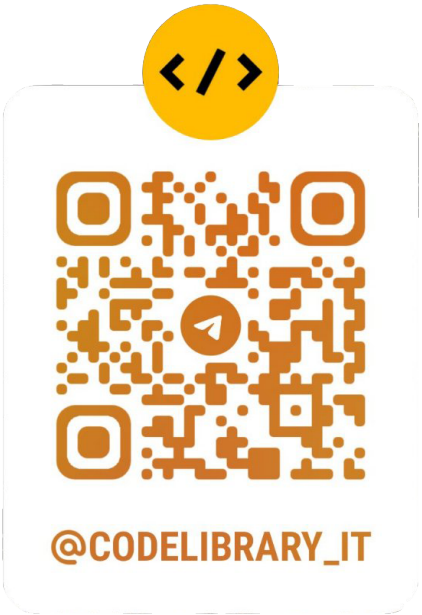
Кен Коузен

Kotlin. Сборник рецептов

Предметный подход



Kotlin Cookbook

A Problem-Focused Approach

Ken Kousen

Beijing • Boston • Farnham • Sebastopol • Tokyo

Kotlin. Сборник рецептов

Предметный подход

Кен Коузен

Москва, 2021

УДК 004.43Kotlin
ББК 32.972
К55

Кен Коузен
К55 Kotlin. Сборник рецептов. Предметный подход / пер. с анг. А. Н. Киселева. – М.: ДМК Пресс, 2021. – 220 с.: ил.

ISBN 978-5-97060-883-8

Этот сборник рецептов охватывает широкий спектр тем, с которыми следует ознакомиться разработчику, планирующему перейти на язык Kotlin или желающему изучить его более глубоко. В начале книги описывается процесс установки и запуска Kotlin, затем обсуждаются фундаментальные особенности языка. Особое внимание уделено его объектно-ориентированным возможностям, которые могут показаться необычными разработчикам на других языках.

Рецепты, собранные в разных главах по тематическому принципу, можно изучать в любом порядке, удобном читателю. Они дополняют друг друга, и каждый рецепт заканчивается ссылками на другие. Материал удобно структурирован: за описанием каждой задачи следуют ее решение и развернутое обсуждение.

Издание предназначено для разработчиков, знакомых с объектно-ориентированным программированием, особенно на Java или другом языке, основанном на JVM. Знание Java предпочтительно, но не обязательно.

УДК 004.43Kotlin
ББК 32.972

ISBN (анг.) 978-1-492-04667-7
ISBN (рус.) 978-5-97060-883-8

Оглавление

ЛАНЬ®

Посвящается Сандре, поддерживавшей меня все это время.

Твоя доброта, неослабевающая поддержка
и профессиональные навыки продолжают менять мою жизнь

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте **www.dmkpress.com**, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу **dmkpress@gmail.com**.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Предисловие

Каждые несколько лет появляется революционно новый язык, обещающий изменить подходы к разработке программного обеспечения. Однако такие обещания редко сбываются. Язык Kotlin – совсем другое дело. С момента создания в 2011 году он медленно и почти незаметно прокрался в базы кода по всему миру. Разработчики, долго использовавшие Java и неоднократно сталкивавшиеся с недостатками этого языка, смогли добавить вкрапления кода на Kotlin и благодаря этому уменьшили размер и увеличили мощность своего кода.

Получив некоторую известность как предпочтительный язык для разработки на Android, Kotlin достиг достаточно высокой степени зрелости, поэтому такая книга крайне необходима. Наполненная множеством полезных советов, «Kotlin. Сборник рецептов» начинает с самого начала. Кен последовательно показывает, как установить Kotlin и настроить его для использования в проекте. Как вызывать код на Kotlin из Java, выполнять его в браузере и в виде отдельного приложения. Но книга быстро движется вперед, описывая приемы решения задач программирования, с которыми каждый день сталкиваются разработчики и архитекторы.

Тестированию кода на Kotlin в книге выделен отдельный раздел, однако вы увидите, что идея тестирования пронизывает книгу от начала до конца. Тесты в книге служат практическими примерами использования языка и позволят вам точнее приспособить рецепты к своим потребностям.

Книга предлагает простую и действенную помощь, которая поможет вам продвинуться по пути освоения Kotlin. Это важное практическое руководство по Kotlin, которое должно лежать на рабочем столе каждого разработчика (реальном или виртуальном).

Дон Гриффитс (Dawn Griffiths) и Дэвид Гриффитс (David Griffiths),
авторы книги «Head First Kotlin»[1]
6 октября 2019 г.

[1] *Дон Гриффитс, Дэвид Гриффитс*. Head First. Kotlin. Питер, 2020. – *Прим. перев.*

Вступление

Добро пожаловать в «Kotlin. Сборник рецептов»! Общая цель книги – не только рассказать и показать синтаксис и семантику Kotlin, но также объяснить, когда и почему следует использовать ту или иную особенность. Книга не стремится охватить все детали синтаксиса Kotlin и представить исчерпывающий список библиотек, но содержит множество практических рецептов решения основных задач и должна быть понятна даже читателям, лишь поверхностно знакомым с Kotlin.

Компания JetBrains активно продвигает в сообщество Kotlin идею мультиплатформенной, нативной разработки и разработки в окружении JavaScript. Но после долгих размышлений было принято решение *не* включать в книгу рецепты, демонстрирующие эти направления, потому что все они или находятся в стадии бета-тестирования, или имеют низкую скорость внедрения. Как результат книга сосредоточена исключительно на Kotlin для JVM.

Код всех примеров можно найти в репозитории GitHub по адресу: https://github.com/kousen/kotlin-cookbook. Он включает оболочку Gradle (с файлом сборки, написанным на Kotlin DSL, разумеется), и все тесты в нем выполняются успешно.

Все примеры кода в книге были скомпилированы и протестированы с обеими доступными версиями Java с долгосрочной поддержкой, а именно с Java 8 и Java 11. Несмотря на то что технически для Java 8 истек срок службы, эта версия по-прежнему широко используется в отрасли, поэтому я решил протестировать примеры кода с ней тоже. На момент написания этой книги текущей была версия Kotlin 1.3.50 и шла работа над версией 1.3.60. Весь код работает с обеими версиями, и репозиторий GitHub будет обновляться для поддержки самой последней версии Kotlin.

Кому адресована эта книга

Эта книга написана для разработчиков, уже знакомых с основами объектно-ориентированного программирования, особенно на Java или другом языке, основанном на JVM. Знание Java пригодится, но не требуется.

Данная книга, как и любые другие книги рецептов, в большей степени ориентирована на описание приемов и идиом Kotlin, чем на исчерпывающее описание языка. Это позволяет без оглядки использовать всю широту возможностей языка в любом рецепте, но ограничивает пространство для описания основ этих возможностей. Каждая глава включает лишь краткое изложение основных методов, поэтому если вы имеете лишь смутное представление о том, как создавать коллекции, работать с массивами или проектировать классы, то не волнуйтесь. Подробное введение в язык вы найдете в справочном онлайн-руководстве (https://kotlinlang.org/docs/reference), и в книге часто упоминаются примеры и обсуждения, имеющиеся в нем.

Кроме того, в книге часто описываются реализации функций из библиотек Kotlin, чтобы показать, как разработчики языка работают с ним на практике, и рассказать, почему что-то делается именно так. Однако наличия у читателя предварительных знаний о реализации не ожидается, и вы можете пропускать эти детали.

Структура книги

Эта книга организована в виде сборника рецептов. Каждый рецепт самодостаточен и независим, но многие из них ссылаются на другие рецепты в книге. В общем и целом их можно читать в любом порядке. Тем не менее главы организованы следующим образом:

- глава 1 описывает процесс установки и запуска Kotlin, включая использование оболочки REPL, работу с инструментами сборки, такими как Maven и Gradle, и использование собственного генератора изображений в Graal;
- глава 2 описывает некоторые фундаментальные возможности и особенности Kotlin, такие как типы с поддержкой `null`, перегрузка операторов и преобразование типов, а затем переходит к исследованию некоторых малоизвестных вопросов, в том числе работы с операторами поразрядного сдвига и с функцией расширения `to` в классе `Pair`;
- глава 3 основное внимание уделяет объектно-ориентированным возможностям языка, которые могут показаться неожиданными или необычными разработчикам на других языках, таким как использование ключевого слова `const`, поддержка свойств, отложенная инициализация и ужасный класс `Nothing`, который гарантированно запутает любого разработчика на Java;
- глава 4 содержит лишь несколько рецептов, демонстрирующих возможности функционального программирования, которые требуют отдельного объяснения. Идеи функционального программирования рассматриваются на протяжении всей книги, особенно в рецептах, использующих коллекции, последовательности и сопрограммы, но в эту главу включено несколько приемов, которые могут показаться необычными и интересными;
- глава 5 охватывает массивы и коллекции, представляя в основном неочевидные методы работы с ними, такие как уничтожение коллекций, сортировка по нескольким свойствам, построение окна для коллекции и создание прогрессий;
- глава 6 описывает, как в Kotlin поддерживается «ленивая» обработка последовательностей элементов, по аналогии с обработкой потоков в Java. Рецепты в этой главе демонстрируют создание последовательностей, получение данных из них и работу с бесконечными последовательностями;
- глава 7 рассматривает еще одну тему, уникальную для Kotlin: функции, выполняющие блок кода в контексте объекта. Такие функции, как `let`, `apply` и `also`, играют очень важную роль в Kotlin, и эта глава рассказывает, почему их следует использовать, и показывает, как это делать;

- глава 8 обсуждает удобную возможность делегирования. Делегирование позволяет использовать композицию вместо наследования, и в стандартной библиотеке самого Kotlin имеется несколько делегатов, таких как `lazy`, `observable` и `vetoable`;
- глава 9 охватывает важную тему тестирования, делая основной упор на Junit 5. Эта текущая версия JUnit прекрасно поддерживает Kotlin и может использоваться для тестирования обычных приложений на Kotlin и кода на Kotlin в приложениях Spring Framework. В этой главе обсуждается несколько подходов, упрощающих написание и выполнение тестов;
- глава 10 включает пару рецептов управления ресурсами. Они демонстрируют приемы работы с файлами, а также использование функции `use`, которая широко применяется в нескольких контекстах;
- глава 11 рассматривает темы, которые трудно отнести к какой-то конкретной категории, в том числе: как получить текущую версию Kotlin, как заставить оператор `when` быть исчерпывающим, даже если он не возвращает значения, и как использовать функцию `replace` с регулярными выражениями. Здесь также обсуждается функция `TODO`, класс `Random` и некоторые способы интеграции с механизмом исключений в Java;
- глава 12 затрагивает вопросы работы с фреймворками Spring Framework и Spring Boot, очень дружелюбными, по отношению к Kotlin. Здесь приводятся рецепты, показывающие, как использовать классы Kotlin в роли управляемых bean-компонентов, как реализовать хранение данных с помощью JPA и как внедрять зависимости;
- глава 13 посвящена сопрограммам, одной из самых популярных особенностей Kotlin, и основам параллельного и конкурентного программирования на этом языке. Рецепты охватывают такие основы, как построители и диспетчеры, а также приемы отмены сопрограмм, их отладки и использования собственного пула потоков Java для их выполнения.

Главы, да и сами рецепты не обязательно читать в каком-либо определенном порядке. Они действительно дополняют друг друга, и каждый рецепт заканчивается ссылками на другие, но вы можете начать читать с любого места в книге. Основная цель глав – объединить похожие рецепты, и они построены так, что вы можете перепрыгивать между рецептами в произвольном порядке, чтобы решить свою задачу, стоящую перед вами в данный момент.

Специальное примечание для разработчиков на Android: в настоящее время Kotlin объявлен предпочтительным языком для разработки на Android, но это гораздо более широкий и универсальный язык программирования. Его можно использовать везде, где применяется Java, и даже шире. В этой книге нет специального раздела, посвященного исключительно Android, и приемы программирования на Kotlin для Android обсуждаются повсюду. Есть несколько конкретных рецептов, связанных с Android, таких как отмена сопрограмм, которые основаны на том факте, что библиотеки Android широко используют Kotlin, но в целом возможности языка, описанные в этой книге, можно использовать где угодно. Есть надежда, что охват языка в более общем плане поможет разработчикам для Android найти приемы, которые пригодятся им в любых других программных проектах.

Соглашения

В этой книге используются следующие соглашения по оформлению:

Курсив

Используется для обозначения новых терминов, адресов URL и электронной почты, имен файлов и расширений.

`Моноширинный шрифт`

Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных и функций, баз данных, типов данных, переменных окружения, инструкций и ключевых слов.

`Моноширинный жирный`

Обозначает команды или другой текст, который должен вводиться пользователем.

`Моноширинный курсив`

Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста.

Так выделяются советы и предложения.

Так обозначаются советы, предложения и примечания общего характера.

Так обозначаются предупреждения и предостережения.

Использование программного кода примеров

Вспомогательные материалы (примеры кода, упражнения и т. д.) доступны для загрузки по адресу: https://github.com/kousen/kotlin-cookbook.

Данная книга призвана оказать вам помощь в решении ваших задач. В общем случае все примеры кода из этой книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения примеров из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цити-

руя данную книгу или примеры из нее, получение разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги в вашу документацию вам необходимо будет получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «Kotlin Cookbook by Ken Kousen (O'Reilly). Copyright 2020 Ken Kousen, 978-1-492-04667-7».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу *permissions@oreilly.com*.

O'Reilly Online Learning

Вот уже более 40 лет *O'Reilly Media* (http://oreilly.com/) предоставляет технологии и бизнес-обучение, знания и опыт, помогающие компаниям добиться успеха.

Наше уникальное сообщество экспертов и новаторов делится своими знаниями и опытом через книги, статьи, конференции, а также нашу платформу онлайн-обучения. Платформа онлайн-обучения O'Reilly Online Learning предлагает доступ к очным курсам, углубленным учебным планам, интерактивным средам программирования и обширной коллекции текстовых и видеоматериалов от O'Reilly и более 200 других издателей. За дополнительной информацией обращайтесь по адресу: http://oreilly.com.

Как с нами связаться

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (США или Канада)
707-829-0515 (международный и местный)
707-829-0104 (факс)

На сайте издательства имеется веб-страница этой книги, где можно найти список опечаток в тексте, примеры кода и дополнительную информацию. Страница доступна по адресу: https://oreil.ly/kotlin-cookbook.

Свои пожелания и вопросы технического характера отправляйте по адресу: *bookquestions@oreilly.com*.

Дополнительную информацию о книгах, обучающие курсы, конференции и новости вы найдете на веб-сайте издательства: http://www.oreilly.com.

Ищите нас в Facebook: http://facebook.com/oreilly.

Следуйте за нами в Твиттере: http://twitter.com/oreillymedia.

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

Благодарности

На конференции Google I/O в 2017 году компания объявила, что Kotlin будет поддерживаемым языком разработки для Android. Позднее в том же году Gradle Inc. – компания, создавшая инструмент сборки Gradle, – объявила, что будет поддерживать предметно-ориентированный язык (DSL) Gradle для сборки. Оба этих события подтолкнули меня начать исследовать этот язык, и я рад, что сделал это.

Последние несколько лет я регулярно проводил презентации и семинары по Kotlin. Хотя основы языка легко изучить и применить на практике, я был впечатлен его глубиной и тем, насколько быстро он перенимает современные идеи разработки из других языков, таких как Groovy или Scala. Kotlin – это синтез многих передовых идей программирования, и я многому научился, углубившись в исследования перед написанием данной книги.

В процессе изучения я познакомился со многими активными разработчиками Kotlin, включая Дона Гриффитса (Dawn Griffiths) и Дэвида Гриффитса (Dave Griffiths), написавших выдающиеся книги «Head First Android Development»[2] и «Head First Kotlin»[3]; они даже согласились написать предисловие к этой книге. Хади Харрири (Hadi Hariri), технический евангелист JetBrains, регулярно проводит презентации о Kotlin. Его выступления всегда вдохновляют меня уделять время языку, и он оказался настолько любезен, что согласился взять на себя труд технического рецензирования для этой книги. Я очень благодарен им.

Билл Флай (Bill Fly) тоже принял участие в рецензировании книги. Я общался с ним на платформе O'Reilly Learning Platform бесчисленное множество раз, и он всегда подавал интересные идеи (и задавал сложные вопросы). Мой хороший друг Джим Хармон (Jim Harmon) помог мне освоить Android много лет назад и всегда был готов ответить на мои вопросы и рассказать о том, как Kotlin используется на практике. Марк Мейнард (Mark Maynard) – активный разработчик, который помог мне понять, как Kotlin взаимодействует с фреймворком Spring Framework, и я очень благодарен ему за это. Наконец, неподражаемый Венкат Субраманиам (Venkat Subramaniam), написавший свою собственную книгу о Kotlin (озаглавленную «Programming Kotlin» и такую же отменную, как и все остальные его книги), любезно согласился выделить время в своем плотном графике, чтобы помочь мне с моей книгой. Я был рад познакомиться со всеми моими техническими рецензентами, и меня впечатляет, сколько времени и сил они потратили на улучшение книги, которую вы сейчас видите.

Я также хочу поблагодарить многих из моих коллег-докладчиков по туру NFJS, в том числе Нейта Шутту (Nate Schutta), Майкла Кардуччи (Michael Carducci), Мэтта Стайна (Matt Stine), Брайана Слеттена (Brian Sletten), Марка Ричардса (Mark Richards), Пратика Пателя (Pratik Patel), Нила Форда (Neal Ford), Крейга

[2] *Дон Гриффитс и Дэвид Гриффитс.* Head First. Программирование для Android. Питер, 2016. ISBN: 978-5-496-02171-5. – *Прим. перев.*

[3] *Дон Гриффитс и Дэвид Гриффитс.* Head First. Kotlin. Питер, 2020. ISBN: 978-5-4461-1335-4. – *Прим. перев.*

Уоллса (Craig Walls), Раджу Ганди (Raju Gandhi), Джонатана Джонсона (Jonathan Johnson) и Дэна Инохоса (Dan «the Man» Hinojosa), за их постоянные внимание и поддержку. Я наверняка пропустил кого-то в этом перечислении, и если это действительно так, то уверяю вас, что это было сделано не намеренно.

Написание книг и преподавание на учебных курсах (моя основная работа) – это не коллективная работа. Приятно иметь друзей и коллег, на внимание и советы которых я могу рассчитывать.

В создании этой книги приняли участие многие сотрудники O'Reilly Media. Очень непросто перечислить их всех, поэтому я особо упомяну Зана МакКуэйда (Zan McQuade), которого часто ставил в неловкое положение из-за своего нерегулярного графика и моего противоречивого характера. Спасибо тебе за терпение, понимание и упорный труд над этой книгой.

Наконец, я хочу выразить всю свою любовь моей жене Джинджер (Ginger) и моему сыну Ксандеру (Xander). Без поддержки моей семьи я не стал бы тем, кем являюсь сегодня, и этот факт становится для меня все очевиднее с каждым годом. Я никогда не смогу выразить, насколько вы оба дороги мне.

ЛАНЬ®

Глава 1

Установка и запуск Kotlin

Рецепты в этой главе помогут вам начать работу с компилятором Kotlin из командной строки и в интегрированной среде разработки (Integrated Development Environment, IDE).

1.1. Запуск Kotlin без локального компилятора

Задача

Опробовать Kotlin без установки на локальный компьютер, например на Chromebook, не поддерживающем такую возможность.

Решение

Использовать Kotlin Playground (https://play.kotlinlang.org/) – онлайн-песочницу для исследования Kotlin.

Обсуждение

Kotlin Playground предлагает простую возможность поэкспериментировать с Kotlin, исследовать его возможности или просто опробовать Kotlin в системах, где отсутствует компилятор этого языка. Эта онлайн-песочница дает доступ к последней версии компилятора, а также к веб-редактору, позволяющему писать и выполнять свой код.

На рис. 1.1 показан скриншот страницы Kotlin Playground в браузере.

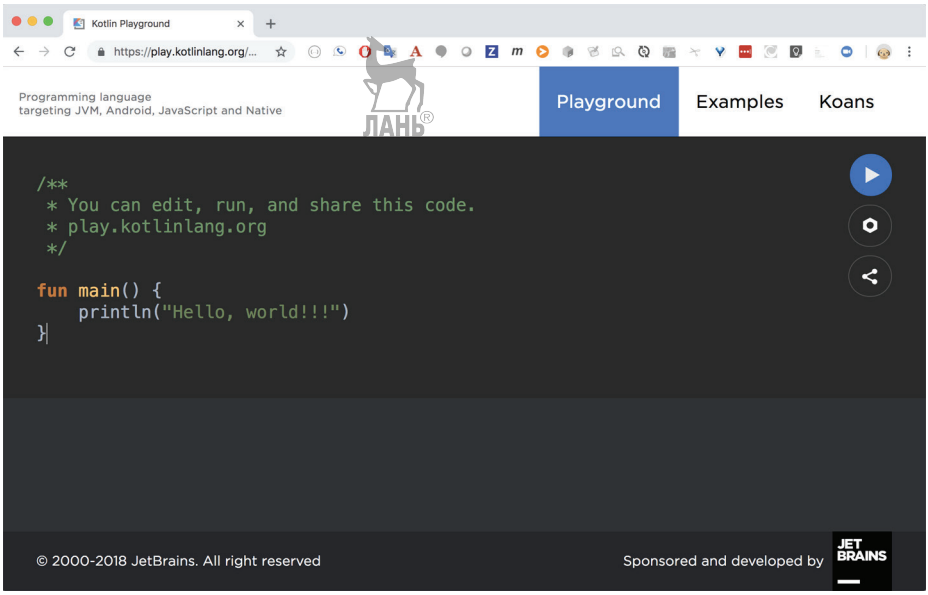


Рис. 1.1. Домашняя страница Kotlin Playground

Просто введите свой код и щелкните на кнопке Run (Запустить), чтобы выполнить его. Кнопка Settings (Настройки), со значком шестеренки, позволяет изменить версию Kotlin, выбрать платформу для запуска (JVM, JS, Canvas или JUnit) или добавить аргументы для программы.

Начиная с версии Kotlin 1.3 функцию main можно объявлять без параметров.

В разделе Examples (Примеры) вы найдете множество примеров программ, организованных по темам, которые можно выполнять прямо в браузере. На рис. 1.2 показана страница с примером программы «Hello world».

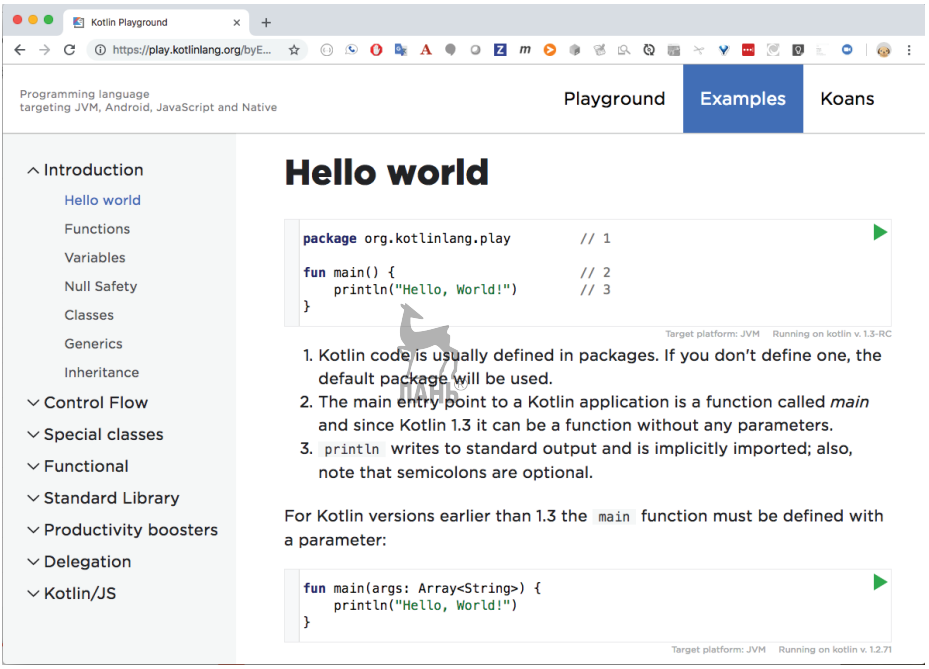


Рис. 1.2. Примеры в Kotlin Playground

В специальном разделе Koans (Задачи) вы найдете серию упражнений, которые помогут вам ближе познакомиться с языком. Упражнения доступны не только в интернете – если вы используете IntelliJ IDEA или Android Studio, то доступ к упражнениям можно получить с помощью плагина EduTools.

1.2. Установка Kotlin на локальный компьютер

Задача

Получить возможность запускать код на Kotlin из командной строки на локальном компьютере.

Решение

Установить компилятор вручную из GitHub или воспользоваться диспетчером пакетов операционной системы.

Обсуждение

На странице http://kotlinlang.org/docs/tutorials/command-line.html описываются возможные варианты установки компилятора командной строки. Один из вариантов – загрузить ZIP-файл с программой установки для своей операционной системы. На этой странице имеется ссылка на репозиторий GitHub (*https://oreil.ly/AXqXM*) с текущими версиями Kotlin. Там вы найдете файлы ZIP с пакетами для Linux, macOS, Windows и с исходным кодом. Просто разархивируйте пакет и добавьте путь к его подкаталогу *bin* в переменную окружения `PATH`.

Установить компилятор вручную несложно, но некоторые разработчики предпочитают использовать диспетчер пакетов. *Диспетчер пакетов* автоматизирует процесс установки, а некоторые из них даже позволяют поддерживать несколько версий компилятора.

SDKMAN!, Scoop и другие диспетчеры пакетов

SDKMAN! (https://sdkman.io/) – одна из самых популярных программ установки пакетов. Первоначально она разрабатывалась для командных оболочек Unix, но уже есть планы создать версии для других платформ.

Установка Kotlin с помощью SDKMAN! начинается с загрузки и установки этой программы с использованием `curl`:

```
> curl -s https://get.sdkman.io | bash
```

После установки следует выполнить команду `sdk`, чтобы установить любой из поддерживаемых продуктов, в число которых входит и Kotlin:

```
> sdk install kotlin
```

По умолчанию устанавливается последняя версия в каталог *~/.sdkman/candidates/kotlin* вместе со ссылкой `current`, указывающей на выбранную версию.

Узнать, какие версии доступны, можно с помощью команды `list`:

```
> sdk list kotlin
```

По умолчанию команда `install` выбирает последнюю версию, но при желании ее можно заставить установить конкретную версию:

```
> sdk use kotlin 1.3.50
```

Эта команда установит версию Kotlin 1.3.50.

IntelliJ IDEA и Android Studio могут использовать и свою собственную, и загруженную версию.

В числе других диспетчеров пакетов, поддерживающих Kotlin, можно назвать: Homebrew (http://brew.sh/), MacPorts (https://www.macports.org/) и Snapcraft (https://snapcraft.io/).

В Windows можно использовать Scoop (https://scoop.sh/). Scoop играет в Windows ту же роль, что другие диспетчеры пакетов в системах, отличных от Windows. Scoop требует наличия в системе PowerShell 5 или выше и .NET Framework 4.5 или выше. Инструкции по установке можно найти на сайте Scoop.

После установки Scoop можно установить текущую версию Kotlin:

```
> scoop install kotlin
```

Эта команда установит сценарии *kotlin.bat*, *kotlinc.bat*, *kotlin-js.bat* и *kotlin-jvm.bat* и добавит путь к ним в переменную окружения `PATH`.

Этого вполне достаточно, но если вы решите поэкспериментировать, попробуйте экспериментальную программу установки `kotlin-native`, которая также устанавливает собственный компилятор для Windows. При этом еще будет установлен LLVM-компилятор для Kotlin, среда выполнения и инструмент генерации низкоуровневого кода с использованием комплекта инструментов LLVM.

Независимо от способа установки Kotlin, убедиться в его доступности и работоспособности можно с помощью простой команды `kotlin -version`. Вот типичный вывод этой команды:

```
> kotlin -version
Kotlin version 1.3.50-release-112 (JRE 13+33)
```

Смотри также

Рецепт 1.3, где обсуждается, как использовать Kotlin из командной строки после установки.

1.3. Компиляция и выполнение кода на Kotlin из командной строки

Задача

Скомпилировать и выполнить код на Kotlin из командной строки.

Решение

Использовать команды `kotlinc-jvm` и `kotlin`.

Обсуждение

Kotlin SDK для JVM включает команду `kotlinc-jvm` вызова компилятора Kotlin и команду `kotlin` выполнения кода на Kotlin. Они используются подобно командам `javac` и `java` в Java.

Дистрибутив Kotlin включает сценарий `kotlinc-js` для компиляции в JavaScript. В этой книге предполагается использование версии для JVM. Базовый сценарий `kotlinc` – это псевдоним для `kotlinc-jvm`.

Рассмотрим для примера простейшую программу «Hello, Kotlin!». Создайте файл с именем *hello.kt* и добавьте в него код из примера 1.1.

Пример 1.1. hello.kt

```
fun main() {
    println("Hello, Kotlin!")
}
```

Команда `kotlinc` скомпилирует этот файл, а команда `kotlin` выполнит полученный файл класса, как показано в примере 1.2.

Пример 1.2. Компиляция и выполнение файла с кодом на Kotlin

```
> kotlinc-jvm hello.kt     ❶

> ls
hello.kt HelloKt.class     ❷

> kotlin HelloKt
Hello, Kotlin!
```

❶ Компиляция исходного кода

❷ Выполнение получившегося файла класса

Компилятор создаст файл *HelloKt.class* с байт-кодом, который можно выполнить в виртуальной машине Java. Kotlin не генерирует исходный код на Java – это не транспилятор. Он генерирует байт-код, который может интерпретироваться виртуальной машиной JVM.

Скомпилированный класс получает имя, соответствующее имени файла, но с первой заглавной буквой, и в конец имени добавляется окончание *Kt*. Этим поведением можно управлять с помощью аннотаций.

Чтобы создать автономный файл JAR, который можно запустить командой `java`, добавьте аргумент `-include-runtime`, как показано в примере 1.3.

Пример 1.3. Включение среды выполнения Kotlin

```
> kotlinc-jvm hello.kt -include-runtime -d hello.jar
```

Эта команда создаст файл *hello.jar*, который можно запустить командой `java`:

```
> java -jar hello.jar
Hello, Kotlin!
```

Без флага `-include-runtime` компилятор создаст JAR-файл, которому необходимо, чтобы среда выполнения Kotlin находилась в пути к классам (classpath).

Команда `kotlinc` без аргументов запускает интерактивную оболочку Kotlin REPL, которая обсуждается в рецепте 1.4.

Смотри также

Рецепт 1.4 демонстрирует, как использовать интерактивную оболочку Kotlin REPL (Read-Eval-Print-Loop – прочитать, вычислить, вывести и повторить). Рецепт 1.5 демонстрирует выполнение сценариев на Kotlin из командной строки.

1.4. Использование Kotlin REPL

Задача

Выполнить код на Kotlin в интерактивной оболочке.

Решение

Запустить Kotlin REPL командой `kotlinc` без аргументов.

Обсуждение

Kotlin включает интерактивную оболочку для работы с компилятором, которая называется REPL и запускается командой `kotlinc` без аргументов. После запуска в оболочке REPL можно вводить произвольные команды Kotlin и сразу же получать результаты.

Оболочка Kotlin REPL также доступна в Android Studio и IntelliJ IDEA в виде пункта меню Tools → Kotlin → Kotlin REPL (Инструменты → Kotlin → Kotlin REPL).

После запуска команды `kotlinc` вы оказываетесь в интерактивной оболочке. В примере 1.4 показан пример сеанса работы в этой оболочке.

Пример 1.4. Использование оболочки Kotlin REPL

```
> kotlinc
Welcome to Kotlin version 1.3.50 (JRE 11.0.4+11)
Type :help for help, :quit for quit
>>> println(“Hello, World!”)
Hello, World!
>>> var name = “Dolly”
>>> println(“Hello, $name!”)
Hello, Dolly!

>>> :help
Available commands:
:help                 show this help
:quit                 exit the interpreter
:dump                 bytecode dump classes to terminal
:load <file>          load script from specified file
>>> :quit
```

Оболочка REPL позволяет легко и быстро вычислять выражения Kotlin без запуска IDE. Используйте ее в случаях, когда нежелательно создавать проект или другую коллекцию файлов в IDE и нужно лишь быстро проверить какую-то идею, помочь другому разработчику либо если IDE вообще отсутствует на компьютере.

1.5. Запуск сценария на Kotlin

Задача

Написать и выполнить сценарий на Kotlin.

Решение

Сохранить код в файле с расширением *.kts* и использовать команду `kotlinc` с параметром `-script`, чтобы запустить его.

Обсуждение

Команда `kotlinc` поддерживает несколько параметров командной строки. Один из них позволяет использовать эту команду как интерпретатор для выполнения сценариев на Kotlin. Сценарий – это текстовый файл с исходным кодом на Kotlin и с расширением *.kts*.

Для демонстрации в примере 1.5 приводится сценарий в файле *southpole.kts*, отображающий текущее время на Южном полюсе и какое время используется – зимнее или летнее. Сценарий использует пакет *java.time*, добавленный в Java 8.

Пример 1.5. southpole.kts

```
import java.time.*

val instant = Instant.now()
val southPole = instant.atZone(ZoneId.of("Antarctica/South_Pole"))
val dst = southPole.zone.rules.isDaylightSavings(instant)
println("It is ${southPole.toLocalTime()} (UTC${southPole.offset}) at the South Pole")
println("The South Pole ${if (dst) "is" else "is not"} on Daylight Savings Time")
```

Запустить этот сценарий можно командой `kotlinc` с параметром `-script`:

```
> kotlinc -script southpole.kts
It is 10:42:56.056729 (UTC+13:00) at the South Pole
The South Pole is on Daylight Savings Time
```

Сценарии содержат код, который обычно помещается в стандартный метод `main` класса. То есть Kotlin можно использовать как язык сценариев для JVM.

1.6. Сборка автономного приложения с помощью GraalVM

Задача

Создать приложение, которое можно запускать из командной строки без применения любых дополнительных зависимостей.

Решение

Использовать компилятор GraalVM и инструмент `native-image`.

Обсуждение

GraalVM (https://www.graalvm.org/) – это высокопроизводительная виртуальная машина с универсальной средой выполнения для запуска приложений, написанных на различных языках. Вы можете написать приложение на любом языке, основанном на JVM, таком как Java или Kotlin, и интегрировать его с программным кодом на JavaScript, Ruby, Python, R и других языках.

Одной из замечательных особенностей GraalVM является возможность использовать ее для создания автономного выполняемого файла. Этот рецепт демонстрирует простой способ использования инструмента `native-image` из GraalVM для создания двоичных файлов из исходного кода на Kotlin.

Получить дистрибутив GraalVM можно по адресу: https://oreil.ly/UcmZD. Для разработки текущего рецепта я установил бесплатную версию Community Edition с помощью диспетчера пакетов SDKMAN!:

```
> sdk install java 19.2.0.1-grl
> java -version
openjdk version «1.8.0_222»
OpenJDK Runtime Environment (build 1.8.0_222-20190711112007.graal.jdk8u-src...
OpenJDK 64-Bit GraalVM CE 19.2.0.1 (build 25.222-b08-jvmci-19.2-b02, mixed mode)

> gu install native-image
// эта команда установит компонент native-image
```

Возьмем за основу Kotlin-версию программы «Hello, World!», изображенную на рис. 1.1, и воспроизведем ее:

```
fun main() {
    println(“Hello, World!”)
}
```

Как отмечалось в рецепте 1.3, этот сценарий легко скомпилировать с помощью `kotlinc-jvm`, получить файл *HelloKt.class*, а затем запустить его командой `kotlin`:

```
> kotlinc-jvm hello.kt // сгенерирует HelloKt.class
> kotlin HelloKt
Hello, World!
```

Но, чтобы получить автономный выполняемый файл, сначала следует скомпилировать сценарий с параметром `-include-runtime` и получить файл *hello.jar*:

```
> kotlinc-jvm hello.kt -include-runtime -d hello.jar
```

А потом с помощью инструмента `native-image`, входящего в состав GraalVM, сгенерировать из него выполняемый файл, как показано в примере 1.6.

Пример 1.6. Создание автономного выполняемого файла с помощью GraalVM

```
> native-image -jar hello.jar
```

Вот выдержка из документации: «Для компиляции *native-image* использует набор инструментов, установленный локально, поэтому в системе должны быть установлены пакеты *glibc-devel*, *zlib-devel* (с файлами заголовков для библиотеки C и zlib) и gcc» .

В процессе работы эта команда выведет следующие строки:

```
> native-image -jar hello.jar
Build on Server(pid: 61247, port: 49590)*
[hello:61247]    classlist:   1,497.63 ms
[hello:61247]        (cap):   2,225.47 ms
[hello:61247]        setup:   3,451.98 ms
[hello:61247]   (typeflow):   2,163.16 ms
[hello:61247]    (objects):   1,793.53 ms
[hello:61247]   (features):     215.90 ms
[hello:61247]     analysis:   4,247.68 ms
[hello:61247]     (clinit):     107.96 ms
[hello:61247]     universe:     399.58 ms
[hello:61247]      (parse):     329.84 ms
[hello:61247]     (inline):     753.12 ms
[hello:61247]    (compile):   3,426.14 ms
[hello:61247]      compile:   4,807.54 ms
[hello:61247]        image:     306.96 ms
[hello:61247]        write:     180.22 ms
[hello:61247]      [total]:  15,246.88 ms
```

В результате будет создан файл *hello*, который можно запустить из командной строки. В Mac или другой Unix-подобной системе для этого достаточно выполнить команду:

```
> ./hello
Hello, World!
```

Теперь мы знаем три способа запуска сценариев на Kotlin:

- скомпилировать его командой `kotlinc-jvm` и затем запустить командой `kotlin`;
- скомпилировать JAR-файл с включением в него среды выполнения и затем выполнить командой `java`;
- скомпилировать командой `kotlinc`, создать двоичный файл с помощью GraalVM и затем выполнить как самую обычную команду.

Размеры файлов, получаемых в этих трех случаях, сильно различаются. Размер скомпилированного файла *HelloKt.class* с байт-кодом составляет около 700 байт. Размер файла *hello.jar* с включенной средой выполнения составляет около 1,2 Мбайт. Автономный двоичный файл получается еще больше – около 2,1 Мбайт. Однако разница в скорости выполнения огромна даже для такого крошечного сценария, как показано в примере 1.7.

Пример 1.7. Хронометраж выполнения сценария hello

```
> time kotlin HelloKt
Hello, World!
kotlin HelloKt 0.13s user 0.05s system 112% cpu 0.157 total

~/Documents/kotlin
> time java -jar hello.jar
Hello, World!
java -jar hello.jar 0.08s user 0.02s system 99% cpu 0.106 total
```

```
~/Documents/kotlin
> time ./hello
Hello, World!
./hello 0.00s user 0.00s system 59% cpu 0.008 total
```

Результаты весьма показательны. JAR-файл выполняется несколько быстрее, чем запуск класса командой `kotlin`, но двоичный файл выполняется на порядок быстрее. В этом примере для его выполнения потребовалось всего около 8 миллисекунд.

Если вы пользуетесь Gradle, то можете использовать плагин поддержки GraalVM с названием `gradle-graal`. Он добавит в вашу систему сборки задачу `gradle-graal` (кроме всего прочего). Подробности смотрите на домашней странице (https://oreil.ly/3eY3Y) плагина.

1.7. Добавление в Gradle плагина поддержки Kotlin (синтаксис Groovy)

Задача

Добавить в систему сборки Gradle плагин поддержки Kotlin, используя синтаксис предметно-ориентированного языка (Domain-Specific Language, DSL) Groovy.

Решение

Добавить в файл сборки зависимость Kotlin и плагин, используя теги Groovy DSL.

Обсуждение

В этом рецепте используется язык Groovy DSL для Gradle. В следующем рецепте будет показано, как использовать Kotlin DSL для Gradle.

Инструмент сборки Gradle (https://gradle.org/) поддерживает компиляцию исходного кода на Kotlin в байт-код JVM с помощью плагина, предлагаемого компанией JetBrains. Плагин `kotlin-gradle-plugin` зарегистрирован в репозитории плагинов Gradle и может быть добавлен в сценарий сборки Gradle, как показано в примере 1.8. Этот код нужно добавить в файл *build.gradle* в корне проекта.

Пример 1-8. Добавление плагина поддержки Kotlin с помощью блока plugins (Groovy DSL)

```
plugins {
    id «org.jetbrains.kotlin.jvm» version «1.3.50»
}
```

Значение `version` представляет одновременно версию плагина и Kotlin. Gradle все еще поддерживает старый синтаксис добавления плагинов, как показано в примере 1.9.

Пример 1.9. Старый синтаксис добавления плагина поддержки Kotlin (Groovy DSL)

```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath 'org.jetbrains.kotlin:kotlin-gradle-plugin:1.3.50'
    }
}

plugins {
    id «org.jetbrains.kotlin.jvm» version «1.3.50»
}
```

В обоих примерах используется синтаксис Groovy DSL для файлов сборки Gradle, который поддерживает строки в одинарных и в двойных кавычках. В языке Groovy, как и в Kotlin, строки в двойных кавычках поддерживают интерполяцию, но здесь этого не требуется, поэтому двойные кавычки можно заменить одинарными.

Блок `plugins`, в отличие от блока `repositories`, не требует указывать местоположение плагина. Это верно для любого плагина Gradle, зарегистрированного в репозитории плагинов Gradle. Блок `plugins` также автоматически «применяет» плагин, поэтому при его использовании оператор `apply` не требуется.

Файл *settings.gradle* рекомендуется, но не требуется. Он обрабатывается на этапе инициализации, когда Gradle определяет, какие файлы сборки в проекте необходимо проанализировать. В случае сборки сразу нескольких проектов файл настроек показывает, какие подкаталоги в корневом каталоге также являются каталогами проектов. Gradle позволяет использовать общие настройки и зависимости для подпроектов, сделать один подпроект зависимым от другого и даже выполнять сборку подпроектов параллельно. За дополнительной информацией о сборках с несколькими проектами обращайтесь к руководству пользователя Gradle (https://oreil.ly/mwGJW).

Исходный код на Kotlin можно смешивать с исходным кодом на Java в одной папке или хранить их отдельно, в разных папках, например *src/main/java* и *src/main/kotlin*.

Проекты для Android

Плагин поддержки Kotlin для Android работает немного иначе. Проекты для Android – это сборки Gradle с несколькими подпроектами, поэтому они обычно имеют два файла *build.gradle*: один в корневом каталоге и один в подкаталоге с именем по умолчанию *app*. В примере 1.10 показан типичный файл *build.gradle* верхнего уровня, содержащий только информацию о плагине Kotlin.

Пример 1.10. Использование Kotlin в проектах для Android (Groovy DSL)

```
buildscript {
    ext.kotlin_version = '1.3.50'
    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.5.0'
        classpath «org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version»
    }
}
// ... другие задачи, не связанные с плагином ...
```

Затем, выражаясь языком Gradle, плагин *применяется*, как показано в примере 1.11 с типичным файлом *build.gradle* в каталоге *app*.

Пример 1.11. Применение плагина Kotlin

```
apply plugin: 'com.android.application'

apply plugin: 'kotlin-android'                ❶

apply plugin: 'kotlin-android-extensions'     ❷

android {
    // ... android information ...
}

dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8:$kotlin_version" ❸

    // ... other unrelated dependencies ...
}
```

❶ Применяет плагин Kotlin для Android

❷ Применяет расширения для плагина Kotlin для Android

❸ Определение зависимости от стандартной библиотеки, можно использовать JDK 8 или JDK 7

Плагин Kotlin для Android объявляется в разделе `buildscript` и затем применяется в этом файле. Плагин умеет компилировать код на Kotlin внутри Android-приложения. В состав загружаемого плагина также входит пакет расширений для Android, которые упрощают доступ к виджетам Android по их идентификаторам.

Плагин поддержки Kotlin может генерировать байт-код для JDK 7 или JDK 8. Измените значение `jdk` в указанной зависимости, чтобы выбрать предпочитаемую версию.

На момент написания этой книги нельзя было выбрать Kotlin DSL при создании проекта для Android. Конечно, можно создать файлы сборки вручную и использовать в них Kotlin DSL, но этот подход редко применяется на практике. Kotlin DSL будет доступен в версии Android Studio 4.0, которая также будет включать полную поддержку файлов KTS и «живых шаблонов» Kotlin.

Смотри также

Тот же процесс с использованием Kotlin DSL, кроме раздела для Android, показан в рецепте 1.8.

1.8. Добавление в Gradle плагина поддержки Kotlin (синтаксис Kotlin)

Задача

Добавить в систему сборки Gradle плагин поддержки Kotlin, используя синтаксис Kotlin DSL.

Решение

Добавить в файл сборки зависимость Kotlin и плагин, используя теги Kotlin DSL.

Обсуждение

В этом рецепте применяется язык Kotlin DSL для Gradle. В предыдущем рецепте показано, как использовать Groovy DSL для Gradle.

Версия Gradle 5.0 и выше включает новый язык Kotlin DSL для настройки файлов сборки. В них также доступен плагин `kotlin-gradle-plugin`, зарегистрированный в репозитории плагинов Gradle, который можно добавить в сценарий сборки Gradle, как показано в примере 1.12. В качестве альтернативы можно использовать старый синтаксис `buildscript` (см. пример 1.13). Этот код нужно добавить в файл *build.gradle.kts* в корне проекта.

Пример 1-12. Добавление плагина поддержки Kotlin с помощью блока plugins (Kotlin DSL)

```
plugins {
    kotlin(«jvm») version «1.3.50»
}
```

Пример 1.13. Старый синтаксис добавления плагина поддержки Kotlin (Kotlin DSL)

```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath(kotlin(«gradle-plugin», version = «1.3.50»))
    }
}

plugins {
    kotlin(“jvm”)
}
```

Блок `plugins`, в отличие от блока `repositories`, не требует указывать местоположение плагина. Это верно для любого плагина Gradle, зарегистрированного в репозитории плагинов Gradle. Блок `plugins` также автоматически «применяет» плагин, поэтому при его использовании оператор `apply` не требуется.

По умолчанию файлам сборки на Kotlin DSL в Gradle даются имена *settings.gradle.kts* и *build.gradle.kts*.

Как можно заметить, самые большие отличия от Groovy DSL заключаются в следующем:

- все строки должны заключаться в двойные кавычки;
- в Kotlin DSL необходимо использовать круглые скобки;
- присваивание в Kotlin определяется знаком равенства (=), а не двоеточием (:).

Файл *settings.gradle.kts* рекомендуется, но не требуется. Он обрабатывается на этапе инициализации, когда Gradle определяет, какие файлы сборки в проекте необходимо проанализировать. В случае сборки сразу нескольких проектов файл настроек показывает, какие подкаталоги в корневом каталоге также являются каталогами проектов. Gradle позволяет использовать общие настройки и зависимости для подпроектов, сделать один подпроект зависимым от другого и даже выполнять сборку подпроектов параллельно. За дополнительной информацией о сборках с несколькими проектами обращайтесь к руководству пользователя Gradle (https://oreil.ly/XG4EN).

Исходный код на Kotlin можно смешивать с исходным кодом на Java в папках *src/main/java* и *src/main/kotlin*, или можно добавить свои собственные исходные файлы, используя свойство `sourceSets` в Gradle. За подробностями обращайтесь к документации Gradle (https://oreil.ly/XG4EN).

Смотри также

Тот же процесс с применением Groovy DSL показан в рецепте 1.7. Там же вы найдете дополнительные сведения о проектах для Android, где в настоящее время Kotlin DSL недоступен для выбора при создании проектов Android.

1.9. Сборка проектов на Kotlin с помощью Gradle

Задача

Собрать проект с кодом на Kotlin, используя Gradle.

Решение

Добавить зависимость от Kotlin JDK на этапе компиляции, помимо плагина Kotlin.

Обсуждение

Примеры в рецептах 1.7 и 1.8 показывают, как добавить плагин поддержки Kotlin для Gradle. Этот рецепт демонстрирует добавление новых возможностей в файл сборки для обработки любого кода на Kotlin в проекте.

Чтобы скомпилировать код Kotlin с помощью Gradle, нужно добавить еще один элемент в блок `dependencies`, как показано в примере 1.14.

Пример 1.14. Файл build.gradle.kts на Kotlin DSL для простого проекта

```
plugins {
    `java-library`                  ❶
    kotlin("jvm") version "1.3.50" ❷
}

repositories {
    jcenter()
}

dependencies {
    implementation(kotlin("stdlib")) ❸
}
```

❶ Добавляет задачи из плагина Java Library

❷ Добавляет плагин Kotlin в Gradle

❸ Добавляет стандартную библиотеку Kotlin в проект

Плагин `java-library` определяет задачи для простых проектов JVM, такие как `build`, `compileJava`, `compileTestJava`, `javadoc`, `jar` и другие.

Раздел `plugins` должен следовать первым, но остальные блоки верхнего уровня (`repositories`, `dependencies` и т. д.) могут располагаться в любом порядке.

Блок `dependencies` добавляет стандартную библиотеку Kotlin на этапе компиляции (чтобы добиться того же эффекта в старой версии Gradle, можно использовать конфигурацию `compile` вместо `implementation`). Блок `repositories` указывает, что зависимость Kotlin будет загружена из `jcenter` – общедоступного репозитория Artifactory Bintray.

Если теперь выполнить команду `gradle build --dry-run`, она перечислит задачи, доступные для выполнения, но не выполнит их:

```
> gradle build -m

:compileKotlin SKIPPED
:compileJava SKIPPED
:processResources SKIPPED
:classes SKIPPED
:inspectClassesForKotlinIC SKIPPED
:jar SKIPPED
:assemble SKIPPED
:compileTestKotlin SKIPPED
:compileTestJava SKIPPED
```

```
:processTestResources SKIPPED
:testClasses SKIPPED
:test SKIPPED
:check SKIPPED
:build SKIPPED

BUILD SUCCESSFUL in 0s
```

Плагин Kotlin добавляет задачи `compileKotlin`, `inspectClassesForKotlinIC` и `compileTestKotlin`.

Собрать проект можно той же командой, опустив параметр `-m`, который является синонимом параметра `--dry-run`.

1.10. Использование Maven с Kotlin

Задача

Скомпилировать код на Kotlin с помощью инструмента сборки Maven.

Решение

Использовать плагин поддержки Kotlin для Maven и добавить стандартную библиотеку в зависимости.

Обсуждение

Основные сведения о Maven можно найти на веб-странице с документацией (https://oreil.ly/LLy3h).

В документации рекомендуется сначала указать версию Kotlin в Maven-файле *pom.xml*, как показано ниже:

```
<properties>
    <kotlin.version>1.3.50</kotlin.version>
</properties>
```

Затем добавить в зависимости стандартную библиотеку Kotlin, как показано в примере 1.15.

Пример 1.15. Добавление стандартной библиотеки Kotlin в зависимости

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-stdlib</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

Так же как в Gradle, можно указать `kotlin-stdlib-jdk7` или `kotlin-stdlib-jdk8`, чтобы использовать функции расширения для Java 1.7 или 1.8 соответственно.

Кроме того, можно использовать артефакты `kotlin-reflect` (поддержка рефлексии) и `kotlin-test` с `kotlin-test-junit` (поддержка тестирования).

Чтобы скомпилировать исходный код на Kotlin, нужно сообщить Maven, в каких каталогах он находится, как показано в примере 1.16.

Пример 1.16. Определение каталогов с исходным кодом на Kotlin

```
<build>
    <sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory>
    <testSourceDirectory>${project.basedir}/src/test/kotlin</testSourceDirectory>
</build>
```

И использовать плагин Kotlin для компиляции и тестирования (пример 1.17).

Пример 1.17. Использование плагина Kotlin

```
<build>
    <plugins>
        <plugin>
            <artifactId>kotlin-maven-plugin</artifactId>
            <groupId>org.jetbrains.kotlin</groupId>
            <version>${kotlin.version}</version>

            <executions>
                <execution>
                    <id>compile</id>
                    <goals><goal>compile</goal></goals>
                </execution>

                <execution>
                    <id>test-compile</id>
                    <goals><goal>test-compile</goal></goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Если проект содержит исходный код на Kotlin и на Java, первым должен компилироваться код на Kotlin. То есть плагин `kotlin-maven-plugin` должен запускаться перед `maven-compiler-plugin`. В упоминавшейся выше документации показано, как это организовать с помощью параметров конфигурации в файле *pom.xml*.

Глава 2

Основы Kotlin

Эта глава содержит рецепты использования основных возможностей Kotlin. Они демонстрируют приемы программирования без использования специализированных библиотек.

2.1. Использование типов с поддержкой значения null

Задача

Гарантировать, что переменная никогда не получит значение `null`.

Решение

Определить переменную с типом без вопросительного знака в имени. Типы с поддержкой значения `null` также могут использоваться с оператором безопасного вызова (`?.`) и с оператором «Элвис» (`?:`).

Обсуждение

Наиболее привлекательной, пожалуй, особенностью Kotlin является почти полное устранение значения `null`. Если в Kotlin определить переменную с типом без знака вопроса в конце, как показано в примере 2.1, то компилятор потребует, чтобы она никогда не смогла принять значение `null`.

Пример 2.1. Объявление переменной, не поддерживающей значение null

```
var name: String

// ... позднее ...
name = «Dolly»  ❶
// name = null  ❷
```

❶ Присваивание непустой строки

❷ Попытка присвоить `null` вызовет ошибку компиляции

Объявление переменной `name` с типом `String` означает, что ей нельзя присвоить значение `null`, иначе код не будет компилироваться.

Если желательно, чтобы переменная могла получать значение `null`, добавьте в объявление вопросительный знак после имени типа, как показано в примере 2.2.

Пример 2.2. Объявление переменной, поддерживающей значение null

```
class Person(val first: String,
             val middle: String?,
             val last: String)

val jkRowling = Person("Joanne", null, "Rowling") ❶
val northWest = Person("North", null, "West")     ❷
```

❶ Джоан Роулинг (JK Rowling) не имеет второго имени; она выбрала букву K в качестве инициала в честь своей бабушки Кэтрин (Katherine)

❷ У детей Ким (Kim) и Канье (Kanye) будут проблемы посерьезнее

В этом классе Person все равно придется указать значение для второго параметра, даже если оно пустое.

Жизнь становится интереснее, когда переменные с поддержкой null начинают использоваться в выражениях. Kotlin требует проверять неравенство переменной значению null, но это не так просто, как кажется. Например, рассмотрим одну такую проверку (пример 2.3).

Пример 2.3. Проверка на неравенство val-переменной значению null

```
val p = Person(first = "North", middle = null, last = "West")
if (p.middle != null) {
    val middleNameLength = p.middle.length ❶
}
```

❶ Интеллектуальное приведение к типу String, не поддерживающему значение null

Оператор if проверит неравенство свойства middle значению null, и если это так, то Kotlin выполнит интеллектуальное приведение типа: он будет обрабатывать свойство p.middle, как если бы оно имело тип String, а не String?. Этот прием работает, но только потому, что переменная p была объявлена с ключевым словом val и не может измениться после инициализации. Если переменную объявить как var, то код должен выглядеть, как показано в примере 2.4.

Пример 2.4. Проверка на неравенство значению null var-переменной

```
var p = Person(first = "North", middle = null, last = "West")

if (p.middle != null) {
    // val middleNameLength = p.middle.length ❶
    val middleNameLength = p.middle!!.length  ❷
}
```

❶ Интеллектуальное приведение к типу String невозможно, потому что p.middle – сложное выражение

❷ Проверка на неравенство null (не делайте этого без крайней необходимости)

Поскольку теперь p объявлена как var, а не val, Kotlin предположит, что она может измениться между моментом ее определения и моментом обращения к свойству middle, и откажется выполнить интеллектуальное приведение типа. Одно из возможных решений – использовать оператор проверки (!!), что считается дурным тоном. Оператор !! заставляет компилятор интерпретировать переменную как имеющую значение, отличное от null, и генерировать исклю-

чение, если это не так. Это один из немногих способов получить исключение NullPointerException в коде на Kotlin, поэтому старайтесь его избегать.

Более удачное решение – использовать оператор *безопасного вызова* (?.). Этот оператор возвращает null, если значение слева от него равно null, как показано в примере 2.5.

Пример 2.5. Использование оператора безопасного вызова

```
var p = Person(first = "North", middle = null, last = "West")
val middleNameLength = p.middle?.length ❶
```

❶ Безопасный вызов; возвращает значение типа Int?

Проблема в том, что этот оператор возвращает тип, который также допускает значение null, поэтому middleNameLength получит тип Int?, что, скорее всего, не то, что вам нужно. Поэтому иногда полезно комбинировать оператор безопасного вызова с оператором «Элвис» (?:), как показано в примере 2.6.

Пример 2.6. Комбинирование оператора безопасного вызова с оператором «Элвис»

```
var p = Person(first = "North", middle = null, last = "West")
val middleNameLength = p.middle?.length ?: 0 ❶
```

❶ Оператор «Элвис» вернет 0, если middle будет иметь значение null

Оператор «Элвис» проверит значение выражения слева и, если оно не равно null, вернет его. Иначе он вернет значение выражения справа. В данном случае он проверит значение выражения p.middle?.length, которое может вернуть целое число или null. Если выражение вернет целое число, то оператор «Элвис» вернет его, иначе – значение выражения 0.

Справа от оператора «Элвис» может находиться выражение, что позволяет использовать return или throw при проверке аргументов функции.

Самая большая сложность – разглядеть в операторе ?: Элвиса Пресли, наклонив голову влево. Очевидно, что Kotlin создавался для разработчиков с богатым воображением[4].

Наконец, в Kotlin имеется оператор as? *безопасного приведения типа*. Он добавлен с целью избежать исключения ClassCastException, если приведение типа невозможно. В примере 2.7 показано, как безопасно привести экземпляр Person к этому типу, если он может хранить пустую ссылку null.

Пример 2.7. Оператор безопасного приведения типа

```
val p1 = p as? Person ❶
```

❶ Переменная p1 получит тип Person?

Приведение либо выполнится успешно – и в результате получится экземпляр Person, либо завершится ошибкой – и p1 получит значение null.

[4] Это в большей степени относится к Groovy. На самом деле оператор «Элвис» был заимствован из Groovy.

2.2. Добавление признака поддержки null в Java

Задача

Код на Kotlin должен взаимодействовать с кодом на Java, и требуется, чтобы он поддерживал значения `null`.

Решение

Добавить поддержку аннотаций JSR-305 в код на Kotlin, используя параметр компиляции `-Xjsr305=strict`.

Обсуждение

Одна из основных особенностей Kotlin – разделение типов с поддержкой и без поддержки значения `null`. Если объявить переменную с типом `String`, она никогда не сможет иметь значение `null`, а если объявить ее с типом `String?`, то сможет, как показано в примере 2.8.

Пример 2.8. Типы с поддержкой и без поддержки null

```
var s: String = "Hello, World!" ❶
var t: String? = null           ❷
```

❶ Не может иметь значение `null`, код, присваивающий `null` этой переменной, просто не скомпилируется

❷ Знак вопроса в имени типа указывает на поддержку значения `null`

Это правило не вызывает трудностей, пока не появляется необходимость взаимодействовать с кодом на Java, который не имеет такого разделения типов. Однако в Java есть аннотация `@Nonnull`, которая определена в пакете *javax.annotation*. В настоящее время эта спецификация считается бездействующей, но во многих библиотеках есть так называемые аннотации, *совместимые с JSR-305*, и Kotlin их поддерживает.

Например, совместимость с Spring Framework можно обеспечить, добавив в файл сборки Gradle код из примера 2.9.

Пример 2.9. Обеспечение совместимости с JSR-305 в Gradle (Groovy DSL)

```
sourceCompatibility = 1.8
compileKotlin {
    kotlinOptions {
        jvmTarget = «1.8»
        freeCompilerArgs = [«-Xjsr305=strict»]
    }
}
compileTestKotlin {
    kotlinOptions {
        jvmTarget = «1.8»
        freeCompilerArgs = [«-Xjsr305=strict»]
    }
}
```

То же самое можно выразить на Kotlin DSL, как показано в примере 2.10.

Пример 2.10. Обеспечение совместимости с JSR-305 в Gradle (Kotlin DSL)

```
tasks.withType<KotlinCompile> {
    kotlinOptions {
        jvmTarget = «1.8»
        freeCompilerArgs = listOf(“-Xjsr305=strict”)
    }
}
```

Использующие Maven могут добавить фрагмент из примера 2.11 в файл POM, как рекомендовано в справочном руководстве по Kotlin.

Пример 2.11. Обеспечение совместимости с JSR-305 в Maven

```
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <executions>...</executions>
    <configuration>
        <nowarn>true</nowarn> <!-- Отключить предупреждения -->
        <args>
            <!-- Включить строгий режим для аннотаций JSR-305 -->
            <arg>-Xjsr305=strict</arg>
            ...
        </args>
    </configuration>
</plugin>
```

Аннотация `@Nonnull`, которая определена в JSR-305, имеет свойство `when`. Если ему присвоить значение `When.ALWAYS`, аннотированный тип будет обрабатываться как не поддерживающий значение `null`. Если присвоить `When.MAYBE` или `When.NEVER`, то будет считаться, что тип поддерживает значение `null`. Если присвоить `When.UNKNOWN`, то будет предполагаться, что тип является *платформенным* типом, для которого допустимость значения `null` неизвестна.

2.3. Добавление перегруженных методов для вызова из Java

Задача

Имеется функция на Kotlin с параметрами по умолчанию. Обеспечить возможность ее вызова из Java без необходимости явно указывать значения для всех параметров.

Решение

Добавить к функции аннотацию `@JvmOverloads`.

Обсуждение

Пусть имеется функция на Kotlin, которая определяет значения по умолчанию для одного или нескольких параметров, как показано в примере 2.12.

Пример 2.12. Функция на Kotlin с параметрами по умолчанию

```
fun addProduct(name: String, price: Double = 0.0, desc: String? = null) =
    "Adding product with $name, ${desc ?: "None" }, and " +
            NumberFormat.getCurrencyInstance().format(price)
```

Функция `addProduct` требует обязательно передавать ей строковое имя (`name`), но описание (`desc`) и цена (`price`) имеют значения по умолчанию. Описание допускает значение `null` и по умолчанию равно `null`, а цена по умолчанию равна 0.

Как показывает тест в примере 2.13, эту функцию легко можно вызвать из Kotlin с одним, двумя или тремя аргументами.

Пример 2.13. Вызов функции с параметрами по умолчанию из Kotlin

```
@Test
fun `check all overloads`() {
    assertAll("Overloads called from Kotlin",
        { println(addProduct("Name", 5.0, "Desc")) },
        { println(addProduct("Name", 5.0)) },
        { println(addProduct("Name")) }
    )
}
```

Каждый последующий вызов `addProduct` передает на один аргумент меньше, чем предыдущий.

Необязательные свойства или свойства с поддержкой значения `null` следует размещать в конце сигнатуры функции, чтобы их можно было не учитывать при вызове функции с позиционными аргументами.

Все вызовы выполняются успешно.

Однако Java не поддерживает параметры по умолчанию, поэтому, чтобы вызвать эту функцию из Java, придется передать все аргументы, как показано в примере 2.14.

Пример 2.14. Вызов функции из Java

```
@Test
void supplyAllArguments() {
    System.out.println(OverloadsKt.addProduct("Name", 5.0, "Desc"));
}
```

Если перед функцией добавить аннотацию `@JvmOverloads`, сгенерированный класс будет включать все необходимые перегруженные версии функции, как показано в примере 2.15.

Пример 2.15. Вызов всех перегруженных версий функции из Java

```
@Test
void checkOverloads() {
    assertAll("overloads called from Java",
        () -> System.out.println(OverloadsKt.addProduct("Name", 5.0, "Desc")),
        () -> System.out.println(OverloadsKt.addProduct("Name", 5.0)),
        () -> System.out.println(OverloadsKt.addProduct("Name"))
    );
}
```

Чтобы понять, как это работает, можно декомпилировать байт-код, сгенерированный компилятором Kotlin. В примере 2.16 показан код, сгенерированный без аннотации `@JvmOverloads`.

Пример 2.16. Декомпилированный байт-код функции, сгенерированный компилятором Kotlin

```
@NotNull
public static final String addProduct(@NotNull String name,
    double price, @Nullable String desc) {
    Intrinsics.checkParameterIsNotNull(name, “name”);

    // ...
}
```

А в примере 2.17 показан код, сгенерированный с аннотацией `@JvmOverloads`.

Пример 2.17. Декомпилированный байт-код функции с перегруженными версиями

```
// public final class OverloadsKt {
@JvmOverloads
@NotNull
public static final String addProduct(@NotNull String name,
    double price, @Nullable String desc) {
    Intrinsics.checkParameterIsNotNull(name, “name”);

    // ...
}

@JvmOverloads
@NotNull
public static final String addProduct(
    @NotNull String name, double price) {
    return addProduct$default(name, price,
        (String)null, 4, (Object)null);
}

@JvmOverloads
@NotNull
public static final String addProduct(@NotNull String name) {
    return addProduct$default(name, 0.0D,
        (String)null, 6, (Object)null);
}
```

Сгенерированный класс включает дополнительные методы, которые вызывают полную версию метода и передают аргументы со значениями по умолчанию.

Аналогичный прием можно применять к конструкторам. Класс `Product` в примере 2.18 генерирует три конструктора: один принимает все три аргумента, один – только имя и цену, и еще один – только имя.

Пример 2.18. Класс на Kotlin с перегруженными конструкторами

```
data class Product @JvmOverloads constructor(
    val name: String,
    val price: Double = 0.0,
    val desc: String? = null
)
```

Чтобы добавить аннотацию `@JvmOverloads`, необходимо явно вызвать `constructor`. Теперь создать экземпляр класса можно тремя разными способами, как показано в примере 2.19.

Пример 2.19. Создание экземпляра класса Product из Kotlin

```
@Test
internal fun `check overloaded Product contructor`() {
    assertAll("Overloads called from Kotlin",
        { println(Product("Name", 5.0, "Desc")) },
        { println(Product("Name", 5.0)) },
        { println(Product("Name")) }
    )
}
```

Все эти конструкторы также можно вызвать из Java, как показано в примере 2.20.

Пример 2.20. Создание экземпляра класса Product из Java

```
@Test
void checkOverloadedProductCtor() {
    assertAll("overloads called from Java",
        () -> System.out.println(new Product("Name", 5.0, "Desc")),
        () -> System.out.println(new Product("Name", 5.0)),
        () -> System.out.println(new Product("Name"))
    );
}
```

Этот прием работает, но имеет одну интересную тонкость. Если взглянуть на декомпилированный код класса `Product`, можно увидеть все необходимые конструкторы (пример 2.21).

Пример 2.21. Перегруженные конструкторы Product в декомпилированном коде

```
@JvmOverloads
public Product(@NotNull String name, double price,
    @Nullable String desc) {
    Intrinsics.checkParameterIsNotNull(name, "name");
    super();
    this.name = name;
    this.price = price;
    this.desc = desc;
}

@JvmOverloads
public Product(String var1, double var2, String var4,
    int var5, DefaultConstructorMarker var6) {

    // ...

    this(var1, var2, var4); ❶
}

@JvmOverloads
public Product(@NotNull String name, double price) {
    this(name, price, (String)null, 4, (DefaultConstructorMarker)null); ❷
}
```

```
@JvmOverloads
public Product(@NotNull String name) {
    this(name, 0.0D, (String)null, 6, (DefaultConstructorMarker)null); ❷
}
```

❶ Вызов конструктора с тремя аргументами

❷ Вызов сгенерированного конструктора, который, в свою очередь, вызывает конструктор с тремя аргументами

Все перегруженные конструкторы в конечном итоге вызывают полную версию с тремя аргументами, подставляя значения по умолчанию. Это нормально, но имейте в виду, что вызов конструктора с необязательными аргументами не вызывает аналогичный конструктор в суперклассе; все вызовы следуют через один конструктор с наибольшим количеством аргументов.

Вызов конструкторов с аннотацией `@JvmOverloads` не приводит к вызову `super` с тем же количеством аргументов. В действительности вызывается полный конструктор с подставленными значениями по умолчанию.

В Java каждый конструктор вызывает конструктор родительского класса с помощью `super`, и в перегруженных конструкторах часто вызывается `super` с тем же количеством аргументов. В данном случае этого не происходит: вызывается конструктор суперкласса со всеми параметрами и подставленными значениями по умолчанию.

2.4. Явное преобразование типов

Задача

Kotlin не преобразует автоматически простые типы в более широкие переменные, например `Int` в `Long`.

Решение

Использовать функции явного преобразования типов, такие как `toInt`, `toLong` и т. д.

Обсуждение

Один из сюрпризов, который Kotlin преподносит разработчикам на Java, – короткие типы не преобразуются автоматически в более длинные. Например, в Java совершенно нормально писать код, как показано в примере 2.22.

Пример 2.22. Автоматическое преобразование коротких типов в более длинные в Java

```
int myInt = 3;
long myLong = myInt; ❶
```

❶ Тип `int` автоматически преобразуется в `long`

Появившаяся в Java 1.5 автоматическая упаковка облегчила преобразование простых типов в типы-обертки, но преобразование из одного типа-обертки

в другой по-прежнему требует дополнительного кода, как показано в примере 2.23.

Пример 2.23. Преобразование типа Integer в тип Long

```
Integer myInteger = 3;
// Long myWrappedLong = myInteger;          ❶
Long myWrappedLong = myInteger.longValue(); ❷
myWrappedLong = Long.valueOf(myInteger);    ❸
```

❶ Не компилируется

❷ Извлекает значение типа `long` и затем обертывает его

❸ Распаковывает значение типа `int`, преобразует в тип `long` и затем обертывает его

Иначе говоря, работа с типами-обертками напрямую требует выполнять распаковку вручную. Нельзя просто присвоить экземпляр типа `Integer` переменной типа `Long`, не распаковав обернутое значение.

В Kotlin простые типы не поддерживаются напрямую. Байт-код может генерировать их эквиваленты, но, работая над своим кодом, вы должны помнить, что имеете дело с классами, а не с примитивами.

К счастью, Kotlin предлагает методы преобразования, такие как `toInt`, `toLong` и т. д., как показано в примере 2.24.

Пример 2.24. Преобразование типа Int в тип Long в Kotlin

```
val intVar: Int = 3
// val longVar: Long = intVar       ❶
val longVar: Long = intVar.toLong() ❷
```

❶ Не компилируется

❷ Явное преобразование типа

Поскольку в Kotlin `intVar` и `longVar` являются экземплярами классов, невозможность автоматически преобразовать экземпляр `Int` в тип `Long` не выглядит удивительной. Но об этом легко забыть, особенно если есть опыт работы с Java.

Вот некоторые из доступных методов преобразования:

- `toByte: Byte`;
- `toChar: Char`;
- `toShort: Short`;
- `toInt(): Int`;
- `toLong(): Long`;
- `toFloat(): Float`;
- `toDouble(): Double`.

К счастью, Kotlin позволяет использовать преимущества перегрузки операторов для прозрачного преобразования типов, поэтому следующий код не требует явного преобразования:

```
val longSum = 3L + intVar
```

Оператор `plus` автоматически преобразует значение `intVar` в тип `long` и сложит его с литералом типа `long`.

2.5. Вывод чисел в разных системах счисления

Задача

Вывести число в системе счисления, отличной от десятичной.

Решение

Использовать функцию-расширение `toString(radix: Int)` с допустимым значением `radix`.

Обсуждение

Этот рецепт предназначен для особых ситуаций, которые возникают нечасто. Однако описываемая в нем возможность представляет определенный интерес и может пригодиться в программах, использующих разные системы счисления.

Есть одна старая шутка:

```
Люди делятся на 10 типов
Которые знают двоичное счисление, и которые не знают его
```

В Java, чтобы вывести число в двоичной системе счисления, нужно использовать статический метод `Integer.toBinaryString` или `Integer.toString(int, int)`. В первом аргументе передается число для вывода, а во втором – основание системы счисления.

Kotlin взял статический метод из Java и превратил его в функцию-расширение `toString (radix: Int)` для типов `Byte`, `Short`, `Int` и `Long`. Например, в примере 2.25 показано, как в Kotlin преобразовать число 42 в строку с двоичным представлением.

Пример 2.25. Вывод числа 42 в двоичном представлении

```
42.toString(2) == “101010”
```

В двоичном представлении биты, справа налево, имеют значения 1, 2, 4, 8, 16 и т. д. Так как 42 – это 2 + 8 + 32, то биты в соответствующих позициях имеют значение 1, а остальные – значение 0.

Вот как выглядит реализация метода `toString` в классе `Int`:

```
public actual inline fun Int.toString(radix: Int): String =
    java.lang.Integer.toString(this, checkRadix(radix))
```

То есть функция-расширение в `Int` вызывает соответствующий статический метод класса `java.lang.Integer`, предварительно проверив второй аргумент `radix`.

Ключевое слово `actual` отмечает, что реализация зависит от платформы.

Метод checkRadix проверяет, находится ли указанное основание системы счисления в диапазоне от Character.MIN_RADIX до Character.MAX_RADIX (в данном случае подразумевается реализация для Java), и если это не так, то возбуждает исключение IllegalArgumentException. Допустимыми минимальным и максимальным значениями являются 2 и 36 соответственно. Пример демонстрирует вывод строковых представлений числа 42 во всех допустимых системах счисления.

Пример 2.26. Вывод строковых представлений числа 42 во всех допустимых системах счисления

```
(Character.MIN_RADIX..Character.MAX_RADIX).forEach { radix ->
    println("$radix: ${42.toString(radix)}")
}
```

Этот пример выведет следующий текст (здесь он немного отформатирован):

```
Radix Value
 2:  101010
 3:    1120
 4:     222
 5:     132
 6:     110
 7:      60
 8:      52
 9:      46
10:      42
...
32:      1a
33:      19
34:      18
35:      17
36:      16
```

Число 42 – это «ответ на главный вопрос жизни, Вселенной и всего сущего» (по крайней мере, согласно Дугласу Адамсу в его серии «Автостопом по галактике»).

Объединение этой возможности с поддержкой многострочного текста дает Kotlin-версию оригинальной шутки (см. пример 2.27).

Пример 2.27. Усовершенствованная версия шутки о знании двоичной системы счисления

```
val joke = """
    Люди делятся на ${3.toString(3)} типов
    Которые знают двоичное счисление, и которые не знают его,
    А для тех, кто не заметил, -- это троичная шутка
""".trimIndent()
println(joke)
```

Этот код выведет следующий текст:

```
Люди делятся на 10 типов
Которые знают двоичное счисление, и которые не знают его,
А для тех, кто не заметил, -- это троичная шутка
```

2.6. Возведение числа в степень

Задача

Возвести число в степень, но обратите внимание, что в Kotlin нет предопределенного оператора возведения в степень.

Решение

Определить инфиксную функцию, которая вызывает уже имеющуюся функцию-расширение `pow` в классах `Int` и `Long`.

Обсуждение

В Kotlin, так же как в Java, нет встроенного оператора возведения в степень. В Java имеется статическая функция `pow` в классе `java.lang.Math` со следующей сигнатурой:

```
public static double Math.pow(double a, double b)
```

Поскольку в Java поддерживается автоматическое расширение простых типов (например, `int` в `double`), достаточно только одной этой функции. В Kotlin, однако, простые типы не поддерживаются непосредственно, а экземпляры классов, таких как `Int`, не преобразуются автоматически в экземпляры `Long` или `Double`. Это вызывает особое раздражение, когда вы обнаруживаете, что стандартная библиотека Kotlin определяет функции-расширения `pow` только для `Float` и `Double`.

Эти функции-расширения имеют следующие сигнатуры:

```
fun Double.pow(x: Double): Double
fun Float.pow(x: Float): Float
```

То есть, чтобы возвести в степень целое число, нужно преобразовать его в тип `Float` или `Double`, вызвать `pow` и затем преобразовать результат обратно в целое число, как показано в примере 2.28.

Пример 2.28. Возведение в степень значения типа Int

```
@Test
fun `raise an Int to a power`() {
    assertThat(256, equalTo(2.toDouble().pow(8).toInt()))
}
```

Для возведения в степень 2 идеально подходят функции `shl` и `shr`, как показано в рецепте 2.7.

Это вполне работоспособное решение, но его можно автоматизировать, определив в классах `Int` и `Long` функции-расширения со следующими сигнатурами:

```
fun Int.pow(x: Int) = toDouble().pow(x).toInt()
fun Long.pow(x: Int) = toDouble().pow(x).toLong()
```

Но еще лучше определить инфиксный оператор. Для перегрузки можно использовать только ограниченный круг символов операторов, но эту проблему

можно преодолеть, заключив оператор в обратные кавычки, как показано в примере 2.29.

Пример 2.29. Определение инфиксного оператора возведения в степень

```
import kotlin.math.pow

infix fun Int.`**`(x: Int) = toDouble().pow(x).toInt()
infix fun Long.`**`(x: Int) = toDouble().pow(x).toLong()
infix fun Float.`**`(x: Int) = pow(x)
infix fun Double.`**`(x: Int) = pow(x)

// Шаблон, напоминающий существующие функции в Float и Double
fun Int.pow(x: Int) = `**`(x)
fun Long.pow(x: Int) = `**`(x)
```


В определении ** было использовано ключевое слово infix, но функции-расширения pow для Int и Long определены без него, чтобы сохранить шаблон, реализованный в Float и Double.

Теперь символ ** можно использовать в роли оператора возведения в степень, как показано в примере 2.30.

Пример 2.30. Использование функции-расширения **

```
@Test
fun `raise to power`() {
    assertAll(
        { assertThat(1, equalTo(2 `**` 0)) },
        { assertThat(2, equalTo(2 `**` 1)) },
        { assertThat(4, equalTo(2 `**` 2)) },
        { assertThat(8, equalTo(2 `**` 3)) },

        { assertThat(1L, equalTo(2L `**` 0)) },
        { assertThat(2L, equalTo(2L `**` 1)) },
        { assertThat(4L, equalTo(2L `**` 2)) },
        { assertThat(8L, equalTo(2L `**` 3)) },

        { assertThat(1F, equalTo(2F `**` 0)) },
        { assertThat(2F, equalTo(2F `**` 1)) },
        { assertThat(4F, equalTo(2F `**` 2)) },
        { assertThat(8F, equalTo(2F `**` 3)) },

        { assertThat(1.0, closeTo(2.0 `**` 0, 1e-6)) },
        { assertThat(2.0, closeTo(2.0 `**` 1, 1e-6)) },
        { assertThat(4.0, closeTo(2.0 `**` 2, 1e-6)) },
        { assertThat(8.0, closeTo(2.0 `**` 3, 1e-6)) },

        { assertThat(1, equalTo(2.pow(0))) },
        { assertThat(2, equalTo(2.pow(1))) },
        { assertThat(4, equalTo(2.pow(2))) },
        { assertThat(8, equalTo(2.pow(3))) },

        { assertThat(1L, equalTo(2L.pow(0))) },
        { assertThat(2L, equalTo(2L.pow(1))) },
        { assertThat(4L, equalTo(2L.pow(2))) },
        { assertThat(8L, equalTo(2L.pow(3))) }
    )
}
```

В тестах с `Double.**` используется функция сравнения `closeTo` из библиотеки Hamcrest, помогающая проверить равенство значений с плавающей точкой. В тестах с `Float`, вероятно, стоило бы сделать то же самое, но в настоящее время тесты успешно выполняются в текущем их виде.

Идея определения инфиксной функции для этой цели была предложена в ответе Оливии Зои (Olivia Zoe) на вопрос на сайте Stack Overflow (https://oreil.ly/1go1V).

Если вам не нравится заключать оператор «звездочка-звездочка» в обратные кавычки, попробуйте определить функцию с другим именем, например `exp`.

2.7. Операторы поразрядного сдвига

Задача

Выполнить операцию поразрядного сдвига.

Решение

Для этой цели в Kotlin имеются поразрядные инфиксные функции, такие как `shr`, `shl` и `ushr`.

Обсуждение

Поразрядные операции имеют широкий круг применения, например для управления списками доступа, в протоколах связи, в алгоритмах сжатия и шифрования и в компьютерной графике. В отличие от многих других языков, Kotlin не использует предопределенные символы операторов для обозначения операций сдвига, а определяет их как функции.

В Kotlin определены следующие операции сдвига в виде функций-расширений в `Int` и `Long`:

`shl`
: сдвиг влево со знаком;

`shr`
: сдвиг вправо со знаком;

`ushr`
: сдвиг вправо без знака.

Согласно правилам арифметики дополнения до двух, поразрядный сдвиг влево или вправо подобен умножению или делению на 2, как показано в примере 2.31.

Пример 2.31. Умножение и деление на 2

```
@Test
fun `doubling and halving`() {
    assertAll("left shifts doubling from 1",     // 0000_0001
        { assertThat(  2, equalTo(1 shl 1)) },   // 0000_0010
        { assertThat(  4, equalTo(1 shl 2)) },   // 0000_0100
        { assertThat(  8, equalTo(1 shl 3)) },   // 0000_1000
        { assertThat( 16, equalTo(1 shl 4)) },   // 0001_0000
        { assertThat( 32, equalTo(1 shl 5)) },   // 0010_0000
        { assertThat( 64, equalTo(1 shl 6)) },   // 0100_0000
        { assertThat(128, equalTo(1 shl 7)) }    // 1000_0000
    )

    assertAll("right shifts halving from 235",   // 1110_1011
        { assertThat(117, equalTo(235 shr 1)) }, // 0111_0101
        { assertThat( 58, equalTo(235 shr 2)) }, // 0011_1010
        { assertThat( 29, equalTo(235 shr 3)) }, // 0001_1101
        { assertThat( 14, equalTo(235 shr 4)) }, // 0000_1110
        { assertThat(  7, equalTo(235 shr 5)) }, // 0000_0111
        { assertThat(  3, equalTo(235 shr 6)) }, // 0000_0011
        { assertThat(  1, equalTo(235 shr 7)) }  // 0000_0001
    )
}
```

Функция ushr используется в случаях, когда требуется выполнить сдвиг без сохранения знака. Для положительных значений shr и ushr ведут себя одинаково. Но для отрицательных значений shr дополняет результат единицами слева, благодаря чему он остается отрицательным, как показано в примере 2.32.

Пример 2.32. Использование функций ushr и shr

```
val n1 = 5
val n2 = -5
println(n1.toString(2)) // 0b0101
println(n2.toString(2)) // -0b0101

assertThat(n1  shr 1, equalTo(0b0010))      // 2
assertThat(n1 ushr 1, equalTo(0b0010))      // 2

assertThat(n2  shr 1, equalTo(-0b0011))     // -3
assertThat(n2 ushr 1, equalTo(0x7fff_fffd)) // 2_147_483_645
```

Кажущееся странным поведение последнего примера обусловлено действием правил арифметики дополнения до двух. Так как ushr заполняет результат слева нулями, она не сохраняет отрицательный знак -3. Результатом является дополнение до двух 32-битного целого числа -3, как показано в комментарии.

Функция ushr встречается во многих местах. Один интересный пример возникает при попытке найти середину двух больших целых чисел, как в примере 2.33.

Функция ushr имеет массу применений. Один из интересных примеров – поиск середины между двумя большими целыми значениями, как показано в примере ниже.

Пример 2.33. Поиск середины между двумя большими целыми значениями

```
val high = (0.99 * Int.MAX_VALUE).toInt()
val low = (0.75 * Int.MAX_VALUE).toInt()

val mid1 = (high + low) / 2    ❶
val mid2 = (high + low) ushr 1 ❷

assertTrue(mid1 !in low..high)
assertTrue(mid2  in low..high)
```

❶ Сумма больше максимально возможного значения `Int`, поэтому результат получится отрицательным

❷ Сдвиг вправо без знака гарантирует, что результат будет находиться в середине указанного диапазона

Если оба значения достаточно большие, то их сумма окажется больше, чем `Int.MAX_VALUE`, и в результате получится отрицательное число. Если деление на 2 выполнить функцией сдвига вправо, то результат будет находиться в середине указанного диапазона.

Вычисление среднего значения двух целых чисел, каждое из которых может оказаться очень большим, требуется во многих алгоритмах, таких как бинарный поиск или сортировка. Использование `ushr` в подобных алгоритмах гарантирует, что результат всегда будет находиться в требуемом диапазоне.

2.8. Использование поразрядных операторов

Задача

Применить маску к битовым значениям.

Решение

Использовать поразрядные операторы `and`, `or`, `xor` и `inv`, поддерживаемые в Kotlin.

Обсуждение

В дополнение к операторам сдвига, определенным в классах `Int` и `Long`, Kotlin поддерживает также операции маскирования: `and`, `or`, `xor` и `inv` (не «not»).

Последняя функция в этом списке – `inv` – переворачивает все биты в числе. Возьмем для примера число 4, которое в двоичном представлении имеет вид `0b00000100`. Если перевернуть все биты, получится число `0b11111011`, которое в десятичном представлении имеет вид 251. Однако если вызвать функцию `inv` с числом 4, то вы получите в результате –5, как показано в примере 2.34.

Литералы, начинающиеся с префикса `0b`, выражают значения в двоичном представлении.

Пример 2.34. Инверсия числа 4

```
// 4 == 0b0000_0100 (в двоичном представлении)
// Поразрядное дополнение (смена значений всех битов) дает:
//     0b1111_1011 == 251 (в десятичном представлении)
assertEquals(-5, 4.inv())
```

Для разделения групп разрядов в числовых литералах можно использовать символы подчеркивания (_). Компилятор игнорирует их.

Но почему вместо 251 получается –5? Дело в том, что система выполняет арифметические действия с дополнением до двух. Дополнение до двух для любого целого числа `n` задается как `-(~n + 1)`, где `~n` – это дополнение числа `n` до единицы (то есть смена значений всех битов на противоположные). Следовательно:

```
0b1111_1011 -> -(0b0000_0100 + 1) -> -0b0000_0101 -> -5
```

Дополнением до двух для числа 251 является число –5.

Поразрядные операторы `and`, `or` и `xor` хорошо знакомы большинству разработчиков. Единственная разница между ними и их логическими аналогами в том, что они не поддерживают вычисления по короткой схеме. Рассмотрим пример 2.35.

Пример 2.35. Использование операторов and, or и xor

```
@Test
fun `and, or, xor`() {
    val n1 = 0b0000_1100 // десятичное 12
    val n2 = 0b0001_1001 // десятичное 25

    val n1_and_n2 = n1 and n2
    val n1_or_n2 = n1 or n2
    val n1_xor_n2 = n1 xor n2

    assertThat(n1_and_n2, equalTo(0b0000_1000)) //  8
    assertThat(n1_or_n2, equalTo(0b0001_1101))  // 29
    assertThat(n1_xor_n2, equalTo(0b0001_0101)) // 21
}
```

В качестве более интересного примера рассмотрим модель RGBA представления цветов, реализованную в классе `java.awt.Color` в Java. Цвет можно представить как 4-байтное целое число, где по 1 байту отводится для значения красного, зеленого и синего каналов, а также альфа-канала (определяющего степень прозрачности). Взгляните на рис. 2.1.

АЛЬФА	КРАСНЫЙ	ЗЕЛЕНЫЙ	СИНИЙ
0b1111 - 0b0000	0b1111 - 0b0000	0b1111 - 0b0000	0b1111 - 0b0000

Рис. 2.1. 32-битное целое, в котором для каждого канала отводится по 1 байту

В описании метода getRGB класса Color говорится, что по умолчанию он возвращает int со «значением RGB в цветовой модели sRGB (биты 24–31 – альфа-канал, 16–23 – канал красного цвета, 8–15 – канал зеленого цвета, 0–7 – канал синего цвета)».

То есть в Kotlin можно извлечь фактические значения каналов RGB и альфа, используя функцию, представленную в примере 2.36.

Пример 2.36. Преобразование целого числа в значения RGB

```
fun intsFromColor(color: Color): List<Int> {
    val rgb   = color.rgb           ❶
    val alpha = rgb shr 24 and 0xff ❷
    val red   = rgb shr 16 and 0xff ❷
    val green = rgb shr  8 and 0xff ❷
    val blue  = rgb and        0xff ❷
    return listOf(alpha, red, green, blue)
}
```

❶ Вызов Java-метода getRGB

❷ Сдвиг вправо и применение маски для извлечения соответствующего значения Int

Возврат отдельных значений в виде списка позволяет использовать прием деструктуризации, как показано в примере 2.37.

Пример 2.37. Деструктуризация и тестирование

```
@Test
fun `colors as ints`() {
    val color = Color.MAGENTA
    val (a, r, g, b) = intsFromColor(color)

    assertThat(color.alpha, equalTo(a))
    assertThat(color.red,   equalTo(r))
    assertThat(color.green, equalTo(g))
    assertThat(color.blue,  equalTo(b))
}
```

Точно так же можно реализовать обратное преобразование значений RGB в целое число Int, как показано в примере 2.38.

Пример 2.38. Создание Int из значений RGB и альфа-канала

```
fun colorFromInts(alpha: Int, red: Int, green: Int, blue: Int) =
    (alpha and 0xff shl 24) or
    (red   and 0xff shl 16) or
    (green and 0xff shl  8) or
    (blue  and 0xff)
```

На этот раз значения сдвигаются влево, а не вправо. Эту функцию тоже легко проверить, как показано в примере 2.39.

Пример 2.39. Преобразование значений RGB и альфа-канала в Int

```
@Test
fun `ints as colors`() {
    val color = Color.MAGENTA
    val intColor = colorFromInts(color.alpha,
        color.red, color.green, color.blue)
    val color1 = Color(intColor, true) ❶
    assertThat(color1, equalTo(color))
}
```

❶ Второй аргумент конструктора сообщает, что определено значение альфа-канала

В завершение этого рецепта представлю шутку о xor: «э-xor-цист устраняет одного или другого демона, но не обоих». Извините, если кого-то задел, но призываю вас: не стесняйтесь шутить над своими друзьями.

2.9. Создание экземпляров Pair с помощью to

Задача

Создать экземпляры класса Pair (например, для включения в ассоциативный массив).

Решение

Вместо создания экземпляра класса Pair непосредственно используйте инфиксную функцию.

Обсуждение

Ассоциативные массивы состоят из элементов, которые являются комбинациями ключей и значений. Для создания ассоциативного массива в Kotlin имеется несколько функций верхнего уровня, таких как mapOf, которые позволяют создать ассоциативный массив из списка экземпляров Pair. Вот как выглядит сигнатура функции mapOf:

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

Pair – это класс данных, содержащий два элемента с именами first и second. Вот как выглядит объявление класса Pair:

```
data class Pair<out A, out B> : Serializable
```

Свойства first и second класса Pair соответствуют обобщенным значениям A и B.

Создать экземпляр класса Pair можно с помощью конструктора с двумя аргументами, но чаще для этого используется функция to. Функция to определена следующим образом:

```
public infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
```

Функция to просто создает экземпляр класса Pair.

В примере 2.40 показано, как создать ассоциативный массив, используя пары, созданные с помощью функции to.

Пример 2.40. Использование функции to для создания пар в вызове mapOf

```
@Test
fun `create map using infix to function`() {
    val map = mapOf("a" to 1, "b" to 2, "c" to 2) ❶
    assertAll(
        { assertThat(map, hasKey("a")) },
        { assertThat(map, hasKey("b")) },
        { assertThat(map, hasKey("c")) },
        { assertThat(map, hasValue(1)) },
        { assertThat(map, hasValue(2)) })
}

@Test
fun `create a Pair from constructor vs to function`() {
    val p1 = Pair("a", 1)                          ❷
    val p2 = "a" to 1                              ❶

    assertAll(
        { assertThat(p1.first,  `is`("a")) },
        { assertThat(p1.second, `is`(1)) },
        { assertThat(p2.first,  `is`("a")) },
        { assertThat(p2.second, `is`(1)) },
        { assertThat(p1,        `is`(equalTo(p2)))}
    )
}
```

❶ Создание экземпляров Pair с помощью to

❷ Создание экземпляров Pair с помощью конструктора

Функция to – это функция-расширение, добавляемая к любому обобщенному типу A, которая имеет обобщенный аргумент B и возвращает экземпляр Pair, объединяющий значения A и B. Это всего лишь более удобный и компактный способ создания литералов ассоциативных массивов.

Между прочим, поскольку Pair – это класс данных, доступ к его отдельным элементам можно получить с помощью приема деструктуризации, как показано в примере 2.41.

Пример 2.41. Деструктуризация экземпляра Pair

```
@Test
fun `destructuring a Pair`() {
    val pair = "a" to 1
    val (x,y) = pair

    assertThat(x, `is`("a"))
    assertThat(y, `is`(1))
}
```

В стандартной библиотеке имеется также класс Triple, представляющий триаду значений. Однако из-за отсутствия удобных функций-расширений создавать экземпляры Triple можно только непосредственно, вызывая конструктор с тремя аргументами.

Глава 3

Объектно-ориентированное программирование на Kotlin

Как и Java, Kotlin является языком объектно-ориентированного программирования (ООП). Он позволяет определять классы, абстрактные и конкретные, и интерфейсы почти точно так же, как Java.

Некоторые аспекты ООП в Kotlin заслуживают особого внимания, и в этой главе мы познакомимся с ними поближе. Здесь вы найдете рецепты инициализации объектов, реализации пользовательских методов чтения и записи свойств, выполнения поздней и отложенной инициализации, создания синглтонов (объектов-одиночек), применения класса `Nothing` и многие другие.

3.1. Различия между const и val

Задача

Определить значение, которое является константой времени компиляции, а не времени выполнения.

Решение

Использовать модификатор `const` для определения констант времени компиляции. Ключевое слово `val` определяет переменную, которую нельзя изменить после инициализации, но эта инициализация может происходить во время выполнения.

Обсуждение

В Kotlin ключевое слово `val` указывает, что значение переменной нельзя изменить. В Java аналогичную роль играет ключевое слово `final`. Но зачем Kotlin поддерживает еще и модификатор `const`?

Константы времени компиляции должны быть свойствами верхнего уровня, членами объекта или объекта-компаньона. Они должны иметь тип `String` или класса-обертки простого типа (`Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `Char` или `Boolean`) и не могут иметь нестандартный метод чтения. Значения им должны присваиваться вне любой функции, включая `main`, потому что их значения должны быть известны во время компиляции.

Давайте рассмотрим определение минимального и максимального приоритетов для задачи, как показано в примере 3.1.

Пример 3.1. Определение констант времени компиляции

```
class Task(val name: String, _priority: Int = DEFAULT_PRIORITY) {

    companion object {
        const val MIN_PRIORITY = 1          ❶
        const val MAX_PRIORITY = 5          ❶
        const val DEFAULT_PRIORITY = 3      ❶
    }

    var priority = validPriority(_priority) ❷
        set(value) {
            field = validPriority(value)
        }

    private fun validPriority(p: Int) =     ❸
        p.coerceIn(MIN_PRIORITY, MAX_PRIORITY)
}
```

❶ Константы времени компиляции

❷ Свойство с нестандартным методом записи

❸ Приватная функция проверки

В этом примере три константы определены с использованием идиомы, обычной для Kotlin (и Java), согласно которой имена констант записываются заглавными буквами. В этом примере также определен нестандартный метод записи для отображения любого переданного приоритета в заданный диапазон.

Обратите внимание, что в Kotlin `val` – это ключевое слово, а `const` – модификатор, подобный модификаторам `private`, `inline` и т. д. Вот почему `const` следует использовать вместе с ключевым словом `val`, а не вместо него.

Смотри также

Рецепт 3.2, где подробнее рассказывается о методах записи, подобных показанному в этом рецепте.

3.2. Создание нестандартных методов чтения и записи свойств

Задача

Реализовать обработку значения перед присваиванием свойству или возвратом из свойства.

Решение

Добавить к свойству класса функции `get` и `set`.

Обсуждение

Так же как в других объектно-ориентированных языках, классы в Kotlin объединяют данные и функции, которые работают с этими данными, используя метод, широко известный как *инкапсуляция*. Но, в отличие от других языков, в Kotlin все члены класса по умолчанию являются общедоступными. В результате создается впечатление, что язык нарушает принцип сокрытия данных, согласно которому структура данных, связанная с информацией, считается деталью реализации.

Kotlin решает эту дилемму необычным способом: поля нельзя объявлять непосредственно в классах. Это звучит странно, особенно если учесть возможность определения свойств, которые выглядят точно так же, как поля (см. пример 3.2).

Пример 3.2. Класс, представляющий задачу

```
class Task(val name: String) {
    var priority = 3

    // ...
}
```

Класс `Task` определяет два свойства: имя (`name`) и приоритет (`priority`). Первое свойство объявляется в главном конструкторе, а второе является членом класса верхнего уровня. Конечно, оба свойства можно было определить в конструкторе, но в данном случае я хотел показать альтернативный синтаксис. Недостаток такого объявления `priority` – невозможность назначить приоритет при создании экземпляра класса, однако его легко преодолеть с помощью блока `apply`:

```
var myTask = Task().apply { priority = 4 }
```

Преимущество подобного способа определения свойства – возможность добавить свои методы чтения и записи свойства. Вот полный синтаксис определения свойства:

```
var <propertyName>[: <PropertyType>] [= <property_initializer]
    [<getter>]
    [<setter>]
```

Выражение инициализации и методы чтения (getter) и записи (setter) можно опустить. Определение типа тоже можно опустить, если его можно определить из выражения инициализации или из типа, возвращаемого методом чтения. Но все это не относится к свойствам, объявленным в конструкторе.

Объявления свойств в конструкторе обязательно должны включать определение типа, даже если указаны значения по умолчанию.

В примере 3.3 показан метод чтения, вычисляющий значение свойства `isLowPriority`.

Пример 3.3. Метод чтения для свойства

```
val isLowPriority
    get() = priority < 3
```

Как уже говорилось, тип `isLowPriority` определяется автоматически по типу значения, возвращаемого функцией `get`, – в данном случае это логический тип.

Метод записи автоматически вызывается при каждой попытке присвоить значение свойству. Например, чтобы гарантировать, что приоритет находится в диапазоне от 1 до 5, можно реализовать метод записи, как показано в примере 3.4.

Пример 3.4. Метод записи для свойства

```
var priority = 3
    set(value) {
        field = value.coerceIn(1..5)
    }
```

Здесь, наконец, можно видеть решение упомянутой выше дилеммы общедоступных/приватных полей. Обычно, когда объекту требуется вспомогательное поле, Kotlin создает его автоматически. Однако в этом нестандартном методе записи для ссылки на поле, соответствующее свойству, используется идентификатор `field`. Идентификатор `field` можно использовать только в нестандартных методах чтения и записи.

Поле создается для свойства, только если оно применяется в созданном по умолчанию методе чтения или записи или если в нестандартном методе чтения или записи используется ссылка `field`. Это означает, что у вычисляемого свойства `lowPriority` такого поля не будет.

В литературе методы чтения и записи формально называются методами *доступа* и *мутации*, по-видимому, потому что сложно брать плату за консультации, если использовать более простые термины *чтение* и *запись*.

Чтобы завершить этот пример, представьте, что вам потребовалась возможность назначать приоритет в вызове конструктора. Один из способов сделать это – добавить параметр конструктора, который не является свойством, исключив ключевое слово `var` или `val`. В результате получается следующее определение класса `Task`, показанного выше в примере 3.1:

```
class Task(val name: String, _priority: Int = DEFAULT_PRIORITY) {

    companion object {
        const val MIN_PRIORITY = 1
        const val MAX_PRIORITY = 5
        const val DEFAULT_PRIORITY = 3
    }

    var priority = validPriority(_priority)
        set(value) {
            field = validPriority(value)
        }

    private fun validPriority(p: Int) =
        p.coerceIn(MIN_PRIORITY, MAX_PRIORITY)
}
```

Параметр `_priority` – это не свойство, а просто аргумент конструктора. Он используется для инициализации фактического свойства `priority`, а нестандартный метод записи будет приводить его значение в допустимый диапазон при каждой попытке изменения. Обратите внимание, что имя `value` – это всего лишь фиктивное имя; вы можете использовать любое другое, какое вам понравится, как и для параметров любых функций.

Смотри также

Рецепт 3.1, демонстрирующий использование констант.

3.3. Определение классов данных

Задача

Определить класс, представляющий сущность, имеющий полный комплект методов `equals`, `hashCode`, `toString` и т. д.

Решение

Использовать ключевое слово `data` в объявлении класса.

Обсуждение

В Kotlin имеется ключевое слово `data`, позволяющее указать, что данный класс предназначен для хранения данных. В Java такие классы, представляющие информацию, например из таблиц в базе данных, называют *сущностями*. Классы данных в Kotlin имеют аналогичное назначение.

Добавление слова `data` в определение класса заставляет компилятор автоматически сгенерировать целый ряд функций, включая согласованные функции `equals` и `hashCode`, функцию `toString`, которая преобразует класс и значения его свойств в строку, функцию `copy` и функции доступа к компонентам для поддержки деструктуризации.

Например, рассмотрим класс `Product`:

```
data class Product(
    val name: String,
    var price: Double,
    var onSale: Boolean = false
)
```

На основе свойств, объявленных в главном конструкторе, компилятор сгенерирует функции `equals` и `hashCode`, используя алгоритм, аналогичный алгоритму, описанному Джошуа Блохом (Joshua Bloch) много лет назад в книге «Effective Java»[5] (Addison-Wesley Professional). Тесты в примере 3.5 показывают, что они работают правильно.

[5] *Джошуа Блох.* Java. Эффективное программирование. Лори, 2014. ISBN: 978-5-85582-347-9. – *Прим. перев.*

Пример 3.5. Использование сгенерированных реализаций equals и hashCode

```
@Test
fun `check equivalence`() {
    val p1 = Product("baseball", 10.0)
    val p2 = Product("baseball", 10.0, false)

    assertEquals(p1, p2)
    assertEquals(p1.hashCode(), p2.hashCode())
}

@Test
fun `create set to check equals and hashcode`() {
    val p1 = Product("baseball", 10.0)
    val p2 = Product(price = 10.0, onSale = false, name = "baseball")

    val products = setOf(p1, p2)
    assertEquals(1, products.size) ❶
}
```

❶ Дубликат не будет добавлен

Поскольку экземпляры p1 и p2 эквивалентны, то при передаче их в вызов функции setOf она добавит в возвращаемую коллекцию только один из них.

Функция toString преобразует экземпляр Product в строку:

```
Product(name=baseball, price=10.0, onSale=false)
```

Метод copy – это метод экземпляра, который создает новый объект, копируя значения свойств из оригинала и изменяя только указанные свойства, как показано в примере 3.6.

Пример 3.6. Тестирование функции copy

```
@Test
fun `change price using copy`() {
    val p1 = Product("baseball", 10.0)
    val p2 = p1.copy(price = 12.0)      ❶
    assertAll(
        { assertEquals("baseball", p2.name) },
        { assertThat(p2.price, `is`(closeTo(12.0, 0.01))) },
        { assertFalse(p2.onSale) }
    )
}
```

❶ Изменит только цену (price)

Тест проверяет, действительно ли вызов copy с параметром price изменит только это значение. Обратите внимание, что для сравнения используется функция closeTo из библиотеки Hamcrest, потому что проверка равенства значений с плавающей запятой с помощью оператора сравнения – не лучшая идея.

Обратите внимание, что функция copy создает лишь поверхностную копию. Для демонстрации этого поведения определим еще один класс данных с именем OrderItem, как показано в примере 3.7.

Пример 3.7. Класс, содержащий экземпляр Product

```
data class OrderItem(val product: Product, val quantity: Int)
```

Тест в примере 3.8 создает экземпляр `OrderItem` и затем получает его копию с помощью функции `copy`.

Пример 3.8. Тест, демонстрирующий поверхностное поведение функции copy

```
@Test
fun `data copy function is shallow`() {
    val item1 = OrderItem(Product("baseball", 10.0), 5)
    val item2 = item1.copy()

    assertAll(
        { assertTrue(item1 == item2) },
        { assertFalse(item1 === item2) },                ❶

        { assertTrue(item1.product == item2.product) },
        { assertTrue(item1.product === item2.product) } ❷
    )
}
```

❶ Экземпляр `OrderItem`, созданный функцией `copy`, – это другой объект

❷ Свойства `product` в обоих экземплярах `OrderItem` ссылаются на один и тот же объект `Product`

Этот тест показывает, что, несмотря на эквивалентность двух экземпляров `OrderItem` (проверяется с помощью функции `equals`, которая вызывается оператором ==), они являются разными объектами, потому что оператор ссылочного равенства === возвращает `false`. Однако они оба ссылаются на один и тот же экземпляр `Product`, поскольку === для внутренних ссылок возвращает `true`.

Функция `copy` в классах данных выполняет поверхностное копирование, а не глубокое.

Кроме функции `copy`, классы данных автоматически получают функции с именами `component1`, `component2` и т. д., которые возвращают значения свойств. Эти функции используются для деструктуризации, как показано в примере 3.9.

Пример 3.9. Деструктуризация экземпляра Product

```
@Test
fun `destructure using component functions`() {
    val p = Product("baseball", 10.0)

    val (name, price, sale) = p        ❶
    assertAll(
        { assertEquals(p.name, name) },
        { assertThat(p.price, `is`(closeTo(price, 0.01))) },
        { assertFalse(sale) }
    )
}
```

❶ Деструктуризация экземпляра `Product`

Любую из этих функций (`equals`, `hashCode`, `toString`, `copy`, а также любую из функций `_componentN_`) можно переопределить. Также можно добавить другие функции.

Если необходимо, чтобы значения свойств обрабатывались сгенерированными функциями, добавляйте свойства в тело класса, а не в главный конструктор.

Классы данных – это удобный способ определения объектов, главной целью которых является хранение данных. Стандартная библиотека включает два класса данных, `Pair` и `Triple`, для хранения двух и трех свойств любых типов соответственно. Если вам потребуется больше свойств, создайте свой класс данных.

3.4. Прием создания теневого свойства

Задача

Обеспечить возможность управлять инициализацией и чтением общедоступного свойства класса.

Решение

Определить второе свойство того же типа и реализовать свои методы чтения и записи, предоставляющие доступ к требуемому свойству.

Обсуждение

Представьте, что у вас есть класс `Customer`, представляющий клиента, и вам нужно организовать хранение списка его сообщений или заметок. Однако необязательно загружать все сообщения при создании экземпляра, поэтому класс можно определить, как показано в примере 3.10.

Пример 3,10. Класс Customer, версия 1

```
class Customer(val name: String) {
    private var _messages: List<String>? = null ❶

    val messages: List<String>                  ❷
        get() {                                 ❸
            if (_messages == null) {
                _messages = loadMessages()
            }
            return _messages!!
        }

    private fun loadMessages(): MutableList<String> =
        mutableListOf(
            “Initial contact”,
            “Convinced them to use Kotlin”,
            “Sold training class. Sweet.”
        ).also { println(“Loaded messages”) }
}
```

❶ Приватное свойство, поддерживающее значение null и используемое для инициализации

❷ Свойство, при обращении к которому производится загрузка

❸ Приватная функция

Свойство messages в этом классе хранит список сообщений данного клиента. Чтобы избежать его немедленной инициализации, добавляется дополнительное свойство _messages того же типа, но с поддержкой значения null. Нестандартный метод чтения загружает сообщения, если прежде они не были загружены. Тест в примере 3.11 проверяет загрузку сообщений.

Пример 3.11. Проверка загрузки сообщений в объекте, представляющем клиента

```
@Test
fun `load messages`() {
    val customer = Customer(“Fred”).apply { messages } ❶
    assertEquals(3, customer.messages.size)            ❷
}
```

❶ Загрузит сообщения при первой попытке обращения к свойству messages

❷ Повторное обращение к свойству messages, но сообщения уже загружены

Загрузить сообщения, используя свойство конструктора, нельзя, потому что _messages является приватным свойством. Если необходимо, чтобы сообщения загружались немедленно, используйте функцию apply, как показано в этом примере. Здесь использование apply приводит к вызову метода чтения, который загружает сообщения и выводит информационное сообщение. При втором обращении к свойству messages сообщения уже загружены и информационное сообщение не выводится.

Это очень интересный пример, но в действительности он реализует отложенную загрузку. Код будет выглядеть намного проще, если использовать встроенную функцию-делегат lazy, как показано в примере 3.12.

Пример 3.12. Отложенная загрузка сообщений с использованием lazy

```
class Customer(val name: String) {
    val messages: List<String> by lazy { loadMessages() } ❶

    private fun loadMessages(): MutableList<String> =
        mutableListOf(
            “Initial contact”,
            “Convinced them to use Kotlin”,
            “Sold training class. Sweet.”
        ).also { println(“Loaded messages”) }
}
```

❶ Использование делегата lazy

И все же прием принудительной инициализации свойства с использованием приватного теневого поля стоит запомнить.

Другая разновидность этого приема состоит в том, чтобы определить параметр конструктора для установки значения и при этом обеспечить соблюдение ограничений для свойств, как было показано в примере 3.1, код из которого повторяется ниже для простоты:

```
class Task(val name: String, _priority: Int = DEFAULT_PRIORITY) {

    companion object {
        const val MIN_PRIORITY = 1
        const val MAX_PRIORITY = 5
        const val DEFAULT_PRIORITY = 3
    }

    var priority = validPriority(_priority)
        set(value) {
            field = validPriority(value)
    }

    private fun validPriority(p: Int) =
        p.coerceIn(MIN_PRIORITY, MAX_PRIORITY)
}
```


Обратите внимание, что `_priority` объявлено без ключевого слова `val`, то есть это обычный параметр конструктора, а не фактическое свойство класса. Фактическое свойство `priority` имеет свой метод записи, который присваивает свойству значение аргумента конструктора.

Прием использования теневых свойств довольно часто встречается в классах Kotlin, поэтому стоит потратить время, чтобы понять, как он работает.

Смотри также

Рецепт 8.2, где обсуждается делегат `lazy`.

3.5. Перегрузка операторов

Задача

Дать возможность клиентам использовать операторы, такие как + и *, с экземплярами библиотечных классов.

Решение

Использовать механизм перегрузки операторов в Kotlin и реализовать необходимые функции.

Обсуждение

Многие операторы, включая сложение, вычитание и умножение, реализованы в Kotlin как функции. Когда в исходном коде вы используете символы +, - или *, то на самом деле делегируете работу этим функциям. То есть, определив аналогичные функции для своих классов, вы дадите возможность клиентам использовать операторы.

В документации приводится классический пример – реализация функции-члена `unaryMinus` в классе `Point`, как показано в примере 3.13.

Пример 3.13. Переопределение оператора `unaryMinus` в классе `Point` (пример из документации)

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)

val point = Point(10, 20)

fun main() {
    println(-point) // выведет: «Point(x=-10, y=-20)»
}
```

Ключевое слово `operator` обязательно должно использоваться при переопределении любых функций-операторов, кроме `equals`.

А можно ли добавить соответствующие функции в чужой класс? Да, можно, но в этом случае функция должна определяться как функция-расширение.

Возьмем для примера класс `Complex` из Java-библиотеки Apache Commons Math, который представляет комплексное число (с действительной и мнимой частями). Как сообщает документация Javadocs, этот класс включает такие методы, как `add`, `subtract` и `multiply`. В языке Kotlin операторам `+`, `-` и `*` соответствуют функции `plus`, `minus` и `times`. Если добавить функции-расширения для `Complex`, которые будут вызывать существующие методы, как в примере 3.14, то вы сможете использовать операторы.

Пример 3.14. Функции-расширения для Complex

```
import org.apache.commons.math3.complex.Complex

operator fun Complex.plus(c: Complex) = this.add(c)
operator fun Complex.plus(d: Double) = this.add(d)
operator fun Complex.minus(c: Complex) = this.subtract(c)
operator fun Complex.minus(d: Double) = this.subtract(d)
operator fun Complex.div(c: Complex) = this.divide(c)
operator fun Complex.div(d: Double) = this.divide(d)
operator fun Complex.times(c: Complex) = this.multiply(c)
operator fun Complex.times(d: Double) = this.multiply(d)
operator fun Complex.times(i: Int) = this.multiply(i)
operator fun Double.times(c: Complex) = c.multiply(this)
operator fun Complex.unaryMinus() = this.negate()
```

Все эти функции-расширения вызывают существующие методы в Java-классе. Тест в примере 3.15 показывает, как использовать эти функции-операторы.

Пример 3.15. Использование операторов с экземплярами Complex

```
import org.apache.commons.math3.complex.Complex
import org.apache.commons.math3.complex.Complex.* ❶

import org.hamcrest.MatcherAssert.assertThat
import org.hamcrest.Matchers.`is`
import org.hamcrest.Matchers.closeTo
import org.junit.jupiter.api.Test
```

```
import org.junit.jupiter.api.Assertions.*
import java.lang.Math.*                                    ❷

internal class ComplexOverloadOperatorsKtTest {
    private val first = Complex(1.0, 3.0)
    private val second = Complex(2.0, 5.0)

    @Test
    internal fun plus() {
        val sum = first + second
        assertThat(sum, `is`(Complex(3.0, 8.0)))
    }

    @Test
    internal fun minus() {
        val diff = second - first
        assertThat(diff, `is`(Complex(1.0, 2.0)))
    }

    @Test
    internal fun negate() {
        val minus1 = -ONE ❶

        assertThat(minus1.real, closeTo(-1.0, 0.000001))
        assertThat(minus1.imaginary, closeTo(0.0, 0.000001))
    }

    @Test
    internal fun `Euler's formula`() {
        val iPI = I * PI ❷

        assertTrue(Complex.equals(iPI.exp(), -ONE, 0.000001))
    }
}
```

❶ Импортирование `Complex.*` позволяет сократить `Complex.ONE` до `ONE`

❷ Ссылки `Complex.I` и `Math.PI` можно сократить до `I` и `PI`

В последнем тесте функция `exp` класса `Complex` вернет значение `e^{arg}`, то есть этот тест фактически иллюстрирует формулу Эйлера, `e^{i * PI} == -1`.

Тесты демонстрируют множество перегруженных операторов. Если вы пишете на Kotlin и используете класс `Complex`, то, потратив совсем немного времени на создание коротких функций-расширений, сможете использовать те же операторы, что и с обычными числами.

3.6. Отложенная инициализация с помощью lateinit

Задача

Отложить инициализацию свойства на более позднее время, если в момент вызова конструктора отсутствует информация, необходимая для этого.

Решение

Использовать модификатор `lateinit` в объявлении свойства.

Обсуждение

Не злоупотребляйте этим приемом. Он может пригодиться, например, для внедрения зависимостей, как описывается в этом рецепте, но в общем случае предпочтительнее использовать другие альтернативы, такие как отложенные, или ленивые, вычисления, о которых рассказывается в рецепте 8.2.

Предполагается, что свойства класса, объявленные как не поддерживающие значение `null`, будут инициализированы в конструкторе. Однако иногда в момент вызова конструктора недостаточно информации, чтобы определить значение для такого свойства. Это обычная ситуация для фреймворков внедрения зависимостей, в которых внедрение происходит только после создания всех объектов, и для методов настройки модульных тестов. В таких случаях можно объявить свойство с модификатором `lateinit`.

Например, фреймворк Spring использует аннотацию `@Autowired` для присваивания значений зависимостям из так называемого контекста приложения. Поскольку значение устанавливается после создания всех экземпляров, свойство следует объявить с модификатором `lateinit`, как показано в примере 3.16.

Пример 3.16. Тестирование контроллера Spring

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class OfficerControllerTests {
    @Autowired
    lateinit var client: WebTestClient          ❶

    @Autowired
    lateinit var repository: OfficerRepository ❶

    @Before
    fun setUp() {
        repository.addTestData()                ❷
    }

    @Test
    fun `GET to route returns all officers in db`() {
        client.get().uri("/route")              ❸
        // ... получить и проверить данные ...
    }

    // ... другие тесты ...
}
```

❶ Инициализация с использованием механизма автоматического связывания

❷ Использование репозитория в функции настройки

❸ Использование клиента в тестах

Модификатор `lateinit` можно применять только к `var`-свойствам, объявленным в теле класса, и только если свойство не имеет нестандартных методов чтения и записи. Начиная с версии Kotlin 1.2 `lateinit` также можно применять к свой-

ствам верхнего уровня и даже к локальным переменным. Типы таких свойств не должны поддерживать значение null и не могут быть простыми типами.

Добавляя модификатор lateinit, вы обещаете инициализировать переменную перед ее первым использованием. Если вы нарушите это обещание, программа сгенерирует исключение, как показано в примере 3.17.

Пример 3.17. Особенности поведения свойств с модификатором lateinit

```
class LateInitDemo {
    lateinit var name: String
}

class LateInitDemoTest {
    @Test
    fun `unitialized lateinit property throws exception`() {
        assertThrows<UninitializedPropertyAccessException> {
            LateInitDemo().name
        }
    }

    @Test
    fun `set the lateinit property and no exception is thrown`() {
        assertDoesNotThrow { LateInitDemo().apply { name = “Dolly” } }
    }
}
```

Попытка обратиться к свойству name до того, как оно будет инициализировано, вызовет исключение UninitializedPropertyAccessException, как показывает тест.

Внутри класса есть возможность проверить, было ли свойство инициализировано, обратившись к свойству isInitialized, которым обладает проверяемое свойство, как показано в примере 3.18.

Пример 3.18. Использование isInitialized для проверки инициализации свойства

```
class LateInitDemo {
    lateinit var name: String

    fun initializeName() {
        println(“Before assignment: ${::name.isInitialized}”)
        name = “World”
        println(“After assignment: ${::name.isInitialized}”)
    }
}

fun main() {
    LateInitDemo().initializeName()
}
```

Если запустить этот пример, его функция initializeName выведет следующее:

```
Before assignment: false
After assignment: true
```

Сравнение lateinit и lazy

Модификатор `lateinit` может применяться только к `var`-свойствам и накладывает некоторые ограничения, перечисленные выше в этом рецепте. Делегат `lazy` принимает лямбда-выражение, которое вычисляется при первом обращении к свойству.

Используйте `lazy`, если для инициализации необходимо выполнить дорогостоящие вычисления и она может никогда не потребоваться в программе. Кроме того, `lazy` можно использовать только для `val`-свойств, а `lateinit` – только для `var`-свойств. Наконец, свойства с модификатором `lateinit` можно инициализировать в любом месте, где это свойство доступно, даже за пределами объекта, как показано в одном из предыдущих примеров.

Смотри также

Рецепт 8.2, где обсуждается делегат `lazy`.

3.7. Использование операторов безопасного приведения типа, ссылочного равенства и «Элвис» для переопределения метода equals

Задача

Реализовать свою версию метода `equals` в классе, чтобы получить возможность проверять эквивалентность его экземпляров.

Решение

Использовать вместе операторы ссылочного равенства (`===`), безопасного приведения типа (`as?`) и «Элвис» (`?:`).

Обсуждение

Все объектно-ориентированные языки поддерживают понятия эквивалентности и равенства объектов. В Java оператор == проверяет равенство ссылок на объекты. Метод `equals` класса `Object`, напротив, проверяет эквивалентность двух объектов.

В Kotlin оператор == автоматически вызывает функцию `equals`. Открытый класс `Any` объявляет функцию `equals`, как показано в примере 3.19, а также функции `hashCode` и `toString`.

Пример 3.19. Объявления методов equals, hashCode и toString в классе Any

```
open class Any {
    open operator fun equals(other: Any?): Boolean

    open fun hashCode(): Int

    open fun toString(): String
}
```

Контракт метода equals требует, чтобы реализация была рефлексивной, симметричной и транзитивной, а также надлежащим образом обрабатывала значения null. Контракт метода hashCode требует, чтобы для эквивалентных (с точки зрения функции equals) объектов он возвращал одинаковые хешкоды. Функцию hashCode необходимо переопределять всегда, когда переопределяется функция equals.

Итак, как правильно реализовать хорошую функцию equals? Отличным примером может служить библиотечный класс KotlinVersion, функция equals которого показана в примере 3.20.

Пример 3.20. Функция equals из класса KotlinVersion

```
override fun equals(other: Any?): Boolean {
    if (this === other) return true
    val otherVersion = (other as? KotlinVersion) ?: return false
    return this.version == otherVersion.version
}
```

Обратите внимание на простоту и элегантность реализации, которая использует некоторые особенности Kotlin:

- сначала проверяется равенство ссылок с помощью ===;
- затем используется оператор безопасного приведения типа as?, который приводит аргумент к указанному типу или возвращает null;
- если оператор безопасного приведения типа вернет null, то оператор «Элвис» (?:) вернет false, потому что экземпляры разных классов не могут быть эквивалентными;
- наконец, последняя строка проверяет (с помощью оператора ==) эквивалентность свойства version текущего экземпляра и аналогичного свойства другого объекта и возвращает результат.

Три строки кода охватывают все возможные случаи. Для полноты обсуждения ниже приводится реализация hashCode:

```
override fun hashCode(): Int = version
```

Она представляет определенный интерес, но не помогает понять, как написать свою функцию equals. Допустим, что у вас есть простой класс Customer со строковым свойством name. В примере 3.21 показаны подходящие реализации функций equals и hashCode для такого класса.

Пример 3.21. Реализация equals и hashCode в Customer

```
class Customer(val name: String) {

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        val otherCustomer = (other as? Customer) ?: return false
        return this.name == otherCustomer.name
    }

    override fun hashCode() = name.hashCode()
}
```

Между прочим, если позволить среде разработки IntelliJ IDEA самой сгенерировать реализацию equals и hashCode, то в результате вы получите функции (при использовании версии Ultimate Edition 2019.2), показанные в примере 3.22.

Пример 3.22. Функции equals и hashCode, сгенерированные в IntelliJ IDEA

```
class Customer(val name: String) {
    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (javaClass != other?.javaClass) return false

        other as Customer

        if (name != other.name) return false
        return true
    }

    override fun hashCode(): Int {
        return name.hashCode()
    }
}
```

Сгенерированная функция equals проверяет эквивалентность свойств javaClass перед приведением типа с помощью оператора as, а затем полагается на результат *интеллектуального приведения* для проверки свойств name. Фактически этот код эквивалентен коду, показанному выше, только немного подробнее.

Классы данных получают свои, сгенерированные автоматически реализации equals и hashCode (а также методы toString, copy и componentN). Однако, как можно видеть, реализация своих версий этих функций не представляет никаких сложностей.

Смотри также

Рецепт 3.3, где обсуждаются классы данных. Рецепт 11.1, где показана версия класса KotlinVersion.

3.8. Создание синглтона

Задача

Гарантировать, что во время выполнения в программе будет существовать не более одного экземпляра класса.

Решение

Использовать ключевое слово object вместо class.

Обсуждение

Шаблон проектирования «Синглтон» (Singleton) определяет механизм, гарантирующий, что во время выполнения в программе будет существовать

не более одного экземпляра данного класса. Чтобы определить синглтон, нужно:

1) объявить все конструкторы класса приватными;
2) реализовать статический фабричный метод, возвращающий ссылку на экземпляр класса и создающий его, если необходимо.

Шаблон проектирования «Синглтон» – довольно неоднозначный, потому что иногда он используется в ситуациях, когда вместо одного вполне можно было бы использовать несколько экземпляров класса. Тем не менее это один из фундаментальных шаблонов проектирования, описанных Эрихом Гаммой (Erich Gamma) с соавторами в книге «Design Patterns»6 (Addison-Wesley Professional), и в некоторых случаях может быть очень полезен.

Примером синглтона в стандартной библиотеке Java может служить класс `Runtime`. Допустим, вам нужно узнать, сколько процессоров доступно на данной платформе. В Java для этого можно использовать код, показанный в примере 3.23.

Пример 3.23. Определение числа процессоров

```
fun main() {
    val processors = Runtime.getRuntime().availableProcessors()
    println(processors)
}
```

`getRuntime` – это статический метод, возвращающий синглтон (единственный экземпляр) класса. В примере 3.24 показана соответствующая часть класса `java.lang.Runtime`.

Пример 3.24. Реализация шаблона проектирования «Синглтон» в классе Runtime

```
public class Runtime {
    private static final Runtime currentRuntime = new Runtime();

    public static Runtime getRuntime() {
        return currentRuntime;
    }

    /** Запретить создавать экземпляры класса любым другим способом */
    private Runtime() {}

    // ...
}
```

Класс `Runtime` содержит приватный, статический, конечный экземпляр класса, который создается в момент инициализации атрибута `currentRuntime`. Единственный конструктор объявлен приватным, а статический фабричный метод `getRuntime` просто возвращает ссылку на единственный экземпляр, как показано в примере 3.24.

Чтобы реализовать шаблон «Синглтон» в Kotlin, достаточно просто использовать ключевое слово `object` вместо `class`, как показано в примере 3.25. Это называется *объявлением объекта*.

---

[6] *Гамма Эрих, Хелм Ричард, Джонсон Ральф, Влиссидес Джон.* Паттерны объектно-ориентированного проектирования. Питер, 2020. ISBN: 978-5-4461-1595-2. – *Прим. перев.*

Пример 3.25. Реализация шаблона «Синглтон» в Kotlin

```
object MySingleton {
    val myProperty = 3

    fun myFunction() = “Hello”
}
```

Если декомпилировать байт-код, сгенерированный для этой программы, получится результат, представленный в примере 3.26.

Пример 3.26. Декомпилированный байт-код реализации синглтона с помощью ключевого слова object

```
public final class MySingleton {
    private static final int myProperty = 3;
    public static final MySingleton INSTANCE; ❶

    private MySingleton() {                    ❷
    }

    public final int getMyProperty() {
        return myProperty;
    }

    public final void myFunction() {
        return “Hello”;
    }

    static {
        MySingleton var0 = new MySingleton(); ❸
        INSTANCE = var0;
        myProperty = 3;
    }
}
```

❶ Сгенерированное свойство INSTANCE

❷ Приватный конструктор

❸ Создание экземпляра синглтона

При работе с синглтоном можно обращаться к его членам по имени объекта, так же как к статическим членам в Java. Функции-члены и свойства становятся статическими конечными методами и атрибутами в декомпилированном классе Java, наряду с любыми методами чтения, а сами свойства инициализируются в статическом блоке вместе с классом. В примере 3.27 приводится код на Kotlin, демонстрирующий приемы доступа к членам объекта.

Пример 3.27. Доступ к членам синглтона в Kotlin

```
MySingleton.myFunction()
MySingleton.myProperty
```

В Java для доступа к синглтону используется обобщенное свойство INSTANCE, как показано в примере 3.28.

Пример 3.28. Доступ к членам синглтона в Java

```
MySingleton.INSTANCE.myFunction();
MySingleton.INSTANCE.getMyProperty();
```

Сложность появляется, когда возникает необходимость создать синглтон с аргументом. Скажем, вы реализуете пул соединений с базой данных, который является естественным синглтоном. При создании пула было бы хорошо иметь возможность передать параметр, определяющий начальный размер пула. К сожалению, объект `object` в Kotlin не может иметь конструктора, поэтому нет простой возможности передать ему аргумент.

В своей статье «Kotlin Singletons with Argument» (https://oreil.ly/P8QCv) Кристоф Бейлс (Christophe Beyls) обсуждает способы обработки аргументов на основе реализации делегата `lazy` в библиотеке Kotlin.

В указанной статье также рассказывается о сложностях безопасного создания синглтонов в многопоточной среде из-за необходимости использовать блокировки с двойной проверкой и `@Volatile`.

3.9. Много шума из ничего

Задача

Использовать класс `Nothing` идиоматическим способом.

Решение

Использовать `Nothing` в функциях, которые никогда не возвращают управления.

Обсуждение

> Теперь ты знаешь, что такое Nothing, Джон Сноу.
> – Игритт, Игра престолов, похвала Джону Сноу за чтение этого рецепта

В Kotlin есть класс `Nothing`, полная реализация которого приведена в примере 3.29.

Пример 3.29. Реализация Nothing

```
package kotlin
public class Nothing private constructor()
```

Так как конструктор объявлен приватным, экземпляр класса не может быть создан за пределами класса, и, как видите, он также не создается и внутри класса. То есть экземпляров `Nothing` не существует. В документации говорится, что «`Nothing` можно использовать для представления несуществующих значений».

Класс `Nothing` имеет два естественных применения: когда тело функции полностью состоит из инструкции возбуждения исключения, как в примере 3.30.

Пример 3.30. Возбуждение исключения в Kotlin

```
fun doNothing(): Nothing = throw Exception("Nothing at all")
```

Тип возвращаемого значения должен быть указан явно, а поскольку этот метод никогда не возвращает управления (вместо этого он генерирует исключение), то тип возвращаемого им значения объявлен как `Nothing`.

Это практически гарантированно запутывает разработчиков на Java. В Java, если метод вызывает исключение любого типа, тип значения, возвращаемого методом, не изменяется. Обработка исключений происходит за рамками обычного потока выполнения, но вам не нужно изменять тип возвращаемого значения метода. Система типов в Kotlin, однако, предъявляет другие требования.

Другое естественное применение `Nothing` – когда переменной, объявленной без типа, присваивается значение `null`, как в примере 3.31.

Пример 3.31. Переменной, объявленной без типа, присваивается значение null

```
val x = null
```

В этом тип переменной `x` будет определен как `Nothing?`, потому что он явно поддерживает значение `null` (которое присваивается переменной) и компилятор не имеет никакой другой информации.

Ситуация становится особенно интересной, если вспомнить, что в Kotlin класс `Nothing` фактически является подтипом любого другого типа.

Чтобы понять, почему это необходимо, рассмотрим оператор `if`, который может вызвать исключение (пример 3.32).

Пример 3.32. Оператор if, который может вызвать исключение

```
val x = if (Random.nextBoolean()) "true" else throw Exception("nope")
```

Тип `x` может быть определен как `String`, `Comparable<String>`, `CharSequence`, `Serializable` или даже `Any`, в зависимости от строки, возвращаемой оператором, когда функция `Random.nextBoolean` сгенерирует истинное логическое значение. Предложение `else` возвращает значение типа `Nothing`, а поскольку `Nothing` является подтипом каждого типа, логическая операция «И» с ним и любым другим типом возвращает этот другой тип.

Возможно, следующий пример поможет придать обсуждению дополнительную ясность. Остаток от деления любого числа на 3 может быть равен 0, 1 или 2. Следовательно, оператору `when` в примере 3.33 не нужно предложение `else`, но компилятор этого не знает.

Пример 3.33. Остаток от деления на 3

```
for (n in 1..10) {
    val x = when (n % 3) {
        0 -> "$n % 3 == 0"
        1 -> "$n % 3 == 1"
        2 -> "$n % 3 == 2"
        else -> throw Exception("Houston, we have a problem...")
    }
    assertTrue(x is string)
}
```

Конструкция `when` возвращает значение, поэтому компилятор требует, чтобы оно было исчерпывающим. Условие `else` в данном случае никогда не должно выполняться, поэтому имеет смысл сгенерировать исключение. В случае исключения возвращаемым типом будет `Nothing`, а в остальных случаях – тип `String`, поэтому компилятор будет знать, что `x` имеет тип `String`.

Функция `TODO` (обсуждается в рецепте 11.9) возвращает тип `Nothing`, и в этом есть определенный смысл, потому что ее реализация генерирует исключение `NotImplementedError`.

Класс `Nothing` может сбивать с толку, но после знакомства с вариантами использования он становится простым и понятным.

Глава 4

Функциональное программирование

Термин «функциональное программирование» описывает стиль, поощряющий использование неизменяемых элементов данных, упрощающий реализацию алгоритмов параллельных вычислений с использованием чистых функций, использующий преобразования вместо циклов и фильтры вместо условных операторов. В этой книге функциональные подходы используются повсюду, особенно в главах 5, 6 и 13. Многие функции в Kotlin, такие как `map` и `filter`, обсуждаются по мере встречи с ними в отдельных рецептах в этих и других главах.

Эта глава представляет рецепты, которые используют приемы функционального программирования, уникальные для Kotlin (в отличие от Java), например хвостовую рекурсию, или реализованные несколько иначе, к примеру функции `fold` и `reduce`.

4.1. Использование fold в алгоритмах

Задача

Реализовать итеративный алгоритм функциональным способом.

Решение

Использовать функцию `fold` для свертки последовательности или коллекции в единственное значение.

Обсуждение

Функция `fold` выполняет операцию свертки и может применяться к массивам и итерируемым объектам. Вот как выглядит определение функции:

```
inline fun <R> Iterable<T>.fold(
    initial: R,
    operation: (acc: R, T) -> R
): R
```

Эта функция также определена в классе `Array` и во всех типизированных массивах, таких как `IntArray`, `DoubleArray` и т. д.

Функция fold принимает два параметра: начальное значение для аккумулятора и функцию двух аргументов, которая возвращает новое значение для аккумулятора. Классический пример использования fold – вычисление суммы (см. пример 4.1).

Пример 4.1. Вычисление суммы целых чисел с помощью fold

```
fun sum(vararg nums: Int) =
    nums.fold(0) { acc, n -> acc + n }
```

В этом случае начальное значение равно 0, а лямбда-выражение, передаваемое в функцию, принимает два аргумента, первый из которых является аккумулятором. Второй выполняет итерации по всем значениям в списке параметров. Тест в примере 4.2 показывает, что эта функция вычисляет верный результат.

Пример 4.2. Тестирование функции sum

```
@Test
fun `sum using fold`() {
    val numbers = intArrayOf(3, 1, 4, 1, 5, 9)
    assertEquals(numbers.sum(), sum(*numbers))
}
```

Результат, полученный функцией sum, сравнивается с результатом, полученным методом sum класса IntArray. Этот тест показывает, что наша функция работает правильно, но не помогает понять, как именно она работает. Чтобы получить более полное представление, добавьте оператор print для наблюдения, как происходит обход значений, как показано в примере 4.3.

Пример 4.3. Функция sum, которая выводит каждое значение

```
fun sumWithTrace(vararg nums: Int) =
    nums.fold(0) { acc, n ->
        println("acc = $acc, n = $n")
        acc + n
    }
```

Если вызвать ее с теми же аргументами, как в тесте выше, она выведет:

```
acc =  0, n = 3
acc =  3, n = 1
acc =  4, n = 4
acc =  8, n = 1
acc =  9, n = 5
acc = 14, n = 9
```

Переменная acc инициализируется первым аргументом функции fold, переменная n последовательно получает значение каждого элемента коллекции, и в каждой итерации в acc записывается результат лямбда-выражения acc + n.

Само лямбда-выражение является *двухместным оператором*, потому что типы данных аккумулятора, каждого элемента коллекции и возвращаемого значения одинаковы.

Хотя первый аргумент функции fold называется initial и инициализирует аккумулятор, технически это должно быть значение идентичности для лямбда-операции.

Более интересный пример: вычисление факториала целого числа. Эту операцию легко выразить с использованием рекурсии, с которой мы снова встретимся в примере 4.10:

```
fun recursiveFactorial (n: Long): BigInteger =
    when (n) {
        0L, 1L -> BigInteger.ONE
        else -> BigInteger.valueOf(n) * recursiveFactorial(n - 1)
    }
```

Эту операцию можно реализовать итеративным способом, с использованием fold, как показано в примере 4.4.

Пример 4.4. Реализация факториала с использованием fold

```
fun factorialFold(n: Long): BigInteger =
    when(n) {
        0L, 1L -> BigInteger.ONE
        else -> (2..n).fold(BigInteger.ONE) { acc, i ->
            acc * BigInteger.valueOf(i) }
}
```

Условие when проверяет входной аргумент, и если он равен 0 или 1, то возвращает BigInteger.ONE. Условие else использует диапазон от 2 до n (значения входного аргумента) и применяет операцию fold, которая получает начальное значение BigInteger.ONE. В каждой итерации лямбда-выражение присваивает аккумулятору произведение предыдущего значения аккумулятора на каждое последующее значение. И снова, BigInteger.ONE является не только начальным значением аккумулятора, но также значением идентичности операции умножения.

Рассмотрим еще один увлекательный пример использования fold – вычисление последовательности чисел Фибоначчи, в которой каждое следующее число является суммой двух предыдущих. В примере 4.5 показано, как реализовать этот алгоритм с помощью fold.

Пример 4.5. Вычисление чисел Фибоначчи с помощью fold

```
fun fibonacciFold(n: Int) =
    (2 until n).fold(1 to 1) { (prev, curr), _ ->
        curr to (prev + curr) }.second
```

В этом случае начальным значением аккумулятора является экземпляр Pair, свойства first и second которого равны 1. Соответственно, лямбда-выражение может создать новое значение для аккумулятора, не обращая внимания на конкретный индекс, поэтому в качестве заполнителя для этого значения используется подчеркивание (_). Лямбда-выражение создает новый экземпляр Pair, присваивая текущее значение предыдущему и делая новое значение curr равным сумме предыдущего и текущего значений. Этот процесс повторяется от 2 до указанного индекса n. В конце значение свойства second последнего экземпляра Pair представляет результат.

Еще одна интересная особенность этого примера – тип аккумулятора отличается от типа элементов в диапазоне. Аккумулятор – это Pair, а элементы – значения типа Int.

Последний пример показывает, что fold обладает намного более широкими возможностями, чем можно подумать, рассматривая типовой пример с функцией sum.

Смотри также

Рецепт 4.3, где повторно рассматривается задача вычисления факториала. Рецепт 4.2, где вместо `fold` используется функция `reduce`.

4.2. Использование функции reduce для свертки

Задача

Требуется свернуть непустую коллекцию значений, не указывая начальное значение для аккумулятора.

Решение

Использовать функцию `reduce` вместо `fold`.

Обсуждение

Функция `reduce` похожа на `fold`, обсуждавшуюся в рецепте 4.1. Вот как выглядит ее определение в `Iterable`:

```
inline fun <S, T : S> Iterable<T>.reduce(
    operation: (acc: S, T) -> S
): S
```

Функция `reduce` действует почти так же, как `fold`, и используется для той же цели. Ее самое большое отличие – она не имеет аргумента, предоставляющего начальное значение аккумулятора. Она инициализирует аккумулятор первым значением из коллекции.

В примере 4.6 показана реализация `reduce` в стандартной библиотеке.

Пример 4.6. Реализация функции reduce

```
public inline fun IntArray.reduce(
    operation: (acc: Int, Int) -> Int): Int {
    if (isEmpty())                                    ❶
        throw UnsupportedOperationException(
            "Empty array can't be reduced.")
    var accumulator = this[0]                         ❷
    for (index in 1..lastIndex) {
        accumulator = operation(accumulator, this[index])
    }
    return accumulator
}
```


❶ Для пустой коллекции генерируется исключение

❷ Аккумулятор инициализируется первым элементом коллекции

Соответственно, функцию `reduce` можно использовать, только когда коллекция содержит хотя бы одно значение и есть возможность инициализировать аккумулятор. Примером использования `reduce` может послужить та же операция суммирования, представленная ранее в примере 4.1. В примере 4.7 демонстрируется ее реализация с использованием `reduce`.

Пример 4.7. Реализация функции sum с помощью reduce

```
fun sumReduce(vararg nums: Int) =
    nums.reduce { acc, i -> acc + i }
```

Если эту функцию вызвать с несколькими аргументами, первый из них инициализирует аккумулятор, а остальные будут последовательно прибавляться к нему. Если эту функцию вызвать без аргументов, она возбудит исключение, как показано в примере 4.8.

Пример 4.8. Тестирование функции sum, реализованной с помощью reduce

```
@Test
fun `sum using reduce`() {
    val numbers = intArrayOf(3, 1, 4, 1, 5, 9)
    assertAll(
        { assertEquals(numbers.sum(), sumReduce(*numbers)) }, ❶
        { assertThrows<UnsupportedOperationException> {
            sumReduce()
        }        ❷
    })
}
```

❶ Проверка массива значений Int

❷ Вызов без аргументов сгенерирует исключение

Функция reduce имеет еще одну тонкость, которая может стать причиной ошибки. Допустим, вы решили удвоить все входные значения перед суммированием. Решение, напрашивающееся само собой, показано в примере 4.9.

Пример 4.9. Удваивание элементов коллекции перед суммированием

```
fun sumReduceDoubles(vararg nums: Int) =
    nums.reduce { acc, i -> acc + 2 * i }
```

Попытавшись суммировать значения {3, 1, 4, 1, 5, 9}, одновременно отображая значения аккумулятора и переменной i, получаем:

```
acc=3, i=1
acc=5, i=4
acc=13, i=1
acc=15, i=5
acc=25, i=9

org.opentest4j.AssertionFailedError:
Expected :46
Actual   :43
```

Результат получился ошибочным, потому что число 3 – первое значение в списке – использовалось для инициализации аккумулятора и не было удвоено. Для этой операции уместнее было бы использовать fold, а не reduce.

Используйте reduce, только когда допустимо инициализировать аккумулятор первым значением коллекции и для других значений не выполняется никакая дополнительная обработка.

В Java потоки определяют метод `reduce`, имеющий две перегруженные версии. Одна принимает двухместный оператор (здесь используется лямбда-выражение), а другая – начальное значение, подобно функции `fold`. Кроме того, версия без начального значения возвращает тип `Optional`, то есть вместо возбуждения исключения в пустом потоке Java возвращает пустой экземпляр `Optional`.

Разработчики библиотеки Kotlin предпочли реализацию с двумя отдельными функциями и генерировать исключение, если `reduce` применяется к пустой коллекции. Работая с Java, помните об этих различиях, когда будете выбирать, какую функцию использовать.

Смотри также

Рецепт 4.1, где обсуждается функция `fold`.

4.3. Хвостовая рекурсия

Задача

Имеется рекурсивный алгоритм и требуется уменьшить потребление памяти этим алгоритмом.

Решение

Выразить алгоритм с использованием хвостовой рекурсии и добавить ключевое слово `tailrec` в определение функции.

Обсуждение

Разработчики склонны отдавать предпочтение итеративным алгоритмам, потому что их проще понять и запрограммировать. Однако некоторые процедуры проще выразить с использованием рекурсивного алгоритма. В качестве тривиального примера рассмотрим вычисление факториала числа, как показано в примере 4.10.

Пример 4.10. Рекурсивная реализация вычисления факториала

```
fun recursiveFactorial(n: Long): BigInteger =
    when (n) {
        0L, 1L -> BigInteger.ONE
        else -> BigInteger.valueOf(n) * recursiveFactorial(n - 1)
    }
```

Идея проста: факториалы чисел 0 и 1 равны 1 (`0! == 1`, `1! == 1`), а факториал каждого числа больше 1 равен произведению этого числа на факториал числа, которое на единицу меньше данного. Поскольку результат растет очень быстро, в этом примере в качестве типа возвращаемого значения используется класс `BigInteger`, даже притом что аргумент имеет тип `Long`.

Каждый новый рекурсивный вызов добавляет новый кадр в стек вызовов, поэтому есть риск, что процесс исчерпает доступную память. Тест, демонстрирующий это, показан в примере 4.11.

Пример 4.11. Тестирование рекурсивной реализации факториала

```
@Test
fun `check recursive factorial`() {
    assertAll(
        { assertThat(recursiveFactorial(0), `is`(BigInteger.ONE)) },
        { assertThat(recursiveFactorial(1), `is`(BigInteger.ONE)) },
        { assertThat(recursiveFactorial(2), `is`(BigInteger.valueOf(2))) },
        { assertThat(recursiveFactorial(5), `is`(BigInteger.valueOf(120))) },
        { assertThrows<StackOverflowError> { recursiveFactorial(10_000) }} ❶
    )
}
```

❶ Достаточно большое число, чтобы вызвать переполнение стека `StackOverflowError`

Виртуальная машина JVM потерпит аварию с ошибкой `StackOverflowError`, как только процесс исчерпает память, отведенную для стека вызовов (в OpenJDK 1.8 по умолчанию имеет объем 1024 Кбайт).

Подход, известный как *хвостовая рекурсия*, – это частный случай рекурсии, который можно реализовать без добавления новых кадров в стек вызовов. Для этого алгоритм нужно переписать так, чтобы рекурсивный вызов был последней выполняемой операцией и появилась возможность повторно использовать текущий кадр стека.

В примере 4.12 показана версия реализации вычисления факториала, которую можно преобразовать в хвостовую рекурсию.

Пример 4.12. Реализация факториала с хвостовым рекурсивным вызовом

```
@JvmOverloads                                                    ❶
tailrec fun factorial(n: Long,                                   ❷
                      acc: BigInteger = BigInteger.ONE): BigInteger =
    when (n) {
        0L -> BigInteger.ONE
        1L -> acc
        else -> factorial(n - 1, acc * BigInteger.valueOf(n)) ❸
    }
```

❶ Аннотация, позволяющая вызывать функцию из Java с единственным аргументом

❷ Используется ключевое слово `tailrec`

❸ Хвостовой рекурсивный вызов

В этом случае функции `factorial` нужен второй аргумент, играющий роль аккумулятора. Благодаря такой организации последнее вычисленное выражение может вызвать самого себя с меньшим числом и увеличенным аккумулятором.

Для второго аргумента определено значение по умолчанию `BigInteger.ONE`, и поскольку этот аргумент имеет значение по умолчанию, функцию `factorial` можно вызывать без него. Так как Java не поддерживает аргументов со значениями по умолчанию, аннотация `@JvmOverloads` помогает обеспечить подобную возможность.

Однако самое главное заключается в добавлении ключевого слова `tailrec`. Без этого компилятор не сможет оптимизировать рекурсию. После применения этого ключевого слова функция становится быстрой и эффективной итеративной версией.

Ключевое слово `tailrec` сообщает компилятору о необходимости оптимизировать рекурсивный вызов. Тот же алгоритм, выраженный на Java, по-прежнему останется рекурсивным и будет иметь те же ограничения в отношении памяти.

Тесты в примере 4.13 показывают, что теперь функции можно передать настолько большое число, что тест просто проверяет количество цифр в ответе.

Пример 4.13. Тестирование реализации с хвостовой рекурсией

```
@Test
fun `factorial tests`() {
    assertAll(
        { assertThat(factorial(0), `is`(BigInteger.ONE)) },
        { assertThat(factorial(1), `is`(BigInteger.ONE)) },
        { assertThat(factorial(2), `is`(BigInteger.valueOf(2))) },
        { assertThat(factorial(5), `is`(BigInteger.valueOf(120))) },
        // ...
        { assertThat(factorial(15000).toString().length, `is`(56130)) }, ❶
        { assertThat(factorial(75000).toString().length, `is`(333061)) } ❶
    )
}
```

❶ Проверяется количество цифр в результате

Если сгенерировать байт-код из реализации на Kotlin и декомпилировать его в Java, то получится результат, показанный в примере 4.14.

Пример 4.14. Код Java, декомпилированный из байт-кода Kotlin

```
public static final BigInteger factorial(long n, BigInteger acc) {
    while(true) {
        BigInteger result;
        if (n == 0L) {
            result = BigInteger.ONE;
        } else {
            if (n != 1L) {
                result = result.multiply(BigInteger.valueOf(n));
                n = n - 1L;
                continue;
            }
        }
        return result;
    }
}
```

Рекурсивный вызов был преобразован компилятором в итеративный алгоритм с использованием цикла `while`.

В заключение отметим, какими свойствами должна обладать функция, чтобы к ней можно было применить модификатор `tailrec`:

- рекурсивный вызов должен быть последней операцией;
- `tailrec` нельзя использовать внутри блоков `try/catch/finally`;
- хвостовая рекурсия поддерживается только на уровне JVM.

Глава 5

Коллекции

Как и Java, Kotlin тоже поддерживает типизированные коллекции для хранения наборов объектов. Но, в отличие от Java, Kotlin добавляет в классы коллекций множество интересных методов.

Рецепты в этой главе обсуждают приемы работы с массивами и коллекциями, начиная от сортировки и поиска и заканчивая созданием представлений только для чтения, доступом к диапазонам данных и т. д.

5.1. Работа с массивами

Задача

Создать и заполнить массив на Kotlin.

Решение

Использовать функцию `arrayOf` для создания массивов, а свойства и методы класса `Array` – для работы со значениями, содержащимися в массивах.

Обсуждение

Практически все языки программирования поддерживают массивы, и Kotlin не исключение. В этой книге основное внимание уделяется использованию Kotlin в JVM, но в Java массивы обрабатываются немного иначе, чем в Kotlin. В Java экземпляр массива создается с помощью ключевого слова `new` и размерности массива, как показано в примере 5.1.

Пример 5.1. Создание массива в Java

```
String[] strings = new String[4];
strings[0] = “an”;
strings[1] = “array”;
strings[2] = “of”;
strings[3] = “strings”;

// или еще проще,
strings = «an array of strings».split(« «);
```

Kotlin предлагает для создания массивов простой фабричный метод `arrayOf`. И хотя для доступа к элементам массивов используется тот же синтаксис, тип `Array` в Kotlin – это класс. Пример 5.2 показывает, как работает фабричный метод.

Пример 5.2. Использование фабричного метода arrayOf

```
val strings = arrayOf("this", "is", "an", "array", "of", "strings")
```

Также можно использовать фабричный метод `arrayOfNulls`, чтобы получить массив (как нетрудно догадаться), содержащий только `null`, как показано в примере 5.3.

Пример 5.3. Создание массива с элементами, содержащими значение null

```
val nullStringArray = arrayOfNulls<String>(5)
```

Обратите внимание: даже притом что массив содержит только значения `null`, вы все еще должны выбрать для него конкретный тип данных. В конце концов, не будет же он вечно хранить значения `null`, и компилятор должен знать, ссылки какого типа вы планируете добавлять в него. Аналогично действует фабричный метод `emptyArray`.

Класс `Array` имеет только один общедоступный конструктор. Он принимает два аргумента:

- `size` типа `Int`;
- `init`, лямбда-выражение типа `(Int) -> T`.

Лямбда-выражение вызывается для каждого элемента. В примере 5.4 показано, как создать массив строк, содержащий строковые представления квадратов пяти первых целых чисел.

Пример 5.4. Массив строк, содержащий строковые представления квадратов целых чисел от 0 до 4

```
val squares = Array(5) { i -> (i * i).toString() } ❶
```

❶ Получится массив: {«0», «1», «4», «9», «16»}

Класс `Array` объявляет общедоступные методы `get` и `set` оператора `[]`, которые вызываются при попытке обратиться к элементу массива с использованием квадратных скобок, например `squares[1]`.

В Kotlin имеются специальные классы, представляющие массивы простых типов и помогающие избежать затрат на автоматическую упаковку и распаковку. Функции `booleanArrayOf`, `byteArrayOf`, `shortArrayOf`, `charArrayOf`, `intArrayOf`, `longArrayOf`, `floatArrayOf` и `doubleArrayOf` создают массивы соответствующих типов (`BooleanArray`, `ByteArray`, `ShortArray` и т. д.).

Несмотря на то что Kotlin не имеет явной поддержки простых типов, сгенерированный байт-код использует классы-обертки на Java, такие как `Integer` и `Double`, если элементы массива могут принимать значения `null`, и простые типы, такие как `int` и `double`, если нет.

Массивы имеют практически тот же набор методов-расширений, что и коллекции, о которых рассказывается в оставшейся части этой главы. Однако есть пара методов, присущих только массивам. Например, метод `indices` позволяет узнать допустимые значения индексов для данного массива, как показано в примере 5.5.

Пример 5.5. Получение допустимых значений индексов в массиве

```
@Test
fun `valid indices`() {
    val strings = arrayOf("this", "is", "an", "array", "of", "strings")
    val indices = strings.indices
    assertThat(indices, contains(0, 1, 2, 3, 4, 5))
}
```


Перебор элементов массивов обычно осуществляется с помощью цикла for-in, но при желании то же самое можно сделать с применением значений индексов, используя функцию `withIndex`.

```
fun <T> Array<out T>.withIndex(): Iterable<IndexedValue<T>>

data class IndexedValue<out T>(public val index: Int,
                               public val value: T)
```

Класс `IndexedValue` – это класс данных с двумя свойствами: `index` и `value`. В примере 5.6 показано, как можно использовать его.

Пример 5.6. Доступ к значениям в массиве с помощью withIndex

```
@Test
fun `withIndex returns IndexValues`() {
    val strings = arrayOf("this", "is", "an", "array", "of", "strings")
    for ((index, value) in strings.withIndex()) {      ❶
        println("Index $index maps to $value")         ❷
        assertTrue(index in 0..5)
    }
}
```

❶ Вызов `withIndex`

❷ Обращение к отдельным индексам и значениям

Этот тест выведет следующие строки:

```
Index 0 maps to this
Index 1 maps to is
Index 2 maps to an
Index 3 maps to array
Index 4 maps to of
Index 5 maps to strings
```


В общем и целом массивы в Kotlin действуют так же, как в других языках.

5.2. Создание коллекций

Задача

Сгенерировать список, множество или ассоциативный массив.

Решение

Использовать одну из функций для создания неизменяемых коллекций, такую как `listOf`, `setOf` и `mapOf`, или их изменяемых эквивалентов: `mutableListOf`, `mutableSetOf` и `mutableMapOf`.

Обсуждение

Пакет `kotlin.collections` предлагает ряд вспомогательных функций для создания неизменяемых коллекций.

Одна из них – функция `listOf(vararg elements: T): List<T>`, реализация которой показана в примере 5.7.

Пример 5.7. Реализация функции listOf

```
public fun <T> listOf(vararg elements: T): List<T> =
    if (elements.size > 0) elements.asList() else emptyList()
```

Используемая здесь функция `asList` – это функция-расширение для `Array`. Она возвращает список, содержащий элементы указанного массива. Полученный в результате список называется неизменяемым, но правильнее считать его доступным только для чтения: вы не сможете добавлять или удалять элементы из него, но если объекты в списке являются изменяемыми, то сам список будет выглядеть доступным для изменения.

Реализация `asList` вызывает метод `Arrays.asList` в Java, который возвращает список, доступный только для чтения.

В этом же пакете присутствуют следующие функции:

- `listOf`;
- `setOf`;
- `mapOf`.

Пример 5.8 иллюстрирует создание списков и множеств.

Пример 5.8. Создание «неизменяемых» списков, множеств и ассоциативных массивов

```
var numList = listOf(3, 1, 4, 1, 5, 9)                    ❶
var numSet = setOf(3, 1, 4, 1, 5, 9)                      ❷
// numSet.size == 5                                       ❸
var map = mapOf(1 to “one”, 2 to “two”, 3 to “three”)     ❹
```


❶ Создание неизменяемого списка

❷ Создание неизменяемого множества

❸ Множества не хранят повторяющиеся значения

❹ Создание ассоциативного массива из экземпляров `Pair`

По умолчанию коллекции в Kotlin являются «неизменяемыми», то есть не поддерживают методов для добавления и удаления элементов. Если сами элементы являются экземплярами изменяемого типа, то со стороны такая коллекция выглядит изменяемой, но сама она будет доступна только для чтения.

Фабричные методы создания изменяемых коллекций имеют имена, начинающиеся с приставки «mutable» (изменяемый):

- `mutableListOf`;
- `mutableSetOf`;
- `mutableMapOf`.

Пример 5.9 демонстрирует создание аналогичных изменяемых коллекций.

Пример 5.9. Создание изменяемых списков, множеств и ассоциативных массивов

```
var numList = mutableListOf(3, 1, 4, 1, 5, 9)
var numSet = mutableSetOf(3, 1, 4, 1, 5, 9)
var map = mutableMapOf(1 to “one”, 2 to “two”, 3 to “three”)
```

Вот как выглядит реализация функции `mapOf` в стандартной библиотеке:

```
public fun <K, V> mapOf(vararg pairs: Pair<K, V>): Map<K, V> =
    if (pairs.size > 0)
        pairs.toMap(LinkedHashMap(mapCapacity(pairs.size)))
    else emptyMap()
```

На входе функция `mapOf` принимает переменный список аргументов с экземплярами `Pair`, поэтому для создания элементов ассоциативных массивов можно использовать функцию инфиксного оператора `to`. Аналогичная функция используется для создания изменяемых ассоциативных массивов.

Также можно создать экземпляры классов, реализующих интерфейсы `List`, `Set` или `Map` напрямую, как показано в примере 5.10.

Пример 5.10. Создание связанного списка

```
@Test
internal fun `instantiating a linked list`() {
    val list = LinkedList<Int>()
    list.add(3)                              ❶
    list.add(1)
    list.addLast(999)                        ❶
    list[2] = 4                              ❷
    list.addAll(listOf(1, 5, 9, 2, 6, 5))
    assertThat(list, contains(3, 1, 4, 1, 5, 9, 2, 6, 5))
}
```

❶ Метод `add` – это псевдоним метода `addLast`

❷ При обращении к элементам массивов вызываются методы `get` и `set`

5.3. Получение представлений только для чтения из существующих коллекций

Задача

Из существующего изменяемого списка, множества или ассоциативного массива получить его версию, доступную только для чтения.

Решение

Создать новую коллекцию, доступную только для чтения, с помощью метода `toList`, `toSet` или `toMap`. Чтобы представление существующей коллекции было доступно только для чтения, его следует присвоить переменной типа `List`, `Set` или `Map`.

Обсуждение

Рассмотрим изменяемый список, созданный вызовом фабричного метода `mutableList`. Этот список имеет такие методы, как `add`, `remove` и т. д., позволяющие увеличивать или уменьшать список по желанию:

```
val mutableNums = mutableListOf(3, 1, 4, 1, 5, 9)
```

Создать из изменяемого списка версию, доступную только для чтения, можно двумя способами. Первый – вызвать метод `toList`, возвращающий ссылку типа `List`:

```
@Test
fun `toList on mutableList makes a readOnly new list`() {
    val readOnlyNumList: List<Int> = mutableNums.toList() ❶
    assertEquals(mutableNums, readOnlyNumList)
    assertNotSame(mutableNums, readOnlyNumList)
}
```

❶ Явное объявление типа подсказывает, что результат является неизменяемым списком `List<T>`

Как показывает тест, метод `toList` возвращает значение типа `List<T>`, то есть неизменяемый список, в котором отсутствуют такие методы, как `add` или `remove`. Остальная часть теста показывает, что метод создает отдельный объект, поэтому, даже притом что он имеет то же содержимое, что и оригинал, эти два списка – разные объекты:

```
@Test
internal fun `modify mutable list does not change read-only list`() {
    val readOnly: List<Int> = mutableNums.toList()
    assertEquals(mutableNums, readOnly)
    mutableNums.add(2)
    assertThat(readOnly, not(contains(2)))
}
```

Если вам потребуется создать представление только для чтения с тем же содержимым, просто присвойте изменяемый список переменной ссылочного типа `List`, как показано в примере 5.11.

Пример 5.11. Создание представления только для чтения из изменяемого списка

```
@Test
internal fun `read-only view of a mutable list`() {
    val readOnlySameList: List<Int> = mutableNums ❶
    assertEquals(mutableNums, readOnlySameList)
    assertSame(mutableNums, readOnlySameList)

    mutableNums.add(2)
    assertEquals(mutableNums, readOnlySameList)
    assertSame(mutableNums, readOnlySameList)     ❷
}
```

❶ Присваивание изменяемого списка переменной ссылочного типа `List`

❷ В действительности это один и тот же объект

На этот раз изменяемый список присваивается переменной ссылочного типа `List`. В результате обе переменные не только ссылаются на один и тот же список в памяти, но если изменить основной изменяемый список, то представление только для чтения тоже изменится. Вы не сможете изменить список, используя ссылку, доступную только для чтения, но она указывает на тот же объект, что и исходная переменная.

Функции `toSet` и `toMap` действуют аналогично и для изменяемых множеств и ассоциативных массивов возвращают неизменяемые ссылки типа `Set` и `Map`.

5.4. Конструирование ассоциативного массива из коллекции

Задача

На основе имеющегося списка ключей сконструировать ассоциативный массив, связав каждый ключ со значением, сгенерированным программно.

Решение

Использовать функцию `associateWith` и лямбда-выражение, которое будет вызвано для каждого ключа.

Обсуждение

Пусть имеется набор ключей и каждый из них требуется связать со сгенерированным значением. Один из способов сделать это – использовать функцию `associate`, как показано в примере 5.12.

Пример 5.12. Конструирование ассоциативного массива с помощью associate

```
val keys = 'a'..'f'
val map = keys.associate { it to it.toString().repeat(5).capitalize() }
println(map)
```

Если выполнить этот фрагмент, то он выведет следующий результат:

```
{a=Aaaaa, b=Bbbbb, c=Ccccc, d=Ddddd, e=Eeeee}
```

Функция associate – это встраиваемая (inline) функция-расширение в Iterable<T>, которая принимает лямбда-выражение, преобразующее T в Pair<K, V>. В этом примере используется инфиксная функция to, которая создает Pair из левого и правого аргументов.

В версии Kotlin 1.3 появилась новая функция associateWith, которая помогает сделать код еще проще. В примере 5.13 показано, как будет выглядеть код из предыдущего примера, если вместо associate использовать associateWith.

Пример 5.13. Конструирование ассоциативного массива с помощью associateWith

```
val keys = 'a'..'f'
val map = keys.associateWith { it.toString().repeat(5).capitalize() }
println(map)
```

Результат тот же самый, но теперь в аргументе передается функция, генерирующая значение String, а не Pair<Char, String>.

Оба примера создают один и тот же результат, но применение функции associateWith выглядит проще.

5.5. Возврат значения по умолчанию в случае пустой коллекции

Задача

При обработке коллекции были отфильтрованы все ее элементы, и требуется вернуть ответ по умолчанию.

Решение

Использовать функции ifEmpty и ifBlank, если коллекция или строка окажется пустой.

Обсуждение

Пусть имеется класс данных Product со строковым свойством name, числовым свойством price и логическим свойством onSale, указывающим, имеется ли в продаже данный продукт, как показано в примере 5.14.

Пример 5.14. Класс данных, представляющий продукт

```
data class Product(val name: String,
                   var price: Double,
                   var onSale: Boolean = false)
```

Чтобы вывести список продуктов, имеющихся в продаже, можно выполнить простую операцию фильтрации, как показано ниже:

```
fun namesOfProductsOnSale(products: List<Product>) =
    products.filter { it.onSale }
        .map { it.name }
        .joinToString(separator = ", ")
```

Этот код получает список продуктов, фильтрует его по логическому свойству onSale и оставляет только их названия, которые затем объединяются в одну строку. Проблема в том, что если в продаже нет ни одного товара из списка, фильтр вернет пустую коллекцию, которая затем будет преобразована в пустую строку.

Если для пустого результата требуется вернуть конкретную строку, можно использовать функцию ifEmpty, имеющуюся в классах Collection и String. Пример 5.15 показывает, как это сделать.

Пример 5.15. Использование метода ifEmpty классов Collection и String

```
fun onSaleProducts_ifEmptyCollection(products: List<Product>) =
    products.filter { it.onSale }
        .map { it.name }
        .ifEmpty { listOf("none") }          ❶
        .joinToString(separator = ", ")

fun onSaleProducts_ifEmptyString(products: List<Product>) =
    products.filter { it.onSale }
        .map { it.name }
        .joinToString(separator = ", ")
        .ifEmpty { "none" }                  ❷
```

❶ Подставляет значение по умолчанию для пустой коллекции

❷ Подставляет значение по умолчанию для пустой строки

В обоих случаях, если в коллекции нет ни одного продукта, имеющегося в продаже, код вернет строку «none», как показывают тесты в примере 5.16.

Пример 5.16. Тестирование продуктов

```
class IfEmptyOrBlankKtTest {
    private val overthruster = Product("Oscillation Overthruster", 1_000_000.0)
    private val fluxcapacitor = Product("Flux Capacitor", 299_999.95, onSale = true)
    private val tpsReportCoverSheet = Product("TPS Report Cover Sheet", 0.25)

    @Test
    fun productsOnSale() {
        val products = listOf(overthruster, fluxcapacitor, tpsReportCoverSheet)

        assertAll( "On sale products",
            { assertEquals("Flux Capacitor",
                onSaleProducts_ifEmptyCollection(products)) },
            { assertEquals("Flux Capacitor",
                onSaleProducts_ifEmptyString(products)) })
    }

    @Test
    fun productsNotOnSale() {
        val products = listOf(overthruster, tpsReportCoverSheet)

    assertAll( "No products on sale",
        { assertEquals("none", onSaleProducts_ifEmptyCollection(products)) },
        { assertEquals("none", onSaleProducts_ifEmptyString(products)) })
    }
}
```

В Java 8 появился новый класс `Optional<T>`, который часто используется как обертка для возвращаемого типа, если запрос может вернуть пустое значение или значение `null`. Kotlin тоже поддерживает этот тип, но в большинстве случаев проще вернуть конкретное значение с помощью функции `ifEmpty`.

5.6. Ограничение значений заданным диапазоном

Задача

Для заданного значения вернуть его, если значение попадает в определенный диапазон, или значение нижней или верхней границы диапазона в противном случае.

Решение

Использовать функцию `coerceIn` и передать ей диапазон или минимальную и максимальную границы.

Обсуждение

Имеется две перегруженные версии функции `coerceIn`: одна принимает закрытый диапазон, а вторая – минимальное и максимальное значения.

Рассмотрим применение первой версии с диапазоном целых чисел от 3 до 8 включительно. Тест в примере 5.17 показывает, что возвращает `coerceIn`, если значение попадает и не попадает в указанный диапазон.

Пример 5.17. Приведение значения к диапазону

```
@Test
fun `coerceIn given a range`() {
    val range = 3..8

    assertThat(5, `is`(5.coerceIn(range)))
    assertThat(range.start, `is`(1.coerceIn(range)))        ❶
    assertThat(range.endInclusive, `is`(9.coerceIn(range))) ❷
}
```

❶ Свойство `range.start` равно 3

❷ Свойство `range.endInclusive` равно 8

Точно так же, если известны минимальное и максимальное значения, не нужно создавать диапазон для использования функции `coerceIn`, как показано в примере 5.18.

Пример 5.18. Приведение значения к диапазону при известных минимальном и максимальном значениях

```
@Test
fun `coerceIn given min and max`() {
    val min = 2
    val max = 6

    assertThat(5, `is`(5.coerceIn(min, max)))
    assertThat(min, `is`(1.coerceIn(min, max)))
    assertThat(max, `is`(9.coerceIn(min, max)))
}
```

Эта версия вернет само значение, если оно находится между минимальным и максимальным значениями, или граничное значение в противном случае.

5.7. Обработка коллекций методом скользящего окна

Задача

Обработать коллекцию значений методом скользящего окна.

Решение

Использовать функцию chunked, если требуется разделить коллекцию на равные части, или функцию windowed, чтобы получить окно, скользящее вдоль коллекции с заданным шагом.

Обсуждение

Функция chunked принимает любую итерируемую коллекцию и разбивает ее на список списков с размерами, равными или меньше заданного. Функция имеет две перегруженные версии, одна просто возвращает список списков, а вторая принимает преобразование для применения к получившимся спискам. Сигнатуры обеих версий функции приводятся ниже:

```
fun <T> Iterable<T>.chunked(size: Int): List<List<T>>

fun <T, R> Iterable<T>.chunked(
    size: Int,
    transform: (List<T>) -> R
): List<R>
```

Ситуация выглядит сложнее, чем есть на самом деле. Например, рассмотрим диапазон целых чисел от 0 до 10. Тест в примере 5.19 разбивает его на группы по три числа в каждой и вычисляет суммы и средние значения в группах.

Пример 5.19. Деление списка на фрагменты и последующая их обработка

```
@Test
internal fun chunked() {
    val range = 0..10

    val chunked = range.chunked(3)
    assertThat(chunked, contains(listOf(0, 1, 2), listOf(3, 4, 5),
        listOf(6, 7, 8), listOf(9, 10)))

    assertThat(range.chunked(3) { it.sum() }, `is`(listOf(3, 12, 21, 19)))
    assertThat(range.chunked(3) { it.average() }, `is`(listOf(1.0, 4.0, 7.0, 9.5)))
}
```

Первый вызов просто возвращает список List<List<Int>>, состоящий из подсписков: [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10]]. Во второй и третий вызовы передаются лямбда-выражения, вычисляющие сумму и среднее значение для каждого подсписка соответственно.

Функция chunked фактически является частным случаем функции windowed. Пример 5.20 демонстрирует, как chunked делегирует работу функции windowed.

Пример 5.20. Реализация функции chunked в стандартной библиотеке

```
public fun <T> Iterable<T>.chunked(size: Int): List<List<T>> {
    return windowed(size, size, partialWindows = true)
}
```

Функция windowed принимает три аргумента, два из которых являются необязательными:

`size`

количество элементов в каждом окне;

`step`

на сколько элементов вперед должно перемещаться окно на каждом шаге (по умолчанию 1);

`partialWindows`

логический флаг со значением по умолчанию false. Сообщает, необходимо ли сохранить последний фрагмент, если он содержит меньше элементов, чем задано параметром size.

Функция chunked вызывает windowed и передает ей в параметрах size и step значение своего аргумента, поэтому на каждом шаге окно смещается точно на его размер. Однако если потребуется обработать коллекцию методом скользящего окна с другим шагом, можно использовать windowed напрямую.

Для иллюстрации рассмотрим вычисление скользящего среднего. В примере 5.21 показано, как от функции windowed добиться поведения chunked и как на каждом шаге передвигать окно только на один элемент.

Пример 5.21. Вычисление скользящего среднего в каждом окне

```
@Test
fun windowed() {
    val range = 0..10

    assertThat(range.windowed(3, 3),
        contains(listOf(0, 1, 2), listOf(3, 4, 5), listOf(6, 7, 8)))

    assertThat(range.windowed(3, 3) { it.average() },
        contains(1.0, 4.0, 7.0))

    assertThat(range.windowed(3, 1),
        contains(
            listOf(0, 1, 2), listOf(1, 2, 3), listOf(2, 3, 4),
            listOf(3, 4, 5), listOf(4, 5, 6), listOf(5, 6, 7),
            listOf(6, 7, 8), listOf(7, 8, 9), listOf(8, 9, 10)))

    assertThat(range.windowed(3, 1) { it.average() },
        contains(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0)
```

Функции chunked и windowed – удобные инструменты для поэтапной обработки временных последовательностей.

5.8. Деструктуризация списков

Задача

Использовать механизм деструктуризации для доступа к элементам списка.

Решение

Присвоить список группе, содержащей до пяти элементов.

Обсуждение

Деструктуризация – это процесс извлечения значений из объекта путем присваивания их коллекции переменных.

Пример 5.22 демонстрирует, как присвоить первые пять элементов списка пяти переменным за один шаг.

Пример 5.22. Деструктуризация элементов списка

```
val list = listOf("a", "b", "c", "d", "e", "f", "g")
val (a, b, c, d, e) = list
println("$a $b $c $d $e")
```

Этот код выведет строку `a b c d e`, потому что первые пять элементов созданного списка будут записаны в пять переменных. Деструктуризация поддерживается функциями-расширениями для класса `List`, реализованными в стандартной библиотеке с именами `componentN`, где *N* – число от 1 до 5, как показано в примере 5.23.

Пример 5.23. Функция-расширение component1 для класса List (из стандартной библиотеки)

```
/**
 * Возвращает 1-й *элемент* коллекции.
 */
@kotlin.internal.InlineOnly
public inline operator fun <T> List<T>.component1(): T {
    return get(0)
}
```

Деструктуризация зависит от функций `componentN`. Класс `List` содержит функции `component1`, `component2`, `component3`, `component4` и `component5`, поэтому предыдущий код работает.

Классы данных автоматически добавляют методы `componentN` для всех своих атрибутов. Если вы определите свой класс (но не объявите его классом данных), то сможете вручную добавить все необходимые методы `componentN`.

Деструктуризация дает удобную возможность извлечь из объекта сразу несколько элементов. В настоящее время класс `List` определяет функции `componentN` только для первых пяти элементов, но в будущих версиях Kotlin ситуация может измениться.

5.9. Сортировка по нескольким свойствам

Задача

Отсортировать список экземпляров класса по одному свойству, затем по второму и т. д.

Решение

Использовать функции `sortedWith` и `compareBy`.

Обсуждение

Пусть имеется простой класс данных `Golfer` и коллекция его экземпляров, как показано в примере 5.24.

Пример 5.24. Класс данных и коллекция его экземпляров

```
data class Golfer(val score: Int, val first: String, val last: String)

val golfers = listOf(
    Golfer(70, “Jack”, “Nicklaus”),
    Golfer(68, “Tom”, “Watson”),
    Golfer(68, “Bubba”, “Watson”),
    Golfer(70, “Tiger”, “Woods”),
    Golfer(68, “Ty”, “Webb”)
)
```

Чтобы отсортировать этот список игроков в гольф сначала по количеству набранных очков, затем по фамилии и наконец по имени, можно использовать код, показанный в примере 5.25.

Пример 5.25. Сортировка списка игроков в гольф по нескольким свойствам

```
val sorted = golfers.sortedWith(
    compareBy({ it.score }, { it.last }, { it.first })
)

sorted.forEach { println(it) }
```

Этот код выведет следующий результат:

```
Golfer(score=68, first=Bubba, last=Watson)
Golfer(score=68, first=Tom, last=Watson)
Golfer(score=68, first=Ty, last=Webb)
Golfer(score=70, first=Jack, last=Nicklaus)
Golfer(score=70, first=Tiger, last=Woods)
```

Трое игроков, набравших по 68 очков, выводятся раньше игроков, набравших по 70 очков. В группе с игроками, набравшими по 68 очков, первыми выводятся игроки с фамилией Watson, а затем Webb. В группе с игроками, набравшими по 70 очков, первым выводится игрок с фамилией Nicklaus, а затем Woods. Из игроков с фамилией Watson, набравших по 68 очков, первым выводится игрок с именем Bubba, а затем Tom.

Полные сигнатуры функций `sortedWith` и `compareBy` показаны в примере 5.26.

Пример 5.26. Сигнатуры функций sortedWith и compareBy из стандартной библиотеки

```
fun <T> Iterable<T>.sortedWith(
    comparator: Comparator<in T>
): List<T>

fun <T> compareBy(
    vararg selectors: (T) -> Comparable<*>?
): Comparator<T>
```


То есть функция `sortedWith` принимает экземпляр `Comparator`, а функция `compareBy` возвращает экземпляр `Comparator`. Интересно отметить, что функции `compareBy` можно передать список селекторов, каждый из которых извлекает свойство `Comparable` (обратите внимание, что тип свойства должен реализовать интерфейс `Comparable`), а функция создаст экземпляр `Comparator`, сортирующий их по порядку.

Функции `sortBy` и `sortWith` сортируют свои элементы на месте и поэтому могут применяться только к изменяемым коллекциям.

Ту же задачу можно решить иным способом: создать экземпляр `Comparator` с помощью последовательности вызовов функции `thenBy`, а затем использовать его для сортировки коллекции, как показано в примере 5.27.

Пример 5.27. Объединение нескольких компараторов

```
val comparator = compareBy<Golfer>(Golfer::score)
    .thenBy(Golfer::last)
    .thenBy(Golfer::first)

golfers.sortedWith(comparator)
    .forEach(::println)
```


Этот код вернет тот же результат, что и предыдущий пример.

5.10. Определение своего итератора

Задача

В собственном классе, обертывающем коллекцию, реализовать перебор ее элементов.

Решение

Определить функцию-оператор, возвращающую итератор с методами `next` и `hasNext`.

Обсуждение

Шаблон проектирования «Итератор» (Iterator) реализуется в Java с помощью интерфейса `Iterator`. В примере 5.28 показано соответствующее определение в Kotlin.

Пример 5.28. Интерфейс Iterator в kotlin.collections

```
interface Iterator<out T> {
    operator fun next(): T
    operator fun hasNext(): Boolean
}
```

В Java обход элементов класса, реализующего интерфейс `Iterable`, можно реализовать с помощью цикла for-each. В Kotlin для этой цели используется цикл for-in. Рассмотрим класс данных `Player` и класс `Team`, представленные в примере 5.29.

Пример 5.29. Классы Player и Team

```
data class Player(val name: String)
class Team(val name: String,
           val players: MutableList<Player> = mutableListOf()) {

    fun addPlayers(vararg people: Player) =
        players.addAll(people)

    // ... другие функции ...
}
```

Класс `Team` содержит изменяемый список экземпляров `Player`. Чтобы выполнить обход всех игроков в команде, необходимо обратиться к свойству `players`, как показано в примере 5.30.

Пример 5.30. Обход игроков в команде

```
val team = Team("Warriors")
team.addPlayers(Player(«Curry»), Player(«Thompson»),
    Player(«Durant»), Player(«Green»), Player(«Cousins»))

for (player in team.players) { ❶
    println(player)
}
```

❶ Доступ к свойству `players` в цикле

Эту задачу можно упростить, если в классе `Team` определить функцию-оператор с именем `iterator`. В примере 5.31 показано, как определить такую функцию в виде функции-расширения и насколько проще выглядит цикл.

Пример 5.31. Обход игроков в команде без обращения к свойству players

```
operator fun Team.iterator() : Iterator<Player> = players.iterator()

for (player in team) { ❶
    println(player)
}
```

❶ Итерации выполняются непосредственно по экземпляру команды, без обращения к свойству `players`

Оба последних примера выведут один и тот же результат:

```
Player(name=Curry)
Player(name=Thompson)
Player(name=Durant)
Player(name=Green)
Player(name=Cousins)
```

На самом деле идея состоит в том, чтобы реализовать в классе `Team` интерфейс `Iterable`, что подразумевает реализацию функции-оператора `iterator`. То есть вместо добавления функции-расширения можно изменить определение класса `Team`, как показано в примере 5.32.

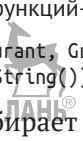
Пример 5.32. Реализация интерфейса Iterable

```
class Team(val name: String,
           val players: MutableList<Player> = mutableListOf()) : Iterable<Player> {

    override fun iterator(): Iterator<Player> =
        players.iterator()

    // ... другие функции ...
}
```

Результат получится тот же, только теперь в классе `Team` будут доступны все функции-расширения для `Iterable` и появится возможность писать код, показанный в примере 5.33.

Пример 5.33. Использование функций-расширений Iterable в Team

```
assertEquals(«Cousins, Curry, Durant, Green, Thompson»,
    team.map { it.name }.joinToString())
```



Здесь функция `map` перебирает игроков, поэтому `it.name` представляет имя каждого игрока. Аналогично можно использовать другие функции-расширения.

5.11. Фильтрация элементов коллекций по типам

Задача

На основе имеющейся коллекции, включающей элементы разных типов, создать новую коллекцию, содержащую только элементы определенного типа.

Решение

Использовать функцию-расширение `filterIsInstance` или `filterIsInstanceTo`.

Обсуждение

Коллекции в Kotlin включают функцию-расширение `filter`, которая принимает предикат для извлечения элементов, удовлетворяющих любому логическому условию, как показано в примере 5.34.

Пример 5.34. Фильтрация элементов коллекции по типу со стиранием типа

```
val list = listOf("a", LocalDate.now(), 3, 1, 4, "b")
val strings = list.filter { it is String }

for (s in strings) {
    // s.length // не компилируется; тип стирается
}
```

Операция фильтрации прекрасно работает в этом примере, но тип переменной `strings` определяется как `List<Any>`, поэтому Kotlin не выполняет интеллектуального приведения отдельных элементов к типу `String`.

При необходимости можно добавить проверку `is` или просто использовать функцию `filterIsInstance`, как в примере 5.35.

Пример 5.35. Овеществление типов

```
val list = listOf("a", LocalDate.now(), 3, 1, 4, "b")

val all = list.filterIsInstance<Any>()
val strings = list.filterIsInstance<String>()
val ints = list.filterIsInstance<Int>()
val dates = list.filterIsInstance(LocalDate::class.java)

assertThat(all, `is`(list))
assertThat(strings, containsInAnyOrder("a", "b"))
assertThat(ints, containsInAnyOrder(1, 3, 4))
assertThat(dates, contains(LocalDate.now()))
```

Функция `filterIsInstance` использует механизм овеществления типа, поэтому получающиеся коллекции имеют известный тип и не нужно проверять тип их элементов перед использованием. Вот как реализована функция `filterIsInstance` в библиотеке:

```
public inline fun <reified R> Iterable<*>.filterIsInstance(): List<R> {
    return filterIsInstanceTo(ArrayList<R>())
}
```

Ключевое слово `reified` сохраняет тип, поэтому возвращаемое значение имеет тип `List<R>`.

Реализация вызывает функцию `filterIsInstanceTo`, которая принимает коллекцию определенного типа и заполняет ее элементами этого типа из оригинала. Эту функцию также можно использовать напрямую, как показано в примере 5.36.

Пример 5.36. Использование овеществления типов для заполнения заданного списка

```
val list = listOf("a", LocalDate.now(), 3, 1, 4, "b")

val all = list.filterIsInstanceTo(mutableListOf())
val strings = list.filterIsInstanceTo(mutableListOf<String>())
val ints = list.filterIsInstanceTo(mutableListOf<Int>())
val dates = list.filterIsInstanceTo(mutableListOf<LocalDate>())

assertThat(all, `is`(list))
assertThat(strings, containsInAnyOrder("a", "b"))
assertThat(ints, containsInAnyOrder(1, 3, 4))
assertThat(dates, contains(LocalDate.now()))
```

Аргумент функции filterIsInstanceTo имеет тип MutableCollection<in R>, поэтому достаточно указать желаемый тип коллекции, чтобы заполнить ее экземплярами этого типа.

5.12. Преобразование диапазона в прогрессию

Задача

Выполнить обход диапазона, который не является диапазоном простых целых чисел или символов.

Решение

Создать прогрессию.

Обсуждение

В Kotlin диапазон создается оператором *двойной точки* (..). Например, выражение 1..5 создаст экземпляр IntRange. *Диапазон* – это закрытый интервал, определяемый двумя конечными точками, входящими в диапазон.

Стандартная библиотека добавляет функцию-расширение rangeTo к любому обобщенному типу T, который реализует интерфейс Comparable. В примере 5.37 показана ее реализация.

Пример 5.37. Реализация функции rangeTo для типов Comparable

```
operator fun <T : Comparable<T>> T.rangeTo(that: T): ClosedRange<T> =
    ComparableRange(this, that)
```

Класс ComparableRange просто расширяет Comparable, определяет свойства start и endInclusive типа T и переопределяет функции equals, hashCode и toString. Значение, возвращаемое функцией rangeTo, имеет тип ClosedRange – простой интерфейс, определение которого показано в примере 5.38.

Пример 5.38. Интерфейс ClosedRange

```
interface ClosedRange<T: Comparable<T>> {
    val start: T
    val endInclusive: T
    operator fun contains(value: T): Boolean =
        value >= start && value <= endInclusive
    fun isEmpty(): Boolean = start > endInclusive
}
```

Функция-оператор contains позволяет использовать инфиксную функцию in для проверки попадания некоторого значения в диапазон.

Из всего этого следует, что диапазон можно создать на основе любого класса, реализующего Comparable, а вся инфраструктура, необходимая для его поддержки, уже существует. Для иллюстрации возьмем тип java.time.LocalDate (см. пример 5.39).

Пример 5.39. Использование LocalDate в диапазоне

```
@Test
fun `LocalDate in a range`() {
    val startDate = LocalDate.now()
    val midDate = startDate.plusDays(3)
    val endDate = startDate.plusDays(5)

    val dateRange = startDate..endDate

    assertAll(
        { assertTrue(startDate in dateRange) },
        { assertTrue(midDate in dateRange) },
        { assertTrue(endDate in dateRange) },
        { assertTrue(startDate.minusDays(1) !in dateRange) },
        { assertTrue(endDate.plusDays(1) !in dateRange) }
    )
}
```

Вроде бы все хорошо, но при попытке выполнить обход диапазона начинают твориться странности:

```
for (date in dateRange) println(it)        // ошибка компиляции!
(startDate..endDate).forEach { /* ... */ } // ошибка компиляции!
```

Проблема в том, что диапазон не является прогрессией. *Прогрессия* – это всего лишь упорядоченная последовательность значений. Пользовательские прогрессии реализуют интерфейс `Iterable`, как и готовые прогрессии `IntProgression`, `LongProgression` и `CharProgression` в стандартной библиотеке.

Рассмотрим создание прогрессий на примере классов в примерах 5.40 и 5.41.

Код этого примера основан на статье Гжегожа Зимонски (Grzegorz Ziemoński) «What Are Kotlin Progressions and Why Should You Care?» в Dzone (https://oreil.ly/-NqqW).

Первый класс – `LocalDateProgression` – реализует интерфейсы `Iterable<LocalDate>` и `ClosedRange<LocalDate>`.

Пример 5.40. Прогрессия для LocalDate

```
import java.time.LocalDate

class LocalDateProgression(
    override val start: LocalDate,
    override val endInclusive: LocalDate,
    val step: Long = 1
) : Iterable<LocalDate>, ClosedRange<LocalDate> {

    override fun iterator(): Iterator<LocalDate> =
        LocalDateProgressionIterator(start, endInclusive, step)

    infix fun step(days: Long) = LocalDateProgression(start, endInclusive, days)
}
```

Из интерфейса Iterator достаточно реализовать единственную функцию iterator. В данном случае она создает экземпляр класса LocalDateProgressionIterator, определение которого показано ниже. Инфиксная функция step создает экземпляр класса, который правильно выполняет приращение в днях. Интерфейс ClosedRange, как показано в примере 5.40, определяет свойства start и endInclusive, поэтому здесь они переопределяются в основном конструкторе.

Пример 5.41. Итератор для класса LocalDateProgression

```
import java.time.LocalDate

internal class LocalDateProgressionIterator(
    start: LocalDate,
    val endInclusive: LocalDate,
    val step: Long
) : Iterator<LocalDate> {

    private var current = start

    override fun hasNext() = current <= endInclusive

    override fun next(): LocalDate {
        val next = current
        current = current.plusDays(step)
        return next
    }
}
```

Интерфейс Iterator требует переопределить методы next и hasNext, как показано здесь.

Наконец, определим функцию-расширение rangeTo, возвращающую экземпляр прогрессии:

```
operator fun LocalDate.rangeTo(other: LocalDate) =
    LocalDateProgression(this, other)
```

Теперь LocalDate можно использовать для создания диапазона, поддерживающего возможность итераций, как показано в примере 5.42.

Пример 5.42. Тесты для прогрессии LocalDate

```
@Test
fun `use LocalDate as a progression`() {
    val startDate = LocalDate.now()
    val endDate = startDate.plusDays(5)

    val dateRange = startDate..endDate
    dateRange.forEachIndexed { index, localDate ->
        assertEquals(localDate, startDate.plusDays(index.toLong()))
    }

    val dateList = dateRange.map { it.toString() }
        assertEquals(6, dateList.size)
    }
```

```
@Test
fun `use LocalDate as a progression with a step`() {
    val startDate = LocalDate.now()
    val endDate = startDate.plusDays(5)

    val dateRange = startDate..endDate step 2
    dateRange.forEachIndexed { index, localDate ->
        assertEquals(localDate, startDate.plusDays(index.toLong() * 2))
    }

    val dateList = dateRange.map { it.toString() }
    assertEquals(3, dateList.size)
}
```

Оператор двойной точки (..) создает диапазон, который в этом случае поддерживает возможность итераций, используемую функцией `forEachIndexed`. Для создания прогрессии в данном примере потребовалось определить два класса и функцию-расширение, но этот шаблон достаточно легко воспроизвести для любого из собственных классов.

Глава 6

Последовательности

В этой главе рассматриваются последовательности Kotlin, которые очень похожи на потоки данных, добавленные в Java 1.8. Это сравнение полезно, только если вы уже знаете, как работают потоки в Java7. Рецепты в данной главе подчеркивают их сходства и различия.

Коллекции обрабатываются *жадно*, то есть целиком и сразу: когда вызывается метод `map` или `filter` коллекции, он обрабатывает все элементы коллекции. Последовательности, напротив, обрабатываются *лениво*: при обработке последовательности каждый ее элемент проходит весь конвейер от начала до конца, прежде чем будет обработан следующий элемент. Это помогает обрабатывать большие объемы данных и выполнять такие операции, как `first`, которые прекращают обработку, встретив первое желаемое значение.

6.1. Использование ленивых последовательностей

Задача

Обработать минимальный объем данных, необходимый для выполнения определенного условия.

Решение

Использовать последовательности Kotlin с функциями, выполняющими вычисления по короткой схеме.

Обсуждение

Kotlin добавляет некоторые функции-расширения в основные коллекции, то есть класс `List` получил такие функции, как `map` и `filter`. Однако эти функции выполняются жадно, в том смысле, что обрабатывают все элементы коллекции.

Рассмотрим простую задачу: взять числа от 100 до 200, удвоить каждое и вернуть первое удвоенное число, которое делится на 3 без остатка. В примере 6.1 показано одно из возможных решений.

---

[7] Самый известный пример подобного объяснения: «Монады – это просто моноиды в категории эндофункторов», что может быть почти правдой, но не совсем. Совершенно бесполезное утверждение.

Пример 6.1. Поиск первого удвоенного числа, кратного 3 (версия 1)

```
(100 until 200).map { it * 2 } ❶
    .filter { it % 3 == 0 }    ❷
    .first()
```

❶ 100 вычислений

❷ Еще 100 вычислений (плохо продуманный алгоритм, можно улучшить)

Недостаток этого решения заключается в крайне низкой эффективности. Сначала оно удваивает все 100 чисел в заданном диапазоне, затем выполняет операцию взятия остатка от деления для всех 100 результатов, а потом извлекает первый элемент. К счастью, существует перегруженная версия функции `first`, принимающая предикат (лямбда-выражение с одним аргументом, возвращающее логическое значение), применение которой показано в примере 6.2.

Пример 6.2. Поиск первого удвоенного числа, кратного 3 (версия 2)

```
(100 until 200).map { it * 2 } ❶
    .first { it % 3 == 0 }     ❷
```

❶ 100 вычислений

❷ Только 3 вычисления

Эта версия `first` обрабатывает элементы коллекции в цикле и останавливается, достигнув первого, удовлетворяющего предикату. Такие вычисления часто называют *вычислениями по короткой схеме* – они обрабатывают только необходимое для достижения условия количество данных. Однако если вы забудете использовать перегруженную версию `first`, ваша программа будет выполнять много ненужной работы.

Последовательности данных в Kotlin обрабатываются иначе. В примере 6.3 демонстрируется еще более оптимальное решение.

Пример 6-3. Поиск первого удвоенного числа, кратного 3 (лучшая версия)

```
(100 until 2_000_000).asSequence() ❶
    .map { println("doubling $it"); it * 2 }
    .filter { println("filtering $it"); it % 3 == 0 }
    .first()
```

❶ Преобразование диапазона в последовательность

Этот код выполнит всего шесть операций до завершения:

```
doubling 100
filtering 200
doubling 101
filtering 202
doubling 102
filtering 204
```


Для последовательностей не имеет значения, используете ли вы функцию `filter`, как в примере 6.1, или перегруженную версию `first`, как в примере 6.2, хотя в последнем случае код будет выглядеть проще. В любом случае будет выполнено только шесть вычислений, потому что каждый элемент последовательности полностью обрабатывается всем конвейером перед переходом

к следующему. Обратите внимание: для наглядности верхний предел последовательности на этот раз был увеличен до двух миллионов, и это никак не повлияло на поведение.

Между прочим, использованная здесь функция `first` сгенерирует исключение, если последовательность окажется пустой. Если такое возможно, используйте вместо нее функцию `firstOrNull`.

Класс `Sequence` предлагает те же функции, что и `Collection`, но операции делятся на две категории: промежуточные и терминальные. *Промежуточные операции*, такие как `map` и `filter`, возвращают новые последовательности. *Терминальные операции*, такие как `first` или `toList`, возвращают нечто иное. Важно запомнить, что без терминальной операции последовательность не будет обрабатывать никаких данных.

Последовательность обрабатывает данные, только если определенный для нее конвейер операций заканчивается терминальной операцией.

В отличие от потоков Java, последовательности Kotlin допускают повторное выполнение итераций, а случаи, когда это невозможно, явно определены в документации.

6.2. Генерирование последовательностей

Задача

Сгенерировать последовательность значений.

Решение

Использовать `sequenceOf`, если имеются отдельные элементы; использовать `asSequence`, если имеется экземпляр `Iterable`; в противном случае использовать генератор последовательностей.

Обсуждение

Первые два решения реализуются тривиально просто. Функция `sequenceOf` действует точно так же, как `arrayOf`, `listOf` и другие подобные функции. Функция `asSequence` преобразует существующий экземпляр `Iterable` (чаще в роли такого экземпляра используются списки и другие коллекции) в `Sequence`. Оба решения показаны в примере 6.4.

Пример 6.4. Создание последовательностей из уже имеющихся значений

```
val numSequence1 = sequenceOf(3, 1, 4, 1, 5, 9)
val numSequence2 = listOf(3, 1, 4, 1, 5, 9).asSequence()
```

Обе инструкции создают `Sequence<Int>` из указанных значений или из заданного списка.

Намного интереснее, когда требуется генерировать значения на лету и тут же преобразовывать их в последовательность, возможно даже бесконечную.

Но, прежде чем увидеть, как это реализовать, рассмотрим пример 6.5, где представлена функция-расширение для `Int`, которая определяет, является ли число простым, пытаясь найти делитель в диапазоне от 2 до квадратного корня из числа, на который это число делится без остатка.

Пример 6.5. Проверка простых чисел

```
import kotlin.math.ceil
import kotlin.math.sqrt

fun Int.isPrime() =
    this == 2 || (2..ceil(sqrt(this.toDouble())).toInt())
        .none { divisor -> this % divisor == 0 }
```

Сначала функция проверяет, равно ли число 2. Если нет, то создает диапазон от 2 до квадратного корня из данного числа, округленного вверх до ближайшего целого числа. Для каждого числа в этом диапазоне функция `none` возвращает `true`, только если ни одно из значений не делит исходное число без остатка.

Тесты для проверки поведения функции `isPrime` включены в репозиторий с исходным кодом для этой книги.

Теперь предположим, что у вас есть некоторое целое число и вам требуется определить ближайшее простое число, которое больше его. Например, если дано число 6, то ближайшим большим простым числом будет 7. Если дано число 182, то ближайшим большим простым числом будет 191. Для 9973 ближайшим большим простым числом будет 10 007. Интересно отметить, что в этой задаче нет никакой возможности узнать, сколько чисел придется проверить, прежде чем будет найдено следующее простое число. Это одно из естественных применений последовательностей. Реализация функции `nextPrime` показана в примере 6.6.

Пример 6.6. Поиск ближайшего простого числа больше заданного

```
fun nextPrime(num: Int) =
    generateSequence(num + 1) { it + 1 } ❶
        .first(Int::isPrime) ❷
```

❶ Начать последовательность с числа, на 1 больше заданного, и генерировать каждое последующее число на 1 больше предыдущего

❷ Вернуть первое простое число

Функция `generateSequence` принимает два аргумента: начальное значение и функцию для вычисления каждого следующего значения в последовательности. Вот ее сигнатура:

```
fun <T : Any> generateSequence(
    seed: T?,
    nextFunction: (T) -> T?
): Sequence<T>
```

В нашем случае seed – это начальное число, следующее сразу за указанным, а функция nextFunction просто увеличивает текущее число на единицу. В соответствии с типичной идиомой Kotlin лямбда-выражение указывается после круглых скобок в вызове функции generateSequence. Функция first возвращает первое значение, которое удовлетворяет условию в лямбда-выражении, которое в данном случае является ссылкой на функцию-расширение isPrime.

В этом примере функция nextPrime генерирует бесконечную последовательность целых чисел, пока не найдет первое простое число, которое больше заданного. Функция first возвращает значение, а не последовательность, поэтому это – *терминальная* операция. Без терминальной операции никакие значения не будут обрабатываться последовательностью. В этом случае операции first передается лямбда-выражение – предикат (потому что возвращает логическое значение) – и последовательность продолжает генерировать значения, пока условие предиката не будет удовлетворено.

Смотри также

Рецепт 6.3, где исследуется бесконечная последовательность.

6.3. Управление бесконечными последовательностями

Задача

Обработать часть бесконечной последовательности.

Решение

Использовать генератор последовательностей, возвращающий null, или одну из функций последовательностей, такую как takeWhile.

Обсуждение

Последовательности похожи на потоки данных в Java. Они поддерживают промежуточные и терминальные операции. Промежуточные операции возвращают новые последовательности, а терминальные – нечто иное. Если конвейер из вызовов функций последовательностей не содержит терминальной операции, то никакие данные не будут течь через последовательность.

Функция firstNPrimes, показанная в примере 6.7, вычисляет первые N простых чисел, начиная с 2. Пример 6.7, показанный ниже, использует функцию nextPrime из примера 6.6, описанную в рецепте 6.2 и повторенную здесь для удобства:

```
fun nextPrime(num: Int) =
    generateSequence(num + 1) { it + 1 }
        .first(Int::isPrime)
```

Эта функция использует функцию-расширение isPrime из того же рецепта.

Напомню, что невозможно заранее узнать, сколько чисел нужно проверить, чтобы найти необходимое количество простых чисел, поэтому применение последовательности является естественным решением задачи.

Пример 6.7. Поиск первых N простых чисел

```
fun firstNPrimes(count: Int) =
    generateSequence(2, ::nextPrime) ❶
        .take(count)                 ❷
        .toList()                    ❸
```

❶ Бесконечная последовательность простых чисел, начинающаяся с 2

❷ Промежуточная операция, ограничивающая количество извлекаемых чисел

❸ Терминальная операция

Генерируемая последовательность бесконечна. Функция `take` – это промежуточная операция без сохранения состояния, которая возвращает последовательность, состоящую только из первых `count` значений исходной последовательности. Если просто вызвать эту функцию без вызова `toList` в конце, простые числа не будут вычисляться. У вас просто будет последовательность без каких-либо значений. Терминальная операция `toList` запускает фактические вычисления и возвращает результаты в виде списка.

В 2017 году даже этот неоптимизированный алгоритм, выполнявшийся на MacBook Pro, сгенерировал 10 000 простых чисел менее чем за 50 миллисекунд. Современные компьютеры – очень быстрые. К слову сказать, 10-тысячное простое число – это 104 729.

Другой способ усечения бесконечной последовательности – использовать функцию генерации, которая в конце концов возвращает `null`. Например, вместо запроса первых `N` простых чисел можно потребовать, чтобы все простые числа были меньше определенного значения. Соответствующая функция `primesLessThan` показана в примере 6-8.

Пример 6.8. Простые числа меньше определенного значения (версия 1)

```
fun primesLessThan(max: Int): List<Int> =
    generateSequence(2) { n -> if (n < max) nextPrime(n) else null }
        .toList().dropLast(1)
```

Функция, передаваемая в вызов `generateSequence`, сравнивает текущее значение с заданной максимальной величиной и, если оно меньше, вычисляет следующее простое число, иначе возвращает `null`, завершая последовательность.

Конечно, невозможно узнать, будет ли следующее простое число больше заданного предела, поэтому функция фактически создает список, включающий первое простое число выше предела. Затем функция `dropLast` обрезает получившийся список перед возвратом.

В большинстве случаев есть более простой способ решить ту же задачу. Например, в этой задаче, вместо того чтобы заставлять генерирующую функцию возвращать `null`, можно использовать функцию последовательностей `takeWhile`, как показано в примере 6.9.

Пример 6.9. Простые числа меньше определенного значения (версия 2)

```
fun primesLessThan(max: Int): List<Int> =
    generateSequence(2, ::nextPrime)
        .takeWhile { it < max }
        .toList()
```

Функция takeWhile продолжает извлекать значения из последовательности, пока заданный предикат не вернет true.

Оба описанных подхода работают хорошо, поэтому выбор между ними – это в основном дело вкуса.

6.4. Извлечение значений из последовательности

Задача

Требуется извлекать значения из последовательности, но через заданные интервалы.

Решение

Использовать функцию sequence в паре с функцией приостановки yield.

Обсуждение

sequence – еще одна функция, связанная с последовательностями. Ее сигнатура показана в примере 6.10.

Пример 6.10. Сигнатура функции sequence

```
fun <T> sequence(
    block: suspend SequenceScope<T>.() -> Unit
): Sequence<T>
```

Функция sequence генерирует последовательность, выполняя заданный блок кода. Блок кода – это лямбда-выражение без аргументов, возвращающее void и воздействующее на получателя типа SequenceScope.

Все это кажется сложным, но только до первого практического примера. Обычно последовательности создаются из существующих данных с помощью sequenceOf, из коллекций с помощью asSequence или генерируются с использованием generateSequence и лямбда-выражения. Однако в этом случае вы должны указать лямбда-выражение, производящее следующее значение в последовательности, которое вы вызываете в требуемый момент времени.

Отличной демонстрацией может послужить генерирование чисел Фибоначчи, как показано в примере 6.11, основанном на примере из документации с описанием стандартной библиотеки.

Пример 6.11. Генерирование чисел Фибоначчи в виде последовательности

```
fun fibonacciSequence() = sequence {
    var terms = Pair(0, 1)

    while (true) {
        yield(terms.first)
        terms = terms.second to terms.first + terms.second
    }
}
```

Лямбда-выражение, передаваемое в вызов функции sequence, начинается с создания пары Pair, содержащей первые два числа Фибоначчи, 0 и 1. Затем

оно использует бесконечный цикл для получения последующих значений. Каждый раз, когда создается новый элемент, функция `yield` возвращает элемент `first` полученной пары.

Функция `yield` – это одна из двух похожих функций, являющихся частью `SequenceScope`, получателя лямбда-выражения, предоставленного для передачи в вызов `sequence`. Сигнатуры `yield` и `yieldAll` и их перегруженных версий показаны в примере 6.12.

Пример 6.12. Функции yield и yieldAll из SequenceScope

```
abstract suspend fun yield(value: T)

abstract suspend fun yieldAll(iterator: Iterator<T>)
suspend fun yieldAll(elements: Iterable<T>)
suspend fun yieldAll(sequence: Sequence<T>)
```

Задача функции `yield` – передать значение итератору и приостановить его до тех пор, пока не будет запрошено следующее значение. Соответственно, в последовательности, генерируемой `suspend`-функцией, функция `yield` используется для вывода отдельных значений. Тот факт, что сама `yield` является `suspend`-функцией, означает, что она прекрасно работает в паре с сопрограммами. Другими словами, среда выполнения Kotlin может передать значение, а затем приостановить текущую сопрограмму, пока не будет запрошено следующее значение. Вот почему бесконечный цикл – цикл `while (true)` в примере 6.11 – возвращает значения одно за другим при вызове операцией `take` в примере 6.13.

Пример 6.13. Извлечение значений, генерируемых операцией sequence

```
@Test
fun `first 10 Fibonacci numbers from sequence`() {
    val fibs = fibonacciSequence()
        .take(10)
        .toList()

    assertEquals(listOf(0, 1, 1, 2, 3, 5, 8, 13, 21, 34), fibs)
}
```

Как нетрудно догадаться, `yieldAll` передает итератору несколько значений. В примере 6.14 показан пример, который приводится в документации Kotlin.

Пример 6.14. Функция yieldAll внутри sequence

```
vale sequence = sequence {
    val start = 0
    yield(start)                          ❶
    yieldAll(1..5 step 2)                   ❷
    yieldAll(generateSequence(8) { it * 3 }) ❸
}

println(sequence.take(8).toList())  // [0, 1, 3, 5, 8, 24, 72, 216]
```

❶ Вернет единственное значение (0)

❷ Вернет результат перебора диапазона (1, 3, 5)

❸ Вернет бесконечную последовательность, начинающуюся с 8, в которой каждое следующее значение является произведением предыдущего на 3

Результатом этого кода является последовательность 0, 1, 3, 5, 8, 24, 72, … При обращении к последовательности с помощью функции `take` она вернет ровно столько элементов, сколько запрошено.

Комбинация `yield` и `yieldAll` внутри `suspend`-функции позволяет настроить последовательность для получения любого желаемого набора сгенерированных значений.

Смотри также

Раздел о сопрограммах в главе 13, где ключевое слово `suspend` описывается более подробно.


ЛАНЬ®


ЛАНЬ®

Глава 7

Функции области видимости

Стандартная библиотека Kotlin содержит несколько функций, предназначенных для выполнения блока кода в контексте некоторого объекта. В частности, в этой главе обсуждаются функции `let`, `run`, `apply` и `also`.

7.1. Инициализация объекта с помощью apply после создания

Задача

Инициализировать объект перед использованием после вызова конструктора с аргументами.

Решение

Использовать функцию `apply`.

Обсуждение

В Kotlin есть несколько *функций области видимости*, которые можно применять к объектам. Функция `apply` – это функция-расширение, которая передает `this` как аргумент и также возвращает его. В примере 7.1 показана сигнатура функции `apply`.

Пример 7.1. Сигнатура функции apply

```
inline fun <T> T.apply(block: T.() -> Unit): T
```

Итак, функция `apply` – это функция-расширение для любого обобщенного типа `T`, которая вызывает указанный блок кода с `this` в качестве получателя и возвращает его по завершении.

Рассмотрим практический пример – задачу сохранения объекта в реляционной базе данных с помощью фреймворка Spring. В фреймворке Spring имеется класс `SimpleJdbcInsert`, основанный на `JdbcTemplate`, который помогает удалить шаблонный код JDBC в Java.

Пусть имеется сущность `Officer`, которая отображается в таблицу `OFFICERS` в базе данных. Написать SQL-оператор `INSERT` для такого класса не представляет большого труда, но есть одна сложность: если первичный ключ гене-

рируется базой данных во время сохранения, то его необходимо получить и записать в копию сущности, находящуюся в памяти. Для этого в классе `SimpleJdbcInsert` есть удобный метод `executeAndReturnKey`, который получает ассоциативный массив с именами и значениями столбцов и возвращает сгенерированное значение.

Используя функцию `apply`, функция `save` может получить экземпляр для сохранения и обновить его, записав новый ключ, как показано в примере 7.2.

Пример 7.2. Сохранение объекта данных и его обновление сгенерированным ключом

```
@Repository
class JdbcOfficerDAO(private val jdbcTemplate: JdbcTemplate) {

    private val insertOfficer = SimpleJdbcInsert(jdbcTemplate)
        .withTableName("OFFICERS")
        .usingGeneratedKeyColumns("id")

    fun save(officer: Officer) =
        officer.apply {
            id = insertOfficer.executeAndReturnKey(
                    mapOf("rank" to rank,
                          "first_name" to first,
                          "last_name" to last))
        }

    // ...
}
```


Экземпляр `Officer` передается в блок `apply` как `this`, поэтому его можно использовать для доступа к свойствам `rank`, `first` и `last`. Свойство `id` обновляется внутри блока `apply`, и обновленная сущность `Officer` возвращается вызывающему коду. При необходимости или желании к этому блоку можно добавить дополнительную инициализацию.

Блок `apply` удобно использовать, если результатом должен быть объект контекста (в этом примере – сущность `Officer`). Чаще всего он используется для дополнительной настройки уже созданных объектов.

7.2. Использование also для создания побочных эффектов

Задача

Вывести сообщение или сгенерировать другой побочный эффект, не прерывая выполнения кода.

Решение

Использовать функцию `also`.

Обсуждение

Функция `also` – это функция-расширение из стандартной библиотеки, реализация которой показана в примере 7.3.

Пример 7.3. Функция-расширение also

```
public inline fun <T> T.also(
    block: (T) -> Unit
): T
```

Как можно судить по определению функции also, она добавляется к любому обобщенному типу T и возвращает экземпляр этого типа после выполнения блока, переданного в аргументе. Чаще всего она используется для применения функции к объекту, как показано в примере 7.4.

Пример 7.4. Вывод и журналирование информации с помощью also

```
val book = createBook()
    .also { println(it) }
    .also { Logger.getAnonymousLogger().info(it.toString()) }
```

Внутри блока объект доступен по ссылке it.

Поскольку also возвращает объект контекста, несколько ее вызовов можно объединить в цепочку, как показано выше, где код сначала вывел название книги в консоль, а затем записал информацию о ней в журнал.

Несмотря на возможность составлять цепочки из нескольких вызовов, все же чаще всего функция also используется как часть серии вызовов бизнес-логики. Например, рассмотрим тест службы геокодирования, представленный в примере 7-5.

Пример 7.5. Тестирование службы геокодирования

```
class Site(val name: String,
           val latitude: Double,
           val longitude: Double)

// ... внутри тестового класса ...

@Test
fun `lat,lng of Boston, MA`() = service.getLatLng("Boston", "MA")
    .also { logger.info(it.toString()) } ❶
    .run {
        assertThat(latitude, `is`(closeTo(42.36, 0.01)))
        assertThat(longitude, `is`(closeTo(-71.06, 0.01)))
    }
```

❶ Журналирование как побочный эффект

Этот тест можно организовать по-разному, но такое использование also подразумевает, что главная цель этого кода – выполнение тестов, а вывод названия местоположения – это лишь побочный эффект. Обратите внимание, что использование функций области видимости преобразует весь тест в единое выражение, что позволяет использовать более короткий синтаксис.

Вызов also должен следовать раньше вызова run, потому что run возвращает значение лямбда-выражения, а не объект контекста.

Обратите внимание, что теоретически вызов run можно заменить вызовом apply, однако тесты JUnit должны возвращать Unit. Вызов run в примере 7.5 возвращает такой результат (потому что утверждения ничего не возвращают), а apply – нет, поскольку она возвращает объект контекста.

Смотри также

Рецепт 7.1, где обсуждается функция apply.

7.3. Использование функции let и оператора «Элвис»

Задача

Выполнить блок кода, только если ссылка не равна null, в противном случае вернуть значение по умолчанию.

Решение

Использовать функцию let с оператором безопасного вызова и с оператором «Элвис».

Обсуждение

Функция let – это функция-расширение из стандартной библиотеки для любого обобщенного типа T, реализация которой показана в примере 7.6.

Пример 7.6. Реализация функции let

```
public inline fun <T, R> T.let(
    block: (T) -> R
): R
```

Важно запомнить, что let возвращает результат выполнения блока, а не объект контекста. То есть фактически она преобразует объекты контекста подобно функции map. Допустим, что вам требуется написать функцию, которая принимает строку и заменяет первую букву в ней заглавной буквой, но при этом пустые строки и ссылки null должны обрабатываться особым образом, как показано в примере 7.7.

Пример 7.7. Преобразование первой буквы в строке с обработкой особых случаев

```
fun processString(str: String) =
    str.let {
        when {
            it.isEmpty() -> “Empty”
            it.isBlank() -> “Blank”
            else -> it.capitalize()
        }
    }
```

Обычно вполне достаточно вызвать функцию capitalize, но для значения null и пустых строк она не даст ничего полезного. Функция let позволяет заключить в блок условное выражение when, обрабатывающее все требуемые случаи и возвращающее «преобразованную» строку.

Ситуация становится еще более интересной, когда аргумент имеет тип, поддерживающий значение null, как показано в примере 7.8.

Пример 7-8. То же самое, но с аргументом, поддерживающим значение null

```
fun processNullableString(str: String?) =
    str?.let {              ❶
        when {
            it.isEmpty() -> “Empty”
            it.isBlank() -> “Blank”
            else -> it.capitalize()
        }
    } ?: “Null”             ❷
```

❶ Безопасный вызов let

❷ Оператор «Элвис» для обработки аргумента со значением null

Обе функции возвращают значение типа String, который легко определяется из тела функции.

В этом случае комбинация оператора безопасного вызова ?., функции let и оператора «Элвис» ?: помогает обрабатывать все возможные случаи. Это распространенная в Kotlin идиома обработки случаев, когда могут появляться значения null.

Многие Java API (например, RestTemplate или WebClient в Spring) возвращают значения null при отсутствии результата, и комбинация безопасного вызова, блока let и оператора «Элвис» позволяет эффективно обрабатывать их.

Смотри также

Рецепт 7.2, где обсуждается функция also. Рецепт 7.4, где демонстрируется применение функции let для замены временных переменных.

7.4. Использование let с временными переменными

Задача

Обработать результат вычислений без сохранения результата во временной переменной.

Решение

Добавить вызов let к вычислению и обработать результат в лямбда-выражении или в функции по ссылке.

Обсуждение

В документации с описанием функций области видимости на веб-сайте Kotlin показан интересный вариант использования функции let. Вариант этот (он приводится в примере 7.9) создает изменяемый список строк, получает их длины и фильтрует результат.

Пример 7.9. Пример использования let из электронной документации (перед рефакторингом)

```
// До
val numbers = mutableListOf("one", "two", "three", "four", "five")
val resultList = numbers.map { it.length }.filter { it > 3 }
println(resultList) ❶
```

❶ Результат присваивается временной переменной для вывода

После рефакторинга – замены временной переменной блоком let – код выглядит, как показано в примере 7.10.

Пример 7.10. После замены временной переменной функцией let

```
// После
val numbers = mutableListOf("one", "two", "three", "four", "five")
numbers.map { it.length }.filter { it > 3 }.let {
    println(it)
    // сюда можно добавить вызовы других функций,
    // если потребуется
}
```

Идея состоит в том, чтобы вместо присваивания результата временной переменной вызвать функцию let, использующую результат в качестве контекстной переменной, которую можно вывести (или сделать что-то еще) в блоке кода. Если достаточно просто вывести результат, то код можно сократить еще больше, как показано в примере 7.11.

Пример 7.11. Использование ссылки на функцию в блоке let

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
numbers.map { it.length }.filter { it > 3 }.let(::println)
```

Рассмотрим немного более интересный пример: класс, обращающийся к удаленной службе в Open Notify, которая возвращает количество космонавтов, находящихся в данный момент в космосе, как описано в рецепте 11.6. Служба возвращает данные в формате JavaScript Object Notation (JSON), а мы должны преобразовать эти данные в экземпляры классов, которые мы снова увидим в примере 11.17:

```
data class AstroResult(
    val message: String,
    val number: Number,
    val people: List<Assignment>
)

data class Assignment(
    val craft: String,
    val name: String
)
```

Код в примере 7.12 использует метод-расширение URL.readText и библиотеку Google Gson, чтобы преобразовать данные в формате JSON в экземпляр AstroResult.

Пример 7.12. Вывод имен космонавтов, находящихся в космосе

```
Gson().fromJson(
    URL(«http://api.open-notify.org/astros.json»).readText(),
        AstroResult::class.java
    ).people.map { it.name }.let(::println) ❶
```

❶ Функция let (или also) используется для вывода List<String>

В этом случае код в вызове Gson().fromJson преобразует данные из формата JSON в экземпляр AstroResult. Затем функция map преобразует экземпляры Assignment в список строк, представляющих имена космонавтов.

Вот вывод этой программы, полученный в августе 2019 г. (в одну строку):

```
[Alexey Ovchinin, Nick Hague, Christina Koch,
    Alexander Skvortsov, Luca Parmitano, Andrew Morgan]
```

В примере 7.12 вызов let можно заменить вызовом also. Разница лишь в том, что let возвращает результат выполнения блока (Unit в случае с println), а also возвращает объект контекста (List<String>). Ни то, ни другое не используется после печати, поэтому разница для данного примера не имеет значения. Использование also, возможно, выглядело бы идиоматичнее, потому что она чаще применяется для получения побочных эффектов, таких как вывод.

Смотри также

Рецепт 7.3, где описывается применение функции let с операторами безопасного вызова и «Элвис» для обработки возможных значений null. Рецепт 7.2, где описывается функция also.


ЛАНЬ®


ЛАНЬ®

Глава 8

Делегаты в Kotlin

Эта глава рассказывает о делегатах в языке Kotlin. Здесь вы узнаете, как использовать делегаты, имеющиеся в стандартной библиотеке, включая `lazy`, `observable`, `vetoable` и `notNull`, а также создавать свои собственные. *Делегаты классов* позволяют заменить наследование композицией, а *делегаты свойств* – использовать свойства из другого класса.

Помимо основных приемов, в этой главе также показана реализация некоторых стандартных делегатов в библиотеке, чтобы вы могли увидеть хорошие примеры идиоматического использования.

8.1. Реализация композиции делегированием

Задача

Создать класс, содержащий экземпляры других классов и делегирующий им выполнение операций.

Решение

Создать интерфейсы, содержащие методы делегирования, реализовать их в классах и создать класс-обертку, используя ключевое слово `by`.

Обсуждение

В современном объектно-ориентированном программировании наблюдается тенденция, когда предпочтение отдается композиции, а не наследованию[8], как способу добавления новых возможностей без образования тесных связей между классами. Ключевое слово `by` в языке Kotlin позволяет классу экспортировать все общедоступные функции во внутреннем объекте через методы контейнера.

Например, смартфон имеет в своем составе телефон, камеру и множество других компонентов. Если представить смартфон как *объект-обертку*, а телефон и камеру – как *внутренние* объекты, то цель создания класса смартфона можно определить как вызов соответствующих функций внутренних объектов.

[8] Не очень удачная шутка: родители Бетховена не давали ему семейных денег, требуя бросить писать музыку, но он все равно предпочел композицию наследству. (Честно говоря, это неправда.)

Чтобы реализовать такой класс в Kotlin, нужно определить интерфейсы, описывающие открытые методы во внутренних объектах. Рассмотрим для примера интерфейсы `Dialable` и `Snappable`, которые реализуют классы `Phone` и `Camera` (см. пример 8.1).

Пример 8.1. Интерфейсы и классы внутренних объектов

```
interface Dialable {
    fun dial(number: String): String
}

class Phone : Dialable {
    override fun dial(number: String) =
        “Dialing $number...”
}

interface Snappable {
    fun takePicture(): String
}

class Camera : Snappable {
    override fun takePicture() =
        “Taking picture...”
}
```

Теперь можно определить класс `SmartPhone`, который создает экземпляры телефона и камеры в конструкторе и делегирует им все общедоступные функции, как показано в примере 8.2.

Пример 8.2. SmartPhone делегирует выполнение операций внутренним экземплярам

```
class SmartPhone(
    private val phone: Dialable = Phone(),

    private val camera: Snappable = Camera()
) : Dialable by phone, Snappable by camera ❶
```

❶ Делегирование с помощью ключевого слова by

Теперь, создав экземпляр `SmartPhone`, можно вызывать все методы в `Phone` и `Camera`, как демонстрируют тесты в примере 8.3.

Пример 8.3. Тесты для SmartPhone

```
import org.junit.jupiter.api.Test
import org.junit.jupiter.api.Assertions.*

class SmartPhoneTest {
    private val smartPhone: SmartPhone = SmartPhone() ❶

    @Test
    fun `Dialing delegates to internal phone`() {
        assertEquals(“Dialing 555-1234...”,
            smartPhone.dial(“555-1234”))              ❷
    }
```

```
    @Test
    fun `Taking picture delegates to internal camera`() {
        assertEquals("Taking picture...",
            smartPhone.takePicture())                    ❷
    }
}
```

❶ Создание экземпляра SmartPhone вызовом конструктора без аргументов

❷ Вызов делегированных функций

Сами внутренние объекты (экземпляры Phone и Camera) недоступны из-за пределов SmartPhone; доступны только их общедоступные функции. Классы Phone и Camera могут иметь много других функций, но доступны только те, которые объявлены в соответствующих интерфейсах Dialable и Snappable. Необходимость определения дополнительных интерфейсов кажется напрасной работой, но она помогает сохранить отношения ясными.

Если сыграть в уже привычную игру в IntelliJ, скомпилировать программу на Kotlin в байт-код и затем декомпилировать ее в исходный код на Java, то получится фрагмент, показанный в примере 8.4.

Пример 8.4. Часть декомпилированного байт-кода класса SmartPhone

```
public final class SmartPhone implements Dialable, Snappable {
    private final Dialable phone;          ❶
    private final Snappable camera;        ❶

    public SmartPhone(@NotNull Dialable phone, @NotNull Snappable camera) {
        // ...
        this.phone = phone;
        this.camera = camera;
    }

    @NotNull
    public String dial(@NotNull String number) {
        return this.phone.dial(number);    ❷
    }

    @NotNull
    public String takePicture() {
        return this.camera.takePicture(); ❷
    }

    // ...
}
```

❶ Поля с типом интерфейсов

❷ Делегирующие методы

Внутренне класс SmartPhone определяет делегированные свойства с типами интерфейсов. Соответствующие экземпляры класса создаются в конструкторе, а делегирующие методы вызывают соответствующие методы в полях.

Смотри также

Рецепт 8.6, где рассказывается о делегировании свойств.

8.2. Использование делегата lazy

Задача

Отложить инициализацию свойства до момента, когда это действительно станет необходимо.

Решение

Использовать делегат `lazy` из стандартной библиотеки.

Обсуждение

Ключевое слово `by` используется в Kotlin для обозначения свойств, методы чтения и записи которых реализованы другим объектом, который называют *делегатом*. В стандартной библиотеке есть несколько функций-делегатов. Одна из самых популярных – `lazy`.

Чтобы использовать ее, необходимо определить лямбда-выражение в форме `() -> T`, цель которого – вычислить значение при первой попытке обратиться к свойству, как показано в примере 8.5.

Пример 8.5. Сигнатуры функции `lazy`

```
fun <T> lazy(initializer: () -> T): Lazy<T>                    ❶

fun <T> lazy(                                                  ❷
    mode: LazyThreadSafetyMode,
    initializer: () -> T
): Lazy<T>

fun <T> lazy(lock: Any?, initializer: () -> T): Lazy<T> ❸
```

❶ По умолчанию синхронизируется по самому себе

❷ Определяет, как синхронизируется инициализация между несколькими потоками выполнения

❸ Использует для синхронизации указанный объект

Версии без свойства `mode` по умолчанию получают значение `LazyThreadSafetyMode.SYNCHRONIZED`. Если лямбда-выражение `initializer` сгенерирует исключение, будет выполнена повторная попытка инициализировать значение при следующем доступе. Простейший способ применения делегата `lazy` показан в примере 8.6.

Пример 8.6. Откладывание инициализации свойства до первой попытки обращения к нему

```
val ultimateAnswer: Int by lazy {
    println("computing the answer")
    42
}
```

Идея состоит в том, чтобы отложить инициализацию значения `ultimateAnswer` до первого обращения к нему, в ходе которого вычисляется лямбда-выражение. `lazy` – это функция, которая принимает лямбда-выражение и возвращает экземпляр `Lazy<Int>`, который выполнит лямбда-выражение при первом обращении к свойству.

То есть следующий код выведет текст «computing the answer» только один раз:

```
println(ultimateAnswer)
println(ultimateAnswer)
```

Первое обращение к `ultimateAnswer` выполнит лямбда-выражение и вернет число `42`, которое будет сохранено в переменной. Внутренне Kotlin сгенерирует специальное свойство с именем `myAnswer$delegate` типа `Lazy`, которое будет использовано как кеш для хранения значения.

Аргумент типа `LazyThreadSafetyMode` принимает элемент перечисления, один из следующих:

`SYNCHRONIZED`

требует использования блокировки, чтобы гарантировать возможность инициализации экземпляра `Lazy` только из одного потока выполнения;

`PUBLICATION`

функцию-инициализатор можно вызывать несколько раз, но использовано будет только первое возвращаемое значение;

`NONE`

блокировка не используется.

Если в аргументе `lock` указан объект, делегат синхронизируется по этому объекту при вычислении значения. В противном случае он синхронизируется по самому себе.

Делегат `lazy` удобно использовать, когда требуется создавать экземпляры сложных классов, но суть от этого не меняется.

Кроме того, реализация `lazy` в стандартной библиотеке не следует типичному шаблону реализации остальных делегатов. Функция `lazy` – это функция верхнего уровня, в то время как большинство остальных делегатов являются частью экземпляра `Delegates`, обсуждаемого в других рецептах в этой главе.

8.3. Гарантия неравенства значению null

Задача

Вызвать исключение, если значение не было инициализировано до первого обращения к нему.

Решение

Использовать функцию `notNull` в роли делегата, который сгенерирует исключение, если значение не было инициализировано до первого обращения.

Обсуждение

Обычно свойства в классах Kotlin инициализируются во время создания экземпляров. Один из способов отсрочить инициализацию – использовать функцию `notNull`, которая предоставляет делегата, генерирующего исключение, если свойство не было инициализировано до обращения к нему.

Пример 8.7 объявляет свойство shouldNotBeNull, которое должно быть инициализировано где-то еще перед его использованием.

Пример 8.7. Требование инициализации перед обращением, без определения способа инициализации

```
var shouldNotBeNull: String by Delegates.notNull<String>()
```

Тесты в примере 8.8 показывают, что если попытаться получить доступ к свойству до присваивания ему некоторого значения, Kotlin сгенерирует исключение IllegalStateException.

Пример 8.8. Проверка поведения делегата notNull

```
@Test
fun `uninitialized value throws exception`() {
    assertThrows<IllegalStateException>{ shouldNotBeNull }
}

@Test
fun `initialize value then retrieve it`() {
    shouldNotBeNull = "Hello, World!"
    assertDoesNotThrow { shouldNotBeNull }
    assertEquals("Hello, World!", shouldNotBeNull)
}
```

Это поведение достаточно прямолинейное, но если заглянуть в реализацию в *Delegates.kt* в стандартной библиотеке, можно заметить немало интересного. Сокращенная версия этого файла показана в примере 8.9.

Пример 8.9. Реализация в стандартной библиотеке

```
object Delegates {                                                    ❶
    fun <T : Any> notNull(): ReadWriteProperty<Any?, T> = NotNullVar() ❷

    // ... другие функции обсуждаются в других рецептах...
}

private class NotNullVar<T : Any>() : ReadWriteProperty<Any?, T> {    ❸
    private var value: T? = null

    override fun getValue(thisRef: Any?, property: KProperty<*>): T {
        return value ?: throw IllegalStateException(
            "Property ${property.name} should be initialized before get.")
    }

    override fun setValue(thisRef: Any?, property: KProperty<*>, value: T) {
        this.value = value
    }
}
```

❶ Delegates – это синглтон – объект-одиночка (объект, а не класс)

❷ Фабричный метод, возвращающий экземпляр класса NotNullVar

❸ Приватный класс, реализующий ReadWriteProperty

Ключевое слово object используется для определения единственного экземпляра Delegates, поэтому функция notNull действует подобно статической

функции в Java. Этот фабричный метод создает экземпляр приватного класса `NotNullVar`, который реализует интерфейс `ReadWriteProperty`.

Как обсуждается в рецепте 8.6, при разработке своих делегатов не требуется реализовать этот интерфейс (или родственный ему интерфейс `ReadOnlyProperty`, определяющий неизменяемое свойство), но требуется добавить два показанных метода. Функция `setValue` в `NotNullVar` просто сохраняет переданное значение, а функция `getValue` сравнивает значение с `null` и либо возвращает его, либо генерирует исключение `IllegalStateException`[9].

Эта комбинация класса-синглтона, фабричного метода и приватного класса реализации – обычная идиома в Kotlin. Если вы решите написать своего делегата, используйте этот шаблон.

8.4. Использование делегатов observable и vetoable

Задача

Перехватить попытку изменения свойства и, если потребуется, отменить ее.

Решение

Использовать функцию `observable` для определения попытки изменения и функцию `vetoable` с лямбда-выражением, принимающим решение об отмене изменения.

Обсуждение

Функции `observable` и `vetoable` находятся в объекте `Delegates`, о котором рассказывалось в рецепте 8.3. Они просты в использовании, но их реализации демонстрируют рекомендуемый шаблон для разработки своих делегатов.

Однако, прежде чем вдаваться в подробности, рассмотрим порядок использования этих функций. В примере 8.10 показаны сигнатуры этих двух функций.

Пример 8.10. Сигнатуры функций observable и vetoable

```
fun <T> observable(
    initialValue: T,
    onChange: (property: KProperty<*>, oldValue: T, newValue: T) -> Unit
): ReadWriteProperty<Any?, T>

fun <T> vetoable(
    initialValue: T,
    onChange: (property: KProperty<*>, oldValue: T, newValue: T) -> Boolean
): ReadWriteProperty<Any?, T>
```

Обе фабричные функции принимают начальное значение типа `T` и лямбда-выражение и возвращают экземпляр класса, реализующего интерфейс `ReadWriteProperty`. Как показывает пример 8.11, использовать их совсем несложно.

[9] Не самая удачная шутка: автор живет в восточном Коннектикуте, поэтому `IllegalStateException` означает «Нью-Йорк».

Пример 8.11. Использование функций observable и vetoable

```
var watched: Int by Delegates.observable(1) { prop, old, new ->
    println("${prop.name} changed from $old to $new")
}

var checked: Int by Delegates.vetoable(0) { prop, old, new ->
    println("Trying to change ${prop.name} from $old to $new")
    new >= 0
}
```

Переменная `watched` имеет тип `Int` и инициализируется значением 1. Всякий раз, когда она изменяется, выводится сообщение, показывающее старое и новое значения. Переменная `checked` тоже имеет тип `Int` и инициализируется значением 0. Но для этой переменной разрешены лишь неотрицательные значения. Лямбда-выражение в аргументе возвращает `true`, только если новое значение больше или равно 0.

Тест для переменной `watched`, представленный в примере 8.12, демонстрирует, что ее значение изменяется в точности так, как ожидается.

Пример 8.12. Тест для переменной watched

```
@Test
fun `watched variable prints old and new values`() {
    assertEquals(1, watched)
    watched *= 2
    assertEquals(2, watched)
    watched *= 2
    assertEquals(4, watched)
}
```

Этот тест выведет в консоль:

```
watched changed from 1 to 2
watched changed from 2 to 4
```

Тест для переменной `checked` с делегатом `vetoable`, представленный в примере 8.13, демонстрирует, что она принимает только значения, большие или равные 0.

Пример 8.13. Тест для переменной checked

```
@Test
fun `veto values less than zero`() {
    assertAll(
        { assertEquals(0, checked) },
        { checked = 42; assertEquals(42, checked) },
        { checked = -1; assertEquals(42, checked) },
        { checked = 17; assertEquals(17, checked) }
    )
}
```

Попытки присвоить переменной `checked` значения 42 и 17 заканчиваются успехом, но попытка присвоить –1 отклоняется.

Обе функции просты в использовании, но, опять же, их реализации представляют особый интерес. Так же как `notNull`, функции `observable` и `vetoable` являются фабричными функциями в синглтоне `Delegates`, как показано в примере 8.14.

Пример 8.14. Фабричные функции в Delegates

```
object Delegates {
    // ... другие ...

    inline fun <T> observable(initialValue: T,
        crossinline onChange: (property: KProperty<*>,
            oldValue: T, newValue: T) -> Unit): ReadWriteProperty<Any?, T> =
        object : ObservableProperty<T>(initialValue) {
            override fun afterChange(property: KProperty<*>,
            oldValue: T, newValue: T) = onChange(property, oldValue, newValue)
        }

    inline fun <T> vetoable(initialValue: T,
        crossinline onChange: (property: KProperty<*>,
            oldValue: T, newValue: T) -> Boolean): ReadWriteProperty<Any?, T> =
        object : ObservableProperty<T>(initialValue) {
            override fun beforeChange(property: KProperty<*>,
            oldValue: T, newValue: T): Boolean =
                onChange(property, oldValue, newValue)
        }
}
```

Реализации, мягко говоря, выглядят сложными. Первое, что следует отметить, – обе функции возвращают объект типа `ObservableProperty`. Этот класс показан в примере 8.15.

Пример 8.15. Класс ObservableProperty для передачи делегата

```
abstract class ObservableProperty<T>(initialValue: T) : ReadWriteProperty<Any?, T> {
    private var value = initialValue

    protected open fun beforeChange(property: KProperty<*>,
        oldValue: T, newValue: T): Boolean = true

    protected open fun afterChange(property: KProperty<*>,
        oldValue: T, newValue: T): Unit {}

    override fun getValue(thisRef: Any?, property: KProperty<*>): T {
        return value
    }

    override fun setValue(thisRef: Any?, property: KProperty<*>, value: T) {
        val oldValue = this.value
        if (!beforeChange(property, oldValue, value)) {
            return
        }
        this.value = value
        afterChange(property, oldValue, value)
    }
}
```

Класс хранит свойство любого обобщенного типа `T` и реализует интерфейс `ReadWriteProperty`. Это означает, что для него необходимо предоставить функции `getValue` и `setValue` с указанными сигнатурами. В данном случае `getValue` просто возвращает свойство.

Функция setValue выглядит несколько интереснее. Она запоминает текущее значение, а затем вызывает метод beforeChange. Если он вернет true (значение по умолчанию равно true), то свойство изменяется и вызывается функция afterChange, которая по умолчанию ничего не делает.

Это абстрактный класс с открытыми функциями beforeChange и afterChange, которые также отмечены как protected. Обе имеют реализации по умолчанию, но подклассы могут переопределять эти функции.

Вот тут и пригодятся реализации из примера 8.14. Функция observable создает объект, который расширяет ObservableProperty и переопределяет функцию afterChange, замещая ее лямбда-выражением onChange. Поскольку beforeChanged не переопределяется, эта функция просто возвращает true, гарантируя изменение свойства.

Функция vetoable тоже создает объект из класса, расширяющего ObservableProperty, но в этом случае переопределяется только функция beforeChanged. Лямбда-выражение, заменяющее ее, передается в аргументе и должно возвращать логическое значение, которое определяет возможность изменения свойства.

И снова работа по созданию класса ObservableProperty для реализации интерфейса ReadWriteProperty, но с дополнительными методами жизненного цикла, упрощает реализацию функций observable и vetoable.

inline и crossinline

Ключевое слово inline требует от компилятора избегать создания нового объекта только для вызова функции и заменять вызов фактическим исходным кодом, составляющим тело функции.

Иногда встраиваемые (inline) функции передаются в лямбда-выражения как параметры и должны выполняться в другом контексте, например в контексте локального объекта или вложенной функции. Такой «нелокальный» поток управления недопустим в лямбда-выражениях. В показанных примерах лямбда-выражение onChange выполняется в контексте функций observable или vetoable, а не класса, расширяющего ObservableProperty, поэтому необходимо использовать модификатор crossinline.

Комбинация фабричных функций внутри синглтона, которые настраивают экземпляр класса делегата, является мощным шаблоном.

8.5. Использование ассоциативных массивов в роли делегатов

Задача

Использовать ассоциативный массив для инициализации объекта.

Решение

Ассоциативные массивы в Kotlin уже реализуют функции getValue и setValue, необходимые делегатам.

Обсуждение

Если значения, необходимые для инициализации объекта, находятся в ассоциативном массиве, то можно автоматически делегировать свойства класса этому ассоциативному массиву. Например, пусть есть класс `Project`, как показано в примере 8.16.

Пример 8.16. Класс данных Project

```
data class Project(val map: MutableMap<String, Any?>) {
    val name: String by map        ❶
    var priority: Int by map       ❶
    var completed: Boolean by map  ❶
}
```

❶ Делегирование аргументу `map`

В этом случае конструктор `Project` принимает аргумент типа `MutableMap` и инициализирует свойства значениями его ключей. Для создания экземпляра типа `Project` требуется ассоциативный массив, как показано в примере 8.17.

Пример 8.17. Создание экземпляра Project на основе ассоциативного массива

```
@Test
fun `use map delegate for Project`() {
    val project = Project(
        mutableMapOf(
            “name” to “Learn Kotlin”,
            “priority” to 5,
            “completed” to true))

    assertAll(
        { assertEquals(“Learn Kotlin”, project.name) },
        { assertEquals(5, project.priority) },
        { assertTrue(project.completed) }
    )
}
```

Это возможно, потому что `MutableMap` имеет функции-расширения `setValue` и `getValue` с соответствующими сигнатурами, необходимые для того, чтобы быть делегатом `ReadWriteProperty`.

Однако у кого-то из вас может возникнуть вопрос: зачем нужен дополнительный уровень косвенности? Иначе говоря, почему бы просто не сделать свойства частью конструктора и не использовать ассоциативный массив? Как отмечается в документации, этот механизм может использоваться в таких приложениях, как «анализ JSON или выполнение других динамических действий».

Логично. Тест в примере 8.18 предполагает, что необходимые свойства находятся в строке JSON, которая здесь для простоты жестко запрограммирована. Для анализа строки используется библиотека Google Gson, а полученный в результате ассоциативный массив применяется для создания экземпляра `Project`.

Пример 8.18. Получение свойств Project из строки JSON

```
private fun getMapFromJSON() =                    ❶
    Gson().fromJson<MutableMap<String, Any?>>(
        """{ "name":"Learn Kotlin", "priority":5, "completed":true}""",
        MutableMap::class.java)

@Test
fun `create project from map parsed from JSON string`() {
    val project = Project(getMapFromJSON())       // ❷
        assertAll(
            { assertEquals("Learn Kotlin", project.name) },
            { assertEquals(5, project.priority) },
            { assertTrue(project.completed) }
    )
}
```

❶ Анализ строки JSON и извлечение из нее ассоциативного массива со свойствами

❷ Использование ассоциативного массива для создания экземпляра `Project`

Функция `fromJson` из библиотеки Gson принимает строку и тип, поэтому в Kotlin можно указать обобщенный тип, как показано в примере. Получившийся в результате ассоциативный массив будет иметь верный тип для представления значений в роли делегата.

8.6. Создание собственных делегатов

Задача

Определить свойства в данном классе так, чтобы при обращении к ним использовались методы чтения и записи из другого класса.

Решение

Определить свои делегаты свойств, создав класс, реализующий интерфейс `ReadOnlyProperty` или `ReadWriteProperty`.

Обсуждение

Обычно свойство класса работает в паре с внутренним полем, но это не обязательно. Вместо чтения или изменения значения в поле эти операции можно делегировать другому объекту. Чтобы создать свой делегат свойства, необходимо реализовать функции из интерфейса `ReadOnlyProperty` или `ReadWriteProperty`.

Определения этих интерфейсов показаны в примере 8.19.

Пример 8.19. Интерфейсы ReadOnlyProperty и ReadWriteProperty

```
interface ReadOnlyProperty<in R, out T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
}

interface ReadWriteProperty<in R, T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
    operator fun setValue(thisRef: R, property: KProperty<*>, value: T)
}
```

Интересно отметить, что для создания делегата не нужно реализовывать ни один из этих интерфейсов. Достаточно просто определить функции `getValue` и `setValue` с показанными сигнатурами.

В стандартной документации приводится тривиальный пример делегата, включающий класс с именем `Delegate` (см. пример 8.20).

Пример 8.20. Класс Delegate из стандартной документации

```
class Delegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return "$thisRef, thank you for delegating '${property.name}' to me!"
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
        println("$value has been assigned to '${property.name}' in $thisRef.")
    }
}
```

Чтобы задействовать этого делегата, нужно создать класс или переменную, делегирующую чтение/запись этому классу, а затем использовать эту переменную, как показано в примере 8.21.

Пример 8.21. Использование класса Delegate

```
class Example {
    var p: String by Delegate()
}

fun main() {
    val e = Example()
    println(e.p)
    e.p = "NEW"
}
```

Этот код выведет следующее:

```
delegates.Example@4c98385c, thank you for delegating 'p' to me!
NEW has been assigned to 'p' in delegates.Example@4c98385c.
```

Строго говоря, создавать свойство класса совсем необязательно. Начиная с версии Kotlin 1.1 также можно делегировать локальные переменные.

Стандартная библиотека включает несколько делегатов. В рецепте 8.3 показан код функции `notNull`, которая создает приватный класс `NotNullVar`, как показано в примере 8.9.

Еще одним примером может служить инструмент сборки Gradle, который предоставляет предметно-ориентированный язык Kotlin DSL, позволяющий взаимодействовать с контейнерами через делегированные свойства. В Gradle есть два основных источника свойств. Один из них – набор свойств, связанных с самим проектом (экземпляр класса `org.gradle.api.Project`), и другой – дополнительные свойства `extra`, которые можно использовать во всем проекте.

Пусть есть файл сборки `build.gradle.kts`. Оба типа свойств можно создать и использовать, как показано в примере 8.22.

Пример 8.22. Создание и использование свойств проекта и дополнительных свойств в Gradle Kotlin DSL

```
val myProperty: String by project                    ❶
val myNullableProperty: String? by project           ❷

val myNewProperty by extra("initial value")          ❸
val myOtherNewProperty by extra { "lazy initial value" } ❹
```

❶ Создать свойство проекта с именем `myProperty`

❷ Создать свойство, допускающее значение `null`

❸ Создать и инициализировать новое дополнительное свойство с именем `myNewProperty`

❹ Создать свойство, инициализирующееся при первом обращении

Свойства проекта можно установить из командной строки, используя синтаксис `-PmyProperty = value`, или определить в файле *gradle.properties*. Дополнительные свойства, определенные как показано в примере 8.22, также инициализируются либо заданным аргументом, либо лямбда-выражением, которое вычисляется при первом обращении.

Создать своего делегата свойства достаточно просто, но, как показывает остальная часть этой главы, на практике чаще используются существующие делегаты, либо из стандартной библиотеки, либо реализованные третьими сторонами, такими как Gradle.

Глава 9

Тестирование

9.1. Настройка жизненного цикла тестового класса

Задача

Создавать экземпляры тестов JUnit 5 только для каждого класса, а не для каждой тестовой функции, как принято по умолчанию.

Решение

Использовать аннотацию `@TestInstance` или установить свойство жизненного цикла по умолчанию в файле *junitplatform.properties*.

Обсуждение

Этот рецепт, как и многие другие в данной главе, основан на публикации в блоге Филиппа Хауэра (Philipp Hauer) под названием «Best Practices for Unit Testing in Kotlin» (https://oreil.ly/v82OQ).

В JUnit 4 по умолчанию новый экземпляр тестового класса создается для каждого тестового метода. Это гарантирует повторную инициализацию атрибутов тестового класса, что делает сами тесты независимыми. Правда, при этом код инициализации выполняется для каждого теста.

Чтобы избежать этого, в Java свойства класса можно пометить как статические (`static`), а код инициализации поместить в статический метод, отмеченный аннотацией `@BeforeClass`, который будет выполняться только один раз.

В качестве примера рассмотрим тест JUnit 4 для `java.util.List`, показанный в примере 9.1.

Пример 9.1. Тест JUnit 4 для проверки работы списков

```
public class JUnit4ListTests {
    private static List<String> strings =                    ❶
        Arrays.asList("this", "is", "a", "list", "of", "strings");

    private List<Integer> modifiable = new ArrayList<>(); ❷

    @BeforeClass                                             ❶
    public static void runBefore() {
        System.out.println(“BeforeClass: “ + strings);
    }
    @Before
    public void initialize() {                               ❷
        System.out.println("Before: " + modifiable);
        modifiable.add(3);
        modifiable.add(1);
        modifiable.add(4);
        modifiable.add(1);
        modifiable.add(5);
    }

    @Test
    public void test1() {
        // ...
    }

    @Test
    public void test2() {
        // ...
    }

    @Test
    public void test3() {
        // ...
    }

    @After                                                   ❷
    public void finish() {
        System.out.println(“After: “ + modifiable);
    }

    @AfterClass                                              ❶
    public static void runAfter() {
        System.out.println("AfterClass: " + strings);
    }
}
```

❶ Выполняется один раз для всего класса

❷ Выполняется для каждого тестового метода

Тестовый класс имеет два атрибута: `strings` и `modifiable`. Список `modifiable` используется для демонстрации жизненного цикла. В точке объявления атрибута он инициализируется как пустой список, а затем заполняется в методе `initialize` с аннотацией `@Before`. Цель этого метода – показать, что первоначально список пуст, а затем заполняется. Второй раз содержимое списка выводится

в методе finish с аннотацией @After, чтобы показать, что теперь он содержит желаемые элементы.

Список strings тоже инициализируется в точке объявления атрибута. Но, чтобы инициализация не выполнялась перед вызовом каждого тестового метода, он отмечен как статический (static). Методы жизненного цикла @BeforeClass и @AfterClass используются здесь, чтобы показать, что коллекция strings заполняется правильно.

Добиться того же поведения в Kotlin труднее, потому что в Kotlin нет ключевого слова static. Простейшее решение – использовать объект-компаньон, как показано в примере 9.2.

Пример 9.2. Тест JUnit 4 для проверки работы списков в Kotlin (см. более удачное решение с использованием JUnit 5)

```
class JUnit4ListTests {
    companion object {              ❶
        @JvmStatic                  ❷
        private val strings = listOf("this", "is", "a", "list", "of", "strings")

        @BeforeClass                ❶
        @JvmStatic                  ❷
        fun runBefore() {
            println("BeforeClass: $strings")
        }

        @AfterClass                 ❶
        @JvmStatic                  ❷
        fun runAfter() {
            println("AfterClass: $strings")
        }
    }

    private val modifiable = ArrayList<Int>()

    @Before
    fun initialize() {
        println("Before: $modifiable")
        modifiable.add(3)
        modifiable.add(1)
        modifiable.add(4)
        modifiable.add(1)
        modifiable.add(5)
    }

    @Test
    fun test1() {
        // ...
    }

    @Test
    fun test2() {
        // ...
    }
```

```
    @Test
    fun test3() {
        // ...
    }

    @After
    fun finish() {
        println("After: $modifiable")
    }
}
```

❶ Для выполнения операций один раз при создании экземпляра класса используется объект-компаньон

❷ Добавляет модификатор `static` в сгенерированный байт-код Java

Объект-компаньон используется, чтобы гарантировать, что коллекция `strings` будет создаваться и заполняться только один раз. Методы `@BeforeClass` и `@AfterClass` используются внутри объекта-компаньона с той же целью. Обратите внимание, что если бы инициализация списка `strings` выполнялась в методе `initialize`, то его следовало бы объявить с модификатором `lateinit`, причем как `var`, а не `val`:

```
class JUnit4ListTests {
    companion object {
        @JvmStatic
        private lateinit var strings

        @BeforeClass
        @JvmStatic
        fun runBefore() {
            strings = listOf("this", "is", "a", "list", "of", "strings")
        }

    // ...
}
```

В случаях, когда это действительно необходимо (например, при тестировании более сложного объекта, чем простой список, требующего дополнительной настройки), использование `var` делает код на Kotlin еще менее идиоматическим.

К счастью, JUnit 5 предлагает более простое решение. JUnit 5 позволяет управлять жизненным циклом самого тестового класса, предлагая аннотацию `@TestInstance`. В примере 9.3 показана более удачная реализация тестов списков.

Пример 9.3. Тест JUnit 5 для проверки работы списков в Kotlin (предпочтительный вариант)

```
import org.junit.jupiter.api.*
import org.junit.jupiter.api.Assertions.assertEquals

@TestInstance(TestInstance.Lifecycle.PER_CLASS)          ❶
class JUnit5ListTests {
    private val strings =                                ❷
        listOf("this", "is", "a", "list", "of", "strings")

    private lateinit var modifiable : MutableList<Int> ❸
```

```
@BeforeEach
fun setUp() {
    modifiable = mutableListOf(3, 1, 4, 1, 5)    ❸
    println("Before: $modifiable")
}

@AfterEach
fun finish() {
    println("After: $modifiable")
}

@Test
fun test1() {
    // ...
}
@Test
fun test2() {
    // ...
}

@Test
fun test3() {
    // ...
}
```

❶ Для всех тестов создается единственный экземпляр тестового класса

❷ Создается и заполняется только один раз

❸ Повторная инициализация выполняется перед каждым тестом

Это гораздо более идиоматичный код. Если установить жизненный цикл `PER_CLASS`, то будет создаваться только один экземпляр тестового класса, независимо от количества тестовых методов. Это означает, что атрибут `strings` можно создать и заполнить как обычно, используя модификатор `val`.

Сложность все еще возникает, если атрибут необходимо повторно инициализировать перед каждым тестом. Для этого, как показано в примере тестового класса, все еще можно использовать методы `@BeforeEach` и `@AfterEach`, правда, сам атрибут должен быть объявлен с модификаторами `lateinit` и `var`. Однако это особенность данного конкретного класса, потому что для создания и заполнения списка мы используем функцию `mutableList`. Для более сложных (и более интересных) объектов можно создавать их экземпляры, а затем использовать метод `apply` для их настройки, как показано в следующем рецепте.

JUnit 5 позволяет установить жизненный цикл теста для всех тестов в файле свойств, чтобы избавиться от необходимости повторять аннотацию `@TestInstance` в каждом тесте. Если создать файл с именем *junit-platform.properties* в пути к классам (обычно в папке *src/test/resources*), то достаточно добавить в него одну строку, показанную в примере 9.4.

Пример 9.4. Настройка жизненного цикла для всех тестов в проекте

```
junit.jupiter.testinstance.lifecycle.default = per_class
```

Единственный недостаток – тот, кто будет читать тестовый класс, должен не забыть заглянуть в этот файл, потому что по умолчанию JUnit 5 все так же создает экземпляр класса для каждой тестовой функции.

9.2. Использование классов данных в тестах

Задача

Проверить несколько свойств объекта, не раздувая код.

Решение

Создать класс данных, включающий все необходимые свойства.

Обсуждение

Классы данных в Kotlin автоматически получают методы `equals`, `toString`, `hashCode`, `copy` и `componentN`. Это делает их идеальным средством упаковки свойств для тестов.

Допустим, у нас есть служба, которая возвращает информацию о книгах по номерам ISBN. Класс `Book` – это класс данных, определение которого показано в примере 9.5.

Пример 9.5. Класс данных Book

```
data class Book(
    val isbn: String,
    val title: String,
    val author: String,
    val published: LocalDate
)
```

Возьмем для примера конкретную книгу и протестируем ее вручную, проверив все свойства, как показано в примере 9.6.

Пример 9.6. Тестирование свойств книги вручную (утомительно)

```
@Test
internal fun `test book the hard way`() {
    val book = service.findBookById("1935182943")
    assertThat(book.isbn, `is`("1935182943"))
    assertThat(book.title, `is`("Making Java Groovy"))
    assertThat(book.author, `is`("Ken Kousen"))
    assertThat(book.published, `is`(LocalDate.of(2013, Month.SEPTEMBER, 30)))
}
```

Однако такой подход требует явно писать инструкции проверки всех свойств. Другая проблема заключается в том, что если первая инструкция потерпит неудачу, это приведет к сбою всего теста, поэтому некоторые свойства могут остаться непроверенными. К счастью, в JUnit 5 был добавлен метод `assertAll`, который принимает список аргументов переменной длины (`vararg`) с экземплярами `Executable`, где `Executable` – это функциональный интерфейс, который не принимает никаких аргументов и ничего не возвращает. Преимущество функции `assertAll` заключается в том, что она выполнит все экземпляры `Executable`, даже если какие-то из них потерпят неудачу.

В примере 9.7 показана измененная версия предыдущего теста, использующая эту новую возможность.

Пример 9.7. Использование assertAll из JUnit 5 для тестирования всех свойств

```
@Test
fun `use JUnit 5 assertAll`() {
    val book = service.findBookById("1935182943")
    assertAll("check all properties of a book",
        { assertThat(book.isbn, `is`("1935182943")) },
        { assertThat(book.title, `is`("Making Java Groovy")) },
        { assertThat(book.author, `is`("Ken Kousen")) },
        { assertThat(book.published,
                    `is`(LocalDate.of(2013, Month.SEPTEMBER, 30))) })
}
```

Обратите внимание на использование лямбда-выражений для представления экземпляров Executable. Все лямбда-выражения в этом примере схожи в том, что ни одно из них не имеет аргументов, а используемая функция assertThat возвращает void.

И все равно пришлось написать отдельные тесты для всех свойств, и это раздражает. Поскольку классы данных в Kotlin уже имеют правильно реализованный метод equals, этот процесс можно упростить, как показано в примере 9.8.

Пример 9.8. Использование класса данных Book для тестирования

```
@Test
internal fun `use data class`() {
    val book = service.findBookById("1935182943")
    val expected = Book(isbn = "1935182943",
        title = "Making Java Groovy",
        author = "Ken Kousen",
        published = LocalDate.of(2013, Month.SEPTEMBER, 30))

    assertThat(book, `is`(expected)) ❶
}
```

❶ Единственная инструкция проверки выполняет всю работу

Теперь инструкция проверки использует метод equals класса данных.

Для тестирования коллекции экземпляров можно использовать методы сравнения из библиотеки Hamcrest, как показано в примере 9.9.

Пример 9.9. Тестирование коллекции книг

```
@Test
internal fun `check all elements in list`() {
    val found = service.findAllBooksByIds(
        "1935182943", "1491947020", "149197317X")

    val expected = arrayOf(
        Book("1935182943", "Making Java Groovy",
             "Ken Kousen", LocalDate.parse("2013-09-30")),
        Book("1491947020", "Gradle Recipes for Android",
             "Ken Kousen", LocalDate.parse("2016-06-17")),
        Book("149197317X", "Modern Java Recipes",
             "Ken Kousen", LocalDate.parse("2017-08-26")))

    assertThat(found, arrayContainingInAnyOrder(*expected))
}
```

Метод `arrayContainingInAnyOrder` из библиотеки Hamcrest принимает список аргументов переменной длины, поэтому в этом примере используется оператор *распаковывания* массива, `*expected`, для разделения массива на отдельные записи.

9.3. Использование вспомогательных функций с аргументами по умолчанию

Задача

Быстро создать тестовые объекты.

Решение

Написать вспомогательную функцию с аргументами по умолчанию и использовать ее вместо `copy` или конструктора с аргументами по умолчанию.

Обсуждение

Создание тестовых объектов иногда может оказаться утомительным занятием. Kotlin позволяет указывать значения по умолчанию для аргументов в основном конструкторе класса, но иногда очевидные значения отсутствуют. Например, рассмотрим класс `Book`, показанный в примере 9.5, определение которого повторяется ниже для справки:

```
data class Book(
    val isbn: String,
    val title: String,
    val author: String,
    val published: LocalDate
)
```

Вместо того чтобы изменять класс и добавлять значения по умолчанию для всех аргументов, достаточно добавить фабричную функцию с аргументами по умолчанию, как показано в примере 9.10.

Пример 9.10. Фабричная функция для создания экземпляров Book

```
fun createBook(
    isbn: String = “149197317X”,
    title: String = “Modern Java Recipes”,
    author: String = “Ken Kousen”,
    published: LocalDate = LocalDate.parse(“2017-08-26”)
) = Book(isbn, title, author, published)
```


В примере 9.11 показано, как пользоваться этой функцией.

Пример 9.11. Создание экземпляра книги с помощью фабричной функции

```
val modern_java_recipes = createBook()                    ❶
val making_java_groovy = createBook(isbn = “1935182943”, ❷
    title = "Making Java Groovy",
    published = LocalDate.parse("2013-09-30"))
```

❶ Все атрибуты получат значения по умолчанию, определяемые фабричной функцией

❷ Другая книга того же автора

Аргументы по умолчанию используются только для создания тестовых данных, поэтому нет необходимости добавлять их в предметный класс.

В принципе, то же самое можно сделать с помощью функции `copy`, которая автоматически создается для классов данных, но при обширном использовании `copy` код будет трудно читать, особенно во вложенных структурах. Фабричная функция в примере 9.12 показывает простоту функционального подхода.

Пример 9.12. Класс книг, написанных несколькими авторами, и его использование

```
data class MultiAuthorBook(
    val isbn: String,
    val title: String,
    val authors: List<String>,
    val published: LocalDate
)

fun createMultiAuthorBook(
    isbn: String = "9781617293290",
    title: String = "Kotlin in Action",
    authors: List<String> = listOf("Dimitry Jeremov",
                                   "Svetlana Isakova"),
    published: LocalDate = LocalDate.parse("2017-08-26")
) = MultiAuthorBook(isbn, title, authors, published)

val kotlin_in_action = createMultiAuthorBook()
```

Если все фабричные функции поместить в служебный класс верхнего уровня, их можно повторно использовать в тестах.

9.4. Повторение тестов JUnit 5 с разными данными

Задача

Выполнить тест JUnit 5 с другим набором данных.

Решение

Использовать параметризованные и динамические тесты JUnit 5.

Обсуждение

Пусть требуется протестировать функцию с разными наборами данных. В JUnit 5 для этой цели можно использовать параметризованные тесты, которые позволяют указать источник этих данных, включая значения, разделенные запятыми (Comma-Separated Values, CSV), и фабричные методы. Несмотря на то что JUnit является библиотекой Java, тесты можно писать и использовать для тестирования кода на Kotlin (как в большей части этой книги).

Рассмотрим функцию, которая вычисляет числа Фибоначчи, реализованную с использованием алгоритма хвостовой рекурсии (пример 9.13).

Хвостовая рекурсия обсуждается в рецепте 4.3.

Пример 9.13. Рекурсивная функция для вычисления n-го числа Фибоначчи

```
@JvmOverloads
tailrec fun fibonacci(n: Int, a: Int = 0, b: Int = 1): Int =
    when (n) {
        0 -> a
        1 -> b
        else -> fibonacci(n - 1, b, a + b)
    }
```

Число Фибоначчи определяется как сумма двух предыдущих чисел Фибоначчи, причем `fibonacci(0) == 0` и `fibonacci(1) == 1`[10]. Числа образуют последовательность: 1, 1, 2, 3, 5, 8, 11 и т. д.

Давайте проверим, так ли это. В примере 9.14 показан явный тест, который просто вызывает несколько раз.

Пример 9.14. Явный вызов функции fibonacci

```
@Test
fun `Fibonacci numbers (explicit)`() {
    assertAll(
        { assertThat(fibonacci(4), `is`(3)) },
        { assertThat(fibonacci(9), `is`(34)) },
        { assertThat(fibonacci(2000), `is`(1392522469)) }
    )
}
```

В JUnit 5 определена функция `assertAll`, гарантирующая выполнение всех тестов, даже если некоторые из них потерпят неудачу. Этот тест можно также оформить как параметризованный, использующий источник данных в формате CSV, как показано в примере 9.15.

Пример 9.15. Использование данных в формате CSV для параметризованного теста

```
@ParameterizedTest
@CsvSource(«1, 1», «2, 1», «3, 2»,
    «4, 3», «5, 5», «6, 8», «7, 13»,
    «8, 21», «9, 34», «10, 55»)
fun `first 10 Fibonacci numbers (csv)`(n: Int, fib: Int) =
    assertThat(fibonacci(n), `is`(fib))
```

Аннотация `@CsvSource` принимает аргумент со списком строк, которые являются входными данными для функции. Каждая строка содержит все необходимые аргументы, разделенные запятыми. В этом примере проверяются первые 10 чисел Фибоначчи, и результат выглядит следующим образом:

[10] Есть такая шутка: конференция Фибоначчи в этом году будет не хуже, чем в двух предыдущих вместе взятых!

```
 [1] 1, 1      first 10 Fibonacci numbers (csv)(int, int)[1]  0s     passed
 [2] 2, 1      first 10 Fibonacci numbers (csv)(int, int)[2]  0s     passed
 [3] 3, 2      first 10 Fibonacci numbers (csv)(int, int)[3]  0s     passed
 [4] 4, 3      first 10 Fibonacci numbers (csv)(int, int)[4]  0s     passed
 [5] 5, 5      first 10 Fibonacci numbers (csv)(int, int)[5]  0s     passed
 [6] 6, 8      first 10 Fibonacci numbers (csv)(int, int)[6]  0s     passed
 [7] 7, 13     first 10 Fibonacci numbers (csv)(int, int)[7]  0s     passed
 [8] 8, 21     first 10 Fibonacci numbers (csv)(int, int)[8]  0s     passed
 [9] 9, 34     first 10 Fibonacci numbers (csv)(int, int)[9]  0s     passed
[10] 10, 55    first 10 Fibonacci numbers (csv)(int, int)[10] 0s     passed
```

Также для генерирования тестовых данных в JUnit 5 можно использовать фабричные методы. В Java фабричный метод в тестовом классе должен быть объявлен статическим (`static`), если тест не снабжен аннотацией `@TestInstance(Lifecycle.PER_CLASS)`, а если он определен во внешнем классе, то всегда должен объявляться статическим. Еще он не может принимать никаких аргументов. Наконец, возвращаемое значение должно иметь тип, поддерживающий итерации, например быть потоком, коллекцией, итератором, итерируемым объектом или массивом.

Если выбрать жизненный цикл `Lifecycle.PER_CLASS`, как в предыдущих рецептах в этой главе, то можно просто добавить функцию для создания данных и ссылаться на нее с помощью `@MethodSource`, как показано в примере 9.16.

Пример 9.16. Доступ к функции экземпляра как к источнику параметров

```
private fun fibnumbers() = listOf(
    Arguments.of(1, 1), Arguments.of(2, 1),
    Arguments.of(3, 2), Arguments.of(4, 3),
    Arguments.of(5, 5), Arguments.of(6, 8),
    Arguments.of(7, 13), Arguments.of(8, 21),
    Arguments.of(9, 34), Arguments.of(10, 55))

@ParameterizedTest(name = “fibonacci({0}) == {1}”)
@MethodSource(“fibnumbers”)
fun `first 10 Fibonacci numbers (instance method)`(n: Int, fib: Int) =
    assertThat(fibonacci(n), `is`(fib))
```

В JUnit имеется класс `Arguments` с фабричным методом `of`, объединяющим оба входных аргумента. Он возвращает список `List<Arguments>`, каждый элемент которого содержит два входных аргумента для тестового метода.

Если выбрать жизненный цикл по умолчанию `Lifecycle.PER_METHOD`, то функцию следует поместить в объект-компаньон, как показано в примере 9.17.

Пример 9.17. Использование объекта-компаньона для хранения функции, возвращающей параметры

```
companion object {
    // это необходимо, если для параметризованного теста
    // выбран жизненный цикл Lifecycle.PER_METHOD
    @JvmStatic                                          ❶
    fun fibs() = listOf(
        Arguments.of(1, 1), Arguments.of(2, 1),
        Arguments.of(3, 2), Arguments.of(4, 3),
        Arguments.of(5, 5), Arguments.of(6, 8),
        Arguments.of(7, 13), Arguments.of(8, 21),
        Arguments.of(9, 34), Arguments.of(10, 55))
}
```

```
@ParameterizedTest(name = “fibonacci({0}) == {1}”)
@MethodSource(“fibs”)
fun `first 10 Fibonacci numbers (companion method)`(n: Int, fib: Int) =
    assertThat(fibonacci(n), `is`(fib))
```

❶ Эта аннотация необходима, чтобы библиотека JUnit (Java) видела функцию как статическую

Единственная странность – необходимость использования аннотации `@JvmStatic`, чтобы библиотека JUnit на Java видела метод-источник как статический метод.

Наконец, обратите внимание, что аннотация `@ParameterizedTest` принимает строковый аргумент, который позволяет вам форматировать вывод результатов тестирования. Результат для любого набора будет выглядеть, как показано ниже:

```
fibonacci(1) == 1      first 10 Fibonacci numbers (method)(int, int)[1]
fibonacci(2) == 1      first 10 Fibonacci numbers (method)(int, int)[2]
fibonacci(3) == 2      first 10 Fibonacci numbers (method)(int, int)[3]
fibonacci(4) == 3      first 10 Fibonacci numbers (method)(int, int)[4]
fibonacci(5) == 5      first 10 Fibonacci numbers (method)(int, int)[5]
fibonacci(6) == 8      first 10 Fibonacci numbers (method)(int, int)[6]
fibonacci(7) == 13     first 10 Fibonacci numbers (method)(int, int)[7]
fibonacci(8) == 21     first 10 Fibonacci numbers (method)(int, int)[8]
fibonacci(9) == 34     first 10 Fibonacci numbers (method)(int, int)[9]
fibonacci(10) == 55    first 10 Fibonacci numbers (method)(int, int)[10]
```

Смотри также

Рецепт 9.5, где показано, как использовать классы данных для конструирования еще более сложных наборов исходных данных.

9.5. Использование классов данных для параметризации тестов

Задача

Получить легко читаемый вывод параметризованного теста.

Решение

Создать класс данных, объединяющий входные и ожидаемые значения, и использовать функцию в качестве метода-источника для генерирования тестовых данных.

Обсуждение

Как мы уже знаем, JUnit 5 поддерживает *параметризованные тесты*, которые получают данные для тестирования из метода или файла, и каждый набор данных проверяется с помощью одного и того же теста. Параметризованные тесты подробно рассматривались в рецепте 9.4. Однако все тесты в примерах сводились к сравнению обрабатываемого значения с ожидаемым.

Рассмотрим функцию `fibonacci` из примера 9.13, определение которой приводится ниже для простоты:

```
@JvmOverloads
tailrec fun fibonacci(n: Int, a: Int = 0, b: Int = 1): Int =
    when (n) {
        0 -> a
        1 -> b
        else -> fibonacci(n - 1, b, a + b)
    }
```

Дополнительные параметры a и b в этой функции предназначены для реализации хвостовой рекурсии и имеют соответствующие значения по умолчанию. То есть эта функция обычно вызывается с одним целым числом и возвращает целое число.

Теперь определим класс данных для хранения входных и ожидаемых выходных данных, как показано в примере 9.18.

Пример 9.18. Класс данных для хранения входных и ожидаемых выходных данных

```
data class FibonacciTestData(val number: Int, val expected: Int)
```

Поскольку классы данных в Kotlin уже имеют метод toString, можно создать тестовый метод для параметризованных тестов, который создает экземпляр класса данных для каждой пары входных и выходных данных, как показано в примере 9.19.

Пример 9.19. Параметризованный тест, использующий класс данных

```
@ParameterizedTest
@MethodSource(«fibonacciTestData»)
fun `check fibonacci using data class`(data: FibonacciTestData) {
    assertThat(fibonacci(data.number), `is`(data.expected))
}

private fun fibonacciTestData() = Stream.of(
    FibonacciTestData(number = 1, expected = 1),
    FibonacciTestData(number = 2, expected = 1),
    FibonacciTestData(number = 3, expected = 2),
    FibonacciTestData(number = 4, expected = 3),
    FibonacciTestData(number = 5, expected = 5),
    FibonacciTestData(number = 6, expected = 8),
    FibonacciTestData(number = 7, expected = 13)
)
```

Для тестов JUnit, использующих метод-источник, функция должна быть статической (для Java), если выбран жизненный цикл, отличный от TestInstance.Lifecycle.PER_CLASS. В противном случае приватную функцию следует поместить в объект-компаньон и отметить ее аннотацией @JvmStatic (подробности см. в рецепте 9.1).

Этот тест выведет следующие результаты:

```
check fibonacci using data class(FibonacciTestData)
    [1] FibonacciTestData(number=1, expected=1)
    [2] FibonacciTestData(number=2, expected=1)
    [3] FibonacciTestData(number=3, expected=2)
    [4] FibonacciTestData(number=4, expected=3)
    [5] FibonacciTestData(number=5, expected=5)
```

```
[6] FibonacciTestData(number=6, expected=8)
[7] FibonacciTestData(number=7, expected=13)
```

Класс данных `FibonacciTestData` автоматически получает метод `toString`, который позволяет выводить результаты в удобочитаемом виде.

Смотри также

Рецепт 9.4, где описывается использование параметризованных тестов, появившихся в JUnit 5.

Глава 10

Ввод и вывод

Операции ввода и вывода реализуются в Kotlin легко и просто, но их стиль отличается от привычного для разработчика на Java. Ресурсы в Kotlin часто закрываются с помощью функции `use`, которая делает это от имени пользователя. В этой главе представлено несколько рецептов, описывающих данный подход на примере файлов, но их можно распространить и на другие ресурсы.

10.1. Управление ресурсами с помощью use

Задача

Обработать ресурс, такой как файл, и гарантировать его закрытие по окончании операций с ним, несмотря на то что Kotlin не поддерживает конструкцию try-with-resources.

Решение

Использовать функцию-расширение `use` или `useLines`.

Обсуждение

В Java 1.7 появилась конструкция try-with-resources, которая позволяет открывать ресурс, указанный в круглых скобках между ключевым словом `try` и соответствующим ему блоком; JVM автоматически закроет ресурс после выполнения блока `try`. Единственное требование – чтобы ресурс был представлен классом, реализующим интерфейс `Closeable`. Этот интерфейс реализуют такие классы, как `File`, `Stream` и многие другие, как показано в примере 10.1.

Пример 10.1. Использование конструкции try-with-resources в Java

```
package io;

import java.io.*;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Stream;

public class TryWithResourcesDemo {
    public static void main(String[] args) throws IOException {
        String path = “src/main/resources/book_data.csv”;
```

```
        File file = new File(path);
        String line = null;
        try (BufferedReader reader = new BufferedReader(new FileReader(file))) { ❶
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        }

        try (Stream<String> lines = Files.lines(Paths.get(path))) {              ❷
            lines.forEach(System.out::println);
        }
    }
}
```

❶ BufferedReader реализует интерфейс Closeable

❷ Stream реализует интерфейс Closeable

Класс BufferedReader и интерфейс Stream оба реализуют интерфейс Closeable, соответственно, оба имеют метод close, который автоматически вызывается по завершении блока try.

Вот некоторые интересные особенности, которые стоит отметить:

- в Java 10 и выше объявление BufferedReader и Stream можно заменить зарезервированным словом var. Фактически это один из основных приемов автоматического определения типа локальной переменной. Здесь он не используется, чтобы избежать путаницы с ключевым словом var в Kotlin;
- в Java 9 и выше больше не нужно создавать переменную Closeable внутри круглых скобок. Ее можно передать извне. И снова этот прием не используется здесь;
- поскольку сигнатура метода main изменилась и теперь предусматривает возбуждение исключения IOException, мы столкнулись с одним из тех редких случаев использования блока try без catch или finally.

Это все хорошо, но, к сожалению, конструкция try-with-resources не поддерживается в Kotlin. Зато Kotlin добавил функции-расширения use для Closeable и useLines для Reader и File.

Вот как выглядит сигнатура useLines:

```
inline fun <T> File.useLines(
    charset: Charset = Charsets.UTF_8,
    block: (Sequence<String>) -> T
): T
```

Первый и необязательный аргумент – это используемый набор символов, по умолчанию UTF-8. Второй аргумент – лямбда-выражение, отображающее последовательность Sequence строк из файла в обобщенный аргумент T. Функция useLines автоматически закрывает средство чтения после завершения обработки.

Например, в системах Unix, основанных на BSD (включая macOS), имеется файл, содержащий все слова из международного словаря Вебстера, который не защищен авторскими правами. В Mac OS этот файл находится в каталоге */usr/share/dict/words* и содержит 238 000 слов, по одному в строке. Код в примере 10.2 возвращает 10 самых длинных слов в этом словаре.

Пример 10.2. Поиск 10 самых длинных слов в словаре

```
fun get10LongestWordsInDictionary() =
    File("/usr/share/dict/words").useLines { line ->
        line.filter { it.length > 20 }
            .sortedByDescending(String::length)
            .take(10)
            .toList()
    }
```

Функция отфильтровывает все строки короче 20 символов (то есть все слова короче 20 символов, потому что каждая строка содержит одно слово), сортирует их по длине в порядке убывания, выбирает первые 10 и возвращает в виде списка.

Вот как можно вызвать эту функцию:

```
get10LongestWordsInDictionary().forEach { word ->
    println(«$word (${word.length})»)
```

Она выведет следующие строки:

```
formaldehydesulphoxylate (24)
pathologicopsychological (24)
scientificophilosophical (24)
tetraiodophenolphthalein (24)
thyroparathyroidectomize (24)
anthropomorphologically (23)
blepharosphincterectomy (23)
epididymodeferentectomy (23)
formaldehydesulphoxylic (23)
gastroenteroanastomosis (23)
```

В примере 10.3 показана реализация `File.useLines` в стандартной библиотеке.

Пример 10.3. Реализация функции-расширения useLines для File

```
inline fun <T> Reader.useLines(
    block: (Sequence<String>) -> T): T =
    buffered().use { block(it.lineSequence()) }
```

Обратите внимание, что реализация создает буферизованный экземпляр `Reader` (возвращаемый функцией `buffered`) и делегирует выполнение операции его функции `use`.

Сигнатура этой функции `use` показана в примере 10.4.

Пример 10.4. Сигнатура функции-расширения use для Closeable

```
inline fun <T : Closeable?, R> T.use(block: (T) -> R): R
```

Ее реализация осложнена необходимостью обработки исключений, но суть ее сводится к следующему:

```
try {
    return block(this)
} catch (e: Throwable) {
    // сохранить исключение для использования  дальнейшем
    throw e
} finally {
    close() // требует еще одного блока try/catch
}
```

Функция use является примером шаблона проектирования Execute Around Method, когда инфраструктурный код находится в библиотеке, а фактическую работу выполняет передаваемое лямбда-выражение. Такое разделение инфраструктурной и прикладной логики помогает сосредоточиться на текущей задаче.

Функция use, применяемая в этом рецепте, определена в интерфейсе Closeable, доступном в Java 6. Если вам использовать JDK, поддерживающий Java 8, то в этой версии та же функция определена и в AutoCloseable.

Смотри также

Рецепт 10.2, где показано, как использовать блок use непосредственно. Рецепт 13.4, где функция use используется для остановки пула потоков выполнения Java.

10.2. Запись в файл

Задача

Выполнить запись в файл.

Решение

В дополнение к обычным Java-методам ввода/вывода использовать функции-расширения для класса File, возвращающие потоки вывода и объекты, реализующие запись.

Обсуждение

В Java-класс java.io.File было добавлено несколько функций-расширений. Выполнить итерации по строкам в файле можно с помощью функции forEachLine. С помощью readLines можно получить коллекцию всех строк, имеющихся в файле, что может пригодиться при работе с небольшими файлами. Функция useLines, описанная в рецепте 10.1, позволяет указать функцию, которая будет вызываться для каждой строки. Если файл достаточно мал, для чтения его содержимого в строку или в массив байтов можно использовать readText или readBytes соответственно.

Чтобы выполнить запись в файл с заменой всего существующего содержимого, можно использовать функцию writeText, как показано в примере 10.5.

Пример 10.5. Запись в текстовый файл с заменой его содержимого

```
File(«myfile.txt»).writeText(«My data»)
```

Трудно представить что-то еще более простое. Функция writeText принимает необязательный параметр, определяющий набор символов, который имеет значение по умолчанию UTF-8.

Класс File также имеет функцию-расширение appendText, которая добавляет данные в конец файла.

Функции `writeText` и `appendText` делегируют выполнение операций функциям `writeBytes` и `appendBytes`, каждая из которых использует функцию `use`, чтобы гарантировать закрытие файла после записи.

Также можно использовать функции `writer` (или `printWriter`) и `bufferedWriter`, которые, как можно догадаться, возвращают `OutputStreamWriter` и `BufferedWriter`. С помощью любого из этих объектов можно добавить блок `use`, выполняющий фактическую запись, как показано в примере 10.6.

Пример 10.6. Запись с помощью функции `use`

```
File(fileName).printWriter().use { writer ->
    writer.println(data) }
```

Экземпляр `bufferedWriter` с блоком `use` используется аналогично.

Смотри также

Рецепт 10.1, где подробно обсуждаются функции `use` и `useLines`.


ЛАНЬ®


ЛАНЬ®

Глава 11

Разное

Эта глава включает рецепты, которые нельзя отнести ни к одной из предыдущих тем. Здесь вы узнаете, как сделать функцию `when` исчерпывающей, как измерить время выполнения функции, как использовать функцию `TODO` из стандартной библиотеки и многое другое.

11.1. Обработка версии Kotlin

Задача

Определить программно используемую версию Kotlin.

Решение

Использовать свойство `CURRENT` объекта-компаньона класса `KotlinVersion`.

Обсуждение

Начиная с версии 1.1 пакет *kotlin* включает класс `KotlinVersion`, содержащий старший и младший номера версии, а также номер исправления. Его метод `toString` возвращает комбинацию в форме *major.minor.patch* для данного экземпляра этого класса. Текущий экземпляр класса хранится в общедоступном поле `CURRENT` объекта-компаньона.

Учитывая вышесказанное, получить текущую версию Kotlin тривиально просто: достаточно прочитать поле `KotlinVersion.CURRENT`, как показано в примере 11.1.

Пример 11.1. Вывод текущей версии Kotlin

```
fun main(args: Array<String>) {
    println("The current Kotlin version is ${KotlinVersion.CURRENT}")
}
```

Результатом является трехкомпонентный номер версии компилятора Kotlin, например `1.3.41`. Все три компонента являются целыми числами от `0` до `MAX_COMPONENT_VALUE`, которое имеет значение `255`.

Свойство CURRENT объявлено как общедоступное и отмечено аннотацией @JvmField, поэтому оно также доступно из Java.

Класс KotlinVersion реализует интерфейс Comparable. То есть его экземпляры можно сравнивать с помощью таких операторов, как < или >. Класс также реализует методы equals и hashCode. Наконец, конструкторы в KotlinVersion позволяют указать старший и младший номера версии или старший и младший номера, а также номер исправления.

В результате можно выполнить любое из действий, показанных в примере 11.2.

Пример 11.2. Сравнение версий Kotlin

```
@Test
fun `comparison of KotlinVersion instances work`() {
    val v12 = KotlinVersion(major = 1, minor = 2)
    val v1341 = KotlinVersion(1, 3, 41)
    assertAll(
        { assertTrue(v12 < KotlinVersion.CURRENT) },
        { assertTrue(v1341 <= KotlinVersion.CURRENT) },
        { assertEquals(KotlinVersion(1, 3, 41),
            KotlinVersion(major = 1, minor = 3, patch = 41)) }
    )
}
```

Имеется также функция isAtLeast, с помощью которой можно убедиться, что версия Kotlin не ниже некоторой конкретной версии, как показано в примере 11.3.

Пример 11.3. Сравнение версии Kotlin с конкретным номером

```
@Test
fun `current version is at least 1_3`() {
    assertTrue(KotlinVersion.CURRENT.isAtLeast(major = 1, minor = 3))
    assertTrue(KotlinVersion.CURRENT.isAtLeast(major = 1, minor = 3, patch = 40))
}
```

Как видите, не составляет никакого труда проверить версию Kotlin и напрямую работать с ней.

11.2. Многократное выполнение лямбда-выражения

Задача

Выполнить заданное лямбда-выражение несколько раз.

Решение

Использовать встроенную функцию repeat.

Обсуждение

В стандартной библиотеке имеется функция `repeat`. Это встраиваемая (`inline`) функция, которая принимает два аргумента: `Int` – количество итераций и функцию `(Int) -> Unit` для выполнения.

Текущая реализация представлена в примере 11.4.

Пример 11.4. Определение функции repeat

```
@kotlin.internal.InlineOnly
public inline fun repeat(times: Int, action: (Int) -> Unit) {
    contract { callsInPlace(action) }

    for (index in 0 until times) {
        action(index)
    }
}
```

Функция выполняет заданное лямбда-выражение указанное количество раз, передавая ему параметр с номером текущей итерации, отсчитываемым с нуля.

В примере 11.5 показан простой случай использования этой функции.

Пример 11.5. Использование repeat

```
fun main(args: Array<String>) {
    repeat(5) {
        println(“Counting: $it”)
    }
}
```

Этот код выведет:

```
Counting: 0
Counting: 1
Counting: 2
Counting: 3
Counting: 4
```

Использование `repeat` вместо цикла является иллюстрацией применения внутреннего итератора, когда фактический процесс повторения выполняется библиотекой.

11.3. Исчерпывающая инструкция when

Задача

Заставить компилятор потребовать определения исчерпывающей инструкции `when`.

Решение

Добавить простое свойство-расширение с именем `exhaustive` к обобщенному типу, которое возвращает значение, и добавить его к блоку `when`.

Обсуждение

Так же как оператор `if`, инструкция `when` возвращает значение. Она действует подобно оператору `switch` в Java, но, в отличие от него, не требует явно выходить из каждого варианта и объявлять внешнюю переменную, чтобы вернуть значение.

Например, предположим, что требуется вывести остаток от деления числа на 3, как показано в примере 11.6.

Пример 11.6. Вывод остатка от деления числа на 3

```
fun printMod3(n: Int) {
    when (n % 3) {
        0 -> println("$n % 3 == 0")
        1 -> println("$n % 3 == 1")
        2 -> println("$n % 3 == 2")
    }
}
```

Если выражение `when` не возвращает значения, Kotlin не требует, чтобы оно было исчерпывающим, и это как раз тот случай, когда такое поведение удобно. Мы знаем, что остаток может быть равен только 0, 1 или 2, поэтому здесь на самом деле реализована исчерпывающая проверка, но компилятор не знает этого. В этом легко убедиться, преобразовав эту простую функцию в выражение, как показано в примере 11.7.

Пример 11.7. Использование when для возврата значения

```
fun printMod3SingleStatement(n: Int) = when (n % 3) {
    0 -> println("$n % 3 == 0")
    1 -> println("$n % 3 == 1")
    2 -> println("$n % 3 == 2")
    else -> println("Houston, we have a problem...") ❶
}
```

❶ Не компилируется без предложения `else`

Компилятор требует добавить предложение `else` в это выражение, даже притом что функция `println` ничего не возвращает. Наличие знака равенства означает присваивание, а следовательно, условное выражение должно быть исчерпывающим.

Поскольку Kotlin требует наличия блока `else` в любом условном выражении, возвращающем значение, этим можно воспользоваться, чтобы потребовать сделать все блоки `when` исчерпывающими. Для этого достаточно создать свойство-расширение с именем `exhaustive`, как показано в примере 11.8.

Пример 11.8. Добавление свойства exhaustive в любой объект

```
val <T> T.exhaustive: T
    get() = this
```

Этот код добавит свойство `exhaustive` в обобщенный тип `T` с собственным методом чтения, который возвращает текущий объект.

Теперь это свойство можно добавить куда угодно, включая блок `when`, чтобы принудительно заставить его возвращать значение. В примере 11.9 показано, как это делается.

Пример 11.9. Вывод остатка от деления числа на 3 (исчерпывающая версия)

```
fun printMod3Exhaustive(n: Int) {
    when (n % 3) {
        0 -> println("$n % 3 == 0")
        1 -> println("$n % 3 == 1")
        2 -> println("$n % 3 == 2")
        else -> println("Houston, we have a problem...")
    }.exhaustive ❶
}
```

❶ Свойство заставляет компилятор потребовать добавить предложение `else`

Свойство `exhaustive` в конце блока `when` возвращает текущий объект, поэтому компилятор Kotlin требует, чтобы он был исчерпывающим.

Цель этого примера состояла в том, чтобы потребовать от инструкции `when` быть исчерпывающей, но он также является хорошей иллюстрацией, насколько полезным может быть добавление простого свойства-расширения к универсальному типу.

11.4. Использование функции replace с регулярными выражениями

Задача

Заменить все вхождения подстроки заданным значением.

Решение

Использовать функцию `replace` класса `String`, которая имеет перегруженные версии, принимающие строку или регулярное выражение.

Обсуждение

Класс `String` реализует интерфейс `CharSequence`, то есть на самом деле имеет две версии функции `replace`, как показано в примере 11.10.

Пример 11.10. Две перегруженные версии функции replace

```
fun String.replace(
    oldValue: String,
    newValue: String,
    ignoreCase: Boolean = false
): String

fun CharSequence.replace(
    regex: Regex,
    replacement: String
): String
```

Обе версии заменяют все вхождения подстроки `oldValue` или совпадения с регулярным выражением `regex` указанным значением `newValue` либо `replacement`. Функция `replace` в классе `String` принимает необязательный аргумент, признак чувствительности к регистру, который по умолчанию *не* игнорирует регистр.

Эти две версии функций могут вызывать путаницу, потому что пользователь может предположить, что первая (принимающая аргумент типа `String`) будет обрабатывать строку, как если бы она была регулярным выражением, но на самом деле это не так. Тест в примере 11.11 показывает различия между двумя функциями.

Пример 11.11. Использование двух перегруженных версий replace

```
@Test
fun `demonstrate replace with a string vs regex`() {
    assertAll(
        { assertEquals("one*two*", "one.two.".replace(".", "*")) },
        { assertEquals("********", "one.two.".replace(".".toRegex(), "*")) }
    )
}
```

В первом примере точки (сами символы) заменяются звездочкой, а во втором точки обрабатываются как элементы регулярного выражения, то есть как соответствующие любому отдельному символу. В результате первый пример заменит звездочками только две точки, а второй – все символы.

Фактически здесь есть две потенциальные ловушки для разработчиков на Java:

- функция `replace` заменяет все вхождения, а не только первое. В Java эквивалентный метод называется `replaceAll`;
- перегруженная версия, принимающая строку в первом аргументе, не интерпретирует эту строку как регулярное выражение. Это тоже не похоже на поведение одноименного метода в Java. Чтобы строка интерпретировалась как регулярное выражение, ее сначала нужно преобразовать с помощью функции `toRegex`.

Рассмотрим более интересный пример и проверим, является ли строка палиндромом. Палиндромы – это строки, которые одинаково читаются слева направо и справа налево, при этом допускается игнорировать регистр и пунктуацию. Обратите внимание, что такую функцию можно реализовать в стиле Java (т. е. «говорить на Kotlin с акцентом на Java»), как показано в примере 11.12.

Пример 11.12. Проверка палиндромов в стиле Java

```
fun isPal(string: String): Boolean {
    val testString = string.toLowerCase().replace("""[\W+]""".toRegex(), "")
    return testString == testString.reversed()
}
```

В таком подходе нет ничего плохого, и он прекрасно работает. Эта функция сначала преобразует строку в нижний регистр, а затем использует версию `replace`, принимающую регулярное выражение, чтобы заменить все символы, «не являющиеся символами слов», пустыми строками. Метасимвол `\w` в регулярном выражении представляет любой символ слова, то есть букву нижнего регистра `a-z`, букву верхнего регистра `A-Z`, цифру `0-9` и символ подчеркивания. Версия `\w` с заглавной буквой – метасимвол `\W` – является противоположностью `\w`.

Более идиоматическая версия этой же функции показана в примере 11.13.

Пример 11.13. Проверка палиндромов в стиле Kotlin

```
fun String.isPalindrome() =
    this.toLowerCase().replace("""[\W+]""".toRegex(), "")
        .let { it == it.reversed() }
```

Вот основные ее отличия:

- функция `isPalindrome` добавляется в `String` как функция-расширение, поэтому отпадает необходимость в передаче аргумента. В теле функции текущая строка доступна по ссылке `this`;
- функция `let` позволяет записать всю проверку как одно выражение со сгенерированной проверяемой строкой. Локальная переменная `testString` больше не нужна;
- поскольку теперь тело функции представляет собой единое выражение, фигурные скобки были заменены знаком равенства, как это часто бывает в определениях функций на Kotlin.

Оба подхода работают одинаково хорошо, но опытные разработчики на Kotlin скорее предпочтут второй. Разумеется, лучше всего знать оба способа. В обоих примерах реализация использует версию `replace`, которая принимает регулярное выражение в первом аргументе.

Смотри также

Рецепты 7.3 и 7.4, где обсуждается функция `let`.

11.5. Преобразование чисел в двоичное представление и обратно

Задача

Преобразовать целое число в строку с двоичным представлением (или с представлением в любой другой системе счисления) или строку – в целое число.

Решение

Использовать перегруженные версии функций `toString` и `toInt`, принимающие аргумент `radix` с основанием системы счисления.

Обсуждение

Класс `StringsKt` содержит встраиваемую (`inline`) функцию-расширение для `Int` с именем `toString`, которая принимает основание системы счисления. Он также содержит функцию-расширение для `String`, выполняющую обратное преобразование. С их помощью можно преобразовать `Int` в строку с двоичным представлением числа (т. е. строку, состоящую из единиц и нулей) и обратно, как показано в примере 11.14.

Пример 11.14. Преобразование Int в строку с двоичным представлением и обратно

```
@Test
internal fun toBinaryStringAndBack() {
    val str = 42.toString(radix = 2)
    assertThat(str, `is`("101010"))

    val num = "101010".toInt(radix = 2)
    assertThat(num, `is`(42))
}
```

Строка, созданная вызовом `toString(Int)`, обрезает ведущие нули. Если это нежелательно, то можно дополнительно обработать строку с помощью функции `padStart`.

Допустим, что нам нужно закодировать данные одним двоичным свойством, например представить все возможные варианты перестановки из четырех игральных карт, которые могут иметь две масти – красную и черную. Это можно организовать как простой счет от 0 до 15 в двоичном формате (где 0 представляет красный цвет, а 1 – черный или наоборот), но при этом мы не можем потерять ведущие нули. Решить эту задачу можно, как показано в примере 11.15.

Пример 11.15. Дополнение нулями слева строк с двоичным представлением чисел

```
@Test
internal fun paddedBinaryString() {
    val strings = (0..15).map {
        it.toString(2).padStart(4, '0')
    }

    assertThat(strings, contains(
        "0000", "0001", "0010", "0011",
        "0100", "0101", "0110", "0111",
        "1000", "1001", "1010", "1011",
        "1100", "1101", "1110", "1111"))

    val nums = strings.map { it.toInt(2) }
    assertThat(nums, contains(
        0, 1, 2, 3,
        4, 5, 6, 7,
        8, 9, 10, 11,
        12, 13, 14, 15))
}
```

Поскольку функции `toString` и `toInt` поддерживают все целочисленные основания систем счисления, можно не ограничиваться двоичным представлением, хотя это, пожалуй, наиболее распространенный вариант использования. Это означает, что вариант классической шутки (упомянутой в примере 2.27) можно представить так:

```
val joke = """
    There are ${3.toString(3)} kinds of developers:
        - Those who know binary,
        - Those who don't, and
        - Those who didn't realize this is actually a ternary joke"""

println(joke)
```

Этот код выведет следующее:

```
There are 10 kinds of developers:
    - Those who know binary,
    - Those who don't, and
    - Those who didn't realize this is actually a ternary joke.
```

11.6. Создание выполняемого класса

Задача

Имеется класс с единственной функцией, и требуется максимально упростить ее вызов.

Решение

Переопределить функцию-оператор `invoke` в классе так, чтобы она вызывала требуемую функцию.

Обсуждение

Kotlin позволяет переопределять многие операторы. Для этого достаточно всего лишь переопределить соответствующую функцию-оператор.

В документации к Kotlin это называется *перегрузкой операторов*, но суть от этого не меняется.

Чтобы реализовать оператор, нужно определить функцию-член или функцию-расширение с предопределенным именем и аргументами. Любая функция, переопределяющая оператор, должна объявляться с модификатором `operator`.

Особое положение занимает функция `invoke`. Функция-оператор `invoke` позволяет вызывать экземпляры класса как функции.

В качестве примера рассмотрим бесплатную веб-службу RESTful (https://oreil.ly/Bs7vn), предоставляемую Open Notify, которая возвращает данные JSON с количеством космонавтов в космосе в любой указанный момент времени. В примере 11.16 показан образец данных, возвращаемых этой службой.

Пример 11.16. Образец данных JSON, возвращаемых службой Open Notify

```
{
    «people»: [
        { «name»: «Oleg Kononenko», «craft»: «ISS» },
        { «name»: «David Saint-Jacques», «craft»: «ISS» },
        { «name»: «Anne McClain», «craft»: «ISS» }
    ],
    "number": 3,
    "message": "success"
}
```

Судя по ответу, в момент обращения к службе на борту международной космической станции находились три космонавта.

Вложенные объекты JSON подразумевают, что для анализа этой структуры требуются два класса Kotlin, как показано в примере 11.17.

Пример 11.17. Классы данных, моделирующие возвращаемые данные JSON

```
data class AstroResult(
    val message: String,
    val number: Number,
    val people: List<Assignment>
)

data class Assignment(
    val craft: String,
    val name: String
)
```

Класс `Assignment` представляет комбинацию из имени космонавта и названия корабля[11]. Класс `AstroReult` используется как общий ответ, который включает (будем надеяться) сообщение об успехе («`success`»), количество космонавтов и их должности.

Для случаев, когда требуется выполнить лишь простой HTTP-запрос GET, Kotlin добавил функцию-расширение `readText` в класс `java.net.URL`. То есть, чтобы обратиться к службе, достаточно выполнить вызов

```
var response = URL("http://...").readText()
```

и обработать полученный ответ в формате JSON. Так как URL – это константа и для анализа ответа можно использовать библиотеку, такую как Google Gson, для доступа к службе разумно создать класс, показанный в примере 11.18.

Пример 11.18. Обращение к службе RESTful и анализ результата

```
import com.google.gson.Gson
import java.net.URL

class AstroRequest {

    companion object {
        private const val ASTRO_URL =
            "http://api.open-notify.org/astros.json"
    }

    // fun execute(): AstroResult {      ❶
    operator fun invoke(): AstroResult { ❷
        val responseString = URL(ASTRO_URL).readText()
        return Gson().fromJson(responseString,
            AstroResult::class.java)
    }
}
```

❶ Произвольное имя для включаемой функции

❷ Функция-оператор делает класс выполняемым

[11] Здесь «ISS» расшифровывается как «International Space Station» – международная космическая станция. – *Прим. перев.*

В этом классе URL-адрес службы объявлен в объекте-компаньоне как константа, а единственная функция используется и для отправки запроса службе, и для преобразования ответа в экземпляр `AstroResult` с помощью библиотеки Gson.

Функцию можно было бы назвать как угодно. Если, например, назвать ее `execute`, то вызвать ее можно было бы следующим образом:

```
val request = AstroRequest()
val result = request.execute()
println(result.message)
```

В этом подходе нет ничего плохого, но обратите внимание, что в классе имеется только одна функция. Конечно, в Kotlin есть возможность определять функции верхнего уровня, но кажется нецелесообразным объявлять константу верхнего уровня с URL службы. Другими словами, гораздо естественнее создать класс `AstroRequest`, как было показано выше.

Поскольку класс преследует единственную цель, можно изменить имя функции и добавить к ней ключевое слово `operator`, чтобы сделать выполняемым сам класс, как показано в примере 11.19.

Пример 11.19. Использование выполняемого класса

```
internal class AstroRequestTest {
    val request = AstroRequest() ❶

    @Test
    internal fun `get people in space`() {
        val result = request()   ❷
        assertThat(result.message, `is`("success"))
        assertThat(result.number.toInt(),
            `is`(greaterThanOrEqualTo(0)))
        assertThat(result.people.size,
            `is`(result.number.toInt()))
    }
}
```

❶ Создание экземпляра класса

❷ Вызов класса как функции (фактически будет вызвана функция-оператор `invoke`)

Поскольку `AstroResult` и `Assignment` являются классами данных, их содержимое, показанное ниже, всегда можно вывести с помощью функции `println`:

```
AstroResult(message=success, number=3,
    people=[Assignment(craft=ISS, name=Oleg Kononenko),
    Assignment(craft=ISS, name=David Saint-Jacques),
    Assignment(craft=ISS, name=Anne McClain)])
```

Тесты проверяют отдельные свойства.

Добавление в класс функции-оператора `invoke` позволяет выполнять его экземпляры непосредственно, добавляя круглые скобки к ссылкам на них. При желании можно также определить перегруженные версии функции `invoke` с любыми необходимыми аргументами.

Смотри также

Рецепт 3.5, где более подробно обсуждается возможность перегрузки операторов.

11.7. Измерение прошедшего времени

Задача

Узнать, как долго выполняется блок кода.

Решение

Использовать функции `measureTimeMillis` и `measureNanoTime` из стандартной библиотеки.

Обсуждение

Пакет *kotlin.system* включает функции `measureTimeMillis` и `measureNanoTime`. Их с успехом можно использовать для определения интервала времени, в течение которого выполнялся блок кода, как показано в примере 11.20.

Пример 11.20. Измерение времени выполнения блока кода

```
fun doubleIt(x: Int): Int {
    Thread.sleep(100L)

    println("doubling $x with on thread ${Thread.currentThread().name}")
    return x * 2
}

fun main() {
    println("${Runtime.getRuntime().availableProcessors()} processors")

    var time = measureTimeMillis {
        IntStream.rangeClosed(1, 6)
            .map { doubleIt(it) }
            .sum()
    }
    println("Sequential stream took ${time}ms")

    time = measureTimeMillis {
        IntStream.rangeClosed(1, 6)
            .parallel()
            .map { doubleIt(it) }
            .sum()
    }

    println("Parallel stream took ${time}ms")
}
```

Этот фрагмент выведет примерно следующее:

```
This machine has 8 processors
doubling 1 with on thread main
doubling 2 with on thread main
doubling 3 with on thread main
```

```
doubling 4 with on thread main
doubling 5 with on thread main
doubling 6 with on thread main
Sequential stream took 616ms
doubling 3 with on thread ForkJoinPool.commonPool-worker-11
doubling 4 with on thread main
doubling 5 with on thread ForkJoinPool.commonPool-worker-7
doubling 6 with on thread ForkJoinPool.commonPool-worker-3
doubling 2 with on thread ForkJoinPool.commonPool-worker-5
doubling 1 with on thread ForkJoinPool.commonPool-worker-9
Parallel stream took 110ms
```

В данном конкретном случае JVM сообщает о восьми процессорах, поэтому функция `parallel` потока данных разделила работу между ними, и каждый процессор получил один элемент для удвоения. В результате параллельное выполнение операций заняло всего около 100 миллисекунд, тогда как последовательное выполнение заняло около 600 миллисекунд.

В примере 11.21 показана реализация функции `measureTimeMillis` в стандартной библиотеке.

Пример 11.21. Реализация функции measureTimeMillis

```
public inline fun measureTimeMillis(block: () -> Unit): Long {
    val start = System.currentTimeMillis()
    block()
    return System.currentTimeMillis() - start
}
```

Это – функция высшего порядка, потому что принимает аргумент с лямбда-выражением, и поэтому, для большей эффективности, она объявлена встраиваемой (`inline`). Реализация просто вызывает Java-метод `System.currentTimeMillis` до и после выполнения аргумента `block`. Реализация `measureNanoTime` действует точно так же, но вызывает `System.nanoTime`.

Эти две функции упрощают профилирование производительности кода. Более точные оценки времени можно получить с помощью проекта Java Microbenchmark Harness (JMH) в OpenJDK (https://oreil.ly/N6BBv).

11.8. Запуск потоков выполнения

Задача

Выполнить блок кода в параллельном потоке выполнения.

Решение

Использовать функцию `thread` из пакета *kotlin.concurrent*.

Обсуждение

В Kotlin имеется простая функция-расширение `thread`, которую можно использовать для создания и запуска потоков выполнения. Вот как выглядит сигнатура функции `thread`:

```
fun thread(
    start: Boolean = true,
    isDaemon: Boolean = false,
    contextClassLoader: ClassLoader? = null,
    name: String? = null,
    priority: Int = -1,
    block: () -> Unit
): Thread
```

Поскольку параметр `start` имеет значение по умолчанию `true`, это упрощает создание и запуск нескольких потоков, как показано в примере 11.22.

Пример 11.22. Запуск нескольких потоков выполнения через случайные интервалы времени

```
(0..5).forEach { n ->
    val sleepTime = Random.nextLong(range = 0..1000L)
    thread {
        Thread.sleep(sleepTime)
        println("${Thread.currentThread().name} for $n after ${sleepTime}ms")
    }
}
```

Этот код запускает шесть потоков выполнения, каждый из которых приостанавливается на случайное количество миллисекунд от 0 до 1000, а затем выводит свое имя. Результат выглядит примерно так:

```
Thread-2 for 2 after 184ms
Thread-5 for 5 after 207ms
Thread-4 for 4 after 847ms
Thread-0 for 0 after 917ms
Thread-3 for 3 after 967ms
Thread-1 for 1 after 980ms
```

Обратите внимание, что нет необходимости вызывать `start` для запуска каждого потока выполнения, потому что параметр `start` в функции `thread` по умолчанию получает значение `true`.

Параметр `isDaemon` позволяет создавать потоки-демоны. Если все потоки выполнения в приложении, кроме главного, являются потоками-демонами, то приложение можно завершить без лишних сложностей. Другими словами, если предыдущий пример переписать, как показано в примере 11.23, он вообще ничего не выведет, потому что функция `main` завершится раньше, чем успеют выполниться потоки.

Пример 11.23. Запуск потоков-демонов

```
(0..5).forEach { n ->
    val sleepTime = Random.nextLong(range = 0..1000L)
    thread(isDaemon = true) { ❶
        Thread.sleep(sleepTime)
        println("${Thread.currentThread().name} for $n after ${sleepTime}ms")
    }
}
```

❶ Потоки запускаются как потоки-демоны, поэтому главный поток выполнения не ждет их завершения и завершает программу раньше

Блок кода, передаваемый в функцию thread, – это лямбда-выражение без аргументов и возвращающее Unit, что соответствует интерфейсу Runnable, или, говоря проще, сигнатуре метода run в Thread. В примере 11.24 показана реализация функции thread.

Пример 11.24. Реализация функции thread в стандартной библиотеке

```
public fun thread(
    start: Boolean = true,
    isDaemon: Boolean = false,
    contextClassLoader: ClassLoader? = null,
    name: String? = null,
    priority: Int = -1,
    block: () -> Unit
): Thread {
    val thread = object : Thread() {
        public override fun run() {
            block()
        }
    }
    if (isDaemon)
        thread.isDaemon = true
    if (priority > 0)
        thread.priority = priority
    if (name != null)
        thread.name = name
    if (contextClassLoader != null)
        thread.contextClassLoader = contextClassLoader
    if (start)
        thread.start()
    return thread
}
```

Реализация создает объект типа Thread и переопределяет его метод run для вызова заданного блока block. Затем она устанавливает различные свойства и вызывает start.

Поскольку функция возвращает созданный поток, есть возможность запустить все потоки последовательно, вызвав метод join, как показано в примере 11.25.

Пример 11.25. Присоединение потоков выполнения друг к другу

```
(0..5).forEach { n ->
    val sleepTime = Random.nextLong(range = 0..1000L)
    thread {
        Thread.sleep(sleepTime)
        println("${Thread.currentThread().name} for $n after ${sleepTime}ms")
    }.join() ❶
}
```

❶ Присоединяет каждый следующий поток выполнения к предыдущему

Вот как примерно будет выглядеть вывод этого кода:

```
Thread-0 for 0 after 687ms
Thread-1 for 1 after 661ms
Thread-2 for 2 after 430ms
Thread-3 for 3 after 412ms
```

```
Thread-4 for 4 after 918ms
Thread-5 for 5 after 755ms
```

Этот прием в первую очередь избавляет от необходимости запускать одни потоки внутри других, но также демонстрирует возможность вызова методов возвращаемых потоков.

Смотри также

Главу 13, где подробно рассматриваются вопросы параллельного и конкурентного выполнения.

11.9. Принуждение к завершению реализации с помощью TODO

Задача

Гарантировать завершение реализации определенной функции или теста.

Решение

Использовать функцию `TODO` (с необязательным аргументом `reason`), которая генерирует исключение.

Обсуждение

Разработчики часто оставляют для себя примечания, описывающие необходимость завершить реализацию функции, которую они не могут завершить в данный момент. В большинстве языков программирования для этого добавляется оператор «TODO» в комментарий, например:

```
fun myCleverFunction() {
    // TODO: придумать хорошую реализацию
}
```


Стандартная библиотека Kotlin включает функцию `TODO`, реализация которой показана в примере 11.26.

Пример 11.26. Реализация функции TODO

```
public inline fun TODO(reason: String): Nothing =
    throw NotImplementedError("An operation is not implemented: $reason")
```

Для большей эффективности функция объявлена встраиваемой и при ее вызове генерирует исключение `NotImplementedError`. Ее легко использовать, как показано в примере 11.27.

Пример 11.27. Типичное использование функции TODO

```
fun main() {
    TODO(reason = "none, really")
}

fun completeThis() {
    TODO()
}
```

При попытке выполнить этот код он выведет:

```
Exception in thread «main» kotlin.NotImplementedError:
    An operation is not implemented: none, really
        at misc.TodosKt.main(todos.kt:4)
        at misc.TodosKt.main(todos.kt)
```

В необязательном аргументе `reason` можно объяснить намерения разработчика.

Функцию `TODO` также можно использовать в тестах, чтобы генерировать исключение, пока тест не будет окончательно реализован, как показано в примере 11.28.

Пример 11.28. Использование функции TODO в тесте

```
fun `todo test`() {
    val exception = assertThrows<NotImplementedError> {
        TODO("seriously, finish this")
    }

    assertEquals("An operation is not implemented: seriously, finish this",
        exception.message)
}
```

Функция `TODO` – одно из множества дополнительных удобств в библиотеке, которые легко не заметить, пока кто-нибудь не укажет на них. Надеюсь, вы сможете найти способы воспользоваться этими удобствами.

11.10. Случайное поведение класса Random

Задача

Сгенерировать случайное число.

Решение

Использовать одну из функций класса `Random`.

Обсуждение

Класс `kotlin.random.Random` прост в использовании, но имеет довольно замысловатую реализацию. Во-первых, самое простое – получить случайное целое число можно с помощью одной из перегруженных версий `nextInt`. В документации с описанием `kotlin.random.Random` говорится, что это абстрактный класс, но он включает методы, перечисленные в примере 11.29.

Пример 11.29. Объявления функций в абстрактном классе Random

```
open fun nextInt(): Int
open fun nextInt(until: Int): Int
open fun nextInt(from: Int, until: Int): Int
```

Все три функции имеют реализации по умолчанию. Использовать их достаточно просто, как показано в примере 11.30.

Пример 11.30. Перегруженные версии функции nextInt

```
@Test
fun `nextInt with no args gives any Int`() {
    val value = Random.nextInt()
    assertTrue(value in Int.MIN_VALUE..Int.MAX_VALUE)
}

@Test
fun `nextInt with a range gives value between 0 and limit`() {
    val value = Random.nextInt(10)
    assertTrue(value in 0..10)
}

@Test
fun `nextInt with min and max gives value between them`() {
    val value = Random.nextInt(5, 10)
    assertTrue(value in 5..10)
}

@Test
fun `nextInt with range returns value in range`() {
    val value = Random.nextInt(7..12)
    assertTrue(value in 7..12)
}
```

Последний пример не перечислен в списке функций в классе Random, потому что это – функция-расширение со следующей сигнатурой:

```
fun Random.nextInt(range: IntRange): Int
```

Если заглянуть в файл с исходным кодом предыдущих тестов, то можно увидеть, что он импортирует kotlin.random.Random и kotlin.random.nextInt, где nextInt – это функция-расширение.

Класс Random имеет довольно интересную реализацию. Она включает методы, перечисленные в примере 11.29, за которыми следует объект-компаньон типа Random. В примере 11.31 показан фрагмент реализации.

Пример 11.31. Объект-компаньон класса Random

```
companion object Default : Random() {
    private val defaultRandom: Random = defaultPlatformRandom()

    override fun nextInt(): Int = defaultRandom.nextInt()
    override fun nextInt(until: Int): Int = defaultRandom.nextInt(until)
    override fun nextInt(from: Int, until: Int): Int =
        defaultRandom.nextInt(from, until)
    // ...
}
```

Объект-компаньон получает реализацию по умолчанию и переопределяет все объявленные методы так, что они вызывают реализацию по умолчанию. defaultPlatformRandom – это внутренняя функция.

Аналогично реализованы другие типы, такие как Boolean, Byte, Float, Long и Double, а также беззнаковые типы UBytes, UInt и ULong.

Но самое интересное, что существует также функция с именем `Random`, которая принимает начальное значение типа `Int` или `Long` и возвращает воспроизводимый генератор случайных чисел, инициализированный аргументом (см. пример 11.32).

Пример 11.32. Использование инициализированного генератора случайных чисел

```
@Test
fun `Random function produces a seeded generator`() {
    val r1 = Random(12345)
    val nums1 = (1..10).map { r1.nextInt() }

    val r2 = Random(12345)
    val nums2 = (1..10).map { r2.nextInt() }

    assertEquals(nums1, nums2)
}
```


Если создать два генератора, инициализированных одним и тем же начальным числом, то оба будут генерировать одну и ту же последовательность случайных чисел.

Генераторы случайных чисел в Kotlin просты в использовании, но изучение реализации методов (с использованием абстрактного класса, методы которого переопределяются внутри объекта-компаньона) может дать представление о том, как разрабатывать собственные классы.

11.11. Использование специальных символов в именах функций

Задача

Написать функции с легко читаемыми именами.

Решение

Использовать символ подчеркивания или заключить имя функции в обратные кавычки, но только в тестах.

Обсуждение

Kotlin позволяет заключать имена функций в обратные кавычки, как показано в примере 11.33.

Пример 11.33. Заключение имен функций в обратные кавычки

```
fun `only use backticks on test functions`() {
    println(“This works but is not a good idea”)
}

fun main() {
    `only use backticks on test functions`()
}
```

Заключение в обратные кавычки позволяет использовать в именах пробелы. Но если применить этот прием в отношении имени обычной функции, IntelliJ IDEA отметит это, сообщив: «Имя функции может содержать только буквы и цифры». Такая функция будет компилироваться и выполняться, как показано выше, но подобную практику нельзя назвать удачной.

Другой вариант – использовать символы подчеркивания, как в примере 11.34.

Пример 11.34. Использование символа подчеркивания в именах функций

```
fun underscores_are_also_okay_only_on_tests() {
    println("Again, please don't do this outside of tests")
}

fun main() {
    underscores_are_also_okay_only_on_tests()
}
```

Такие функции тоже будут компилироваться и выполняться, но компилятор выдаст предупреждение: «Имя функции не должно содержать символов подчеркивания».

Однако в тестах можно свободно использовать любой из перечисленных приемов, и такой код будет считаться идиоматическим, как отмечается в руководстве «Coding Conventions» (https://oreil.ly/xIguq). Так что имена, показанные в примере 11.35, с успехом могут использоваться в тестах.

Пример 11.35. Имена тестовых функций могут оформляться любым удобным способом

```
class FunctionNamesTest {

    @Test
    fun `backticks make for readable test names`() {
        // ...
    }

    @Test
    fun underscores_are_fine_here_too() {
        // ...
    }
}
```

Более того, такие легко читаемые имена будут отображаться в отчете о тестировании, поэтому все, что делает результаты тестирования более понятными, – это хорошо.

11.12. Передача исключений в Java

Задача

Исключение, генерируемое функцией на Kotlin, должно рассматриваться кодом на Java как контролируемое, но об этом нужно сообщить явно.

Решение

Добавить в сигнатуру функции аннотацию `@Throws`.

Обсуждение

В Kotlin все исключения считаются неконтролируемыми, то есть компилятор не требует их обработки. Чтобы перехватить исключение в функции на Kotlin, можно добавить блок try/catch/finally, но поступать так необязательно.

В Kotlin нет ключевого слова `throws`, которое Java использует для объявления методов, которые могут сгенерировать исключение.

Ситуация меняется, когда возникает необходимость вызвать такую функцию из Java. Если функция на Kotlin может сгенерировать исключение, которое код на Java должен считать контролируемым, то вы должны сообщить об этом, чтобы дать возможность перехватить его.

Например, предположим, что есть функция на Kotlin, которая генерирует исключение `IOException`, являющееся контролируемым в Java, как показано в примере 11.36.

Пример 11.36. Функция на Kotlin, генерирующая исключение IOException

```
fun houstonWeHaveAProblem() {
    throw IOException("File or resource not found")
}
```

Чтобы скомпилировать эту функцию на Kotlin, не нужен блок try/catch или предложение `throws`. Как показано в примере, функция генерирует исключение `IOException`.

Эту функцию можно вызвать из Java, и в результате возникнет исключение, как показано в примере 11.37.

Пример 11.37. Вызов функции на Kotlin из Java

```
public static void doNothing() {
    houstonWeHaveAProblem(); ❶
}
```

❶ Вызовет аварийное завершение с исключением `IOException`

(Исходный код для этого примера включает статический импорт вызываемой функции.)

Проблема возникнет, если вы решите обработать исключение `IOException`, заключив вызов в блок try/catch или добавив предложение `throws` к определению функции в Java, как показано в примере 11.38.

Пример 11.38. Попытка перехватить ожидаемое исключение

```
public static void useTryCatchBlock() {
    try {                          ❶
        houstonWeHaveAProblem();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void useThrowsClause() throws IOException { ❷
    houstonWeHaveAProblem();
}
```

❶ Не компилируется

❷ Компилируется, но компилятор предупреждает о «ненужности» предложения `throws`

Ни один из этих приемов не дает желаемого результата. Если добавить явный блок try/catch, код не скомпилируется, так как Java считает, что исключение `IOException`, указанное в блоке `catch`, никогда не возникает в соответствующем блоке `try`. Во втором случае код скомпилируется, но среда разработки (и компилятор) предупредит, что в коде присутствует ненужное предложение.

Чтобы оба подхода заработали, нужно добавить аннотацию `@Throws` в код на Kotlin, как показано в примере 11.39.

Пример 11.39. Добавление аннотации @Throws

```
@Throws(IOException::class) ❶
fun houstonWeHaveAProblem() {
    throw IOException("File or resource not found")
}
```


❶ Сообщить Java, что эта функция может сгенерировать исключение `IOException`

Теперь компилятор Java будет знать, что может возникнуть исключение `IOException`, и функция `doNothing` перестанет компилироваться, потому что вы должны организовать обработку контролируемого исключения `IOException`.

Аннотация `@Throws` существует только для интеграции Java и Kotlin. Она решает конкретную проблему, но работает именно так, как можно заключить из ее названия.

Глава 12

Фреймворк Spring

Фреймворк Spring – один из самых популярных фреймворков с открытым исходным кодом в мире Java. Цель Spring – предоставить инфраструктуру для вашего проекта. Он позволяет сосредоточиться на разработке *bean*-компонентов с прикладной логикой, беря на себя решение всех остальных задач, таких как безопасность, транзакции, управление ресурсами и многих других, с учетом предоставленных вами метаданных.

Фреймворк Spring всегда дружил с «альтернативными» языками JVM. Он поддерживает Groovy начиная с версии 2.5. В последних версиях разработчики Spring также добавили возможности, уникальные для Kotlin.

В этой главе представлено несколько избранных приемов, которые можно использовать в коде на Kotlin в приложениях для Spring. Поддержка Kotlin в Spring продолжает эволюционировать, но эта глава должна дать вам представление о том, как Kotlin вписывается в экосистему Spring.

12.1. Открытие классов для расширения фреймворком Spring

Задача

Фреймворк Spring генерирует прокси-объекты, расширяя прикладные классы, но в Kotlin классы закрыты по умолчанию (объявлены как `final` в Java).

Решение

Добавить плагин поддержки Spring в файл сборки. Он автоматически откроет для расширения все необходимые классы.

Обсуждение

Фреймворк Spring предоставляет приложению множество услуг. Для этого он использует шаблон проектирования «Заместитель» (Proxy)[12]. Организация и работа этого шаблона показаны на диаграмме UML классов и диаграмме последовательности выполнения операций на рис. 12.1.

[12] https://ru.wikipedia.org/wiki/Заместитель_(шаблон_проектирования). – *Прим. перев.*

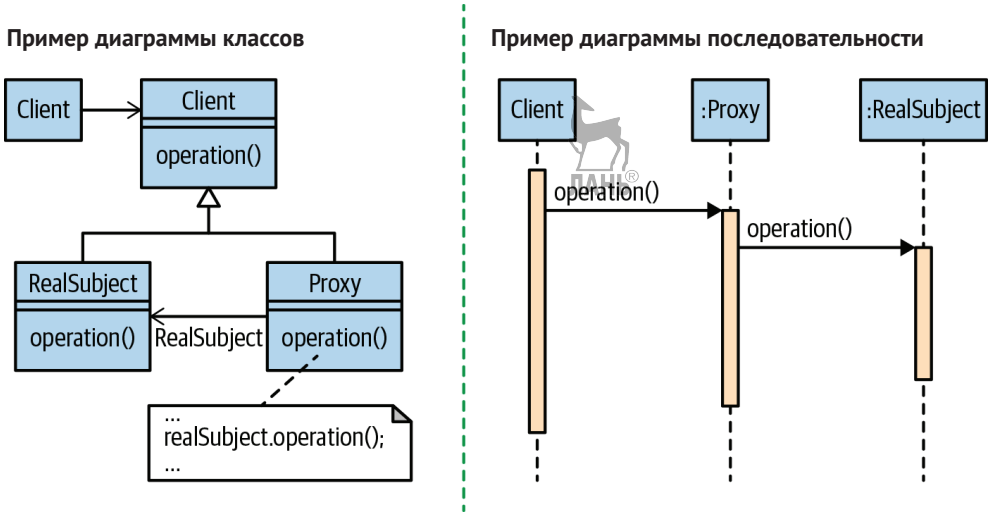


Рис. 12.1. Диаграмма UML шаблона проектирования «Заместитель»

Идея состоит в том, что заместитель (Proxy) и реальный субъект (RealSubject) реализуют один и тот же интерфейс или расширяют (наследуют) один и тот же класс. Входящий запрос перехватывается заместителем, который выполняет все необходимые предварительные операции, а затем пересылает запрос реальному субъекту. Заместитель также может перехватить ответ и при необходимости выполнить дополнительную работу. Например, заместитель механизма управления транзакциями в Spring перехватывает вызов метода, запускает транзакцию, вызывает метод и затем вызывает `commit` или `rollback`, в зависимости от результата выполнения метода реального субъекта.

Фреймворк Spring генерирует объекты-заместители во время запуска. Если реальный субъект – это класс, то фреймворк должен расширить его, и в этом случае в Kotlin возникает проблема. Дело в том, что код на Kotlin по умолчанию связывается статически, то есть вы не сможете переопределить метод или расширить класс, если он не был отмечен как открытый для расширения с помощью ключевого слова `open`.

Для решения подобных проблем в Kotlin есть плагин *all-open*. Этот плагин открывает классы, отмеченные указанной аннотацией, не требуя явно добавлять ключевое слово `open` к классу и к функциям в нем.

Это довольно удобный плагин, но разработчики языка пошли еще дальше и создали плагин *kotlin-spring* специально для поддержки Spring. Чтобы задействовать этот плагин, его нужно добавить в файл сборки Gradle или Maven. В примере 12.1 показан фрагмент файла сборки Gradle (на языке Kotlin DSL), подключающий этот плагин. Файл называется *build.gradle.kts* и был создан с помощью инструмента Spring Initializr (https://start.spring.io/).

Пример 12.1. Добавление плагина kotlin-spring в файл сборки

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile

plugins {
    id(“org.springframework.boot”) version “2.1.8.RELEASE”
    id(“io.spring.dependency-management”) version “1.0.8.RELEASE”
    kotlin(“jvm”) version “1.2.71”                                    ❶
    kotlin("plugin.spring") version "1.2.71"                          ❷
}

group = "com.mycompany"
version = "1.0"

java.sourceCompatibility = JavaVersion.VERSION_11

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.springframework.boot:spring-boot-starter")
    implementation("org.jetbrains.kotlin:kotlin-reflect")       ❸
    implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8")   ❸
    testImplementation("org.springframework.boot:spring-boot-starter-test")
}

tasks.withType<KotlinCompile> {
    kotlinOptions {
        freeCompilerArgs = listOf("-Xjsr305=strict")            ❹
        jvmTarget = "1.8"
    }
}
```

❶ Добавление плагина Kotlin JVM в проект

❷ Добавление плагина Kotlin Spring

❸ Необходимо, если исходный код проекта написан на Kotlin

❹ Подключение аннотаций JSR-305 поддержки значения `null`

Плагин *all-open* позволяет определить аннотации для объявления открытых классов Kotlin. Плагин *kotlin-spring* уже настроен для поддержки следующих аннотаций Spring:

- `@Component`;
- `@Async`;
- `@Transactional`;
- `@Cacheable`;
- `@SpringBootTest`.

Аннотация `@Component` используется в нескольких других аннотациях Spring, включая `@Configuration`, `@Controller`, `@RestController`, `@Service` и `@Repository`. Все управляемые bean-компоненты Spring, отмеченные любой из этих аннотаций, автоматически открываются для расширения, чего часто бывает более чем достаточно.

Если потребуется выйти за рамки данного плагина, добавьте плагин *all-open*, но обычно в этом нет необходимости.

Чтобы увидеть файл сборки для Maven, использующий тот же плагин, сгенерируйте проект Maven с помощью Initializr.

Смотри также

Рецепт 12.2, где обсуждается плагин *kotlin-jpa*.

12.2. Хранимые классы данных на Kotlin

Задача

Использовать Java Persistence API (JPA) с классами данных Kotlin.

Решение

Добавить плагин *kotlin-jpa* в файл сборки.

Обсуждение

Обычно при определении класса данных все необходимые свойства добавляются в основной конструктор, как показано в примере 12.2.

Пример 12.2. Класс данных с основным конструктором

```
data class Person(val name: String,
                  val dob: LocalDate)
```

С точки зрения JPA здесь есть две проблемы. Во-первых, JPA требует наличия в классе конструктора по умолчанию, а для этого необходимо указать значения по умолчанию для всех свойств. Во-вторых, объявление класса данных со свойствами `val` повлечет создание неизменяемых объектов, а механизм JPA не предназначен для работы с неизменяемыми объектами.

Сначала решим проблему с конструктором по умолчанию. Для этого в Kotlin имеется два плагина. Первый плагин – *no-arg* – позволяет выбирать, какие классы должны иметь конструктор без аргументов, и определять аннотации для их вызова. Второй плагин – *kotlin-jpa* – автоматически настраивает *сущности* Kotlin (например, классы, отмеченные аннотацией `@Entity`, которые демонстрируются в этом рецепте), добавляя конструкторы по умолчанию.

По аналогии с плагином *kotlin-spring*, описанным в рецепте 12.1, чтобы воспользоваться этими плагинами, необходимо добавить соответствующие инструкции в файл сборки. В предыдущем рецепте, в примере 12.1, было показано, как подключить плагин *kotlin-spring* в файле сборки Gradle (на Kotlin DSL). Опираясь на этот пример, внесите необходимые дополнения в *build.gradle.kts* из примера 12.1, как показано в примере 12.3.

Пример 12.3. Дополнительные зависимости для поддержки сущностей JPA

```
plugins {
    // ... как и прежде ...
    kotlin(«plugin.jpa») version «1.2.71»
}

// ... другие настройки, как и прежде ...

dependencies {
    // ... другие зависимости из предыдущего рецепта ...
    implementation(«org.springframework.boot:spring-boot-starter-data-jpa»)
    implementation(«com.fasterxml.jackson.module:jackson-module-kotlin»)
}
```

Зависимость `jackson-module-kotlin` не требуется для поддержки сущностей, но она помогает сериализовать классы Kotlin в формат JSON и обратно с помощью библиотеки Jackson 2.

Плагин *no-arg* компилятора добавляет *синтетический* конструктор по умолчанию в классы на Kotlin. Этот конструктор невозможно вызвать из Java или Kotlin, но Spring сможет сделать это с помощью механизма рефлексии. Вы можете использовать этот плагин, но тогда вам придется определить аннотации для отметки классов, которым необходим конструктор без аргументов.

Как показано в примере 12.3, плагин *kotlin-jpa* намного проще в использовании. Он основан на плагине *no-arg* и автоматически добавляет конструкторы по умолчанию в любые классы, отмеченные аннотациями:

- `@Entity`;
- `@Embeddable`;
- `@MappedSuperclass`.

Вторая проблема заключается в том, что JPA не умеет работать с неизменяемыми сущностями. Поэтому команда Spring рекомендует использовать в роли сущностей только обычные классы (не классы данных) со свойствами `var`, чтобы значения полей можно было изменять. В примере 12.4 показан вариант использования Spring Boot с Kotlin из учебника Spring.

Пример 12.4. Классы Kotlin, отображающиеся в таблицы в базе данных

```
@Entity
class Article(
    var title: String,
    var headline: String,
    var content: String,
    @ManyToOne var author: User,
    var slug: String = title.toSlug(),
    var addedAt: LocalDateTime = LocalDateTime.now(),
    @Id @GeneratedValue var id: Long? = null)

@Entity
class User(
    var login: String,
    var firstname: String,
```

```
    var lastname: String,
    var description: String? = null,
    @Id @GeneratedValue var id: Long? = null)
```

Классы `Article` и `User` используют `var`-свойства и даже допускают значение `null` в поле первичного ключа. На языке Hibernate (наиболее известный производитель поддержки JPA) первичный ключ со значением `null` (отмеченный здесь аннотацией `@Id`) указывает, что экземпляр находится в переходном состоянии, то есть когда в соответствующей таблице базы данных нет записи, связанной с этим экземпляром. Например, сразу после создания экземпляра класса до его сохранения или после удаления записи из базы данных, пока экземпляр все еще находится в памяти.

Аннотация `@GeneratedValue` сообщает, что значения первичного ключа генерирует сама база данных.

Вам как разработчику на Kotlin широкое использование `var` и отсутствие автоматически сгенерированных функций `toString`, `equals` и `hashCode` может показаться неудобным. Тем не менее такой подход лучше совместим с JPA. Если вы используете другой API, основанный на Spring Data, например Spring Data MongoDB или Spring Data JDBC, то можете использовать классы данных.

Смотри также

Рецепт 12.1, где обсуждается плагин *kotlin-spring*.

12.3. Внедрение зависимостей

Задача

Организовать автоматическое связывание компонентов с использованием механизма внедрения зависимостей и объявлять, какие компоненты необходимы, а какие – нет.

Решение

Использовать механизм внедрения в конструктор, поддерживаемый в Kotlin. Для внедрения в поля применять свойства `lateinit var`. Необязательные компоненты можно объявлять с использованием типов, поддерживающих значение `null`.

Обсуждение

Связывание компонентов в Spring осуществляется с помощью механизма *внедрения зависимостей*. Но пусть вас не пугает это название, кажущееся сложным. На самом деле все довольно просто. Идея заключается в добавлении ссылки из одного типа на экземпляр другого типа, а Spring сам найдет способ, как предоставить экземпляр этого типа.

По возможности Spring использует внедрение в конструктор. В Kotlin для этого можно использовать аннотацию `@Autowired` для отметки аргументов конструктора. Если в классе имеется только один конструктор, то вам даже не придется использовать аннотацию `@Autowired`, потому что все аргументы единственного конструктора будут связываться автоматически.

Если в компоненте Spring имеется только один конструктор, то Spring автоматически внедрит все аргументы.

Допустим, что у вас есть контроллер REST, который должен взаимодействовать с внедряемой службой. Организовать автоматическое внедрение можно с помощью любого из подходов, показанных в примере 12.5.

Пример 12.5. Автоматическое внедрение зависимости в Spring

```
@RestController  ❶
class GreetingController(val service: GreetingService) { /* ... */ }

@RestController  ❷
class GreetingController(@Autowired val service: GreetingService) { /* ... */ }

@RestController  ❸
class GreetingController @Autowired constructor(val service: GreetingService) {
    // ... (обычный отступ с 4 пробелами)
}

@RestController  ❹
class GreetingController {
    @Autowired
    lateinit var service: GreetingService

    // ... остальная часть определения класса ...
}
```

❶ Вариант 1: класс с единственным конструктором

❷ Вариант 2: явное автоматическое связывание

❸ Вариант 3: вызов конструктора автоматического связывания, этот прием используется в основном для классов с несколькими зависимостями

❹ Вариант 4: внедрение в поле (не самый лучший, но вполне работоспособный способ)

В этом примере показаны все способы внедрения зависимостей:

- просто объявить зависимости; в классе с единственным конструктором все зависимости будут подключены автоматически;
- явно использовать аннотацию `@Autowired`, которая действует точно так же, но явная аннотация `@Autowired` продолжает работать, даже если в класс добавить вторичный конструктор;
- добавить аннотацию `@Autowired` перед функцией `constructor`. Обычно этот прием используется как упрощенный вариант, когда требуется внедрить несколько зависимостей;
- наконец, если по каким-то причинам необходимо организовать внедрение в поле, можно определить поле с модификаторами `lateinit var`.

Поскольку свойства `val` должны получать значение при объявлении, их нельзя инициализировать значениями, определяемыми позже. Вот почему ключе-

вое слово lateinit используется с var. Недостаток этого способа состоит в возможности изменить свойство var в любой момент, а это может быть не совсем то, что вам нужно. Это одна из причин, почему внедрение в конструктор предпочтительнее.

Если свойство класса является необязательным, его можно объявить с типом, допускающим значение null. Например, рассмотрим функцию в GreetingController, которая генерирует приветствие из необязательного параметра запроса, как показано в примере 12.6.

Пример 12.6. Функция контроллера с необязательным параметром

```
@GetMapping(«/hello»)
fun greetUser(@RequestParam name: String?) =
    Greeting(service.sayHello(name?: “World”)) else Greeting()
```

Объявив параметр name как допускающий значение null (например, с типом String? вместо String), мы сообщаем компилятору Kotlin, что параметр является необязательным.

Фреймворк Spring также поддерживает использование JUnit 5 для тестирования. Вот две особенности JUnit 5, не поддерживаемые в JUnit 4:

- в тестах JUnit 5 можно определять конструкторы, отличные от конструкторов по умолчанию;
- управляя жизненным циклом тестового класса, можно создавать один экземпляр для класса в целом, а не для каждого его метода.

В примере 12.7 показан фрагмент теста из учебника Kotlin/Spring.

Пример 12.7. Внедрение зависимостей с использованием аргументов конструктора в JUnit 5

```
@DataJpaTest
class RepositoriesTests @Autowired constructor(
    val entityManager: TestEntityManager,
    val userRepository: UserRepository,
    val articleRepository: ArticleRepository) {

    // ... здесь следуют определения тестов ...
}

// Другой тест, использующий Spring-класс RestTemplate
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class IntegrationTests(@Autowired val restTemplate: TestRestTemplate) {
    // ... здесь следуют определения тестов ...
}
```

При использовании варианта с аннотацией @Autowired перед функцией constructor не требуется использовать свойства lateinit var. В первом случае, с классом RepositoriesTests, производится автоматическое внедрение экземпляров пользовательских классов. Во втором случае, с классом IntegrationTests, производится запуск тестового сервера на случайном порту, на котором развертывается веб-приложение, а затем используется класс TestRestTemplate (который уже настроен с соответствующим портом) для выполнения запросов REST с применением таких функций, как getForObject или getForEntity.

Глава 13

Сопрограммы и структурированная конкуренция

Одной из самых популярных особенностей Kotlin является поддержка *сопрограмм*, которые дают возможность писать код, выполняющийся конкурентно, как если бы он был синхронным. Эта поддержка упрощает написание конкурентного кода, использующего сопрограммы, по сравнению с другими методами, такими как обратные вызовы или реактивные потоки.

Обратите внимание: ключевое слово здесь «упрощает», а не «просто». Управление параллельным выполнением всегда было сложной задачей, особенно когда требуется координировать несколько разных действий, обрабатывать отмену и исключения и многое другое.

В этой главе рассматриваются вопросы, связанные с использованием сопрограмм в Kotlin. К их числу относятся: определение области видимости и контекста сопрограммы, выбор подходящих функции запуска и диспетчера сопрограмм, а также управление их поведением.

Сопрограммы – это фрагменты кода, которые можно приостанавливать и возобновлять. Отмечая функцию ключевым словом `suspend`, вы сообщаете системе, что она может временно приостановить выполнение функции и возобновить ее позже в другом потоке выполнения, и при этом вам не нужно писать сложный многопоточный код.

13.1. Выбор функции запуска сопрограмм

Задача

Выбрать подходящую функцию для создания сопрограммы.

Решение

Сделать выбор между несколькими доступными функциями запуска сопрограмм.

Обсуждение

Создать новую сопрограмму можно с помощью одной из доступных функций запуска: `runBlocking`, `launch` и `async`. Первая из них – `runBlocking` – является функцией верхнего уровня, а `launch` и `async` – это функции-расширения для `CoroutineScope`.

Прежде чем перейти к изучению приемов их использования, важно отметить, что существуют также версии `launch` и `async` для класса `GlobalScope`, пользоваться которыми не рекомендуется, если в этом нет абсолютной необходимости. Дело в том, что эти функции запускают сопрограммы, не привязанные к какой-то конкретной задаче, и охватывают весь жизненный цикл приложения, если их не отменить преждевременно. Поэтому старайтесь не использовать их без веской на то причины.

Этот раздел можно было бы озаглавить как «Выбор функции запуска сопрограмм, за исключением GlobalScope.launch».

Функция runBlocking

Функция `runBlocking` – одна из рекомендуемых функций запуска сопрограмм – может пригодиться для демонстрации из командной строки или в тестах. Как можно догадаться по ее имени, она блокирует текущий поток и ждет завершения всех запущенных сопрограмм.

Вот как выглядит сигнатура функции `runBlocking`:

```
fun <T> runBlocking(block: suspend CoroutineScope.() -> T): T
```

Функция `runBlocking` сама по себе не является функцией, доступной для приостановки, поэтому ее можно вызывать из обычных функций. Она принимает аргумент с функцией, доступной для приостановки, добавляет ее как функцию-расширение в `CoroutineScope`, запускает и возвращает любое значение, возвращаемое этой функцией.

Как показано в примере 13.1, использовать `runBlocking` довольно просто.

Пример 13.1. Использование функции runBlocking

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.runBlocking

fun main() {
    println(“Before creating coroutine”)
    runBlocking {
        print(“Hello, “)
        delay(200L)
        println(“World!”)
    }
    println(“After coroutine is finished”)
}
```

Этот код выведет следующее:

```
Before creating coroutine
Hello, World!
After coroutine finished
```

Но обратите внимание, что между выводом «Hello» и «World!» имеет место 200-миллисекундная задержка.

Функция launch

Чтобы запустить сопрограмму, выполняющую отдельный процесс и ничего не возвращающую, можно использовать функцию `launch`. Функция `launch` – это функция-расширение для `CoroutineScope`, поэтому ее можно использовать, только если доступен экземпляр `CoroutineScope`. Она возвращает экземпляр `Job`, который можно использовать для отмены сопрограммы.

Вот как выглядит сигнатура функции `launch`:

```
fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job
```

Экземпляр `CoroutineContext` представляет состояние, совместно используемое несколькими сопрограммами. `CoroutineStart` – это класс-перечисление, включающий значения `DEFAULT`, `LAZY`, `ATOMIC` и `UNDISPATCHED`.

Лямбда-выражение `block` должно быть функцией, допускающей приостановку, не принимающей аргументов и ничего не возвращающей. В примере 13.2 показано, как использовать функцию `launch`.

Пример 13.2. Использование функции launch

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() {
    println(“Before runBlocking”)
    runBlocking {                ❶
        println("Before launch")
        launch {                 ❷
            print("Hello, ")
            delay(200L)
            println("World!")
        }
        println("After launch")
    }
    println("After runBlocking")
}
```

❶ Создание области видимости сопрограмм

❷ Запуск сопрограммы

Этот код выведет следующие строки:

```
Before runBlocking
Before launch
After launch
Hello, World!
After runBlocking
```

И снова между выводом «Hello» и «World!» имеет место 200-миллисекундная задержка.

Порядок отмены возвращаемого задания Job демонстрируется в примере 13.5.

Функция async

В обычной ситуации, когда нужно вернуть значение, используйте функцию async. Это тоже функция-расширение для CoroutineScope, и вот как выглядит ее сигнатура:

```
fun <T> CoroutineScope.async(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> T
): Deferred<T>
```

И снова параметры CoroutineContext и CoroutineStart имеют разумные значения по умолчанию.

На этот раз функция, поддерживающая возможность приостановки, возвращает значение, которое функция async заключает в экземпляр Deferred. Экземпляр Deferred действует подобно объекту Promise в JavaScript или Future в Java. Важно помнить, что Deferred имеет функцию await, которая ожидает завершения сопрограммы и возвращает произведенное ею значение.

В примере 13.3 показано, как использовать async.

Пример 13.3. Создание сопрограммы с помощью async

```
import kotlinx.coroutines.async
import kotlinx.coroutines.coroutineScope
import kotlinx.coroutines.delay
import kotlin.random.Random

suspend fun add(x: Int, y: Int): Int {
    delay(Random.nextLong(1000L))                    ❶
    return x + y
}

suspend fun main() = coroutineScope {                ❷
    val firstSum = async {                           ❸
        println(Thread.currentThread().name)
        add(2, 2)
    }
    val secondSum = async {                          ❸
        println(Thread.currentThread().name)
        add(3, 4)
    }
    println("Awaiting concurrent sums...")
    val total = firstSum.await() + secondSum.await() ❹
    println("Total is $total")
}
```

❶ Случайная задержка до 1000 мс

❷ Еще одна функция запуска сопрограмм, обсуждаемая далее в этом рецепте

❸ Использование async для запуска сопрограммы

❹ Вызов await, чтобы дождаться завершения сопрограммы

Функция add приостанавливает выполнение на случайное время до 1000 мс, а затем возвращает вычисленную сумму. Два вызова async запускают функцию add и возвращают экземпляры Deferred. Вызовы await блокируют выполнение главного потока до завершения сопрограмм.

Вот результаты выполнения этого кода:

```
DefaultDispatcher-worker-2
Awaiting concurrent sums...
DefaultDispatcher-worker-1
Total is 11
```

Обратите внимание, что функция delay – это функция с поддержкой возможности приостановки, которая переводит сопрограмму в режим ожидания, не блокируя потока, в котором она выполняется.

Два вызова функции async используют диспетчер по умолчанию – один из обсуждаемых в рецепте 13.3. Вызов runBlocking будет ждать завершения всех запущенных сопрограмм. Порядок вывода строк зависит от случайно сгенерированных задержек.

Функция coroutineScope

Наконец мы добрались до функции coroutineScope. Она также поддерживает возможность приостановки и ждет завершения всех запущенных сопрограмм перед выходом. Ее преимущество в том, что она не блокирует основной поток (в отличие от runBlocking), но должна вызываться как часть suspend-функции.

Это подводит нас к одному из фундаментальных принципов – сопрограммы должны выполняться в определенной области видимости. Преимущество coroutineScope заключается в отсутствии необходимости вручную определять, завершились ли сопрограммы, – она автоматически ждет завершения всех дочерних сопрограмм.

Вот как выглядит сигнатура функции coroutineScope:

```
suspend fun <R> coroutineScope(
    block: suspend CoroutineScope.() -> R
): R
```

Как видите, функция принимает лямбда-выражение (с приемником CoroutineScope) без аргументов и возвращает обобщенное значение. Функция поддерживает возможность приостановки, поэтому должна вызываться из другой функции с поддержкой приостановки или из другой сопрограммы.

Простой пример использования coroutineScope показан непосредственно на домашней странице Kotlin (http://kotlinlang.org/) и в примере 13.4.

Пример 13.4. Использование функции coroutineScope

```
import kotlinx.coroutines.coroutineScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

suspend fun main() = coroutineScope {  ❶
    for (i in 0 until 10) {
        launch {                       ❷
            delay(1000L - i * 10)      ❸
            print("♥$i “)
        }
    }
}
```

❶ Функция coroutineScope

❷ Запуск 10 сопрограмм

❸ Постепенно уменьшающаяся задержка перед запуском очередной сопрограммы

Этот пример запускает 10 сопрограмм, перед каждой из которых выполняется задержка на 10 миллисекунд меньше, чем перед предыдущей. В результате выводится последовательность сердечек и чисел в порядке убывания:

```
♥9 ♥8 ♥7 ♥6 ♥5 ♥4 ♥3 ♥2 ♥1 ♥0
```

В этом примере показан общий шаблон: сначала вызывается coroutineScope, чтобы создать область видимости для сопрограмм, а внутри получившегося блока вызываются функции launch или async для запуска конкретных сопрограмм. После этого область видимости будет ждать завершения всех сопрограмм, а если какая-либо из сопрограмм завершится аварийно, то отменит все остальные. Это обеспечивает хороший баланс между управлением и обработкой ошибок, не требуя проверять – продолжают ли выполняться сопрограммы, и предотвращает утечки в случае сбоев.

Соглашение о запуске всех сопрограмм внутри coroutineScope, гарантирующее остановку всех в случае сбоя одной, известно как *структурированная конкуренция* (structured concurrency).

Сопрограммы могут вызывать сложности при рассуждении о порядке выполнения кода, потому что в этом механизме очень много движущихся частей и очень много возможных комбинаций. К счастью, на практике используется лишь несколько комбинаций, которые были показаны в этом рецепте.

13.2. Замена async/await на withContext

Задача

Упростить код, запускающий сопрограмму с помощью async и затем просто ожидающий ее завершения с помощью await.

Решение

Заменить комбинацию вызовов `async` и `await` вызовом `withContext`.

Обсуждение

Для класса `CoroutineScope` имеется еще одна функция-расширение – `withContext`. Вот ее сигнатура:

```
suspend fun <T> withContext(
    context: CoroutineContext,
    block: suspend CoroutineScope.() -> T
): T
```

В документации говорится, что `withContext` «вызывает указанный блок с заданным контекстом сопрограммы, приостанавливается до завершения этого блока и возвращает результат». На практике `withContext` используется для замены комбинации вызовов `async` и `await`, как показано в примере 13.5.

Пример 13.5. Замена комбинации вызовов async и await вызовом withContext

```
suspend fun retrieve1(url: String) = coroutineScope {
    async(Dispatchers.IO) {
        println("Retrieving data on ${Thread.currentThread().name}")
        delay(100L)
        "asyncResults"
    }.await()
}

suspend fun retrieve2(url: String) =
    withContext(Dispatchers.IO) {
        println("Retrieving data on ${Thread.currentThread().name}")
        delay(100L)
        "withContextResults"
    }

fun main() = runBlocking<Unit> {
    val result1 = retrieve1("www.mysite.com")
    val result2 = retrieve2("www.mysite.com")
    println("printing result on ${Thread.currentThread().name} $result1")
    println("printing result on ${Thread.currentThread().name} $result2")
}
```

Функция `main` начинается с вызова `runBlocking`, что тоже типично для такой простой демонстрации. Две функции, `retrieve1` и `retrieve2`, делают одно и то же: приостанавливаются на 100 миллисекунд и затем возвращают строку. Запустив этот код, я получил следующие результаты (обратите внимание, что у вас порядок вывода строк может быть другим):

```
Retrieving data on DefaultDispatcher-worker-2
Retrieving data on DefaultDispatcher-worker-2
printing result on main withContextResults
printing result on main asyncResults
```

Обе функции используют диспетчера `Dispatchers.IO` (обсуждается в рецепте 13.3), поэтому они отличаются только тем, что одна использует комбина-

цию async/await, а другая заменяет ее вызовом withContext. Фактически, когда среда разработки IntelliJ IDEA видит, что в коде за вызовом async сразу же следует вызов await, она предложит заменить эту пару вызовом withContext и сделает это за вас, если вы разрешите.

13.3. Диспетчеры

Задача

Использовать выделенный пул потоков выполнения для ввода/вывода или других задач.

Решение

Использовать подходящего диспетчера из класса Dispatchers.

Обсуждение

Сопрограммы выполняются в контексте, определяемом типом CoroutineContext, который включает диспетчера сопрограмм, представленного экземпляром класса CoroutineDispatcher. Диспетчер определяет, какой поток или пул потоков выполнения использовать для выполнения сопрограмм.

При использовании функции запуска сопрограмм, такой как launch или async, можно явно указать тип диспетчера для использования, передав необязательный параметр CoroutineContext.

Вот список встроенных диспетчеров, предлагаемых библиотекой:

- Dispatchers.Default;
- Dispatchers.IO;
- Dispatchers.Unconfined.

Последний обычно не рекомендуется использовать в прикладном коде.

Диспетчер Default использует общий пул фоновых потоков выполнения. Он подходит для сопрограмм, потребляющих большой объем вычислительных ресурсов.

Диспетчер IO использует общий пул потоков выполнения, создаваемых по требованию, и предназначен для операций ввода/вывода, активно использующих блокировки, таких как операции с файловой системой или сетевой ввод/вывод.

Диспетчеры довольно просты в использовании. Достаточно передать подходящего диспетчера функции launch, async или withContext, как показано в примере 13.6.

Пример 13.6. Использование диспетчеров Default и IO

```
fun main() = runBlocking<Unit> {
    launchWithIO()
    launchWithDefault()
}
```

```
suspend fun launchWithIO() {
    withContext(Dispatchers.IO) {          ❶
        delay(100L)
        println("Using Dispatchers.IO")
        println(Thread.currentThread().name)
    }
}

suspend fun launchWithDefault() {
    withContext(Dispatchers.Default) {     ❷
        delay(100L)
        println("Using Dispatchers.Default")
        println(Thread.currentThread().name)
    }
}
```

❶ Диспетчер `IO`

❷ Диспетчер `Default`

Вот результат выполнения этого примера (у вас номера рабочих потоков могут отличаться):

```
Using Dispatchers.IO
DefaultDispatcher-worker-3
Using Dispatchers.Default
DefaultDispatcher-worker-2
```

Запуская сопрограммы, можно указать любого диспетчера.

В некоторых руководствах предлагается использовать функции `newSingleThreadContext` и `newFixedThreadPoolContext` для создания диспетчеров. В настоящее время обе считаются устаревшими и будут заменены в будущем. Вместо них можно использовать функцию `asCoroutineDispatcher` из Java-класса `ExecutorService`, как описано далее в этом рецепте.

Диспетчеры в Android

Кроме уже рассмотренных диспетчеров, Android API предлагает также диспетчера `Dispatchers.Main`. Обычно он используется для реализации инструментов пользовательского интерфейса, задачей которых является обновление пользовательского интерфейса в `Main`, но выполняющих работу, которая может вызвать дополнительные задержки в `Main`.

Чтобы получить диспетчера `Main` в Android, необходимо подключить зависимость `kotlinx-coroutines-android`. Вот как это выглядит в файле сборки Gradle:

```
dependencies {
    implementation “org.jetbrains.kotlinx:kotlinx-coroutines-core:x.x.x”
    implementation «org.jetbrains.kotlinx:kotlinx-coroutines-android:x.x.x»
}
```

Здесь значения `x.x.x` следует заменить номером последней версии.

Библиотека компонентов Android также включает несколько дополнительных диспетчеров жизненного цикла. Подробности ищите в описании библиотеки Android KTX, в частности в описании реализации `lifecycle-viewmodel`. Фак-

тически Android часто рекомендует запускать сопрограммы с использованием `viewModelScope` из этой библиотеки.

Смотри также

Рецепт 13.4, где обсуждается служба исполнения в Java, которая предлагает своих диспетчеров сопрограмм. Рецепт 13.5, где обсуждаются диспетчеры Android.

13.4. Запуск сопрограмм в пуле потоков Java

Задача

Создать свой пул потоков для выполнения сопрограмм.

Решение

Использовать функцию `asCoroutineDispatcher` в Java-классе `ExecutorService`.

Обсуждение

Библиотека Kotlin определяет метод-расширение для `java.util.concurrent.ExecutorService` с именем `asCoroutineDispatcher`. Как сказано в документации, эта функция преобразует экземпляр `ExecutorService` в реализацию `ExecutorCoroutineDispatcher`.

Чтобы использовать его, нужно с помощью класса `Executors` определить пул потоков, а затем преобразовать его для использования в роли диспетчера, как показано в примере 13.7.

Пример 13.7. Использование пула потоков выполнения в роли диспетчера сопрограмм

```
import kotlinx.coroutines.asCoroutineDispatcher
import kotlinx.coroutines.delay
import kotlinx.coroutines.runBlocking
import kotlinx.coroutines.withContext
import java.util.concurrent.Executors

fun main() = runBlocking<Unit> {
    val dispatcher = Executors.newFixedThreadPool(10)  ❶
        .asCoroutineDispatcher()

    withContext(dispatcher) {                          ❷
        delay(100L)
        println(Thread.currentThread().name)
    }

    dispatcher.close()                                 ❸
}
```

❶ Создание пула с 10 потоками выполнения

❷ Использование пула в роли диспетчера сопрограмм

❸ Остановка пула потоков

Этот код выведет pool-1-thread-2, сообщая, что система запустила сопрограмму в потоке 2 пула 1.

Обратите внимание на последнюю строку в этом примере, которая вызывает функцию close диспетчера. Это необходимо, потому что иначе служба исполнения будет продолжать работать, а значит, функция main никогда не завершится.

Предыдущий прием также является интересной иллюстрацией, как в Kotlin решаются подобные проблемы. Обычно, чтобы остановить Java-класс ExecutorService, вызывается метод shutdown или shutdownNow. Поэтому пример можно переписать, сохранив ссылку на ExecutorService и останавливая его вручную, как показано в примере 13.8.

Пример 13.8. Остановка пула потоков выполнения вручную

```
val pool = ExecutorService.newFixedThreadPool(10)
withContext(pool.asCoroutineDispatcher()) {
    // ... тот же код, что и прежде ...
}
pool.shutdown()
```


Проблема этого подхода заключается в том, что пользователь может забыть вызвать метод shutdown. В Java подобные проблемы решаются путем реализации интерфейса AutoCloseable с методом close, что позволяет заключить код в блок try-with-resources. К сожалению, здесь требуется вызвать метод shutdown, а не close.

По этой причине разработчики библиотеки Kotlin внесли изменения в базовый класс ExecutorCoroutineDispatcher, экземпляр которого создается в предыдущем коде. Они реорганизовали его и реализовали интерфейс Closeable, назвав новый абстрактный класс CloseableCoroutineDispatcher, чей метод close выглядит, как показано ниже:

```
import java.util.concurrent.ExecutorService

abstract class ExecutorCoroutineDispatcher: CoroutineDispatcher(), Closeable {
    abstract override fun close()
    abstract val executor: Executor
}

// Далее в подклассах:
override fun close() {
    (executor as? ExecutorService)?.shutdown()
}
```

Это означает, что диспетчеры, созданные с помощью службы исполнения, сейчас имеют функцию close, которая останавливает службу исполнения. Теперь возникает вопрос: как обеспечить вызов функции close, учитывая, что, в отличие от Java, язык Kotlin не поддерживает конструкцию try-with-resources? В Kotlin есть функция use. Ее определение показано в примере 13.9.

Пример 13.9. Функция use

```
inline fun <T : Closeable?, R> T.use(block: (T) -> R): R
```

Как видите, use определяется как функция-расширение для Java-интерфейса Closeable. Это дает прямое решение проблемы остановки службы исполнения в Java, как показано в примере 13.10.

Пример 13.10. Автоматическое завершение диспетчера с помощью use

```
Executors.newFixedThreadPool(10).asCoroutineDispatcher().use {
    withContext(it) {
        delay(100L)
        println(Thread.currentThread().name)
    }
}
```


Этот код закроет диспетчера, как только блок use достигнет конца, который, в свою очередь, остановит пул потоков выполнения.

Смотри также

Рецепт 10.1, где описывается функция use.

13.5. Отмена сопрограмм

Задача

Остановить асинхронный процесс, выполняемый в сопрограмме.

Решение

Использовать ссылку на экземпляр Job, которая возвращается функцией launch, или запускать сопрограммы с помощью таких функций, как withTimeout и withTimeoutOrNull.

Обсуждение

Функция launch возвращает ссылку на экземпляр типа Job, с помощью которого можно прервать выполнение сопрограммы. В примере 13.11 приводится код, взятый из справочного руководства по Kotlin.

Пример 13.11. Отмена задания

```
fun main() = runBlocking {
    val job = launch {
        repeat(100) { i ->
            println("job: I'm waiting $i...")
            delay(100L)
        }
        }
    delay(500L)
    println("main: That's enough waiting")
    job.cancel()
    job.join()
    println("main: Done")
}
```


Функция launch возвращает ссылку на экземпляр Job, которая сохраняется в локальной переменной. Затем с помощью функции repeat запускается 100 сопрограмм.

После выхода из блока launch функция main спустя какое-то время решает отменить задание. Функция join ждет завершения задания, а затем программа завершается. Вот как выглядит вывод этой программы:

```
job: I'm waiting 0...
job: I'm waiting 1...
job: I'm waiting 2...
job: I'm waiting 3...
job: I'm waiting 4...
main: That's enough waiting
main: Done
```

Существует также функция `cancelAndJoin`, которая объединяет в себе `cancel` и `join`.

Если основной причиной отмены задания является превышение некоторого времени, то для запуска сопрограмм можно использовать функцию `withTimeout`. Вот сигнатура этой функции:

```
suspend fun <T> withTimeout(
    timeMillis: Long,
    block: suspend CoroutineScope.() -> T
): T
```

Функция запускает блок кода внутри сопрограммы и генерирует исключение `TimeoutCancellationException`, если время его выполнения превысит установленный предел. В примере 13.12 показано, как использовать эту функцию (и снова код для примера взят из справочного руководства).

Пример 13.12. Использование функции withTimeout

```
fun main() = runBlocking {
    withTimeout(1000L) {
        repeat(50) { i ->
            println(«job: I'm waiting $i...»)
            delay(100L)
        }
    }
}
```

Этот пример выведет:

```
job: I'm waiting 0...
job: I'm waiting 1...
job: I'm waiting 2...
job: I'm waiting 3...
job: I'm waiting 4...
job: I'm waiting 5...
job: I'm waiting 6...
job: I'm waiting 7...
job: I'm waiting 8...
job: I'm waiting 9...
Exception in thread «main» kotlinx.coroutines.TimeoutCancellationException:
    Timed out waiting for 1000 ms
at kotlinx.coroutines.TimeoutKt.TimeoutCancellationException(Timeout.kt:126)
    // ... остальная часть трассировки стека ...
```

Это исключение можно перехватить и обработать. Точно так же можно использовать функцию `withTimeoutOrNull`, которая не генерирует исключение, а просто возвращает `null`, если будет превышен порог времени выполнения.

Отмена заданий в Android

В Android имеется дополнительный диспетчер `Dispatchers.Main`, который работает с потоком пользовательского интерфейса. Типичный способ его использования – реализовать `CoroutineScope` в `MainActivity`, предоставить контекст, когда это необходимо, а затем закрыть его, если потребуется. Типовая реализация данного подхода показана в примере 13.13.

Пример 13.13. Использование диспетчеров в Android

```
class MainActivity : AppCompatActivity(), CoroutineScope {
    override val coroutineContext: CoroutineContext  ❶
        get() = Dispatchers.Main + job

    private lateinit var job: Job                    ❷

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        job = Job()                                  ❸
    }

    override fun onDestroy() {
        job.cancel()                                 ❹
        super.onDestroy()
    }
}
```

❶ Создание контекста с помощью перегруженного оператора `plus`

❷ Это свойство будет инициализировано по готовности значения для него

❸ Собственно инициализация свойства

❹ Отмена задания с разрушением пользовательского интерфейса

Использование переменной `job` с отложенной инициализацией обеспечивает ее доступность на случай отмены. Теперь достаточно просто запускать сопрограммы по мере необходимости, как показано в примере 13.14.

Пример 13.14. Запуск сопрограмм в Android

```
fun displayData() {
    launch {                                         ❶
        val data = async(Dispatchers.IO) {           ❷
            // ... получить данные из сети ...
        }
        updateDisplay(data.await())                  ❸
    }
}
```

❶ Запуск с использованием свойства `coroutineContext`

❷ Выбор `Dispatchers.IO` для выполнения сетевого вызова

❸ Возврат в `Dispatchers.Main` для обновления пользовательского интерфейса

В момент уничтожения пользовательского интерфейса задание автоматически будет отменено.

Последние версии компонентов архитектуры Android предлагают дополнительные области видимости, такие как `viewModelScope`, которые автоматически отменяют задания при очистке `ViewModel`. Это – часть библиотеки Android KTX, поэтому, чтобы воспользоваться этими возможностями, нужно добавить соответствующую зависимость в файл сборки:

```
dependencies {
    // ... как и прежде ...
    implementation «androidx.lifecycle:lifecycle-viewmodel-ktx:x.x.x»
}
```

Эта строка добавит свойство `viewModelScope`, которое можно использовать для запуска сопрограмм с любым диспетчером.

13.6. Отладка сопрограмм

Задача

Получить больше информации о выполнении сопрограммы.

Решение

Запустить программу в JVM с флагом `-Dkotlinx.coroutines.debug`.

Обсуждение

Отладка асинхронных программ всегда была сложной задачей, потому что они могут выполнять сразу несколько операций. К счастью, библиотека сопрограмм включает простую в использовании поддержку отладки.

Чтобы запустить сопрограмму в режиме отладки (под управлением JVM), используйте системное свойство `kotlinx.coroutines.debug`.

Как вариант можно включить отладку с помощью флага `-ea` в командной строке Java.

В режиме отладки каждой запущенной сопрограмме присваивается уникальное имя. В примере 13.5, повторно воспроизведенном ниже для удобства, параллельно с основным потоком запускаются две сопрограммы:

```
suspend fun retrieve1(url: String) = coroutineScope {
    async(Dispatchers.IO) {
        println(“Retrieving data on ${Thread.currentThread().name}”)
        delay(100L)
        “asyncResults”
    }.await()
}
```

```
suspend fun retrieve2(url: String) =
    withContext(Dispatchers.IO) {
        println("Retrieving data on ${Thread.currentThread().name}")
        delay(100L)
        "withContextResults"
    }

fun main() = runBlocking<Unit> {
    val result1 = retrieve1("www.mysite.com")
    val result2 = retrieve2("www.mysite.com")
    println("printing result on ${Thread.currentThread().name} $result1")
    println("printing result on ${Thread.currentThread().name} $result2")
}
```

Если выполнить эту программу с флагом `-Dkotlinx.coroutines.debug`, она выведет:

```
Retrieving data on DefaultDispatcher-worker-1 @coroutine#1
Retrieving data on DefaultDispatcher-worker-1 @coroutine#2
printing result on main @coroutine#1 withContextResults
printing result on main @coroutine#1 asyncResults
```

Каждая сопрограмма получила уникальное имя (`@coroutine#1` и т. д.), которое отображается как часть имени потока.

Иногда это может пригодиться, но чаще бывает желательно определить свои имена для сопрограмм. Для этой цели библиотека Kotlin предлагает класс `CoroutineName`. Конструктор `CoroutineName` создает элемент контекста, который можно использовать в качестве имени потока выполнения, как показано в примере 13.15.

Пример 13.15. Именование сопрограмм

```
suspend fun retrieve1(url: String) = coroutineScope {
    async(Dispatchers.IO + CoroutineName("async")) { ❶
        // ... как и прежде ...
    }.await()
}

suspend fun retrieve2(url: String) =
    withContext(Dispatchers.IO + CoroutineName("withContext")) { ❶
        // ... как и прежде ...
    }
```

❶ Добавляет имя сопрограммы

Вот как теперь выглядит результат выполнения этого кода:

```
Retrieving data on DefaultDispatcher-worker-1 @withContext#1
Retrieving data on DefaultDispatcher-worker-1 @async#2
printing result on main @coroutine#1 withContextResults
printing result on main @coroutine#1 asyncResults
```

Слова «async» и «withContext» теперь отображаются как имена сопрограмм. Это также хороший пример использования перегруженного оператора `plus` при работе с `CoroutineContext`. Другой пример использования оператора `plus` в приложениях для Android показан в рецепте 13.5.

Предметный указатель

Ж

З

И

К

Об авторе

Кен Коузен (Ken Kousen) – обладатель званий Java Champion, Oracle Developer Champion и Grails Rock Star, автор книг «Modern Java Recipes»[13], «Gradle Recipes for Android» и «Making Java Groovy». Также является сертифицированным партнером JetBrains по обучению языку Kotlin.

Выходные данные

На обложке «Kotlin. Сборник рецептов» изображен кинкажу (*Potos flavus*), длиннохвостое хищное млекопитающее из семейства енотовых, обитающее в Центральной и Южной Америке.

У кинкажу золотисто-коричневый мех, маленькие лапы с небольшими перепонками и большие черные глаза, которые помогают этому ночному зверьку лучше видеть ночью. Цепкие задние лапы и хвост помогают кинкажу вести древесный образ жизни. Несмотря на внешнее сходство с приматами, кинкажу относятся к семейству енотовых.

Хотя кинкажу всеяден, большую часть его рациона составляют фрукты и особенно инжир. Иногда он поедает мелких позвоночных или птичьи яйца. Удивительно длинный язык позволяет кинкажу лакомиться насекомыми или нектаром, за что он получил прозвище «медовый медведь». В неволе кинкажу действительно с большим удовольствием ест мед.

Кинкажу живут стаями. Они могут быть агрессивными, когда напуганы или если их разбудить днем, атакуют острыми зубами и когтями и издают громкий визг и лай, эхом разносящиеся по окружающему лесу.

В настоящее время кинкажу присвоен природоохранный статус «вызывающий наименьшее беспокойство», тем не менее, промышленная вырубка лесов представляет потенциальную угрозу популяции кинкажу. Многие животные, изображенные на обложках книг издательства O'Reilly находятся под угрозой исчезновения; они все важны для нашего мира.

Иллюстрацию для обложки нарисовала Карен Монтгомери (Karen Montgomery) на основе черно-белой гравюры из книги Лидеккера (Lydekker) «Natural History». Текст на обложке набран шрифтами Gilroy Semibold и Guardian Sans. Текст книги набран шрифтом Adobe Minion Pro; текст заголовков — шрифтом Adobe Myriad Condensed; а фрагменты программного кода — шрифтом Ubuntu Mono, созданным Далтоном Маагом (Dalton Maag).

[13] *Кен Коузен.* Современный Java. Рецепты программирования. ДМК-Пресс, 2018. ISBN: 978-5-97060-134-1. – *Прим. перев.*

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.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Кен Коузен

Kotlin. Сборник рецептов

Предметный подход

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Луценко С. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 17,88. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com

Отпечатано в ООО «Принт-М»
142300, Московская обл., Чехов, ул. Полиграфистов, 1