций. Для этого может потребоваться объявить второй типовой параметр, зависящий от того, который первоначально был проекцией, как в листинге 9.15.

Конечно, правильнее реализовать функцию copyData с использованием типа List<T> для аргумента source, потому что она использует только методы, объявленные в List, но не в MutableList, а вариантность параметра типа List определена в его объявлении. Однако понимание этого примера всё равно полезно, если учесть, что многие классы не имеют отдельного

ковариантного интерфейса для чтения и инвариантного интерфейса для чтения/записи, таких как `List` и `MutableList`.

Бессмысленно определять проекцию, такую как `List<out T>`, для параметра типа, который уже имеет исходящую (`out`) вариантность. Она означает то же самое, что `List<T>`, потому что `List` объявлен как `class List<out T>`. Компилятор Kotlin предупредит вас, когда встретит такую избыточную проекцию.

Аналогично можно использовать модификатор `in`, чтобы обозначить, что в данном конкретном случае соответствующее значение действует как потребитель, а типовой параметр можно заменить любым его супертипом. Вот как можно переписать листинг 9.16 с использованием входящей (`in`) проекции.

Листинг 9.17. Функция копирования данных с входящей проекцией типового параметра

```
fun <T> copyData(source: MutableList<T>,
                 destination: MutableList<in T>) {
    for (item in source) {
        destination.add(item)
    }
}
```

Допускает возможность использования целевой коллекции с элементами другого типа, если он является супертипом для типа элементов в исходной коллекции

Примечание. Определение вариантности в месте использования в языке Kotlin прямо соответствует ограниченным метасимволам в Java. Объявление `MutableList<out T>` в Kotlin означает то же самое, что `MutableList<? extends T>` в Java, а входящая проекция `MutableList<in T>` соответствует `MutableList<? super T>`.

Проекции в месте использования помогают расширить диапазон допустимых типов. А теперь исследуем экстремальный случай: когда допустимыми становятся типы с любыми типовыми аргументами.

9.3.6. Проекция со звездочкой: использование * вместо типового аргумента

Обсуждая операции проверки и приведения типов в начале этой главы, мы упомянули о поддержке особого синтаксиса *проекций со звездочкой*, который можно использовать, чтобы сообщить об *отсутствии информации об обобщенном аргументе*. Например, список элементов неизвестного типа выражается с применением синтаксиса `List<*>`. Давайте подробно исследуем семантику проекций со звездочкой.

Прежде всего отметим, что `MutableList<*>` – это не то же самое, что `MutableList<Any?>` (важно, что `MutableList<T>` инвариантен по `T`). Тип `MutableList<Any?>` определяет список, содержащий элементы любого

типа. С другой стороны, тип `MutableList<*>` определяет список, содержащий элементы конкретного типа, но неизвестно, какого именно. Список создавался как список элементов конкретного типа, такого как `String` (нельзя создать новый `ArrayList<*>`), и код, создавший его, ожидает, что он будет содержать только элементы этого типа. Поскольку конкретный тип нам неизвестен, мы не можем добавлять новые элементы в этот список, потому что рискуем обмануть ожидания вызывающего кода. Зато мы можем извлекать элементы из списка, потому что точно знаем, что все значения, хранящиеся в нём, соответствуют типу `Any?` – супертипу всех типов в Kotlin:

```
>>> val list: MutableList<Any?> = mutableListOf('a', 1, "qwe")
>>> val chars = mutableListOf('a', 'b', 'c')
>>> val unknownElements: MutableList<*> =
...     if (Random().nextBoolean()) list else chars
>>> unknownElements.add(42)
Error: Out-projected type 'MutableList<*>' prohibits
the use of 'fun add(element: E): Boolean'
>>> println(unknownElements.first())
a
```

MutableList<*> – не то же самое, что MutableList<Any?>

Компилятор отвергнет попытку вызвать этот метод

Извлечение элементов не представляет опасности: first() вернет элемент типа Any?

Почему компилятор интерпретирует `MutableList<*>` как исходящую проекцию типа? В данном контексте `MutableList<*>` проецируется в (действует как) тип `MutableList<out Any?>`: если о типе элементов ничего не известно, безопасно только извлекать элементы типа `Any?`, а добавлять новые элементы в список небезопасно. Если провести аналогию с метасимволами типов в Java, `MyType<*>` в Kotlin соответствует `MyType<?>` в Java.

> **Примечание.** Для контравариантных типовых параметров, таких как `Consumer<in T>`, проекция со звездочкой эквивалентна `<in Nothing>`. Компилятор не позволит вызывать методы таких проекций со звездочками, имеющие `T` в сигнатуре. Если типовой параметр объявлен как контравариантный, он может действовать только как потребитель, и, как обсуждалось выше, мы не знаем точно, что именно он может потреблять. Поэтому мы ничего не можем предложить ему для потребления. Если вас заинтересовала эта тема, загляните в документацию Kotlin (`http://mng.bz/3Ed7`).

Синтаксис проекций со звездочкой используется, когда информация о типовом аргументе не имеет никакого значения, то есть когда ваш код не использует методов, ссылающихся на типовой параметр в сигнатуре, или только читает данные без учета их типа. Например, вот как можно реализовать функцию `printFirst`, принимающую параметр с типом `List<*>`:

```
fun printFirst(list: List<*>) {
    if (list.isNotEmpty()) {
```

В аргументе можно передать любой список

isNotEmpty() не использует параметра обобщенного типа

```
        println(list.first())          <┐ first() вернет Any?, но в данном
    }                                    │ случае этого достаточно
}

>>> printFirst(listOf("Svetlana", "Dmitry"))
Svetlana
```

Как и в случае определения вариантности в месте использования, можно использовать альтернативное решение – ввести дополнительный параметр обобщенного типа:

```
fun <T> printFirst(list: List<T>) {      <┐ И снова допускается передать в
    if (list.isNotEmpty()) {               │ аргументе любой список
        println(list.first())          <┐ Но теперь first() вернет
    }                                    │ значение типа T
}
```

Синтаксис проекций со звездочкой лаконичнее, но его можно использовать только тогда, когда точное значение параметра обобщенного типа не играет никакой роли: когда вы используете только методы, извлекающие значения, не заботясь о типах этих значений.

Теперь рассмотрим другой пример использования проекции типа со звездочкой и часто встречающиеся ошибки. Допустим, что нам нужно проверить ввод пользователя, и мы объявили интерфейс `FieldValidator`. Он содержит параметр типа только во входящей позиции, поэтому его можно объявить контравариантным. Это правильный способ объявления валидатора, способного проверить любой элемент, когда ожидается валидатор строк (контравариантное объявление позволяет сделать это). Мы также объявили два валидатора, проверяющих значения типа `String` и `Int`.

Листинг 9.18. Интерфейсы для создания валидаторов ввода

```
interface FieldValidator<in T> {                    <┐ Интерфейс объявлен
    fun validate(input: T): Boolean  <┐               │ контравариантным по T
}                                     │ T используется только в
                                      │ позиции «in» (этот метод
                                      │ потребляет значение типа T)
object DefaultStringValidator : FieldValidator<String> {
    override fun validate(input: String) = input.isNotEmpty()
}

object DefaultIntValidator : FieldValidator<Int> {
    override fun validate(input: Int) = input >= 0
}
```

Теперь представьте, что нам нужно сохранить все валидаторы в одном контейнере и выбрать нужный в соответствии с типом введенного значения. У многих читателей сразу возникнет соблазн использовать в качестве

хранилища словарь. Поскольку нужно хранить валидаторы любых типов, можно объявить словарь с ключами типа `KClass` (представляющего класс в Kotlin – мы обсудим этот тип в главе 10) и значениями типа `FieldValidator<*>` (который может ссылаться на валидатор любого типа):

```
>>> val validators = mutableMapOf<KClass<*>, FieldValidator<*>>()
>>> validators[String::class] = DefaultStringValidator
>>> validators[Int::class] = DefaultIntValidator
```

Но в этом случае возникают сложности с использованием валидаторов. Мы не можем проверить строку, указав тип валидатора `FieldValidator<*>`. Это небезопасно, потому что компилятор ничего не знает о типе валидатора:

```
>>> validators[String::class]!!.validate("")
Error: Out-projected type 'FieldValidator<*>' prohibits
the use of 'fun validate(input: T): Boolean'
```

Значение, хранящееся в словаре, имеет тип FieldValidator<*>

Мы уже видели эту ошибку выше, когда пытались добавить элемент в `MutableList<*>`. В данном случае эта ошибка означает, что значение конкретного типа небезопасно передавать валидатору для неизвестного типа. Одно из решений этой проблемы – явно привести валидатор к требуемому типу. Это небезопасно, так поступать не рекомендуется. Однако мы используем этот трюк как быстрое решение, помогающее скомпилировать код, чтобы потом приступить к его реорганизации.

Листинг 9.19. Извлечение валидатора с использованием явного приведения типа

```
>>> val stringValidator = validators[String::class]
                         as FieldValidator<String>
>>> println(stringValidator.validate(""))
false
```

Warning: unchecked cast (Внимание: неконтролируемое приведение типа)

Компилятор выведет предупреждение о неконтролируемом приведении типа. Однако обратите внимание, что этот код будет терпеть неудачу только при попытке выполнить проверку, но не во время приведения типа, потому что во время выполнения вся информация об обобщенных типах стирается.

Листинг 9.20. Извлечение неправильного валидатора

```
>>> val stringValidator = validators[Int::class]
                         as FieldValidator<String>
>>> stringValidator.validate("")
java.lang.ClassCastException:
  java.lang.String cannot be cast to java.lang.Number
  at DefaultIntValidator.validate
```

Извлекается неверный валидатор (возможно, по ошибке), но этот код компилируется

Это только предупреждение

Истинная ошибка не проявится, пока вы не попробуете использовать валидатор

Этот код и код в листинге 9.19 схожи в том смысле, что в обоих случаях компилятор только выводит предупреждение. Вся ответственность за правильное приведение типов ложится на ваши плечи.

Как видите, это решение небезопасно и чревато ошибками. Поэтому давайте поищем другое решение задачи хранения валидаторов разных типов в одном месте.

Решение в листинге 9.21 использует тот же словарь `validators`, но все операции доступа к нему инкапсулированы в два обобщенных метода, на которые возлагается ответственность за регистрацию и выбор правильных валидаторов. Этот код тоже заставляет компилятор вывести предупреждение (то же самое) о неконтролируемом приведении типа, но на этот раз объект `Validators` контролирует доступ к словарю и гарантирует, что никто не сможет изменить словарь по ошибке.

Листинг 9.21. Инкапсуляция доступа к коллекции валидаторов

```
object Validators {
    private val validators =                                    <-- Используется тот же словарь, что прежде, но теперь доступ извне к нему закрыт
            mutableMapOf<KClass<*>, FieldValidator<*>>()

    fun <T: Any> registerValidator(
            kClass: KClass<T>, fieldValidator: FieldValidator<T>) {
        validators[kClass] = fieldValidator                     <-- Добавляет в словарь только правильную пару ключ/значение, когда валидатор соответствует классу
    }

    @Suppress("UNCHECKED_CAST")                                 <-- Подавляет вывод предупреждения о неконтролируемом приведении к типу FieldValidator<T>
    operator fun <T: Any> get(kClass: KClass<T>): FieldValidator<T> =
        validators[kClass] as? FieldValidator<T>
            ?: throw IllegalArgumentException(
            "No validator for ${kClass.simpleName}")
}
```

```
>>> Validators.registerValidator(String::class, DefaultStringValidator)
>>> Validators.registerValidator(Int::class, DefaultIntValidator)

>>> println(Validators[String::class].validate("Kotlin"))
true
>>> println(Validators[Int::class].validate(42))
true
```

Теперь у нас есть безопасный API. Вся небезопасная логика скрыта в теле класса, и мы гарантируем невозможность её ошибочного использования путем локализации. Компилятор отвергнет любую попытку использовать неправильный валидатор, потому что объект `Validators` всегда дает правильную реализацию валидатора:

```
>>> println(Validators[String::class].validate(42))
Error: The integer literal does not conform to the expected type String
```

Теперь метод «get» вернет экземпляр типа FieldValidator<String>

Этот шаблон легко можно распространить на хранилище любых обобщенных классов. Локализация небезопасного кода в отдельном месте предотвращает ошибки и делает использование контейнера безопаснее. Обратите внимание, что шаблон, описанный здесь, не специфичен для Kotlin: тот же подход можно использовать в Java.

Обобщенные типы и вариантность в Java справедливо считаются наиболее сложным аспектом языка. В Kotlin мы постарались сделать его максимально простым и понятным, сохранив совместимость с Java.

9.4. Резюме

- Поддержка обобщенных типов в Kotlin очень похожа на аналогичную поддержку в Java – она позволяет объявлять обобщенные функции или классы тем же способом.
- Как и в Java, аргументы обобщенных типов существуют только на этапе компиляции.
- Вы не можете использовать типы с аргументами вместе с оператором `is`, потому что типовые аргументы стираются во время выполнения.
- Типовые параметры встраиваемых (`inline`) функций можно объявить овеществляемыми, что позволит использовать их во время выполнения для проверок и получения экземпляров `java.lang.Class`.
- Вариантность позволяет определить, является ли один из двух обобщенных типов с одним и тем же базовым классом и разными типовыми аргументами, подтипом или супертипом другого, если один из типовых аргументов является подтипом другого. Класс можно объявить ковариантным по типовому параметру, если параметр используется только в исходящих позициях.
- Противоположное верно для контравариантных случаев: класс можно объявить контравариантным по типовому параметру, если он используется только во входящих позициях.
- Неизменяемый интерфейс `List` в Kotlin объявлен ковариантным, а это означает, что `List<String>` – это подтип `List<Any>`.
- Интерфейс функции объявлен контравариантным по первому типовому параметру и контравариантным по второму, что делает `(Animal)->Int` подтипом для `(Cat)->Number`.

- Kotlin позволяет объявить вариантность и для обобщенного класса в целом (вариантность в месте объявления, declaration-site variance), и в месте конкретного использования обобщенного типа (use-site variance).
- Синтаксис вариантности со звездочкой можно использовать, когда типовой аргумент неизвестен или неважен.

Глава 10

Аннотации и механизм рефлексии

В этой главе:

- применение и определение аннотаций;
- использование механизма рефлексии для интроспекции классов во время выполнения;
- пример проекта на языке Kotlin.

К настоящему моменту мы познакомились со многими приёмами работы с классами и функциями, но все они требовали явно указывать имена классов и функций в исходном коде. Чтобы вызвать функцию, вы должны знать класс, в котором она определена, а также имена и типы её параметров. *Аннотации* и *механизм рефлексии* дают возможность преодолеть это ограничение и писать код, способный работать с неизвестными заранее произвольными классами. С помощью аннотаций можно присвоить таким классам особую семантику, а механизм рефлексии позволит исследовать структуру классов во время выполнения.

Пользоваться аннотациями просто, а вот писать собственные аннотации и особенно код, обрабатывающий их, – не самая простая задача. Синтаксис использования аннотаций в Kotlin точно такой же, как в Java, но синтаксис объявления собственных аннотаций несколько отличается. Механизм рефлексии очень похож на Java по общей структуре прикладного интерфейса, но отличается в деталях.

Для демонстрации использования аннотаций и инструментов рефлексии мы покажем реализацию настоящего проекта: библиотеки сериализации/десериализации в формат JSON с названием JKid. Библиотека использует механизм рефлексии для доступа к свойствам произвольных объектов во время выполнения, а также для создания объектов на основе

данных, полученных в файлах JSON. Аннотации помогут вам настраивать, как именно библиотека сериализует и десериализует конкретные классы и свойства.

10.1. Объявление и применение аннотаций

Большинство современных фреймворков на Java широко использует аннотации, поэтому вы наверняка сталкивались с ними, работая над Java-приложениями. Основная идея в Kotlin та же самая. Аннотация позволяет связать дополнительные метаданные с объявлением. Затем доступ к метаданным можно получить с использованием инструментов, работающих с исходным кодом, с файлами скомпилированных классов или во время выполнения – в зависимости от того, как настроены аннотации.

10.1.1. Применение аннотаций

Аннотации в Kotlin используются так же, как в Java. Чтобы применить аннотацию, нужно добавить её имя с приставкой @ перед аннотируемым объявлением. Аннотировать можно разные элементы кода, такие как функции и классы.

Например, при использовании фреймворка JUnit (http://junit.org/junit4/) можно отметить тестовый метод аннотацией `@Test`:

```
import org.junit.*

class MyTest {
    @Test fun testTrue() {
        Assert.assertTrue(true)
    }
}
```

Аннотация @Test сообщает фреймворку JUnit, что этот метод должен вызываться как тестовый

Рассмотрим более интересный пример – аннотацию `@Deprecated`. В Kotlin она имеет то же значение, что и в Java, но Kotlin добавляет в неё параметр `replaceWith`, в котором можно указать шаблон замены для поддержки постепенного перехода на новую версию API. Следующий пример показывает, как передать аргументы в аннотацию (сообщение об использовании устаревшего API и шаблон замены):

```
@Deprecated("Use removeAt(index) instead.", ReplaceWith("removeAt(index)"))
fun remove(index: Int) { ... }
```

Аргументы передаются в круглых скобках, в точности как при вызове обычной функции. С таким объявлением, если кто-либо использует функцию `remove`, IntelliJ IDEA не только покажет, какая функция должна использоваться взамен (в данном случае `removeAt`), но также предложит возможность выполнить замену автоматически.

Аннотации могут иметь параметры только определенных типов, в том числе простые типы, строки, перечисления, ссылки на классы, классы других аннотаций и их массивы. Синтаксис определения аргументов аннотаций немного отличается от используемого в Java:

- *Чтобы передать в аргументе класс*, нужно добавить `::class` после имени класса: `@MyAnnotation(MyClass::class)`.
- *Чтобы передать в аргументе другую аннотацию*, перед её именем не нужно добавлять символа `@`. Например, `ReplaceWith` – это аннотация, но если она используется как аргумент аннотации `Deprecated`, перед её именем нужен символ `@`.
- *Чтобы передать в аргументе массив*, используйте функцию `arrayOf`:

  ```
  @RequestMapping(path = arrayOf("/foo", "/bar")).
  ```

  Если класс аннотации объявлен на языке Java, параметр `value` автоматически преобразуется в массив, если необходимо, что позволяет передавать аргументы без использования функции `arrayOf`.

Аргументы аннотаций должны быть известны на этапе компиляции – то есть мы не можем ссылаться в аргументах на произвольные свойства. Чтобы использовать свойство как аргумент аннотации, его необходимо отметить модификатором `const`, который сообщает компилятору, что свойство является *константой времени компиляции*. Вот пример аннотации `@Test` из фреймворка JUnit, которая определяет предельное время ожидания завершения теста в миллисекундах в параметре `timeout`:

```
const val TEST_TIMEOUT = 100L

@Test(timeout = TEST_TIMEOUT) fun testMethod() { ... }
```

Как обсуждалось в разделе 3.3.1, свойства с модификатором `const` должны объявляться на верхнем уровне файла или объекта и должны инициализироваться значениями простых типов или строками. Если попытаться использовать в аргументе аннотации обычное свойство, компилятор сгенерирует ошибку «Only 'const val' can be used in constant expressions» (В выражениях-константах можно использовать только 'const val').

10.1.2. Целевые элементы аннотаций

Во многих случаях одному объявлению в исходном коде на Kotlin соответствует несколько объявлений на Java, каждое из которых может быть целью аннотации. Например, свойство в Kotlin соответствует полю в Java, методу чтения и, возможно, методу записи и его параметру. Свойству, объявленному в основном конструкторе, соответствует ещё один дополнительный элемент: параметр конструктора. Соответственно, может возникнуть потребность определить, какой именно элемент аннотируется.

Аннотируемый элемент можно указать с помощью *объявления цели*. Объявление цели помещается между знаком @ и именем аннотации и отделяется от имени двоеточием. Слово get на рис. 10.1 указывает, что аннотация @Rule применяется к методу чтения (getter) свойства.

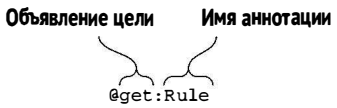


Рис. 10.1. Синтаксис объявления цели аннотации

Рассмотрим пример использования этой аннотации. В JUnit можно указать правило для выполнения перед каждым тестовым методом. Например, стандартное правило TemporaryFolder используется для создания файлов и папок, которые будут автоматически удалены после выполнения метода.

Чтобы указать правило, в Java нужно объявить общедоступное (public) поле или метод с аннотацией @Rule. Но если в тестовом классе на Kotlin просто добавить аннотацию @Rule перед свойством folder, фреймворк JUnit возбудит исключение: «The @Rule 'folder' must be public» (Правило 'folder' должно быть общедоступным). Это происходит потому, что @Rule применяется к полю, которое по умолчанию приватное. Чтобы применить аннотацию к методу чтения, нужно явно указать цель аннотации @get:Rule, как показано ниже:

```
class HasTempFolder {
    @get:Rule                                  <-- Аннотируется метод чтения, а не свойство
    val folder = TemporaryFolder()

    @Test
    fun testUsingTempFolder() {
        val createdFile = folder.newFile("myfile.txt")
        val createdFolder = folder.newFolder("subfolder")
        // ...
    }
}
```

При применении к свойству аннотации, объявленные в Java, по умолчанию применяются к соответствующему полю. Kotlin, в отличие от Java, позволяет также объявлять аннотации, которые могут применяться непосредственно к свойствам.

Вот полный перечень поддерживаемых объявлений целей:

- property – Java-аннотации не могут применяться с этим объявлением цели;
- field – поле, сгенерированное для свойства;
- get – метод чтения свойства;
- set – метод записи в свойство;
- receiver – параметр-получатель функции-расширения или свойства-расширения;
- param – параметр конструктора;

- `setparam` – параметр метода записи в свойство;
- `delegate` – поле, хранящее экземпляр делегата для делегированного свойства;
- `file` – класс, содержащий функции и свойства верхнего уровня, объявленные в файле.

Все аннотации с целью `file` должны находиться на верхнем уровне файла, перед директивой `package`. Например, к файлам часто применяется аннотация `@JvmName`, которая изменяет имя соответствующего класса. В разделе 3.2.3 был показан пример её использования: `@file:JvmName("StringFunctions")`.

> **Управление Java API с помощью аннотаций**
>
> Kotlin поддерживает несколько аннотаций для управления компиляцией объявлений на Kotlin в байт-код Java и их представлением для вызывающего кода на Java. Некоторые из них замещают соответствующие ключевые слова языка Java: например, аннотации `@Volatile` и `@Strictfp` служат прямой заменой `volatile` и `strictfp` – ключевых слов в Java. Другие помогают уточнить, как будет выглядеть объявление на Kotlin для кода на Java:
>
> - `@JvmName` изменяет имя Java-метода или поля, сгенерированного на основе Kotlin-объявления;
> - `@JvmStatic` может применяться к методам в объявлениях объектов или объектов-компаньонов, чтобы со стороны Java они выглядели как статические методы;
> - `@JvmOverloads` (аннотация, которая упоминалась в разделе 3.2.2) требует от компилятора Kotlin сгенерировать перегруженные версии функции с параметрами, имеющими значения по умолчанию;
> - `@JvmField` позволяет открыть доступ к свойству как к общедоступному Java-полю, без методов чтения/записи.
>
> Более подробные сведения об использовании этих аннотаций можно найти в документирующих комментариях, в их исходном коде, а также в электронной документации – в разделе, посвященном совместимости с Java.

Обратите внимание, что, в отличие от Java, Kotlin позволяет применять аннотации не только к объявлениям классов функций и типов, но также к произвольным выражениям. Типичным примером может служить аннотация `@Suppress`, которую можно использовать для подавления конкретных предупреждений компилятора в контексте выражения. Вот пример объявления локальной переменной с подавлением предупреждения о неконтролируемом приведении типа:

```
fun test(list: List<*>) {
    @Suppress("UNCHECKED_CAST")
```

```
    val strings = list as List<String>
    // ...
}
```

Обратите внимание, что IntelliJ IDEA дает возможность быстро вставлять эту аннотацию: нужно только нажать комбинацию клавиш **Alt-Enter** на предупреждении компилятора и выбрать **Suppress** в контекстном меню.

10.1.3. Использование аннотаций для настройки сериализации JSON

Аннотации часто применяются для настройки сериализации объектов. *Сериализация* – это процесс преобразования объекта в двоичное или текстовое представление, которое затем может быть сохранено или передано по сети. Обратный процесс, *десериализация*, преобразует такое представление обратно в объект. Часто для сериализации используется формат JSON. Существует большое множество библиотек, реализующих сериализацию Java-объектов в формат JSON, включая Jackson (https://github.com/FasterXML/jackson) и GSON (https://github.com/google/gson). Как и любые другие Java-библиотеки, они полностью совместимы с Kotlin.

На протяжении этой главы мы будем обсуждать реализацию библиотеки, предназначенную для этой цели, на чистом Kotlin – JKid. Она достаточно маленькая, чтобы вы смогли без труда читать её исходный код, и мы настоятельно рекомендуем делать это в процессе чтения этой главы.

> **Исходный код библиотеки JKid и упражнения**
>
> Полную реализацию библиотеки можно найти в пакете с исходным кодом примеров для книги, доступном по адресу: https://manning.com/books/kotlin-in-action, – а также в репозитории: http://github.com/yole/jkid. Для изучения реализации библиотеки и примеров откройте файл *ch10/jkid/build.gradle* как Gradle-проект в своей IDE. Примеры можно найти в папке *src/test/kotlin/examples*. Наша библиотека не такая богатая и гибкая, как GSON или Jackson, но её вполне достаточно для практического использования, и вы можете задействовать её в своих проектах.
>
> Проект JKid включает несколько упражнений, которые вы можете выполнить после прочтения этой главы, чтобы проверить, насколько хорошо вы понимаете основные идеи. Описание упражнений вы найдете в файле проекта *README.md* или на странице проекта на сайте GitHub.

Начнем с простейшего примера, проверяющего работу библиотеки: сериализацию и десериализацию экземпляра класса `Person`. Попробуем передать экземпляр в функцию `serialize`, которая должна вернуть строку с JSON-представлением:

```
data class Person(val name: String, val age: Int)

>>> val person = Person("Alice", 29)
>>> println(serialize(person))
{"age": 29, "name": "Alice"}
```

JSON-представление объекта включает пары ключ/значение: пары имен свойств и их значений для конкретного экземпляра, такие как `"age": 29`.

Чтобы воссоздать объект из JSON-представления, нужно вызвать функцию `deserialize`:

```
>>> val json = """{"name": "Alice", "age": 29}"""
>>> println(deserialize<Person>(json))
Person(name=Alice, age=29)
```

Формат JSON не хранит информацию о типах объектов. Поэтому, создавая экземпляр из данных в формате JSON, нужно явно указать аргумент типа. В данном случае мы передали класс `Person`.

Рисунок 10.2 иллюстрирует эквивалентность между объектом и его JSON-представлением. Обратите внимание, что сериализованная версия объекта может содержать не только свойства простых типов или строки, как показано на рис. 10.2, но также коллекции и экземпляры других объектов-значений и классов.

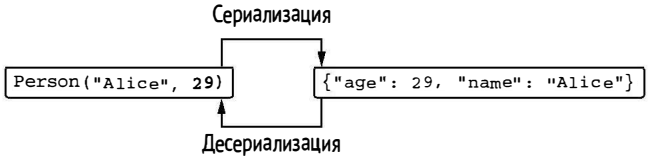


Рис. 10.2. Сериализация и десериализация экземпляра Person

Для настройки сериализации и десериализации объектов можно использовать аннотации. Выполняя сериализацию объекта в формат JSON, библиотека по умолчанию пытается сериализовать все свойства и использовать в качестве ключей их имена. Аннотации позволяют изменить это поведение. В этом разделе мы обсудим две такие аннотации, `@JsonExclude` и `@JsonName`, а далее в этой главе познакомимся с их реализациями:

- аннотация `@JsonExclude` отмечает свойство, которое должно быть исключено из процесса сериализации/десериализации;
- аннотация `@JsonName` позволяет указать строковый ключ в паре ключ/значение, который должен представлять свойство вместо его имени.

Рассмотрим их на примере:

```
data class Person(
    @JsonName("alias") val firstName: String,
    @JsonExclude val age: Int? = null
)
```

Мы аннотировали свойство `firstName`, чтобы изменить ключ, используемый для его представления в формате JSON. Мы также аннотировали свойство `age`, чтобы исключить его из процесса сериализации/десериализации. Обратите внимание, что при этом мы должны добавить значение по умолчанию для свойства `age` – иначе не получится создать новый экземпляр `Person` в ходе десериализации. На рис. 10.3 показано, как изменилось представление экземпляра класса `Person`.

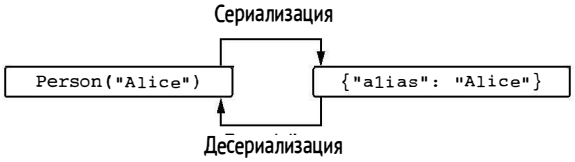


Рис. 10.3. Сериализация и десериализация экземпляра Person после добавления аннотаций

Итак, мы познакомились с самыми основными возможностями, реализованными в библиотеке JKid: `serialize()`, `deserialize()`, `@JsonName` и `@JsonExclude`. Теперь приступим к изучению их реализации. Начнем с объявления аннотаций.

10.1.4. Объявление аннотаций

В этом разделе вы узнаете, как объявлять аннотации, на примере аннотаций из библиотеки JKid. Аннотация `@JsonExclude` – самая простая, потому что не имеет никаких параметров:

```
annotation class JsonExclude
```

Синтаксис выглядит как объявление обычного класса с дополнительным модификатором `annotation` перед ключевым словом `class`. Так как классы аннотаций используются только для определения структур метаданных, связанных с объявлениями и выражениями, они не могут содержать программного кода. Соответственно, компилятор не допускает возможности определить тело класса-аннотации.

Параметры для аннотаций с параметрами объявляются в основном конструкторе:

```
annotation class JsonName(val name: String)
```

Для этого используется самый обычный синтаксис объявления основного конструктора. Ключевое слово `val` обязательно для всех параметров в классе-аннотации.

Для сравнения ниже показано объявление той же аннотации в Java:

```
/* Java */
public @interface JsonName {
```

```
    String value();
}
```

Обратите внимание, что Java-аннотации имеют метод `value`, а аннотации в Kotlin – свойство `name`. У метода `value` особое значение в Java: применяя аннотацию, вы должны явно указывать имена всех атрибутов, кроме `value`. В Kotlin, напротив, применение аннотации – это обычный вызов конструктора. Можно использовать синтаксис именованных аргументов, чтобы явно указать их значения, или опустить их: например, аннотация `@JsonName(name = "first_name")` означает то же самое, что `@JsonName("first_name")`, потому что `name` – это первый параметр конструктора `JsonName`. Однако, чтобы применить Java-аннотацию в Kotlin, придется использовать синтаксис именованных аргументов и явно перечислить все аргументы, кроме `value`, который Kotlin также распознает как специальный.

Далее обсудим, как управлять использованием аннотаций и как применять аннотации к другим аннотациям.

10.1.5. Метааннотации: управление обработкой аннотаций

По аналогии с Java классы-аннотаций в Kotlin тоже могут аннотироваться. Аннотации, применяемые к классам-аннотациям, называют *метааннотациями*. Стандартная библиотека содержит несколько таких аннотаций, управляющих обработкой аннотаций компилятором. Также есть примеры использования метааннотаций в других фреймворках: например, многие библиотеки, реализующие механизм внедрения зависимостей, используют метааннотации, чтобы отметить аннотации, идентифицирующие разные внедряемые объекты одного типа.

Из метааннотаций, объявленных в стандартной библиотеке, наиболее широко используется `@Target`. Библиотека JKid использует их в объявлениях `JsonExclude` и `JsonName`, чтобы ограничить круг допустимых целей. Вот как она применяется:

```
@Target(AnnotationTarget.PROPERTY)
annotation class JsonExclude
```

Метааннотация `@Target` определяет типы элементов, к которым может применяться объявляемая следом аннотация. Без неё аннотацию можно будет применять к любым объявлениям – это не подходит для библиотеки JKid, потому что она обрабатывает только аннотации свойств.

Список значений перечисления `AnnotationTarget` включает все возможные цели, в том числе: классы, файлы, функции, свойства, методы доступа к свойствам, типы, все выражения и так далее. Если нужно, можно объявить несколько целей: `@Target(AnnotationTarget.CLASS, AnnotationTarget.METHOD)`.

Чтобы объявить свою метааннотацию, необходимо использовать цель ANNOTATION_CLASS:

```
@Target(AnnotationTarget.ANNOTATION_CLASS)
annotation class BindingAnnotation

@BindingAnnotation
annotation class MyBinding
```

Имейте в виду, что аннотацию с целью PROPERTY нельзя использовать в Java-коде. Но к ней можно добавить вторую цель AnnotationTarget.FIELD – в этом случае аннотация будет применяться к свойствам в Kotlin и к полям в Java.

> **Аннотация @Retention**
>
> Программируя на Java, вы наверняка сталкивались с ещё одной важной метааннотацией, @Retention. С её помощью можно определить, должна ли объявляемая аннотация сохраняться в файле *.class* и будет ли она доступна во время выполнения через механизм рефлексии. По умолчанию Java сохраняет аннотации в файлах *.class*, но они остаются недоступными во время выполнения. Однако в большинстве случаев доступ к аннотациям во время выполнения был бы весьма желателен, поэтому в Kotlin используются иные умолчания: аннотации получают признак хранения RUNTIME. Поэтому аннотации в JKid не определяют этого признака явно.

10.1.6. Классы как параметры аннотаций

Теперь вы знаете, как определять аннотации, хранящие статические данные в виде аргументов. Но иногда требуется нечто иное: возможность ссылаться на *класс* как объявление метаданных. Этого можно добиться, если объявить класс аннотации со ссылкой на класс в параметре. Подходящий пример – аннотация @DeserializeInterface в библиотеке JKid, которая позволяет управлять десериализацией свойств с типом интерфейса. Как известно, нельзя создать экземпляр интерфейса непосредственно, поэтому нужно явно указать класс, который будет использоваться для создания экземпляра в процессе десериализации. Вот как используется эта аннотация:

```
interface Company {
    val name: String
}

data class CompanyImpl(override val name: String) : Company

data class Person(
```

```
    val name: String,
    @DeserializeInterface(CompanyImpl::class) val company: Company
)
```

Прочитав вложенный объект company для экземпляра Person, библиотека JKid создает и десериализует экземпляр CompanyImpl, а затем сохраняет его в свойстве company. Для этого в примере выше мы использовали аннотацию @DeserializeInterface с аргументом CompanyImpl::class. Чтобы сослаться на класс, нужно указать его имя и ключевое слово ::class вслед за ним.

Теперь посмотрим, как объявлена эта аннотация. Она имеет единственный аргумент – ссылку на класс – это видно из примера @DeserializeInterface(CompanyImpl::class):

```
annotation class DeserializeInterface(val targetClass: KClass<out Any>)
```

Тип KClass – это Kotlin-версия типа java.lang.Class в Java. Он используется для хранения ссылок на классы Kotlin (с возможностями этого типа мы познакомимся в разделе «Рефлексия» далее в этой главе).

Типовой параметр в KClass определяет, на какие классы Kotlin может ссылаться данная ссылка. Например, CompanyImpl::class имеет тип KClass<CompanyImpl>, который является подтипом типового параметра в аннотации (см. рис. 10.4).

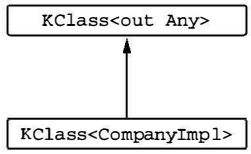


Рис. 10.4. Тип аргумента аннотации CompanyImpl::class (KClass<CompanyImpl>) – это подтип типа параметра аннотации (KClass<out Any>)

Если указать типовой параметр KClass<Any> без модификатора out, попытка передать аргумент CompanyImpl::class будет отвергнута, и единственным допустимым аргументом в этом случае будет Any::class. Ключевое слово out указывает, что можно ссылаться не только на класс Any, но и на классы, наследующие Any. В следующем разделе вы увидите ещё одну аннотацию, которая принимает параметр с обобщенным классом.

10.1.7. Обобщенные классы в параметрах аннотаций

По умолчанию библиотека JKid сериализует свойства всех типов, кроме примитивных, строковых и коллекций, как вложенные объекты. Но это поведение можно изменить и предоставить свою логику для сериализации некоторых значений.

Аннотация `@CustomSerializer` принимает в качестве аргумента ссылку на нестандартный класс-сериализатор. Этот класс должен реализовывать интерфейс `ValueSerializer`:

```
interface ValueSerializer<T> {
    fun toJsonValue(value: T): Any?
    fun fromJsonValue(jsonValue: Any?): T
}
```

Допустим, нам нужно организовать сериализацию дат, и мы написали свой класс `DateSerializer`, реализующий интерфейс `ValueSerializer<Date>`. (Этот класс включен в исходный код JKid как пример: http://mng.bz/73a7.) Вот как можно подключить этот сериализатор к классу `Person`:

```
data class Person(
    val name: String,
    @CustomSerializer(DateSerializer::class) val birthDate: Date
)
```

Посмотрим, как объявлена аннотация `@CustomSerializer`. Интерфейс `ValueSerializer` – обобщенный и имеет типовой параметр, поэтому мы должны указать типовой аргумент, чтобы сослаться на конкретный тип. Поскольку типы свойств, к которым будет применяться эта аннотация, заранее неизвестны, то как аргумент здесь используется проекция со звездочкой (этот приём обсуждался в разделе 9.3.6):

```
annotation class CustomSerializer(
    val serializerClass: KClass<out ValueSerializer<*>>
)
```

На рис. 10.5 виден тип параметра `serializerClass` и описаны разные его части. Мы должны гарантировать, что аннотация сможет ссылаться только на классы, реализующие интерфейс `ValueSerializer`. Например, объявление `@CustomSerializer(Date::class)` должно быть отвергнуто, потому что `Date` не реализует интерфейса `ValueSerializer`.

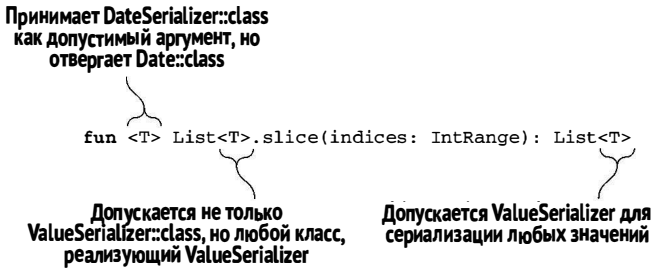


Рис. 10.5. Тип параметра аннотации serializerClass. Допускаются только ссылки на классы, наследующие ValueSerializer

Хитро, правда? Дальше мы можем применять этот шаблон всякий раз, когда требуется использовать класс как аргумент аннотации. Можно написать `KClass<out YourClassName>`, а если `YourClassName` имеет собственные типовые аргументы, то заменить их звездочкой (*).

Теперь вы познакомились со всеми важными аспектами объявления и применения аннотаций в Kotlin. Следующий шаг – узнать, как обращаться к данным, хранящимся в аннотациях. Для этого нам понадобится использовать механизм рефлексии.

10.2. Рефлексия: интроспекция объектов Kotlin во время выполнения

Механизм рефлексии позволяет обращаться к свойствам и методам объектов динамически, во время выполнения, не зная заранее, каковы они. Обычно обращение к методу или свойству в исходном коде программы оформляется как ссылка на конкретное объявление, которую компилятор обрабатывает *статически* и проверяет существование объявления. Но иногда требуется написать код, который смог бы работать с объектами любых типов или с объектами, имена свойств и методов которых становятся известны только во время выполнения. Библиотека сериализации с поддержкой формата JSON – отличный пример такого кода: она должна уметь сериализовать любой объект, поэтому не может ссылаться на конкретные классы или свойства. Механизм рефлексии помогает ей справиться с этой задачей.

Программы на Kotlin, использующие механизм рефлексии, имеют доступ к двум разным прикладным интерфейсам. Первый – стандартный механизм рефлексии в Java, реализованный в пакете `java.lang.reflect`. Поскольку классы Kotlin компилируются в обычный байт-код Java, механизм рефлексии в Java прекрасно поддерживает их. В частности, Java-библиотеки, использующие механизм рефлексии, полностью совместимы с кодом на Kotlin.

Второй – механизм рефлексии в Kotlin, реализованный в пакете `kotlin.reflect`. Он поддерживает понятия, отсутствующие в мире Java, например свойства и типы, способные принимать значение `null`. Но он не может служить полной заменой механизма рефлексии в Java, и ниже вы увидите, что иногда удобнее использовать механизм в Java. Также важно отметить, что механизм рефлексии в Kotlin не ограничивается классами Kotlin: его с успехом можно использовать для работы с классами, написанными на любом языке JVM.

Примечание. Чтобы уменьшить объем библиотеки времени выполнения на платформах, где это важно (таких как Android), библиотека рефлексии Kotlin распространяется как отдельный файл – *kotlin-reflect.jar*, который по умолчанию не добавляется в зависимости

новых проектов. Чтобы воспользоваться механизмом рефлексии Kotlin, необходимо явно добавить эту библиотеку в зависимости. IntelliJ IDEA способна обнаруживать недостающую зависимость и помогает добавлять её. Идентификатор группы/артефакта в Maven для этой библиотеки: `org.jetbrains.kotlin:kotlin-reflect`.

В этом разделе вы увидите, как JKid использует механизм рефлексии. Мы начнём с той её части, что отвечает за сериализацию (потому что так нам проще будет давать пояснения к коду), а затем перейдем к реализации парсинга JSON и десериализации. Но сначала посмотрим, что входит в механизм рефлексии.

10.2.1. Механизм рефлексии в Kotlin: KClass, KCallable, KFunction и KProperty

Главная точка входа в механизм рефлексии Kotlin – это `KClass`, представляющий класс. Это аналог `java.lang.Class`, и его можно использовать для доступа ко всем объявлениям в классе, его суперклассах и так далее. Получить экземпляр `KClass` можно с помощью выражения `MyClass::class`. Чтобы узнать класс объекта во время выполнения, сначала нужно получить его Java-класс с помощью свойства `javaClass` – прямого эквивалента Java-метода `java.lang.Object.getClass()`. Затем, чтобы выполнить переход между механизмами рефлексии Java и Kotlin, следует обратиться к свойству-расширению `kotlin`:

```
class Person(val name: String, val age: Int)

>>> val person = Person("Alice", 29)
>>> val kClass = person.javaClass.kotlin    <┐ Вернет экземпляр
>>> println(kClass.simpleName)                KClass<Person>
Person
>>> kClass.memberProperties.forEach { println(it.name) }
age
name
```

Этот простой пример выводит имя класса и имена всех его свойств. Для обхода всех его свойств, не являющихся расширениями и объявленных в классе и в суперклассах, используется `memberProperties`.

Если заглянуть в объявление класса `KClass`, можно увидеть множество полезных методов для доступа к содержимому класса:

```
interface KClass<T : Any> {
    val simpleName: String?
    val qualifiedName: String?
    val members: Collection<KCallable<*>>
    val constructors: Collection<KFunction<T>>
    val nestedClasses: Collection<KClass<*>>
```

```
    ...
}
```

Многие другие возможности KClass (включая свойство memberProperties, использованное в предыдущем примере) объявлены как расширения. Полный список методов в классе KClass (включая расширения) можно найти в справочнике по стандартной библиотеке (http://mng.bz/em4i).

Вы могли заметить, что список всех членов класса – это коллекция экземпляров KCallable. KCallable - это суперинтерфейс для функций и свойств. Он объявляет метод call, с помощью которого можно вызвать соответствующую функцию или метод чтения свойства:

```
interface KCallable<out R> {
    fun call(vararg args: Any?): R
    ...
}
```

Аргументы для вызываемой функции передаются в списке vararg. Следующий фрагмент демонстрирует, как использовать метод call для вызова функции с использованием механизма рефлексии:

```
fun foo(x: Int) = println(x)
>>> val kFunction = ::foo
>>> kFunction.call(42)
42
```

Вы уже видели синтаксис ::foo в разделе 5.1.5, а теперь можете видеть, что в этом выражении его значение – экземпляр класса KFunction, часть механизма рефлексии. Вызвать требуемую функцию можно с помощью метода KCallable.call. В данном случае нужно передать единственный аргумент: 42. Попытка вызвать функцию с неправильным количеством аргументов – например, вообще без аргументов: kFunction.call(), – приведет к появлению исключения «IllegalArgumentException: Callable expects 1 arguments, but 0 were provided» (IllegalArgumentException: Вызываемый объект ожидал получить 1 аргумент, но получил 0).

Однако в данном случае для вызова функции можно использовать более конкретный метод. Выражение ::foo имеет тип KFunction1<Int, Unit>, содержащий информацию о типах параметров и возвращаемого значения. Цифра 1 в имени означает, что у функции только один параметр. Чтобы вызвать функцию с помощью этого интерфейса, следует использовать метод invoke. Он принимает фиксированное количество аргументов (в данном случае 1), типы которых соответствуют типовым параметрам интерфейса KFunction1. Также можно вызвать kFunction непосредственно[1]:

```
import kotlin.reflect.KFunction2

fun sum(x: Int, y: Int) = x + y
```

[1] В разделе 11.3 мы расскажем, почему можно вызвать kFunction, не указывая имени метода invoke.

```
>>> val kFunction: KFunction2<Int, Int, Int> = ::sum
>>> println(kFunction.invoke(1, 2) + kFunction(3, 4))
10
>>> kFunction(1)
ERROR: No value passed for parameter p2
```

Теперь вы не сможете вызвать метод `invoke` объекта `kFunction` с неверным количеством аргументов: этот вызов просто не будет компилироваться. Соответственно, если у вас есть `KFunction` определенного типа с известными типами параметров и возвращаемого значения, предпочтительнее использовать метод `invoke`. Метод `call` – более универсальный инструмент, работающий со всеми типами функций, но он не поддерживает безопасность типов.

Где и как определены интерфейсы KFunctionN?

Типы, такие как `KFunction1`, представляют функции с разным количеством параметров. Каждый тип наследует `KFunction` и добавляет дополнительный метод `invoke` с соответствующим количеством параметров. Например, `KFunction2` объявляет `operator fun invoke(p1: P1, p2: P2): R`, где `P1` и `P2` представляют типы параметров, а `R` – тип возвращаемого значения.

Эти типы функций – *синтетические типы, генерируемые компилятором*, и вы не найдете их объявлений в пакете `kotlin.reflect`. Это означает возможность использовать интерфейс для функций с любым количеством параметров. Приём с использованием синтетических типов уменьшает размер *kotlin-runtime.jar* и помогает избежать искусственных ограничений на возможное количество параметров функций.

Для вызова метода чтения свойства тоже можно использовать метод `call` экземпляра `KProperty`. Но интерфейс поддерживает более удобный способ получения значения свойства: метод `get`.

Чтобы вызвать метод `get`, нужно использовать правильный интерфейс для свойства в зависимости от его объявления. Свойства верхнего уровня представлены экземплярами интерфейса `KProperty0`, метод `get` которого не имеет аргументов:

```
var counter = 0
>>> val kProperty = ::counter
>>> kProperty.setter.call(21)
>>> println(kProperty.get())
21
```

Вызов метода записи посредством механизма рефлексии с передачей аргумента 21

Получение значения свойства вызовом «get»

Свойство-член представлено экземпляром `KProperty1`, который имеет метод `get` с одним аргументом. Чтобы получить значение такого свой-

ства, нужно передать в аргументе экземпляр объекта, владеющего свойством. Следующий пример сохраняет ссылку на свойство в переменной `memberProperty`, а затем вызывает `memberProperty.get(person)`, чтобы получить значение этого свойства из конкретного экземпляра `person`. То есть если `memberProperty` будет ссылаться на свойство `age` класса `Person`, тогда вызов `memberProperty.get(person)` вернет значение свойства `person.age`:

```
class Person(val name: String, val age: Int)

>>> val person = Person("Alice", 29)
>>> val memberProperty = Person::age
>>> println(memberProperty.get(person))
29
```

Обратите внимание, что `KProperty1` – это обобщенный класс. Переменная `memberProperty` получит тип `KProperty<Person, Int>`, где первый типовой параметр соответствует типу получателя, а второй – типу свойства. То есть метод `get` можно вызвать, только указав верный тип получателя, – вызов `memberProperty.get("Alice")` просто не скомпилируется.

Также отметьте, что механизм рефлексии можно использовать только для обращения к свойствам, объявленным на верхнем уровне или в классе, но не к локальным переменным внутри функций. Если объявить локальную переменную `x` и попытаться обратиться к ней с использованием синтаксиса `::x`, компилятор сообщит об ошибке: «References to variables aren't supported yet» (Ссылки на переменные пока не поддерживаются).

На рис. 10.6 изображена иерархия интерфейсов, которые можно использовать для доступа к исходному коду элементов во время выполнения. Поскольку все объявления могут аннотироваться, интерфейсы, представляющие объявления во время выполнения, такие как `KClass`, `KFunction` и `Kparameter`, наследуют `KAnnotatedElement`. `KClass` используется для представления элементов двух видов: классов и объектов. `KProperty` может представлять любые свойства, а его подкласс `KMutableProperty` – только изменяемые свойства, объявленные с использованием `var`. Для работы с методами доступа к свойствам как с функциями можно использовать специальные интерфейсы `Getter` и `Setter`, объявленные в `Property` и `KMutableProperty`, – например, если понадобится извлечь их аннотации. Оба интерфейса, представляющих методы доступа, наследуют `KFunction`. Для простоты мы опустили на рис. 10.6 некоторые специальные интерфейсы для свойств, такие как `KProperty0`.

Теперь, после знакомства с основными особенностями механизма рефлексии в языке Kotlin, можно переходить к обзору реализации библиотеки JKid.

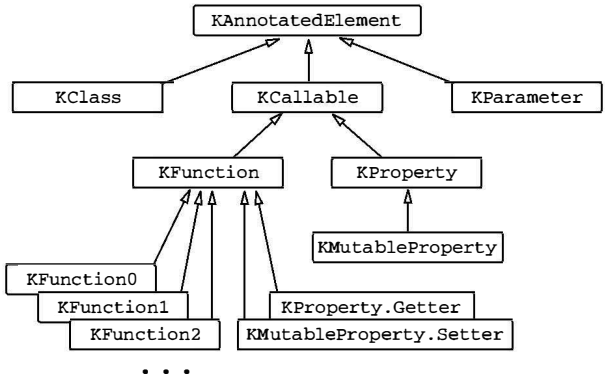


Рис. 10.6. Иерархия интерфейсов в механизме рефлексии Kotlin

10.2.2. Сериализация объектов с использованием механизма рефлексии

Для начала вспомним, как выглядит объявление функции сериализации в JKid:

```
fun serialize(obj: Any): String
```

Эта функция принимает объект и возвращает строку с его представлением в формате JSON. Строка конструируется в экземпляре `StringBuilder`. По мере сериализации свойств объекта и их значений они добавляются в конец объекта `StringBuilder`. Чтобы сделать вызов `append` более компактным, поместим реализацию в функцию-расширение для `StringBuilder`. Это позволит вызывать `append` без квалификатора:

```
private fun StringBuilder.serializeObject(x: Any) {
    append(...)
}
```

Преобразование параметра функции в получатель функции-расширения – распространенный шаблон в Kotlin, и мы подробно обсудим его в разделе 11.2.1. Обратите внимание, что функция `serializeObject` не расширяет интерфейса класса `StringBuilder`. Она выполняет операции, осмысленные только в данном конкретном контексте, поэтому мы отметили её модификатором `private`, который гарантирует невозможность использования функции в любом другом месте. Она объявлена как расширение, чтобы подчеркнуть, что конкретный объект первичен для данного блока кода, и упростить работу с этим объектом.

В результате всю работу функция `serialize` делегирует методу `serializeObject`:

```
fun serialize(obj: Any): String = buildString { serializeObject(obj) }
```

Как показано в разделе 5.5.2, `buildString` создает экземпляр `StringBuilder` и дает возможность заполнить его с использованием лямбда-выражения. В данном случае требуемое содержимое возвращается вызовом `serializeObject(obj)`.

Теперь обсудим поведение функции сериализации. По умолчанию она сериализует все свойства объекта. Простые типы и строки сериализуются как числа, логические значения и строки. Коллекции сериализуются в JSON-массивы. Свойства других типов сериализуются во вложенные объекты. Как отмечалось в предыдущем разделе, это поведение можно настраивать с применением аннотаций.

Давайте рассмотрим реализацию `serializeObject`, где вы увидите применение механизма рефлексии на практике.

Листинг 10.1. Сериализация объекта

```
private fun StringBuilder.serializeObject(obj: Any) {
    val kClass = obj.javaClass.kotlin
    val properties = kClass.memberProperties

    properties.joinToStringBuilder(
            this, prefix = "{", postfix = "}") { prop ->
        serializeString(prop.name)
        append(": ")
        serializePropertyValue(prop.get(obj))
    }
}
```

Получить экземпляр KClass для объекта

Получить все свойства класса

Получить имя свойства

Получить значение свойства

Реализация этой функции должна выглядеть очевидной: она поочередно сериализует все свойства класса. Строка JSON, получающаяся в результате, выглядит примерно так: `{ prop1: value1, prop2: value2 }`. Функция `joinToStringBuilder` обеспечивает разделение свойств запятыми. Функция `serializeString` экранирует специальные символы, как того требует формат JSON. Функция `serializePropertyValue` проверяет тип значения – примитивный, строка, коллекция или вложенный объект – и сериализует его соответственно.

В предыдущем разделе мы узнали, как получить значение экземпляра `KProperty`: вызвать метод `get`. Там мы использовали ссылку на свойство `Person::age` с типом `KProperty1<Person, Int>`, что позволило компилятору точно определить типы получателя и значения свойства. Однако в этом примере точные типы неизвестны, потому что мы перебираем все свойства класса объекта. Поэтому переменная `prop` получит тип `KProperty1<Any, *>`, а вызов `prop.get(obj)` вернет значение типа `Any`. Мы не даем никакой информации для проверки типа получателя во время компиляции, но поскольку передаем тот же объект, для которого извлекаем

список свойств, тип получателя всегда будет правильным. Посмотрим далее, как реализованы аннотации, управляющие сериализацией.

10.2.3. Настройка сериализации с помощью аннотаций

Выше в этой главе вы видели объявления аннотаций, позволяющих настраивать процесс сериализации в формат JSON. В частности, мы обсудили аннотации `@JsonExclude`, `@JsonName` и `@CustomSerializer`. Теперь пришло время посмотреть, как они обрабатываются функцией `serializeObject`.

Начнем с `@JsonExclude`. Эта аннотация позволяет исключить некоторые свойства из процесса сериализации. Давайте посмотрим, как следует изменить функцию `serializeObject` для поддержки этой возможности.

Как вы помните, для получения всех свойств класса мы использовали свойство-расширение `memberProperties` экземпляра `KClass`. Но теперь задача сложнее: мы должны отфильтровать свойства с аннотацией `@JsonExclude`. Посмотрим, как это сделать.

Интерфейс `KAnnotatedElement` определяет свойство `annotations`, возвращающее коллекцию экземпляров всех аннотаций (сохраняющихся во время выполнения), которые применялись к элементам в исходном коде. Так как `KProperty` наследует `KAnnotatedElement`, мы легко можем получить все аннотации для данного свойства с помощью `property.annotations`.

Но нам не нужны все аннотации – мы должны отыскать только конкретные, необходимые нам. Эту задачу решает вспомогательная функция `findAnnotation`:

```
inline fun <reified T> KAnnotatedElement.findAnnotation(): T?
    = annotations.filterIsInstance<T>().firstOrNull()
```

Функция `findAnnotation` возвращает аннотации с типом, указанным в качестве типового аргумента, если они есть. Она использует шаблон, обсуждавшийся в разделе 9.2.3, с овеществляемым типовым параметром (`reified`), чтобы дать возможность передавать класс аннотации в типовом аргументе.

Теперь функцию `findAnnotation` можно использовать вместе с функцией `filter` из стандартной библиотеки, чтобы отфильтровать свойства с аннотацией `@JsonExclude`:

```
val properties = kClass.memberProperties
    .filter { it.findAnnotation<JsonExclude>() == null }
```

Следующая аннотация: `@JsonName`. Мы напомним её объявление и пример использования:

```
annotation class JsonName(val name: String)

data class Person(
```

```
    @JsonName("alias") val firstName: String,
    val age: Int
)
```

В данном случае нас интересует не только сама аннотация, но и её аргумент: имя, которое должно использоваться для обозначения свойства в JSON-представлении. К счастью, в этом случае нам тоже поможет функция findAnnotation:

```
val jsonNameAnn = prop.findAnnotation<JsonName>()
val propName = jsonNameAnn?.name ?: prop.name
```

Получить экземпляр аннотации @JsonName, если имеется

Получить аргумент «name» или использовать «prop.name» по умолчанию

Если свойство не снабжено аннотацией @JsonName, то jsonNameAnn получит значение null и мы будем использовать имя свойства prop.name. Если аннотация присутствует, то будет использовано указанное в ней имя.

Рассмотрим порядок сериализации экземпляра класса Person, объявленного выше. В ходе сериализации свойства firstName переменная jsonNameAnn получит ссылку на соответствующий экземпляр класса аннотации JsonName. То есть jsonNameAnn?.name вернет непустое значение "alia", которое будет использовано как ключ в JSON-представлении. В ходе сериализации свойства age аннотация не обнаружится, поэтому в качестве ключа будет использовано собственное имя свойства age.

Добавим все изменения, о которых мы рассказали, и посмотрим на получившуюся реализацию.

Листинг 10.2. Сериализация объекта с фильтрацией свойств

```
private fun StringBuilder.serializeObject(obj: Any) {
    obj.javaClass.kotlin.memberProperties
            .filter { it.findAnnotation<JsonExclude>() == null }
            .joinToStringBuilder(this, prefix = "{", postfix = "}") {
                serializeProperty(it, obj)
            }
}
```

Теперь свойства с аннотацией @JsonExclude будут отфильтровываться. Мы также извлекли логику сериализации свойства в отдельную функцию serializeProperty.

Листинг 10.3. Сериализация единственного свойства

```
private fun StringBuilder.serializeProperty(
        prop: KProperty1<Any, *>, obj: Any
) {
    val jsonNameAnn = prop.findAnnotation<JsonName>()
```

```
    val propName = jsonNameAnn?.name ?: prop.name
    serializeString(propName)
    append(": ")

    serializePropertyValue(prop.get(obj))
}
```

Имя свойства обрабатывается в соответствии с аннотацией `@JsonName`, как обсуждалось выше.

Перейдем к реализации поддержки последней оставшейся аннотации, `@CustomSerializer`. Реализация основана на функции `getSerializer`, которая возвращает экземпляр `ValueSerializer`, зарегистрированный в аннотации `@CustomSerializer`. Например, если объявить класс `Person`, как показано ниже, и вызвать `getSerializer()` во время сериализации свойства `birthDate`, она вернет экземпляр `DateSerializer`:

```
data class Person(
    val name: String,
    @CustomSerializer(DateSerializer::class) val birthDate: Date
)
```

Повторим объявление аннотации `@CustomSerializer`, чтобы вам проще было понять реализацию `getSerializer`:

```
annotation class CustomSerializer(
    val serializerClass: KClass<out ValueSerializer<*>>
)
```

А вот как реализована функция `getSerializer`.

Листинг 10.4. Извлечение объекта, реализующего сериализацию

```
fun KProperty<*>.getSerializer(): ValueSerializer<Any?>? {
    val customSerializerAnn = findAnnotation<CustomSerializer>() ?: return null
    val serializerClass = customSerializerAnn.serializerClass

    val valueSerializer = serializerClass.objectInstance
            ?: serializerClass.createInstance()
    @Suppress("UNCHECKED_CAST")
    return valueSerializer as ValueSerializer<Any?>
}
```

Это функция-расширение для `KProperty`, потому что свойство – это первичный объект, который обрабатывается методом. Она вызывает функцию `findAnnotation`, чтобы получить экземпляр аннотации `@CustomSerializer`, если он существует. Её аргумент, `serializerClass`, определяет класс, экземпляр которого требуется получить.

Самое интересное здесь – способ обработки классов и объектов («одиночек» в Kotlin) как значений аннотации `@CustomSerializer`. И те, и другие представлены классом `KClass`. Разница лишь в том, что объекты имеют непустое свойство `objectInstance`, которое можно использовать для доступа к единственному экземпляру, созданному для объекта. Например, `DateSerializer` объявлен как объект, поэтому его свойство `objectInstance` хранит ссылку на единственный экземпляр `DateSerializer`. Для сериализации будет использоваться этот экземпляр, и вызов `createInstance` не произойдет.

Если `KClass` представляет обычный класс, то вызовом `createInstance` создается новый экземпляр. Эта функция похожа на `java.lang.Class.newInstance`.

Теперь можно задействовать `getSerializer` в реализации `serializeProperty`. Ниже приводится окончательная версия функции.

Листинг 10.5. Сериализация свойства с поддержкой нестандартного способа сериализации

```
private fun StringBuilder.serializeProperty(
        prop: KProperty1<Any, *>, obj: Any
) {
    val name = prop.findAnnotation<JsonName>()?.name ?: prop.name
    serializeString(name)
    append(": ")

    val value = prop.get(obj)
    val jsonValue =
            prop.getSerializer()?.toJsonValue(value)
            ?: value
    serializePropertyValue(jsonValue)
}
```

Использовать нестандартную реализацию сериализации для свойства, если указана

Иначе использовать значение свойства, как прежде

Если для преобразования значения свойства в JSON-совместимый формат указан нестандартный механизм сериализации, `serializeProperty` использует его, вызывая `toJsonValue`. Если же он не указан, используется фактическое значение свойства.

Теперь, после знакомства с частью библиотеки, осуществляющей сериализацию в формат JSON, перейдем к парсингу и десериализации. Для десериализации потребуется написать больше кода, поэтому мы не будем исследовать его полностью, а рассмотрим только структуру реализации и объясним, как использовать механизм рефлексии для десериализации объектов.

10.2.4. Парсинг формата JSON и десериализация объектов

Начнем со второй части нашей истории: с реализации логики десериализации. Прежде всего отметим, что API десериализации состоит из единственной функции:

```
inline fun <reified T: Any> deserialize(json: String): T
```

Вот как она используется:

```
data class Author(val name: String)
data class Book(val title: String, val author: Author)

>>> val json = """{"title": "Catch-22", "author": {"name": "J. Heller"}}"""
>>> val book = deserialize<Book>(json)
>>> println(book)
Book(title=Catch-22, author=Author(name=J. Heller))
```

Мы передаем функции `deserialize` тип объекта как овеществляемый типовой параметр и получаем от неё новый экземпляр объекта.

Десериализация формата JSON – более сложная задача, чем сериализация, потому что вовлекает в себя не только механизм рефлексии, но и парсинг JSON-строки. Десериализация JSON в библиотеке JKid реализована традиционным образом и состоит из трёх этапов: лексический анализ, синтаксический анализ (парсинг), а также собственно компонент десериализации.

В процессе лексического анализа входная строка разбивается на список лексем. Различаются два вида лексем: *символьные лексемы*, представляющие отдельные символы со специальным значением (запятые, двоеточия и скобки, квадратные и фигурные), и *лексемы-значения*, соответствующие строкам, числам, логическим значениям и значениям `null`. Примерами лексем разного вида могут служить: левая фигурная скобка ({), строковое значение (`"Catch-22"`) и целочисленное значение (`42`).

Парсер (выполняющий синтаксический анализ) отвечает за преобразование списка лексем в структурированное представление. Его задача в библиотеке JKid – распознать общую структуру JSON и преобразовать отдельные лексемы в семантически значимые элементы: пары ключ/значение, объекты и массивы.

Интерфейс `JsonObject` представляет объект или массив, который десериализуется в данный момент. Обнаруживая новые свойства текущего объекта (простые значения, составные свойства или массивы), парсер вызывает соответствующие методы.

Листинг 10.6. Интерфейс обратных вызовов парсера JSON

```
interface JsonObject {
    fun setSimpleProperty(propertyName: String, value: Any?)

    fun createObject(propertyName: String): JsonObject

    fun createArray(propertyName: String): JsonObject
}
```

В параметре `propertyName` эти методы принимают ключ JSON. То есть, когда парсер встречает свойство `author` с объектом в качестве значения, он вызывает метод `createObject("author")`. Для свойств с простыми значениями вызывается метод `setSimpleProperty`, которому в аргументе `value` передается фактическая лексема. Реализации `JsonObject` отвечают за создание новых объектов для свойств и сохранение ссылок на них во внешнем объекте.

На рис. 10.7 показаны входные и выходные данные для каждого этапа лексического и синтаксического анализа при десериализации строки примера. Еще раз напомним: в ходе лексического анализа входная строка преобразуется в список лексем, а затем этот список обрабатывается на этапе синтаксического анализа (парсинга), и для каждого значимого элемента вызываются соответствующие методы интерфейса `JsonObject`.

Собственно десериализация выполняется реализацией `JsonObject`, которая постепенно конструирует новый экземпляр соответствующего типа. В ходе этого процесса нужно определить соответствие между свойствами класса и ключами JSON (`title`, `author` и `name` на рис. 10.7) и создать вложенные объекты-значения (например, экземпляр `Author`), и только после этого можно создать новый экземпляр требуемого класса (`Book`).

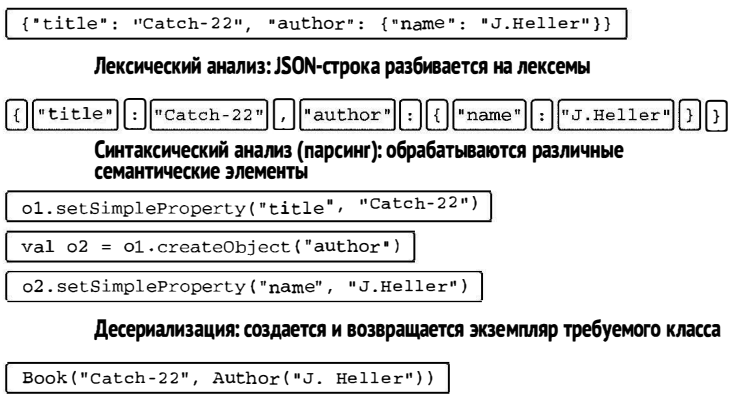


Рис. 10.7. Парсинг строки в формате JSON: лексический и синтаксический анализ и десериализация

Библиотека JKid создавалась для работы с классами данных, поэтому все пары ключ/значение, загруженные из файла JSON, она передает конструктору десериализуемого класса как параметры. Она не поддерживает установку свойств в экземплярах объектов после их создания. Это означает, что данные должны где-то храниться до момента создания объекта, пока происходит чтение файла JSON.

Требование хранить компоненты до создания объекта очень напоминает традиционный шаблон «Строитель» (Builder), с той лишь разницей, что реализации этого шаблона обычно предназначены для создания объектов конкретного типа, а наше решение должно быть полностью универсальным. Чтобы было веселее, используем для обозначения реализации термин «зерно» (seed). Десериализуя формат JSON, мы должны уметь создавать разные составные конструкции: объекты, коллекции и словари. Классы `ObjectSeed`, `ObjectListSeed` и `ValueListSeed` отвечают за создание объектов и списков составных объектов или простых значений соответственно. Конструирование словарей мы оставляем вам как упражнение для самостоятельного решения.

Базовый интерфейс `Seed` наследует `JsonObject` и объявляет дополнительный метод `spawn` («прорастить»), возвращающий созданный экземпляр по окончании процесса конструирования. Он также объявляет метод `createCompositeProperty` для создания вложенных объектов и списков (для их конструирования используется та же базовая логика).

Листинг 10.7. Интерфейс для создания объектов из JSON-файлов

```
interface Seed: JsonObject {
    fun spawn(): Any?

    fun createCompositeProperty(
        propertyName: String,
        isList: Boolean
    ): JsonObject

    override fun createObject(propertyName: String) =
        createCompositeProperty(propertyName, false)

    override fun createArray(propertyName: String) =
        createCompositeProperty(propertyName, true)

    // ...
}
```

Метод `spawn` можно считать аналогом метода `build`, возвращающего окончательное значение. Он возвращает сконструированный объект для `ObjectSeed` или список для `ObjectListSeed` или `ValueListSeed`. Мы не бу-

дем подробно обсуждать, как десериализуются списки, а сосредоточим все внимание на создании объектов – более сложной и вместе с тем наглядно демонстрирующей основную идею части.

Но прежде рассмотрим основную функцию `deserialize`, которая выполняет всю работу по десериализации значения.

Листинг 10.8. Функция десериализации верхнего уровня

```
fun <T: Any> deserialize(json: Reader, targetClass: KClass<T>): T {
    val seed = ObjectSeed(targetClass, ClassInfoCache())
    Parser(json, seed).parse()
    return seed.spawn()
}
```

Перед началом парсинга создается экземпляр `ObjectSeed` для хранения свойств будущего объекта, а затем вызывается парсер, которому передается объект `json`, читающий входную строку. После обработки входной строки вызывается функция `spawn`, которая должна сконструировать окончательный объект.

Теперь обратим внимание на реализацию `ObjectSeed`, который хранит состояние конструируемого объекта. `ObjectSeed` принимает ссылку на целевой класс и объект `classInfoCache`, хранящий информацию о свойствах класса. Эта информация позже будет использована для создания экземпляров класса. `ClassInfoCache` и `ClassInfo` – вспомогательные классы, которые мы обсудим в следующем разделе.

Листинг 10.9. Десериализация объекта

```
class ObjectSeed<out T: Any>(
        targetClass: KClass<T>,
        val classInfoCache: ClassInfoCache
) : Seed {

    private val classInfo: ClassInfo<T> =
            classInfoCache[targetClass]

    private val valueArguments = mutableMapOf<KParameter, Any?>()
    private val seedArguments = mutableMapOf<KParameter, Seed>()

    private val arguments: Map<KParameter, Any?>
        get() = valueArguments +
                seedArguments.mapValues { it.value.spawn() }

    override fun setSimpleProperty(propertyName: String, value: Any?) {
        val param = classInfo.getConstructorParameter(propertyName)
```

Кэширует информацию, необходимую для создания экземпляра targetClass

Создание словаря из параметров конструктора и их значений

```
        valueArguments[param] =
                classInfo.deserializeConstructorArgument(param, value)
    }

    override fun createCompositeProperty(
            propertyName: String, isList: Boolean
    ): Seed {
        val param = classInfo.getConstructorParameter(propertyName)
        val deserializeAs =
            classInfo.getDeserializeClass(propertyName)
        val seed = createSeedForType(
            deserializeAs ?: param.type.javaType, isList)
        return seed.apply { seedArguments[param] = this }
    }

    override fun spawn(): T =
            classInfo.createInstance(arguments)
}
```

Запись значения для параметра конструктора, если это простое значение

Загружает значение из аннотации DeserializeInterface, если имеется

Создание ObjectSeed или Co le tionSeed, в зависимости от типа параметра...

...и запись его в словарь seedArguments

Создание экземпляра targetC ass с передачей словаря аргументов

`ObjectSeed` конструирует словарь из параметров конструктора и их значений. Для этого используются два изменяемых словаря: `valueArguments` для простых значений и `seedArguments` для составных значений. Новые аргументы добавляются в словарь `valueArguments` вызовом `setSimpleProperty`, а в словарь `seedArguments` – вызовом `createCompositeProperty`. Новые составные «зёрна» изначально имеют пустое значение, а затем заполняются данными, извлекаемыми из входного потока. В заключение метод `spawn` рекурсивно собирает все сконструированные «зёрна», вызывая метод `spawn` каждого из них.

Обратите внимание, что вызов `arguments` в теле метода `spawn` запускает рекурсивную процедуру конструирования аргументов: метод чтения свойства `arguments` вызывает методы `spawn` всех аргументов в `seedArguments`. Функция `createSeedForType` анализирует тип параметра и создает `ObjectSeed`, `ObjectListSeed` или `ValueListSeed`, в зависимости от типа параметра. Исследование её реализации мы оставляем вам в качестве самостоятельного упражнения.

Теперь посмотрим, как функция `ClassInfo.createInstance` создает экземпляр `targetClass`.

10.2.5. Заключительный этап десериализации: callBy() и создание объектов с использованием рефлексии

Последний этап, с которым мы должны разобраться, – как класс `ClassInfo` конструирует окончательный экземпляр и кэширует информацию о параметрах конструктора. Он используется в `ObjectSeed`. Но, преж-

де чем углубиться в детали реализации, познакомимся с программным интерфейсом механизма рефлексии, позволяющим создавать объекты.

Вы уже видели метод `KCallable.call`, который вызывает функцию или конструктор, передавая им полученный список аргументов. Этот метод прекрасно справляется со своей задачей в большинстве случаев, но имеет существенное ограничение: он не поддерживает параметров со значениями по умолчанию. В нашем случае, если пользователь пытается десериализовать объект, конструктор которого имеет параметры со значениями по умолчанию, мы определенно не должны требовать присутствия соответствующих аргументов в строке JSON. Поэтому используем другой метод, поддерживающий параметры со значениями по умолчанию: `KCallable.callBy`.

```
interface KCallable<out R> {
    fun callBy(args: Map<KParameter, Any?>): R
    ...
}
```

Метод принимает словарь с параметрами и их значениями, которые впоследствии будут переданы как аргументы. Если параметр в словаре отсутствует, будет использовано его значение по умолчанию (если определено). Также интересно отметить, что параметры необязательно должны указываться в правильном порядке – можно просто читать пары имя/значение из JSON, отыскивать соответствующий параметр и добавлять его значение в словарь.

Единственное, о чем действительно необходимо позаботиться, – выбор правильного типа. Тип значения в словаре `args` должен соответствовать типу параметра в конструкторе – в противном случае возникнет исключение `IllegalArgumentException`. Это особенно важно для числовых типов: мы должны точно знать тип параметра – `Int`, `Long`, `Double` или какой-то другой простой тип – и преобразовать число из JSON в значение правильного типа. Для этого используется свойство `KParameter.type`.

Преобразование типа выполняется через тот же интерфейс `ValueSerializer`, использовавшийся для нестандартной сериализации. Если свойство не отмечено аннотацией `@CustomSerializer`, мы извлекаем стандартную реализацию для данного типа.

Листинг 10.10. Получение сериализатора по типу значения

```
fun serializerForType(type: Type): ValueSerializer<out Any?>? =
    when(type) {
        Byte::class.java -> ByteSerializer
        Int::class.java -> IntSerializer
        Boolean::class.java -> BooleanSerializer
        // ...
```

```
        else -> null
    }
```

Соответствующая реализация `ValueSerializer` выполнит необходимую проверку типа или преобразование.

Листинг 10.11. Сериализатор для логических значений

```
object BooleanSerializer : ValueSerializer<Boolean> {
    override fun fromJsonValue(jsonValue: Any?): Boolean {
        if (jsonValue !is Boolean) throw JKidException("Boolean expected")
        return jsonValue
    }

    override fun toJsonValue(value: Boolean) = value
}
```

Метод `callBy` дает возможность вызвать основной конструктор объекта и передать ему словарь с параметрами и соответствующими значениями. Механизм `ValueSerializer` гарантирует, что значения в словаре будут иметь правильные типы. Теперь посмотрим, как вызвать этот метод.

Цель класса `ClassInfoCache` – уменьшить накладные расходы, связанные с использованием механизма рефлексии. Напомним, что аннотации, управляющие сериализацией и десериализацией (`@JsonName` и `@CustomSerializer`), применяются к свойствам, а не к параметрам. Когда производится десериализация объекта, мы имеем дело с параметрами конструктора, а не со свойствами, поэтому для получения аннотаций необходимо отыскать соответствующие свойства. Если выполнять такой поиск при извлечении каждой пары ключ/значение, это слишком замедлит процесс десериализации, поэтому мы делаем это только один раз для каждого класса и сохраняем информацию в кэше. Ниже приводится полная реализация `ClassInfoCache`.

Листинг 10.12. Хранилище данных, полученных с помощью механизма рефлексии

```
class ClassInfoCache {
    private val cacheData = mutableMapOf<KClass<*>, ClassInfo<*>>()

    @Suppress("UNCHECKED_CAST")
    operator fun <T : Any> get(cls: KClass<T>): ClassInfo<T> =
            cacheData.getOrPut(cls) { ClassInfo(cls) } as ClassInfo<T>
}
```

Здесь используется шаблон, обсуждавшийся в разделе 9.3.6: мы удаляем информацию о типе, когда сохраняем значения в словаре, но реализация метода `get` гарантирует, что возвращаемый `ClassInfo<T>` имеет правиль-

ный аргумент типа. Обратите внимание на вызов `getOrPut`: если словарь `cacheData` уже хранит элемент для `cls`, он вернет этот элемент. Иначе будет вызвано лямбда-выражение, которое вычислит значение для ключа, сохранит его в словаре и вернет.

Класс `ClassInfo` отвечает за создание нового экземпляра целевого класса и кэширование необходимой информации. Чтобы упростить код, мы опустили некоторые функции и тривиальные операции инициализации. Также можно заметить, что вместо использования `!!` готовый код возбуждает исключение с информативным сообщением (вы тоже можете взять на вооружение этот отличный приём).

Листинг 10.13. Кэширование параметров конструктора и данных из аннотаций

```
class ClassInfo<T : Any>(cls: KClass<T>) {
    private val constructor = cls.primaryConstructor!!

    private val jsonNameToParamMap = hashMapOf<String, KParameter>()
    private val paramToSerializerMap =
        hashMapOf<KParameter, ValueSerializer<out Any?>>()
    private val jsonNameToDeserializeClassMap =
        hashMapOf<String, Class<out Any>?>()

    init {
        constructor.parameters.forEach { cacheDataForParameter(cls, it) }
    }

    fun getConstructorParameter(propertyName: String): KParameter =
            jsonNameToParam[propertyName]!!

    fun deserializeConstructorArgument(
            param: KParameter, value: Any?): Any? {
        val serializer = paramToSerializer[param]
        if (serializer != null) return serializer.fromJsonValue(value)

        validateArgumentType(param, value)
        return value
    }

    fun createInstance(arguments: Map<KParameter, Any?>): T {
        ensureAllParametersPresent(arguments)
        return constructor.callBy(arguments)
    }

    // ...
}
```

В разделе инициализации этот код отыскивает свойства, соответствующие параметрам конструктора, и извлекает их аннотации. Данные сохраняются в трех словарях: `jsonNameToParam` – определяет параметр, соответствующий каждому ключу в файле JSON; `paramToSerializer` – хранит реализации сериализации для всех параметров; и `jsonNameToDeserializeClass` – хранит класс, заданный в аргументе аннотации `@DeserializeInterface`, если имеется. Затем `ClassInfo` передает параметры конструктора по именам свойств и вызывает код, использующий параметр как ключ для отображения параметра в аргумент.

Функции `cacheDataForParameter`, `validateArgumentType` и `ensureAllParametersPresent` – приватные. Ниже приводится реализация `ensureAllParametersPresent`, а код других функций вы можете исследовать самостоятельно.

Листинг 10.14. Проверка наличия требуемых параметров

```
private fun ensureAllParametersPresent(arguments: Map<KParameter, Any?>) {
    for (param in constructor.parameters) {
        if (arguments[param] == null &&
            !param.isOptional && !param.type.isMarkedNullable) {
            throw JKidException("Missing value for parameter ${param.name}")
        }
    }
}
```

Эта функция проверяет наличие значений для всех параметров. Обратите внимание, как здесь использован механизм рефлексии. Если параметр имеет значение по умолчанию, то `param.isOptional` принимает значение `true` и мы можем опустить аргумент для него – в этом случае будет использоваться значение по умолчанию. Если тип параметра допускает значение `null` (`type.isMarkedNullable` сообщит об этом), в качестве значения по умолчанию будет использовано `null`. Для всех остальных параметров требуется передать соответствующие аргументы, иначе будет возбуждено исключение. Кэширование информации, извлеченной с использованием механизма рефлексии, гарантирует, что поиск аннотаций будет производиться только один раз, а не для каждого свойства, встреченного в JSON-данных.

На этом мы завершаем обсуждение реализации библиотеки JKid. В этой главе мы исследовали библиотеку сериализации/десериализации JSON, реализованную поверх механизма рефлексии и использующую аннотации для управления её поведением. Все продемонстрированные приёмы и подходы вы можете успешно использовать в своих фреймворках.

10.3. Резюме

- Синтаксис применения аннотаций в Kotlin практически неотличим от Java.
- Kotlin позволяет применять аннотации к более широкому кругу элементов, чем Java, включая файлы и выражения.
- Аргумент аннотации может быть значением простого типа, строкой, перечислением, ссылкой на класс, экземпляром класса другой аннотации или их массивом.
- Определяя целевой элемент аннотаций в месте использования, например `@get:Rule`, можно указать, к какому элементу применяется аннотация, если единственное объявление на Kotlin порождает несколько элементов в байт-коде.
- Класс аннотации объявляется как класс с основным конструктором, не имеющим тела, все параметры которого являются `val`-свойствами.
- С помощью метааннотаций можно определить цель, режим сохранения и другие атрибуты аннотаций.
- Механизм рефлексии позволяет перебирать и обращаться к методам и свойствам объектов во время выполнения. Он имеет интерфейсы, представляющие разные виды объявлений, такие как классы (`KClass`), функции (`KFunction`) и так далее.
- Получить экземпляр `KClass` можно с помощью `ClassName::class`, если класс заранее известен, или `obj.javaClass.kotlin`, чтобы получить класс из экземпляра объекта.
- Оба интерфейса, `KFunction` и `KProperty`, наследуют `KCallable`, который определяет обобщенный метод `call`.
- Для вызова методов, имеющих параметры со значениями по умолчанию, можно использовать метод `KCallable.callBy`.
- `KFunction0`, `KFunction1` и так далее – это функции с разным количеством параметров, которые можно использовать для вызова методов.
- `KProperty0` и `KProperty1` – это свойства с разным количеством приемников. Они поддерживают метод `get` для получения значения. `KMutableProperty0` и `KMutableProperty1` наследуют эти интерфейсы для поддержки свойств, позволяющих изменять значения вызовом метода `set`.

Глава 11

Конструирование DSL

В этой главе:

- создание предметно-ориентированных языков;
- использование лямбда-выражений с получателями;
- применение соглашения `invoke`;
- примеры существующих предметно-ориентированных языков на Kotlin.

В этой главе мы обсудим подходы к проектированию выразительных и идиоматичных программных интерфейсов (API) классов на языке Kotlin с использованием *предметно-ориентированных языков* (Domain-Specific Languages, DSL). Исследуем различия между традиционными и DSL-ориентированными API и посмотрим, как можно применять DSL-ориентированные API для решения широкого круга практических задач: операций с базами данных, динамического создания страниц HTML, тестирования, создания сценариев сборки, определения макетов пользовательского интерфейса для Android и многих других.

Проектирование предметно-ориентированных языков в Kotlin опирается на многие возможности языка, две из которых мы пока не исследовали полностью. С одной из них мы уже сталкивались в главе 5: лямбда-выражения с получателями, которые дают возможность создать структуру DSL, изменяя правила разрешения имен в блоках кода. Другая – новая для вас: соглашение `invoke`, позволяющее более гибко комбинировать лямбда-выражения и операции присваивания значений свойствам в коде DSL. В этой главе мы детально исследуем обе эти особенности.

11.1. От API к DSL

Прежде чем углубиться в обсуждение предметно-ориентированных языков, обсудим задачу, стоящую перед нами. Наша конечная цель – достичь

максимальной читаемости и выразительности кода. Этой цели невозможно достичь, если всё внимание сосредоточить только на отдельных классах. Большая часть кода в классах взаимодействует с другими классами, поэтому мы должны уделить внимание интерфейсам, через которые протекают взаимодействия, или, иными словами, программным интерфейсам классов.

Важно помнить, что создание выразительных и удобных API – прерогатива не только создателей библиотек, но и каждого разработчика. Подобно библиотекам, предоставляющим программные интерфейсы для их использования, каждый класс в приложении предоставляет другим классам возможность взаимодействовать с ним. Удобство и выразительность взаимодействий помогут обеспечить простоту сопровождения проекта в будущем.

В ходе чтения этой книги вы познакомились со множеством особенностей Kotlin, которые позволяют создавать *ясные API* для классов. Но что имеется в виду под выражением «ясный API»? Две вещи:

- читателям кода должно быть ясно, что он делает. Этого можно добиться, выбирая говорящие имена и понятия (что справедливо для кода на любом языке программирования);
- код должен выглядеть прозрачным, простым и не перегруженным избыточными синтаксическими конструкциями. То, как достичь этой цели, – основное содержание этой главы. Ясный API порой даже может не отличаться от встроенных особенностей языка.

В число особенностей Kotlin, помогающих создавать ясные API, входят: функции-расширения, инфиксные вызовы, сокращенный синтаксис лямбда-выражений и перегрузка операторов. В табл. 11.1 демонстрируется, как они помогают уменьшить синтаксическую нагрузку на код.

Таблица 11.1. Особенности Kotlin для поддержки ясного синтаксиса

Обычный синтаксис	Улучшенный синтаксис	Используемая особенность
`StringUtil.capitalize(s)`	`s.capitalize()`	Функция-расширение
`1.to("one")`	`1 to "one"`	Инфиксный вызов
`set.add(2)`	`set += 2`	Перегрузка операторов
`map.get("key")`	`map["key"]`	Соглашение о вызове метода `get`
`file.use({ f -> f.read() } )`	`file.use { it.read() }`	Лямбда-выражения вне круглых скобок

`sb.append("yes")`
`sb.append("no")`
	`with (sb) {`
	`  append("yes")`
	`  append("no")`
	`}`
		Лямбда-выражения с получателем

В этой главе мы выйдем за рамки определения ясных API и познакомимся с поддержкой создания предметно-ориентированных языков в Kotlin. Эта поддержка основывается на особенностях, помогающих создавать ясные API, и дополняет их возможностью создания структур из вызовов нескольких методов. Получающиеся в результате предметно-ориентированные языки могут быть более выразительными и удобными, чем программные интерфейсы, сконструированные из отдельных вызовов методов.

Так же как другие особенности, поддержка DSL в Kotlin является *полностью статически типизированной* и включает все преимущества статической типизации, такие как выявление ошибок на этапе компиляции и улучшенная поддержка в IDE.

Чтобы вы могли получить общее представление, ниже приводится пара примеров, демонстрирующих возможности поддержки DSL в Kotlin. Следующее выражение возвращает предыдущий день (точнее, всего лишь его дату):

```
val yesterday = 1.days.ago
```

а эта функция генерирует HTML-таблицу:

```
fun createSimpleTable() = createHTML().
    table {
        tr {
            td { +"cell" }
        }
    }
```

В данной главе вы узнаете, как устроены эти примеры. Но, перед тем как углубиться в детали, давайте выясним, что же такое эти предметно-ориентированные языки.

11.1.1. Понятие предметно-ориентированного языка

Понятие предметно-ориентированного языка появилось почти одновременно с понятием языка программирования как такового. Мы различаем *языки программирования общего назначения*, обладающие достаточно полным набором возможностей для решения практически любых задач с использованием компьютера, и *предметно-ориентированные языки* (Domain-Specific Language, DSL), ориентированные на решение задач из одной конкретной *предметной области* и не обладающие средствами для решения любых других задач.

Типичные примеры предметно-ориентированных языков, с которыми вы наверняка знакомы, – SQL и регулярные выражения. Они прекрасно приспособлены для решения узкого круга задач управления базами данных и текстовыми строками соответственно, но у вас не получится написать на них целое приложение. (По крайней мере, мы на это надеемся.

Даже мысль о том, что кто-то может написать целое приложение на языке регулярных выражений, вызывает у нас дрожь.)

Заметьте, как в этих языках увеличивается эффективность в достижении цели за счет ограничения их технических возможностей. Когда нужно выполнить SQL-запрос, мы не пишем определение, класса или функции, а используем конкретное ключевое слово, которое определяет тип запроса и предполагает определенный синтаксис и набор сопутствующих ключевых слов. Язык регулярных выражений обладает ещё более узким кругом синтаксических конструкций: программа непосредственно описывает, какой текст будет совпадать с шаблоном, и использует компактный синтаксис из знаков препинания, чтобы показать, какие отличия в совпавшем тексте могут считаться допустимыми. Благодаря такому компактному синтаксису на DSL можно более кратко выразить предметную операцию, чем на языке общего назначения.

Другая важная особенность предметно-ориентированных языков – их стремление к декларативному синтаксису, тогда как языки общего назначения в большинстве случаев императивные. *Императивный язык* описывает последовательность действий, которые требуется выполнить для завершения операции, а *декларативный язык* описывает желаемый результат, оставляя детали его получения на усмотрение движка, который интерпретирует код на этом языке. Благодаря этому часто можно повысить эффективность кода, потому что все необходимые оптимизации реализуются только один раз – в исполняющем движке. Императивный подход, напротив, требует оптимизировать каждую операцию отдельно.

В противовес всем этим достоинствам предметно-ориентированные языки имеют один существенный недостаток: код на этих языках сложно встраивать в приложения, написанные на языках общего назначения. Они имеют свой отличительный синтаксис, который нельзя напрямую использовать в программах на другом языке. Поэтому, чтобы вызвать программу, написанную на предметно-ориентированном языке, её приходится сохранять в отдельном файле или встраивать в основную программу в виде строкового литерала. Это существенно усложняет проверку правильности взаимодействий DSL с основным языком на этапе компиляции, отладку программ на DSL и поддержку в IDE. Кроме того, отличительный синтаксис требует дополнительных знаний и часто делает код трудным для чтения.

Чтобы решить эту проблему и сохранить основные преимущества DSL, недавно была предложена идея внутренних предметно-ориентированных языков. Давайте посмотрим, в чем она заключается.

11.1.2. Внутренние предметно-ориентированные языки

В противоположность *внешним DSL*, обладающим собственным синтаксисом, *внутренние DSL* – это часть программы, написанная на языке обще-

го назначения и имеющая точно такой же синтаксис. То есть внутренний DSL нельзя считать полностью независимым языком – скорее, это иной способ использования основного языка с сохранением преимуществ, присущих предметно-ориентированным языкам.

Для сравнения посмотрим, как одну и ту же задачу можно решить с помощью внешнего и внутреннего DSL. Представьте, что в нашей базе данных есть две таблицы, `Customer` и `Country`, причём каждая запись в `Customer` ссылается на запись в таблице `Country`, которая определяет страну проживания клиента. Наша задача: определить страну, где живет большинство наших клиентов. В качестве внешнего DSL будем использовать SQL, а в качестве внутреннего – язык, реализованный в фреймворке Exposed (https://github.com/JetBrains/Exposed), написанном на языке Kotlin и предназначенном для доступа к базам данных. Вот как выглядит решение на языке SQL:

```
SELECT Country.name, COUNT(Customer.id)
    FROM Country
    JOIN Customer
      ON Country.id = Customer.country_id
GROUP BY Country.name
ORDER BY COUNT(Customer.id) DESC
   LIMIT 1
```

Мы не можем просто вставить код SQL в программу: нам нужен некоторый механизм, поддерживающий взаимодействия между кодом на основном языке приложения (в данном случае Kotlin) и кодом на языке запросов. Обычно лучшее, на что можно рассчитывать, – это поместить SQL-запрос в строковый литерал и надеяться, что наша IDE поможет написать и проверить его.

Для сравнения ниже приводится тот же запрос, сконструированный с использованием возможностей Kotlin и Exposed:

```
(Country join Customer)
    .slice(Country.name, Count(Customer.id))
    .selectAll()
    .groupBy(Country.name)
    .orderBy(Count(Customer.id), isAsc = false)
    .limit(1)
```

Как видите, эти две версии довольно схожи. Фактически вторая версия генерирует и выполняет тот же SQL-запрос, что мы написали вручную, но она написана на языке Kotlin, а `selectAll`, `groupBy`, `orderBy` и другие инструкции являются обычными методами. Кроме того, нам не нужно тратить времени и сил на преобразование данных, возвращаемых SQL-запросом, в объекты Kotlin – они сразу возвращаются в виде самых обычных объектов Kotlin. Итак, внутренним DSL мы называем код, пред-

назначенный для решения конкретной задачи (конструирования SQL-запросов) и реализованный в виде библиотеки на языке общего назначения (Kotlin).

11.1.3. Структура предметно-ориентированных языков

Вообще говоря, не существует четких границ между DSL и обычным API, и часто критерием становится субъективное мнение: «Для меня этот код выглядит как код на предметно-ориентированном языке». Предметно-ориентированные языки очень часто опираются на те особенности языков программирования, которые широко используются в других контекстах, например инфиксные вызовы и перегрузка операторов. Но помимо этого, предметно-ориентированные языки часто обладают характерной чертой, отсутствующей в других API: *структурой*, или *грамматикой*.

Типичная библиотека состоит из множества методов, и использующий её клиент вызывает эти методы по одному. Последовательности вызовов методов не имеют предопределенной структуры, а контекст выполняемых операций не сохраняется между вызовами. Такие API иногда называют *командными API*. Напротив, вызовы методов в DSL образуют более крупные структуры, определяемые *грамматикой* DSL. В Kotlin структура DSL обычно создается с применением вложенных лямбда-выражений или цепочек из вызовов методов. Это видно в предыдущем примере: чтобы выполнить запрос, необходимо вызвать комбинацию методов, описывающих разные аспекты требуемого набора результатов, и такая комбинация читается проще, чем единственный вызов, принимающий все необходимые для создания запроса аргументы.

Наличие грамматики – вот что позволяет нам называть внутренний DSL *языком*. В естественных языках, таких как английский или русский, предложения составляются из слов, с учетом определенных грамматических правил, диктующих порядок объединения этих слов. Аналогично в DSL единственная операция может состоять из нескольких вызовов функций, а проверка типов в компиляторе гарантирует их объединение в осмысленные конструкции. В результате функциям обычно даются имена-глаголы (`groupBy`, `orderBy`)[1], а их аргументы играют роль существительных (`Country.name`).

Одно из преимуществ наличия структуры в DSL – она позволяет использовать общий контекст в нескольких вызовах функций, не воссоздавая его заново в каждом вызове. Это иллюстрирует следующий пример, демонстрирующий описание зависимостей в сценариях сборки Gradle (https://github.com/gradle/gradle-script-kotlin):

```
dependencies {
    compile("junit:junit:4.11")
```

Структура формируется вложенными лямбда-выражениями

[1] Сгруппировать, упорядочить. – *Прим. ред.*

```
    compile("com.google.inject:guice:4.1.0")
}
```

Для сравнения ниже приводится реализация тех же операций с использованием обычного командного API. Обратите внимание, как много повторений возникает в этом коде:

```
project.dependencies.add("compile", "junit:junit:4.11")
project.dependencies.add("compile", "com.google.inject:guice:4.1.0")
```

Составление цепочек из вызовов методов – еще один способ создания структуры в DSL. Этот приём широко используется в фреймворках тестирования для разбития проверки на несколько вызовов методов. Такие проверки легче читаются, особенно если используется инфиксный синтаксис вызовов. Следующий пример взят из kotlintest (https://github.com/kotlintest/kotlintest), стороннего фреймворка тестирования для Kotlin, который мы обсудим в разделе 11.4.1:

```
str should startWith("kot")
```

Структура формируется цепочкой из вызовов методов

Обратите внимание, насколько сложнее воспринимается та же проверка на JUnit API, перегруженная излишними синтаксическими элементами:

```
assertTrue(str.startsWith("kot"))
```

Теперь рассмотрим пример внутреннего DSL более подробно.

11.1.4. Создание разметки HTML с помощью внутреннего DSL

Чтобы пробудить ваш интерес, в начале этой главы мы привели фрагмент кода на предметно-ориентированном языке, предназначенном для конструирования разметки HTML. В этом разделе мы обсудим этот язык более подробно. Программный интерфейс, описываемый здесь, реализует библиотека kotlinx.html (https://github.com/Kotlin/kotlinx.html). Вот маленький фрагмент, создающий таблицу с единственной ячейкой:

```
fun createSimpleTable() = createHTML().
    table {
        tr {
            td { +"cell" }
        }
    }
```

Очевидно, что предыдущая структура создает следующую HTML-таблицу:

```
<table>
  <tr>
    <td>cell</td>
```

```
  </tr>
</table>
```

Функция `createSimpleTable` возвращает строку с HTML-таблицей.

Почему предпочтительнее создавать разметку HTML в коде на Kotlin, а не хранить ее в отдельном файле? Во-первых, Kotlin-версия гарантирует безопасность типов: вы сможете использовать тег `td` только внутри тега `tr`; в противном случае код просто не скомпилируется. Но, что особенно важно, это самый обычный программный код, и в нём можно использовать любые конструкции языка. То есть такой подход позволяет создавать ячейки динамически (например, для отображения элементов словаря) в том же месте, где определена таблица:

```
fun createAnotherTable() = createHTML().table {
    val numbers = mapOf(1 to "one", 2 to "two")
    for ((num, string) in numbers) {
        tr {
            td { +"$num" }
            td { +string }
        }
    }
}
```

Сгенерированная в результате разметка HTML будет содержать желаемые данные:

```
<table>
  <tr>
    <td>1</td>
    <td>one</td>
  </tr>
  <tr>
    <td>2</td>
    <td>two</td>
  </tr>
</table>
```

HTML – канонический пример языка разметки, который прекрасно подходит для иллюстрации идеи. Но тот же подход можно использовать для любого языка с похожей структурой, такого как XML. Чуть ниже мы обсудим, как работает такой код в Kotlin.

Теперь, когда мы узнали, что такое предметно-ориентированный язык и зачем он может понадобиться, давайте посмотрим, как Kotlin помогает создавать такие языки. Для начала детальнее рассмотрим лямбда-выражения с получателями – ключевую особенность, помогающую построить грамматику DSL.

11.2. Создание структурированных API: лямбда-выражения с получателями в DSL

Лямбда-выражения с получателями – мощная особенность Kotlin, позволяющая конструировать API с определенной структурой. Как уже говорилось, наличие структуры – один из ключевых аспектов, отличающих предметно-ориентированные языки от обычных программных интерфейсов. Исследуем эту особенность детальнее и познакомимся с некоторыми использующими её предметно-ориентированными языками.

11.2.1. Лямбда-выражения с получателями и типы функций-расширений

Лямбда-выражения с получателями коротко рассматривались в разделе 5.5, где были представлены функции `buildString`, `with` и `apply` из стандартной библиотеки. Теперь на примере функции `buildString` нам предстоит узнать, как они реализованы. Эта функция позволяет конструировать строки из фрагментов, по очереди добавляемых в `StringBuilder`.

Для начала обсуждения определим функцию `buildString`, которая принимает аргумент с обычным лямбда-выражением. Похожий пример вы уже видели в главе 8, поэтому следующее определение не станет для вас новинкой.

Листинг 11.1. Определение buildString, которая принимает аргумент с лямбда-выражением

```
fun buildString(
        builderAction: (StringBuilder) -> Unit
): String {
    val sb = StringBuilder()
    builderAction(sb)
    return sb.toString()
}

>>> val s = buildString {
...     it.append("Hello, ")
...     it.append("World!")
... }
>>> println(s)
Hello, World!
```

Объявление параметра с типом функции

Передача StringBuilder лямбда-выражению, полученному в качестве аргумента

Ссылка «it» указывает на экземпляр StringBuilder

Этот код легко понять, но пользоваться такой функцией сложнее, чем хотелось бы. Обратите внимание, что мы вынуждены использовать `it` в теле лямбда-выражения для ссылки на экземпляр `StringBuilder` (мы могли бы определить свой, именованный параметр и использовать его вместо

it, но это не избавит нас от ненужной работы). Главная задача лямбда-выражения – заполнить StringBuilder текстом. Поэтому хотелось бы как-то избавиться от необходимости повторять ссылку it. и получить возможность вызывать методы StringBuilder непосредственно, заменив it.append простым вызовом append.

Для этого мы должны преобразовать лямбда-выражение в *лямбда-выражение с получателем*. Так мы сможем придать одному из параметров лямбда-выражения специальный статус *получателя* и ссылаться на его члены непосредственно, без всякого квалификатора. Следующий листинг показывает, как этого добиться.

Листинг 11.2. Определение buildString, которая принимает лямбда-выражение с получателем

```
fun buildString(
        builderAction: StringBuilder.() -> Unit
) : String {
    val sb = StringBuilder()
    sb.builderAction()
    return sb.toString()
}

>>> val s = buildString {
...     this.append("Hello, ")
...     append("World!")
... }
>>> println(s)
Hello, World!
```

Объявление параметра с типом функции с получателем

Передача StringBuilder лямбда-выражению в качестве получателя

Ключевое слово «this» ссылается на экземпляр StringBuilder

При желании «this» можно опустить и ссылаться на StringBuilder неявно

Обсудим различия между листингами 11.1 и 11.2. Прежде всего обратите внимание, насколько проще стало пользоваться функцией buildString. Теперь мы можем передать лямбда-выражение с получателем и избавиться от ссылки it в его теле. Мы заменили вызов it.append() на append(). Полная форма вызова теперь имеет вид this.append(), но, как и в случае с обычными членами класса, явная ссылка this, как правило, нужна только для устранения неоднозначности.

Теперь рассмотрим изменения в объявлении функции buildString. Для параметра мы указали *тип функции-расширения* вместо обычного типа функции. Чтобы объявить, что параметр имеет тип функции-расширения, мы должны вынести один из параметров этого типа функции за круглые скобки и расположить его перед ними, отделив точкой от остальной части объявления типа. В листинге 11.2 мы заменили (StringBuilder) -> Unit на StringBuilder.() -> Unit. Этот специальный тип называется *типом получателя* (receiver type), а значение этого типа, что передается в

лямбда-выражение, – *объектом-получателем* (receiver object). На рис. 11.1 показано более сложное объявление типа функции-расширения.

Рис. 11.1. Тип функции-расширения с получателем типа String, двумя параметрами типа Int и возвращаемым значением типа Unit

Почему именно тип *функции-расширения*? Идея доступа к членам внешнего типа без явного использования квалификатора напоминает поддержку функций-расширений, которая позволяет определять свои методы для классов, объявленных где-то ещё. Функции-расширения и лямбда-выражения с получателем получают *объект-получатель*, который должен быть передан в вызов функции и доступен в её теле. Фактически тип функции-расширения описывает блок кода, который можно вызвать как функцию-расширение.

Способ вызова лямбда-выражения также изменился после того, как мы заменили обычный тип функции типом функции-расширения. Теперь лямбда-выражение вызывается не как обычная функция, получающая объект в аргументе, а как функция-расширение. Обычное лямбда-выражение получало экземпляр `StringBuilder` в аргументе, и мы вызывали его, используя синтаксис `builderAction(sb)`. При использовании лямбда-выражения с получателем его вызов приобрел вид `sb.builderAction()`. Повторим ещё раз: `builderAction` не является методом, объявленным в классе `StringBuilder`, – это параметр, имеющий тип функции, который вызывается с применением того же синтаксиса, что используется для функций-расширений.

На рис. 11.2 показано соответствие между аргументом и параметром функции `buildString`. Он также иллюстрирует объект-получатель, для которого вызывается тело лямбда-выражения.

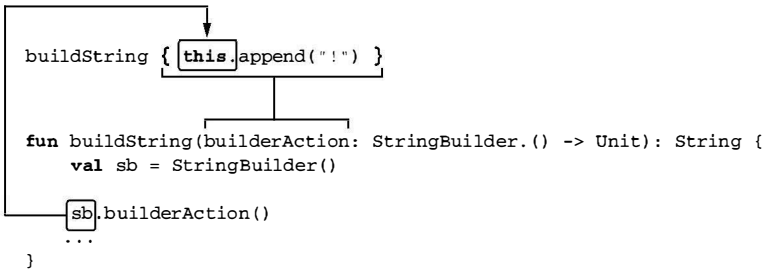


Рис. 11.2. Аргумент функции buildString (лямбда-выражение с получателем) соответствует параметру типа функции-расширения (builderAction); когда вызывается тело лямбда-выражения, получатель (sb) превращается в неявный получатель (this)

Также можно объявить переменную с типом функции-расширения, как показано в листинге 11.3. В результате появится возможность вызывать её как функцию-расширение или передавать как аргумент в вызов функции, принимающей лямбда-выражение с получателем.

Листинг 11.3. Сохранение лямбда-выражения с получателем в переменной

```
val appendExcl : StringBuilder.() -> Unit =
        { this.append("!") }

>>> val stringBuilder = StringBuilder("Hi")
>>> stringBuilder.appendExcl()
>>> println(stringBuilder)
Hi!

>>> println(buildString(appendExcl))
!
```

Обратите внимание, что в исходном коде лямбда-выражение с получателем выглядит в точности как обычное лямбда-выражение. Чтобы увидеть, есть ли у лямбда-выражения получатель, нужно рассмотреть функцию, принимающую лямбда-выражение: её сигнатура сообщит, есть ли получатель у лямбда-выражения и какого типа. Например, можно посмотреть на объявление функции `buildString` или заглянуть в документацию в своей IDE, увидеть, что она принимает лямбда-выражение типа `StringBuilder.() -> Unit`, и сделать вывод, что в теле лямбда-выражения можно вызывать методы `StringBuilder` без квалификатора.

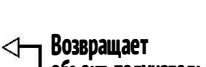
Реализация `buildString` в стандартной библиотеке короче, чем в листинге 11.2. Вместо явного вызова `builderAction` она передает аргумент функции `apply` (с которой мы познакомились в разделе 5.5). Это позволяет сократить определение функции до одной строки:

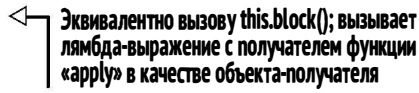
```
fun buildString(builderAction: StringBuilder.() -> Unit): String =
    StringBuilder().apply(builderAction).toString()
```

Функция `apply` фактически принимает объект, для которого произведен вызов (в данном случае новый экземпляр `StringBuilder`), и использует его в качестве неявного получателя для вызова функции или лямбда-выражения, переданного в аргументе (`builderAction` в данном примере).

В разделе 5.5 мы познакомились с ещё одной интересной функцией из стандартной библиотеки: `with`. Давайте исследуем их реализации:

```
inline fun <T> T.apply(block: T.() -> Unit): T {
    block()
    return this
}
```





```
}

inline fun <T, R> with(receiver: T, block: T.() -> R): R =
    receiver.block()
```

Возвращает результат вызова лямбда-выражения

Попросту говоря, функции `apply` и `with` просто вызывают аргумент с типом функции-расширения на заданном получателе. Функция `apply` объявлена как расширение типа получателя, тогда как `with` принимает получателя в первом аргументе. Кроме того, `apply` возвращает сам объект-получатель, а `with` возвращает результат вызова лямбда-выражения.

Если результат не имеет значения, эти функции можно считать взаимозаменяемыми:

```
>>> val map = mutableMapOf(1 to "one")
>>> map.apply { this[2] = "two"}
>>> with (map) { this[3] = "three" }
>>> println(map)
{1=one, 2=two, 3=three}
```

Функции `with` и `apply` широко используются в Kotlin, и мы надеемся, что вы уже оценили их удобство в вашем собственном коде.

Мы ещё раз подробно рассмотрели лямбда-выражения с получателем и поговорили о типах функций-расширений. Теперь посмотрим, как они используются в контексте предметно-ориентированных языков.

11.2.2. Использование лямбда-выражений с получателями в построителях разметки HTML

Kotlin DSL для HTML, или «предметно-ориентированный язык для создания разметки HTML», обычно называют *построителем HTML*, и он представляет более общее понятие типизированных построителей. Впервые идея построителей завоевала большую популярность в сообществе Groovy (www.groovy-lang.org/dsls.html#_builders). Построители дают возможность создавать иерархии объектов декларативным способом, который хорошо подходит для создания XML или размещения компонентов пользовательского интерфейса.

В Kotlin используется та же идея, но построители на Kotlin типизированные. Это делает их более удобными в использовании, безопасными и в некотором смысле более привлекательными, чем динамические построители в Groovy. Давайте посмотрим, как построители HTML работают в Kotlin.

Листинг 11.4. Создание простой HTML-таблицы с помощью построителя на Kotlin

```
fun createSimpleTable() = createHTML().
    table {
        tr {
```

```
            td { +"cell" }
    }
    }
```

Это обычный код на языке Kotlin, а не специальный язык шаблонов или что-то подобное: `table`, `tr` и `td` – это всего лишь функции. Все они – функции высшего порядка и принимают лямбда-выражения с получателями.

Важно отметить, что лямбда-выражения *изменяют правила разрешения имен*. В лямбда-выражении, переданном в функцию `table`, можно использовать функцию `tr`, чтобы создать HTML-тег `<tr>`. За пределами этого лямбда-выражения функция `tr` будет недоступна. Аналогично функция `td` доступна только внутри `tr`. (Обратите внимание, как дизайн API вынуждает следовать правилам грамматики языка HTML.)

Контекст разрешения имен в каждом блоке определяется типом получателя каждого лямбда-выражения. Лямбда-выражение, передаваемое в вызов `table`, получает получатель специального типа `TABLE`, который определяет метод `tr`. Аналогично функция `tr` ожидает лямбда-расширения типа `TR`. Следующий листинг – значительно упрощенное представление объявлений этих классов и методов.

Листинг 11.5. Определение классов тегов для построителя разметки HTML

```
open class Tag

class TABLE : Tag {
    fun tr(init : TR.() -> Unit)
}

class TR : Tag {
    fun td(init : TD.() -> Unit)
}

class TD : Tag
```

Функция tr принимает лямбда-выражение с приемником типа TR

Функция td принимает лямбда-выражение с приемником типа TD

`TABLE`, `TR` и `TD` – это вспомогательные классы, которые не должны явно появляться в коде, и именно поэтому их имена состоят только из заглавных букв. Все они наследуют суперкласс `Tag`. Каждый класс определяет методы для создания допустимых тегов: класс `TABLE`, кроме других, определяет метод `tr`, а класс `TR` определяет метод `td`.

Обратите внимание на типы параметров `init` функций `tr` и `td`: это функции-расширения `TR.() -> Unit` и `TD.() -> Unit`. Они определяют типы получателей в лямбда-выражениях: `TR` и `TD` соответственно.

Чтобы было понятнее, мы можем переписать листинг 11.4, сделав все получатели более явными. Напомним, что обратиться к получателю лямбда-выражения, которое передается функции `foo`, можно как `this@foo`.

Листинг 11.6. Явное использование получателей в вызовах построителя HTML

```
fun createSimpleTable() = createHTML().
    table {                                  this@table имеет
        (this@table).tr {                    тип TABLE
            (this@tr).td {                                  this@tr имеет
                +"cell"                                     тип TR
            }                    Здесь доступен неявный
        }                        получатель this@td типа TD
    }
```

Если вместе с построителем попытаться использовать обычное лямбда-выражение вместо лямбда-выражения с получателем, синтаксис может стать нечитаемым, как в примере выше. Вам пришлось бы использовать ссылку `it` для вызова методов создания тегов или присваивать имя параметру в каждом лямбда-выражении. Возможность использовать неявный получатель и скрыть ссылку `this` делает синтаксис построителей компактнее и более похожим на оригинальную разметку HTML.

Обратите внимание, что если лямбда-выражение с получателем поместить в другое лямбда-выражение (как в листинге 11.6), то получатель внешнего лямбда-выражения останется доступным во вложенных лямбда-выражениях. Например, в лямбда-выражении, которое передается как аргумент функции `td`, доступны все три получателя (`this@table`, `this@tr`, `this@td`). Но в версии Kotlin 1.1 появится возможность использовать аннотацию `@DslMarker` для ограничения доступности внешних получателей.

Итак, мы знаем, как синтаксис построителей HTML зависит от лямбда-выражений с получателями. Теперь обсудим, как генерируется желаемая разметка.

В листинге 11.6 использована функция, объявленная в библиотеке kotlinx.html. Далее мы реализуем более простую версию библиотеки построителя HTML: для этого расширим объявления тегов `TABLE`, `TR` и `TD` и добавим поддержку для создания окончательной разметки. Точка входа для этой упрощенной версии создает HTML-тег `<table>` вызовом функции `table`.

Листинг 11.7. Создание разметки HTML в виде строки

```
fun createTable() =
    table {
        tr {
            td {
        }
    }
```

```
}

>>> println(createTable())
<table><tr><td></td></tr></table>
```

Функция `table` создает новый экземпляр тега `TABLE`, инициализирует (вызывая функцию, переданную в параметре `init`) и возвращает его:

```
fun table(init: TABLE.() -> Unit) = TABLE().apply(init)
```

Внутри `createTable` в функцию `table` передается лямбда-выражение, содержащее вызов функции `tr`. Вот как можно переписать этот вызов, чтобы неявное сделать явным: `table(init = { this.tr { ... } })`. Функция `tr` будет вызвана как метод вновь созданного экземпляра `TABLE` – как если бы мы записали его так: `TABLE().tr { ... }`.

В этом коротком примере `<table>` – это тег верхнего уровня, а остальные вложены в него. Каждый тег хранит список ссылок на своих потомков. Поэтому функция `tr` должна не только инициализировать новый экземпляр тега `TR`, но также добавить его в список потомков внешнего тега.

Листинг 11.8. Определение функции создания тега

```
fun tr(init: TR.() -> Unit) {
    val tr = TR()
    tr.init()
    children.add(tr)
}
```

Такая логика инициализации тега и добавления его в список потомков внешнего тега характерна для всех тегов, поэтому её можно выделить в отдельную функцию-член `doInit` суперкласса `Tag`. Функция `doInit` отвечает за две операции: сохранение ссылки на дочерний тег и вызов лямбда-выражения, полученного в аргументе. Другие теги будут просто вызывать её: например, функция `tr` создает новый экземпляр класса `TR` и затем передает его функции `doInit` вместе с лямбда-выражением в аргументе `init`: `doInit(TR(), init)`. В листинге 11.9 приводится полный пример, демонстрирующий создание желаемой разметки HTML.

Листинг 11.9. Полная реализация простого построителя HTML

```
open class Tag(val name: String) {
    private val children = mutableListOf<Tag>()   <- Хранит все вложенные теги

    protected fun <T : Tag> doInit(child: T, init: T.() -> Unit) {
        child.init()        <- Инициализация дочернего тега
```

```
        children.add(child)
    }

    override fun toString() =
        "<$name>${children.joinToString("")}</$name>"
}

fun table(init: TABLE.() -> Unit) = TABLE().apply(init)

class TABLE : Tag("table") {
    fun tr(init: TR.() -> Unit) = doInit(TR(), init)
}

class TR : Tag("tr") {
    fun td(init: TD.() -> Unit) = doInit(TD(), init)
}
class TD : Tag("td")

fun createTable() =
    table {
        tr {
            td {
        }
    }
}

>>> println(createTable())
<table><tr><td></td></tr></table>
```

Сохранит ссылку на дочерний тег

Возврат получившейся разметки в виде строки

Создает и инициализирует тег TR и добавляет его в список потомков TABLE

Добавляет новый экземпляр TD в список потомков TR

Каждый тег хранит список вложенных тегов и отображает себя соответственно: сначала свое имя и затем рекурсивно все вложенные теги. Текст внутри тегов и атрибуты тегов в данной реализации не поддерживаются. Реализацию с полными возможностями вы найдете в исходных текстах вышеупомянутой библиотеки kotlinx.html.

Обратите внимание, что функции создания тегов добавляют соответствующий тег в список потомков своего родительского тега. Это дает возможность генерировать теги динамически.

Листинг 11.10. Динамическое создание тегов с помощью построителя HTML

```
fun createAnotherTable() = table {
    for (i in 1..2) {
        tr {
            td {
            }
        }
```

Каждый вызов «tr» создает новый тег TR и добавляет его в список потомков TABLE

```
    }
}

>>> println(createAnotherTable())
<table><tr><td></td></tr><tr><td></td></tr></table>
```

Как видите, лямбда-выражения с получателями – отличный инструмент для сознания предметно-ориентированных языков. Благодаря возможности изменения контекста разрешения имен в блоке кода они позволяют определять структурированные API – это характерная черта, отличающая DSL от простых последовательностей вызовов методов. Теперь давайте обсудим преимущества интеграции такого DSL в статически типизированный язык программирования.

11.2.3. Построители на Kotlin: поддержка абстракций и многократного использования

Разрабатывая программный код, мы можем пользоваться множеством инструментов, чтобы избежать дублирования и сделать код более выразительным и читаемым. Например, можно оформить повторяющийся код в виде новых функций и дать им говорящие имена. Это не так просто (если вообще возможно) проделать с SQL или HTML. Но внутренние предметно-ориентированные языки в Kotlin, используемые для решения аналогичных задач, позволяют выделять повторяющиеся фрагменты в новые функции и повторно использовать их.

Рассмотрим пример из библиотеки Bootstrap (http://getbootstrap.com), популярного фреймворка для разработки HTML, CSS и JS в динамичных веб-приложениях. Возьмем за основу конкретный пример: добавление раскрывающихся списков в приложение. Чтобы добавить такой список непосредственно в страницу HTML, нужно скопировать необходимый фрагмент и вставить его в нужное место, связав его с кнопкой или другим элементом, отображающим список. От нас требуется только добавить в раскрывающееся меню необходимые ссылки и их заголовки. В листинге 1.11 приводится начальный HTML-код (сильно упрощенный, чтобы не загромождать разметку атрибутами `style`).

Листинг 11.11. Создание раскрывающегося меню средствами Bootstrap

```
<div class="dropdown">
  <button class="btn dropdown-toggle">
    Dropdown
    <span class="caret"></span>
  </button>
  <ul class="dropdown-menu">
    <li><a href="#">Action</a></li>
```

```
    <li><a href="#">Another action</a></li>
    <li role="separator" class="divider"></li>
    <li class="dropdown-header">Header</li>
    <li><a href="#">Separated link</a></li>
  </ul>
</div>
```

Чтобы воспроизвести аналогичную структуру в Kotlin, можно воспользоваться библиотекой kotlinx.html и её функциями `div`, `button`, `ul`, `li` и другими.

Листинг 11.12. Создание раскрывающегося меню средствами построителя HTML в Kotlin

```
fun buildDropdown() = createHTML().div(classes = "dropdown") {
    button(classes = "btn dropdown-toggle") {
        +"Dropdown"
        span(classes = "caret")
    }
    ul(classes = "dropdown-menu") {
        li { a("#") { +"Action" } }
        li { a("#") { +"Another action" } }
        li { role = "separator"; classes = setOf("divider") }
        li { classes = setOf("dropdown-header"); +"Header" }
        li { a("#") { +"Separated link" } }
    }
}
```

Но код можно сделать более компактным. Так как `div`, `button` и прочие – это обычные функции, мы можем вынести повторяющуюся логику в отдельные функции и сделать код более удобочитаемым. Результат может выглядеть так, как в листинге 11.13.

Листинг 11.13. Создание раскрывающегося меню с помощью вспомогательных функций

```
fun dropdownExample() = createHTML().dropdown {
    dropdownButton { +"Dropdown" }
    dropdownMenu {
        item("#", "Action")
        item("#", "Another action")
        divider()
        dropdownHeader("Header")
        item("#", "Separated link")
    }
}
```

Теперь ненужные детали скрыты, код стал более ясным. Давайте посмотрим, как удалось добиться такого результата. Начнем с функции `item`. У неё два параметра: ссылка и имя соответствующего пункта меню. Код функции должен добавить новый элемент списка: `li { a(href) { +name } }`. Единственная неясность – как вызвать `li` в теле функции? Следует ли объявить эту функцию расширением? Да, фактически мы можем сделать её функцией-расширением для класса `UL`, потому что `li` сама является функцией-расширением для этого класса. В листинге 11.13 функция `item` неявно вызывается как метод `this` типа `UL`:

```
fun UL.item(href: String, name: String) = li { a(href) { +name } }
```

Вновь объявленную функцию `item` можно использовать в любом теге `UL`, и каждый её вызов будет добавлять экземпляр тега `LI`. Выделив функцию `item`, мы можем изменить исходную версию так, как показано в листинге 11.14, не влияя на сгенерированную разметку HTML.

Листинг 11.14. Использование функции `item` для конструирования раскрывающегося меню

```
ul {
    classes = setOf("dropdown-menu")
    item("#", "Action")                                    <-- Теперь вместо «li» можно использовать функцию «item»
    item("#", "Another action")
    li { role = "separator"; classes = setOf("divider") }
    li { classes = setOf("dropdown-header"); +"Header" }
    item("#", "Separated link")
}
```

Аналогично добавляются другие функции-расширения для `UL`, позволяющие заменить остальные теги `li`.

```
fun UL.divider() = li { role = "separator"; classes = setOf("divider") }

fun UL.dropdownHeader(text: String) =
    li { classes = setOf("dropdown-header"); +text }
```

Теперь посмотрим, как реализована функция `dropdownMenu`. Она создает тег `ul` с заданным классом[2] `dropdown-menu` и принимает лямбда-выражение с получателем, которое заполняет тег содержимым.

```
dropdownMenu {
    item("#", "Action")
    ...
}
```

[2] Здесь имеется в виду класс CSS. – *Прим. ред.*

Мы заменили блок `ul { ... }` вызовом `dropdownMenu { ... }`, поэтому получатель в лямбда-выражении останется прежним. Функция `dropdownMenu` принимает лямбда-выражения как расширения для UL, что позволяет вам вызывать такие функции, как `UL.item`, как вы делали раньше. Вот как объявлена эта функция:

```
fun DIV.dropdownMenu(block: UL.() -> Unit) = ul("dropdown-menu", block)
```

Функция `dropdownButton` реализована аналогично. Мы опустим её здесь, но вы можете найти полную реализацию в примерах, поставляемых в составе библиотеки kotlinx.html.

Наконец, рассмотрим функцию `dropdown`. Это одна из самых нетривиальных функций в нашем примере, потому что может вызываться из любого тега: раскрывающееся меню можно поместить куда угодно.

Листинг 11.15. Функция верхнего уровня для создания раскрывающихся меню

```
fun StringBuilder.dropdown(
        block: DIV.() -> Unit
): String = div("dropdown", block)
```

Эту упрощенную версию можно использовать, чтобы просто вывести разметку HTML в строку. Полная реализация в kotlinx.html использует абстрактный класс `TagConsumer` как получатель, благодаря чему поддерживаются разные варианты для вывода HTML.

Данный пример иллюстрирует, как обычные приёмы абстрагирования и повторного использования могут улучшить код и сделать его проще и понятнее.

А теперь познакомимся с ещё одним инструментом, помогающим создавать гибкие структуры в предметно-ориентированных языках: соглашение `invoke`.

11.3. Гибкое вложение блоков с использованием соглашения «invoke»

Соглашение `invoke` позволяет вызывать объекты как функции. Вы уже видели, что объекты с типами функций можно вызывать как функции. Но благодаря соглашению `invoke` мы можем определять собственные объекты, поддерживающие тот же синтаксис.

Обратите внимание, что эта особенность не предназначена для повсеместного использования, потому что она позволяет писать трудные для понимания выражения, например `1()`. Но иногда она пригождается в предметно-ориентированных языках. Далее покажем, где это может применяться, но сначала обсудим само соглашение.

11.3.1. Соглашение «invoke»: объекты, вызываемые как функции

В главе 7 мы детально обсудили идею *соглашений* в Kotlin: в частности, именованные функции, которые вызываются не как обычно, а с использованием другого, более компактного синтаксиса. Например, в той главе мы обсудили соглашение `get`, позволяющее обращаться к объектам с использованием оператора индексирования. Обращение `foo[bar]` к переменной `foo` типа `Foo` транслируется в вызов `foo.get(bar)`, если соответствующая функция `get` определена как член класса `Foo` или как функция-расширение для `Foo`.

Соглашение `invoke` фактически делает то же самое – с той лишь разницей, что квадратные скобки заменяются круглыми. Если класс определяет функцию `invoke` с модификатором `operator`, его экземпляры можно вызывать как функции, как показано в листинге 11.16.

Листинг 11.16. Определение метода `invoke` в классе

```
class Greeter(val greeting: String) {
    operator fun invoke(name: String) {
        println("$greeting, $name!")
    }
}

>>> val bavarianGreeter = Greeter("Servus")
>>> bavarianGreeter("Dmitry")
Servus, Dmitry!
```

Определение метода «invoke» в классе Greeter

Вызов экземпляра Greeter как функции

Здесь в классе `Greater` определяется метод `invoke`. Это позволяет вызывать экземпляры `Greeter` как обычные функции. За кулисами выражение `bavarianGreeter("Dmitry")` компилируется в вызов `bavarianGreeter.invoke("Dmitry")`. Здесь нет никакой мистики – это самое обычное соглашение, помогающее писать более компактные и ясные выражения.

Метод `invoke` не ограничивается какой-то одной конкретной сигнатурой. Его можно объявить с любым количеством параметров и с любым типом возвращаемого значения и даже определить перегруженные версии `invoke` с разными типами параметров. Эти сигнатуры можно использовать для вызова экземпляров класса как функций. Рассмотрим пример, когда это соглашение может пригодиться на практике: сначала в контексте обычного программирования, а затем в DSL.

11.3.2. Соглашение «invoke» и типы функций

Многие из вас наверняка помнят, что мы уже использовали соглашение `invoke` ранее в этой книге. В разделе 8.1.2 мы обсуждали возможность вы-

зова переменной с типом функции, поддерживающим значение `null`, как `lambda?.invoke()`, использовав безопасный синтаксис вызова с именем метода `invoke`.

Теперь, после знакомства с соглашением `invoke`, вам должно быть понятно, что вызов лямбда-выражения (с добавлением круглых скобок после него: `lambda()`) – это не что иное, как применение данного соглашения. Лямбда-выражения, кроме встраиваемых, компилируются в классы, реализующие интерфейсы функций (`Function1` и другие), а эти интерфейсы определяют метод `invoke` с соответствующим количеством параметров:

```
interface Function2<in P1, in P2, out R> {
    operator fun invoke(p1: P1, p2: P2): R
}
```

Этот интерфейс обозначает функцию, принимающую точно два аргумента

Когда мы вызываем лямбда-выражение как функцию, данная операция в соответствии с соглашением транслируется в вызов метода `invoke`. Где может пригодиться это знание? Оно поможет разбить код сложных лямбда-выражений на несколько методов, сохранив возможность их использования с функциями, принимающими параметры с типами функций. Для этого можно определить класс, реализующий интерфейс типа функции. В качестве базового можно явно использовать один из типов `FunctionN` или, как показано в листинге 11.17, применить сокращенный синтаксис: `(P1, P2) -> R`. В этом примере такой класс используется для фильтрации списка проблем с применением сложных условий.

Листинг 11.17. Наследование типа функции и переопределение метода `invoke()`

```
data class Issue(
    val id: String, val project: String, val type: String,
    val priority: String, val description: String
)

class ImportantIssuesPredicate(val project: String)
        : (Issue) -> Boolean {

    override fun invoke(issue: Issue): Boolean {
        return issue.project == project && issue.isImportant()
    private fun Issue.isImportant(): Boolean {
        return type == "Bug" &&
            (priority == "Major" || priority == "Critical")
    }
}

>>> val i1 = Issue("IDEA-154446", "IDEA", "Bug", "Major",
...                "Save settings failed")
>>> val i2 = Issue("KT-12183", "Kotlin", "Feature", "Normal",
```

В качестве базового класса используется тип функции

Реализация метода «invoke»

```
... "Intention: convert several calls on the same receiver to with/apply")
>>> val predicate = ImportantIssuesPredicate("IDEA")
>>> for (issue in listOf(i1, i2).filter(predicate)) {      <┐ Передает предикат
...     println(issue.id)                                   | в вызов filter()
... }
IDEA-154446
```

Здесь у предиката слишком сложная логика, чтобы уместить её в единственном лямбда-выражении. Поэтому мы разбили её на несколько методов, чтобы отчетливее осмыслить каждый из них. Преобразование лямбда-выражения в класс, реализующий интерфейс типа функции и переопределяющий метод `invoke`, – это один из способов выполнить такой рефакторинг. Преимущество этого подхода заключается в том, что область видимости методов, выделяемых из тела лямбда-выражения, сужена до минимума; они доступны только из класса предиката. Это ценно, когда в классе предиката и окружающем коде есть много логики, и её стоит четко разделить по зонам ответственности.

Теперь посмотрим, как соглашение `invoke` помогает создавать более гибкие структуры в предметно-ориентированных языках.

11.3.3. Соглашение «invoke» в предметно-ориентированных языках: объявление зависимостей в Gradle

Вернемся к примеру Gradle DSL для настройки зависимостей модуля. Вот код, который демонстрировался выше:

```
dependencies {
    compile("junit:junit:4.11")
}
```

Часто бывает желательно, чтобы в одном API поддерживались и вложенные блоки, как в данном примере, и простые последовательности вызовов. Иными словами, хотелось бы иметь возможность использовать любую из двух форм записи:

```
dependencies.compile("junit:junit:4.11")

dependencies {
    compile("junit:junit:4.11")
}
```

При такой организации пользователи DSL смогут использовать вложенные блоки, когда потребуется определить несколько зависимостей, и простые вызовы для единственных зависимостей, что позволит сделать код более компактным.

В первом случае вызывается метод `compile` переменной `dependencies`. Вторую форму записи можно получить, определив метод `invoke` в `depen-`

dencies, который принимает аргумент с лямбда-выражением. Полный синтаксис этого вызова: dependencies.invoke({...}).

Объект dependencies – это экземпляр класса DependencyHandler, который определяет оба метода, compile и invoke. Метод invoke принимает лямбда-выражение с получателем как аргумент, и тип получателя этого метода – снова DependencyHandler. Происходящее в теле лямбда-выражения вам уже знакомо: имея получатель типа DependencyHandler, оно может вызвать метод вроде compile напрямую. Следующий небольшой пример в листинге 11.18 демонстрирует, как осуществляется эта часть DependencyHandler.

Листинг 11.18. Использование invoke для поддержки гибкого синтаксиса DSL

```
class DependencyHandler {
    fun compile(coordinate: String) {
        println("Added dependency on $coordinate")
    }

    operator fun invoke(
            body: DependencyHandler.() -> Unit) {
        body()
    }
}
```

Определяется обычный командный API

Определяется «invoke» для поддержки DSL API

«this» ссылается на функцию body: this.body()

```
>>> val dependencies = DependencyHandler()

>>> dependencies.compile("org.jetbrains.kotlin:kotlin-stdlib:1.0.0")
Added dependency on org.jetbrains.kotlin:kotlin-stdlib:1.0.0

>>> dependencies {
...     compile("org.jetbrains.kotlin:kotlin-reflect:1.0.0")
>>> }
Added dependency on org.jetbrains.kotlin:kotlin-reflect:1.0.0
```

Добавляя первую зависимость, мы вызвали метод compile непосредственно. Второй вызов фактически транслируется в:

```
dependencies.invoke({
    this.compile("org.jetbrains.kotlin:kotlin-reflect:1.0.0")
})
```

Иными словами, здесь мы вызываем dependencies как функцию и передаем лямбду как аргумент. Параметр лямбда-выражения имеет тип функции с получателем, а тип получателя – тот же тип DependencyHandler. Метод invoke вызывает лямбда-выражение. Поскольку этот метод принадлежит

классу `DependencyHandler`, то экземпляр этого класса доступен как неявный получатель, и нам не требуется явно указывать его, вызывая `body()`.

Переопределение метода `invoke` – это всего лишь один маленький фрагмент кода, но он существенно повысил гибкость DSL API. Это универсальный шаблон, и его можно использовать в самых разных DSL почти без модификаций.

Теперь вы знакомы с двумя новыми особенностями Kotlin, помогающими создавать свои предметно-ориентированные языки: лямбда-выражения с получателем и соглашение `invoke`. Давайте посмотрим, как эти и другие особенности Kotlin работают в контексте DSL.

11.4. Предметно-ориентированные языки Kotlin на практике

Теперь вы знакомы со всеми особенностями Kotlin, используемыми для создания предметно-ориентированных языков. Некоторые, такие как расширения и инфиксный вызов, должны уже стать для вас старыми добрыми друзьями. Другие, как лямбда-выражения с получателями, впервые подробно обсуждались в этой главе. Давайте соберем все эти знания вместе и исследуем несколько практических примеров создания DSL. Мы охватим самые разные темы: тестирование, поддержка литералов дат, запросы к базам данных и конструирование пользовательского интерфейса для Android.

11.4.1. Цепочки инфиксных вызовов: «should» в фреймворках тестирования

Как упоминалось выше, ясный синтаксис – одна из отличительных черт внутренних DSL, и его можно достичь за счет уменьшения количества знаков препинания в коде. Большинство внутренних DSL сводится к выполнению последовательностей вызовов методов, поэтому всё, что способствует уменьшению синтаксического шума в вызовах методов, широко используется в этой области. В число таких особенностей Kotlin входят сокращенный синтаксис вызова лямбда-выражений (который мы уже детально обсудили) и *инфиксный вызов функций*. Синтаксис инфиксных вызовов упоминался в разделе 3.4.3, а здесь мы сосредоточимся на его применении в DSL.

Рассмотрим пример использования предметно-ориентированного языка kotlintest (`https://github.com/kotlintest/kotlintest`, библиотека тестирования, разработанная по примеру Scalatest), который мы уже видели в этой главе.

Листинг 11.19. Запись проверки на предметно-ориентированном языке kotlintest

```
s should startWith("kot")
```

Этот тестовый вызов потерпит неудачу, если значение переменной `s` не будет начинаться с последовательности символов «kot». Программный код читается практически как обычное предложение на английском языке: «The `s` string should start with this constant» (Строка `s` должна начинаться с этой константы). Чтобы добиться такой выразительности, мы должны объявить функцию `should` с модификатором `infix`.

Листинг 11.20. Реализация функции `should`

```
infix fun <T> T.should(matcher: Matcher<T>) = matcher.test(this)
```

Функция `should` ожидает получить экземпляр `Matcher`, обобщенного интерфейса для проверки значений. `startWith` реализует интерфейс `Matcher` и проверяет, начинается ли строка с указанной подстроки.

Листинг 11.21. Определение интерфейса `Matcher` в kotlintest DSL

```
interface Matcher<T> {
    fun test(value: T)
}

class startWith(val prefix: String) : Matcher<String> {
    override fun test(value: String) {
        if (!value.startsWith(prefix))
            throw AssertionError("String $value does not start with $prefix")
    }
}
```

Обратите внимание, что в обычном коде рекомендуется начинать имена классов с большой буквы (то есть имя `startWith` следовало бы записать как `StartWith`), но предметно-ориентированные языки часто отступают от этого правила. Листинг 11.19 демонстрирует, как применение инфиксной формы записи вызовов в контексте DSL помогает уменьшить синтаксический шум в коде. Проявив немного изобретательности, этот шум можно уменьшить ещё больше – например, kotlintest DSL поддерживает такую форму записи.

Листинг 11.22. Составление цепочки вызовов в kotlintest DSL

```
"kotlin" should start with "kot"
```

На первый взгляд этот код не имеет ничего общего с языком Kotlin. Чтобы понять, как такое возможно, преобразуем инфиксные вызовы в обычные.

```
"kotlin".should(start).with("kot")
```

Как видите, в действительности код в листинге 11.22 – это последовательность из двух инфиксных вызовов, а `start` – аргумент первого из них. Фактически `start` ссылается на объявление объекта, а `should` и `with` – функции, вызываемые с применением инфиксной нотации.

Функция `should` представлена в перегруженной версии, которая использует объект `start` как тип параметра и возвращает промежуточную обертку, обладающую методом `with`.

Листинг 11.23. Определение API для поддержки цепочек инфиксных вызовов

```
object start

infix fun String.should(x: start): StartWrapper = StartWrapper(this)

class StartWrapper(val value: String) {
    infix fun with(prefix: String) =
        if (!value.startsWith(prefix))
            throw AssertionError(
                "String does not start with $prefix: $value")
}
```

Обратите внимание, что вне контекста DSL редко имеет смысл использовать объект в роли типа параметра, потому что он имеется в единственном экземпляре и к нему проще обратиться напрямую, чем передавать в аргументе. Но в данном случае такой приём оправдан: объект используется не для передачи данных в функцию, а как часть грамматики DSL. Передавая `start` в аргументе, мы можем выбрать правильную перегруженную версию `should` и получить экземпляр `StartWrapper` в результате. У класса `StartWrapper` есть функция-член `with`, принимающая аргумент с фактическим значением, необходимым для проверки.

Библиотека поддерживает и другие средства сопоставления, и всё применение в коде читается как обычное предложение на английском языке:

```
"kotlin" should end with "in"
"kotlin" should have substring "otl"
```

С этой целью для функции `should` определено несколько перегруженных версий, которые принимают экземпляры объектов `end` и `have` и возвращают экземпляры `EndWrapper` и `HaveWrapper` соответственно.

Это довольно необычный пример предметно-ориентированного языка, но настолько впечатляющий, что мы должны были показать, как он

работает. Комбинирование инфиксных вызовов и экземпляров объектов позволяет конструировать весьма сложные грамматики и использовать ясный и выразительный синтаксис. И конечно, DSL остается полностью типизированным. Неправильное сочетание функций и объектов просто не будет компилироваться.

11.4.2. Определение расширений для простых типов: обработка дат

Теперь рассмотрим ещё один пример, обещанный в начале главы:

```
val yesterday = 1.days.ago
val tomorrow = 1.days.fromNow
```

Для реализации этого DSL с использованием `java.time` из Java 8 и Kotlin достаточно написать всего несколько строк кода. Вот часть реализации, имеющая отношение к примеру.

Листинг 11.24. Определение предметно-ориентированного языка для работы с датами

```
val Int.days: Period
    get() = Period.ofDays(this)

val Period.ago: LocalDate
    get() = LocalDate.now() - this

val Period.fromNow: LocalDate
    get() = LocalDate.now() + this

>>> println(1.days.ago)
2016-08-16
>>> println(1.days.fromNow)
2016-08-18
```

«this» ссылается на значение числовой константы

Вызов LocalDate.minus с использованием синтаксиса операторов

Вызов LocalDate.plus с использованием синтаксиса операторов

Здесь `days` – это свойство-расширение для типа `Int`. Kotlin не ограничивает типов, которые можно использовать в качестве получателей для функций-расширений: мы легко можем определить желаемые расширения для простых типов и вызывать их относительно констант. Свойство `days` возвращает значение типа `Period`, который в JDK 8 представляет интервал между двумя датами.

Чтобы завершить предложение и поддержать слово `ago`, нужно определить ещё одно свойство-расширение – на этот раз для класса `Period`. Тип этого свойства – `LocalDate`, и оно представляет дату. Обратите внимание, что оператор `-` (минус) в реализации свойства `ago` не опирается ни на какие расширения в Kotlin. JDK-класс `LocalDate` определяет метод с име-

нем `minus` и единственным параметром, который соответствует соглашению об операторе – (минус) в языке Kotlin, поэтому Kotlin автоматически отображает этот оператор в вызов метода. Полную реализацию библиотеки, поддерживающей дни и другие единицы измерения времени, вы найдете в библиотеке kxdate на GitHub (https://github.com/yole/kxdate).

Теперь, поняв, как работает этот простой DSL, перейдем к более сложному примеру: реализации DSL-запросов к базе данных.

11.4.3. Члены-расширения: внутренний DSL для SQL

Вы уже знаете, какую важную роль играют функции-расширения в архитектуре предметно-ориентированных языков. В этом разделе мы исследуем еще один трюк, упоминавшийся выше: объявление функций-расширений в классе. Такие функции или свойства являются членами содержащего их класса и в то же время расширяют какой-либо другой тип. Мы называем такие функции и свойства *членами-расширениями*.

Рассмотрим пару примеров использования членов-расширений. Они взяты из внутреннего DSL для SQL, реализованного в фреймворке Exposed, уже упоминавшемся выше. Но сначала мы должны обсудить, как Exposed позволяет определять структуру базы данных.

Для работы с таблицами в базе данных SQL фреймворк Exposed требует объявлять их как объекты, наследующие класс `Table`. Вот как выглядит объявление простой таблицы `Country` с двумя столбцами.

Листинг 11.25. Объявление таблицы в Exposed

```
object Country : Table() {
    val id = integer("id").autoIncrement().primaryKey()
    val name = varchar("name", 50)
}
```

Это объявление соответствует таблице в базе данных. Чтобы создать эту таблицу, нужно вызвать метод `SchemaUtils.create(Country)`, который сгенерирует все необходимые инструкции SQL, опираясь на объявление структуры таблицы:

```
CREATE TABLE IF NOT EXISTS Country (
    id INT AUTO_INCREMENT NOT NULL,
    name VARCHAR(50) NOT NULL,
    CONSTRAINT pk_Country PRIMARY KEY (id)
)
```

По аналогии с созданием разметки HTML объявления в оригинальном коде на Kotlin превращаются в элементы сгенерированной инструкции SQL.

Если проверить типы свойств в объекте `Country`, можно заметить, что все они имеют тип `Column` с обязательным типовым аргументом: `id` – тип `Column<Int>`, а `name` – тип `Column<String>`.

Класс `Table` в фреймворке Exposed определяет все типы столбцов, которые можно объявить в определении таблицы, включая использованные выше:

```
class Table {
    fun integer(name: String): Column<Int>
    fun varchar(name: String, length: Int): Column<String>
    // ...
}
```

Методы `integer` и `varchar` создают новые столбцы для хранения целочисленных и строковых значений соответственно.

Теперь посмотрим, как задать свойства столбцов. Для этого используются члены-расширения:

```
val id = integer("id").autoIncrement().primaryKey()
```

Методы `autoIncrement` и `primaryKey` определяют свойства столбцов. Все эти методы могут быть вызваны относительно экземпляра `Column` и возвращают сам этот экземпляр, что дает возможность составлять цепочки из вызовов методов. Вот упрощенные объявления этих функций:

```
class Table {
    fun <T> Column<T>.primaryKey(): Column<T>
    fun Column<Int>.autoIncrement(): Column<Int>
    // ...
}
```

Объявляет этот столбец первичным ключом

Автоматическое увеличение поддерживают только целочисленные значения

Эти функции – члены класса `Table`. То есть мы не сможем использовать их за пределами области видимости этого класса. Теперь вы знаете, почему имеет смысл объявлять методы как члены-расширения: это способ ограничить область их применения. У вас не получится определить свойства столбцов вне контекста таблицы: компилятор не найдет необходимых для этого методов.

Другая замечательная особенность используемых здесь функций-расширений – возможность ограничить тип получателя. Даже притом, что любой столбец таблицы может быть первичным ключом, автоматическое наращивание (auto-increment) поддерживают только числовые столбцы. Чтобы выразить это ограничение в API, нужно объявить метод `autoIncrement` как расширение для типа `Column<Int>`. Попытка объявить столбец любого другого типа автоматически наращиваемым вызовет ошибку на этапе компиляции.

Кроме того, когда какой-либо столбец объявляется первичным ключом с помощью `primaryKey`, эта информация сохраняется в таблице, содержа-

щей столбец. Объявление этой функции как члена класса `Table` позволяет сохранить необходимую информацию прямо в экземпляре таблицы.

> **Члены-расширения остаются обычными членами**
>
> Члены-расширения имеют один важный недостаток: ограничение расширяемости. Они принадлежат r-классу, поэтому у вас не получится определить новые члены-расширения на стороне.
>
> Например, представьте, что вам требуется добавить в Exposed поддержку новой базы данных и эта база данных поддерживает некоторые новые атрибуты столбцов. Чтобы добиться своей цели, вы должны были бы изменить определение класса `Table` и добавить в него члены-расширения для новых атрибутов. Но вы не сможете добавить необходимые объявления, не прикасаясь к исходному классу, как это возможно в случае с обычными расширениями (не членами), потому что у таких расширений не будет доступа к экземпляру `Table`, где они смогли бы сохранить определения.

Рассмотрим ещё один член-расширение, который можно найти в простом запросе SELECT. Представьте, что мы объявили две таблицы, `Customer` и `Country`, и каждая запись в таблице `Customer` хранит ссылку на страну (запись в `Country`), где проживает клиент. Следующий код выводит имена всех клиентов, проживающих в США (USA).

Листинг 11.26. Соединение двух таблиц в Exposed

```
val result = (Country join Customer)
    .select { Country.name eq "USA" }
result.forEach { println(it[Customer.name]) }
```

Этим строкам соответствует SQL-код: WHERE Country.name = "USA"

Метод `select` можно вызвать относительно экземпляра `Table` или соединения двух таблиц. Его аргумент – лямбда-выражение, определяющее условие выбора требуемых данных.

Но где определен метод `eq`? Мы можем уверенно сказать, что это инфиксная функция, принимающая строку `"USA"` как аргумент, и справедливо предположить, что это ещё один член-расширение.

В данном случае мы имеем дело с ещё одной функцией-расширением для `Column`, которая одновременно проявляется как член и потому может использоваться только в соответствующем контексте – например, для определения условия в вызове метода `select`. Вот как выглядят упрощенные объявления методов `select` и `eq`:

```
fun Table.select(where: SqlExpressionBuilder.() -> Op<Boolean>) : Query

object SqlExpressionBuilder {
    infix fun<T> Column<T>.eq(t: T) : Op<Boolean>
```

```
    // ...
}
```

Объект `SqlExpressionBuilder` определяет несколько способов выражения условий: сравнение значений, проверка на `null`, выполнение арифметических операций и так далее. Вам не придется явно использовать их в программном коде, но вы будете регулярно вызывать их посредством неявного получателя. Функция `select` принимает лямбда-выражение с получателем, которым неявно становится объект `SqlExpressionBuilder`. Это позволяет использовать в теле лямбда-выражения все возможные функции-расширения, объявленные в этом объекте, такие как `eq`.

Вы увидели два типа расширений для столбцов: используемые в объявлении `Table` и применяемые для сравнения значений в условиях. Без поддержки членов-расширений вам пришлось бы объявлять все эти функции как расширения или члены класса `Column`, что позволило бы использовать их в любом контексте. Поддержка членов-расширений дает возможность управлять их доступностью.

> **Примечание.** В разделе 7.5.6 мы видели код, работающий с Exposed, когда обсуждали использование делегируемых свойств в фреймворках. Делегируемые свойства часто применяются в DSL, и фреймворк Exposed прекрасно иллюстрирует это. Мы не будем возвращаться к обсуждению делегируемых свойств, потому что достаточно детально охватили их. Но если вы намерены создать свой предметно-ориентированный язык или усовершенствовать свой программный интерфейс, не забывайте об этой интересной особенности.

11.4.4. Anko: динамическое создание пользовательских интерфейсов в Android

Обсуждая лямбда-выражения с получателями, мы упоминали, что они прекрасно подходят для размещения компонентов пользовательского интерфейса. Давайте посмотрим, как библиотека Anko (https://github.com/Kotlin/anko) может помочь конструировать пользовательские интерфейсы для Android-приложений.

Сначала посмотрим, как библиотека Anko обертывает знакомый программный интерфейс Android в DSL-подобную структуру. В листинге 11.27 определяется диалог, демонстрирующий некоторое надоедливое предупреждение и две кнопки (подтверждающая желание продолжить и останавливающая обработку).

Листинг 11.27. Использование Anko для вывода диалога в Android

```
fun Activity.showAreYouSureAlert(process: () -> Unit) {
    alert(title = "Are you sure?",
            message = "Are you really sure?") {
```

```
        positiveButton("Yes") { process() }
        negativeButton("No") { cancel() }
    }
}
```

Заметили три лямбда-выражения в коде? Первое передается функции `alert` в третьем аргументе. Другие два – передаются функциям `positiveButton` и `negativeButton`. Получатель первого (внешнего) лямбда-выражения имеет тип `AlertDialogBuilder`. Здесь мы снова видим тот же шаблон: имя класса `AlertDialogBuilder` нигде не появляется в коде непосредственно, но мы обращаемся к его членам, добавляя элементы в диалог. Вот как объявлены члены, используемые в листинге 11.27.

Листинг 11.28. Объявление программного интерфейса `alert`

```
fun Context.alert(
        message: String,
        title: String,
        init: AlertDialogBuilder.() -> Unit
)

class AlertDialogBuilder {
    fun positiveButton(text: String, callback: DialogInterface.() -> Unit)
    fun negativeButton(text: String, callback: DialogInterface.() -> Unit)
    // ...
}
```

Мы добавили две кнопки в диалог. Если пользователь щелкнет на кнопке **Yes** (Да), будет выполнена затребованная операция. Если пользователь не уверен, операция будет отменена. Метод `cancel` является членом интерфейса `DialogInterface`, поэтому он вызывается относительно неявного получателя данного лямбда-выражения.

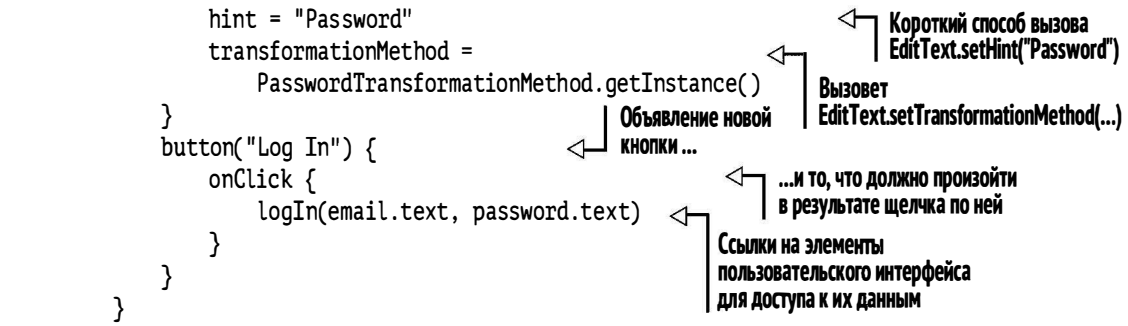
Теперь рассмотрим более сложный пример, где Anko DSL действует как полноценная замена макету размещения элементов пользовательского интерфейса в XML. В листинге 11.29 объявляется простая форма с двумя текстовыми полями ввода: одно служит для ввода адреса электронной почты, а второе – пароля. В конце добавляется кнопка с обработчиком события щелчка на ней.

Листинг 11.29. Использование Anko для определения простой формы

```
verticalLayout {
    val email = editText {
        hint = "Email"
    }
    val password = editText {
```

Объявление элемента EditText и сохранение ссылки на него

Неявным получателем в этом лямбда-выражении является обычный класс из Android API: android.widget.EditText



Лямбда-выражения с получателями – замечательный инструмент, упрощающий объявление структурированных элементов пользовательского интерфейса. Объявление их в коде (в сравнении с файлами XML) позволяет выделять повторяющуюся логику и использовать её повторно, как было показано в разделе 11.2.3. Мы можем отделить пользовательский интерфейс от прикладной логики и поместить их в разные компоненты, но и то, и другое будет реализовано в коде на языке Kotlin.

11.5. Резюме

- Внутренние предметно-ориентированные языки – это шаблон проектирования программных интерфейсов, который можно использовать для создания более выразительных API со структурами, состоящими из нескольких вызовов методов.
- Лямбда-выражения с получателями используют вложенную структуру, чтобы переопределить порядок разрешения методов в теле лямбда-выражения.
- Параметр, принимаемый лямбда-выражением с получателем, имеет тип функции-расширения, и вызывающая функция передает экземпляр получателя в вызов лямбда-выражения.
- Преимущество внутренних предметно-ориентированных языков в Kotlin перед внешними языками шаблонов или разметки заключается в возможности повторно использовать код и создавать абстракции.
- Использование объектов со специальными именами в параметрах инфиксных вызовов помогает создавать предметно-ориентированные языки, выражения на которых читаются как предложения на английском языке, без лишних знаков.
- Определение расширений для простых типов дает возможность создавать удобочитаемые синтаксические конструкции для литералов разного рода, таких как даты.

- Соглашение `invoke` позволяет вызывать произвольные объекты так, как если бы они были функциями.
- Библиотека kotlinx.html реализует внутренний предметно-ориентированный язык для создания HTML-страниц, который легко можно дополнить поддержкой фреймворков, предназначенных для создания пользовательских интерфейсов.
- Библиотека kotlintest реализует внутренний предметно-ориентированный язык для поддержки удобочитаемых конструкций проверки в модульных тестах.
- Библиотека Exposed реализует внутренний предметно-ориентированный язык для работы с базами данных.
- Библиотека Anko реализует несколько разных инструментов для разработки Android-приложений, включая внутренний предметно-ориентированный язык для определения макетов пользовательских интерфейсов.

Приложение A

Сборка проектов на Kotlin

В этом приложении рассказывается, как организовать сборку кода на Kotlin с использованием Gradle, Maven и Ant. Также здесь вы узнаете, как собирать приложения на Kotlin для Android.

A.1. Сборка кода на Kotlin с помощью Gradle

Для сборки проектов на Kotlin рекомендуется использовать систему Gradle. Gradle используется как стандартная система сборки проектов для Android и, кроме того, поддерживает все другие виды проектов, которые могут быть написаны на Kotlin. Gradle имеет гибкую модель проектов и обеспечивает высокую производительность, в первую очередь благодаря поддержке инкрементальной сборки, долгоживущих процессов сборки (демон Gradle) и других продвинутых приемов.

Разработчики Gradle работают над поддержкой Kotlin для создания сценариев сборки, что позволит вам использовать один язык и для приложений, и для сценариев сборки. В момент написания этих строк работа ещё не была завершена, поэтому более подробную информацию по этой теме ищите по адресу: https://github.com/gradle/gradle-script-kotlin. Для сценариев сборки Gradle в этой книге мы использовали синтаксис Groovy.

Вот как выглядит стандартный Gradle-сценарий для сборки Kotlin-проекта:

```
buildscript {
    ext.kotlin_version = '1.0.6'          <-- Используемая версия Kotlin

    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.jetbrains.kotlin:" +
                  "kotlin-gradle-plugin:$kotlin_version"   <-- Добавление зависимости сценария сборки от плагина поддержки Kotlin в Gradle
    }
```

```
}

apply plugin: 'java'
apply plugin: 'kotlin'

repositories {
    mavenCentral()
}

dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
```

Применяет плагин Kotlin для Gradle

Добавляет зависимость от стандартной библиотеки Kotlin

Сценарий ищет файлы с исходным кодом на языке Kotlin в следующих каталогах:

- *src/main/java* и *src/main/kotlin* – файлы с исходным кодом приложения;
- *src/test/java* и *src/test/kotlin* – файлы с исходным кодом тестов.

В большинстве случаев рекомендуется хранить файлы с исходным кодом на Kotlin и Java в одном каталоге. В частности, когда Kotlin внедряется в существующий проект, использование одного общего каталога уменьшит сложности, возникающие при преобразовании Java-файлов на Kotlin.

Если вы используете механизм рефлексии в Kotlin, добавьте ещё одну зависимость: библиотеку рефлексии Kotlin. Для этого включите следующую строку в раздел `dependencies`:

```
compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"
```

A.1.1. Сборка Kotlin-приложений для Android с помощью Gradle

Сборка приложений для Android осуществляется иначе, чем сборка обычных Java-приложений, поэтому для их сборки используют другой плагин для Gradle. Вместо `apply plugin: 'kotlin'` добавьте в сценарий сборки следующую строку:

```
apply plugin: 'kotlin-android'
```

Остальные настройки остаются такими же, как для обычных приложений.

Если у вас появится желание хранить исходный код на Kotlin в отдельных каталогах (например, в *src/main/kotlin*), зарегистрируйте их, чтобы Android Studio распознавала их как корневые каталоги с исходными текстами. Вот как это можно сделать:

```
android {
    ...
    sourceSets {
        main.java.srcDirs += 'src/main/kotlin'
    }
}
```

А.1.2. Сборка проектов с обработкой аннотаций

Многие Java-фреймворки, особенно те, что применяются для создания Android-приложений, опираются на обработку аннотаций во время компиляции. Чтобы использовать такие фреймворки в программах на Kotlin, нужно разрешить обработку аннотаций в сценарии сборки. Для этого добавьте такую строку:

```
apply plugin: 'kotlin-kapt'
```

Если у вас уже есть проект на Java, зависящий от обработки аннотаций, и вы решили внедрить в него Kotlin, то вам придется удалить существующую конфигурацию инструмента `apt`. Инструмент обработки аннотаций для Kotlin обрабатывает аннотации в классах на обоих языках, Java и Kotlin, а использование двух отдельных инструментов обработки аннотаций будет излишним. Чтобы указать зависимости, требуемые для обработки аннотаций, добавьте их в конфигурацию зависимостей `kapt`:

```
dependencies {
    compile 'com.google.dagger:dagger:2.4'
    kapt 'com.google.dagger:dagger-compiler:2.4'
}
```

В случае, когда процессоры аннотаций используются для каталогов `androidTest` или `test`, соответствующие конфигурации `kapt` называются `kaptAndroidTest` и `kaptTest`.

А.2. Сборка проектов на Kotlin с помощью Maven

Если вы предпочитаете собирать проекты с помощью Maven, то его тоже можно использовать для сборки проектов на Kotlin. Самый простой способ создать Maven-проект на Kotlin – использовать архетип `org.jetbrains.kotlin: kotlin-archetype-jvm`. Чтобы добавить поддержку Kotlin в существующий Maven-проект, достаточно выбрать пункт **Tools** → **Kotlin** → **Configure Kotlin** (Инструменты → Kotlin → Настройка Kotlin) в представлении **Project** (Проект), в плагине Kotlin IntelliJ IDEA.

Чтобы вручную добавить поддержку Maven в проект на Kotlin, нужно выполнить следующие шаги:

1. Добавить зависимость от стандартной библиотеки Kotlin (идентификатор группы: `org.jetbrains.kotlin`; идентификатор артефакта: `kotlin-stdlib`).
2. Добавить плагин Kotlin Maven (идентификатор группы: `org.jetbrains.kotlin`; идентификатор артефакта: `kotlin-maven-plugin`) и настроить его запуск на этапах компиляции и тестирования.
3. Настроить каталоги с исходными текстами, если вы храните исходный код на Kotlin отдельно от исходного кода на Java.

Ради экономии места мы не будем приводить здесь полных примеров файлов *pom.xml*, но вы найдете их в электронной документации по адресу: https://kotlinlang.org/docs/reference/usingmaven.html.

В смешанных Java/Kotlin-проектах необходимо настраивать плагин Kotlin так, чтобы он запускался перед запуском плагина Java. Это нужно потому, что плагин Kotlin способен анализировать исходный код на Java, тогда как плагин Java может читать только классы *.class* – то есть файлы с исходным кодом на Kotlin должны быть скомпилированы в файлы *.class* до того, как запустится плагин Java. Пример настройки вы найдете по адресу: http://mng.bz/73od.

А.3. Сборка кода на Kotlin с помощью Ant

Для сборки проектов с помощью Ant Kotlin реализует две задачи: задача `<kotlinc>` компилирует модули на Kotlin, а `<withKotlin>` служит расширением задачи `<javac>` для сборки смешанных модулей на Kotlin/Java. Вот простой пример использования `<kotlinc>`:

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
             classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlinc output="hello.jar">
            <src path="src"/>
        </kotlinc>
    </target>
</project>
```

Определение задачи <kotlinc>

Выполнит сборку единственного каталога с исходным кодом с помощью <kotlinc> и упакует результат в jar-файл

Задача `<kotlinc>` для Ant автоматически добавляет зависимость от стандартной библиотеки, поэтому вам не придется добавлять дополнительные аргументы для настройки этой задачи. Она также поддерживает упаковку скомпилированных файлов *.class* в jar-файл.

Вот пример использования задачи `<withKotlin>` для сборки смешанного модуля на Java/Kotlin:

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
             classpath="${kotlin.lib}/kotlin-ant.jar"/>
    <target name="build">
        <javac destdir="classes" srcdir="src">
            <withKotlin/>
        </javac>
        <jar destfile="hello.jar">
            <fileset dir="classes"/>
        </jar>
    </target>
</project>
```

В отличие от `<kotlinc>`, задача `<withKotlin>` не поддерживает автоматическую упаковку скомпилированных классов, поэтому в данном примере используется отдельная задача `<jar>` для упаковки.

Приложение B

Документирование кода на Kotlin

В этом приложении кратко рассказывается, как писать документирующие комментарии в коде на Kotlin и как на основе этих комментариев генерировать документацию с описанием API.

B.1. Документирующие комментарии в Kotlin

Документирующие комментарии с описанием объявлений на языке Kotlin напоминают аналогичные комментарии Javadoc в Java и называются *KDoc*. По аналогии с комментариями Javadoc KDoc-комментарии начинаются с последовательности `/**` и для описания конкретных частей объявлений используют теги, начинающиеся с `@`. Основное отличие KDoc от Javadoc заключается в том, что для записи комментариев используется формат Markdown (https://daringfireball.net/projects/markdown) вместо HTML. Чтобы упростить создание комментариев, KDoc поддерживает несколько дополнительных соглашений для ссылки на элементы документации – например, с описанием параметров функции.

Вот простой пример KDoc-комментария с описанием функции.

Листинг B.1. Комментарий KDoc

```
/**
 * Вычисляет сумму двух чисел, [a] и [b]
 */
fun sum(a: Int, b: Int) = a + b
```

Чтобы сослаться на объявление внутри комментария KDoc, нужно заключить имя этого объявления в квадратные скобки. В листинге B.1 это соглашение используется для ссылки на параметры описываемой функции, но точно так же можно ссылаться на другие объявления. Если объявление, на которое нужно сослаться, импортируется в код, содержащий коммен-

тарий KDoc, его имя можно использовать непосредственно. В противном случае следует указывать полностью квалифицированные имена. Если у вас появится желание определить свою метку для ссылки, используйте две пары квадратных скобок и поместите метку в первую пару, а имя объявления – во вторую: `[например][com.mycompany.SomethingTest.simple]`.

Вот более сложный пример, демонстрирующий использование тегов в комментариях.

Листинг В.2. Теги в комментариях

```
/**
 * Выполняет сложную операцию.                                          Описание параметра
 *
 * @param remote Если имеет значение true, операция выполняется удаленно
 * @return Результат выполнения операции          <— Описание возвращаемого значения
 * @throws IOException если соединение с удаленным узлом будет разорвано
 * @sample com.mycompany.SomethingTest.simple                           Описание возможного исключения
 */
fun somethingComplicated(remote: Boolean): ComplicatedResult { ... }
```

Включает в текст документации указанную функцию как пример

Синтаксис тегов в точности совпадает с Javadoc. Но, кроме стандартных тегов Javadoc, в KDoc поддерживается несколько дополнительных тегов для описания понятий, отсутствующих в Java, таких как тег `@receiver` для описания получателя функции-расширения или свойства-расширения. Полный список поддерживаемых тегов можно найти на странице http://kotlinlang.org/docs/reference/kotlin-doc.html.

Тег `@sample` можно использовать для включения в текст документации текста указанной функции в качестве примера использования описываемого API. Значение тега – полное квалифицированное имя включаемого метода.

Кроме того, в KDoc не поддерживаются следующие теги Javadoc:

- `@deprecated` – вместо него используется аннотация `@Deprecated`;
- `@inheritdoc` – не поддерживается, потому что документирующие комментарии в Kotlin всегда автоматически наследуются переопределяющими объявлениями;
- `@code`, `@literal` и `@link` – не поддерживаются, потому что используется соответствующее форматирование Markdown.

Обратите внимание: в команде разработчиков Kotlin предпочитают документировать параметры и возвращаемые значения функций непосредственно в тексте документации, как в листинге В.1. Использовать теги, как это сделано в листинге В.2, рекомендуется только тогда, когда параметр

или возвращаемое значение имеет сложную семантику и его описание должно быть четко отделено от основного текста документации.

B.2. Создание документации с описанием API

Инструмент, генерирующий документацию из исходного кода на Kotlin, называется Dokka: https://github.com/kotlin/dokka. Так же как сам Kotlin, Dokka полностью поддерживает смешанные проекты на Java/Kotlin. Он может читать комментарии Javadoc в коде на Java и комментарии KDoc в коде на Kotlin, а также генерировать документацию, охватывающую весь API модуля вне зависимости от языка, использовавшегося для определения каждого класса. Dokka поддерживает несколько выходных форматов, включая простую разметку HTML, разметку HTML в стиле Javadoc (которая использует синтаксис Java во всех объявлениях и показывает, как API выглядит с точки зрения Java) и Markdown.

Запускать Dokka можно из командной строки или из сценариев сборки Ant, Maven или Gradle. Рекомендуемый способ – добавить вызов Dokka в сценарии Gradle для сборки вашего модуля. Вот минимально необходимая конфигурация Dokka в сценарии сборки для Gradle:

```
buildscript {
    ext.dokka_version = '0.9.13'          <-- Определяет используемую версию Dokka

    repositories {
        jcenter()
    }

    dependencies {
        classpath "org.jetbrains.dokka:dokka-gradle-plugin:${dokka_version}"
    }
}

apply plugin: 'org.jetbrains.dokka'
```

С этой конфигурацией вы сможете сгенерировать документацию в формате HTML с описанием своего модуля, выполнив команду `./gradlew dokka`.

Информацию о дополнительных параметрах инструмента Dokka вы найдете по адресу: https://github.com/Kotlin/dokka/blob/master/README.md. В документации также описывается, как пользоваться Dokka в роли автономного инструмента или интегрировать его в сценарии сборки для Maven и Ant.

Приложение C

Экосистема Kotlin

Несмотря на относительно юный возраст, Kotlin уже имеет обширную экосистему библиотек, фреймворков и инструментов, большая часть из которых создана внешним сообществом разработчиков. В этом приложении мы дадим несколько советов, которые пригодятся для исследования этой экосистемы. Конечно, книга – не лучший способ передачи сведений о быстро расширяющейся коллекции инструментов, поэтому в первую очередь отметим ресурс, где вы сможете найти самую свежую информацию: https://kotlin.link/.

И напомним ещё раз: Kotlin полностью совместим с экосистемой библиотек Java. Подбирая библиотеку для решения своей задачи, не ограничивайте круг поиска только библиотеками, написанными на Kotlin, – вы можете с успехом использовать стандартные библиотеки для Java. Теперь перечислим некоторые библиотеки, достойные вашего внимания. Некоторые из Java-библиотек предлагают расширения для Kotlin с более ясными и идиоматичными API, и вы должны стремиться использовать эти расширения, если они доступны.

C.1. Тестирование

Помимо стандартных JUnit и TestNG, которые прекрасно работают с Kotlin, существуют другие фреймворки, реализующие выразительные предметно-ориентированные языки для записи тестов на Kotlin:

- *KotlinTest* (https://github.com/kotlintest/kotlintest) – гибкий фреймворк тестирования, написанный в духе ScalaTest и упоминавшийся в главе 11. Поддерживает несколько разных подходов к записи тестов;
- *Spek* (https://github.com/jetbrains/spek) – фреймворк тестирования в стиле BDD для Kotlin, первоначально разработанный в JetBrains и теперь поддерживаемый сообществом.

Если вы уверенно чувствуете себя с JUnit и вас интересует лишь более выразительный предметно-ориентированный язык для выполнения проверок, обратите внимание на *Hamkrest* (https://github.com/npryce/ham-

krest). Если вы широко используете фиктивные объекты в своих тестах, вам определенно стоит взглянуть на фреймворк *Mockito-Kotlin* (https://github.com/nhaarman/mockito-kotlin), который решает некоторые проблемы, связанные с созданием фиктивных экземпляров классов Kotlin, и реализует выразительный DSL.

С.2. Внедрение зависимостей

Популярные фреймворки с поддержкой внедрения зависимостей, такие как Spring, Guice и Dagger, прекрасно работают с Kotlin. Если вам интересно познакомиться с аналогичным решением на Kotlin, познакомьтесь с фреймворком *Kodein* (https://github.com/SalomonBrys/Kodein), который предлагает удобный DSL для настройки зависимостей и имеет весьма эффективную реализацию.

С.3. Сериализация JSON

Те, кто ищут более мощное решение для сериализации JSON, чем библиотека JKid, описанная в главе 10, имеют богатый выбор. Предпочитающие Jackson могут использовать модуль *jackson-module-kotlin* (https://github.com/FasterXML/jackson-module-kotlin), обладающий глубокой интеграцией с Kotlin, включая поддержку классов данных. Для GSON есть великолепная библиотека оберток *Kotson* (https://github.com/SalomonBrys/Kotson). А те, кому требуется более легкое решение, реализованное исключительно на Kotlin, могут попробовать *Klaxon* (https://github.com/cbeust/klaxon).

С.4. Клиенты HTTP

Если вам понадобится написать на Kotlin клиента для REST API, обратите внимание на Retrofit (http://square.github.io/retrofit). Это Java-библиотека, также совместимая с Android, которая прекрасно работает с Kotlin. Для более низкоуровневых решений можно порекомендовать OkHttp (http://square.github.io/okhttp/) или Fuel, библиотеку поддержки HTTP на Kotlin (https://github.com/kittinunf/Fuel).

С.5. Веб-приложения

Для разработки серверной части веб-приложений наиболее зрелые варианты на сегодняшний день – Java-фреймворки Spring, Spark Java и vert.x. Версия Spring 5.0 будет включать встроенную поддержку Kotlin. Желающие использовать Kotlin с предыдущими версиями Spring могут найти дополнительную информацию и вспомогательные функции в проекте Spring Kotlin (https://github.com/sdeleuze/spring-kotlin). Библиотека vert.x

также официально поддерживает Kotlin: https://github.com/vert-x3/vertx-lang-kotlin/.

Из решений, написанных исключительно на Kotlin, можно порекомендовать:

- *Ktor* (https://github.com/Kotlin/ktor) – пробный проект JetBrains, эксперимент по созданию современного, полнофункционального фреймворка для разработки веб-приложений с идиоматичным API;
- *Kara* (https://github.com/TinyMission/kara) – оригинальный веб-фреймворк на Kotlin, используемый в JetBrains и в других компаниях;
- *Wasabi* (https://github.com/wasabifx/wasabi) – HTTP-фреймворк, основанный на библиотеке Netty. Имеет выразительный Kotlin API;
- *Kovert* (https://github.com/kohesive/kovert) – REST-фреймворк, основанный на библиотеке vert.x.

Для создания разметки HTML можно использовать библиотеку *kotlinx.html* (https://github.com/kotlin/kotlinx.html), обсуждавшуюся в главе 11. Если вы предпочитаете более традиционные подходы, то используйте механизмы шаблонов для Java – например, Thymeleaf (www.thymeleaf.org).

С.6. Доступ к базам данных

В дополнение к традиционным средствам для работы с базами данных, доступным в Java (Hibernate и другие), существует несколько вариантов специально для Kotlin. Мы ближе всего знакомы с *Exposed* (https://github.com/jetbrains/Exposed) – фреймворком, позволяющим генерировать SQL-код и несколько раз обсуждавшимся в этой книге. Некоторые альтернативы перечислены на странице https://kotlin.link.

С.7. Утилиты и структуры данных

В последнее время набирает популярность парадигма *реактивного программирования*, и язык Kotlin прекрасно подходит для неё. Библиотека RxJava (https://github.com/ReactiveX/RxJava) де-факто стала стандартом для реактивного программирования в JVM и имеет официальную поддержку Kotlin https://github.com/ReactiveX/RxKotlin.

Ниже перечислены библиотеки, содержащие утилиты и структуры данных, которые могут пригодиться в ваших проектах:

- *funKTionale* (https://github.com/MarioAriasC/funKTionale) – реализует широкий диапазон примитивов функционального программирования (например, частичное применение функций);
- *Kovenant* (https://github.com/mplatvoet/kovenant) – реализация отложенных вычислений для Kotlin и Android.

C.8. Настольные приложения

Если вы занимаетесь разработкой настольных приложений на основе JVM, вы почти наверняка используете JavaFX. *TornadoFX* (https://github.com/edvin/tornadofx) включает разнообразные адаптеры для JavaFX, образуя естественную среду для разработки настольных приложений на Kotlin.

Предметный указатель

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. (499) 782-38-89

Электронный адрес: **books@alians-kniga.ru**.

Жемеров Дмитрий Борисович
Исакова Светлана Сергеевна

Kotlin в действии

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16. Печать цифровая.
Усл. печ. л. 37,68. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф