



**Big Nerd
Ranch**

О. Котлин
Кронштадт

Kotlin

**ПРОГРАММИРОВАНИЕ
ДЛЯ ПРОФЕССИОНАЛОВ**



**ДЖОШ
СКИН**

**ДЭВИД
ГРИНХОЛ**

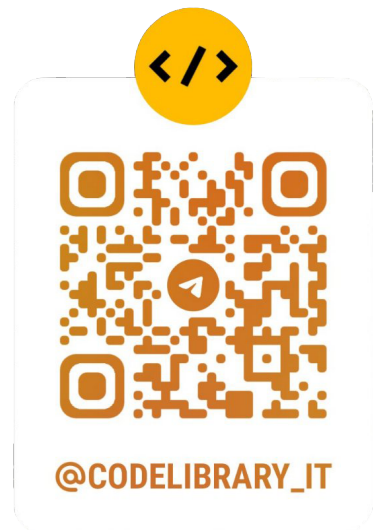
**ЭНДРЮ
БЭЙЛИ**

**ВТОРОЕ
издание**

Kotlin Programming: The Big Nerd Ranch Guide

by Josh Skeen, David Greenhalgh, Andrew Bailey

2ND EDITION



Kotlin

ПРОГРАММИРОВАНИЕ ДЛЯ ПРОФЕССИОНАЛОВ

Джош Скин, Дэвид Гринхол, Эндрю Бэйли

ВТОРОЕ ИЗДАНИЕ



Санкт-Петербург • Москва • Минск

2023

ББК 32.973.2-018.1
УДК 004.43
С42

Скин Д., Гринхол Д., Бэйли Э.

С42 Kotlin. Программирование для профессионалов. 2-е изд. — СПб.: Питер, 2023. — 560 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-2319-3

Kotlin — это язык программирования со статической типизацией, который взяла на вооружение Google в ОС Android. Кроме того, это мультиплатформенный язык, позволяющий создавать приложения для macOS, Windows и iOS.

Джош Скин, Дэвид Гринхол, Эндрю Бэйли на практических примерах познакомят вас с ключевыми концепциями Kotlin и фундаментальными API.

Вы начнете с основных принципов и перейдете к расширенному использованию нетривиальных возможностей Kotlin, чтобы создавать надежные и эффективные приложения, а так же освоите среду разработки IntelliJ IDEA от JetBrains.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

Права на издание получены по соглашению с Pearson Education Inc. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-0136891055 англ.

Authorized translation from the English language edition, entitled Kotlin Programming: The Big Nerd Ranch Guide, 2nd Edition by David Greenhalgh and Josh Skeen, published by Pearson Education, Inc, publishing as Big Nerd Ranch Guides, © 2022

ISBN 978-5-4461-2319-3

© Перевод на русский язык ООО «Прогресс книга», 2022

© Издание на русском языке, оформление ООО «Прогресс книга», 2022

© Серия «Для профессионалов», 2022

Оглавление

Благодарности.....	16
Представляем Kotlin.....	18
Почему Kotlin?	18
Для кого написана эта книга	19
Как пользоваться этой книгой	19
Для любознательных.....	20
Задания.....	20
Шрифтовые обозначения	20
Заглядывая вперед.....	21
От издательства	21

ЧАСТЬ I. ПЕРВЫЕ ШАГИ

Глава 1. Ваше первое приложение на Kotlin	24
Установка IntelliJ IDEA.....	24
Ваш первый проект на Kotlin.....	25
Ваш первый файл на Kotlin	31
Запуск вашего файла на языке Kotlin.....	33
Компиляция и выполнение кода Kotlin/JVM.....	34
Kotlin REPL.....	35
Для любознательных: зачем использовать IntelliJ.....	37
Для любознательных: программирование для JVM	37
Задание: арифметические вычисления в REPL.....	38
Глава 2. Переменные, константы и типы	40
Типы	40
Объявление переменной	40
Встроенные типы языка Kotlin.....	44
Переменные, доступные только для чтения.....	44
Автоматическое определение типов	48

Константы времени компиляции	49
Изучаем байт-код Kotlin	52
Для любознательных: простые типы Java в Kotlin	55
Задание: hasSteed	56
Задание: «Рог единорога»	57
Задание: волшебное зеркало.....	57

ЧАСТЬ II. БАЗОВЫЙ СИНТАКСИС

Глава 3. Условные конструкции.....	60
Операторы if/else.....	60
Добавление условий.....	64
Вложенные команды if/else.....	66
Более элегантные условные выражения	67
Логические операторы	68
Условные выражения	71
Убираем скобки в выражениях if/else	74
Интервалы.....	75
Условное выражение when	77
Выражения when с объявлением переменных	79
Выражения when без аргументов	80
Задание: эксперименты с интервалами.....	81
Глава 4. Функции	82
Выделение кода в функции	82
Анатомия функции	85
Заголовок функции	86
Модификатор видимости	86
Объявление имени функции	87
Параметры функции.....	87
Тип возвращаемого значения	88
Тело функции.....	89
Область видимости функции	89
Вызов функции.....	91
Пишем свои функции.....	92
Аргументы по умолчанию	94
Функции с единственным выражением	96
Функции с возвращаемым типом Unit	98

Именованные аргументы функций	99
Для любознательных: тип <code>Nothing</code>	102
Для любознательных: функции уровня файла в <code>Java</code>	103
Для любознательных: перегрузка функций	104
Для любознательных: имена функций в обратных кавычках	106
Глава 5. Числа	108
Числовые типы	108
Целые числа	110
Дробные числа	111
Форматирование значений типа <code>Double</code>	113
Преобразования числовых типов	115
Для любознательных: числа без знака	116
Для любознательных: манипуляции с битами	119
Глава 6. Строки	121
Интерполяция строк	121
Необработанные строки	124
Чтение ввода с консоли	126
Преобразование строк в числа	127
Регулярные выражения	130
Операции со строками	131
Строки неизменяемы	133
Сравнение строк	133
Для любознательных: Юникод	134
Глава 7. Null-безопасность и исключения	136
Допустимость <code>null</code>	136
Явный тип <code>null</code> в <code>Kotlin</code>	139
Время компиляции и время выполнения	141
Null-безопасность	141
Первый вариант: проверка <code>null</code> в операторе <code>if</code>	143
Второй вариант: оператор безопасного вызова	144
Использование безопасного вызова с <code>let</code>	145
Оператор объединения с <code>null</code>	147
Третий вариант: оператор проверки	149
Исключения	151

Выдача исключений	151
Обработка исключений	153
Выражения try/catch.....	154
Проверка предусловий	155
Для любознательных: пользовательские исключения	158
Для любознательных: проверяемые и непроверяемые исключения	160

ЧАСТЬ III. ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ И КОЛЛЕКЦИИ

Глава 8. Лямбда-выражения и тип функции	162
Представляем NyetHack.....	162
Анонимные функции.....	163
Лямбда-выражения	164
Тип функции	166
Неявный возврат	167
Аргументы функции	167
Идентификатор it	169
Получение нескольких аргументов	169
Поддержка автоматического определения типов.....	170
Более эффективные лямбда-выражения.....	171
Определение функции, которая получает функцию.....	174
Сокращенный лямбда-синтаксис.....	176
Встраиваемые функции	177
Лямбда-выражения и стандартная библиотека Kotlin.....	179
Для любознательных: ссылки на функции	182
Для любознательных: захват лямбда-выражений.....	182
Задание: новые титулы и настройки	184
Глава 9. Списки и множества	185
Списки.....	187
Обращение к элементам списка	188
Границы индексов и безопасные обращения по индексу	189
Проверка содержимого списка	190
Изменение содержимого списка	192
Итерация.....	196
Чтение файла в список.....	200

Деструктуризация.....	202
Множества	204
Создание множества	204
Добавление элементов в множество.....	206
Цикл while	208
Преобразование коллекций	210
Для любознательных: типы массивов	211
Для любознательных: «только для чтения» vs «неизменяемый»	212
Для любознательных: выражение break	213
Для любознательных: метки return	214
Задание: форматированный вывод меню таверны	216
Задание: улучшенное форматирование меню таверны	216
Глава 10. Ассоциативные массивы	217
Создание ассоциативного массива.....	217
Доступ к значениям в ассоциативном массиве.....	219
Добавление записей в ассоциативный массив	221
Изменение значений в ассоциативном массиве	223
Преобразования между списками и ассоциативными массивами.....	225
Перебор элементов ассоциативного массива	228
Задание: составные заказы	231
Глава 11. Основы функционального программирования.....	232
Преобразование данных.....	233
map.....	233
associate.....	235
Деструктуризация средствами функционального программирования.....	237
flatMap	238
map vs flatMap	241
Фильтрация данных.....	241
filter.....	241
Комбинирование данных	243
zip	243
Почему именно функциональное программирование?	245
Последовательности.....	246
Для любознательных: профилирование.....	248
Для любознательных: агрегирование данных	249

reduce	249
fold.....	250
sumBy	251
Для любознательных: ключевое слово vararg	251
Для любознательных: Arrow.kt	252
Задание: перестановка ключей и значений в ассоциативном массиве	253
Задание: поиск самого популярного пункта меню.....	254
Глава 12. Функции области видимости.....	255
apply.....	255
let.....	256
run	258
with	259
also.....	259
takeIf.....	260
Использование функций области видимости.....	261

ЧАСТЬ IV. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Глава 13. Классы	266
Объявление класса.....	266
Создание экземпляров.....	266
Функции класса	267
Видимость и инкапсуляция	268
Свойства класса	270
Get-методы и set-методы свойств	272
Видимость свойств	275
Вычисляемые свойства	276
Использование пакетов	278
Для любознательных: более пристальный взгляд на свойства var и val	280
Для любознательных: защита от изменяемости	283
Для любознательных: ограничение видимости рамками пакета.....	285
Глава 14. Инициализация.....	286
Конструкторы	286
Главный конструктор.....	287
Объявление свойств в главном конструкторе	289

Дополнительные конструкторы	290
Аргументы по умолчанию	292
Именованные аргументы	293
Блок инициализации	294
Порядок инициализации.....	296
Задержка инициализации	297
Поздняя инициализация	298
Отложенная инициализация.....	302
Для любознательных: подводные камни инициализации	305
Для любознательных: делегаты свойств.....	307
Задание: загадка Эскалибура.....	308
Глава 15. Наследование.....	310
Объявление класса Room	310
Создание подкласса	312
Проверка типов.....	318
Иерархия типов в языке Kotlin	320
Приведение типа	320
Умное приведение типа	322
Рефакторинг кода таверны.....	323
Для любознательных: Any	331
Для любознательных: оператор безопасного приведения типа.....	332
Глава 16. Объекты, классы данных и перечисления	333
Ключевое слово object.....	333
Объявления объекта	334
Объекты-выражения	339
Объекты-компаньоны	340
Вложенные классы	342
Классы данных	344
toString.....	345
equals и hashCode.....	345
сору	346
Деструктуризация объявлений	347
Классы-перечисления.....	348
Перегрузка операторов	350
Исследуем мир NyetHack.....	352

Для любознательных: объявление структурного сравнения.....	356
Для любознательных: алгебраические типы данных.....	358
Для любознательных: классы-значения.....	360
Задание: другие команды	361
Задание: реализация карты мира	362
Задание: колокольный звон.....	362
Глава 17. Интерфейсы и абстрактные классы.....	363
Определение интерфейса	363
Реализация интерфейса	364
Реализация по умолчанию.....	367
Абстрактные классы	368
Сражение в NyetHack.....	371
Задание: дополнительные монстры	376

ЧАСТЬ V. KOTLIN ДЛЯ ОПЫТНЫХ ПРОГРАММИСТОВ

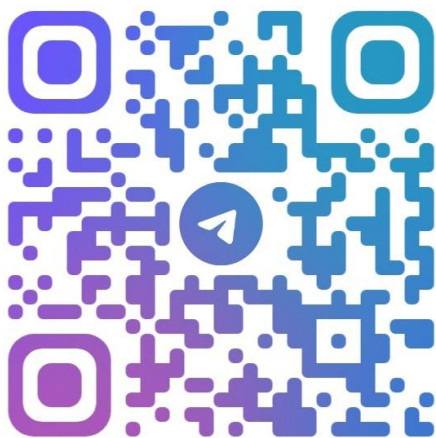
Глава 18. Обобщения.....	378
Определение обобщенных типов	378
Обобщенные функции	380
Ограничения обобщений	381
in и out	384
Добавление наград в NyetHack	388
Для любознательных: ключевое слово reified	394
Глава 19. Расширения.....	397
Определение функции-расширения.....	397
Объявление расширения для суперкласса	399
Обобщенные функции-расширения	400
Операторные функции-расширения.....	402
Свойства-расширения.....	404
Расширения для типов, допускающих null.....	406
Расширения: как это устроено	407
Видимость расширений.....	408
Расширения в стандартной библиотеке Kotlin	409
Для любознательных: литералы функций с получателями.....	410
Задание: расширение рамок.....	412

Глава 20. Сопрограммы	413
Блокирующие вызовы	414
Включение сопрограмм	416
Строители сопрограмм	417
Области видимости сопрограмм	418
Структурированный параллелизм	419
Использование клиента HTTP	422
asunc и await	428
Для любознательных: состояние гонки	430
Для любознательных: Kotlin на стороне сервера	433
Задание: никаких отмен	434
Глава 21. Потоки данных	435
Создание потоков данных	436
MutableStateFlow	443
Завершение потоков данных	447
Преобразования потоков данных	450
Обработка ошибок в потоках данных	452
Для любознательных: SharedFlow	454
Глава 22. Каналы	457
Распределение работы с использованием каналов	457
Отправка данных в канал	459
Получение данных из канала	460
Закрытие канала	463
Объединение заданий	464
Для любознательных: другие особенности поведения каналов	469
Встречные каналы (rendezvous)	469
Буферизованные каналы (buffered)	469
Неограниченные каналы (unlimited)	470
Каналы с заменой (conflated)	470

ЧАСТЬ VI. СОВМЕСТИМОСТЬ И МУЛЬТИПЛАТФОРМЕННЫЕ ПРИЛОЖЕНИЯ

Глава 23. Совместимость с Java	474
Взаимодействие с классом Java	474
Совместимость и null	476

Соответствие типов	479
Get-методы, set-методы и совместимость	481
За пределами класса	483
Исключения и совместимость	492
Функциональные типы в Java	495
Глава 24. Знакомство с Kotlin Multiplatform	497
Что такое Kotlin Multiplatform?	498
Планирование мультиплатформенного проекта	499
Первый мультиплатформенный проект	500
Определение целевой платформы Kotlin/JVM	502
Определение общего кода	504
expect и actual	505
Глава 25. Kotlin/Native	511
Объявление целевой платформы macOS	513
Написание нативного кода на Kotlin	515
Запуск приложения Kotlin/Native	517
Вывод Kotlin/Native	519
Для любознательных: Kotlin Multiplatform Mobile	520
Для любознательных: другие нативные платформы	522
Глава 26. Kotlin/JS	524
Объявление поддержки Kotlin/JS	524
Взаимодействие с DOM	528
Ключевое слово external	533
Выполнение произвольного кода JavaScript	535
Динамические типы	537
Для любознательных: фреймворки клиентской части	540
Задание: комиссионные при обмене валюты	541
Послесловие	542
Что дальше?	542
Бесстыдная самореклама	542
Спасибо вам	543
Глоссарий	544



@KOTLINSENIOR

Благодарности

Хотя на обложке указаны наши имена, книга создается не только усилиями авторов. Множество людей внесли свой вклад в этот труд. Мы благодарны всем, кто помог нашей книге стать такой, какой вы ее видите.

- Брайан Силлз (Bryan Sills), Майкл Йотив (Michael Yotive), Нейт Соттек (Nate Sottek), Джереми Шерман (Jeremy Sherman) и Марк Дьюран (Mark Duran) великодушно делились своим мнением относительно второго издания книги.
- Эрик Максвелл (Eric Maxwell) проводил обучение по ранней версии второго издания и подготовил материал для глав о сопрограммах, каналах и потоках данных.
- Лорен Клингман (Loren Klingman) и Джейк Соуэр (Jake Sower) дали нам информацию для главы, посвященной Kotlin/JS.
- Дрю Фицпатрик (Drew Fitzpatrick) полностью прочитал раннюю версию второго издания и снабдил нас материалами о Kotlin Multiplatform и Kotlin/Native.
- Лив Витал (Liv Vitale), Кристиан Кеур (Christian Keur), Закари Вальдовски (Zachary Waldowski) и Дэвид Хауз (David House) постоянно делились опытом работы на других платформах и участвовали в обсуждениях Kotlin/JS и Kotlin/Native. Спасибо вам за то, что вы отвечали на наши странные вопросы, когда мы пытались казаться более осведомленными, чем были на самом деле.
- Джавонтай Макэлрой (Javontay McElroy), наш талантливый дизайнер из Big Nerd Ranch, создал шпаргалку IntelliJ IDEA для печатного издания. Ты смело вступил на неизведанную территорию печатных материалов и разработал дизайн с нуля — благодарим!
- Эрик Уилсон (Eric Wilson), Мэдисон Уицлер (Madison Witzler), Фрэнклин О'Нил (Franklin O'Neal) и CJ Best — спасибо великим умам отдела повышения квалификации Big Nerd Ranch. Наши занятия — и как следствие, эта книга — стали возможными лишь благодаря вашей усердной работе.
- Когда мы рассказывали нашему редактору Элизабет Холадей (Elizabeth Holaday) о своих планах на второе издание книги, первое, что мы от нее услышали: «Похоже, обновление будет серьезным». Конечно, ее оценка была абсо-

лотно верной. Спасибо за самоотверженную работу по совершенствованию книги, доработку ее сильных сторон и устранение недостатков.

- Саймону Пэйменту (Simone Payment), нашему корректору и редактору, спасибо за помощь в последних приготовлениях к выпуску книги.
- Элли Волкхаузен (Ellie Volckhausen) создала дизайн обложки. Это круто!
- Крис Лопер (Chris Loper) из [IntelligentEnglish.com](https://intelligentenglish.com) разработал и подготовил печатную и цифровую версию книги. Мы также постоянно пользовались его инструментарием DocBook.
- Аарону Хиллегассу (Aaron Hillegass) и Стейси Генри (Stacy Henry) — наша благодарность! Эта книга не появилась бы на свет без Big Nerd Ranch — компании, которую Аарон основал, а Стейси возглавляет.

Наконец, мы благодарим всех наших студентов. Работа преподавателя дает нам массу возможностей оказаться в шкуре студента, и это здорово! Постоянное узнавание нового — одна из самых замечательных сторон нашей работы, мы делали это совместно с вами — и это было настоящим удовольствием. Надеемся, что уровень книги будет соответствовать вашим энтузиазму и решимости.

Подписывайся на самый топовый канал по Kotlin:
<https://t.me/KotlinSenior>

Представляем Kotlin

В 2011 году компания JetBrains анонсировала альтернативу языкам Java и Scala — язык программирования Kotlin, код которого тоже выполняется под управлением виртуальной машины Java (Java Virtual Machine). Шесть лет спустя Google объявил об официальной поддержке Kotlin как языка разработки для операционной системы Android.

И Kotlin быстро превратился из просто «перспективного» в язык поддержки приложений для лидирующей мобильной операционной системы. Сегодня крупные компании вроде Google, Uber, Netflix, Capital One, Amazon и других официально приняли на вооружение Kotlin, чему способствовали его компактность, современные возможности и полная совместимость с Java.

Почему Kotlin?

Чтобы оценить привлекательность Kotlin, стоит сначала разобраться, какое место в современном мире разработки ПО занимает Java. Код на Kotlin выполняется под управлением Java Virtual Machine, поэтому эти два языка тесно взаимосвязаны.

Java — надежный и проверенный язык, чаще других используемый для разработки промышленных приложений на протяжении многих лет. Но он был создан в далеком 1995 году, и с того времени критерии оценки хорошего языка программирования изменились. В Java отсутствуют многие удобные опции, которые есть у современных языков.

Создатели Kotlin учли недостатки проектных решений, принятых при разработке Java (и других языков, например Scala). Они расширили возможности языка и исправили в нем многое, что доставляло массу неудобств в языках, разработанных ранее. Из этой книги вы узнаете, чем Kotlin лучше Java и почему работать с ним удобнее.

Kotlin — это не просто улучшенный язык для виртуальной машины Java. Это мультиплатформенный язык общего назначения: на Kotlin можно писать нативные приложения для macOS, Windows и iOS, приложения на JavaScript и, конечно, приложения для Android. В последнее время компания JetBrains прилагает значительные усилия для разработки кросс-платформенных возможностей; Kotlin Multiplatform предоставляет уникальную возможность совместного использования кода разными приложениями, что привело к росту популярности Kotlin за пределами виртуальной машины Java.

Для кого написана эта книга

Мы написали эту книгу для разработчиков разного уровня: тех, кто имеет богатый опыт создания приложений для Android и кому не хватает возможностей Java; тех, кто разрабатывает серверный код и заинтересован в возможностях Kotlin; тех, кто стремится к совместному использованию кода Kotlin в нативных и веб-приложениях; а также для новичков, решившихся на самостоятельное изучение высокопроизводительного компилируемого языка.

Поддержка Android может стать мотивом для изучения Kotlin, но наша книга не ограничивается рассказом о программировании для Android. Более того, весь код в книге не зависит от фреймворка Android. Тем не менее, если вас интересует именно использование Kotlin для разработки Android-приложений, здесь вы найдете основные приемы, которые упростят процесс написания приложений для Android на Kotlin.

Несмотря на то что на Kotlin оказали влияние некоторые другие языки, вам не придется изучать все тонкости этих языков, чтобы успешно работать с Kotlin. Время от времени мы будем приводить код Java, эквивалентный написанному вами коду на Kotlin. Также мы будем указывать на сходство с другими языками там, где это актуально. Программистам с опытом разработки на Java это поможет понять связь Kotlin с другими поддерживаемыми платформами. Но если эти параллели вам не очень знакомы, примеры решения тех же задач на другом языке полезны, поскольку помогают понять идеи, повлиявшие на формирование Kotlin.

Как пользоваться этой книгой

Эта книга — не справочник. Наша цель — помочь вам освоить важнейшие особенности программирования на Kotlin. Вы будете изучать язык в процессе создания проектов. Чтобы извлечь максимум пользы из книги, мы рекомендуем вручную набирать все примеры кода по ходу чтения. Такая работа с примерами поможет вам развить мышечную память и даст понимание, позволяющее переходить от одной главы к другой.

Материал каждой следующей главы основан на предыдущем, и мы рекомендуем ничего не пропускать. Даже если вы считаете, что та или иная тема знакома вам по другим языкам, мы рекомендуем прочитать об этом здесь: в Kotlin многое реализовано иначе. Мы начнем с вводных тем, таких как переменные и управление программной логикой, а затем перейдем к приемам объектно-ориентированного и функционального программирования, опробуем подход к выполнению асинхронного кода и познакомимся с мультиплатформенными возможностями Kotlin. К концу чтения книги вы станете продвинутым разработчиком на Kotlin.

Хотим добавить, что спешить не стоит: делайте паузы и обращайтесь к документации Kotlin по ссылке kotlinlang.org/docs/reference, если какая-то тема вас особенно заинтересовала, — и экспериментируйте.

Для любознательных

В большинстве глав есть один-два раздела «Для любознательных». В них раскрываются внутренние принципы работы языка Kotlin. Примеры в основном тексте глав не зависят напрямую от этой информации, но дополнительные сведения из разделов для любознательных могут вам пригодиться.

Задания

Многие главы заканчиваются одним или несколькими заданиями. Их решение поможет вам лучше понять язык Kotlin. Выполняя их, вы совершенствуете мастерство владения языком Kotlin.

Мы рекомендуем создавать копии проектов, прежде чем браться за новые задания; проекты в книге часто базируются на основе предыдущих решений, и ваши действия не должны нанести им вред. Решения к упражнениям из книги можно загрузить по адресу bignerdranch.com/kotlin-2e-solutions/.

Шрифтовые обозначения

В процессе работы над проектами мы будем знакомить вас с теорией, а затем показывать, как применить ее на практике. Для большей наглядности мы используем в книге следующие шрифтовые обозначения.

Переменные, их значения и типы набраны **моноширинным** шрифтом.

Для листингов с кодом мы также используем моноширинный шрифт. Код в листинге, который вам предлагается дописать, выделен **полу жирным**. Если же код из листинга нужно удалить, то он будет **зачеркнут**. В следующем примере вам предлагается удалить строку кода, объявляющую переменную `y`, и добавить в код переменную `z`:

```
var x = "Python"  
var y = "Java"  
var z = "Kotlin"
```

Kotlin — относительно молодой язык, поэтому многие соглашения по его оформлению еще только формируются. Вероятно, со временем вы выработаете собственный стиль, но мы рекомендуем придерживаться стилевых руководств JetBrains и Google.

- Правила оформления кода JetBrains — kotlinlang.org/docs/coding-conventions.html.
- Руководство по стилю Google — developer.android.com/kotlin/style-guide.

Заглядывая вперед

Не спешите, работая над примерами из этой книги. Когда вы освоите синтаксис Kotlin, вы убедитесь, что этот язык — ясный, гибкий и прагматичный. А пока этого не случилось, просто вдумчиво изучайте его — всегда полезно.

От издательства

Язык Kotlin был создан российской компанией JetBrains. Само название Kotlin происходит от острова Котлин в Финском заливе, недалеко от Санкт-Петербурга.

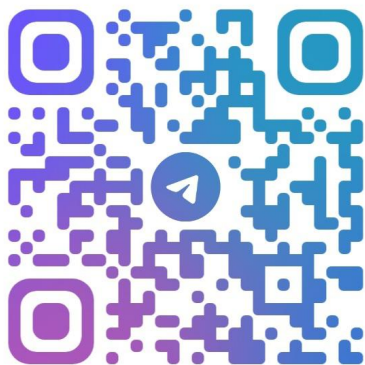
При переводе терминов использовался англо-русский глоссарий Kotlin, представленный JetBrains.

Примеры в книге основаны на игре в стиле фэнтези. Переменные, функции и другие элементы кода имеют «говорящие» имена; на консоль выводятся тексты, связанные с приключениями героя. Поскольку код должен оставаться в нетронutom виде, нарратив игры не переводился (за исключением случаев, когда это было необходимо для понимания логики программы). Мы надеемся, что базового знания английского языка будет достаточно, чтобы следить за развитием сюжета :).

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.



@KOTLINSENIOR

Часть I

Первые шаги

Первые две главы книги посвящены основам работы IntelliJ IDEA, ведущей IDE для разработки приложений Kotlin. Вы создадите простой проект, в котором освоите базовые возможности языка. А начнем мы с рассказа о типах Kotlin — механизме классификации данных, с которыми вы будете работать.

1. Ваше первое приложение на Kotlin

В этой главе вы напишете свою первую программу на языке Kotlin, используя IntelliJ IDEA. Это можно рассматривать как обряд посвящения в программирование — вы познакомитесь со средой разработки, создадите в ней новый проект на Kotlin, напишете и запустите код, а также увидите результаты его выполнения. Проект, созданный в этой главе, станет вашей испытательной площадкой для разработки других приложений на Kotlin.

Установка IntelliJ IDEA

IntelliJ IDEA — это интегрированная среда разработки (integrated development environment, IDE) для языка Kotlin, созданная, как и сам язык, командой JetBrains. Чтобы приступить к работе, загрузите IntelliJ IDEA Community Edition с сайта JetBrains по ссылке [jetbrains.com/idea/download](https://www.jetbrains.com/idea/download)¹ (рис. 1.1).

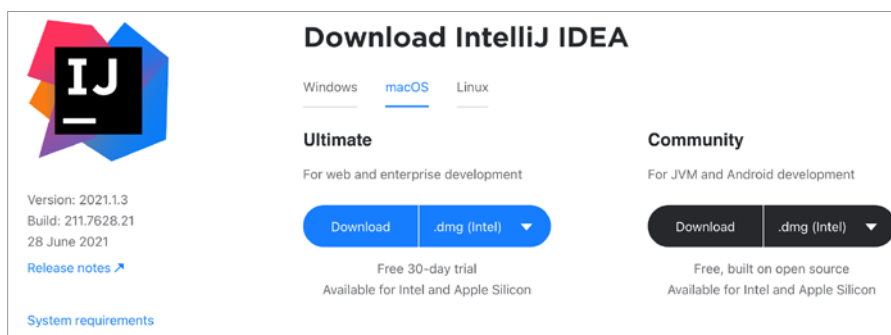


Рис. 1.1. Загрузка IntelliJ IDEA Community Edition

Затем выполните указания для своей системы, приведенные в инструкции по установке и настройке на сайте JetBrains: [jetbrains.com/help/idea/installation-guide.html#standalone](https://www.jetbrains.com/help/idea/installation-guide.html#standalone).

¹ Есть русскоязычная версия страницы загрузки <https://www.jetbrains.com/ru-ru/idea/download>. Но упоминаемая далее инструкция доступна только на английском языке. — *Примеч. ред.*

IntelliJ IDEA, или просто IntelliJ, помогает писать хорошо структурированный код на Kotlin. Кроме того, она упрощает процесс разработки с помощью встроенных инструментов для запуска, отладки, исследования и рефакторинга кода. Узнать, почему мы рекомендуем IntelliJ для создания кода на Kotlin, вы можете в разделе «Для любознательных: зачем использовать IntelliJ?».

Ваш первый проект на Kotlin

Поздравляем: теперь у вас есть язык программирования Kotlin и мощная среда разработки. Осталось только научиться свободно на нем «разговаривать». Повестка дня — создать Kotlin-проект.

Большинство проектов, над которыми вы будете работать в этой книге, связаны с игрой в стиле фэнтези, в которой герой выполняет героические миссии, побеждает злобных монстров, спасает города от опасностей и вообще делает то, что положено делать героям. В первом проекте мы построим «доску поручений» — систему, которая будет направлять нашего героя по имени Мадригал (Madrigal) к задачам, требующим его внимания. Запустите IntelliJ. Откроется окно приветствия **Welcome to IntelliJ IDEA** (рис. 1.2).

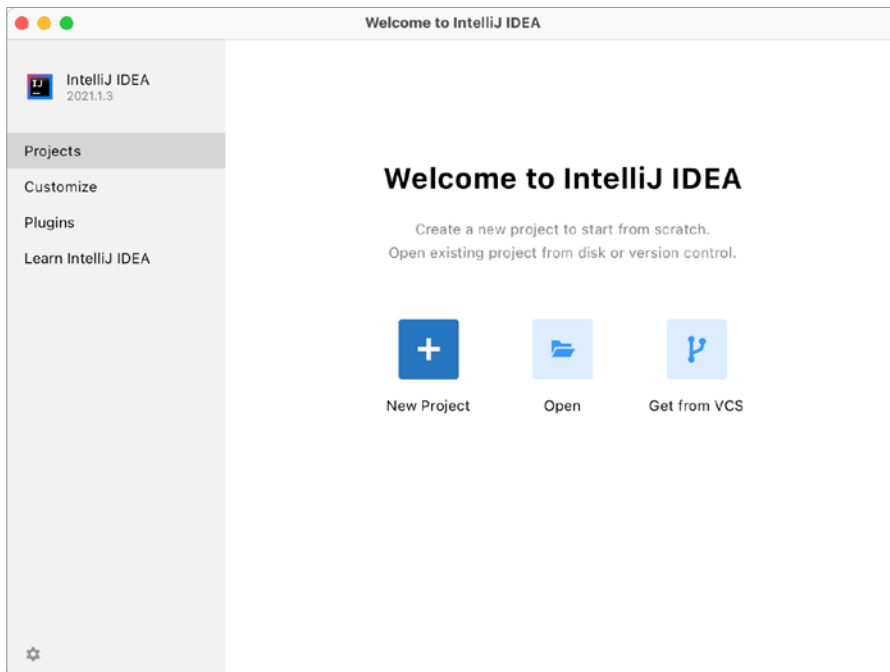


Рис. 1.2. Диалоговое окно с приветствием

(Если вы уже запускали IntelliJ после установки, среда может автоматически загрузить последний открывавшийся проект. Чтобы вернуться к окну приветствия, закройте проект командой меню **File ▶ Close Project.**)

Выберите вариант **New Project**. IntelliJ отобразит новое окно **New Project**, как показано на рис. 1.3.

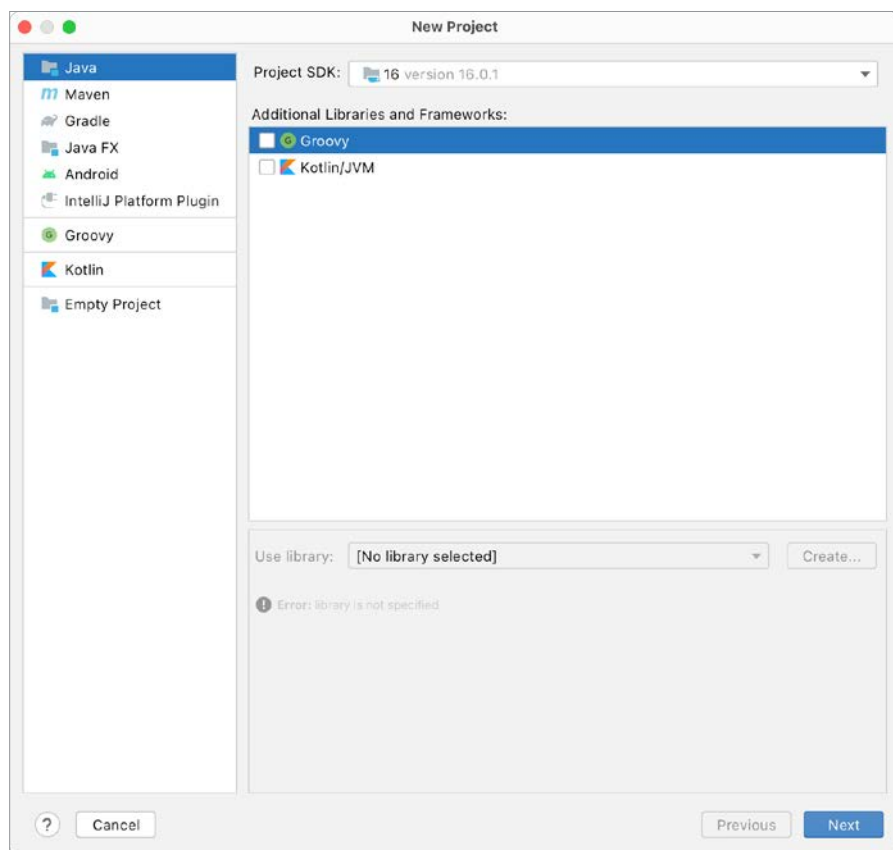


Рис. 1.3. Диалоговое окно **New Project**

В окне **New Project** в левой панели выберите **Kotlin** (рис. 1.4).

В IntelliJ можно писать код и на других языках, кроме Kotlin, например на Java и Groovy, для которых имеется встроенная поддержка. Установив дополнительные плагины, вы получите возможность писать код в IntelliJ на таких языках, как Python, Scala, Dart и Rust (список далеко не полный). Выбрав вариант **Kotlin** в левой части окна **New Project**, вы указываете, что собираетесь работать на Kotlin.

Теперь рассмотрим настройки проекта на центральной панели диалогового окна.

В верхней части окна **New Project** введите в поле **Name** текст `bounty-board`. Поле **Location** заполняется автоматически. Вы можете оставить информацию в этом поле без изменений или выбрать другую папку, щелкнув на значке с папкой справа от поля.

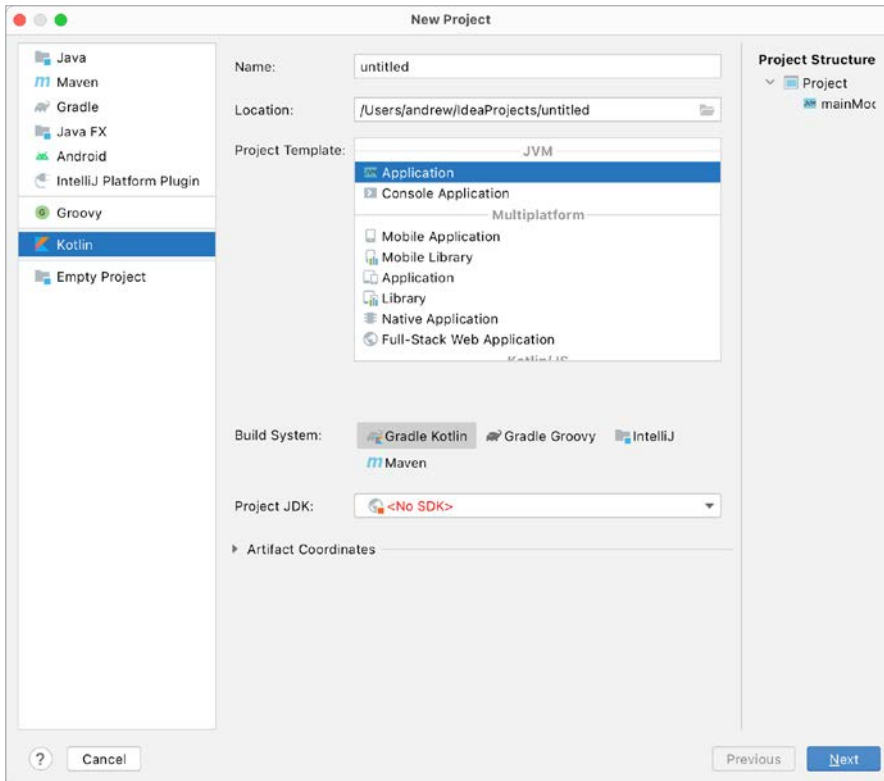


Рис. 1.4. Создание проекта Kotlin

Меню **Project Template** содержит три категории параметров: **JVM**, **Multiplatform** и **Kotlin/JS** (также возможна категория **Experimental**). Выберите вариант **Application** под заголовком **JVM**. Тем самым вы сообщаете IntelliJ, что собираетесь писать код на Kotlin, предназначенный для виртуальной машины Java.

Код на Kotlin может компилироваться для любой из трех платформ, поддерживаемых языком: виртуальной машины Java, платформ x86 и ARM (из раздела **Multiplatform** списка **Project Template**), а также JavaScript (сокращается до JS). Иногда для обозначения «специализации» Kotlin в зависимости от платформы используются термины **Kotlin/JVM**, **Kotlin/Native** и **Kotlin/JS**.

Kotlin/JVM — то, что большинство разработчиков имеют в виду, когда говорят о Kotlin. **Kotlin/JVM** обозначает код для виртуальной машины Java. Аналогично

Kotlin/JS используется для вывода кода JavaScript из кода Kotlin. А Kotlin/Native обозначает все программы Kotlin, компилируемые в нативный машинный код. Как подсказывает название категории **Multiplatform**, Kotlin/Native можно использовать для построения программных продуктов для многих платформ: библиотек iOS, нативных настольных приложений, встроенных устройств (для самых амбициозных) и т. д.

Далее мы будем везде применять сокращенное название виртуальной машины Java — JVM (Java Virtual Machine). Эта аббревиатура часто используется в сообществе Java-разработчиков. Узнать больше о программировании для JVM можно в разделе «Для любознательных: программирование для JVM» в конце главы.

Kotlin/JVM — наиболее зрелая из трех платформ. Она изначально поддерживалась Kotlin и продолжает оставаться одной из самых распространенных платформ для разработки на этом языке. В большей части книги будет использоваться Kotlin/JVM. Мы приняли такое решение по нескольким причинам.

- JVM значительно упрощает задачу достижения платформенной независимости: ваш код может выполняться везде, где может работать JVM. Распространяется один двоичный файл, который будет работать на любом компьютере независимо от его архитектуры.
- Java — чрезвычайно зрелый язык, и для него существует множество API, которые мы используем в книге. Высокоуровневые API упрощают выполнение некоторых задач (по сравнению с низкоуровневыми платформенными API).
- Примеры, приведенные в книге, выполняются внутри IDE, но их также можно запускать из любого терминала. Это несколько снижает ценность JavaScript как платформы, так как он выполняется в основном в браузере, а не в терминале.

В особенностях Kotlin для этих трех платформ много общего, и то, что вы знаете о Kotlin/JVM, применимо к Kotlin/JS и Kotlin/Native. Не все API доступны на всех платформах, поэтому мы будем особо отмечать API, которые используются только с JVM. Использование Kotlin с другими платформами мы более подробно рассмотрим в главах 24, 25 и 26.

Но вернемся к настройке нового проекта. В группе **Build System** выберите вариант **Gradle Groovy**. В списке **Project JDK** выберите версию Java, которая будет использована при компоновке с JDK (Java Development Kit). Мы рекомендуем применять версии от Java 8 (часто обозначается Java 1.8) до Java 15.

Если нужная версия Java отсутствует в раскрывающемся списке, значит, IntelliJ не находит установленную копию на вашем компьютере. Если вы точно знаете, что пакет JDK у вас установлен, и хотите использовать именно эту версию, выберите команду **Add JDK...** и найдите копию на диске. В противном случае IntelliJ установит для вас JDK при помощи команды **Download JDK....** Мы рекомендуем выбрать в поле

Version значение 15, а в поле Vendor — значение AdoptOpenJDK (HotSpot), но подойдет любой вендор. Когда IntelliJ завершит установку, можно браться за дело.

Зачем нужен JDK для написания программы на Kotlin? JDK открывает среде IntelliJ доступ к JVM и инструментам Java, которые необходимы для перевода кода на Kotlin в байт-код (об этом ниже). В принципе, подойдет любая версия, начиная с шестой. Но по нашему опыту, на момент написания этой книги JDK 8 и более ранние версии работали наиболее стабильно.

Если диалоговое окно выглядит так, как показано на рис. 1.5, щелкните на кнопке Next, а затем на кнопке Finish в следующем диалоговом окне для подтверждения введенных значений.

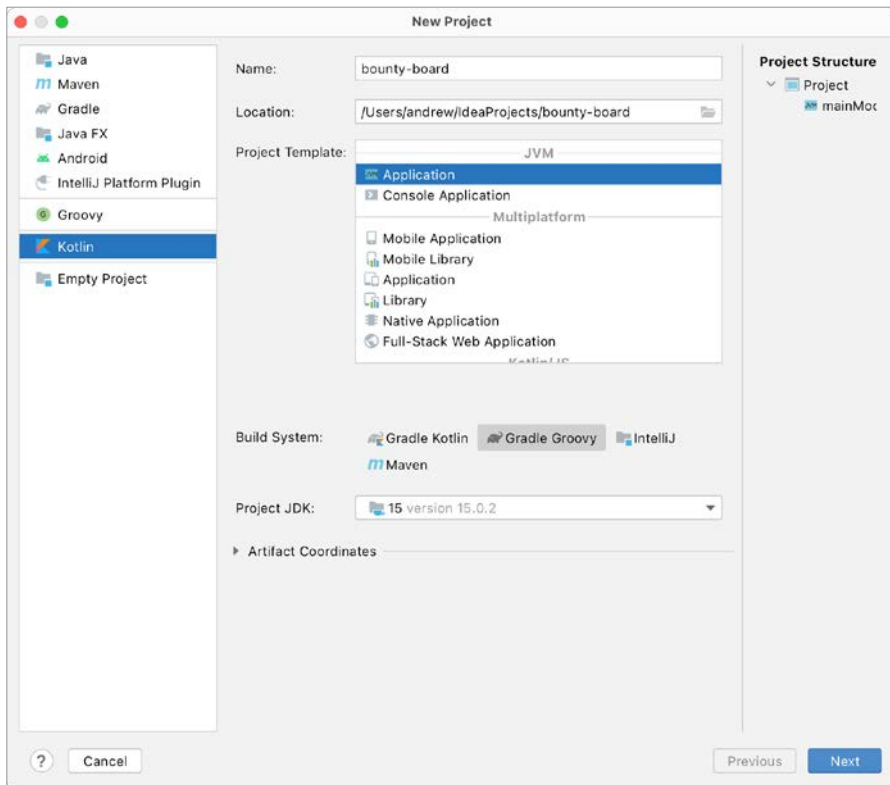


Рис. 1.5. Настройка проекта

IntelliJ сгенерирует проект с названием **bounty-board** и отобразит проект в стандартном двухпанельном представлении (рис. 1.6). IntelliJ создаст на диске папку и ряд подпапок с файлами проекта в месте, путь к которому указан в поле **Project location**.

Панель слева отображает окно с *инструментами проекта*. Панель справа в данный момент пуста. Здесь будет отображаться *окно редактора*, где вы сможете просматривать и редактировать содержимое своих Kotlin-файлов.

В окне инструментов проекта выводится список файлов, используемых в проекте `bounty-board`, как показано на рис. 1.7.

Проект — это весь исходный код программы, а также информация о зависимостях и конфигурациях. Проект может быть разбит на один или более *модулей*, которые можно рассматривать как своего рода подпроекты. По умолчанию новый проект содержит всего один модуль, которого более чем достаточно для начала вашей работы.

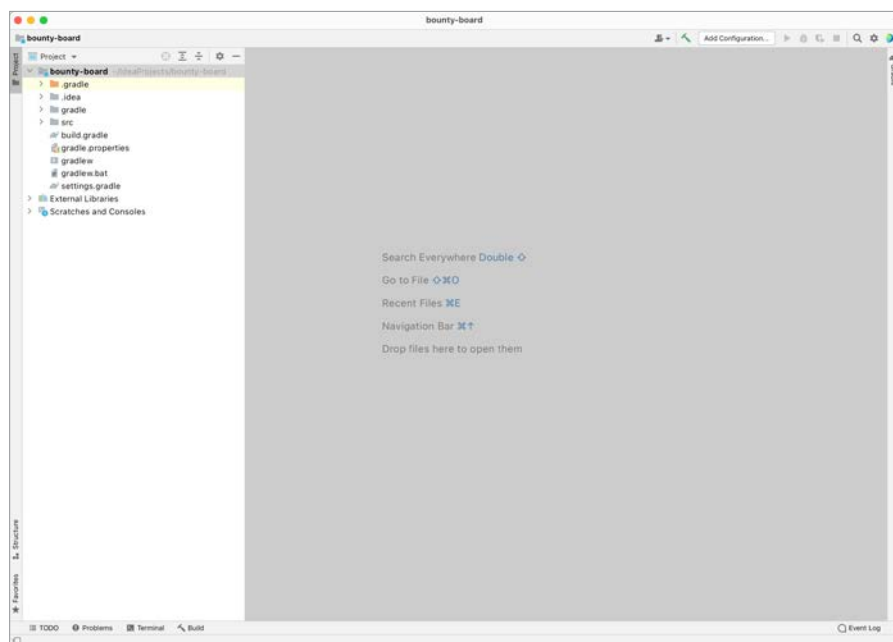


Рис. 1.6. Стандартное двухпанельное представление

Папки `.gradle` и `.idea` на рис. 1.7 могут быть скрыты при открытии проекта в проводнике файлов. Это служебные папки, к которым вам обращаться не стоит; в папке `.gradle` хранятся кэши, используемые системой сборки, а в папке `.idea` — файлы с параметрами конфигурации проекта и IDE.

Папка `gradle` и такие файлы, как `gradlew` и `gradlew.bat`, относятся к Gradle Wrapper — автономной копии системы построения Gradle, которая требуется для установки Gradle на вашем компьютере. Файлы `build.gradle`, `gradle.properties` и `settings.gradle` содержат параметры конфигурации системы сборки Gradle. Они определяют различные данные: имя проекта, уровень языка, модули проекта и зависимости

вашего проекта. Не трогайте эти автоматически сгенерированные файлы.

Категория **External Libraries** содержит информацию о библиотеках, от которых зависит проект. Раскрыв эту категорию, вы увидите, что среда IntelliJ автоматически добавила Java и некоторые стандартные библиотеки Kotlin как зависимости вашего проекта. (Узнать больше о структуре проектов в IntelliJ можно на сайте JetBrains — jetbrains.org/intellij/sdk/docs/basics/project_structure.html. Структура проектов Gradle также рассматривается на сайте Gradle — docs.gradle.org/current/userguide/organizing_gradle_projects.html.)

В папке `src/main/kotlin` будут сохраняться все файлы проекта. А теперь пришло время создавать и редактировать ваш первый файл на Kotlin.

Ваш первый файл на Kotlin

Щелкните правой кнопкой мыши на папке `src/main/kotlin` в окне с инструментами проекта. В открывшемся контекстном меню выберите сначала пункт **New**, а затем **Kotlin File/Class** (рис. 1.8).

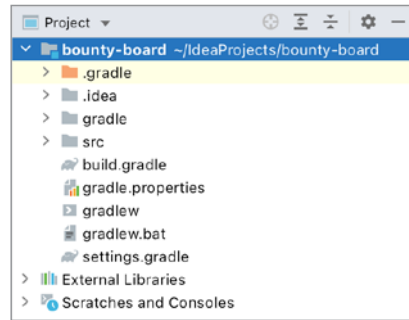


Рис. 1.7. Представление проекта

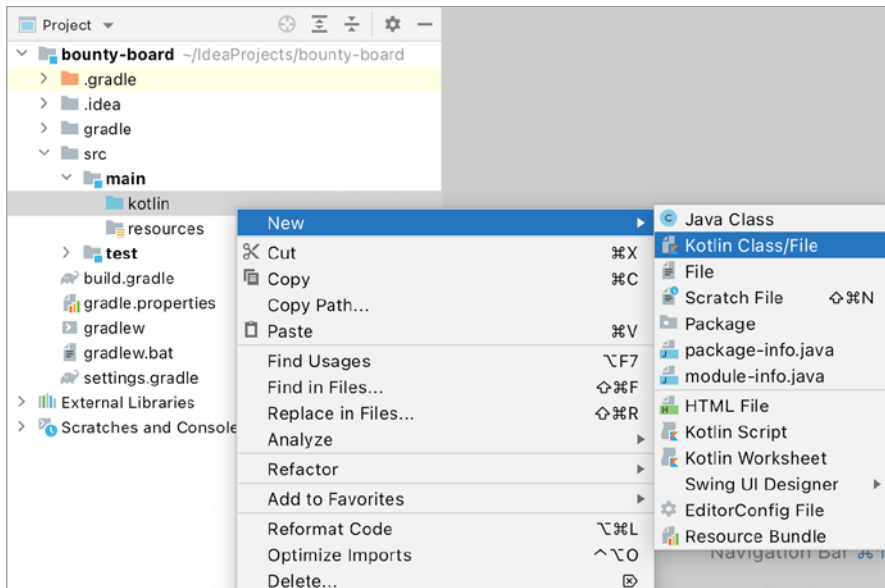


Рис. 1.8. Создание нового файла Kotlin

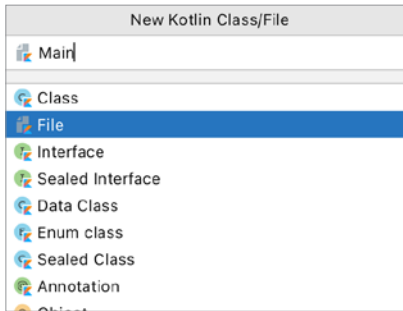


Рис. 1.9. Присваивание имени файлу

В диалоговом окне **New Kotlin Class/File** в поле **Name** введите **Main**, затем дважды щелкните на пункте **File** в открывшемся списке (рис. 1.9).

IntelliJ создаст в проекте новый файл `src/main/kotlin/Main.kt` и отобразит его содержимое в окне редактора справа (рис. 1.10). Расширение `.kt` указывает, что файл содержит исходный код на языке Kotlin, подобно тому как расширение `.java` сообщает, что файл содержит код Java или `.py` — код Python.

Наконец-то все готово к написанию кода на Kotlin. Разомните пальцы и приступайте.

Введите следующий код в окне редактора `Main.kt`. (Напоминаем, что в книге код, который вам следует ввести, будет выделяться полужирным шрифтом.)

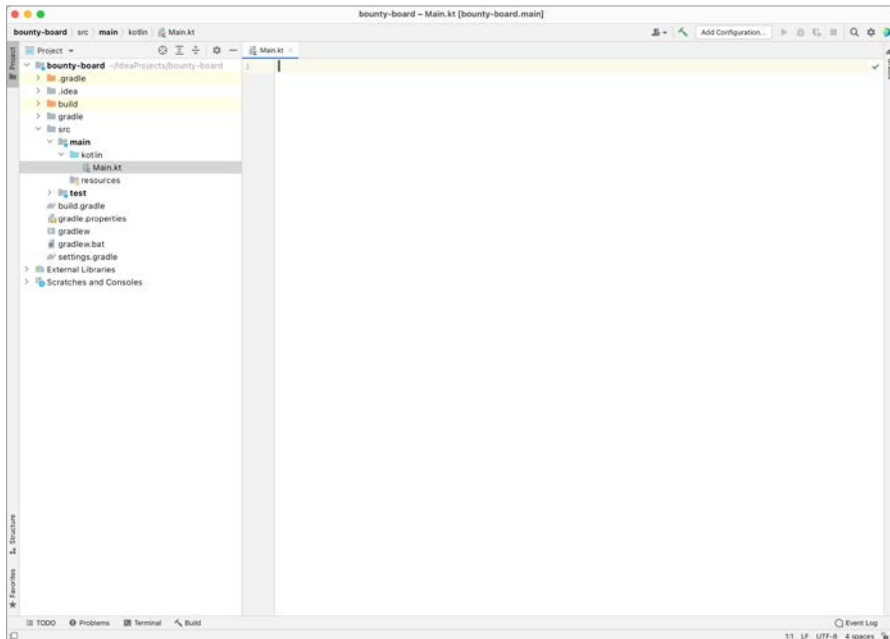


Рис. 1.10. Пустой файл `Main.kt` в окне редактора

Листинг 1.1. "Hello, world!" на Kotlin (`Main.kt`)

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

Написанный код может выглядеть непривычно. Не отчаивайтесь — к концу этой книги чтение и написание на Kotlin станет для вас абсолютно естественным. А пока что достаточно понимать код хотя бы в общих чертах.

Код в листинге 1.1 создает новую *функцию*. Функция — это группа инструкций, которые можно выполнить позднее. В главе 4 вы научитесь определять функции и работать с ними.

Конкретно эта функция — `main` — имеет особое значение в Kotlin. Она определяет начальную точку программы. Ее также называют *точкой входа приложения*; чтобы проект `bounty-board` (или любую другую программу) можно было запустить, в нем обязательно должна быть определена точка входа. Все проекты в этой книге начинаются с функции `main`.

Ваша функция `main` содержит одну инструкцию (иногда инструкции называют *командами*): `println("Hello,world!").println()` — это тоже функция, встроенная в *стандартную библиотеку Kotlin*. Она доступна для всех платформ, поддерживаемых Kotlin. После запуска программа выполнит `println("Hello,world!")`, и IntelliJ выведет на экран строку, указанную в скобках (без кавычек — `Hello, world!`).

Запуск вашего файла на языке Kotlin

Когда вы закончите вводить код из листинга 1.1, IntelliJ отобразит зеленую стрелку ▶ слева от первой строки кода (рис. 1.11). (Если значок не появился или вы видите красное подчеркивание под именем файла на вкладке или где-нибудь во введенном коде, то это значит, что в коде допущена ошибка. Исправьте введенный код: он должен точно совпадать с кодом в листинге 1.1.)

Пришло время оживить программу — пусть она поприветствует мир. Нажмите кнопку запуска. В появившемся меню выберите `Run 'MainKt'` (рис. 1.12). Тем самым вы сообщаете IntelliJ, что хотите посмотреть на программу в действии.

После запуска IntelliJ выполнит код, содержащийся в фигурных скобках (`{}`), строку за строкой, и завершит выполнение. Также в нижней части окна IntelliJ появится новое инструментальное окно (рис. 1.13).



Рис. 1.11. Кнопка запуска программы



Рис. 1.12. Запуск `Main.kt`



Рис. 1.13. Инструментальное окно Run (консоль)

Это — *инструментальное окно запуска* (run tool window), также известное как *консоль* (далее мы будем называть его именно так). Оно отображает информацию о происходящем после того, как IntelliJ запустит программу, и результаты, которые выводит программа. В консоли должно появиться сообщение **Hello, world!** и сообщение **Process finished with exit code 0**, что означает успешное завершение программы. Эта строка завершает консольный вывод при отсутствии ошибок, и с этого момента мы больше не будем ее упоминать.

Компиляция и выполнение кода Kotlin/JVM

С момента выполнения команды **Run 'MainKt'** до вывода **Hello, World!** в консоли происходит множество событий.

Прежде всего, IntelliJ компилирует код Kotlin, используя *компилятор* `kotlinc-jvm`. Это означает, что IntelliJ транслирует код на Kotlin в *байт-код* — язык, на котором «разговаривает» JVM. Если у `kotlinc` возникнут проблемы с переводом, он выведет сообщение об ошибке (ошибках), которое подскажет, что именно необходимо исправить. Однако если компиляция прошла гладко, IntelliJ перейдет к фазе выполнения.

В фазе выполнения байт-код, сгенерированный `kotlinc-jvm`, исполняется JVM. На консоли отображается все, что выводит программа в процессе выполнения, например текст, указанный в вызове функции `println()`.

После выполнения всех инструкций в байт-коде JVM прекратит работу и IntelliJ выведет код завершения в консоль, сообщая вам о том, была работа завершена успешно или с ошибкой.

Для изучения материала этой книги вам не обязательно досконально понимать процесс компиляции в Kotlin. Тем не менее мы более подробно рассмотрим байт-код в главе 2.

Kotlin REPL

Иногда возникает необходимость протестировать маленький кусочек кода на Kotlin, чтобы посмотреть, что происходит при его выполнении, — по аналогии с тем, как вы записываете последовательность вычислений на листке бумаги. Такая возможность особенно полезна в процессе изучения языка. Вам повезло: IntelliJ предоставляет инструмент для быстрого тестирования кода без создания файла. Этот инструмент называется *Kotlin REPL*. Название объясним позже, а сейчас посмотрим, что он делает.

В IntelliJ откройте Kotlin REPL, выбрав в меню команду **Tools ▸ Kotlin ▸ Kotlin REPL** (рис. 1.14).

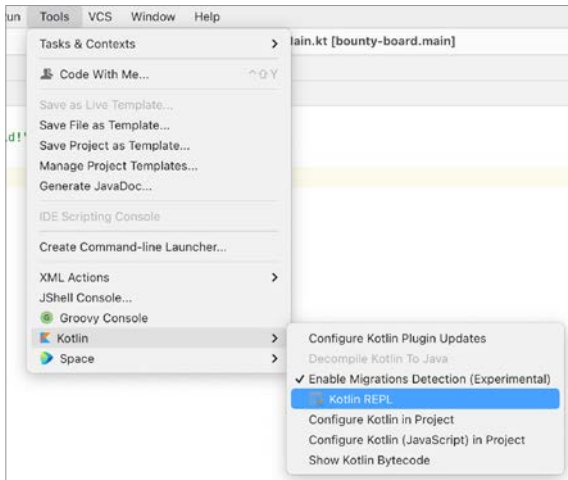


Рис. 1.14. Открытие инструментального окна Kotlin REPL

IntelliJ отображает панель REPL в нижней части окна (рис. 1.15).

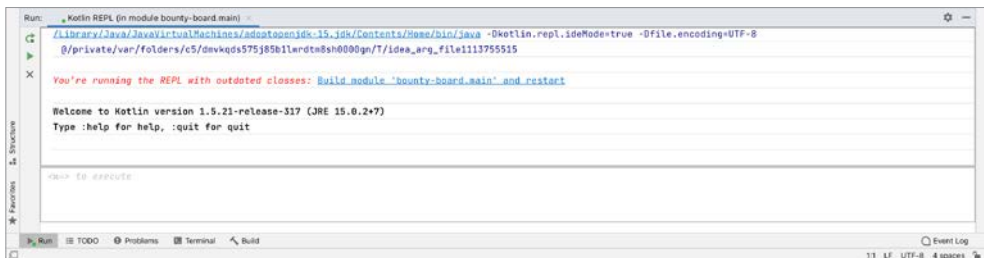


Рис. 1.15. Инструментальное окно Kotlin REPL

Код в REPL можно вводить так же, как в редакторе. Разница в том, что вы быстро получите результат без компиляции всего проекта.

(При запуске REPL могут появиться предупреждения красного цвета "running the REPL with outdated classes" — «запуск REPL с устаревшими классами». Обычно на такие предупреждения можно не обращать внимания. Общие проблемы платформы или JVM не нанесут вреда при использовании REPL, а поскольку в REPL вы не будете обращаться к коду из bounty-board, предупреждения об устаревших классах можно игнорировать.)

Введите следующий код в REPL.

Листинг 1.2. "Hello, Kotlin!" (REPL)

```
println("Hello, Kotlin!")
```

После ввода текста нажмите Command-Return (Ctrl-Enter), чтобы выполнить код в REPL. Через мгновение вы увидите результат под введенной строкой — это должен быть текст *Hello, Kotlin!* (рис. 1.16).

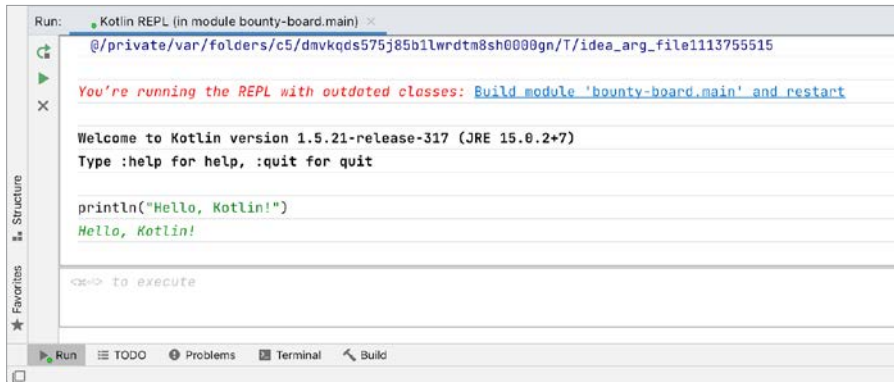


Рис. 1.16. Выполнение кода

REPL — это сокращение от Read, Evaluate, Print, Loop (прочитать, выполнить, вывести, повторить). Вы вводите фрагмент кода после подсказки и отправляете его в обработку, нажав на зеленую кнопку запуска слева в окне REPL или Command-Return (Ctrl-Enter). Далее REPL *читает* код, *выполняет* его и *выводит* результат. Завершив выполнение, REPL возвращает управление пользователю и дает возможность *повторить* процесс.

Ваше путешествие по миру Kotlin началось! Вы проделали важную работу в этой главе, заложив фундамент для дальнейшего изучения программирования на Kotlin. В следующей главе мы начнем знакомить вас с деталями языка: вы узнаете, как использовать переменные, константы и типы для представления данных.

Для любознательных: зачем использовать IntelliJ

Для написания кода на языке Kotlin подойдет любой текстовый редактор. Однако мы рекомендуем использовать IntelliJ, особенно пока вы учитесь. Подобно тому как функции проверки грамматики и правописания в текстовом редакторе помогают грамотно написать любую статью, IntelliJ предлагает инструменты, позволяющие писать хорошо структурированный код на Kotlin. Итак, IntelliJ поможет вам:

- писать синтаксически и семантически правильный код с помощью таких функций, как подсветка синтаксиса, контекстные подсказки, автодополнение;
- выполнять код и производить его отладку с помощью таких функций, как точка останова и пошаговое выполнение программы;
- изменять структуру существующего кода с помощью методов рефакторинга (таких, как переименование или извлечение констант) и форматирования кода для правильной расстановки отступов.

Кроме того, так как и язык Kotlin, и среда разрабатывались командой JetBrains, интеграция Kotlin с IntelliJ была тщательно проработана, что делает работу легкой и приятной. Также стоит упомянуть, что среда IntelliJ была заложена в основу Android Studio, поэтому горячие клавиши и средства, описанные в этой книге, вы сможете использовать и там (если потребуется, конечно).

Для любознательных: программирование для JVM

JVM — это программа, которая умеет выполнять набор инструкций, называемых байт-кодом.

Программирование для JVM означает, что ваш исходный код на Kotlin будет компилироваться, или транслироваться, в байт-код Java для последующего выполнения под управлением JVM (рис. 1.17).

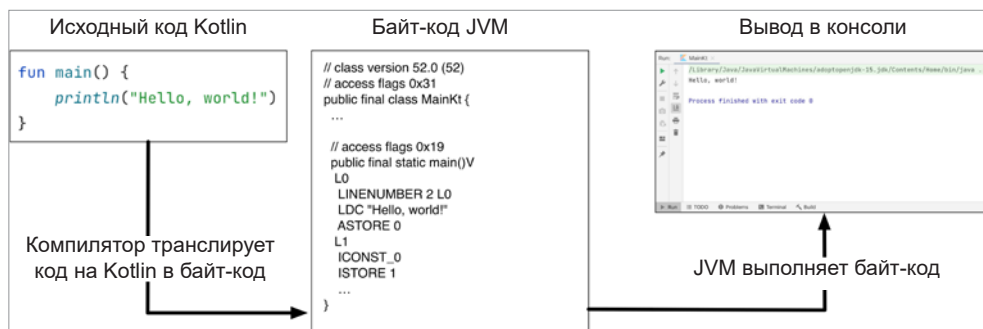


Рис. 1.17. Процесс компиляции и выполнения кода

Каждая платформа (например, Windows или macOS) имеет свой набор инструкций. Виртуальная машина JVM связывает байт-код с различными программными и аппаратными средствами, на которых работает JVM; она читает байт-код и выполняет соответствующие ему машинные инструкции. Это позволяет разработчикам на языке Kotlin только один раз написать платформенно независимый код, который после компиляции в байт-код будет выполняться на разных устройствах вне зависимости от операционной системы.

Так как Kotlin может транслироваться в байт-код для JVM, он считается JVM-языком. Вероятно, Java стал самым известным JVM-языком, потому что он был первым. Впоследствии появились другие JVM-языки — Scala, Groovy и Kotlin, которые были призваны устранить некоторые недостатки Java с точки зрения разработчиков.

Задание: арифметические вычисления в REPL

Большинство глав в этой книге заканчиваются одним или несколькими заданиями. Они позволят вам поработать самостоятельно и получить дополнительный опыт.

С помощью REPL изучите, как работают арифметические *операторы* в Kotlin: +, -, *, /, %. Например, введите `(9+12)*2` в REPL. Соответствует ли вывод вашим ожиданиям?

Если хотите узнать больше, просмотрите список математических функций, поддерживаемых стандартной библиотекой Kotlin, по адресу kotlinlang.org/api/latest/jvm/stdlib/kotlin.math/ и поэкспериментируйте с ними в REPL. Например, `min(94, -99)` выведет наименьшее из двух чисел, указанных в скобках.

На этой стадии изучения языка вам стоит ознакомиться с веб-сайтом kotlinlang.org и особенно с размещенной на нем документацией. Если вы планируете использовать Kotlin вне JVM, обратите внимание на цветные метки рядом со ссылками на API. Они обозначают, какие платформы поддерживает данный API. Многие API, например `absoluteValue` (рис. 1.18), работают на всех платформах, поддерживаемых Kotlin.

CommonJVMJSNative

Version1.5

[kotlin-stdlib](#) / [kotlin.math](#)

Package kotlin.math

Mathematical functions and constants.

The functions include trigonometric, hyperbolic, exponentiation and power, logarithmic, rounding, sign and absolute value.

Properties

CommonJVMJSNative

1.2

absoluteValue

Returns the absolute value of this value.

```
val Double.absoluteValue: Double
val Float.absoluteValue: Float
val Int.absoluteValue: Int
val Long.absoluteValue: Long
```

1.2

E

Base of the natural logarithms, approximately 2.71828.

```
const val E: Double
```

Рис. 1.18. Поддержка платформ в документации API Kotlin

Подписывайся на самый топовый канал по Kotlin:
<https://t.me/KotlinSenior>

2. Переменные, константы и типы

В этой главе мы познакомим вас с переменными, константами и базовыми типами данных в Kotlin — фундаментальными элементами любой программы. *Переменные* и *константы* используются для хранения значений или передачи данных внутри приложения. *Типы* описывают конкретные данные, хранимые переменной или константой.

Есть важные различия между типами данных, а также между переменными и константами, которые и определяют порядок их использования.

Типы

Данные, хранимые в переменных и константах, относятся к определенному типу. Тип описывает данные, хранящиеся в константе или переменной, и указывает, как при компиляции будет происходить *проверка типа*. Такая проверка предотвращает присваивание переменной или константе данных неправильного типа.

Чтобы увидеть, как работает проверка типа, отредактируйте файл `Main.kt` из проекта `bounty-board`, созданный в главе 1. Если вы закрыли среду IntelliJ, запустите ее заново. Скорее всего, проект `bounty-board` откроется автоматически, так как IntelliJ при запуске открывает последний проект, с которым вы работали. Если этого не произошло, выберите `bounty-board` в списке недавних проектов в середине окна приветствия или при помощи команды `File ▶ Open Recent ▶ bounty-board`.

Объявление переменной

Представьте, что вы пишете игру-приключение, в которой игроку каждый раз предлагается выполнить некую миссию. Сложность миссии возрастает по мере того, как игрок становится сильнее и поднимается на следующий уровень в игре. Вероятно, вам понадобится переменная для хранения текущего уровня игрока.

В файле `Main.kt` создайте первую переменную с именем `playerLevel` и присвойте ей значение.

Листинг 2.1. Объявление переменной `playerLevel` (`Main.kt`)

```
fun main() {  
    println("Hello, world!")  
    var playerLevel: Int = 4  
}
```

```
println(playerLevel)
}
```

Здесь экземпляр типа `Int` присваивается переменной с именем `playerLevel`. Давайте подробнее рассмотрим, что у нас получилось.

Для определения переменной используется ключевое слово `var`, которое объявляет новую переменную. После ключевого слова мы задаем имя переменной.

Затем указывается тип переменной `Int`. Это означает, что в `playerLevel` будет храниться целое число.

И наконец, мы используем *оператор присваивания* (`=`), чтобы присвоить значение в правой части (экземпляр типа `Int`, а именно `4`) переменной в левой части (`playerLevel`).

На рис. 2.1 показано объявление `playerLevel` в виде диаграммы.

После объявления значение переменной можно вывести в консоль с помощью функции `println`.

Запустите программу, щелкнув на кнопке запуска рядом с функцией `main` и выбрав `Run 'Main.kt'`. Также для этого можно воспользоваться кнопкой запуска на панели инструментов IntelliJ.

В консоли отобразится число `4` — то значение, которое было присвоено `playerLevel`.

Теперь попробуйте присвоить `playerLevel` значение `"thirty-two"`. (Зачеркнутая строка означает, что код надо удалить.)



Рис. 2.1. Порядок объявления переменной

Листинг 2.2. Присваивание "thirty-two" переменной `playerLevel` (`Main.kt`)

```
fun main() {
    println("Hello, world!")
    var playerLevel: Int = 4
    var playerLevel: Int = "thirty-two"
    println(playerLevel)
}
```

Снова запустите `main` при помощи кнопки запуска. На этот раз компилятор Kotlin сообщит об ошибке:

```
e: Main.kt: (3, 28): Type mismatch: inferred type is String but Int was
expected
```

Набирая код, вы могли заметить, что `"thirty-two"` подчеркнуто красным. Так IntelliJ показывает вам обнаруженную ошибку. Наведите указатель мыши на `"thirty-two"`, отобразится описание проблемы (рис. 2.2).



Рис. 2.2. Подсказка: обнаружено несоответствие типа

Kotlin использует *статическую типизацию*, то есть компилятор проверяет все типы в исходном коде, чтобы убедиться, что написанный код корректен. Также статическая типизация означает, что после определения переменной вы не сможете изменить тот тип, с которым она была объявлена.

IntelliJ проверяет код в процессе набора и обнаруживает ошибки, связанные с попытками присвоить переменной значение неверного типа. Такая возможность, называемая *статической проверкой согласованности типов*, помогает выявлять ошибки программирования еще до компиляции кода.

Чтобы устранить ошибку, надо присвоить переменной `playerLevel` другое значение типа `Int` — например, заменить `"thirty-two"` целым числом 4.

Листинг 2.3. Исправление ошибки типа (Main.kt)

```
fun main() {
    println("Hello, world!")
    var playerLevel: Int = "thirty-two"
    var playerLevel: Int = 4
    println(playerLevel)
}
```

Если вы соблюдаете правила назначения типов, в процессе выполнения переменной можно присвоить другое значение. Например, при повышении уровня игрока переменной `playerLevel` может быть присвоено новое значение. Например, `playerLevel` можно присвоить значение 5.

Листинг 2.4. Переменной `playerLevel` присваивается 5 (Main.kt)

```
fun main() {
    println("Hello, world!")
    var playerLevel: Int = 4
    println(playerLevel)

    println("The hero embarks on her1 journey to locate the enchanted sword.")
}
```

¹ В примерах кода авторы используют местоимения женского рода *she*, *her* для героя игры (возможно, из соображений политкорректности). На русском языке «герой» и «игрок»


```
    playerLevel = 5
    println(playerLevel)
}
```

Снова выполните обновленную функцию `main`, чтобы увидеть, как работает присваивание. В консоли выводится число 5 в отдельной строке после сообщения `The hero embarks on her journey to locate the enchanted sword` (Герой отправляется в путешествие, чтобы найти заколдованный меч).

Оператор присваивания (`=`) связывает переменную с новым значением. Такое решение работает, но правильнее было бы увеличить `playerLevel` на 1, вместо того чтобы присваивать переменной значение 5. Увеличение уровня работает лучше, потому что ваш код станет более гибким — например, если вам понадобится изменить начальный уровень игрока.

Обновите операцию присваивания, чтобы значение переменной увеличивалось на 1. Заодно обновите приветственное сообщение, которое сейчас кажется немного неуместным.

Листинг 2.5. Увеличение `playerLevel` (Main.kt)

```
fun main() {
    println("Hello, world!")
    println("The hero announces her presence to the world.")
    var playerLevel: Int = 4
    println(playerLevel)

    println("The hero embarks on her journey to locate the enchanted sword.")
    playerLevel = 5
    playerLevel += 1
    println(playerLevel)
}
```

После присваивания переменной `playerLevel` значения 4 используем *оператор сложения с присваиванием* (`+=`) для увеличения исходного значения на 1. Запустите программу снова. Вы увидите новое приветствие, а уровень игрока изменится с 4-го на 5-й, как и прежде.

Kotlin также предоставляет другие возможности присваивания значений. Так как `playerLevel` увеличивается на 1, вместо оператора `+=` можно использовать *оператор инкремента* (`++`):

```
playerLevel++
```

Для уменьшения значения на 1 можно использовать оператор декремента (`--`), а для уменьшения на произвольную величину — оператор вычитания с присваиванием (`-=`). Также существуют аналогичные операторы для умножения с при-

звучит более привычно, чем «героиня» или «игрокиня», поэтому в русской версии герой Мадригал будет мужского рода. — *Примеч. ред.*

сваиванием (`*`) и деления с присваиванием (`/=`). Математические операторы мы рассмотрим более подробно в главе 5.

Встроенные типы языка Kotlin

Вы уже видели переменные типа `Int`, а также пользовались типом `String` при вызове функции `println`. Kotlin также поддерживает типы для работы со значениями «истина/ложь», списками и парами «ключ — значение». В табл. 2.1 перечислены наиболее часто используемые типы, доступные в Kotlin.

Таблица 2.1. Наиболее часто применяемые встроенные типы

Тип	Описание	Примеры
<code>String</code> (строка)	Текстовая информация	"Madrigal" "happy meal"
<code>Char</code> (символ)	Один символ	'X' Символ Юникод U+0041
<code>Boolean</code> (логический)	Истина/ложь Да/Нет	true false
<code>Int</code> (целочисленный)	Целое число	5 "Madrigal".length
<code>Double</code> (с плавающей запятой)	Дробные числа	3.14 2.718
<code>List</code> (список)	Коллекция элементов	3, 1, 2, 4, 3 "root beer", "club soda", "coke"
<code>Set</code> (множество)	Коллекция уникальных значений	"Larry", "Moe", "Curly", "Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune"
<code>Map</code> (ассоциативный массив)	Коллекция пар «ключ — значение»	"small" to 5.99, "medium" to 7.99, "large" to 10.99

Если какие-то типы вам неизвестны, не переживайте — вы познакомитесь с ними в процессе чтения книги. В частности, строки рассматриваются в главе 6, числа — в главе 5, а списки, множества, ассоциативные массивы (относящиеся к категории *коллекций*) — в главах 9 и 10.

Переменные, доступные только для чтения

До настоящего времени вам попадались только переменные, которым можно присвоить новые значения; такие переменные называются *изменяемыми*. Но часто

возникает необходимость использовать переменные, значение которых должно оставаться постоянным все время выполнения программы. Например, в текстовой приключенческой игре имя игрока не должно меняться после начального присваивания.

Язык Kotlin позволяет объявлять переменные, *доступные только для чтения*, — значения таких переменных нельзя изменить после присваивания.

Переменная, которую можно изменить, объявляется с помощью ключевого слова `var`. Чтобы объявить переменную, доступную только для чтения, используется ключевое слово `val`.

Переменные, которые могут изменяться, мы будем называть `vars`, а переменные, доступные только для чтения, — `vals`.

Добавьте определение `val` для хранения имени игрока и выведите его после вывода начального уровня игрока.

Листинг 2.6. Добавление `val heroName` (Main.kt)

```
fun main() {
    println("The hero announces her presence to the world.")

    val heroName: String = "Madrigal"
    println(heroName)
    var playerLevel: Int = 4
    println(playerLevel)

    println("The hero embarks on her journey to locate the enchanted sword.")
    playerLevel += 1
    println(playerLevel)
}
```

Запустите программу при помощи кнопки запуска рядом с функцией `main` или в строке меню. Значения `playerLevel` и `heroName` должны появиться в консоли:

```
The hero announces her presence to the world.
Madrigal
4
The hero embarks on her journey to locate the enchanted sword.
5
```

Далее попытаемся присвоить `heroName` другое строковое значение оператором `=` и снова запустим программу.

Листинг 2.7. Попытка изменения значения `heroName` (Main.kt)

```
fun main() {
    println("The hero announces her presence to the world.")

    val heroName: String = "Madrigal"
    println(heroName)
```

```
var playerLevel: Int = 4
println(playerLevel)

heroName = "Estragon"

println("The hero embarks on her journey to locate the enchanted sword.")
playerLevel += 1
println(playerLevel)
}
```

При попытке запуска в консоли появится следующее сообщение об ошибке компиляции:

```
e: Main.kt: (9, 5): Val cannot be reassigned
```

Компилятор сообщил о попытке изменить `val`. После начального присваивания значение `val` нельзя изменить.

Удалите вторую операцию присваивания, чтобы исправить ошибку повторного присваивания.

Листинг 2.8. Исправление ошибки повторного присваивания значения `val` (Main.kt)

```
fun main() {
    println("The hero announces her presence to the world.")

    val heroName: String = "Madrigan"
    println(heroName)
    var playerLevel: Int = 4
    println(playerLevel)

    heroName = "Estragon"

    println("The hero embarks on her journey to locate the enchanted sword.")
    playerLevel += 1
    println(playerLevel)
}
```

`vals` полезны для защиты от случайного изменения значений переменных, которые предназначены только для чтения. По этой причине мы рекомендуем использовать `val` везде, где не требуется `var`.

Среда IntelliJ способна определить на основании статического анализа кода, когда `var` можно превратить в `val`. Если `var` не изменяется по ходу программы, IntelliJ предложит преобразовать его в `val`. Мы советуем следовать рекомендациям IntelliJ, если, конечно, вы не планируете писать код для изменения значений `var`. Чтобы увидеть, как выглядит рекомендация от IntelliJ, преобразуйте `heroName` в `var`.

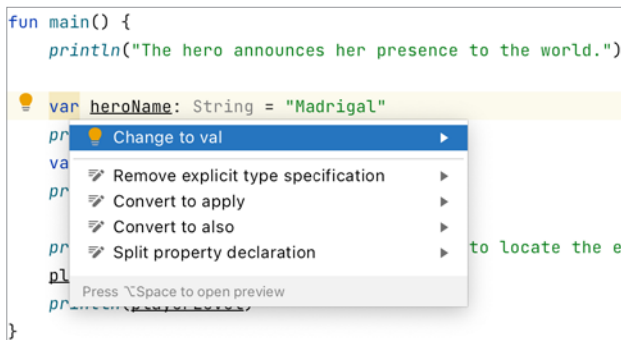
Листинг 2.9. Замена heroName на var (Main.kt)

```
fun main() {  
    println("The hero announces her presence to the world.")  
  
    val heroName: String = "Madrigal"  
    var heroName: String = "Madrigal"  
    println(heroName)  
    var playerLevel: Int = 4  
    println(playerLevel)  
  
    println("The hero embarks on her journey to locate the enchanted sword.")  
    playerLevel += 1  
    println(playerLevel)  
}
```

Так как значение heroName нигде не изменяется, нет необходимости (и не следует) объявлять его как var. Обратите внимание, что среда IntelliJ выделила строку с ключевым словом var горчичным цветом. Если навести указатель мыши на ключевое слово var, IntelliJ сообщит о предлагаемом изменении (рис. 2.3).

**Рис. 2.3.** Подсказка: переменная нигде не изменяется

Как и ожидалось, IntelliJ предлагает преобразовать heroName в val. Чтобы подтвердить изменение, щелкните на ключевом слове var рядом с heroName и нажмите Option-Return (Alt-Enter). В появившемся меню выберите Change to val (рис. 2.4).

**Рис. 2.4.** Переменная становится неизменяемой

IntelliJ автоматически заменит `var` на `val`:

```
val heroName: String = "Madrigal"
println(heroName)
```

Как мы уже говорили, рекомендуется использовать `val` всегда, когда это возможно, чтобы компилятор Kotlin мог предупредить о случайных попытках присвоить ей другое значение. Также советуем не игнорировать предложения IntelliJ касательно возможного улучшения кода. Им можно и не следовать, но обратить внимание определенно стоит.

Автоматическое определение типов

Обратите внимание, что типы, которые вы указали для переменных `heroName` и `playerLevel`, выделены серым цветом в IntelliJ. Серый цвет показывает элементы, которые являются необязательными или не используются. Наведите указатель мыши на определение типа `String`, и IntelliJ объяснит, почему эти элементы необязательны (рис. 2.5).

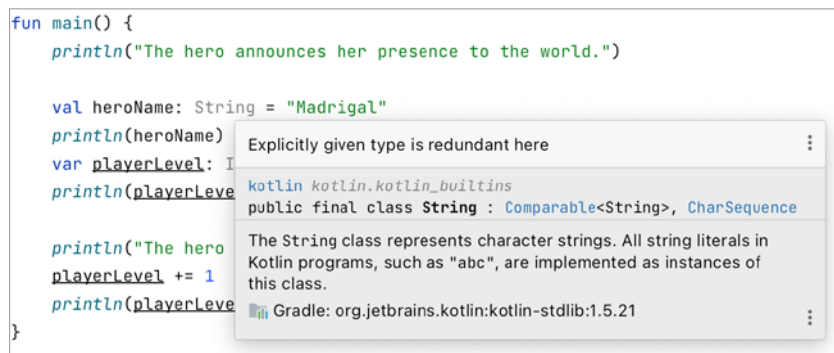


Рис. 2.5. Избыточная информация о типе

Как видите, Kotlin считает, что ваше объявление типа избыточно. Что это значит?

Kotlin поддерживает механизм *автоматического определения* типов (type inference), что позволяет опустить типы для переменных, которым присваиваются значения при объявлении. Так как при объявлении переменной `heroName` присваивается значение типа `String` и переменной `playerLevel` присваивается значение типа `Int`, компилятор Kotlin автоматически определяет тип каждой переменной.

Подобно тому как среда IntelliJ помогает поменять `var` на `val`, она может помочь убрать ненужное объявление типа. Щелкните на объявлении типа `String` (`: String`) рядом с `heroName` и нажмите Option-Return (Alt-Enter). Затем щелкните на строке `Remove explicit type specification` в появившемся меню (рис. 2.6).

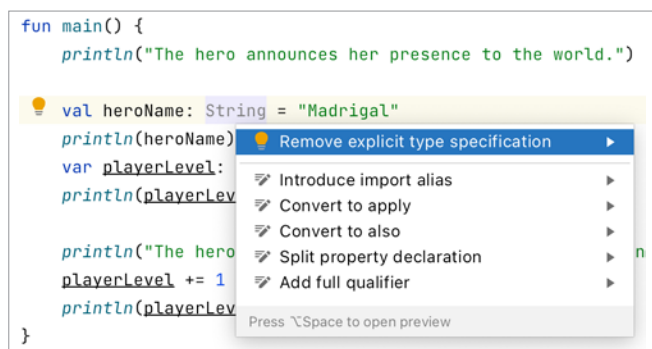


Рис. 2.6. Удаление явного определения типа

Объявление типа `: String` исчезнет. Повторите процесс для `playerLevel` `var`, чтобы убрать `: Int`.

Вне зависимости от того, используете вы автоматическое определение типов или указываете тип в объявлении каждой переменной, компилятор отслеживает тип. В этой книге мы используем автоматическое определение типов, если это не создает двусмысленности. Такой прием делает код более чистым и компактным и упрощает его изменение в будущем.

Обратите внимание, что IntelliJ покажет тип любой переменной по вашему запросу, даже если ее тип не был объявлен явно. Чтобы узнать тип переменной, щелкните на ее имени или выделите часть кода и выполните команду `View ▶ Type Info` (или нажмите `Control-Shift-P` [`Ctrl-Shift-P`]). Результат показан на рис. 2.7.

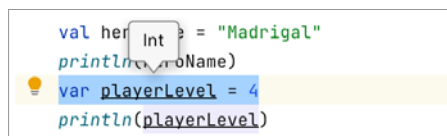


Рис. 2.7. Вывод информации о типе

Константы времени компиляции

Ранее мы рассказали о том, что `vars` могут менять свои значения, а `vals` нет. Мы немного приврали, но... из благих побуждений. На самом деле иногда `val` может возвращать разные значения, и мы обсудим это в главе 13. Если есть значения, которые вы не хотите менять никогда и ни при каких условиях, стоит использовать *константы времени компиляции*.

Константа времени компиляции объявляется вне какой-либо функции, даже не в пределах функции `main`, потому что ее значение присваивается *во время компиляции* (в момент, когда программа компилируется), — отсюда и такое название. Функция `main` и другие вызываются *во время выполнения* (когда программа за-

пущена), и переменные внутри функций получают свои значения в этот период. Константа времени компиляции к тому моменту уже существует.

Константы времени компиляции могут иметь значения только одного из следующих базовых типов (использование более сложных типов может поставить под угрозу гарантию времени компиляции). Вы узнаете больше о конструировании типов в главе 14. Итак, вот допустимые базовые типы для констант времени компиляции:

● String	● Float	● Byte
● Int	● Long	● Char
● Double	● Short	● Boolean

Имя героя игры (**Madrigal**) никогда не изменится — это главный персонаж, и у игрока нет возможности его сменить. Чтобы отразить эту неизменность в коде, можно задать значение имени константой. В файле **Main.kt** переместите переменную **heroName** над объявлением функции **main** и добавьте модификатор **const**.

Листинг 2.10. Объявление константы времени компиляции (Main.kt)

```
const val heroName = "Madrigal"

fun main() {
    println("The hero announces her presence to the world.")

    val heroName = "Madrigal"
    println(heroName)
    var playerLevel: Int = 4
    println(playerLevel)

    println("The hero embarks on her journey to locate the enchanted sword.")
    playerLevel += 1
    println(playerLevel)
}
```

Модификатор **const**, предшествующий **val**, предупреждает компилятор, что значение **val** нигде не должно изменяться. В данном случае имя героя всегда будет представлено значением **"Madrigal"**, что бы ни произошло. Это дает возможность компилятору применить дополнительные оптимизации. Снова выполните этот код и убедитесь, что результат остался прежним после внесения изменений.

В Kotlin принято использовать верблюжий регистр (**camelCase**) для имен переменных и ЗМЕИНЫЙ_РЕГИСТР (**SNAKE_CASE**) для имен констант. Таким образом, вместо того чтобы присваивать новой константе имя **heroName**, правильнее было бы назвать ее **HERO_NAME**.

Так как наш проект невелик, вы можете легко изменить имя и обновить всего одно упоминание в функции **main**. Но в более крупных проектах эта задача заметно

усложняется. К счастью, IntelliJ включает средства рефакторинга, которые помогают вносить изменения на уровне всего кода.

Щелкните правой кнопкой мыши на константе `heroName`, затем выберите команду **Refactor** ▶ **Rename...** (рис. 2.8).

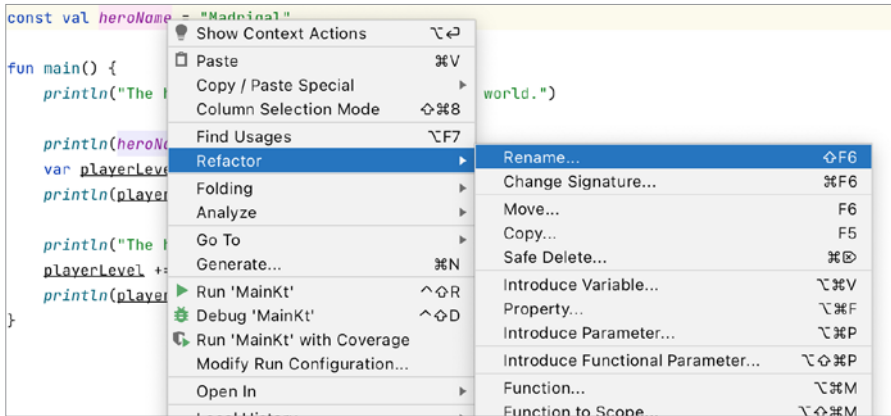


Рис. 2.8. Команда меню **Refactor** ▶ **Rename...**

Имя константы выделяется цветом. Введите `HERO_NAME`, заменяя выделенный текст. В процессе ввода IntelliJ продолжает выделять константу, как показано на рис. 2.9.

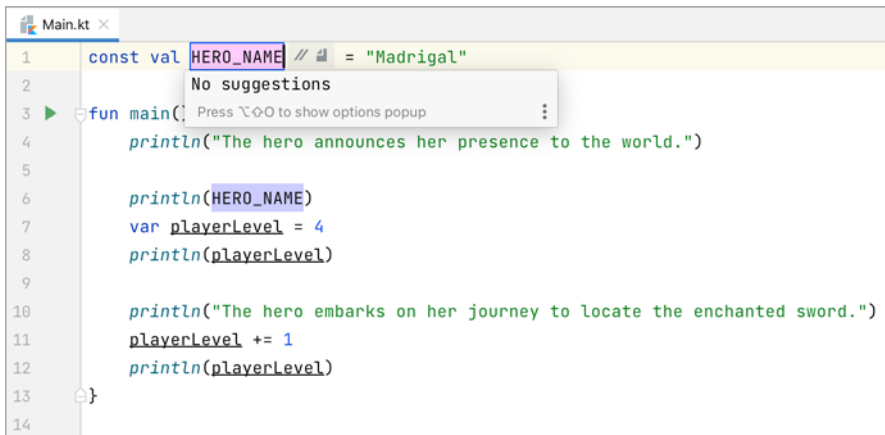


Рис. 2.9. Переименование константы

В процессе ввода обратите внимание, что при вводе нового имени строка `println(heroName)` автоматически обновляется. IntelliJ находит все случаи

употребления константы в вашем проекте и обновляет их, задавая новое имя. В проекте `bounty-board` замена затронет только одну строку кода, но в большом проекте IntelliJ может автоматически обновить множество файлов.

Когда вы завершите ввод нового имени, нажмите `Return`, чтобы подтвердить изменения.

Изучаем байт-код Kotlin

Из главы 1 вы узнали, что на Kotlin можно писать программы для виртуальной машины JVM, которая исполняет байт-код Java. Зачастую бывает полезно взглянуть на байт-код Java, который генерируется компилятором языка Kotlin и запускается под управлением JVM. Кое-где в этой книге мы будем рассматривать байт-код, чтобы понять, как конкретные особенности языка работают в JVM.

Умение анализировать Java-эквиваленты кода на Kotlin поможет вам понять, как работает Kotlin, особенно если у вас есть опыт работы с Java. Если опыта именно с Java у вас нет, вы все равно сможете увидеть в Kotlin знакомые черты языка, с которым вам приходилось работать, поэтому рассматривайте байт-код как своего рода псевдокод, упрощающий понимание. Ну а если вы новичок в программировании — поздравляем! Kotlin позволит вам выразить ту же логику программы, что и в Java, но гораздо короче.

Допустим, вам хочется узнать, как автоматическое определение типов переменных в Kotlin влияет на байт-код, сгенерированный для выполнения в JVM. Для этого воспользуйтесь инструментальным окном байт-кода Kotlin.

В файле `Main.kt` дважды нажмите клавишу `Shift`, чтобы открыть диалоговое окно `Search Everywhere` (поиск везде). Начните вводить: «`show Kotlin bytecode`» (показать байт-код Kotlin) и выберите из списка доступных действий `Show Kotlin bytecode`, как только оно появится (рис. 2.10).

Откроется инструментальное окно байт-кода Kotlin (рис. 2.11). (Также можно открыть его с помощью меню `Tools ▸ Kotlin ▸ Show Kotlin Bytecode`.)

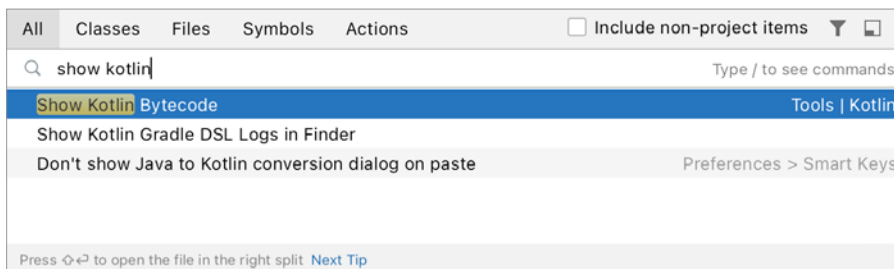


Рис. 2.10. Вывод байт-кода Kotlin

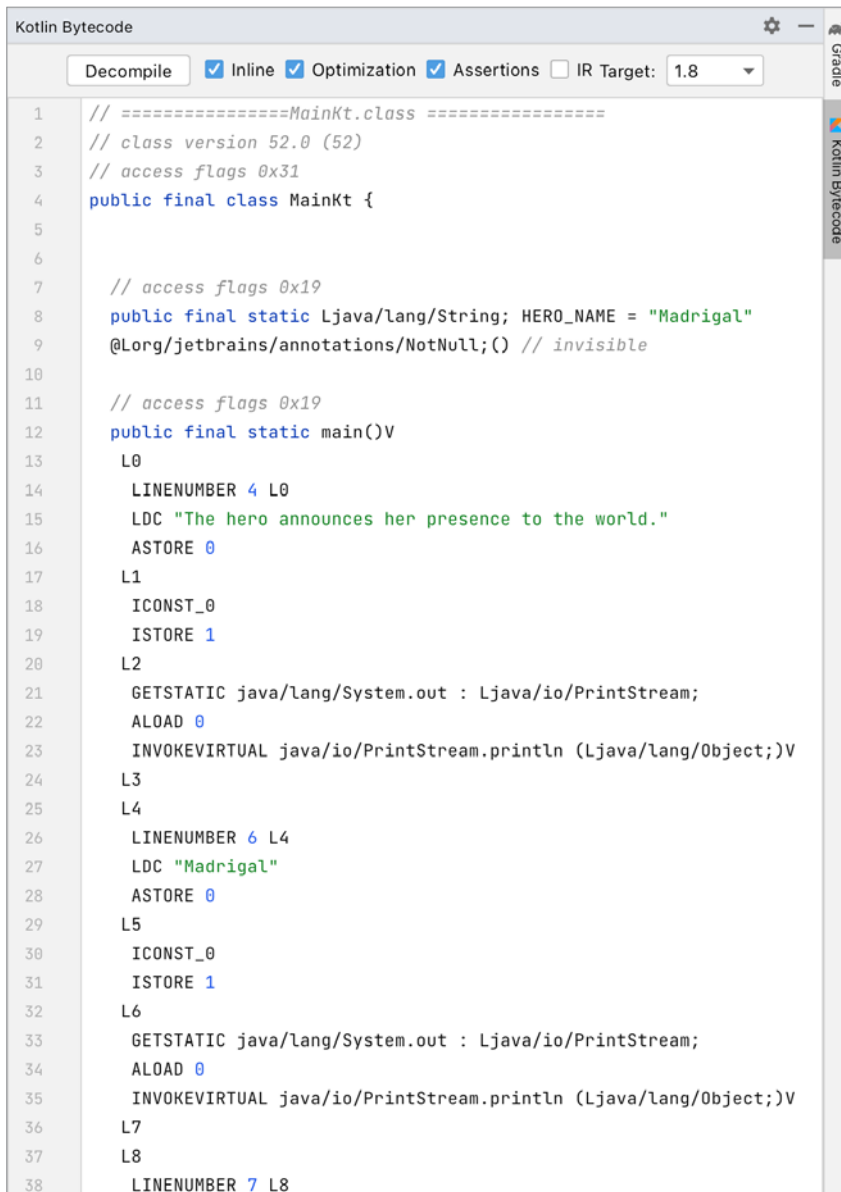


Рис. 2.11. Инструментальное окно байт-кода Kotlin

Если байт-код не ваш родной язык, не бойтесь! Переведите байт-код обратно в Java, чтобы увидеть его в более знакомом варианте. В окне байт-кода нажмите кнопку **Decompile** слева наверху.

Откроется новая вкладка `Main.decompiled.java` с Java-версией байт-кода, сгенерированного компилятором Kotlin для JVM (рис. 2.12).

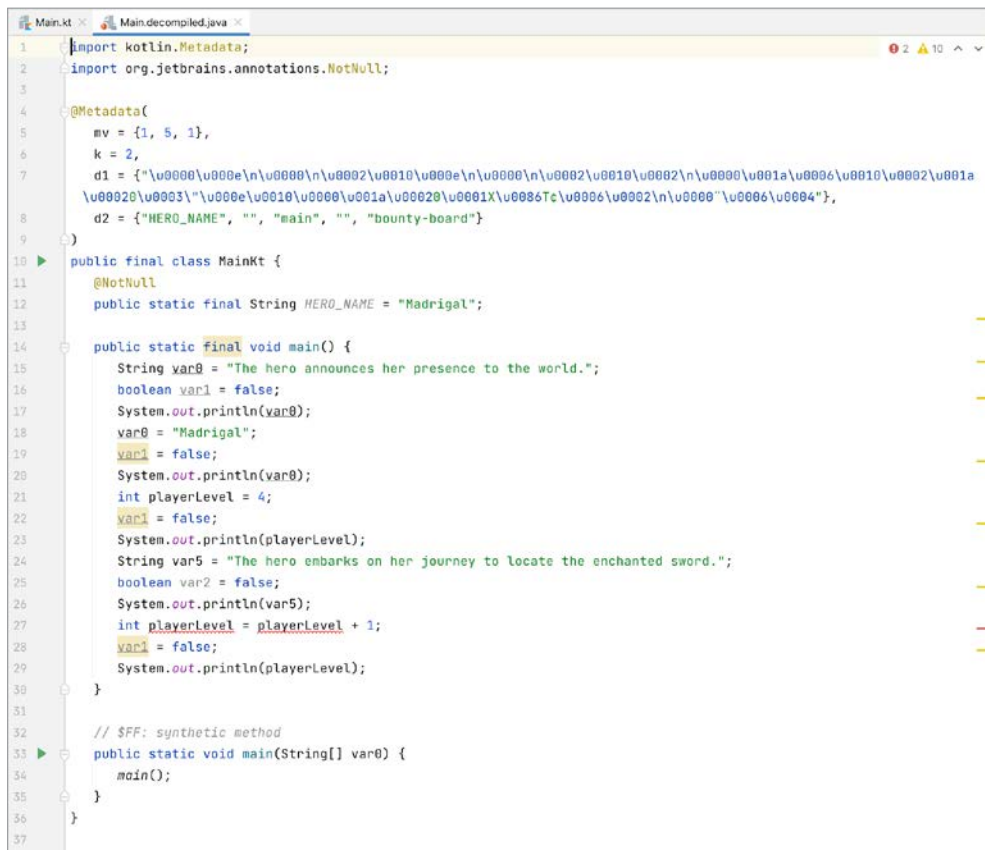


Рис. 2.12. Декомпилированный байт-код

(Иногда в декомпилированном коде появляются красные подчеркивания. Они указывают на странности взаимодействия между Kotlin и Java, а не сообщают об ошибке, — вы можете смело игнорировать ошибки и предупреждения, встречающиеся в сгенерированном байт-коде Java.)

Найдите объявление переменной `playerLevel`:

```
int playerLevel = 4;
```

Несмотря на то что вы опустили объявление типа в определениях обеих переменных в Kotlin, сгенерированный байт-код содержит явное объявление типа.

Именно так переменные были бы объявлены в Java, и именно так байт-код позволяет увидеть внутреннюю реализацию поддержки автоматического определения типов в языке Kotlin.

О декомпилированном байт-коде Java мы подробнее расскажем в следующих главах. А пока закройте `Main.decompiled.java` (нажав X на вкладке) и инструментальное окно байт-кода (используя значок _ в правом верхнем углу).

К сожалению, этот инструмент работает только с кодом Kotlin, предназначенным для JVM. Код Kotlin/JS транслируется в JavaScript, а код Kotlin/Native компилируется в низкоуровневый машинный код. Для этих платформ не генерируется байт-код, который можно было бы проанализировать. Декомпиляция чрезвычайно полезна при изучении Kotlin, хотя со временем, когда вы начнете более уверенно пользоваться языком, вы будете применять этот инструмент все реже.

В этой главе вы научились сохранять данные базовых типов в `vals` и `vars`, а также узнали, когда стоит выбирать тот или иной вариант типа в зависимости от необходимости менять его значения. Вы научились объявлять неизменяемые значения в виде констант времени компиляции. Наконец, узнали, как Kotlin использует автоматическое определение типов, чтобы не тратить время на лишний ввод при объявлении переменных. Вы еще неоднократно воспользуетесь этими базовыми инструментами по ходу работы над проектами.

В следующей главе вы узнаете, как выразить более сложные состояния с помощью условных конструкций.

Для любознательных: простые типы Java в Kotlin

В Java есть два вида типов: ссылочные и простые. Ссылочные определяются в исходном коде. Некоторые ссылочные типы также называются объектными, или упакованными (`boxed`), типами. Простые типы (часто их называют примитивами) не имеют определения в исходном коде и представлены ключевыми словами.

Имена ссылочных типов в Java всегда начинаются с прописной буквы; это признак того, что где-то в исходном коде находится определение этого типа. Вот как выглядит объявление `playerLevel` со ссылочным типом на языке Java:

```
Integer playerLevel = 5;
```

Имена примитивных типов в Java начинаются со строчной буквы:

```
int playerLevel = 5;
```

Для всех примитивов в Java имеется соответствующий ссылочный тип (но не все ссылочные типы имеют соответствующий простой тип). Зачем нужны эти две категории?

Часто ссылочные типы выбираются просто потому, что некоторые возможности языка Java доступны только при их применении. Например, механизм обобщения типов, с которым вы познакомитесь в главе 18, не работает с примитивами. Ссылочные типы также упрощают использование объектно-ориентированных возможностей Java. (Об объектно-ориентированном программировании и его особенностях в Kotlin разговор пойдет в главе 13.)

С другой стороны, примитивы обеспечивают лучшую производительность и имеют некоторые другие полезные особенности.

В отличие от Java, Kotlin поддерживает только один вид типов — ссылочный.

```
var playerLevel: Int = 4
```

На то есть несколько причин. Прежде всего, отсутствие выбора между разновидностями типа не позволит вам так же легко загнать себя в угол, как при наличии такого выбора. Например, вы объявили переменную простого типа, а потом оказалось, что механизм обобщения требует применения ссылочного типа. Поддержка в Kotlin только ссылочных типов навсегда избавляет вас от этой проблемы.

Если вы знакомы с Java, то, наверное, думаете сейчас: «Но примитивы работают производительнее ссылочных типов!» И это правда. Но давайте посмотрим, как выглядит переменная `playerLevel` в байт-коде, который вы видели раньше:

```
int playerLevel = 4;
```

Как видите, вместо ссылочного использован примитивный тип. Как такое возможно, если Kotlin поддерживает только ссылочные типы? Компилятор Kotlin, если есть такая возможность, использует примитивы байт-кода Java, потому что они действительно обеспечивают лучшую производительность.

Kotlin совмещает удобство ссылочных типов с быстродействием примитивов. В нем вы найдете соответствующий ссылочный тип для каждого из восьми примитивов, с которыми вы, возможно, уже знакомы по работе с Java.

Задание: `hasSteed`

В главе 1 мы предлагали вам поэкспериментировать с математическими операциями в Kotlin REPL. Многие задания в других главах базируются на проекте, над которым вы работаете, — в данном случае `bounty-board`. Прежде чем приступить к заданиям, создайте копию своего проекта. Проекты многих глав вы будете создавать на материале предыдущих глав, и конечно же, ошибки неизбежны. Поэтому не ленитесь и делайте копии своих проектов, прежде чем вносить в них изменения.

Вот ваше первое задание для `bounty-board`. В нашей текстовой приключенческой игре игрок может приручить дракона или минотавра, чтобы ездить на нем. Объявите переменную с именем `hasSteed` (стать хозяином боевого коня), чтобы

отслеживать, удалось ли игроку обзавестись средством передвижения. Задайте переменной начальное состояние, указывающее, что в данный момент средства передвижения нет.

Не забудьте создать копию `bounty-board`, прежде чем вносить эти изменения. В следующей главе мы продолжим работать над `bounty-board`, и в ней наличие переменной `hasSteed` не предполагается.

Задание: «Рог единорога»

Представьте следующую сцену из игры.

Герой игры прибыл в таверну «Рог единорога». Трактирщик спрашивает: «Вам нужна конюшня?»

«Нет, — отвечает тот, — у меня нет боевого коня. Но у меня есть пятьдесят монет, и я хочу выпить».

«Замечательно! — говорит трактирщик. — Могу предложить мед, вино и пиво. Что вы желаете?»

А теперь задание: добавьте после переменной `hasSteed` дополнительные переменные, необходимые для реализации сцены в таверне «Рог единорога», используя автоматическое определение типов и присваивая значения переменным при необходимости. Добавьте переменные для названия таверны, имени трактирщика и количества монет у игрока.

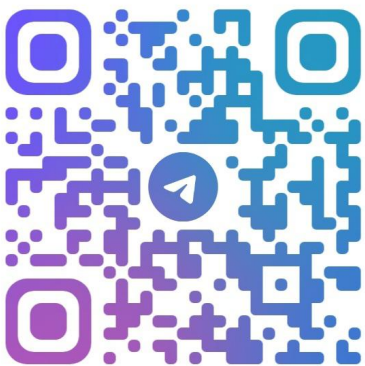
Обратите внимание: в таверне есть меню напитков, в котором можно сделать выбор. Каким типом можно воспользоваться для представления меню? За подсказкой обращайтесь к табл. 2.1.

Задание: волшебное зеркало

Отдохнув, герой готов отправляться на поиски приключений. А вы?

Герой обнаружил волшебное зеркало, которое показывает его `HERO_NAME` наоборот. Используя магию типа `String`, превратите строку `HERO_NAME "Madrigal"` в `"lagirdaM"`, зеркальное отражение значения этой переменной.

Чтобы решить эту задачу, посмотрите описание типа `String` по адресу kotlinlang.org/api/latest/jvm/stdlib/kotlin/-string/index.html. Там вы узнаете, что действия, которые поддерживает конкретный тип, обычно имеют очевидные названия (это подсказка).



@KOTLINSENIOR

Часть II

Базовый синтаксис

Если ограничиваться только переменными и командами вывода, много не напишешь. В следующих пяти главах мы познакомим вас с фундаментальными компонентами, встречающимися практически в каждом приложении Kotlin.

Вы расширите проект bounty-board посредством базовых языковых средств. Если у вас уже есть опыт программирования на других языках, то скорее всего, все эти средства уже знакомы вам. Впрочем, даже в этом случае вы узнаете много полезного о том, как они работают в Kotlin.

Подписывайся на самый топовый канал по Kotlin:
<https://t.me/KotlinSenior>

3. Условные конструкции

В этой главе вы научитесь определять правила, по которым выполняется код. Эта возможность языка, называемая *потоком выполнения* (control flow), или *программной логикой*, позволяет задавать условия, при которых должны выполняться те или иные части программы. Мы рассмотрим оператор и выражение `if/else`, а также выражение `when`. Вы научитесь проверять истинность/ложность, используя операторы сравнения и логики.

Чтобы опробовать все эти возможности на практике, вы продолжите разработку «доски поручений», на которой предлагаются миссии для героя в проекте bounty-board. Мы будем их усложнять по мере того, как герой становится сильнее, и в этом нам помогут условные конструкции.

Операторы `if/else`

В проекте bounty-board сила игрока определяется переменной `playerLevel`, созданной в предыдущей главе. Маленькие значения указывают на то, что уровень игрока невелик и он находится в начале своего путешествия, а большие — что уровень игрока повысился, а герой набрал опыт и стал сильнее.

Герою следует предлагать миссии, подходящие для его текущего уровня. Например, если игрок находится на уровне 1 (исходное значение для новых персонажей), ему должны предлагаться самые простые миссии.

Включите в функцию `main` первую конструкцию `if/else`, приведенную ниже. Мы проанализируем новый код после того, как вы введете его.

Листинг 3.1. Вывод миссии игрока (Main.kt)

```
const val HERO_NAME = "Madrigal"

fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME)
    var playerLevel = 4
    println(playerLevel)

    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else {
```

```
        println("Locate the enchanted sword.")
    }

    println("The hero embarks on her journey to locate the enchanted sword.")
    println("Time passes...")
    println("The hero returns from her quest.")

    playerLevel += 1
    println(playerLevel)
}
```

Разберем этот код от начала до конца.

Сначала в программу включаются операторы if/else. За ключевым словом if следует условие, заключенное в круглые скобки. Проверяемое условие выражает следующий вопрос: «Переменная playerLevel для игрока содержит 1?». Оно выражается оператором структурного равенства ==. Этот оператор можно читать как «равно», так что условие можно прочесть в виде «Если playerLevel равно 1».

За оператором if следует команда, заключенная в фигурные скобки ({}). Код в фигурных скобках определяет, что должна делать ваша программа, если в результате вычисления условия будет получен логический результат true — в данном случае если playerLevel содержит значение 1.

```
if (playerLevel == 1) {
    println("Meet Mr. Bubbles in the land of soft things.")
}
```

В эту команду включена знакомая функция println, которая используется для вывода информации в консоль. Короче говоря, если оператор if/else обнаруживает, что герой находится на уровне 1, программа должна вывести миссию для начального уровня.

(Хотя в данном случае в фигурных скобках содержится только одна команда, можно включить и другие действия, которые должны выполняться при истинности условия if.)

А если значение playerLevel не равно 1? В этом случае оператор if определит, что условие ложно, и ваша программа пропустит выражение в фигурных скобках после if и перейдет к else. Блок else можно рассматривать как «иначе»: если условие if истинно, то выполнится одно действие, иначе выполнится другое. За else — как и за if — следует заключенный в фигурные скобки набор инструкций, которые сообщают компилятору, что следует сделать. В отличие от if, часть else не нуждается в условии. Она просто применяется, если не выполнено условие if, поэтому сразу за ключевым словом следуют фигурные скобки:

```
else {
    println("Locate the enchanted sword.")
}
```

Блок `else` не является обязательным с точки зрения языка Kotlin. Как будет показано в последующих главах, программа может содержать оператор `if` без ветви `else`. В этом случае, если условие `if` дает результат `false`, программа выполнит команды, следующие за `if`. Также можно объявить блок `else` с пустым телом, результат будет тем же.

Две ветви кода различаются только разными миссиями в вызове `println`. Первая миссия не дает возможности игроку достичь более высокого уровня, а вторая — `else` — предлагает весьма серьезное поручение. (Пока большинство вызовов функций, которые вам встречались, использовались только для вывода строк в консоль. Вы больше узнаете о функциях, а также научитесь определять их самостоятельно в главе 4.)

Итак, проще говоря, ваш код сообщает компилятору: «Если герой находится на уровне 1, вывести в консоль сообщение `Meet Mr. Bubbles in the land of soft things`. Если герой не находится на уровне 1, вывести в консоль сообщение `Locate the enchanted sword`».

Оператор структурного равенства `==` относится к категории *операторов сравнения*. В табл. 3.1 перечислены операторы сравнения языка Kotlin. Сейчас вам необязательно изучать их, так как мы познакомимся с ними позже. Просто заглядывайте в эту таблицу, когда будете решать, каким оператором лучше выразить то или иное условие.

Таблица 3.1. Операторы сравнения

Оператор	Описание
<code><</code>	Проверяет, что значение слева меньше значения справа
<code><=</code>	Проверяет, что значение слева меньше или равно значению справа
<code>></code>	Проверяет, что значение слева больше значения справа
<code>>=</code>	Проверяет, что значение слева больше или равно значению справа
<code>==</code>	Проверяет, что значение слева равно значению справа
<code>!=</code>	Проверяет, что значение слева не равно значению справа
<code>===</code>	Проверяет, что две ссылки указывают на один экземпляр
<code>!==</code>	Проверяет, что две ссылки указывают на разные экземпляры

Вернемся к делу. Запустите `Main.kt`, нажав кнопку запуска слева от функции `main`. Вы увидите следующий вывод:

```

The hero announces her presence to the world.
Madrigal
4
Locate the enchanted sword.
Time passes...
The hero returns from her quest.
5

```

Так как заданное состояние `playerLevel == 1` ложно, ветвь `if` конструкции `if/else` была пропущена и вместо нее была выполнена ветвь `else`. (Мы использовали слово *ветвь*, потому что поток выполнения кода пойдет по тому пути, который соответствует условию.) Теперь попробуйте изменить значение `playerLevel` на 1.

Листинг 3.2. Изменение `playerLevel` (Main.kt)

```

const val HERO_NAME = "Madrigal"

fun main() {
    println("The hero announces her presence to the world.")
    println(HERO_NAME)

    var playerLevel = 4
    var playerLevel = 1
    println(playerLevel)

    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else {
        println("Locate the enchanted sword.")
    }

    println("Time passes...")
    println("The hero returns from her quest.")
    playerLevel += 1
    println(playerLevel)
}

```

Снова запустите программу. Результат выглядит так:

```

The hero announces her presence to the world.
Madrigal
1
Meet Mr. Bubbles in the land of soft things.
Time passes...
The hero returns from her quest.
2

```

Теперь определенное вами условие истинно (значение `playerLevel` равно 1), поэтому выполняется ветвь `if`.

Добавление условий

Наш код выбора миссии дает примерное представление о том, какая миссия может быть поручена герою, но код... сыроват. На начальном уровне он работает хорошо, но при любом уровне больше 1 существует только один вариант — поиск волшебного меча (*Locate the enchanted sword*). А если вы нашли волшебный меч, другой вам уже не понадобится.

Чтобы сделать выбор *if/else* более интересным, можно добавить больше условий для проверки и больше ветвей для всех возможных результатов. Для этого используем ветви *else if*, синтаксис которых точно такой же, как у ветви *if* (но находится между *if* и *else*). Добавьте в *if/else* четыре ветви *else if*, проверяющие промежуточные значения *playerLevel*. А заодно верните *playerLevel* значение 4.

Листинг 3.3. Проверка состояния игрока (Main.kt)

```
const val HERO_NAME = "Madrigal"

fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME)
var playerLevel = 1
    var playerLevel = 4
    println(playerLevel)

    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else if (playerLevel <= 5) {
        println("Save the town from the barbarian invasions.")
    } else if (playerLevel == 6) {
        println("Locate the enchanted sword.")
    } else if (playerLevel == 7) {
        println("Recover the long-lost artifact of creation.")
    } else if (playerLevel == 8) {
        println("Defeat Nogartse, bringer of death and eater of worlds.")
    } else {
        println("Locate the enchanted sword.")
        println("There are no quests right now.")
    }

    println("Time passes...")
    println("The hero returns from her quest.")

    playerLevel += 1
    println(playerLevel)
}
```

Новая логика выглядит так:

Если герой находится на уровне...	...вывести это сообщение.
1	Meet Mr. Bubbles in the land of soft things (познакомиться с Пузырем в стране мягких игрушек)
2–5	Save the town from the barbarian invasions (спасти город от вторжения варваров)
6	Locate the enchanted sword (найти заколдованный меч)
7	Recover the long-lost artifact of creation (вернуть утраченный артефакт творения)
8	Defeat Nogartse, bringer of death and eater of worlds (победить Nogartse, вестника смерти и пожирателя миров)
9+	There are no quests right now (больше нет миссий)

Снова запустите программу. Так как значение `playerLevel` равно 4, сначала при проверке `if` будет получен результат `false` и соответствующая ветвь выполнена не будет. Но условие `else if (playerLevel <= 5)` дает результат `true`, поэтому вы увидите в консоли сообщение `Save the town from the barbarian invasions`.

Компилятор проверяет условия `if/else` сверху вниз и прекращает проверку, как только будет найдено первое истинное условие. Если все условия окажутся ложными, то будет выполнена ветвь `else`.

Это означает, что порядок условий важен: если мы расположим условие `playerLevel <= 5` до проверки `playerLevel == 1`, то миссия для уровня 1 не будет выводиться никогда. (Не меняйте свой код, следующий фрагмент мы приводим просто для примера.)

```
if (playerLevel <= 5) {    // Срабатывает для любого значения 5 и менее
    println("Save the town from the barbarian invasions.")
} else if (playerLevel == 1) { // Срабатывает только для значения 1
    println("Meet Mr. Bubbles in the land of soft things.")
} else if (playerLevel == 6) {
    println("Locate the enchanted sword.")
} else if (playerLevel == 7) {
    println("Recover the long-lost artifact of creation.")
} else if (playerLevel == 8) {
    println("Defeat Nogartse, bringer of death and eater of worlds.")
} else {
    println("There are no quests right now.")
}
```

В этом примере для любых значений `playerLevel`, меньших либо равных 5, будет срабатывать первое условие, но вторая ветвь активизируется только для значения 1. Так как первое же условие `if` будет выполнено, до ветви `else if (playerLevel == 1)` дело не дойдет.

Вы добавили больше уровней для игрока, включив операторы `else if` с большим количеством условий, которые проверяются, когда исходное условие `if` оценивается как ложное. Попробуйте изменить значение `playerLevel`, чтобы активизировать результат каждой из определенных вами ветвей. Когда все будет сделано, верните `playerLevel` значение 4.

Вложенные операторы `if/else`

Одна из миссий на «доске поручений» — спасти город от вторжения варваров (Save the town from the barbarian invasions) — выглядит довольно абстрактно. Нужного результата можно достигнуть несколькими способами, включая дипломатию. Если игрок находится в хороших отношениях с вождем варварского племени, возможно, ему удастся дружески обсудить происходящее и прояснить все недоразумения.

Чтобы эта возможность стала более очевидной для игрока, название миссии должно меняться в зависимости от того, дружит ли игрок с варварами. Прежде чем обновлять логику определения миссии, необходимо добавить переменную для мониторинга дружеских отношений. (Как вы думаете, к какому типу должна относиться эта переменная?)

После определения переменной для мониторинга дружбы с варварами нужно снова обновить оператор `if/else`. Если уровень игрока лежит в диапазоне от 2 до 5, можно воспользоваться дополнительным вложенным оператором `if/else` для вывода правильного названия миссии. (При вводе представленных ниже изменений не пропустите добавленную скобку `}` перед `else if (playerLevel == 6)`.)

Листинг 3.4. Мониторинг дружеских отношений с варварами (Main.kt)

```
const val HERO_NAME = "Madrigal"

fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME)
    var playerLevel = 4
    println(playerLevel)

    val hasBefriendedBarbarians = true
    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else if (playerLevel <= 5) {
```



```
        if (hasBefriendedBarbarians) {
            println("Convince the barbarians to call off their invasion.")
        } else {
            println("Save the town from the barbarian invasions.")
        }
    } else if (playerLevel == 6) {
        println("Locate the enchanted sword.")
    } else if (playerLevel == 7) {
        println("Recover the long-lost artifact of creation.")
    } else if (playerLevel == 8) {
        println("Defeat Nogartse, bringer of death and eater of worlds.")
    } else {
        println("There are no quests right now.")
    }

    println("Time passes...")
    println("The hero returns from her quest.")

    playerLevel += 1
    println(playerLevel)
}
```

Мы добавили переменную `Boolean val`, которая показывает, находится ли игрок в дружеских отношениях с варварами, и оператор `if/else` для создания нового вывода, если игрок дружит с варварами, а его уровень — в диапазоне от 2 до 5. Вспомните, что `playerLevel` имеет значение 4, так что при запуске программы должно быть выведено новое сообщение. Запустите программу и убедитесь в этом. Результат должен выглядеть так:

```
The hero announces her presence to the world.
Madrigal
4
Convince the barbarians to call off their invasion.
Time passes...
The hero returns from her quest.
5
```

Если вы получили другой результат, убедитесь в том, что код точно совпадает с листингом 3.4, в частности, имеет ли `playerLevel` значение 4.

Вложенные операторы позволяют создавать дополнительные логические ветки внутри крупных ветвей, что помогает задавать более точные и сложные условия.

Более элегантные условные выражения

За условными выражениями нужно следить в оба, иначе они заполняют все вокруг, как саранча. К счастью, Kotlin позволяет пользоваться преимуществами условных выражений, которые при этом остаются компактными и удобочитаемыми. Рассмотрим несколько примеров.

Логические операторы

Не исключено, что в bounty-board вам придется проверять более сложные условия. Например, дипломатическое решение возможно, если игрок подружился с варварами *или* сам принадлежит к их племени; либо такое решение будет заблокировано, если он чем-то рассердил племя.

Чтобы определить, какую миссию следует выводить, можно воспользоваться набором операторов `if/else`, но это приведет к большому количеству повторяющегося кода, а логика условий станет малопонятной. Есть более элегантный и понятный способ: использовать логические операторы в условии.

Добавьте две новые переменные и обновите условие вложенного оператора `if` для изменения логики назначения миссии.

Листинг 3.5. Использование логических операторов в условии (Main.kt)

```
const val HERO_NAME = "Madrigal"

fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME)
    var playerLevel = 4
    println(playerLevel)

    val hasBefriendedBarbarians = true
    val hasAngeredBarbarians = false
    val playerClass = "paladin"
    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else if (playerLevel <= 5) {
        if (hasBefriendedBarbarians) {
        // Проверить возможность дипломатического решения
        if (!hasAngeredBarbarians &&
            (hasBefriendedBarbarians || playerClass == "barbarian")) {
            println("Convince the barbarians to call off their invasion.")
        } else {
            println("Save the town from the barbarian invasions.")
        }
    } else if (playerLevel == 6) {
        println("Locate the enchanted sword.")
    } else if (playerLevel == 7) {
        println("Recover the long-lost artifact of creation.")
    } else if (playerLevel == 8) {
        println("Defeat Nogartse, bringer of death and eater of worlds.")
    } else {
        println("There are no quests right now.")
    }

    println("Time passes...")
}
```

```
println("The hero returns from her quest.")

playerLevel += 1
println(playerLevel)
}
```

Вы добавили две переменные `val` — `hasAngeredBarbarians` (рассердил варваров) и `playerClass` — для мониторинга этих вариантов (переменные доступны только для чтения, так как их значения не должны изменяться в `bounty-board`). Эта часть вам знакома, но появилось кое-что новое. Прежде всего, в код был добавлен *комментарий*, начинающийся с пары символов `//`.

Все символы справа от `//` считаются комментарием и игнорируются компилятором, поэтому в комментариях можно писать все что угодно. Комментарии очень полезны для структурирования и добавления информации о коде, что делает его более читабельным для других пользователей (или для вас самих в будущем, так как вы можете забыть некоторые подробности кода).

Далее внутри `if` использованы два *логических оператора*. Логические операторы позволяют объединить операторы сравнения в одно длинное условие.

`!` — *оператор логического «не»* (`not`), который возвращает значение, обратное заданному логическому значению: если элемент, за которым он следует, является истинным, то выражение становится ложным, и наоборот. `&&` — *оператор логического «и»* (`and`) — требует, чтобы *оба* условия — слева и справа от него — были истинны, тогда все выражение будет истинно. `||` — *оператор логического «или»* (`or`). Все выражение с этим оператором будет считаться истинным, если хотя бы *одно* из условий (или оба) справа *или* слева от него истинно.

В табл. 3.2 перечислены логические операторы языка Kotlin.

Таблица 3.2. Логические операторы

Оператор	Описание
!	Логическое «не»: преобразует истину в ложь и ложь в истину
&&	Логическое «и»: истинно, когда оба выражения истинны (иначе ложно)
	Логическое «или»: истинно, когда хотя бы одно выражение истинно (ложно, если оба ложны)

Обратите внимание: все операторы имеют приоритет, который определяет очередность их вычисления при объединении в выражение. Операторы с одинаковым приоритетом применяются слева направо. Операторы можно группировать, заключая в круглые скобки. Далее операторы перечислены в порядке убывания приоритета — от высокого к низкому:

- `!` (логическое «не»)
- `<` (меньше чем), `<=` (меньше или равно), `>` (больше чем), `>=` (больше или равно)

- == (равно), != (не равно)
- && (логическое «и»)
- || (логическое «или»)

Вернемся к bounty-board и разберем новое условие:

```
if (!hasAngeredBarbarians &&
    (hasBefriendedBarbarians || playerClass == "barbarian")) {
    println("Convince the barbarians to call off their invasion.")
}
```

Иначе говоря, если герой *не* рассердил варваров и он или с ними подружился, *или* сам является варваром, игроку будет предложен дипломатический способ предотвращения войны.

Наш герой ничем не рассердил варваров. Сам он варваром не является, но находится с ними в дружеских отношениях. Таким образом, условие выполняется, и герой получает предложение вступить в переговоры. Запустите программу и убедитесь сами. Вывод должен выглядеть так:

```
The hero announces her presence to the world.
Madrigal
4
Convince the barbarians to call off their invasion.
Time passes...
The hero returns from her quest.
5
```

Только подумайте, сколько вложенных условных команд потребовалось бы для выражения этой логики без логических операторов. Эти операторы предоставляют средства для ясной формулировки сложной логики.

Применение логических операторов не ограничивается условными командами. Они также могут применяться в обычных выражениях и даже в объявлении переменной. Добавьте новую логическую переменную, которая инкапсулирует условие, необходимое для переговоров с варварами, и выполните *рефакторинг* (то есть перепишите без изменения поведения) условную инструкцию с использованием новой переменной.

Листинг 3.6. Использование логических операторов в объявлении переменной (Main.kt)

```
...
fun main() {
    ...
    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else if (playerLevel <= 5) {
        // Проверить возможность дипломатического решения
        val canTalkToBarbarians = !hasAngeredBarbarians &&
            (hasBefriendedBarbarians || playerClass == "barbarian")
    }
}
```

```
if (!hasAngeredBarbarians &&
    (hasBefriendedBarbarians || playerClass == "barbarian")) {
    if (canTalkToBarbarians) {
        println("Convince the barbarians to call off their invasion.")
    } else {
        println("Save the town from the barbarian invasions.")
    }
} else if (playerLevel == 6) {
    println("Locate the enchanted sword.")
} else if (playerLevel == 7) {
    println("Recover the long-lost artifact of creation.")
} else if (playerLevel == 8) {
    println("Defeat Nogartse, bringer of death and eater of worlds.")
} else {
    println("There are no quests right now.")
}
...
}
```

Проверка условия была вынесена в новое значение `val` с именем `canTalkToBarbarians` (может вести переговоры с варварами), и мы изменили конструкцию `if/else`, чтобы в ней проверялось это новое значение. Функционально наш код эквивалентен предыдущему, но теперь правило выражается через присваивание значения. Имя значения ясно отражает смысл правила в удобочитаемом виде: может ли игрок вступить в переговоры с варварами? Этот прием особенно полезен при усложнении правил — так вы можете донести новые правила для будущих читателей вашего кода.

Запустите программу и убедитесь, что она работает как прежде. Вывод должен быть таким же.

Условные выражения

Теперь конструкция `if/else` выводит правильное описание миссии — с некоторыми нюансами.

С другой стороны, код получился немного громоздким, потому что во всех ветвях используются похожие инструкции `println`. А теперь представьте, что потребовалось изменить общий формат того, как выводится описание миссии.

Придется пройти по каждой ветви `if/else` и изменить сообщение в каждом вызове функции `println`.

Проблему можно решить, заменив операторы `if/else` *условным выражением*. Условное выражение — это почти условный оператор, с той лишь разницей, что результат `if/else` присваивается переменной, которая будет использоваться в дальнейшем. Чтобы увидеть, как это работает, включите условное выражение в ветвь миссии с варварами.

Листинг 3.7. Использование условного выражения (Main.kt)

```

...
fun main() {
    ...
    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else if (playerLevel <= 5) {
        // Проверить возможность дипломатического решения
        val canTalkToBarbarians = !hasAngeredBarbarians &&
            (hasBefriendedBarbarians || playerClass == "barbarian")

        val barbarianQuest: String = if (canTalkToBarbarians) {
            println("Convince the barbarians to call off their invasion.")
            "Convince the barbarians to call off their invasion."
        } else {
            println("Save the town from the barbarian invasions.")
            "Save the town from the barbarian invasions."
        }
        println(barbarianQuest)
    } else if (playerLevel == 6) {
        println("Locate the enchanted sword.")
    } else if (playerLevel == 7) {
        println("Recover the long-lost artifact of creation.")
    } else if (playerLevel == 8) {
        println("Defeat Nogartse, bringer of death and eater of worlds.")
    } else {
        println("There are no quests right now.")
    }
    ...
}

```

Посредством выражения `if/else` новой переменной `barbarianQuest` (миссия с варварами) присваивается строковое значение из одной ветви `if` в зависимости от значения `canTalkToBarbarians`. В этом вся прелесть условного выражения. Так как теперь миссия сохраняется в переменной `canTalkToBarbarians`, можно использовать один вызов `println` для обоих случаев.

Чтобы дополнительно упростить логику, можно внести аналогичные изменения в сложную конструкцию `if/else`. Проведите рефакторинг логики миссий — вы увидите, что шесть практически одинаковых команд вывода исчезли из программы.

Листинг 3.8. Определение миссий посредством условного выражения (Main.kt)

```

...
fun main() {
    ...
    val hasBefriendedBarbarians = true
    val hasAngeredBarbarians = false
    val playerClass = "paladin"
    val quest: String = if (playerLevel == 1) {

```

```

println("Meet Mr. Bubbles in the land of soft things.")
"Meet Mr. Bubbles in the land of soft things."
} else if (playerLevel <= 5) {
    // Проверить возможность дипломатического решения
    val canTalkToBarbarians = !hasAngeredBarbarians &&
        (hasBefriendedBarbarians || playerClass == "barbarian")

    val barbarianQuest: String = if (canTalkToBarbarians) {
        "Convince the barbarians to call off their invasion."
    } else {
        "Save the town from the barbarian invasions."
    }
    println(barbarianQuest)
} else if (playerLevel == 6) {
    println("Locate the enchanted sword.")
    "Locate the enchanted sword."
} else if (playerLevel == 7) {
    println("Recover the long-lost artifact of creation.")
    "Recover the long-lost artifact of creation."
} else if (playerLevel == 8) {
    println("Defeat Nogartse, bringer of death and eater of worlds.")
    "Defeat Nogartse, bringer of death and eater of worlds."
} else {
    println("There are no quests right now.")
    "There are no quests right now."
}
println("The hero approaches the bounty board. It reads:")
println(quest)

println("Time passes...")
println("The hero returns from her quest.")

playerLevel += 1
println(playerLevel)
}

```

(Если вам надоело расставлять отступы в коде после каждого изменения, обратитесь к IntelliJ. Выберите команду **Code ▶ Auto-Indent Lines** и наслаждайтесь четкими отступами.)

Если значение переменной должно присваиваться в зависимости от условия, лучше всего использовать условное выражение. Однако следует помнить, что такие выражения выглядят проще, если все ветви возвращают значения одного типа (например, строки для `quest`).

Запустите код еще раз и убедитесь, что все работает как задумано. Вы увидите уже знакомый вывод (с небольшим добавлением), но при этом код стал элегантнее и понятнее.

```

The hero announces her presence to the world.
Madrigal

```

```
4
The hero approaches the bounty board. It reads:
Convince the barbarians to call off their invasion.
Time passes...
The hero returns from her quest.
5
```

Убираем скобки в выражениях if/else

В тех случаях, когда для подходящего условия требуется просто вернуть значение, допустимо (по крайней мере с точки зрения синтаксиса — подробнее см. дальше) снять фигурные скобки вокруг выражения. Фигурные скобки {} можно убрать, если ветвь содержит только одно выражение. Удаление скобок, окружающих ветвь с несколькими выражениями, повлияет на порядок выполнения кода, и Kotlin запрещает использовать if без команды или пары фигурных скобок.

Взгляните на версию quest без скобок:

```
val quest: String = if (playerLevel == 1)
    "Meet Mr. Bubbles in the land of soft things."
else if (playerLevel <= 5) {
    // Проверить возможность дипломатического решения
    val canTalkToBarbarians = !hasAngeredBarbarians &&
        (hasBefriendedBarbarians || playerClass == "barbarian")
    if (canTalkToBarbarians) "Convince the barbarians to call off their
invasion."
    else "Save the town from the barbarian invasions."
} else if (playerLevel == 6) "Locate the enchanted sword."
else if (playerLevel == 7) "Recover the long-lost artifact of creation."
else if (playerLevel == 8)
    "Defeat Nogartse, bringer of death and eater of worlds."
else "There are no quests right now."
```

Версия условного выражения quest делает то же самое, что и версия в вашем предыдущем коде. Она даже выражает ту же логику, но использует меньше кода. Какую версию вы находите более удобной для чтения и понимания? Вариант со скобками (как в вашем примере) — стиль, который считается предпочтительным в сообществе языка Kotlin.

Мы рекомендуем не пропускать скобки в условных операторах и выражениях, которые занимают более одной строки. Во-первых, без скобок при увеличении количества условий труднее понять, где заканчивается одна ветвь и начинается другая. Во-вторых, если убрать фигурные скобки, новый участник проекта может случайно обновить не ту ветвь или неправильно понять реализацию. Такие риски не стоят мизерной экономии времени от пары нажатий клавиш.

Кроме того, хотя в данном случае обе приведенные версии кода, со скобками и без, действуют совершенно одинаково, это не всегда бывает верно. Если ветвь

включает несколько выражений, а вы решите убрать фигурные скобки, тогда в этой ветви выполнится только первое выражение. Пример:

```
var arrowsInQuiver = 2
if (arrowsInQuiver >= 5) {
    println("Plenty of arrows")
    println("Cannot hold any more arrows")
}
```

Если у героя пять и более стрел, значит, его колчан полон, и в консоли появляется сообщение, что больше стрел он взять не может (cannot hold any more arrows). Если у героя две стрелы, то никакое сообщение в консоль не выводится. Однако без фигурных скобок логика меняется:

```
var arrowsInQuiver = 2
if (arrowsInQuiver >= 5)
    println("Plenty of arrows")
    println("Cannot hold any more arrows")
```

Без скобок вторая команда `println` более не считается частью ветви `if`. Текст "Plenty of arrows" (полно стрел) будет напечатан, если `arrowsInQuiver` больше или равно 5, но "Cannot hold any more arrows" будет выводиться всегда, независимо от количества стрел у героя.

Для выражений, уместающихся в одной строке, стоит следовать правилу, сформулированному в вопросе: «Какая запись выражения будет проще и понятнее для читателя?» Часто запись однострочных выражений без фигурных скобок упрощает чтение кода. Например, исключение фигурных скобок поможет прояснить логику простого однострочного условного выражения, как в следующем примере:

```
val healthSummary = if (healthPoints != 100) "Need healing!"
                    else "Looking good."
```

Кстати, если вы думаете: «Хорошо, но мне не нравится синтаксис `if/else`, даже с фигурными скобками, он такой *некрасивый*»... не беспокойтесь! Скоро мы перепишем выражение для определения миссии последний раз, используя более компактный — и ясный — синтаксис.

Интервалы

Все условия в выражении `if/else` для `quest`, по сути, проверяют целочисленное значение `playerLevel`. В некоторых используется оператор сравнения для проверки равенства `playerLevel` какому-то значению, в других используется несколько операторов сравнения, чтобы проверить, попадает ли значение `playerLevel` в интервал между двумя числами. Для второго случая есть альтернатива получше: в Kotlin поддерживаются *интервалы* (ranges) для представления линейного набора значений.

Интервал определяется оператором `..`. Он включает все значения, начиная с находящегося слева от оператора `..` и заканчивая тем, что расположен справа. Например, интервал `1..5` включает числа 1, 2, 3, 4 и 5. Интервалы также могут представлять последовательности символов.

Для проверки принадлежности заданного числа интервалу можно воспользоваться оператором `in`. Произведем рефакторинг выражения `quest`, чтобы вместо `<=` в нем использовались интервалы.

Листинг 3.9. Рефакторинг `quest` с использованием интервалов (Main.kt)

```
...
fun main() {
    ...
    val quest: String = if (playerLevel == 1) {
        "Meet Mr. Bubbles in the land of soft things."
    } else if (playerLevel <= 5) {
    } else if (playerLevel in 2..5) {
        // Проверить возможность дипломатического решения
        val canTalkToBarbarians = !hasAngeredBarbarians &&
            (hasBefriendedBarbarians || playerClass == "barbarian")
        if (canTalkToBarbarians) {
            "Convince the barbarians to call off their invasion."
        } else {
            "Save the town from the barbarian invasions."
        }
    } else if (playerLevel == 6) {
        "Locate the enchanted sword."
    } else if (playerLevel == 7) {
        "Recover the long-lost artifact of creation."
    } else if (playerLevel == 8) {
        "Defeat Nogartse, bringer of death and eater of worlds."
    } else {
        "There are no quests right now."
    }
    ...
}
```

Кроме того, использование интервалов в условных выражениях, как показано выше, решает проблему с порядком выполнения `else if`, которую мы наблюдали ранее в этой главе. С интервалами ветви могут располагаться в любом порядке, код все равно будет работать одинаково.

Кроме оператора `..` для создания интервалов можно воспользоваться специальными функциями. Например, функция `downTo` создает интервал с убывающими значениями вместо интервала с возрастающими значениями. Функция `until` создает интервал, не включающий верхнюю границу выбранного диапазона. Их применение мы покажем в «Заданиях» в конце главы, а сами интервалы более подробно рассмотрим в главе 10.

Условное выражение when

Условное выражение `when` — еще один способ управлять потоком выполнения в Kotlin. Как и `if/else`, выражение `when` позволяет сформулировать набор условий и выполнить код, соответствующий истинному условию. Выражение `when` обеспечивает более компактный синтаксис и особенно хорошо для условий с тремя и более ветвями.

Допустим, игрок может быть представителем одной из вымышленных рас — например, орком или гномом, а эти расы вступают в коалиции друг с другом. Выражение `when` берет выбранную расу и возвращает название коалиции, к которой она принадлежит:

```
val race = "gnome"
val faction: String = when (race) {
    "dwarf" -> "Keepers of the Mines"
    "gnome" -> "Tinkerers of the Underground"
    "orc", "human" -> "Free People of the Rolling Hills"
    else -> "Shadow Cabal of the Unseen Realm" // Неизвестная раса
}
```

Сначала объявляется `val race (раса)`. Затем следует объявление следующей переменной `faction` (коалиция), значение которой определяется выражением `when`. Выражение проверяет `race` на соответствие каждому значению слева от оператора `->` (*стрелка*), и когда находит соответствующее значение, присваивает `faction` значение, указанное справа от стрелки. Несколько вариантов с одинаковым выводом (как `orc` и `human` в приведенном примере) можно совместить в один, разделяя их запятыми до `->`.

(`->` имеет другой смысл в иных языках и даже в Kotlin находит другие применения, как будет показано ниже.)

По умолчанию выражение `when` ведет себя так, словно между аргументом в круглых скобках и условиями, задаваемыми в фигурных скобках, находился оператор сравнения `==`. (*Аргумент* — это входные данные, передаваемые блоку кода. Вы узнаете о них больше в главе 4.)

В приведенном примере с выражением `when` значение `race` служит аргументом, поэтому компилятор сравнит значение `race` (в данном случае `"gnome"`) с первым условием (`"dwarf"`). Они не равны, поэтому сравнение дает результат `false`, а выражение переходит к следующему условию.

Следующее сравнение вернет истинный результат, поэтому `faction` будет присвоено значение `"Tinkerers of the Underground"` из соответствующей ветви.

Обратите внимание: выражение `when` используется для присваивания значения переменной `faction`. Присваивание выполняется за пределами выражения `when`, поэтому выражение всегда должно возвращать некоторое значение.

Когда вы используете оператор `when` как выражение (как при присваивании), компилятор будет требовать, чтобы оператор `when` был *исчерпывающим* (exhaustive), то есть распространялся на все возможные случаи. В приведенном примере без ветви `else` оператор `when` не был бы исчерпывающим — переменной `race` можно было бы присвоить много других значений строк, которые не были учтены. Однако ветвь `else` добавляет резервный вариант на случай, если в коде будет использовано неучтенное значение, поэтому компилятор это устраивает.

Иногда выражение `when` может быть исчерпывающим и без ветви `else`. Примеры такого рода мы покажем в главе 16.

Теперь вы узнали, как пользоваться выражениями `when`, и логику вычисления `quest` можно усовершенствовать. В отличие от ранее использовавшейся команды `if/else`, условие `when` делает код более простым и компактным. На практике можно применять простое эмпирическое правило: используйте условие `when`, если выражение `if/else` в вашем коде содержит ветви `else if`.

Обновите логику `quest`, используя `when`.

Листинг 3.10. Рефакторинг кода `quest` с выражением `when` (Main.kt)

```
...
fun main() {
    ...
    val quest: String = if (playerLevel == 1) {
        "Meet Mr. Bubbles in the land of soft things."
    val quest: String = when (playerLevel) {
        1 -> "Meet Mr. Bubbles in the land of soft things."
    } else if (playerLevel in 2..5) {
        in 2..5 -> {
            // Проверить возможность дипломатического решения
            val canTalkToBarbarians = !hasAngeredBarbarians &&
                (hasBefriendedBarbarians || playerClass == "barbarian")

            if (canTalkToBarbarians) {
                "Convince the barbarians to call off their invasion."
            } else {
                "Save the town from the barbarian invasions."
            }
        }
    } else if (playerLevel == 6) {
        "Locate the enchanted sword."
        6 -> "Locate the enchanted sword."
    } else if (playerLevel == 7) {
        "Recover the long-lost artifact of creation."
        7 -> "Recover the long-lost artifact of creation."
    } else if (playerLevel == 8) {
        "Defeat Nogartse, bringer of death and eater of worlds."
    }
        8 -> "Defeat Nogartse, bringer of death and eater of worlds."
    else {
        "There are no quests right now."
    }
}
```

```
        else -> "There are no quests right now."
    }
    ...
}
```

Условное выражение `when` работает так же, как условное выражение `if/else`: оно определяет условия и ветви, которые выполняются при истинности проверяемого условия. Выражение `when` отличается тем, что автоматически выбирает условие слева, соответствующее значению аргумента в *области видимости*. Подробнее тема областей видимости рассматривается в главах 4 и 13. А теперь несколько слов об условном выражении `in 2..5`.

Вы уже видели, как работает ключевое слово `in` для проверки принадлежности значения интервалу, то же происходит и здесь: проверяется значение `playerLevel`, хотя имя и не упоминается явно. В области видимости интервала слева от `->` находится переменная `playerLevel`, поэтому компилятор вычисляет выражения `when`, как если бы имя `playerLevel` входило в каждое условие ветвления.

Часто `when` лучше передает логику кода. В данном случае для получения того же результата с выражением `if/else` потребовалось бы использовать четыре ветви `else if`. Выражение `when` позволяет выразить логику более четко.

Кстати, а вы заметили вложенные операторы `if/else` в одной из ветвей условного выражения `when`? Такая конструкция встречается нечасто, но `when` обеспечивает гибкость, необходимую для ее реализации.

Запустите `bounty-board` и убедитесь в том, что рефакторинг `quest` с использованием `when` не изменил логику выполнения программы.

Выражения `when` с объявлением переменных

Иногда выражение `when` используется с аргументом, который вычисляется только в контексте выражения `when`. Часто удобно использовать значение переменной в одном из условий выражения `when`.

Допустим, вы хотите присвоить игроку титул в зависимости от его текущего опыта, который представляется переменной `totalExperience` типа `Int`. Для простоты считаем, что для перехода между уровнями требуется 100 очков опыта (таким образом, уровень 1 означает, что у игрока 0–99 очков опыта, уровень 2 — 100–199 очков опыта и т. д.). Выражение `when` для генерирования титула может выглядеть так:

```
val playerLevel: Int = totalExperience / 100 + 1
val playerTitle: String = when (playerLevel) {
    1 -> "Apprentice"
    in 2..8 -> "Level " + playerLevel + " Warrior"
    9 -> "Vanquisher of Nogartse"
    else -> "Distinguished Knight"
}
```

Неплохо, но этот код можно еще упростить, переместив объявление переменной в аргумент выражения `when`:

```
val playerTitle = when (val playerLevel = totalExperience / 100 + 1) {  
    1 -> "Apprentice"  
    in 2..8 -> "Level " + playerLevel + " Warrior"  
    9 -> "Vanquisher of Nogartse"  
    else -> "Distinguished Knight"  
}
```

В этом выражении `when`, содержащем объявление переменной, `val playerLevel` содержится внутри выражения `when` и уничтожается после завершения выражения. Основной код не загроможден лишними переменными, а значение не приходится вычислять заново каждый раз, когда оно вам понадобится.

Выражения `when` без аргументов

До сих пор во всех приводимых выражениях `when` передавался аргумент. Такое решение хорошо работало, потому что поведение приложения менялось в зависимости от одной переменной.

Однако у выражений с аргументами есть ограничения:

- выражение `when` не может получать несколько аргументов;
- в выражениях `when` с аргументом могут использоваться только операторы `==`, `in` и `is`.

Если в условиях проверяется несколько возможностей либо вам потребуется использовать другой оператор сравнения, применить выражение `when` с аргументом не удастся. В таких ситуациях возможны два варианта: конструкция `if/else` вроде той, которая была показана ранее в этой главе, либо выражение `when` без аргумента.

Допустим, вы хотите сообщить игроку, сколько очков опыта ему понадобится для достижения следующего уровня. В программе используются две переменные типа `Int` с именами `experiencePoints` и `requiredExperiencePoints`. Выражение `when`, вычисляющее количество очков опыта для перехода на следующий уровень, может выглядеть так:

```
val levelUpStatus: String = when {  
    experiencePoints > requiredExperiencePoints -> {  
        "You already leveled up!"  
    }  
    experiencePoints == requiredExperiencePoints -> {  
        "You have enough experience to level up!"  
    }  
}
```

```
requiredExperiencePoints - experiencePoints < 20 -> {  
    // Игроку остается менее 20 очков опыта  
    "You are very close to leveling up!"  
}  
else -> "You need more experience to level up!"  
}
```

Такая гибкость означает, что операторы `if/else` и выражения `when` взаимозаменяемы. Любые условия, которые могут проверяться командой `if`, также могут присутствовать в выражениях `when` без аргументов. Вы даже можете использовать логические операторы из табл. 3.2.

Задание: эксперименты с интервалами

Интервалы — мощный инструмент, и после небольшой тренировки их синтаксис покажется вам простым и понятным. Для выполнения этого несложного задания откройте Kotlin REPL (Tools ▶ Kotlin ▶ REPL) и поэкспериментируйте с синтаксисом интервалов и функциями `toList()`, `downTo` и `until`. Введите следующие интервалы один за другим. Прежде чем нажать Command-Return (Ctrl-Enter), попробуйте предположить, какой результат вы получите.

Листинг 3.11. Эксперименты с интервалами (REPL)

```
1 in 1..3  
(1..3).toList()  
1 in 3 downTo 1  
1 in 1 until 3  
3 in 1 until 3  
2 in 1..3  
2 !in 1..3  
'x' in 'a'..'z'
```

Подписывайся на самый топовый канал по Kotlin:
<https://t.me/KotlinSenior>

4. Функции

Функция — это фрагмент кода, который выполняет определенную задачу и может использоваться повторно. Функции — это очень важная часть программирования. Фактически программа представляет собой последовательность функций, взаимосвязанных для выполнения сложной задачи.

Ранее вы уже использовали функцию `println`, которую стандартная библиотека Kotlin предоставляет для вывода данных в консоль. Вы также можете объявлять собственные функции в своем коде. Некоторые функции должны получать данные, необходимые для решения задачи. Другие возвращают значения и создают выходные данные, которые будут использованы где-то еще после завершения работы функции.

Начнем с того, что организуем при помощи функций существующий код в `bounty-board`. Для этого определим собственную функцию, которая расширит возможности чтения текста на «доске поручений».

Выделение кода в функции

Логика, которой вы воспользовались в главе 3 для `bounty-board`, была разумной, но лучше организовать код при помощи функций. Ваше первое задание: реорганизовать проект и инкапсулировать большую часть логики в функции. Так вы получите возможность добавить новые опции в `bounty-board`.

Означает ли это, что придется удалить весь код и переписать всю программу? Ни в коем случае. IntelliJ поможет легко сгруппировать логику в функции.

Для начала откройте проект `bounty-board`. Убедитесь, что файл `Main.kt` открыт в редакторе.

Далее выделите условное выражение, которое было определено для описания миссии в `quest`. Нажмите левую кнопку мыши и, удерживая ее, протяните указатель мыши от первой строки с объявлением `quest` до конца выражения, включая закрывающую фигурную скобку, — примерно так:

```
...  
val quest: String = when (playerLevel) {  
    1 -> "Meet Mr. Bubbles in the land of soft things."  
}
```



```

in 2..5 -> {
    val canTalkToBarbarians = !hasAngeredBarbarians &&
        (hasBefriendedBarbarians || playerClass == "barbarian")
    if (canTalkToBarbarians) {
        "Convince the barbarians to call off their invasion."
    } else {
        "Save the town from the barbarian invasions."
    }
}
6 -> "Locate the enchanted sword."
7 -> "Recover the long-lost artifact of creation."
8 -> "Defeat Nogartse, bringer of death and eater of worlds."
else -> "There are no quests right now."
}
...

```

Щелкните правой кнопкой мыши (Control-щелчок) на выделенном фрагменте. Выберите в контекстном меню пункт **Refactor** ▶ **Function...** (рис. 4.1).

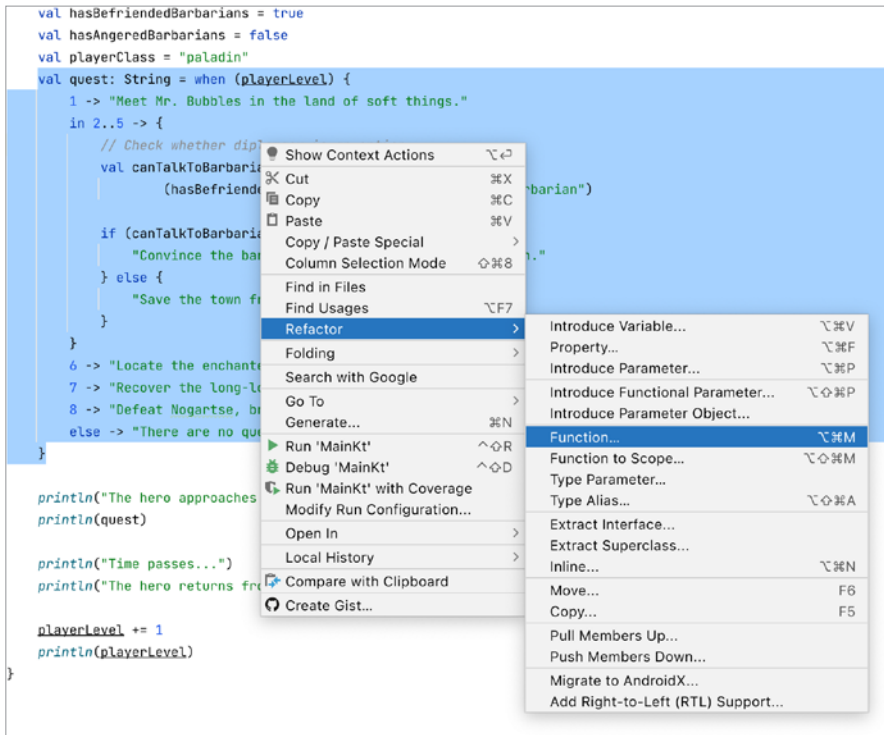


Рис. 4.1. Выделение логики в функцию

На экране появляется диалоговое окно для выделения функции как на рис. 4.2.

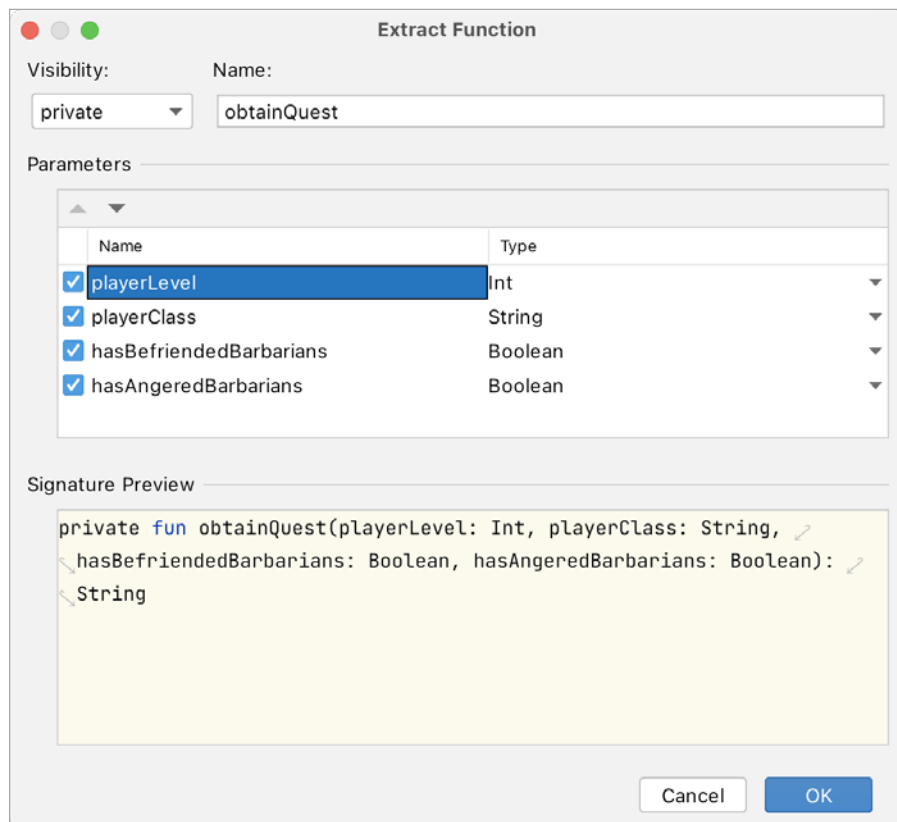


Рис. 4.2. Диалоговое окно выделения функции

Введите имя функции `obtainQuest`. В диалоговом окне присутствуют и другие элементы для назначения модификаторов видимости и параметров. Позднее в этой главе мы объясним, что они делают, а пока проследите за тем, чтобы они находились в состоянии как на рис. 4.2: в раскрывающемся списке **Visibility** выбрана строка `private`, а параметры на панели **Parameters** следуют в показанном порядке — `playerLevel`, `playerClass`, `hasBefriendedBarbarians`, `hasAngeredBarbarians`. Если вам потребуется изменить порядок параметров, перетащите их мышью в списке.

Когда все будет готово, нажмите кнопку **OK**. IntelliJ добавит определение функции в конец файла `Main.kt`:

```
private fun obtainQuest(
    playerLevel: Int,
```

```
playerClass: String,
hasBefriendedBarbarians: Boolean,
hasAngeredBarbarians: Boolean
): String {
    val quest: String = when (playerLevel) {
        1 -> "Meet Mr. Bubbles in the land of soft things."
        in 2..5 -> {
            val canTalkToBarbarians = !hasAngeredBarbarians &&
                (hasBefriendedBarbarians || playerClass == "barbarian")
            if (canTalkToBarbarians) {
                "Convince the barbarians to call off their invasion."
            } else {
                "Save the town from the barbarian invasions."
            }
        }
        6 -> "Locate the enchanted sword."
        7 -> "Recover the long-lost artifact of creation."
        8 -> "Defeat Nogartse, bringer of death and eater of worlds."
        else -> "There are no quests right now."
    }
    return quest
}
```

В нашей функции `obtainQuest` появился новый код. Давайте разберем его.

Анатомия функции

На рис. 4.3 показаны две основные части функции, *заголовок* и *тело*, на примере функции `obtainQuest`:

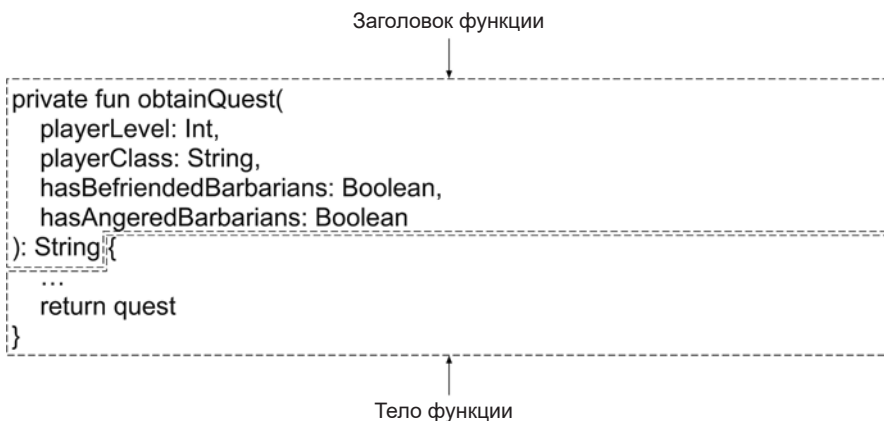


Рис. 4.3. Функция состоит из заголовка и тела

Заголовок функции

Первая часть функции — это заголовок. Он состоит из пяти частей: модификатора видимости, ключевого слова объявления функции, имени функции, параметров функции, типа возвращаемого значения (рис. 4.4).

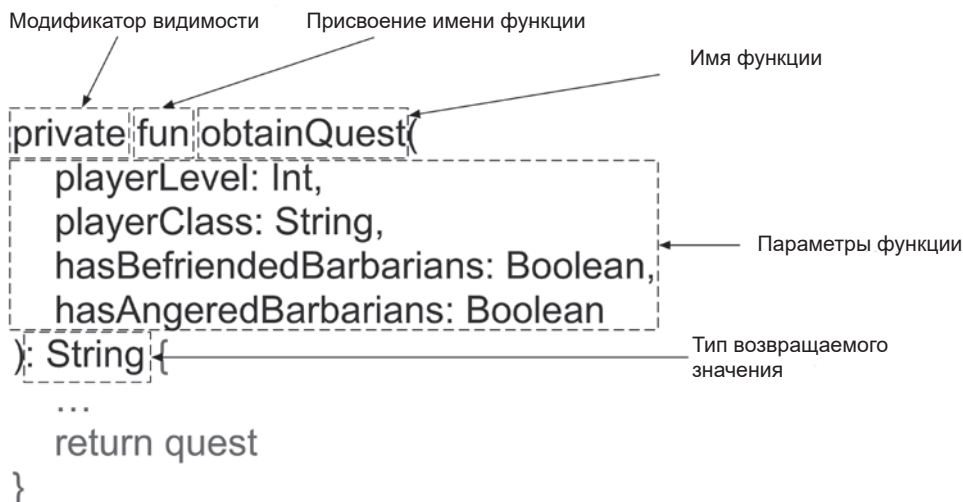


Рис. 4.4. Анатомия заголовка функции

Рассмотрим каждый из этих элементов подробнее.

Модификатор видимости

Не все функции должны быть *видимы* (доступны) для других функций. Например, некоторые функции могут оперировать данными, которые не должны быть доступны за пределами конкретного файла.

При необходимости объявление функции может начинаться с *модификатора видимости* (рис. 4.5). Он определяет, какие другие функции смогут «видеть», а следовательно использовать, данную функцию.

По умолчанию функция имеет глобальную видимость (`public`), то есть является открытой, — это означает, что она может использоваться всеми остальными функ-

```

private fun obtainQuest(
    playerLevel: Int,
    playerClass: String,
    hasBefriendedBarbarians: Boolean,
    hasAngeredBarbarians: Boolean
): String {
    ...
    return quest
}

```

Рис. 4.5. Модификатор видимости функции

циями (даже теми, которые объявлены в других файлах проекта). Другими словами, если вы не указали модификатор, считается, что используется модификатор `public`.

В данном случае среда IntelliJ выбрала модификатор `private`, так как функция `obtainQuest` используется только в файле `Main.kt`. Больше о модификаторах видимости и о том, как их использовать, вы узнаете в главе 13.

Объявление имени функции

После модификатора видимости (если он присутствует) идет ключевое слово `fun`, сопровождаемое именем функции (рис. 4.6).

Имя функции `obtainQuest` было указано в диалоговом окне выделения функции, поэтому IntelliJ добавила это имя после `fun`.

Обратите внимание, что имя, выбранное для функции `obtainQuest`, начинается со строчной буквы и использует верблюжий Регистр без подчеркиваний. Старайтесь всем своим функциям давать имена в таком официально признанном стиле.

```
private fun obtainQuest(  
    playerLevel: Int,  
    playerClass: String,  
    hasBefriendedBarbarians: Boolean,  
    hasAngeredBarbarians: Boolean  
): String {  
    ...  
    return quest  
}
```

Рис. 4.6. Ключевое слово `fun` и объявление имени

Параметры функции

Далее следуют параметры функции (рис. 4.7).

Параметры функции определяют имена и типы входных данных, необходимых функции для решения задачи. Функции может требоваться от нуля до нескольких параметров в зависимости от того, для какой задачи она проектировалась. Так как эта функция получает много параметров, каждый из них размещается в отдельной строке в соответствии с официальным стилем Kotlin. Если бы параметров было меньше, всю сигнатуру функции можно было бы уместить в одной строке. (Пример такого рода вы увидите при определении второй функции.)

Чтобы функция `obtainQuest` могла определить, какое описание миссии ей вывести, ей необходимы переменные `playerLevel`, `playerClass`, `hasBefriendedBarbarians`

```
private fun obtainQuest(  
    playerLevel: Int,  
    playerClass: String,  
    hasBefriendedBarbarians: Boolean,  
    hasAngeredBarbarians: Boolean  
): String {  
    ...  
    return quest  
}
```

Рис. 4.7. Параметры функции

и `hasAngeredBarbarians`, потому что они требуются условному выражению `when` для проверки условий:

```
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String,
    hasBefriendedBarbarians: Boolean,
    hasAngeredBarbarians: Boolean
): String {
    val quest: String = when (playerLevel) {
        1 -> "Meet Mr. Bubbles in the land of soft things."
        in 2..5 -> {
            val canTalkToBarbarians = !hasAngeredBarbarians &&
                (hasBefriendedBarbarians || playerClass == "barbarian")
            ...
        }
        ...
    }
    return quest
}
```

Таким образом, определение функции `obtainQuest` указывает, что эти четыре переменные необходимы ей как параметры.

Для каждого параметра определяется также его тип. Параметр `playerLevel` должен иметь тип `Int`, параметр `playerClass` — тип `String`, а параметры `hasBefriendedBarbarians` и `hasAngeredBarbarians` — тип `Boolean`.

Обратите внимание, что параметры функции всегда доступны только для чтения — их изменение в теле функции не поддерживается. Другими словами, параметры в теле функции — это `val`, а не `var`.

Тип возвращаемого значения

Многие функции генерируют выходные данные; это их основная задача — возвращать значение какого-то типа туда, откуда они вызваны. Последняя часть заголовка функции — это *тип возвращаемого значения*, который определяет тип результата функции после завершения ее работы. Если вашей функции не нужно возвращать никакие данные, информацию о возвращаемом типе можно не включать в сигнатуру функции. (Эта возможность также будет продемонстрирована во второй функции.)

```
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String,
    hasBefriendedBarbarians: Boolean,
    hasAngeredBarbarians: Boolean
): String {
    ...
    return quest
}
```

Рис. 4.8. Тип возвращаемого значения

Тип `String` возвращаемого значения `obtainQuest` указывает, что функция возвращает строку (рис. 4.8).

В данном случае функция возвращает одну из строк в зависимости от уровня игрока и значений других переменных.

Тело функции

За заголовком следует тело функции, заключенное в фигурные скобки. Тело — это та часть функции, где происходит основное действие. Оно может содержать команду `return`, которая определяет возвращаемые данные.

В нашем случае команда выделения функции переместила объявление `val quest` (код, который вы выбирали ранее при запуске программы) в тело функции `obtainQuest`.

Далее следует новая строка `return quest`. Ключевое слово `return` указывает компилятору, что функция завершила работу и готова передать выходные данные. Выходные данные в нашем случае — `quest`. Это означает, что функция вернет значение переменной `quest` — строку, выбранную на основании логики определения `quest`.

Область видимости функции

Переменная `quest` объявляется и инициализируется внутри тела функции, а ее значение возвращается в конце:

```
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String,
    hasBefriendedBarbarians: Boolean,
    hasAngeredBarbarians: Boolean
): String {
    val quest = when (playerLevel) {
        ...
    }
    return quest
}
```

Переменная `quest` является *локальной переменной*, так как существует только в теле функции `obtainQuest`. Также можно сказать, что переменная `quest` существует только в *области видимости* функции `obtainQuest`. Область видимости можно рассматривать как продолжительность жизни переменной.

Так как переменная `quest` существует только в области видимости функции, она прекратит свое существование после завершения работы функции `obtainQuest`. Функция возвращает значение `quest` на сторону вызова, но переменная, в которой это значение хранилось, пропадает при завершении функции.

Это верно и для параметров функции: переменные `playerLevel`, `playerClass`, `hasBefriendedBarbarians` и `hasAngeredBarbarians` существуют только внутри области видимости функции и перестают существовать после завершения функции.

В главе 2 мы показали пример переменной, которая не была локальной для функции или класса, — *переменную уровня файла*:

```
const val HERO_NAME = "Madrigal"
```

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

Переменная уровня файла, такая как `HERO_NAME`, доступна из любого места в проекте (однако в объявление можно добавить модификатор видимости и изменить область видимости переменной). Переменные уровня файла существуют, пока не завершится выполнение всей программы. Кстати говоря, хотя `HERO_NAME` является константой, переменные также могут определяться на уровне файла — более того, вскоре вы познакомитесь с примером такого рода.

Из-за различий между локальными переменными и переменными уровня файла компилятор выдвигает разные требования к тому, когда им должно присваиваться начальное значение, или, говоря иначе, когда они должны *инициализироваться*.

Значения переменным уровня файла должны присваиваться сразу при объявлении, иначе код не будет компилироваться. (Есть исключения, и они описаны в главе 16.) Это требование защитит вас от непредвиденного и нежелательного поведения, например, при попытке использовать переменную до ее инициализации.

Так как локальная переменная имеет более ограниченную область применения — внутри функции, в которой определена, — компилятор более снисходителен к тому, где она может инициализироваться, лишь бы она инициализировалась до использования. Это означает, что следующий код верен:

```
fun main() {  
    var playerLevel: Int  
    println("The hero announces her presence to the world.")  
    println(HERO_NAME)  
    playerLevel = 5  
    println(playerLevel)  
    ...  
}
```

Если код не обращается к переменной до ее инициализации, компилятор почитает его допустимым.

Вызов функции

Среда IntelliJ не только сгенерировала функцию `obtainQuest`, но и добавила строку кода в место, откуда выделена функция:

```
...
fun main() {
    ...
    val hasBefriendedBarbarians = true
    val hasAngeredBarbarians = false
    val playerClass = "paladin"
    val quest: String = obtainQuest(playerLevel, playerClass,
                                    hasBefriendedBarbarians, hasAngeredBarbarians)
    ...
}
```

(Мы разбили код на две строки, чтобы он помещался по ширине страницы. Возможно, среда IntelliJ разместит его в одной строке проекта; этот код работает одинаково независимо от того, сколько строк он занимает.)

В этой строке находится *вызов функции*, которая активирует функцию для выполнения действий, заданных в ее теле. Для вызова функции нужно указать ее имя и данные, соответствующие параметрам, как определено в заголовке.

Сравните заголовок функции `obtainQuest` с ее вызовом:

```
obtainQuest(                // Заголовок
    playerLevel: Int,
    playerClass: String,
    hasBefriendedBarbarians: Boolean,
    hasAngeredBarbarians: Boolean
): String
obtainQuest(                // Вызов
    playerLevel,
    playerClass,
    hasBefriendedBarbarians,
    hasAngeredBarbarians
)
```

В объявлении `obtainQuest` видно, что функция требует передачи четырех параметров. Вызывая `obtainQuest`, вы должны указать входные данные для этих параметров, заключив их в круглые скобки. Входные данные называются *аргументами*, а передача их в функцию называется *передачей аргументов*.

(Терминологическая справка: строго говоря, параметр — это то, что требуется функции, а аргумент — то, что передается при вызове функции для выполнения этого требования, но эти термины часто используются как синонимы.)

Здесь, как указано в определении функции, вы передаете значение `playerLevel` (которое должно быть значением типа `Int`), строковое значение `playerClass` и логические значения `hasBefriendedBarbarians` и `hasAngeredBarbarians`.

Запустите `bounty-board` — и вуаля! Вы увидите такой же вывод, как и раньше:

```
The hero announces her presence to the world.
Madrigal
4
The hero approaches the bounty board. It reads:
Convince the barbarians to call off their invasion.
Time passes...
The hero returns from her quest.
5
```

Хотя вывод не изменился, код `bounty-board` стал более организованным, и его удобнее сопровождать.

Пишем свои функции

Теперь, когда мы выделили часть логики `bounty-board` в функцию, можно переходить к реализации новой функции для чтения «доски поручений». Под функцией `main` определите функцию `readBountyBoard`, которая не получает аргументов. Назначьте ей модификатор видимости `private`. Функция `readBountyBoard` не должна иметь возвращаемого значения, но должна выводить содержимое «доски поручений».

Функция `readBountyBoard` также должна иметь доступ к переменной `playerLevel`. Хотя ее можно передать в аргументе, перемещение `playerLevel` в свойство уровня файла упростит вызов новой функции. При перемещении объявления переменной также повысьте уровень героя с 4 до 5, изменив значение `playerLevel`. Это позволит вам увидеть более интересные миссии.

Листинг 4.1. Добавление функции `readBountyBoard` (`Main.kt`)

```
const val HERO_NAME = "Madrigal"
var playerLevel = 5

fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME)
    var playerLevel = 4
    println(playerLevel)

    val hasBefriendedBarbarians = true
    val hasAngeredBarbarians = false
    val playerClass = "paladin"
    val quest: String = obtainQuest(playerLevel, playerClass,
        hasBefriendedBarbarians, hasAngeredBarbarians)

    println("The hero approaches the bounty board. It reads:")
    println(quest)
```

```
println("Time passes...")
println("The hero returns from her quest.")

playerLevel += 1
println(playerLevel)
}

private fun readBountyBoard() {
    println("The hero approaches the bounty board. It reads:")
    println(observeOnQuest(playerLevel, "paladin", true, false))
}
...
```

С помощью новой функции можно упростить `main`, вызывая `readBountyBoard` вместо включения полной логики определения миссии. Замените существующую логику вывода вызовом `readBountyBoard`. После завершения миссии герой также должен вернуться к «доске поручений», поэтому добавьте второй вызов новой функции в конец `main`. (Функция `readBountyBoard` была определена без параметров, поэтому при вызове ей не передаются никакие аргументы — просто оставьте скобки пустыми.)

Листинг 4.2. Вызов `readBountyBoard` (Main.kt)

```
...
fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME)
    println(playerLevel)

    val hasBefriendedBarbarians = true
    val hasAngeredBarbarians = false
    val playerClass = "paladin"
    val quest: String = obtainQuest(playerLevel, playerClass,
        hasBefriendedBarbarians, hasAngeredBarbarians)

    println("The hero approaches the bounty board. It reads:")
    println(quest)
    readBountyBoard()

    println("Time passes...")
    println("The hero returns from her quest.")

    playerLevel += 1
    println(playerLevel)
    readBountyBoard()
}
...
```

Запустите bounty-board и полюбуйтесь новым выводом:

```
The hero announces her presence to the world.
Madrigal
5
The hero approaches the bounty board. It reads:
Convince the barbarians to call off their invasion.
Time passes...
The hero returns from her quest.
6
The hero approaches the bounty board. It reads:
Locate the enchanted sword.
```

Большая часть вывода осталась неизменной. Но с помощью новой функции `readBountyBoard` доступ к описаниям миссий стал как никогда простым и удобным. Теперь для этого достаточно всего одной строки кода, что заметно упрощает структуру кода и его повторное использование.

Аргументы по умолчанию

Иногда некое значение аргумента функции используется чаще других. Например, весьма маловероятно, что герой рассердил варваров. В зависимости от этапа игры может оказаться, что он еще не встретил варварское племя (если игра только начинается) или ссоры с варварами ниже его достоинства (если он прокачался до высокого уровня). Передавать эту информацию при каждом вызове избыточно. Чтобы упростить вызов `obtainQuest`, добавьте *аргумент по умолчанию*.

Если параметр имеет значение по умолчанию, то функцию можно вызывать без указания значения этого параметра. В этом случае значение по умолчанию будет автоматически передано как аргумент функции. Обновите функцию `obtainQuest`, добавив значение по умолчанию для `hasAngeredBarbarians`.

Листинг 4.3. Значение по умолчанию для параметра `hasAngeredBarbarians` (Main.kt)

```
...

private fun obtainQuest(
    playerLevel: Int,
    playerClass: String,
    hasBefriendedBarbarians: Boolean,
    hasAngeredBarbarians: Boolean = false
): String {
    ...
}
```

Теперь при вызове `obtainQuest` для параметра `hasAngeredBarbarians` будет передаваться логическое значение `false`, если явно не указано иное значение. Хотя синтаксис почти не отличается от других объявлений переменных, которые

встречались вам ранее, существует одно тонкое различие. Kotlin не выполняет автоматическое определение типа с параметрами функций, поэтому тип аргумента обязательно следует задавать явно (даже если он очевиден для компилятора, как логическое значение `false`).

Аргумент по умолчанию определен, но еще не используется в программе. Обновите функцию `readBountyBoard` и удалите аргумент `false` из вызова `obtainQuest`.

Листинг 4.4. Использование значения аргумента по умолчанию в `obtainQuest` (Main.kt)

```
...
private fun readBountyBoard() {
    println("The hero approaches the bounty board. It reads:")
    println(obtainQuest(playerLevel, "paladin", true, false))
    println(obtainQuest(playerLevel, "paladin", true))
}
...
```

Снова запустите `bounty-board`. И хотя аргумент для `hasAngeredBarbarians` не задан, вы получите тот же результат, что и прежде.

Потратьте еще немного времени на обновление параметров `playerClass` и `hasBefriendedBarbarians`, чтобы они также получили значения по умолчанию.

Листинг 4.5. Значения по умолчанию для других параметров (Main.kt)

```
...
private fun readBountyBoard() {
    println("The hero approaches the bounty board. It reads:")
    println(obtainQuest(playerLevel, "paladin", true))
    println(obtainQuest(playerLevel))
}
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
    hasAngeredBarbarians: Boolean = false
): String {
    ...
}
```

Аргументы по умолчанию — полезный инструмент, он позволяет передавать необязательные входные значения. Многие функции стандартной библиотеки Kotlin используют аргументы по умолчанию для изменения поведения функции.

Например, для типа `String` есть функция `equals` с логическим параметром `ignoreCase`, который указывает, следует ли игнорировать регистр символов, когда вы проверяете, содержат ли две строки одинаковый текст. Значение по умолчанию для `ignoreCase` равно `false`, потому что регистр символов обычно

важен при сравнении строк — это «поведение по умолчанию». Аргументы по умолчанию в Kotlin позволяют легко задавать поведение для повторяющихся сценариев или же переключиться на другой набор особенностей поведения, предоставляемых функцией.

Мы рекомендуем использовать аргумент по умолчанию, когда для соответствующего параметра существует типичное входное значение. Для аргументов с большим разбросом значений или для аргументов, имеющих много разумных значений по умолчанию, мы не советуем задавать аргумент по умолчанию. Кроме того, иногда лучше требовать обязательной передачи параметра, чтобы сделать его более очевидным, чем стремиться к компактному вызову функций.

Функции с единственным выражением

Просматривая свой код, вы можете заметить, что каждая из функций `main` и `readBountyBoard` содержит несколько вызовов `println`, тогда как функция `obtainQuest` делает только одно: она выдает строку с описанием миссии в зависимости от статуса игрока.

В программировании вычисляемая конструкция, такая как вызов функции или объявление переменной, называется *выражением*. В текущей версии `obtainQuest` содержит два выражения: объявление/присваивание переменной `quest` и команду `return quest`. Тело функции можно упростить, чтобы в нем использовалось только одно выражение, удалив промежуточную переменную `quest`.

Листинг 4.6. Удаление промежуточной переменной `quest` (Main.kt)

```
...
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
    hasAngeredBarbarians: Boolean = false
): String {
    val quest: String = when (playerLevel) {
    return when (playerLevel) {
        1 -> "Meet Mr. Bubbles in the land of soft things."
        ...
        else -> "There are no quests right now."
    }
    return quest
}
```

Если ваша функция содержит только одно выражение, Kotlin предлагает альтернативный синтаксис: можно не указывать тип возвращаемого значения, фигурные скобки и команду `return`. Внесите следующие изменения в функцию `obtainQuest`:

Листинг 4.7. Альтернативный синтаксис определения функций с единственным выражением (Main.kt)

```
...
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
    hasAngeredBarbarians: Boolean = false
): String {
    return when (playerLevel) {
    } = when (playerLevel) {
        1 -> "Meet Mr. Bubbles in the land of soft things."
        in 2..5 -> {
            val canTalkToBarbarians = !hasAngeredBarbarians &&
                (hasBefriendedBarbarians || playerClass == "barbarian")
            if (canTalkToBarbarians) {
                "Convince the barbarians to call off their invasion."
            } else {
                "Save the town from the barbarian invasions."
            }
        }
        6 -> "Locate the enchanted sword."
        7 -> "Recover the long-lost artifact of creation."
        8 -> "Defeat Nogartse, bringer of death and eater of worlds."
        else -> "There are no quests right now."
    }
}
```

Вместо того чтобы использовать тело функции для определения работы, которую она должна проделать, можно применить альтернативный синтаксис с оператором присваивания (`=`), за которым следует выражение.

Альтернативный синтаксис позволит сократить объявления функций с единственным выражением, которое должно быть вычислено для выполнения работы. Если вам нужны результаты вычисления нескольких выражений, то используйте синтаксис, описанный ранее.

Также из заголовка функции была удалена информация о типе возвращаемого значения. По аналогии с автоматическим определением типов переменных, Kotlin может автоматически определить возвращаемый тип функции при использовании синтаксиса функции с единственным выражением. Вы вправе вручную включить возвращаемый тип, чтобы улучшить удобочитаемость кода. В некоторых сценариях включение информации о типе в функцию с единственным выражением может быть обязательным, например, если тело функции определяет свой тип на основании возвращаемого типа функции.

Так как `obtainQuest` содержит сложное выражение `when`, занимающее несколько строк, мы рекомендуем включать тип возвращаемого значения. Чтобы добавить возвращаемый тип для функции с единственным выражением, вставьте его между закрывающей круглой скобкой `()` и оператором присваивания `=`.

Листинг 4.8. Включение возвращаемого типа для функций с единственным выражением (Main.kt)

```
...
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
    hasAngeredBarbarians: Boolean = false
): String = when (playerLevel) {
    1 -> "Meet Mr. Bubbles in the land of soft things."
    ...
}
```

Функции с возвращаемым типом Unit

Не все функции возвращают значение. Некоторые выполняют работу за счет побочных эффектов, например изменения состояния переменной или вызова других функций, которые обеспечивают вывод данных. Вспомните функцию `readBountyBoard` и даже функцию `main`: они объявлены без возвращаемого типа и не содержат команды `return`, а для выполнения используют `println`.

```
private fun readBountyBoard() {
    println("The hero approaches the bounty board. It reads:")
    println(obtainQuest(playerLevel))
}
```

В Kotlin такие функции известны как функции с возвращаемым типом `Unit`. Если функция не определяет возвращаемый тип, Kotlin автоматически использует возвращаемый тип `Unit`. Щелкните на одном из вызовов функции `readBountyBoard` и нажмите `Control-Shift-P`. IntelliJ отобразит тип возвращаемого значения (рис. 4.9).

Что это за тип такой — `Unit`? Kotlin использует `Unit` для обозначения функции, которая не возвращает значения. Если ключевое слово `return` отсутствует, то считается, что функция возвращает значение типа `Unit`.

До появления Kotlin многие языки сталкивались с проблемой описания функции, которая ничего не возвращает. Некоторые языки выбрали ключевое слово `void`, которое сообщает: «Возвращаемый тип отсутствует, пропустите его, так как он не применяется». Это достаточно очевидная мысль: если функция ничего не возвращает, то тип можно пропустить.

К сожалению, это решение не учитывает важную черту, реализованную в современных языках, — обобщение. Обобщение — это особенность современных компилируемых языков, которая позволяет достичь большой гибкости. Вы позна-



Рис. 4.9. `readBountyBoard` как функция с возвращаемым типом `Unit`

комитесь с обобщением в языке Kotlin, которое позволяет определять функции, способные работать с разными типами, в главе 18.

Как связаны обобщения с `Unit` или `void`? Языки, использующие ключевое слово `void`, не обладают достаточными возможностями для работы с обобщенными функциями, которые не возвращают ничего. `void` — это не тип. Фактически это ключевое слово сообщает: «Информация о типе не имеет смысла; его можно пропустить». Поэтому такие языки не могут описать обобщенную функцию, которая не возвращает ничего.

В Kotlin проблема решается введением `Unit` как типа возвращаемого значения. `Unit` обозначает функцию, которая ничего не возвращает, но может применяться к обобщенным функциям, для работы с которыми нужен хоть какой-то тип. Вот почему в Kotlin используется `Unit`: вам достается лучшее от обоих подходов.

Именованные аргументы функций

Допустим, в местной деревне живет кузнец. В программе может присутствовать функция для изготовления нового снаряжения для нашего героя:

```
fun forgeItem(
    itemName: String,
    material: String,
    encrustWithJewels: Boolean = false,
    quantity: Int = 1
): String = ...
```

Другой способ вызова той же функции:

```
forgeItem(
    itemName = "sword",
    material = "iron",
    encrustWithJewels = false,
    quantity = 5
)
```

Этот необязательный синтаксис использует *именованные аргументы функции* и является альтернативой простой передаче аргументов. В некоторых ситуациях этот способ имеет ряд преимуществ.

Например, именованные аргументы можно передавать функции в любом порядке. То есть `forgeItem` можно вызвать так:

```
forgeItem(
    quantity = 5,
    material = "iron",
    itemName = "sword",
    encrustWithJewels = false
)
```

Без использования именованных аргументов аргументы можно передавать функции только в том порядке, в каком они перечисляются в ее заголовке функции.

Еще один плюс именованных аргументов — они делают код более ясным. Если функция имеет большое число параметров, порой трудно понять, какие аргументы каким параметрам соответствуют.

Это особенно заметно, когда имена передаваемых переменных не совпадают с именами параметров функции, а также при передаче нескольких параметров одного типа. Например, вызов функции `forgeItem("iron", "sword")` *выглядит* нормально и будет компилироваться; вы не поймете, что здесь что-то не так, пока вместо железного меча герой не получит железо, сделанное из мечей.

Именованные аргументы всегда имеют те же имена, что и соответствующие им параметры. Такое четкое соответствие существенно упрощает чтение кода, например, когда вам придется просматривать свой же код, который вы уже успели подзабыть, когда ваш коллега возьмется рецензировать ваш код или когда вы решите просмотреть файл за пределами IDE.

В некоторых ситуациях использование именованных аргументов становится обязательным. Для примера рассмотрим версию `forgeItem`, которая не задает аргумент по умолчанию для `quantity`:

```
fun forgeItem(  
    itemName: String,  
    material: String,  
    encrustWithJewels: Boolean = false,  
    quantity: Int  
): String = ...
```

Если вы попытаетесь вызвать эту версию `forgeItem`, не указав значение для `encrustWithJewels` (инкрустировать драгоценными камнями) и без использования именованных аргументов, код не будет компилироваться. Некорректный вызов функции будет выглядеть примерно так: `forgeItem("steel", "dagger", 2)`. При таком вызове функции компилятор полагает, что вы передаете значение 2 как входные данные для `encrustWithJewels`, а это недопустимо.

Чтобы вызвать эту версию `forgeItem` с аргументом по умолчанию, необходимо использовать именованный параметр, чтобы однозначно сообщить, к какому аргументу относится это значение:

```
obtainQuest("steel", "dagger", quantity = 2)
```

При комбинированном использовании аргументов по умолчанию с именованными аргументами мы рекомендуем при объявлении функций размещать аргументы со значениями по умолчанию в конце списка аргументов.

Также можно использовать именованные параметры только с частью предоставляемых аргументов. Например, ниже приведен допустимый вызов функции `forgeItem` (с единственным аргументом по умолчанию для `encrustWithJewels`):

```
forgeItem(  
    "gauntlet",  
    material = "bronze",  
    encrustWithJewels = true,  
    1  
)
```

Если вы предоставляете аргументы в порядке их объявления, Kotlin позволит выбрать, какие аргументы будут именованными, а какие нет. Но стоит вам разместить именованный аргумент в порядке, отличном от порядка в заголовке функции, все остальные аргументы также должны стать именованными.

Например, если использовать именованные аргументы для размещения аргумента `encrustWithJewels` до `material`, после него также необходимо использовать именованные аргументы:

```
forgeItem(  
    "gauntlet",  
    encrustWithJewels = true,  
    material = "bronze",  
    quantity = 1  
)
```

В общем случае мы рекомендуем либо использовать именованные аргументы со всеми параметрами, которые должны передаваться функции, либо не использовать их вообще. Если смысл первого аргумента достаточно очевиден, имя первого аргумента можно не указывать, но следует предоставить имена для второго и всех последующих аргументов. Поэкспериментируйте в IDE и посмотрите, что происходит при различных перестановках именованных аргументов.

Из этой главы вы узнали, как объявлять функции и инкапсулировать логику работы. Эти возможности делают ваш код более упорядоченным и чистым. Вы также узнали о некоторых удобных особенностях синтаксиса функций языка Kotlin, которые позволяют писать более компактный и содержательный код: о функциях с единственным выражением, именованных аргументах и аргументах по умолчанию.

В следующих главах вы используете другие функции Kotlin, чтобы улучшить проект `bounty-board`. А начнем мы с более глубокого изучения числовых типов Kotlin.

Для любознательных: тип `Nothing`

В этой главе вы познакомились с типом `Unit` и узнали, что функции с таким типом ничего не возвращают.

Есть другой тип, похожий на `Unit`, — тип `Nothing` (ничего). Как и `Unit`, тип `Nothing` показывает, что функция не возвращает значения, но на этом их сходство и заканчивается. `Nothing` сообщает компилятору, что функция гарантированно не будет выполнена успешно; функция либо породит исключение, либо по другой причине никогда не вернет управление в точку вызова.

Зачем нужен тип `Nothing`? Один из примеров использования этого типа — функция `TODO` из стандартной библиотеки языка Kotlin.

Давайте познакомимся с `TODO` поближе. Дважды нажмите `Shift`, чтобы вызвать диалоговое окно `Search Everywhere` (Искать везде). Установите флажок `Include non-project items` и введите имя функции `TODO` в поле поиска. Выберите в списке результатов вариант `TODO() (kotlin)` (рис. 4.10).

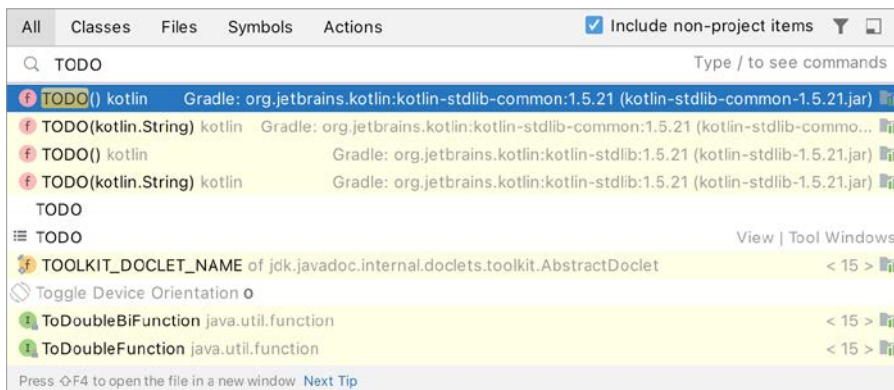


Рис. 4.10. Выбор `TODO() (kotlin)` в диалоговом окне `Search Everywhere`

Вы увидите следующее объявление:

```
/**
 * Always throws [NotImplementedError] stating that operation is not implemented.
 */
public inline fun TODO(): Nothing = throw NotImplementedError()
```

(«Всегда выдает `[NotImplementedError]` как сигнал о том, что операция не реализована») `TODO` выдает исключение — иначе говоря, гарантированно завершается ошибкой — и возвращает тип `Nothing`.

Когда надо использовать `TODO`? Ответ кроется в ее названии: оно говорит вам, что «надо сделать» («to do»). Вот, например, следующая функция, которую еще предстоит реализовать, но пока она просто вызывает `TODO`:

```
fun shouldReturnAString(): String {  
    TODO("implement the string building functionality here to return a string")  
}
```

Разработчик знает, что функция `shouldReturnAString` должна вернуть строку (`String`), но пока отсутствуют другие функции, необходимые для ее реализации. Обратите внимание, для `shouldReturnAString` указано возвращаемое значение `String`, но на самом деле функция ничего не возвращает. Из-за возвращаемого значения `TODO` это абсолютно нормально.

Возвращаемый тип `Nothing` у `TODO` показывает компилятору, что функция гарантированно породит ошибку, поэтому проверять возвращаемое значение после `TODO` не имеет смысла, так как `shouldReturnAString` ничего не вернет. Компилятор доволен, а разработчик может продолжать разработку, отложив реализацию функции `shouldReturnAString` до того момента, когда будут проработаны все детали.

Другая полезная при разработке особенность `Nothing` заключается в возможности добавить код после вызова `TODO`. В этом случае компилятор выведет предупреждение, что этот код недостижим (`unreachable`) (рис. 4.11).

Благодаря возвращаемому типу `Nothing` компилятор знает, что `TODO` завершится с ошибкой, поэтому и весь код после `TODO` не будет выполнен.

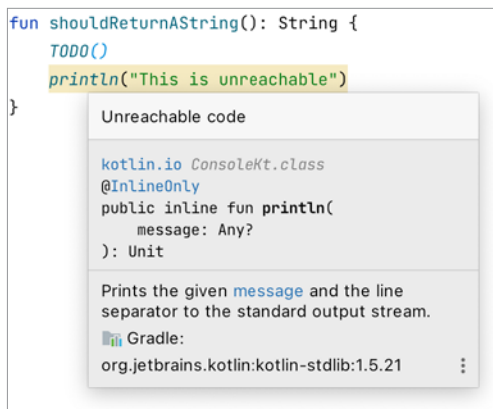


Рис. 4.11. Недостижимый код

Для любознательных: функции уровня файла в Java

Все функции, которые вы писали до этого момента, объявлялись на уровне файла `Main.kt`. Если вы — разработчик Java, это может вас удивить. В Java функции и переменные могут быть объявлены только внутри классов. Это правило не относится к Kotlin.

Возможно ли такое, чтобы код языка Kotlin компилировался в байт-код Java для запуска на JVM? Разве Kotlin работает не по тем же правилам? Если посмотреть на скомпилированный байт-код Java для `Main.kt`, то многое прояснится:

```
public final class MainKt {
    ...

    public static final void main() {
        ...
    }

    private static final void readBountyBoard() {
        ...
    }

    private static final String obtainQuest(...) {
        ...
    }

    // $FF: синтетический метод
    static String obtainQuest$default(...) {
        ...
    }
}
```

Функции уровня файла представлены в Java статическими методами класса с именем, совпадающим с именем файла в Kotlin. (*Методы* в Java — это то же самое, что и *функции*.) В этом случае функции и переменные, объявленные в `Main.kt`, объявляются в Java в классе с именем `MainKt`.

В главе 13 вы узнаете, как объявить функции в классах, но возможность объявлять функции и переменные вне классов дает свободу выбора при создании функций, которые не привязаны к конкретному классу. (А если вас интересует, откуда взялся метод `obtainQuest$default` в `MainKt`, — он используется для определения аргументов по умолчанию. Подробнее об этом мы расскажем в главе 23.)

Для любознательных: перегрузка функций

Функцию `obtainQuest`, которую вы определили, с ее аргументами по умолчанию для параметров `playerClass`, `hasBefriendedBarbarians` и `hasAngeredBarbarians`, можно вызвать многими способами. Вот лишь некоторые из них:

```
obtainQuest(playerLevel)
obtainQuest(playerLevel, playerClass)
obtainQuest(playerLevel, playerClass, hasBefriendedBarbarians)
obtainQuest(playerLevel, playerClass, hasBefriendedBarbarians,
            hasAngeredBarbarians)
```

Если функция имеет несколько сигнатур (как, например, `obtainQuest`), то ее называют *перегруженной* (overloaded). Перегруженность не всегда является след-

ствием использования аргумента по умолчанию. Можно реализовать несколько функций с одним и тем же именем. Чтобы увидеть, как это работает, откройте Kotlin REPL (Tools ► Kotlin ► Kotlin REPL) и введите следующие определения функций:

Листинг 4.9. Определение перегруженной функции (REPL)

```
fun performCombat() {
    println("You see nothing to fight!")
}

fun performCombat(enemyName: String) {
    println("You begin fighting $enemyName.")
}

fun performCombat(enemyName: String, isBlessed: Boolean) {
    val combatMessage: String = if (isBlessed) {
        "You begin fighting $enemyName. You are blessed with 2X damage!"
    } else {
        "You begin fighting $enemyName."
    }
    println(combatMessage)
}
```

Вы объявили три реализации функции `performCombat` (начать битву). Все они являются функциями `Unit` без возвращаемого значения. Первая не получает аргументов. Вторая получает только один аргумент — имя противника. Последняя получает два аргумента: имя противника и логическую переменную с указанием, благословлен ли игрок. Каждая функция генерирует отдельное сообщение (или сообщения) через вызов `println`.

Когда вы вызываете `performCombat`, как оболочка REPL понимает, какая именно вам нужна? Она проверяет переданные аргументы и находит ту функцию, которая соответствует им. Вызовите в REPL реализацию каждой функции `performCombat`, как показано ниже (не забывайте нажимать Command-Return [Ctrl-Enter] для выполнения кода).

Листинг 4.10. Вызов перегруженных функций (REPL)

```
performCombat()
performCombat("Ulrich")
performCombat("Hildr", true)
```

Вывод будет выглядеть так:

```
You see nothing to fight!
You begin fighting Ulrich.
You begin fighting Hildr. You are blessed with 2X damage!
```

Обратите внимание, что выбор той или иной реализации осуществляется на основе количества аргументов.

Для любознательных: имена функций в обратных кавычках

В Kotlin есть возможность, которая на первый взгляд выглядит немного странно. Она позволяет объявить или вызвать функцию с именем, содержащим пробелы и другие редко используемые символы. Для этого достаточно заключить имя в обратные кавычки ```. Например, можно объявить такую функцию:

```
fun `**~prolly not a good idea!~**`() {  
    ...  
}
```

А дальше можно вызвать ``**~prolly not a good idea!~**``:

```
`**~prolly not a good idea!~**`()
```

Зачем это нужно? К слову, никогда не надо называть функцию ``**~prolly not a good idea!~**``. (Или использовать смайлики. Пожалуйста, относитесь к обратным кавычкам ответственно.) Есть несколько веских причин, оправдывающих имена функций в кавычках.

Прежде всего, эта функция обеспечивает совместимость с Java. Kotlin поддерживает возможность вызова кода Java, C и JavaScript из файлов Kotlin. (Обзор совместимых возможностей Java см. в части VI.) Так как Kotlin и Java имеют разные *зарезервированные ключевые слова* (то есть слова, которые нельзя использовать в качестве имен функций), имена функций в обратных кавычках позволяют избежать потенциальных конфликтов.

Например, представьте метод Java с именем `is` из старого проекта на Java:

```
public static void is() {  
    ...  
}
```

В языке Kotlin `is` — это зарезервированное ключевое слово. (В стандартной библиотеке Kotlin оператор `is` используется для проверки типа экземпляра, о чем вы узнаете в главе 15.) Однако в других языках `is` может быть именем метода. Заключив имя функции в обратные кавычки, можно вызвать метод `is` из Kotlin, например:

```
fun doStuff() {  
    `is`() // Вызов метода Java `is` из Kotlin  
}
```

В этом случае обратные кавычки помогают поддерживать совместимость с методом Java, который в противном случае был бы недоступен из-за своего имени.

Вторая причина — поддержка более выразительных имен функций, которые можно использовать, например, в тестах. К примеру, такое имя:

```
fun `users should be signed out when they click logout`() {  
    // Выполнить тест  
}
```

выглядит более выразительно и лучше читается, чем

```
fun usersShouldBeSignedOutWhenTheyClickLogout() {  
    // Выполнить тест  
}
```

Обратные кавычки позволяют использовать более выразительные имена для тестовых функций, чем при стандартных правилах именования — «верблюжий регистр с первым символом в нижнем регистре».

Подписывайся на самый топовый канал по Kotlin:
<https://t.me/KotlinSenior>

5. Числа

В Kotlin существует множество разнообразных типов для работы с числами и арифметическими вычислениями. Для каждой из двух основных разновидностей чисел — целых и дробных — в Kotlin есть несколько разных типов.

В этой главе вы узнаете, как в Kotlin работать с обеими разновидностями чисел. Проект bounty-board мы трогать не будем; вместо этого мы выполним код в REPL. Впрочем, проект bounty-board стоит держать под рукой, потому что мы вернемся к нему в следующей главе.

Числовые типы

В Kotlin поддерживается несколько числовых типов. Независимо от того, для какой платформы вы пишете код, правила использования этих числовых типов остаются неизменными. В версии Kotlin 1.5 числовые типы делятся на две категории: *со знаком* и *без знака*. Числа со знаком могут представлять положительные и отрицательные числа. Числа без знака могут представлять только положительные числа. Сначала рассмотрим числа со знаком, а с числами без знака познакомимся в разделе «Для любознательных: числа без знака» в конце этой главы.

Кроме наличия знака, у числовых типов Kotlin есть и другие особенности. Числовые типы различаются количеством битов, которое занимают в памяти, и, соответственно, их максимальным и минимальным значениями.

Если вы имели дело с языком Java, эти правила могут показаться очень знакомыми. В Kotlin для числовых типов используются те же правила, что и в Java. Читателей с опытом работы на JavaScript может удивить обилие числовых типов, потому что в JavaScript существует единственный тип `Number`.

Наконец, разработчику, который собирается использовать платформу Kotlin/Native, стоит обратить внимание на число битов, которое выделяется для каждого типа. Числовые типы Kotlin ведут себя идентично и используют одинаковое количество битов независимо от платформы (в отличие от типа `int` языка C, который может использовать разное количество битов в зависимости от того, как компилируется программа).

В табл. 5.1 перечислены некоторые числовые типы в Kotlin, количество битов и максимальное/минимальное значение, поддерживаемое этим типом. (Подробности — далее.)

Таблица 5.1. Часто используемые числовые типы

Тип	Биты	Максимальное значение	Минимальное значение
Byte	8	127	-128
Short	16	32 767	-32 768
Int	32	2 147 483 647	-2 147 483 648
Long	64	9 223 372 036 854 775 807	-9 223 372 036 854 775 808
Float	32	3.4028235E38	1.4E-45
Double	64	1.7976931348623157E308	4.9E-324

Между размером типа в битах (разрядностью) и его максимальным/минимальным значением существует связь. Компьютеры хранят целые числа в двоичной форме с фиксированным числом битов. Бит может принимать только одно из двух значений: 0 или 1.

Для представления числа Kotlin использует ограниченное число битов в зависимости от выбранного числового типа. Крайний левый бит выражает знак (плюс или минус). Остальные биты выражают степени 2, где самый правый бит представляет 2^0 . Чтобы вычислить значение двоичного числа, надо сложить все степени 2 для позиций числа, представленных битом 1.

На рис. 5.1 показан пример представления числа 42 в двоичной системе счисления.

$$\begin{array}{cccccc}
 \boxed{1} & \boxed{0} & \boxed{1} & \boxed{0} & \boxed{1} & \boxed{0} \\
 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0
 \end{array}
 = 2^5 + 2^3 + 2^1 = 32 + 8 + 2 = 42$$

Рис. 5.1. Число 42 в двоичной записи

Тип `Int` содержит 32 бита, поэтому самое большое значение, которое может хранить `Int`, в двоичной форме представлено 31 единицей (учтите, что крайний левый бит представляет знак). Если сложить все степени двойки, то получим наибольшее значение для типа `Int` в Kotlin — 2 147 483 647.

Так как число битов определяет максимальное и минимальное значения, которые может представлять числовой тип, разница между типами — это количество битов, доступных для выражения числа. Тип `Long` использует 64 бита вместо 32, значит, может хранить намного большее значение (2^{63}).

Немного о `Short` и `Byte`: оба типа редко используются для представления традиционных чисел. Они применяются в особых случаях и для поддержки совместимости со старыми программами.

Например, тип `Byte` можно использовать для чтения потока данных из файла или при работе с графикой (цвет пикселя часто выражается тремя байтами: по одному для каждого цвета в RGB). Тип `Short` иногда используется для работы с машинным кодом процессоров, не поддерживающих 32-битные команды. Однако в большинстве случаев для представления целых чисел используется тип `Int`, а если нужно большее допустимое значение — тип `Long`.

Целые числа

В главе 2 мы рассказывали, что целочисленное значение — это число, не имеющее дробной части, то есть целое число, и в Kotlin оно представлено типом `Int`. `Int` хорошо подходит для выражения количественных показателей: уровня игрока, количества очков опыта, остатков пинт меда или числа золотых и серебряных монет у игрока.

Чтобы получить практический опыт использования `Int`, мы воспользуемся REPL для выполнения арифметических операций. Откройте REPL командой `Tools ▶ Kotlin ▶ Kotlin REPL`.

В REPL введите следующую операцию. Как вы думаете, какой результат вы получите?

Листинг 5.1. Выполнение арифметических операций с целыми числами (REPL)

```
2 + 4 * 5
```

Вычислите это выражение нажатием клавиш `Command-Return` (`Ctrl-Enter`). REPL выводит результат 22. Ваше предположение было правильным?

Как показывает этот пример, операторы умножения и деления в Kotlin (`*`, `/` и `%`) обладают более высоким приоритетом по сравнению с операторами сложения и вычитания (`+`, `-`), как и в традиционной арифметике. Это означает, что умножение `4 * 5` будет вычислено до прибавления 2 к произведению.

Если вам нужно задать другой порядок (или вы хотите сделать этот порядок более очевидным), группируйте операции с помощью круглых скобок. Как было показано в главе 3, выражения, заключенные в круглые скобки, вычисляются первыми.

Теперь введите следующую операцию в REPL. Как вы думаете, какой результат будет получен при вычислении этого выражения?

Листинг 5.2. Выполнение целочисленного деления (REPL)

```
9 / 5
```

Вычислите выражение в REPL. Возможно, вы ожидали увидеть результат 1.8, но REPL выведет результат 1. Почему это произошло?

Когда вы делите целое число на другое целое число, результатом всегда будет целое число. Если результат операции целочисленного деления не является целым числом, Kotlin отбрасывает все цифры в дробной части. Аналогичным образом, если вы прикажете REPL вычислить $9 / -5$, результат будет равен -1 .

Результат целочисленного деления всегда округляется до меньшего целого. Округление происходит «молча», поэтому если остаток важен для вашего приложения, при выполнении целочисленного деления необходимо действовать осторожно.

Для вычисления остатка от деления двух целых чисел можно воспользоваться оператором `%`. Например, выражение $9 \% 5$ возвращает результат 4 .

Если вы работаете с числами, имеющими дробную часть, тип `Int` вам не подойдет.

Дробные числа

В Kotlin дробные числа представлены типами `Float` и `Double`. Эти числа также называются числами с плавающей точкой¹, потому что точка — разделитель дробной части может находиться в произвольной позиции, то есть является «плавающей» (в отличие от фиксированной точки) в зависимости от порядка числа.

Тип `Double` обозначает число с плавающей точкой двойной точности. Числа `Double` используют вдвое больше битов, чем обычные числа `Float` (отсюда и название), и позволяют хранить дробные числа с большей точностью.

Числа с плавающей точкой в Kotlin также могут принимать специальные значения для представления бесконечности, отрицательной бесконечности и `NaN` (сокращение от *Not a Number*, то есть «не число»). Эти значения обычно возвращаются при выполнении недействительных или неопределенных операций вроде деления на нуль (возвращается бесконечность или отрицательная бесконечность) или квадратного корня из отрицательного числа (возвращается `NaN`). К этим специальным значениям можно обращаться в виде `Double.POSITIVE_INFINITY` (или `Float.POSITIVE_INFINITY`), `Double.NEGATIVE_INFINITY` (или `Float.NEGATIVE_INFINITY`) и `Double.NaN` (или `Float.NaN`).

Вернитесь к выражению деления $9 / 5$ в REPL. Чтобы в этом выражении использовалось деление с плавающей точкой, необходимо сообщить Kotlin, что это дробные числа, а не целые. Для этого можно включить в запись числа десятичную точку.

Листинг 5.3. Выполнение деления с плавающей точкой (REPL)

```
9.0 / 5.0
```

¹ В российской записи — числами с плавающей запятой. — *Примеч. ред.*

Вычислите это выражение. REPL выведет `kotlin.Double = 1.8`, что больше соответствует человеческим ожиданиям. Обратите внимание: выражение имеет тип `Double`. Kotlin по умолчанию предпочитает использовать `Double`, но вы можете специально потребовать, чтобы значение интерпретировалось как `Float`; для этого добавьте к числам суффикс `f`: `9.0f / 5.0f`.

(Добавляя суффикс `f`, при желании можно опустить `.0` в обоих числах. Собственно, то же выражение можно записать в виде `9f / 5`, потому что Kotlin будет использовать деление с плавающей точкой, если хотя бы один из операндов является числом с плавающей точкой.)

Ранее упоминалось о том, что дробные числа характеризуются «точностью». Чтобы понять, о чем идет речь, введите следующее выражение в REPL.

Листинг 5.4. Погрешность вычислений с плавающей точкой (REPL)

```
0.01f * 5
```

Интуиция подсказывает, что выражение должно вернуть `0.05`. Тем не менее при вычислении этого выражения REPL выведет результат `0.049999997`. Теперь выполним операцию сравнения, чтобы установить, равен ли этот результат `0.05`.

Листинг 5.5. Проверка точного равенства для вычислений с плавающей точкой (REPL)

```
0.01f * 5 == 0.05f
```

При выполнении этой команды REPL выведет результат `false`. Почему?

При работе с целыми числами каждый бит имеет конкретное значение, которое никогда не изменяется. При работе с дробными числами все не так просто. На двоичном уровне значения с плавающей точкой образуются из одного бита для представления знака и двух дополнительных групп битов. Первая группа представляет экспоненту, определяющую порядок числа. Вторая группа определяет значащие цифры в представляемом числе.

Как следствие, числа с плавающей точкой не могут точно представлять каждое число — они являются *аппроксимациями*. Хотя значение `0.05` может быть точно представлено в формате `Float`, для значения `0.01` это невозможно. Ближайшее число, которое может использовать `Float` для представления `0.01`, — `0.009999999776482582`. При умножении `0.01f * 5` потеря точности распространяется на результат, поэтому вы получаете неточный (хотя и весьма близкий!) результат.

Проблему с точностью можно решить несколькими способами.

- *По возможности используйте `Double` вместо `Float`*: использование числа с плавающей точкой с более высокой точностью снижает вероятность возникновения погрешности чисел с плавающей точкой, но за счет более высоких затрат

памяти. Впрочем, учтите, что `Double` еще не позволяет полностью избежать этой проблемы. (Попробуйте вычислить `10.1 - 5.9` в REPL.)

- *Округляйте значения с плавающей точкой*: если вы точно знаете, сколько цифр в дробной части числа с плавающей точкой должно использоваться, можно выполнить соответствующее округление. В Kotlin доступны API, позволяющие выводить дробные значения с заданным количеством цифр в дробной части, а также функция `round`, которая округляет числа до ближайшего целого числа.
- Объединяя функцию `round` с операциями умножения и деления, можно округлить число до заданного количества цифр в дробной части. Например, `round(number * 100) / 100` округляет число до двух знаков в дробной части.
- *Выбирайте другой тип данных с более высокой точностью*: если ваше приложение должно хранить критически важное дробное значение, для которого округление и потеря точности неприемлемы (например, если вы работаете над банковскими программными продуктами), можно перейти на тип данных с расширенными возможностями.
- Иногда можно использовать `Int` для представления типов данных, которые должны рассматриваться как дробные. Например, для денежных средств на банковском счете можно воспользоваться `Int` для хранения значений в центах — вместо типа `Double`, хранящего значение в долларах.

В крайнем случае воспользуйтесь классом `BigDecimal`, доступным при программировании для JVM. Он позволяет намного точнее управлять округлением и выполнением операций и решает проблемы с точностью за счет повышения сложности по сравнению с базовыми числовыми типами. `BigDecimal` также использует больше ресурсов для хранения значений и выполнения арифметических операций по сравнению со значениями с плавающей точкой. (Если ваш код на Kotlin предназначен для iOS или macOS, используйте `Decimal` — эквивалент класса `BigDecimal`.)

Форматирование значений типа Double

Поразмыслите над экономикой игрового мира. Допустим, вы хотите отслеживать количество игровых денег у героя. Для этого в программу можно включить логику, которая выглядит примерно так:

```
val currentBalance = 1120.40
println(currentBalance)
```

Если выполнить этот код, баланс будет выведен в виде `1120.4` без какого-либо дополнительного форматирования.

Впрочем, работать с `1120.4` неизвестных денежных единиц неудобно. Было бы лучше отформатировать баланс в виде, более похожем на денежную сумму.

Например, для Соединенных Штатов баланс выводился бы в виде **\$1,120.40**. Для применения форматирования к `double` можно воспользоваться функцией `format` класса `String`, с указанием обозначения денежной единицы, разделителя групп разрядов и количества знаков в дробной части.

Начнем с количества знаков в дробной части. Выполните следующий код в REPL; обратите особое внимание на использование функции `format`.

Листинг 5.6. Форматирование `double` (REPL)

```
val currentBalance = 1120.40
println("%.2f".format(currentBalance))
```

(Здесь для вызова функции `format` используется *точечный синтаксис*. Он применяется при вызове любых функций, которые включаются как часть определения типа.)

REPL выводит значение **1120.40**, которое читается немного лучше.

При вызове `format` указывается *форматная строка* `"%.2f"`. В форматных строках специальные последовательности символов определяют, как должны форматироваться данные. Конкретная форматная строка из примера сообщает, что число с плавающей точкой должно округляться до двух знаков в дробной части. Форматируемое значение (или значения) передается в аргументе функции `format`.

В форматных строках используются те же обозначения, что и в стандартных форматных строках Java, C/C++, Ruby и многих других языков. За дополнительной информацией о форматных строках обращайтесь к документации Java API по адресу docs.oracle.com/javase/8/docs/api/java/util/Formatter.html.

Чтобы добавить запятую и знак \$, приведите форматную строку к виду `"$%,.2f"`:

Листинг 5.7. Форматирование денежной суммы (REPL)

```
val currentBalance = 1120.40
println("$%,.2f".format(currentBalance))
```

Использование `format` создает некоторые затруднения. Прежде всего, форматирование может усложнить локализацию приложения. Если вы хотите, чтобы баланс выводился с учетом региональных настроек пользователя, то знак \$ следует заменить подходящим символом денежной единицы. Кроме того, во многих странах разделителем дробной части является запятая, а группы разрядов разделяются точками, что усложнит ваши усилия по локализации.

Вторая проблема заключается в том, что (на момент написания книги) функция `format` доступна только для JVM. Если вы ориентируетесь на другие платформы, придется использовать другой способ форматирования чисел.

Для решения обеих проблем можно воспользоваться API форматирования для конкретной платформы. В Java доступен класс `NumberFormat`. Чтобы прийти

к тому же результату с использованием `NumberFormat`, можно использовать код следующего вида:

```
val currentBalance = 1120.40
val formatter = NumberFormat.getCurrencyInstance()
val formattedBalance = formatter.format(currentBalance)
println("Madrigal's life savings: " + formattedBalance)
```

(Чтобы выполнить этот код в REPL, необходимо включить в начало программы строку `import java.text.NumberFormat`, чтобы импортировать класс.)

Сбережения героя автоматически преобразуются в отформатированную строку, наиболее подходящую для пользователя на основании региональных настроек его системы.

Кроме класса для Java, в Android также существует собственный класс `NumberFormat` из пакета `android.icu.text`. Оба класса `NumberFormat` могут использоваться для получения форматов денежных сумм, соответствующих региональным настройкам. Если же ваш код Kotlin предназначен для iOS или macOS, вам будет доступен класс `NSNumberFormatter`. Наконец, на платформе Kotlin/JS для форматирования числовой информации можно воспользоваться классом `Intl.NumberFormat`.

Преобразования числовых типов

Иногда необходимо выполнять преобразования между числами с плавающей точкой и целыми числами. Допустим, вы хотите хранить уровень игрока в виде очков, а не в виде значения уровня, как в `bounty-board`. Код мог бы выглядеть так:

```
var experiencePoints = 460.25
val playerLevel = experiencePoints / 100
```

Как вы думаете, к какому типу будет относиться `playerLevel`? Чтобы найти ответ самостоятельно, введите код прямо в REPL, выберите `playerLevel` и нажмите Control-Shift-P (Ctrl-Shift-P).

Как вы, возможно, догадались, для `playerLevel` будет автоматически определен тип `Double` и при выполнении кода уровню будет присвоено значение `4.6025`. Вероятно, это не то, что вам нужно. Для задания сложности миссии персонажу уровня 4 его уровень должен быть равен 4 независимо от того, сколько у него очков опыта — 400 или 499. Чтобы эта особенность учитывалась в программе, переменная `playerLevel` должна иметь тип `Int`, а не `Double`.

Для проведения такого преобразования вызовите функцию `toInt` для выражения. Опробуйте эту возможность в REPL.

Листинг 5.8. Преобразование Double в Int (REPL)

```
var experiencePoints = 460.25
val playerLevel = (experiencePoints / 100).toInt()
println(playerLevel)
```

При выполнении этого кода REPL выводит результат 4. Когда `Double` преобразуется в `Int` таким способом, Kotlin действует по тем же правилам, которые действуют при выполнении целочисленного деления: дробная часть отсекается, а число округляется в меньшую сторону.

Это поведение иногда называется *потерей точности*. Часть исходных данных теряется, потому что вы запросили целочисленное представление значения `Double`, включающего дробную часть, и целочисленное представление неизбежно становится менее точным.

Иногда усечение нежелательно. В некоторых случаях требуется, чтобы значение `Double` округлялось до ближайшего `Int`. К счастью, в Kotlin есть функция `roundToInt`, которая делает именно это. Введите код из листинга 5.9 в REPL, чтобы увидеть эту функцию в действии.

Листинг 5.9. Округление `Double` до `Int` (REPL)

```
import kotlin.math.roundToInt
val distanceToObjective = 4.6
println("The objective is about " + distanceToObjective.roundToInt() +
      " miles away")
```

При выполнении этого кода будет выведен результат `The objective is about 5 miles away` (цель находится примерно в 5 милях отсюда). Kotlin выводит 5 вместо 5.0; это указывает на то, что значение является целым числом, а не числом с плавающей точкой. Функция округляет и преобразует `Double` в `Int` за один шаг. Таким образом, `roundToInt` может стать удобной альтернативой для `toInt` в зависимости от ваших целей. (Кстати говоря, если вам потребуется преобразовать `Int` в `Double`, воспользуйтесь соответствующей функцией `toDouble`. При вызове этой функции будет возвращено число с плавающей точкой с нулями в дробной части.)

Сейчас вы поработали с числовыми типами Kotlin и узнали, как в Kotlin обрабатываются две основные категории чисел: целые и дробные. Вы также научились выполнять преобразования между разными типами и узнали, какие числа поддерживаются каждым типом. Следующая глава посвящена строкам в языке Kotlin.

Для любознательных: числа без знака

В Kotlin 1.5 числовые типы без знака вошли в число основных возможностей языка. Они очень похожи на числовые типы, которые мы рассмотрели выше, не считая того, что они не позволяют выражать отрицательные значения. В табл. 5.2 перечислены числовые типы без знака, доступные в Kotlin.

У числовых типов без знака и со знаком есть как общие черты, так и различия. Например, числовые типы с плавающей точкой — `Float` и `Double` — не имеют аналогов без знака. Причины мы объясним в конце этого раздела.

Таблица 5.2. Числовые типы без знака в Kotlin

Тип	Биты	Максимальное значение	Минимальное значение
<code>UByte</code>	8	255	0
<code>UShort</code>	16	65 535	0
<code>UInt</code>	32	4 294 967 295	0
<code>ULong</code>	64	18 446 744 073 709 551 615	0

У каждого целого числового типа — `Byte`, `Short`, `Int` и `Long` — существует эквивалентный тип — `UByte`, `UShort`, `UInt` и `ULong`. Числа этих типов со знаком и без знака содержат равное количество битов; например, каждый из типов `Int` и `UInt` состоит из 32 бит. Однако у всех версий без знака минимальное значение равно 0, а максимальное значение намного выше, чем у их аналогов со знаком.

Использовать числовой тип без знака чуть сложнее, чем числовой тип со знаком. Объявление `playerLevel` с типом `UInt` и присваивание значения 5 выглядит так:

```
var playerLevel: UInt = 5.toUInt()
```

Также можно присоединить к числовому литералу префикс `u`, чтобы пометить его как число без знака:

```
var playerLevel: UInt = 5u
```

При желании явную информацию типа (`: UInt`) можно удалить, но вам придется использовать либо префикс числа без знака, либо функцию `toUInt`, чтобы пометить число как значение без знака.

Kotlin не выполняет неявные преобразования между числами со знаком и без. Это также влияет на выполнение операций с экземпляром типа без знака. Например:

```
var playerLevel = 5u
val levelsToAdd = 1
playerLevel += 1.toUInt() // Сложение UInt с UInt разрешено
playerLevel += 1u         // Тоже разрешено (сокращенная запись
                           // строки, приведенной выше)
playerLevel += 1          // Ошибка компиляции: сначала необходимо
                           // преобразовать 1 в UInt
playerLevel += levelsToAdd // Ошибка компиляции: нельзя сложить
                           // Int с UInt
print(playerLevel * 10u)   // Разрешено
print(playerLevel * 10)    // Ошибка компиляции
```

Так как Kotlin не выполняет автоматическое преобразование между типами со знаком и без, вам придется либо объявлять многие переменные в программе с типом без знака, либо использовать функции `toUInt` и `toInt` для преобразования в том или ином направлении по мере надобности.

Числа без знака могут быть полезны в конкретных ситуациях, например, если нужно гарантировать, что число не будет отрицательным, обеспечить выполнение установленных правил для аргументов функций или при работе с платформенно-зависимым кодом, использующим числа без знака. Тем не менее целые числа без знака ненадежны. Небольшая потеря внимания — и в программе могут возникнуть неожиданные ситуации. Например, посмотрим, что произойдет, если использовать тип `UInt` для хранения `playerLevel`, а затем присвоить ему `-1`:

```
val playerLevel = (-1).toUInt()
println(playerLevel)           // Выводит 4294967295
```

Если вычесть `1` из наименьшего значения, допустимого в любом числовом типе, значение переходит к наибольшему возможному значению — происходит так называемое *целочисленное антипереполнение* (integer underflow). В данном случае наименьшее значение, которое может принимать `UInt` или другой тип без знака, — `0`. Вычитание из него `1` приводит к наибольшему возможному `UInt`: `4 294 967 295`. Этот неожиданный результат — следствие одной из ловушек при работе со смешанными типами (со знаком и без знака).

Возможна и обратная ситуация — *целочисленное переполнение* (integer overflow). Обычно оно возникает при работе с очень большими числами: если увеличить `Int.MAX_VALUE` на `1`, произойдет перенос, и вы получите `Int.MIN_VALUE`. Возможно, вас интересует, почему числа без знака ведут себя иначе, чем числовые типы со знаком. Зачем все эти ухищрения? В Kotlin целые числа без знака реализуются не так, как целые числа со знаком. Собственно, они строятся на базе целых чисел со знаком.

Используя целое число без знака, вы фактически приказываете Kotlin использовать целое со знаком, но интерпретировать его биты особым образом. Преимущество такого решения — отсутствие лишних затрат памяти по сравнению с целым числом со знаком, а разница в быстродействии при выполнении операций между ними пренебрежимо мала.

Почему же Kotlin не поддерживает числа с плавающей точкой без знака? Числа без знака представляют собой альтернативные интерпретации своих разновидностей со знаком. И хотя теоретически возможна такая альтернативная интерпретация значения с плавающей точкой с изменением смысла знакового бита, такое решение в высшей степени непривычно, и Kotlin такую возможность не поддерживает.

Из-за сложностей с выполнением арифметических операций с плавающей точкой компьютеры оснащаются специальными аппаратными компонентами для

эффективного выполнения этих операций. Если вы захотите изменить смысл знакового бита, это не позволит использовать аппаратные ускорители, и быстроедействие приложения пострадает. Чтобы избежать подобных проблем, практически ни один язык с числами без знака не поддерживает дробные значения без знака.

Так как типы без знака реализованы в Kotlin на базе типов со знаком, обычно к ним относятся как к «второсортным» типам. Типы со знаком используются несравнимо чаще типов без знака. Кроме того, есть некоторые странности в реализации (например, IntelliJ иногда интерпретирует `UInt` как `Int` при выводе результатов в REPL).

Где использовать типы без знака и стоит ли использовать их вообще — ваше дело. Мы отказались от их использования в `bounty-board`, но кто-то может сказать, что некоторые переменные (например, `playerLevel`) следовало бы объявить без знака, потому что они не бывают отрицательными. Попробуйте сами и посмотрите, насколько они, на ваш взгляд, подходят для вашего кода.

Для любознательных: манипуляции с битами

Ранее мы уже говорили, что числа на компьютере представлены в двоичной системе. Двоичное представление числа можно вывести в любой момент. Например, получить двоичное представление целого числа 42 можно так:

```
42.toString(radix = 2)
"101010"
```

(Параметр `radix` задает основание системы счисления при выводе числа. Значение 2 обозначает основание 2, то есть двоичную систему счисления. По умолчанию значение `radix` равно 10, то есть числа выводятся в десятичной системе. Другое популярное значение `radix` — 16 — выводит шестнадцатеричную строку.)

В Kotlin имеются функции для выполнения операций с двоичными представлениями, которые называются поразрядными операциями, включая уже, возможно, знакомые вам операции из других языков, таких как Java, C и JavaScript. В табл. 5.3 перечислены часто применяемые в Kotlin двоичные операции.

Таблица 5.3. Двоичные операции

Функция	Описание	Пример
<code>shl(bitcount)</code>	Сдвиг влево на указанное число разрядов	<code>42.shl(2)</code> <code>10101000</code>
<code>shr(bitcount)</code>	Сдвиг вправо на указанное число разрядов	<code>42.shr(2)</code> <code>1010</code>

Функция	Описание	Пример
<code>inv()</code>	Инверсия битов	<code>42.inv()</code> <i>111111111111111111111111111111010101</i>
<code>xor(number)</code>	Выполняет логическую операцию «исключающее ИЛИ» с битами и для каждой позиции возвращает 1, если бит в этой позиции в одном числе равен 1, а в другом 0	<code>42.xor(33)</code> <i>001011</i>
<code>and(number)</code>	Выполняет логическую операцию «И» с битами и для каждой позиции возвращает 1, только если оба числа в этой позиции имеют бит, равный 1	<code>42.and(10)</code> <i>1010</i>

6. Строки

В программировании текстовые данные представлены *строками* — упорядоченными последовательностями символов. Вы уже использовали строки для вывода сообщений в bounty-board:

```
"The hero announces her presence to the world."
```

В этой главе вы узнаете, что еще можно делать со строками. Мы доработаем проект bounty-board и улучшим форматирование строк, а также обновим программу, чтобы уровень игрока вводился с консоли (а не фиксировался в коде программы).

Интерполяция строк

В нашем проекте bounty-board у героя есть имя. И было бы лучше использовать имя вместо безличного «нашего героя». В программе определяется константа `HERO_NAME`; остается лишь добавить ее значение к строке. Для решения этой задачи можно воспользоваться *конкатенацией строк*.

Чтобы увидеть, как это делается, обновим приветственное сообщение в `main`:

Листинг 6.1. Конкатенация строк (Main.kt)

```
const val HERO_NAME = "Madrigal"
var playerLevel = 5

fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME + " announces her presence to the world.")
    println(playerLevel)
    ...
}
...
```

Оператор `+` можно использовать для конкатенации (соединения) двух строковых значений. В данном случае конкатенация объединяет строковое значение `HERO_NAME` и строковый литерал `" announces her presence to the world."`. Запустите bounty-board, чтобы увидеть результат:

```
Madrigal announces her presence to the world.
5
...
```

Получилось! Однако у Kotlin в запасе есть еще один трюк, который позволяет добиться того же результата посредством более компактной записи. Обновите приветственное сообщение так, как показано в листинге 6.2.

Листинг 6.2. Использование интерполяции строк (Main.kt)

```
const val HERO_NAME = "Madrigal"
var playerLevel = 5
fun main() {
    println(HERO_NAME + " announces her presence to the world.")
    println("$HERO_NAME announces her presence to the world.")
    println(playerLevel)
    ...
}
```

Перед вами пример *интерполяции строк*. Знак \$ обозначает подставляемое значение. *Строковые шаблоны* — строки с одним или несколькими подставляемыми значениями. Во время выполнения Kotlin заменяет их фактическими значениями.

Когда за \$ непосредственно следует имя переменной (в данном случае HERO_NAME), вы сообщаете Kotlin, что хотите *интерполировать*, то есть подставить, значение переменной в строку. Снова выполните код и убедитесь, что результат не изменился.

Шаблоны позволяют создавать код, который лучше читается и занимает меньше места по сравнению с кодом с конкатенацией строк. Чтобы понять, о чем идет речь, сравните следующие две команды, которые выводят одинаковые сообщения в консоль:

```
println(HERO_NAME + " is at level " + playerLevel + ".")
println("$HERO_NAME is at level $playerLevel.")
```

Вам не кажется, что второй вариант читается проще? Мы тоже так думаем, поэтому обычно предпочитаем именно этот синтаксис.

Выделите немного времени и внесите изменения в остальной код bounty-board, используя интерполяцию строк.

Листинг 6.3. Другие примеры использования строковых шаблонов

```
const val HERO_NAME = "Madrigal"
var playerLevel = 5

fun main() {
    println("$HERO_NAME announces her presence to the world.")
    println(playerLevel)

    readBountyBoard()

    println("Time passes...")
}
```



```
println("The hero returns from her quest.")
println("$HERO_NAME returns from her quest.")

playerLevel += 1
println(playerLevel)
readBountyBoard()
}

private fun readBountyBoard() {
    println("The hero approaches the bounty board. It reads:")
    println("$HERO_NAME approaches the bounty board. It reads:")
    println(observeOnQuest(playerLevel))
}
...
```

Запустите программу и убедитесь, что вывод соответствует вашим ожиданиям.

Шаблоны также можно использовать с более сложными выражениями. Предположим, вы хотите заключить название миссии в кавычки и напечатать с отступом для выразительности. Для этого обновите вызов `println`, как показано в листинге 6.4. Сначала введите код, а потом мы объясним синтаксис.

Листинг 6.4. Использование сложного строкового шаблона (Main.kt)

```
...
private fun readBountyBoard() {
    println("The hero approaches the bounty board. It reads:")
    println(observeOnQuest(playerLevel))
    println("\t\"${observeOnQuest(playerLevel)}\"")
}
...
```

В этом фрагменте представлены две новые возможности. Начнем с интерполяции строк. Для интерполяции значения, возвращаемого функцией `observeOnQuest` в шаблон, используется запись `${observeOnQuest(playerLevel)}`. Здесь за символом `$`, обозначающим строковый шаблон, следует пара фигурных скобок `{}`.

Полный синтаксис строкового шаблона имеет вид ``${ваше-выражение}``. В фигурных скобках Kotlin позволяет записать любое выражение на ваше усмотрение. Разрывы строк, комментарии, выражения `if/else`, выражения `when` — все это можно использовать в фигурных скобках.

Почему же для `$HERO_NAME` фигурные скобки не нужны? Если вы ограничиваетесь чтением переменной, Kotlin позволяет опустить фигурные скобки. С любыми другими выражениями скобки необходимы.

Новая строка также содержит несколько обратных косых черт (символ `\`). Для чего они нужны?

Как вы уже видели, двойная кавычка (`"`) обозначает начало и конец строкового литерала: `"Madrigal"`. Если включить в код строку вида `"Madrigal proclaims,`

"Hello, world!""", компилятор читает вторую кавычку и считает ее завершением строки: "Madrigal proclaims, ".

Нужно как-то сообщить компилятору, что кавычка является частью строки, а не признаком ее завершения. Для этого символ " нужно *экранировать*, — то есть сообщить компилятору, что он должен рассматриваться как обычный текст, а не как часть синтаксиса. Экранирование как раз и осуществляется при помощи символа \.

Также можно воспользоваться другим символом экранирования — \t. Последовательность \t приказывает Kotlin вставить символ табуляции. В Kotlin существует много разных экранированных последовательностей, имеющих особый смысл. В табл. 6.1 перечислены экранированные последовательности (состоящие из символа \ и экранируемого символа) и их назначение.

Таблица 6.1. Экранированные последовательности

Экранированная последовательность	Описание	Экранированная последовательность	Описание
\t	Символ табуляции	\'	Символ одиночной кавычки
\b	Символ забоя (backspace)	\\	Обратная косая черта
\n	Символ перевода строки	\\$	Символ доллара
\r	Символ возврата каретки (return)	\u	Символ Юникода
\"	Символ двойной кавычки		

Запустите bounty-board, чтобы увидеть результат применения новой шаблонной строки:

```
Madrigal announces her presence to the world.
5
Madrigal approaches the bounty board. It reads:
    "Convince the barbarians to call off their invasion."
Time passes...
Madrigal returns from her quest.
6
Madrigal approaches the bounty board. It reads:
    "Locate the enchanted sword."
```

Необработанные строки

Если строка содержит слишком много экранированных символов, то разобраться в ее назначении довольно сложно. Даже если вы хорошо понимаете каждую часть строки "\t\"\${obtainQuest(playerLevel)}\"", то чтобы уложить все в голове,

потребуется какое-то время. К счастью, в Kotlin для таких случаев существует более подходящий инструмент — *необработанные строки* (raw strings). Они позволяют использовать специальные символы без экранирования.

Необработанные строки могут содержать символы новой строки, кавычки и другие знаки, которые обычно должны экранироваться. Более того, необработанные строки вообще не поддерживают экранирование. Как вы вскоре увидите, при использовании таких строк все еще можно пользоваться строковыми шаблонами. Необработанная строка начинается с трех кавычек (""") и завершается также серией из трех кавычек.

Давайте рассмотрим пример использования необработанных строк, для этого обновим функцию `readBountyBoard`, чтобы она ограничивалась всего одним вызовом `println`.

Листинг 6.5. Использование необработанных строк (Main.kt)

```
...
private fun readBountyBoard() {
    println("$HERO_NAME approaches the bounty board. It reads:")
    println(observeOn(playerLevel))
    println(
        """
        |$HERO_NAME approaches the bounty board. It reads:
        |   "${observeOn(playerLevel)}"
        """).trimMargin()
    )
}
...
```

Вторая вертикальная черта (|) отделена от кавычки (") четырьмя пробелами, но допустимо любое другое количество пробелов.

Элегантность необработанных строк заключается в том, что они сохраняют пропуски. Все разрывы строк, все отступы в начале и конце строки будут сохранены при компиляции кода.

Три кавычки ("""), которые открывают и закрывают необработанную строку, необязательно надо размещать в отдельной строке. Но многие программисты ради удобочитаемости поступают именно так — см. листинг 6.5. Однако короткие необработанные строки часто размещаются в одной отдельной строке, например:

```
""""Welcome, Madrigal!" the mayor proclaimed.""".
```

Вертикальная черта (|) определяет начало каждой внутренней (логической) строки. Функция `trimMargin`, вызываемая для необработанных строк, ищет строки, начинающиеся с вертикальной черты, и удаляет все пропуски слева от нее. Вертикальная черта также удаляется из итоговой строки.

Если опустить вертикальную черту и вызов `trimMargin` для необработанной строки, текст будет иметь точно такие же отступы, как в вашем коде. В таком

случае можно удалить все отступы из необработанных строк, но это затруднит чтение.

В Kotlin также существует функция `trimIndent`, которая работает аналогично вертикальной черте (`|`) при условии, что хотя бы одна внутренняя строка выровнена по левому краю. При вызове функция определяет величину начальных пропусков, общих для всех строк, и удаляет их из текста. Таким образом, необработанную строку можно записать с `trimIndent` следующим образом:

```
private fun readBountyBoard() {
    println(
        """
        $HERO_NAME approaches the bounty board. It reads:
        "${obtainQuest(playerLevel)}"
        """.trimIndent()
    )
}
```

Мы выбрали более наглядную схему записи с `|` и `trimMargin`, но вы можете использовать любой вариант на свое усмотрение.

Снова запустите `bounty-board`. Результат будет выглядеть так же, как прежде, но посмотрев внимательно, вы заметите, что символ табуляции заменен четырьмя пробелами.

Необработанные строки серьезно упрощают чтение текста по сравнению со строками, содержащими многочисленные экранированные символы. Они позволяют сразу увидеть, как будет выглядеть текст.

Чтение ввода с консоли

Наши строки отформатированы и содержат имя героя; пора переходить к следующей задаче.

До сих пор уровень опыта героя был жестко зафиксирован в коде приложения. Его можно изменить, только откорректировав код и перекомпилировав приложение. Это означает, что игрок всегда будет получать одну и ту же миссию, пока игра не будет перекомпилирована. Решение, мягко говоря, не идеальное.

Чтобы преодолеть этот недостаток, нам понадобится читать ввод от пользователя с консоли. Обновите функцию `main` вызовом функции `readLine`, которая делает именно это.

Листинг 6.6. Чтение текста с консоли (Main.kt)

```
...
fun main() {
    println("$HERO_NAME announces her presence to the world.")
    println("What level is $HERO_NAME?")
}
```

```
val input = readLine()
println("$HERO_NAME's level is $input.")
println(playerLevel)

readBountyBoard()
...
}
...
```

Запустите `bounty-board`. Вы увидите, что программа приостанавливается после вывода сообщения `What level is Madrigal?` (на каком уровне находится Мадригал?). В окне консоли IntelliJ введите число 1 и нажмите Return. Ваша программа выведет сообщение `Madrigal's level is 1.` и продолжит выполнение с этой точки. Полный результат должен выглядеть так:

```
Madrigal announces her presence to the world.
What level is Madrigal?
1
Madrigal's level is 1.
Madrigal approaches the bounty board. It reads:
    "Convince the barbarians to call off their invasion."
Time passes...
Madrigal returns from her quest.
6
Madrigal approaches the bounty board. It reads:
    "Locate the enchanted sword."
```

Возможно, вы уже заметили наличие некоторых проблем с логикой в коде. Вскоре мы займемся их исправлением.

Функция `readLine`, по сути, противоположна `println`. Она возвращает `String` с одной строкой текста, которая была введена с консоли. Функцию применяют для получения пользовательского ввода в приложениях командной строки.

Преобразование строк в числа

Программа считывает уровень игрока, но из вывода видно, что уровень еще не используется для определения доступных миссий. На уровне 1 игроку должно быть поручено более простое задание, чем переговоры с варварами.

Чтобы использовать ввод пользователя, необходимо присвоить его `var`-переменной `playerLevel`. Но прежде чем вносить изменения, потратим немного времени на анализ только что введенного кода.

Переместите курсор к объявлению `input` и нажмите Control-Shift-P (Ctrl-Shift-P), чтобы увидеть, какой тип был автоматически определен Kotlin для новой переменной. Открывшееся временное окно сообщает, что это тип `String?`. В главе 7 мы расскажем, что означает вопросительный знак, а пока обратите внимание на следующее: `input` — строка, а `playerLevel` — целое число. Как пре-

образовать строку в число, чтобы результат `readLine` можно было присвоить `playerLevel`?

К счастью, тип `Kotlin String` содержит функцию с именем `toInt`, которая может выполнить это преобразование автоматически. Обновите код `bounty-board`, чтобы выполнить это преобразование и присвоить полученное значение `playerLevel`.

Листинг 6.7. Преобразование `String` в `Int` (`Main.kt`)

```
const val HERO_NAME = "Madrigal"
var playerLevel = 5
var playerLevel = 0

fun main() {
    println("$HERO_NAME announces her presence to the world.")
    println("What level is $HERO_NAME?")
    val input = readLine()
    println("$HERO_NAME's level is $input.")
    playerLevel = readLine()!!.toInt()
    println("$HERO_NAME's level is $playerLevel.")

    readBountyBoard()
    ...
}
```

При вызове `toInt` вся строка анализируется и интерпретируется как число. (Заметили два восклицательных знака в коде? Это связано с защитой от неопределенных значений в `Kotlin`, о которой мы подробнее расскажем в главе 7.)

Снова запустите `bounty-board`. Когда вам будет предложено ввести уровень игрока, введите 1. Результат должен выглядеть так:

```
Madrigal announces her presence to the world.
What level is Madrigal?
1
Madrigal's level is 1.
Madrigal approaches the bounty board. It reads:
    "Meet Mr. Bubbles in the land of soft things."
Time passes...
Madrigal returns from her quest
2
Madrigal approaches the bounty board. It reads:
    "Convince the barbarians to call off their invasion."
```

В этой точке приложение работает так, как предполагалось. Оно запрашивает у игрока уровень и выводит соответствующие миссии. В ходе дальнейшей работы с `bounty-board` мы доработаем код, и прежде всего добавим проверку ввода.

Снова запустите bounty-board, но на этот раз по запросу уровня введите `one`. В программе происходит фатальный сбой, а результат выглядит так:

```
Madrigal announces her presence to the world.
What level is Madrigal?
one
Exception in thread "main" java.lang.NumberFormatException:
    For input string: "one"
    at java.lang.NumberFormatException.forInputString
        (NumberFormatException.java:68)
    at java.lang.Integer.parseInt(Integer.java:652)
    at java.lang.Integer.parseInt(Integer.java:770)
    at MainKt.main(Main.kt:7)
    at MainKt.main(Main.kt)
```

Запустите bounty-board еще раз, на этот раз введите значение `1.0`. В программе снова происходит сбой и выводится похожее сообщение об ошибке:

```
Madrigal announces her presence to the world.
What level is Madrigal?
1.0
Exception in thread "main" java.lang.NumberFormatException:
    For input string: "1.0"
    at java.lang.NumberFormatException.forInputString
        (NumberFormatException.java:68)
    at java.lang.Integer.parseInt(Integer.java:652)
    at java.lang.Integer.parseInt(Integer.java:770)
    at MainKt.main(Main.kt:7)
    at MainKt.main(Main.kt)
```

В обоих случаях в программе возникает проблема: она не может преобразовать входную строку в `Int`. В главе 7 вы больше узнаете о распространении ошибок. А пока подумайте, почему Kotlin не удается преобразовать эти входные значения.

В случае с `one` функция `toInt` не может преобразовать значение, потому что оно не является числовым. Если вам нужно разобрать число, записанное в текстовом виде, вам придется реализовать эту функциональность самостоятельно или воспользоваться другой библиотекой — обе темы выходят за рамки книги.

Чтобы понять, почему значение `1.0` не может быть преобразовано в `Int`, вспомните материал главы 5: `Int` — целое число, которое не имеет дробной части. Следовательно, по аналогии с тем, как `1.0` невозможно просто присвоить `Int`, вы не можете просто интерпретировать строку `"1.0"` как `Int`. Вам придется использовать функцию `toFloat` или `toDouble`, которая понимает, как поступить с точкой-разделителем. (Также существуют аналогичные функции `toLong` и `toBigDecimal` на случай, если потребуется очень большое число или числа должны иметь очень высокую точность.)

Регулярные выражения

В общем случае входные данные желательно проверять перед выполнением «рискованных» операций, которые могут привести к аварийному завершению программы. Существует несколько способов предотвратить фатальный сбой приложения при передаче недопустимого ввода. Кроме того, есть различные способы обработки ошибок после их возникновения — подробнее об этом в главе 7.

А пока для предотвращения сбоя можно проверить число, введенное пользователем, прежде чем пытаться вызывать для него `toInt`. А именно нужно убедиться в том, что строка, возвращенная `readLine`, содержит только цифры.

В Kotlin есть ряд инструментов для проверки пользовательского ввода на правильность. Один из них — *регулярные выражения* — весьма часто применяется для разбора и работы со строками.

Регулярное выражение (regular expression) представляет собой последовательность символов, описывающую некий шаблон. Этот шаблон можно применить к тексту для поиска совпадений. В приложении `bounty-board` при помощи регулярного выражения проверим, состоит ли ввод только из цифр, как показано в листинге 6.8.

Листинг 6.8. Применение регулярных выражений (Main.kt)

```
...
fun main() {
    println("$HERO_NAME announces her presence to the world.")
    println("What level is $HERO_NAME?")
    playerLevel = readLine()!!.toInt()
    val playerLevelInput = readLine()!!
    playerLevel = if (playerLevelInput.matches("\\d+".toRegex())) {
        playerLevelInput.toInt()
    } else {
        1
    }
    println("$HERO_NAME's level is $playerLevel.")

    readBountyBoard()
    ...
}
```

Здесь используются две новые функции: `matches` и `toRegex`. Функция `toRegex` преобразует строку в экземпляр типа `Regex`, который должен передаваться функции `matches`. Строка, передаваемая `toRegex`, содержит регулярное выражение `\d+`, которое означает «одна или более цифр». В этом регулярном выражении `\d` обозначает любую цифру от 0 до 9, а знак `+` обозначает одно или несколько вхождений символа слева от него.

Для определения регулярного выражения используется необработанная строка, ограниченная тройными кавычками — `"""`. Делать так необязательно, но необработанные строки упрощают объявление регулярных выражений с использованием зарезервированных символов, таких как `\`, `$`, `"` и т. д. Напомним, что необработанные строки не поддерживают экранированные последовательности. Это означает, что `\` в необработанной строке интерпретируется как символ «обратная косая черта», а не как начало экранированной последовательности.

Если бы здесь использовалась обычная строка, то вам пришлось бы экранировать символ `\` для передачи функции `toRegex`: `"\\d+"`. Такая запись работает, но синтаксис регулярного выражения становится менее понятным.

Функция `matches` проверяет, все ли строки совпадают с заданным регулярным выражением. (Также можно создавать регулярные выражения, совпадающие с подмножеством символов строки, но здесь это не требуется.) Итак, подытожим: условие `if (playerLevelInput.matches("""\d+""").toRegex())` проверяет, что все символы в `playerLevelInput` являются цифрами.

Несколько раз запустите `bounty-board`. Сначала введите уровень 5, а затем уровень 8, чтобы убедиться в том, что `toInt` правильно вызывается для допустимых входных значений. Затем попробуйте вводить недопустимые значения, например `seven`, `3.0` или `-5`. Программа должна работать без ошибок, и при вводе недопустимого значения по умолчанию должен задаваться уровень 1.

Тип `Regex` доступен для всех платформ, поддерживаемых Kotlin. Дополнительную информацию о поддерживаемом синтаксисе регулярных выражений можно найти по адресу docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html. Пользователям Kotlin/JS стоит обратиться на developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions/Cheatsheet.

Если целевой платформой является JVM, то во внутренней реализации Kotlin тип `Regex` использует класс `Java Pattern`. При использовании Kotlin/JS тип `Regex` использует встроенный тип `RegExp`. Для пользователей Kotlin/Native в поставку Kotlin включается собственная внутренняя реализация поиска по шаблону. Она очень похожа на класс `Java Pattern`, поэтому дополнительную информацию об определении шаблонов стоит искать в документации `Java Pattern`.

Операции со строками

Оказывается, произносить вслух имя темного властелина — `Nogartse` — запрещено (легенда гласит, что это может пробудить его). К счастью, тип `String` содержит много функций, позволяющих изменять хранимый текст во время выполнения программы. Некоторые функции также используют регулярные выражения для определения той части строки, которая должна быть изменена при вызове.

В частности, функция `replace` позволяет удалить запретное имя и избежать печальной судьбы. Включите его в реализацию `readBountyBoard`.

Листинг 6.9. Вызов `replace` (Main.kt)

```
private fun readBountyBoard() {
    println(
        """
        |$HERO_NAME approaches the bounty board. It reads:
        | "${obtainQuest(playerLevel).replace("Nogartse", "xxxxxxx")}"
        |""".trimMargin()
    )
}
```

`replace` получает два аргумента. Первый выбирает заменяемую часть строки. Второй указывает, какое значение должно заменить выбранные части. Вызов `replace` находит все совпадения для заданного критерия выбора и заменяет их новой строкой.

Первый аргумент, который передается функции `replace`, — `"Nogartse"`. Функция `replace` ищет все точные совпадения последовательности символов `"Nogartse"` для замены. Также можно использовать параметр `Regex`, если вы хотите использовать шаблон, который невозможно определить в виде простой строки. Например, можно проверить версии `Nogartse` в верхнем и нижнем регистре с использованием параметра `"[Nn]ogartse".toRegex()`.

Второй аргумент при вызове `replace` — строка `"xxxxxxx"`. Она заменяет имя `Nogartse` несколькими символами `x`, по числу букв в имени. Содержимое строки замены никак не ограничено. Если вы захотите полностью удалить имя `Nogartse` из всех записей, можно просто передать пустую строку (`"`) во втором параметре.

С этими двумя аргументами наш вызов `replace` заменяет имя `Nogartse` символами `x`.

Снова запустите `bounty-board`. На этот раз по запросу введите `8`. Программа выводит описание миссии, которое можно, ничего не опасаясь, прочитать вслух:

```
Madrigal announces her presence to the world.
What level is Madrigal?
8
Madrigal's level is 8.
Madrigal approaches the bounty board. It reads:
    "Defeat xxxxxxxx, bringer of death and eater of worlds."
Time passes...
Madrigal returns from her quest.
9
Madrigal approaches the bounty board. It reads:
    "There are no quests right now."
```

Строки неизменяемы

Стоит пояснить, что значит заменить символы при вызове `replace`. Преобразующие функции (такие, как `replace`) на самом деле не изменяют исходную строку, а возвращают новую с внесенным изменением. Чтобы понять, о чем идет речь, выполните следующий фрагмент кода в REPL.

Листинг 6.10. Неизменяемость строк (REPL)

```
val finalBoss = "Nogartse"
println(finalBoss)                // Выводит "Nogartse"
val replaced = input.replace("Nogartse", "*****")
println(finalBoss)                // Выводит "Nogartse"
println(replaced)                 // Выводит "*****"
```

Все строки Kotlin, независимо от того, определяются они с `var` или `val`, на самом деле неизменяемые (как в Java и JavaScript). Хотя переменным, содержащим значение `String`, можно присвоить другое значение (для `var`), сам экземпляр измениться не может. Любая функция, которая вроде бы меняет значение строки (как `replace`), на самом деле создает новую строку с внесенными изменениями.

Сравнение строк

Вспомните, как функция `obtainQuest`, которая была частью логики `canTalkToBarbarians`, проверяла, является ли игрок варваром.

```
private fun obtainQuest(
    playerLevel: Int,
    hasAngeredBarbarians: Boolean = false,
    hasBefriendedBarbarians: Boolean = true,
    playerClass: String = "paladin"
): String = when (playerLevel) {
    1 -> "Meet Mr. Bubbles in the land of soft things."
    in 2..5 -> {
        val canTalkToBarbarians = !hasAngeredBarbarians &&
            (hasBefriendedBarbarians || playerClass == "barbarian")
        ...
    }
    ...
}
```

В этом коде проверяется *структурное равенство* `playerClass` и `"barbarian"`, для чего используется оператор структурного равенства `==`. Применительно к строкам он проверяет, содержат ли они одинаковый текст. (Оператор также можно применять к числовым значениям; тогда он проверяет значения на равенство.)

Есть еще один способ удостовериться в равенстве двух переменных — использовать *равенство ссылок*, которое определяет, хранят ли две переменные ссылку на один и тот же экземпляр типа. Другими словами, проверяет тот факт, что две переменные указывают на один и тот же объект в памяти. Равенство ссылок проверяется *оператором ссылочного равенства* `===`.

Сравнение ссылок не всегда дает нужный результат. Обычно нам не так важно, являются ли строки одинаковыми или разными экземплярами, — важно то, что они содержат (или не содержат) одинаковые символы в одинаковом порядке (то есть имеют одинаковую *структуру*). Равенство ссылок зависит от реализации исполнительной среды, и поведение оператора ссылочного равенства может отличаться от желаемого.

Если вы знакомы с Java или языком на базе C, то знаете, что поведение оператора `==` при сравнении строк может отличаться от ожидаемого, потому что в Java оператор `==` используется для сравнения ссылок. Низкоуровневый язык на базе C в такой ситуации также проверяет целостность ссылок, так как код обычно сравнивает значения указателей.

При проверке структурного равенства оператором `==` код Kotlin компилируется в вызов функции `equals`. Каждый тип в Kotlin поддерживает функцию `equals`. Эта функция ведет себя так же, как метод Java `equals` (а на самом деле *является* им). Подробнее об этой функции — в главе 16.

Итак, в этой главе вы научились работать со строками в Kotlin — узнали, как объявлять строки со строковыми шаблонами и необработанными строками и как получать текст от пользователя. Также вы получили представление о некоторых методах, которые применяются для операций со строками и преобразования их в более полезные значения.

В следующей главе мы дополним приложение bounty-board фрагментами для проверки ввода и форматирования вывода. Для этого мы применим средства проверки неопределенных значений в Kotlin и обработки исключений.

Для любознательных: Юникод

Как вы уже знаете, строка представляет собой упорядоченную последовательность символов, а символы — это экземпляры типа `Char`. Следует уточнить, что `Char` — это *символ Юникода*. Система кодирования Юникод (Unicode) разрабатывалась для поддержки «обмена, обработки и отображения письменных текстов на разных языках и из разных технических областей современного мира» (unicode.org).

Это означает, что символы в строке могут быть выбраны из широкого набора из 143 859 символов (количество растёт), включая символы алфавита любого языка в мире, значков, глифов, эмодзи и т. д.

Объявить символ можно двумя способами. В обоих случаях его определение заключается в одинарные кавычки. Для символов, которые можно ввести с клавиатуры, проще всего указать сам символ в одинарных кавычках:

```
val capitalA: Char = 'A'
```

Но не все 143 859 символов можно ввести с клавиатуры. Вторым способом представления символа — использование кода символа в Юникоде с экранированной последовательностью `\u`:

```
val unicodeCapitalA: Char = '\u0041'
```

Для буквы «А» на клавиатуре есть клавиша, но для символа  клавиши нет. Чтобы представить его в программе, придется использовать код символа в одинарных кавычках. Чтобы опробовать эту возможность, откройте REPL и выполните код из листинга 6.11.

Листинг 6.11. Ом... (REPL)

```
val omSymbol = '\u0950'  
print(omSymbol)
```

На консоль выводится символ .

7. Null-безопасность и исключения

Некоторым элементам в Kotlin может присваиваться специальное значение *null* — оно показывает, что значение элемента не существует. Во многих языках программирования *null* — частая причина сбоев и ошибок, потому что несуществующая величина не может выполнить какое-либо действие. В Kotlin реализованы некоторые языковые средства, которые заставляют программиста учитывать возможность *null* в программах, что помогает избежать упомянутых ошибок.

В этой главе вы узнаете, почему *null* вызывает сбои, как Kotlin защищается от *null* по умолчанию во время компиляции и как безопасно работать с *null*-значениями. Также вы увидите, как в Kotlin обрабатывать *исключения*, которые показывают, что в программе что-то пошло не так.

Для демонстрации этих проблем мы снова используем проект *bounty-board*. Мы доработаем логику назначения миссии: смоделируем ситуацию, когда миссия, которую можно назначить игроку, отсутствует.

Допустимость *null*

Как упоминалось ранее, одним элементам в Kotlin может быть присвоено значение *null*, а другим нет. Первые называются допускающими *null* (*nullable*), а вторые — не допускающими *null* (*non-nullable*).

Не путайте *null* с нулем или командой, не возвращающей значение. Например, для нашей фэнтезийной игры в приложении *bounty-board* мы реализуем привычную «доску поручений» со списком миссий. Доска, на которой нет ни одной миссии (иначе говоря, со значением *null*), — не то же самое, что доска с сообщением «There is no quests right now» («Миссий нет»). Вроде бы похоже, но путать их не следует. *null* означает, что здесь буквально ничего нет: ни миссий, ни сообщений — ничего. «Миссий нет» означает, что на «доске поручений» присутствует хотя бы это сообщение.

Точно так же *null* не следует путать с пустой строкой (*""*). Пустая строка может означать, что на «доске поручений» висит чистый, незаполненный листок бумаги. Значение *null* означает, что на доске нет ничего — даже этого пустого листка.

Числовые типы в Kotlin также могут допускать *null*, и в этом случае значения *null* и *0* также имеют разный смысл. Если у вас есть значение *Int*, отслеживающее количество доступных миссий, значение *0* показывает их отсутствие. А вот

null свидетельствует, что переменная вообще не содержит значения, то есть его невозможно увеличить или выполнить другую арифметическую операцию, пока переменной не будет присвоено целое значение.

Что это означает для bounty-board? Чтобы функция `obtainQuest` могла выразить, что на «доске поручений» может не быть ни одной миссии, мы обновим ее, и она будет возвращать строку, допускающую null.

Конечный результат будет таким же: если на «доске поручений» нет ни одной миссии, выводится сообщение о том, что «доска поручений» пуста. Зачем нам такое изменение? Допустимость null в таких ситуациях решает определенную задачу. Она заставляет разработчика учитывать граничные случаи в коде — как при чтении с пустой «доски поручений».

Допустим, вы хотите добавить в приложение функцию для вывода текущей миссии. Возможно, вы решите реализовать ее посредством `obtainQuest`. Если не учитывать допустимость null, можно забыть о таких граничных случаях, как пустая доска, и случайно вывести сообщение "Madrigal's current quest is: There are no quests right now.". Однако с null в Kotlin вы скорее вспомните о граничном случае и выведете сообщение "Madrigal does not have a quest now."

И хотя вы вряд ли будете реализовывать такое напоминание, все равно полезно спрашивать себя, должны ли ваши типы допускать null вместо значения 0 или пустой строки.

Обновите функцию `obtainQuest`, чтобы она возвращала значение null в ветви `else`, как показано в листинге 7.1. (После внесения этого изменения компилятор выдает сообщение об ошибке; мы обсудим его после того, как это произойдет.)

Листинг 7.1. Возвращение null (Main.kt)

```
...
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
    hasAngeredBarbarians: Boolean = false
): String = when (playerLevel) {
    1 -> "Meet Mr. Bubbles in the land of soft things."
    ...
    8 -> "Defeat Nogartse, bringer of death and eater of worlds."
    else -> "There are no quests right now."
    else -> null
}
```

Еще до запуска кода IntelliJ предупредит красным подчеркиванием, что здесь что-то не так. Запустите код, и вы получите следующее сообщение:

```
Null can not be a value of a non-null type String
```

Kotlin не позволяет вернуть значение `null` из функции `obtainQuest`, потому что функция имеет тип возвращаемого значения, не допускающий `null` (`String`). Типам, не допускающим `null`, не может быть присвоено значение `null`.

Это правило относится не только к функциям. Следующий код не будет компилироваться по той же причине:

```
var quest: String = "Rescue the princess"
quest = null
```

Данное поведение может отличаться от того, к чему вы привыкли, работая с другими языками. Например, в Java следующий код допустим:

```
String quest = "Rescue the princess";
quest = null;
```

Повторное присваивание `quest` значения `null` работает не только в Java, но и в JavaScript, и в большинстве платформенных языков. Но как вы думаете, что произойдет при попытке выполнить операцию с такой строкой?

```
String quest = "Rescue the princess";
quest = null;
quest = quest.replace("princess", "prince");
```

Этот код выдаст исключение `NullPointerException`, которое, в свою очередь, приведет к аварийному завершению программы. В JavaScript эта проблема имеет название `TypeError`. Пользователи платформенных приложений обычно видят устрашающую ошибку сегментации.

Независимо от платформы этот код работать не будет, потому что он приказывает несуществующей строке заменить часть себя. Выполнить такой приказ невозможно.

(Если вы еще не поняли, почему значение `null` в строковой переменной — это не то же самое, что пустая строка, попробуем разъяснить на примере. Значение `null` подразумевает, что переменной не существует. Пустая строка означает, что переменная существует и имеет значение "", и при вызове `replace` легко может быть возвращена другая пустая строка.)

В Java и многих других языках поддерживается команда, которая на псевдокоде выглядит так: «Эй, строка, которая может существовать или не существовать, замени это слово другим!» В таких языках любая переменная может быть равна `null` (за исключением примитивов, отсутствующих в языке Kotlin). В языках, допускающих `null` для любого типа, ошибки из-за использования `null` являются источником сбоев приложения.

В Kotlin используется противоположный подход к проблеме `null`. Если не указано иное, то переменной нельзя присвоить значение `null`. Это предотвращает проблемы «Эй, несуществующий элемент, сделай что-нибудь» еще во время компиляции, а не в момент выполнения программы. (Мы еще вернемся к сравнению ошибок времени компиляции с ошибками времени выполнения.)

Явный тип null в Kotlin

Надо любой ценой избегать исключений `NullPointerException`, как в приведенном выше примере. Язык Kotlin и компилятор содержат ряд средств, которые защищают вас от подобных сбоев. Прежде всего запрещается присваивание значения `null` переменной, тип которой не допускает `null`. Если вы хотите, чтобы тип допускал `null`, это необходимо явно указать в программе.

Чтобы пометить тип как допускающий `null`, поставьте вопросительный знак после имени типа, например `String?`. Чтобы исправить ошибку компиляции, возникшую в листинге 7.1, обновите тип возвращаемого значения `obtainQuest`.

Листинг 7.2. Использование типа, допускающего null (Main.kt)

```
...
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
    hasAngeredBarbarians: Boolean = false
):- String = when (playerLevel) {
): String? = when (playerLevel) {
    1 -> "Meet Mr. Bubbles in the land of soft things."
    ...
    8 -> "Defeat Nogartse, bringer of death and eater of worlds."
    else -> null
}
```

Тип `String?` допускает `null`. Типы, допускающие `null`, могут хранить либо значение исходного типа (в данном случае `String`), либо `null`. Вскоре мы покажем, как обрабатывать значения `null` в коде.

Ошибка компилятора в `obtainQuest` исчезла, но вместо нее появилась новая ошибка — на этот раз в `readBountyBoard`. Мы вернемся к ней чуть позже, а пока удалите вызов `replace` в `readBountyBoard`. (Не беспокойтесь, вскоре операция замены вернется на место.)

Листинг 7.3. Удаление цензуры (Main.kt)

```
...
private fun readBountyBoard() {
    println(
        """
        |$HERO_NAME approaches the bounty board. It reads:
        |-----${obtainQuest(playerLevel).replace("Nogartse", "xxxxxxx")}
        |      "${obtainQuest(playerLevel)}"
        |-----
        """).trimMargin()
    )
}
...
```

Снова выполните bounty-board. Когда программа запросит уровень, введите 10. Вы получите следующий результат:

```
Madrigal announces her presence to the world.
What level is Madrigal?
10
Madrigal's level is 10.
Madrigal approaches the bounty board. It reads:
  "null"
Time passes...
Madrigal returns from her quest.
11
Madrigal approaches the bounty board. It reads:
  "null"
```

На уровне 10 Мадригал выходит за границы мира смертных, поэтому bounty-board не предлагает новые миссии. Чтобы понять, что здесь произошло, взгляните на нашу реализацию readBountyBoard.

```
private fun readBountyBoard() {
    println(
        """
        |$HERO_NAME approaches the bounty board. It reads:
        |  "${obtainQuest(playerLevel)}"
        |""".trimMargin()
    )
}
```

Так как obtainQuest при выполнении возвращает null, Kotlin вставляет текст «null» в строковый шаблон. Это лучше аварийного завершения, но все же не то, что нам нужно. Немного позже мы усовершенствуем этот результат. Но сначала надо понять, как безопасно работать со значениями, допускающими null.

Для начала добавим промежуточную переменную для хранения результата obtainQuest. Она предоставляет больше возможностей для контроля, чем прямое встраивание результата в необработанную строку.

Листинг 7.4. Определение переменной, допускающей null (Main.kt)

```
...
private fun readBountyBoard() {
    val quest: String? = obtainQuest(playerLevel)

    println(
        """
        |$HERO_NAME approaches the bounty board. It reads:
        |  "${obtainQuest(playerLevel)}"
        |  "$quest"
        |""".trimMargin()
    )
}
...
```

Новая переменная `quest` объявляется с типом `String?`, допускающим `null`. Если бы вместо него использовался тип `String`, то при попытке выполнить этот код вы получили бы ошибку компиляции: `Type mismatch: inferred type is String? but String was expected`. (Чтобы убедиться в этом, попробуйте удалить `?`. Не забудьте вернуть символ на место, прежде чем двигаться дальше.)

Важно понимать, что `String?` и `String` — два разных типа. `obtainQuest` теперь возвращает значение типа `String?`, допускающего `null`, а не значение типа `String`, не допускающего `null`. Kotlin проверяет все операции присваивания и следит за тем, чтобы значения `null` не были случайно присвоены переменным, не помеченным как допускающие `null`. Это происходит во время компиляции для предотвращения проблем еще до выполнения вашей программы.

Время компиляции и время выполнения

Kotlin является *компилируемым языком*. Это означает, что программа транслируется в инструкции машинного языка специальной программой — *компилятором*. Компилятор должен убедиться, что ваш код соответствует некоторым условиям, прежде чем переводить его в машинный код.

Например, компилятор проверяет, можно ли присвоить `null` некоторому типу. Если попробовать присвоить `null` типу, который не допускает такой возможности, Kotlin откажется компилировать программу.

Ошибки, возникшие во время компиляции, называются *ошибками времени компиляции*, и они считаются одним из преимуществ при работе с Kotlin. Конечно, утверждение о том, что ошибки могут быть преимуществом, звучит немного странно, однако если компилятор проверит программу прежде, чем вы позволите другим запустить ее, и найдет все ошибки, это заметно облегчит их поиск и устранение.

Ошибка времени выполнения возникает после компиляции, когда программа уже запущена, потому что компилятор не смог ее обнаружить. Например, так как в Java отсутствует разграничение между типами, допускающими и не допускающими `null`, компилятор Java не способен обнаружить проблему, связанную с присваиванием переменной значения `null`. Такой код успешно компилируется в Java, но во время выполнения вызывает сбой.

Проще говоря, ошибки времени компиляции предпочтительнее ошибок времени выполнения. Лучше обнаружить проблему в процессе написания кода, чем потом. Всем известно: узнать об ошибке только тогда, когда программа уже выпущена, — хуже некуда.

Null-безопасность

Так как Kotlin различает типы, допускающие и не допускающие `null`, компилятор знает о возможной опасности обращения к переменной, объявленной с типом,

допускающим `null`, когда эта переменная может не существовать. Чтобы защититься от подобных неприятностей, Kotlin запрещает вызывать функции для значений, которые могут принимать значение `null`, если только вы не возьмете на себя ответственность за эту ситуацию.

Чтобы понять, как это выглядит на практике, верните на место вызов `replace` для `quest`.

Листинг 7.5. Использование переменной, допускающей `null` (Main.kt)

```
...
private fun readBountyBoard() {
    val quest: String? = obtainQuest(playerLevel)
    val censoredQuest = quest.replace("Nogartse", "xxxxxxx")

    println(
        """
        |$HERO_NAME approaches the bounty board. It reads:
        |-----"$quest"
        |  "$censoredQuest"
        |""".trimMargin()
    )
}
...
```

Компилятор предупреждает красным подчеркиванием о проблеме с новым кодом. (Это та же проблема, которая появилась при замене возвращаемого типа `obtainQuest` на `String?`.) Не обращайте внимания на предупреждение и запустите `Main.kt`. Вместо измененной версии описания миссии вы получите ошибку времени компиляции:

```
Only safe (?.) or non-null asserted (!!) calls
are allowed on a nullable receiver of type String?
```

Kotlin не разрешает вызывать функцию `replace`, потому что вы еще не разобрались с тем, что `quest` может принимать значение `null`. Независимо от того, получает ли `quest` во время выполнения `null` или другое значение, ее тип по-прежнему допускает `null`. Kotlin прерывает компиляцию, чтобы не допустить возникновения ошибки времени выполнения, потому что компилятор осознал возможность вашей ошибки с типом, допускающим `null`.

Сейчас вы, наверное, думаете: «Ну так и что делать с возможностью `null`? У меня важные дела с Тем, Кого Нельзя Называть». Есть несколько вариантов безопасной работы с типом, допускающим `null`, и сейчас мы разберем три из них и еще некоторые дополнительные возможности.

Начнем с варианта номер ноль: используйте тип, не допускающий `null`, если это возможно.

Как мы уже говорили ранее в этой главе, возможность использования `null` служит определенной цели, и в некоторых случаях дополнительная работа

по внедрению в ваш код типов, допускающих null, оправданна. Но если у вас нет убедительной причины для этого, подумайте о применении типа, не допускающего null. Типы, не допускающие null, проще анализировать, потому что они гарантированно содержат значение, для которого можно вызывать функции.

Часто тип с null просто не нужен. И в таких случаях отказ от него — самое надежное решение.

Первый вариант: проверка null в операторе if

Бывает, что без типа, допускающего применение null, никак не обойтись. Либо же вы работаете с чужим кодом и не можете быть уверены в том, что переменная не вернет null.

Самое прямолинейное решение для безопасной работы с null — уже известный вам инструмент, оператор if/else. Вспомните табл. 3.1 из главы 3, где перечислены операторы сравнения, доступные в Kotlin. Оператор != проверяет, что значение в левой части не равно значению в правой части, и с его помощью можно убедиться, что значение отлично от null. Попробуем применить его с readBountyBoard.

Листинг 7.6. Использование !=null для проверки null

```
...
private fun readBountyBoard() {
    val quest: String? = obtainQuest(playerLevel)
    if (quest != null) {
        val censoredQuest: String = quest.replace("Nogartse", "xxxxxxx")
        println(
            """
                |$HERO_NAME approaches the bounty board. It reads:
                | "$censoredQuest"
                | """.trimMargin()
        )
    }
}
...
```

Если quest содержит null, код проигнорирует значение и ничего не выведет. Запустите bounty-board, введите уровень 8 и убедитесь, что все работает, как предполагалось:

```
Madrigal announces her presence to the world.
What level is Madrigal?
8
Madrigal's level is 8.
Madrigal approaches the bounty board. It reads:
    "Defeat xxxxxxxx, bringer of death and eater of worlds."
Time passes...
Madrigal returns from her quest.
9
```

Возможно, вы заметили, что среда IntelliJ выделяет `quest` зеленым цветом, когда вы ссылаетесь на `quest` внутри оператора `if`:

```
if (quest != null) {  
    val censoredQuest: String = quest.replace("Nogartse", "xxxxxxx")  
    ...  
}
```

Эта подсказка: IntelliJ сообщает об использовании так называемого *умного приведения типа* (smart casting). При проверке типа переменной командой `if` Kotlin автоматически *приводит* его к типу внутри ветви — это означает, что переменная будет интерпретирована как относящаяся к указанному типу. Данная возможность языка работает и с проверками `null`: если вы убедились в том, что некоторое свойство не содержит `null`, Kotlin автоматически преобразует его к соответствующему типу, не допускающему `null`.

Тем не менее умное приведение типа имеет некоторые ограничения. Например, при использовании `var` уровня файла выполнять умное приведение небезопасно, потому что значение может измениться между моментом проверки и моментом использования автоматически преобразованного значения. Чтобы обойти это, можно создать временную копию переменной в области видимости функции или же воспользоваться одним из инструментов обеспечения безопасности, предоставляемых Kotlin.

Второй вариант: оператор безопасного вызова

Хотя оператор `if` кажется естественным при работе со значениями `null`, часто это не лучший вариант. Конструкции `if` могут стать слишком громоздкими и длинными, если вам нужно только вызвать одну функцию для значения, допускающего `null`, или если у вас имеется цепочка вызовов функций, каждая из которых может вернуть `null`.

Чтобы избавиться от лишнего кода, можно воспользоваться *оператором безопасного вызова* (safe call operator) (`?.`) для вызова функций с объектами, допускающими `null`. Примените его в `readBountyBoard`, переместив переменную `censoredQuest` за пределы конструкции `if`.

Листинг 7.7. Использование оператора безопасного вызова (Main.kt)

```
...  
private fun readBountyBoard() {  
    val quest: String? = obtainQuest(playerLevel)  
    val censoredQuest: String? = quest?.replace("Nogartse", "xxxxxxx")  
}
```

```
if (quest != null) {
    val censoredQuest: String = quest.replace("Nogartse", "xxxxxxx")
    if (censoredQuest != null) {
        println(
            """
            |$HERO_NAME approaches the bounty board. It reads:
            | "$censoredQuest"
            | """).trimMargin()
    }
}
...
}
```

Когда компилятор встречает оператор безопасного вызова, он знает, что надо проверить значение на `null`. Если во время выполнения оператор безопасного вызова вызывается для `null`, компилятор пропустит вызов функции и просто не будет его вычислять, вместо того чтобы возвращать `null`.

Например, если `quest` хранит значение, отличное от `null`, будет возвращена цензурированная версия. Если же `quest` *содержит* `null`, функция `replace` не будет вызвана, так как это небезопасно. Снова запустите `bounty-board` и введите уровень 8. Должна выводиться версия с замененным именем, а после повышения уровня до 9 программа должна завершиться без фатального сбоя.

Оператор безопасного вызова гарантирует, что функция будет вызвана в том и только в том случае, если переменная, с которой она работает, отлична от `null`. Это предотвращает исключение, вызванное обращением к `null`-указателю. В таких случаях, как в примере выше, мы говорим, что функция `replace` вызывается безопасно и риска `NullPointerException` не существует.

Использование безопасного вызова с `let`

Безопасный вызов позволяет вызвать одну функцию с типом, допускающим `null`. Но что, если вы хотите выполнить дополнительную работу, например создать новую переменную или передать переменную, допускающую `null`, в аргументе, который не должен допускать `null` (вместо того, чтобы вызвать для него функцию)? Одно из возможных решений — использовать оператор безопасного вызова с функцией `let`.

Функция `let` может вызываться с любым значением. Она создает новую область видимости с доступом к значению, для которого она вызывается, где вы можете выполнять код по своему усмотрению. Вы узнаете больше о `let` в главе 12, а сейчас изменим реализацию `readBountyBoard`, чтобы вместо оператора `if` в ней использовалась функция `let`.

Листинг 7.8. Использование `let` с оператором безопасного вызова (`Main.kt`)

```
...
private fun readBountyBoard() {
    val quest: String? = obtainQuest(playerLevel)
    val censoredQuest: String? = quest?.replace("Nogartse", "xxxxxxx")

    if (censoredQuest != null) {
        censoredQuest?.let {
            println(
                """
                |$HERO_NAME approaches the bounty board. It reads:
                | "$censoredQuest"
                """).trimMargin()
            )
        }
    }
}
...
```

Пока использование `let` вместо `if/else` не дает никаких преимуществ. В данном случае обе версии кода делают одно и то же (и на них распространяются ограничения умного приведения типа, о котором мы говорили выше). Чтобы эффективнее использовать мощь `let`, консолидируйте логику замены так, как показано в листинге 7.9.

Листинг 7.9. Консолидация кода с использованием `let` (`Main.kt`)

```
...
private fun readBountyBoard() {
    val quest: String? = obtainQuest(playerLevel)
    val censoredQuest: String? = quest?.replace("Nogartse", "xxxxxxx")

    censoredQuest?.let {
        println(
            val message: String? = quest?.replace("Nogartse", "xxxxxxx")
            ?.let { censoredQuest ->
                """
                |$HERO_NAME approaches the bounty board. It reads:
                | "$censoredQuest"
                """).trimMargin()
            )
    }

    println(message)
}
...
```

Здесь `message` определяется как переменная, допускающая `null`. Ей присваивается значение, полученное в результате безопасного вызова `let` для `quest` после вызова `replace`. Если значение `quest` отлично от `null`, то при вызове `let` будет выполнено все содержимое блока, следующего за `let`.

Мы подробнее расскажем об этом синтаксисе в главах 8 и 12. А пока вам достаточно понять, что в фигурных скобках `let` определяется новая переменная с именем `censoredQuest`, которой присваивается значение `quest?.replace("Nogartse", "xxxxxxx")`. Так как `let` используется с оператором безопасного вызова (`?.`), значение `censoredQuest` отлично от `null`, потому что оператор безопасного вызова пропускает вызов `let`, если во время выполнения значение `quest` отлично от `null`.

Снова запустите `bounty-board`, на этот раз укажите уровень 6. Вы должны получить тот же вывод, который уже видели ранее:

```
Madrigal announces her presence to the world.
What level is Madrigal?
6
Madrigal's level is 6
Madrigal approaches the bounty board. It reads:
    "Locate the enchanted sword."
Time passes...
Madrigal returns from her quest.
7
Madrigal approaches the bounty board. It reads:
    "Recover the long-lost artifact of creation."
```

Оператор объединения с `null`

Снова запустите `bounty-board` и введите уровень 10. Результат должен выглядеть так:

```
Madrigal announces her presence to the world.
What level is Madrigal?
10
Madrigal's level is 10
null
Time passes...
Madrigal returns from her quest.
11
null
```

Значение `null` снова «пробралось» в вывод, потому что безопасный вызов `let` передает значение `null` переменной `quest` в `message`. В следующей итерации `bounty-board` эту проблему мы решим раз и навсегда. Если на «доске поручений» нет миссии, то будет выводиться сообщение `Madrigal approaches the bounty board, but it is blank` (Мадригал подходит к доске поручений, но она пуста).

Того же результата можно добиться командой `if/else`, которая проверяет `message` на `null`, однако в Kotlin есть и другой инструмент, идеально подходящий для этой цели.

Если вы хотите использовать резервное значение при обнаружении `null` в коде, используйте *оператор объединения с `null` `?:`* (также известный как оператор Elvis, потому что при взгляде сбоку похож на прическу Элвиса Пресли). Оператор как бы говорит: «Если слева от меня стоит `null`, то выполняй операцию справа от меня».

Воспользуемся оператором объединения с `null`, чтобы при пустой «доске поручений» выводилось специальное сообщение.

Листинг 7.10. Использование оператора объединения с `null` (Main.kt)

```
...
private fun readBountyBoard() {
    val quest: String? = obtainQuest(playerLevel)

    val message: String? = quest?.replace("Nogartse", "xxxxxxx")
    val message: String = quest?.replace("Nogartse", "xxxxxxx")
        ?.let { censoredQuest ->
            """
                |$HERO_NAME approaches the bounty board. It reads:
                | "$censoredQuest"
                | """.trimMargin()
            } ?: "$HERO_NAME approaches the bounty board, but it is blank."

    println(message)
}
...
```

Чаще всего в этой книге мы не обозначаем тип переменной, если компилятор Kotlin может определить его автоматически. Мы добавили тип `String` для `message` в этот пример, чтобы продемонстрировать роль оператора `?:`.

Если `let` возвращает значение `null` (или если безопасный вызов передаст значение `null`, для которого он был вызван), то `message` будет присвоено значение `"$HERO_NAME approaches the bounty board, but it is blank."`. Если значение `quest` (и как следствие `censoredQuest`) отлично от `null`, используется уже знакомая вам необработанная строка.

Так или иначе, `message` присваивается значение типа `String`, а не `String?`. И это правильно: теперь сообщение, предназначенное для пользователя, заведомо будет отлично от `null`.

Рассматривайте оператор `?:` как способ убедиться, что значение отлично от `null`, и получить значение по умолчанию, если все-таки оно `null`. Оператор `?:` помогает избавиться от значений `null`, чтобы вы могли спокойно работать.

Запустите `Main.kt`, введите уровень 10 и посмотрите, как выглядит результат:

```
Madrigal announces her presence to the world.
What level is Madrigal?
10
Madrigal's level is 10
Madrigal approaches the bounty board, but it is blank.
Time passes...
Madrigal returns from her quest.
11
Madrigal approaches the bounty board, but it is blank.
```

Оператор объединения с `null` также может использоваться независимо от функции `let`. Тот же результат дает следующий код (не вносите это изменение в `bounty-board`).

```
private fun readBountyBoard() {
    val quest: String? = obtainQuest(playerLevel)
    val message: String? = quest?.replace("Nogartse", "xxxxxxx")
        ?.let { censoredQuest ->
            """
                |$HERO_NAME approaches the bounty board. It reads:
                |   "$censoredQuest"
                |""$.trimMargin()
        }

    println(message ?: "$HERO_NAME approaches the bounty board, but it is
        blank.")
}
```

Данный пример демонстрирует встроенное применение оператора объединения с `null` как аргумента `println`. С точки зрения функциональности этот вариант эквивалентен коду в листинге 7.10. Если `message` содержит `null`, то в консоль выводится сообщение "\$HERO_NAME approaches the bounty board, but it is blank." В противном случае выводится `message`.

Оба способа использования оператора объединения с `null` допустимы. Какой из них лучше? Мы не можем ответить на этот вопрос, потому что выбор зависит от стиля.

В некоторых случаях сложные цепочки безопасных вызовов и операторов объединения с `null` усложняют чтение кода или понимание ожидаемого результата. В таких случаях мы выбираем операторы `if/else` за их ясность, за которую приходится расплачиваться потерей компактности. Если вы или ваша команда не согласны с нами, ничего страшного, так как оба стиля приемлемы.

Третий вариант: оператор проверки

Последний способ проверки `null`-безопасности, который мы рассмотрим, уже встречался вам в главе 6. Это *оператор утверждения не-null-значения* (non-null assertion operator), который записывается двумя восклицательными знаками (`!!`) и часто называется просто оператором двойного восклицания.

При помощи оператора `!!` можно заставить компилятор вызвать функцию для типа, допускающего `null`. Но предупреждаем: это более радикальный метод, чем безопасный вызов, и его не следует использовать без крайней необходимости. Визуально `!!` должен выделяться в коде, потому что это рискованный вариант. Использовать оператор двойного восклицания — все равно что сказать компилятору: «Я *требую*, чтобы ты выполнил эту операцию! А если не сможешь, то остальной код можно уже не выполнять!»

Мы используем этот оператор при вызове `readLine`:

```
fun main() {
    println("$HERO_NAME announces her presence to the world.")
    println("What level is $HERO_NAME?")
    playerLevel = readLine()!!.replace("[^0-9]".toRegex(), "").toInt()
    println("$HERO_NAME's level is $playerLevel.")
    ...
}
```

`readLine()!!.replace(...).toInt()` означает: «Меня не интересует, вернет `readLine` значение `null` или нет, все равно преобразуй его в число!» Если `readLine` действительно вернет `null`, выдается исключение `NullPointerException`.

(Это относится даже к Kotlin/JS и Kotlin/Native. Kotlin вводит собственный тип `NullPointerException`, вместо того чтобы вызвать `TypeError` или ошибку сегментации. Тем не менее в очень специфических ситуациях все равно можно столкнуться с неожиданной ошибкой `TypeError` или ошибкой сегментации.)

Иногда использование оператора `!!` оправданно. Например, вы не контролируете тип переменной, но точно знаете, что она никогда не получит значение `null`. Если вы абсолютно уверены, что переменная отлична от `null`, то применение оператора `!!` может быть неплохим вариантом.

В случае `readLine` значение `null` будет возвращено, если пользователь перенаправил входной поток в пустой файл. Если вы ожидаете, что пользователи всегда будут вводить данные с консоли, такая возможность будет чрезвычайно редкой, и отсутствие поддержки этого сценария использования вполне оправданно.

С другой стороны, у вызова `readLine` есть и другие альтернативы, и их стоит исследовать, если это позволит предотвратить сбой приложения. Обновите функцию `main` с использованием других средств null-безопасности, о которых мы рассказывали ранее. Также можно воспользоваться функцией `toIntOrNull`, которая возвращает `null` вместо того, чтобы вызывать сбой программы, если строку не удастся разобрать в целое число. Так как `toIntOrNull` позволяет выполнять преобразования более корректно, необходимость в проверке ввода с применением регулярного выражения отпадает.

Листинг 7.11. Удаление оператора двойного восклицания (Main.kt)

```
...
fun main() {
    println("$HERO_NAME announces her presence to the world")
    println("What level is $HERO_NAME?")
    val playerLevelInput = readLine()!!
    playerLevel = if (playerLevelInput.matches("[^0-9]".toRegex())) {
        playerLevelInput.toInt()
```

```
} else {  
    1  
}  
playerLevel = readLine()?.toIntOrNull() ?: 0  
println("$HERO_NAME's level is $playerLevel")  
...  
}  
...
```

Запустите приложение bounty-board несколько раз, вводя разные значения уровня героя. Убедитесь в том, что оно работает так, как предполагалось.

Исключения

Как и во многих других языках, в Kotlin используются исключения для предупреждения о том, что в программе что-то пошло не так. Вы уже сталкивались с исключениями, которые выдает система или стандартная библиотека Kotlin. Исключение `NumberFormatException` возникает при попытке преобразования некоторых строк в `Int`, а об исключении `NullPointerException` мы говорили в этой главе.

Однако использование исключений не ограничивается системой. Вы можете вводить собственные исключения для обозначения потенциальных проблем в вашем приложении.

Выдача исключений

Kotlin, подобно другим языкам программирования, позволяет в ручном режиме послать сигнал о возникновении исключения. Для этого используется ключевое слово `throw`, а сам процесс называется *выдачей* исключения. Кроме уже известного вам исключения `null`-указателя, возможно много других исключений.

Зачем они нужны? Ответ кроется в названии — исключения требуются для описания исключительных ситуаций. Столкнувшись с непреодолимой проблемой, код может выдать исключение и таким способом сообщить, что ее необходимо устранить, чтобы продолжить выполнение.

Одно из таких распространенных исключений, с которым вы познакомитесь, — `IllegalArgumentException`. Безусловно, имя `IllegalArgumentException` выглядит несколько расплывчато — оно означает, что программа получила ввод, который она считает недопустимым. Это исключение удобно тем, что с ним можно передать строку и предоставить больше информации об ошибке.

Возьмем нашу функцию `obtainQuest`. Допустим, вы хотите проверить уровень игрока и убедиться, что это число не является отрицательным. Запрос миссии для игрока уровня `-1` должен быть запрещен, потому что наименьший уровень в bounty-board равен 1. Исключение `IllegalArgumentException` позволит активно оповестить об использовании недопустимого уровня игрока.

Внесем эту проверку в `obtainQuest`. Но сначала вспомним, что `obtainQuest` является функцией с единственным выражением, и воспользуемся сокращенным синтаксисом, предоставляемым Kotlin. Так как мы добавим еще одну команду в эту функцию, ее синтаксис необходимо преобразовать, включив блок тела.

Чтобы это было проще сделать, переместите курсор к имени функции и нажмите Option-Return (Alt-Enter) — откроется меню контекстных действий. Выберите команду **Convert to block body**, которая автоматически вставит фигурные скобки и ключевое слово `return`. (Если тип возвращаемого значения был определен автоматически, среда IntelliJ также автоматически вставит возвращаемый тип.)

```
...
private fun obtainQuest(
    ...
): String? = when (playerLevel) {
    return when (playerLevel) {
        1 -> "Meet Mr. Bubbles in the land of soft things."
        ...
        else -> null
    }
}
```

Теперь добавьте условие, которое проверит, что значение `playerLevel` не отрицательно.

Листинг 7.12. Выдача исключения `IllegalArgumentException` (Main.kt)

```
...
private fun obtainQuest(
    ...
): String? {
    if (playerLevel <= 0) {
        throw IllegalArgumentException("The player's level must be at least 1.")
    }

    return when (playerLevel) {
        1 -> "Meet Mr. Bubbles in the land of soft things."
        ...
        else -> null
    }
}
```

Здесь мы предупреждаем, что значение `playerLevel` должно быть не менее 1; любой другой ввод является недопустимым. А это означает, что каждый, кто захочет работать с переменной `playerLevel`, должен обработать исключительное состояние, обусловленное минимальным разрешенным значением. Такой подход хорош тем, что он повышает вероятность выявления исключительного состояния в ходе разработки, прежде чем это вызовет фатальный сбой при использовании приложения.

Запустите `bounty-board` и введите уровень `-1`. Результат должен выглядеть так:

```
Madrigal announces her presence to the world.
What level is Madrigal?
-1
Madrigal's level is -1.
Exception in thread "main" java.lang.IllegalArgumentException:
    The player's level must be at least 1.
    at MainKt.obtainQuest(Main.kt:41)
    at MainKt.obtainQuest$default(Main.kt:38)
    at MainKt.readBountyBoard(Main.kt:21)
    at MainKt.main(Main.kt:10)
    at MainKt.main(Main.kt)
```

Так как с `IllegalArgumentException` передается сообщение об ошибке, вы точно знаете, почему в вашей программе произошел сбой.

Обработка исключений

Исключения прерывают нормальное выполнение программы. Они представляют ошибочное состояние, которое надо исправить. Kotlin позволяет указать, как должны обрабатываться исключения; для этого код, в котором они могут возникнуть, включается в команду `try/catch`.

Синтаксис `try/catch` похож на синтаксис `if/else`. Чтобы увидеть, как работает `try/catch`, воспользуемся им в `readBountyBoard` для защиты от потенциально опасных операций.

Листинг 7.13. Добавление команды `try/catch` (`Main.kt`)

```
...
private fun readBountyBoard() {
    try {
        val quest: String? = obtainQuest(playerLevel)

        val message: String = quest?.replace("Nogartse", "xxxxxxx")
            ?.let { censoredQuest ->
                """
                |$HERO_NAME approaches the bounty board. It reads:
                |   "$censoredQuest"
                |   """.trimMargin()
            } ?: "$HERO_NAME approaches the bounty board, but it is blank."

        println(message)
    } catch (e: Exception) {
        println("$HERO_NAME can't read what's on the bounty board.")
    }
}
...
```

Добавляя команду `try/catch`, вы определяете блок кода, который должен быть выполнен, и что должно происходить, если в какой-либо команде блока возникает сбой с исключением. Если исключение не происходит, то команда `try` выполняется, а команда `catch` — нет. Эта логика ветвления подобна командам `if/else`.

В блоке `catch` вы определили, что случится, если выражение в блоке `try` породит исключение. Блок `catch` принимает аргумент с определенным типом исключения, от которого нужно защититься. В данном случае перехватываются любые исключения типа `Exception`, с которым будут перехватываться и все ошибки, возникающие в любой строке кода.

Блоки `catch` могут включать любую логику, но в этом примере она предельно упрощена. В данном случае блок `catch` используется для вывода сообщения.

Если теперь назначить герою уровень `-1`, `obtainQuest` все равно выдаст `IllegalArgumentException`. Но поскольку исключение обрабатывается командой `try/catch`, выполнение программы продолжится и блок `catch` будет выполнен, в результате чего в консоли появится следующий результат (убедитесь в этом сами).

```
Madrigal announces her presence to the world.  
What level is Madrigal?  
-1  
Madrigal's level is -1.  
Madrigal can't read what's on the bounty board.  
Time passes...  
Madrigal returns from her quest.  
0  
Madrigal can't read what's on the bounty board.
```

(Мадригал не может прочесть, что написано на доске поручений.)

Сообщение, передаваемое `IllegalArgumentException` ("The player's level must be at least 1."), не выводится на консоль. Сообщения, передаваемые при выдаче исключений, предназначены для отладки, а не для вывода. Сообщения об исключениях часто включают технические подробности о коде и обычно непонятны большинству пользователей вашей программы. Чтобы просмотреть сообщение, вызовите `obtainQuest` без использования блока `try/catch` или запустите `bounty-board` под управлением отладчика, например, встроенного в IntelliJ.

Выражения `try/catch`

В главе 3 мы показали, что операторы `if/else` и блоки `when` можно использовать как выражения. Это верно и для блоков `try/catch`, хотя такой вариант считается нетипичным по сравнению с использованием условного выражения. Здесь мы избавимся от лишних вызовов `println` при помощи того же приема, который впервые был представлен в главе 3. Внесите изменения и посмотрите, как они работают.

Листинг 7.14 Использование try/catch как выражения (Main.kt)

```

...
private fun readBountyBoard() {
    try {
        val message: String = try {
            val quest: String? = obtainQuest(playerLevel)

            val message: String = quest?.replace("Nogartse", "xxxxxxx")
            quest?.replace("Nogartse", "xxxxxxx")
                ?.let { censoredQuest ->
                    """
                        |$HERO_NAME approaches the bounty board. It reads:
                        |   "$censoredQuest"
                        |   """.trimMargin()
                    } ?: "$HERO_NAME approaches the bounty board, but it is blank."

            println(message)
        } catch (e: Exception) {
            println("$HERO_NAME can't read what's on the bounty board.")
            "$HERO_NAME can't read what's on the bounty board."
        }
        println(message)
    }
}
...

```

Это изменение не повлияет на ход выполнения программы (протестируйте и убедитесь сами). С другой стороны, оно сводит почти идентичные вызовы `println` в один. Кроме того, оно гарантирует, что исключение по-прежнему будет обрабатываться. Если удалить строку в блоке `catch`, компилятор пожалуется на отсутствие строкового значения, которое должно быть присвоено `message`.

Этот прием оказывается чрезвычайно полезным, если вы хотите выполнить потенциально опасный код для вычисления некоторого значения и у вас есть резервный вариант, который вы можете использовать взамен опасного кода.

Проверка предусловий

Непредусмотренные значения иногда приводят к тому, что программа ведет себя непредсказуемо. Вам как разработчику придется потратить много времени, проверяя достоверность входных значений, чтобы убедиться, что вы получаете ожидаемые данные. Некоторые источники исключений тривиальны, например непредусмотренные значения `null`. В стандартной библиотеке Kotlin есть несколько удобных функций, упрощающих проверку и отладку входных данных. Они позволяют использовать встроенную функцию для выдачи исключения с произвольным сообщением.

Такие функции называются *функциями проверки предусловий* (precondition functions), потому что позволяют задать условия, которые должны быть истинными до выполнения кода.

Например, в этой главе мы показали несколько вариантов защиты от `NullPointerException` и других исключений. Еще один вариант — использование функции проверки предусловия `require`, которая проверяет значение на `null` и возвращает его, если оно не равно `null`, а в противном случае выдает исключение `IllegalArgumentException`.

Попробуйте заменить `IllegalArgumentException` вызовом функции проверки предусловия.

Листинг 7.15. Использование функции проверки предусловия (Main.kt)

```
...
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
    hasAngeredBarbarians: Boolean = false
): String? {
    if (playerLevel <= 0) {
        throw IllegalArgumentException("The player's level must be at least 1.")
    }
    require(playerLevel > 0) {
        "The player's level must be at least 1."
    }

    return when (playerLevel) {
        ...
    }
}
```

Функция `require` позволяет удалить шаблонный код выдачи `IllegalArgumentException`. При вызове она получает два аргумента. В первом передается проверяемое логическое выражение. Если оно равно `false`, выдается исключение `IllegalArgumentException`. Второй аргумент, заключенный в фигурные скобки, содержит сообщение об ошибке, включаемое в выданное исключение. В главе 8 мы расскажем, что означает эта пара фигурных скобок.

Функции проверки предусловий хорошо подходят для проверки требований перед выполнением кода. Они гораздо понятнее, чем ручная выдача исключений, потому что проверяемое условие включается в имя функции.

Стандартная библиотека Kotlin содержит шесть функций проверки предусловий — они перечислены в табл. 7.1.

Обратите особое внимание на функции `checkNotNull` и `requireNotNull`. Эти две функции проверки предусловий получают аргумент и выдают исключение, если аргумент — `null`. Они различаются по типу выдаваемого исключения: `require` и `requireNotNull` выдают `IllegalArgumentException`, а `check` и `checkNotNull` выдают `IllegalStateException`. Лучше применять `IllegalArgumentException`, когда вы

проверяете входные данные функции, и `IllegalStateException` во всех остальных ситуациях.

Таблица 7.1. Функции проверки предусловий

Функция	Описание
<code>check</code>	Выдает <code>IllegalStateException</code> , если аргумент — <code>false</code>
<code>checkNotNull</code>	Выдает <code>IllegalStateException</code> , если аргумент — <code>null</code> . В противном случае возвращает полученное значение
<code>require</code>	Выдает <code>IllegalArgumentException</code> , если аргумент — <code>false</code>
<code>requireNotNull</code>	Выдает <code>IllegalArgumentException</code> , если аргумент — <code>null</code> . В противном случае возвращает полученное значение
<code>error</code>	Выдает <code>IllegalArgumentException</code> с заданным сообщением, если аргумент — <code>null</code> . В противном случае возвращает полученное значение
<code>assert</code>	Выдает <code>AssertionError</code> , если аргумент — <code>false</code> и на этапе компиляции установлен флаг, разрешающий проверку тестовых утверждений ¹

Функции проверки предусловий также используются для обеспечения `null`-безопасности. Хотя конечный результат аналогичен оператору утверждения о не-`null`-значении (`!!`), они позволяют добиться большей выразительности. Вместо обобщенного исключения `NullPointerException` вы получаете возможность дать в исключении больше информации о том, что именно было `null`.

Например, можно использовать `checkNotNull` в сочетании с `readLine`:

```
val input: String = checkNotNull(readLine()) {
    // Выдает IllegalStateException с сообщением "No input was provided"
    "No input was provided"
}
```

Проверка предусловий позволяет гарантировать, что приложение работает нормально, а если оно *не* работает нормально, то как можно быстрее выдать ошибку для упрощения отладки. Она поможет легко выдавать самые распространенные типы исключений, и мы надеемся, что вы будете активно пользоваться ею для выявления потенциальных проблем в приложениях.

Запустите `bounty-board` в последний раз и убедитесь в том, что проверка предусловия не сработала. Поведение программы не должно измениться.

¹ О проверке тестовых утверждений мы не будем рассказывать в этой книге. Если вас интересует эта тема, см. kotlinlang.org/api/latest/jvm/stdlib/kotlin/assert.html и docs.oracle.com/cd/E19683-01/806-7930/assert-4/index.html.

В этой главе мы рассказали, как Kotlin справляется с проблемами, связанными с неопределенным значением `null`. Вы узнали, что типы, допускающие `null`, надо объявлять явно, потому что по умолчанию значение `null` недопустимо, а также что следует по возможности отдавать предпочтение типам, не допускающим `null`, потому что такие значения помогают компилятору предотвращать ошибки времени выполнения.

Кроме того, мы познакомили вас с приемами безопасной работы с типами, допускающими `null`, когда без них не обойтись, — с использованием оператора безопасного вызова, оператора `?:` или непосредственной проверкой равенства с `null`. Мы рассмотрели функцию `let` и способ ее использования в сочетании с оператором безопасного вызова для безопасного вычисления выражений с переменными, способными принимать значение `null`.

Наконец, вы познакомились с исключениями, научились обрабатывать их с помощью синтаксиса `try/catch`, а также определять предусловия с целью перехвата исключительных состояний раньше, чем те вызовут сбой.

На этом наша работа над `bounty-board` подходит к концу. В следующей главе мы создадим новый проект `NyetHack` и расскажем вам об *анонимных функциях* — важнейшем структурном элементе многих функций стандартной библиотеки Kotlin.

Для любознательных: пользовательские исключения

В этой главе вы узнали, как использовать оператор `throw`, чтобы сообщить об исключительной ситуации. Исключение `IllegalArgumentException` сообщает, что был предоставлен недопустимый ввод, и дает возможность добавить строку с дополнительной информацией.

Чтобы дать больше подробностей в сообщении об исключении, можно создать пользовательское исключение для конкретной проблемы. Для этого нужно объявить новый *класс*, наследующий от другого исключения. Классы позволяют определять в программе конкретные вещи — монстров, оружие, еду, инструменты и т. д. Мы расскажем о классах в главе 13, поэтому пока в подробности синтаксиса вы можете не вникать.

Определите пользовательское исключение с именем `InvalidPlayerLevelException` в `Main.kt`.

Листинг 7.16. Определение пользовательского исключения (Main.kt)

```
...
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
```

```
    hasAngeredBarbarians: Boolean = false
): String? {
    ...
}
```

```
class InvalidPlayerLevelException() :
    IllegalArgumentException("Invalid player level (must be at least 1).")
```

`InvalidPlayerLevelException` — пользовательское исключение, которое действует как `IllegalArgumentException` с конкретным сообщением.

Новое, нестандартное исключение выдается так же, как исключение `IllegalArgumentException`, — ключевым словом `throw`. В этом смысле пользовательское исключение ничем не отличается от других типов исключений.

Листинг 7.17. Выдача пользовательского исключения (Main.kt)

```
...
private fun obtainQuest(
    playerLevel: Int,
    playerClass: String = "paladin",
    hasBefriendedBarbarians: Boolean = true,
    hasAngeredBarbarians: Boolean = false
): String? {
    require(playerLevel > 0) {
        "The player's level must be at least 1."
    }
    if (playerLevel <= 0) {
        throw InvalidPlayerLevelException()
    }

    return when (playerLevel) {
        ...
    }
}
```

`InvalidPlayerLevelException` — нестандартная ошибка, которая должна появляться, если `playerLevel` меньше 1. Никакая часть кода, определяющего исключение, не указывает, когда конкретно оно должно выдаваться, — это зависит исключительно от вас.

Пользовательские исключения удобны и гибки. Их применяют не только для вывода произвольных сообщений, но и для выполнения каких-либо операций при их выдаче. Они также сокращают дублирование кода, так как их можно использовать в проектах повторно.

Для любознательных: проверяемые и непроверяемые исключения

В Kotlin все исключения *непроверяемые*. Это означает, что компилятор Kotlin не заставляет упаковывать весь код, который может вызвать исключения, в команду `try/catch`.

Такой подход особенно непривычен для пользователей Java, так как Java поддерживает и проверяемые, и непроверяемые типы исключений. В случае с проверяемыми исключениями компилятор выясняет, установлена ли защита от исключения, требуя добавить в программу блок `try/catch` или явно указать, что функция выдает исключение. Непроверяемые исключения не тестируются во время компиляции: компилятор позволяет выдавать их независимо от наличия соответствующего блока `try/catch`.

Это *выглядит* разумно. Но на практике проверяемые исключения работают не так хорошо, как задумывали их создатели. Часто проверяемые исключения перехватываются (потому что компилятор требует их обработки), а затем просто игнорируются, лишь бы программа скомпилировалась. Этот процесс «поглощения исключения» сильно усложняет отладку программы, потому что он маскирует информацию о том, что же пошло не так. В большинстве случаев игнорирование проблемы во время компиляции приводит к ошибкам времени выполнения.

Непроверяемые исключения победили в современных языках программирования, потому что опыт показал, что проверяемые исключения создают проблем больше, чем решают: дублирование кода, сложная для понимания логика и поглощенные исключения, которые не позволяют получить информацию о возникших проблемах.

Часть III

Функциональное программирование и коллекции

Функциональное программирование — это парадигма, в которой интенсивно используются функции высшего порядка, то есть те, которые могут передаваться и возвращаться другими функциями для изменения своего поведения и поведения других функций во время выполнения кода. Некоторые языки программирования являются исключительно функциональными; это означает, что все приложение строится в стиле функционального программирования. Хотя Kotlin не относится к таким языкам, он предоставляет много инструментов, чтобы разработчики могли выражать логику программы в функциональной парадигме.

В Kotlin функции являются «первоклассными сущностями», то есть они обрабатываются точно так же, как и любой другой тип. Стандартная библиотека Kotlin использует это преимущество, предоставляя разработчикам гибкий и компактный набор API для решения типовых задач.

В следующих пяти главах мы исследуем мощные средства для работы с функциями в Kotlin. Также мы познакомим вас с типами коллекций Kotlin, предназначенных для хранения групп данных. Вы увидите, как функциональное программирование с использованием коллекций позволяет выполнять сложные операции и алгоритмы всего в нескольких строках кода. Чтобы показать эти возможности в действии, мы начнем строить новый проект — NyetHack — текстовую ролевую игру, над которой мы будем работать вплоть до главы 19.

8. Лямбда-выражения и тип функции

В главе 4 вы узнали, как определять функции в Kotlin с указанием имени и как вызывать их по этому имени. В этой главе мы покажем другой способ определения функций. Он позволяет рассматривать функцию как значение — практически так же, как `String` и `Int`. Этот стиль позволяет хранить функции в переменных, передавать функции другим функциям и возвращать из функций.

Представляем NyetHack

Чтобы опробовать все это на практике, создадим проект NyetHack, над которым и будем работать до главы 19.

Почему NyetHack? Хороший вопрос. Возможно, вы помните игру NetHack, выпущенную в 1987 году командой The NetHack DevTeam. Это однопользовательская текстовая фэнтезийная игра, использующая ASCII-графику; с ней можно ознакомиться на nethack.org. Мы создадим подобную текстовую игру (правда, без такой потрясающей ASCII-графики).

У команды создателей языка Kotlin — JetBrains — есть свое представительство в России. Более того, Kotlin назван по имени российского острова¹. Если в текстовую игру типа NetHack добавить русский колорит, то получится NyetHack.

Приступим. Откройте IntelliJ и создайте новый проект. Если в IntelliJ у вас уже открыт другой проект, выполните команду **File ▶ New ▶ Project...**, чтобы запустить мастер New Project.

В левом столбце выберите **Kotlin**. На центральной панели укажите вариант **Application** под заголовком **JVM** и систему сборки **Gradle Groovy**. Проследите за тем, чтобы был задан верный вариант **Project JDK** (мы рекомендуем любую версию Java от 1.8 до 15). Это те же настройки, которые использовались для **bounty-board**.

Укажите имя проекта — **NyetHack**. IntelliJ предложит сохранить новый проект в той же папке, что и предыдущий; при желании вы можете изменить папку. Когда все будет готово, щелкните на кнопке **Next**.

¹ Остров Котлин, на котором находится город Кронштадт. — *Примеч. ред.*

Второй экран мастера позволяет задать дополнительную информацию о шаблоне, который вы хотите использовать. Мы будем строить NyetHack без шаблона, поэтому если ранее не был указан шаблон `None`, выберите его в раскрывающемся списке. Нажмите кнопку `Finish`, и среда IntelliJ создаст пустой проект NyetHack.

Добавьте новый файл в проект. Найдите и разверните папку `src` в инструментальном окне проекта. (Возможно, чтобы увидеть папку `src`, нужно щелкнуть на стрелке слева от названия проекта, чтобы раскрыть список папок.) Раскройте папку `main`, которая содержит папку `kotlin`. Щелкните правой кнопкой мыши на папке `kotlin`, выберите команду `New ► Kotlin Class/File` и создайте файл (не класс!) с именем `NyetHack`. Новый файл откроется в редакторе.

Функция `main`, как было показано в главе 1, определяет точку входа вашей программы. Среда IntelliJ предлагает средства для ускоренного написания этой функции: напечатайте слово «`main`» в `NyetHack.kt` и нажмите клавишу `Tab`. IntelliJ автоматически добавляет основные элементы функции:

```
fun main() {  
  
}
```

Чтобы начать нашу ролевую игру, предложите пользователю ввести свое имя; для этого можно воспользоваться функциями `println` и `readLine`.

Листинг 8.1. Скромное начало NyetHack (NyetHack.kt)

```
fun main() {  
    println("A hero enters the town of Kronstadt. What is their name?")  
    val heroName = readLine() ?: ""  
}
```

Запустите функцию `main` из `NyetHack.kt`. На консоль будет выведен запрос имени игрока, а после ввода приложение завершается.

Анонимные функции

Функции, подобные тем, которые вы только что вызывали, — а на самом деле все функции, которые вызывались до настоящего момента, — называются *именованными функциями*. Они определяются ключевым словом `fun`, и у них всегда есть имя, которое является частью сигнатуры. Однако функцию также можно задать без ключевого слова `fun`. Более того, ее можно задать даже без имени.

Такого рода функции называются *анонимными*, потому что в их определение не входит имя. Анонимные функции взаимодействуют с остальным кодом немного иначе, чем именованные, — обычно они передаются в аргументах или возвращаются из других функций. Это возможно благодаря *типу функции*, о котором мы также расскажем в этой главе.

Анонимные функции — важная часть языка Kotlin. Среди прочего они нужны, чтобы адаптировать функции из стандартной библиотеки Kotlin под решение конкретных задач. Анонимная функция позволяет описать дополнительные правила для функций из стандартной библиотеки, так что вы можете настроить их поведение.

Для примера рассмотрим функцию `count` типа `String`. По умолчанию `count` возвращает общее количество символов в строке. Однако `count` также способна получать в качестве параметра функцию, которая получает единственный параметр `Char` и возвращает `Boolean`. Эта версия `count` вызывает переданную функцию для каждого символа строки и возвращает общее количество полученных результатов `true`.

Чтобы увидеть `count` в действии, введите следующий код в REPL.

Листинг 8.2. Подсчет букв в строке (REPL)

```
"Mississippi".count({ letter -> letter == 's' })
```

Нажмите Command-Return (Ctrl-Enter), чтобы выполнить код. В REPL будет выведено значение 4.

Обратите внимание на новый синтаксис функции `count` в круглых скобках:

```
{ letter -> letter == 's' }
```

Такой синтаксис называется *лямбда-выражением* (также иногда встречается термин «*литерал функции*»), он используется для создания анонимных функций. Лямбда-выражение — это функция, которая передается функции `count`.

Далее мы будем называть анонимные функции *лямбда-функциями*, а их определения — *лямбда-выражениями*. Это общепринятая терминология. (Почему лямбда, спросите вы? Этот термин, также обозначаемый греческой буквой λ , является сокращенной формой названия «лямбда-исчисление» — системы логики для выражения вычислений, разработанной в 1930 году математиком Алонзо Черчем (Alonzo Church). Определяя анонимные функции, мы используем нотацию лямбда-исчисления.)

Чтобы понять, как работает `count`, присмотримся к синтаксису лямбда-выражений в Kotlin. Мы создадим небольшой файл-хелпер, который будет выполнять функцию рассказчика. У рассказчика может быть разное настроение, и определять его мы будем при помощи лямбда-выражений.

Лямбда-выражения

В проекте NyetHack создайте новый файл с именем `Narrator.kt`, в котором мы определим рассказчика и его настроение. Пока у нашего рассказчика будет только один вариант настроения. Создайте новую функцию с именем `narrate`

и реализуйте ее; для этого определите лямбда-выражение, вызовите его и выведите результат.

Листинг 8.3. Определение лямбда-выражения (Narrator.kt)

```
fun narrate(  
    message: String  
) {  
    println({  
        val numExclamationPoints = 3  
        message.uppercase() + "!".repeat(numExclamationPoints)  
    })  
}
```

Как вы записываете строки? Вы размещаете символы между открывающей и закрывающей кавычками. Функции записываются аналогичным образом: выражения или команды размещаются между открывающей и закрывающей фигурными скобками. В данном примере все начинается с вызова `println`. В круглых скобках, в которых содержится аргумент `println`, определяется анонимная функция в фигурных скобках с использованием синтаксиса лямбда-выражений. Лямбда-выражение определяет переменную и возвращает преобразованную версию аргумента `message`:

```
{  
    val numExclamationPoints = 3  
    message.uppercase() + "!".repeat(numExclamationPoints)  
}
```

После закрывающей фигурной скобки лямбда-выражения функция вызывает-ся при помощи пары пустых круглых скобок. Если убрать круглые скобки после лямбда-выражения, строка с сообщением выводиться не будет. Как и обычные функции, анонимные выполняют свою работу только при вызове, для чего в программу включаются фигурные скобки со всеми аргументами, которые ожидает получить функция (в данном случае аргументов нет):

```
{  
    val numExclamationPoints = 3  
    message.uppercase() + "!".repeat(numExclamationPoints)  
}()
```

Чтобы воспользоваться новой функцией, замените вызов `println` в `NyetHack.kt` вызовом `narrate`.

Листинг 8.4. Вызов `narrate` (NyetHack.kt)

```
fun main() {  
    println("A hero enters the town of Kronstadt. What is their name?")  
    narrate("A hero enters the town of Kronstadt. What is their name?")  
}
```

```
    val heroName = readLine() ?: ""
}
```

Снова выполните функцию `main`. Результат должен выглядеть так:

```
A HERO ENTERS THE TOWN OF KRONSTADT. WHAT IS THEIR NAME?!!!
```

Тип функции

В главе 2 вы познакомились с типами данных `Int` и `String`. У литералов функций также имеется тип, который называется *типом функции*. Переменные с типом функции содержат функцию, которая является их значением, и эта функция может передаваться в коде, как любая другая переменная.

(Не путайте тип функции с типом, который называется `Function`. Как вы вскоре увидите, функция конкретизируется с использованием объявления типа функции, который зависит от подробностей входных данных конкретной функции, ее вывода и параметров.)

Включите в `Narrator.kt` определение переменной, которая содержит функцию, и присвойте ей лямбда-выражение для форматирования сообщения. Мы поясним встречающийся в листинге незнакомый синтаксис, но сначала введите код из листинга 8.5.

Листинг 8.5. Присваивание лямбда-выражения переменной (`Narrator.kt`)

```
fun narrate(
    message: String
) {
    val narrationModifier: () -> String = {
        val numExclamationPoints = 3
        message.uppercase() + "!".repeat(numExclamationPoints)
    }

    println({
        val numExclamationPoints = 3
        message.uppercase() + "!".repeat(numExclamationPoints)
    }())
    println(narrationModifier())
}
```

При объявлении переменной после ее имени ставится двоеточие и указывается тип, который вам нужно (или вы хотите) выразить явно. В данном примере — `narrationModifier: () -> String`.

Подобно тому как `:` `Int` сообщает компилятору, какие данные может содержать переменная (целое число), объявление типа функции `: () -> String` сообщает компилятору, что переменная `narrationModifier` должна содержать функцию.

Определение типа функции состоит из двух частей: параметров функции в круглых скобках и возвращаемого типа, перед которым ставится стрелка ($\rightarrow$), как показано на рис. 8.1.

Объявление типа, указанное для переменной `narrationModifier`, `() -> String`, сообщает компилятору, что `narrationModifier` может быть присвоена любая функция без аргументов (на что указывает пара пустых круглых скобок), которая возвращает `String`. Как и при объявлениях других типов переменных, компилятор проверяет, что функция, присваиваемая переменной или передаваемая в аргументе, имеет правильный тип.

Запустите `main` и убедитесь в том, что вы получите тот же вывод:

```
A HERO ENTERS THE TOWN OF KRONSTADT. WHAT IS THEIR NAME?!!!
```



```
fun narrationModifier(): String
      ↓      ↓
    () -> String
```

Рис. 8.1. Синтаксис типа функции

Неявный возврат

Вы могли заметить, что в определенном выше лямбда-выражении отсутствует ключевое слово `return`:

```
val narrationModifier: () -> String = {
    val numExclamationPoints = 3
    message.uppercase() + "!".repeat(numExclamationPoints)
}
```

Однако тип функции показывает, что функция должна вернуть строку, и компилятор не жалуется. Как можно судить по выводу, строка с приветствием действительно была возвращена. Но как это возможно, если ключевое слово `return` отсутствует?

В отличие от обычных функций, синтаксис лямбда-выражений не требует, а в большинстве случаев даже запрещает использовать ключевое слово `return` для вывода данных. Лямбда-выражения *неявно* возвращают результат выполнения последней строки в определении функции, позволяя опустить ключевое слово `return`.

Эта особенность лямбда-выражений и удобна, и необходима для их синтаксиса. Ключевое слово `return` запрещено в лямбда-выражениях, так как оно может создать неоднозначность для компилятора: из какой функции возвращается значение — из самого лямбда-выражения или из функции, из которой оно было вызвано?

Аргументы функции

Как и другие функции, лямбда-функция может принимать один или несколько аргументов любого типа (или ни одного). Параметры лямбда-функции объявляются

перечислением типов в определении типа функции и получают имена в определении лямбда-функции.

Так как настроение рассказчика не должно меняться по ходу изложения, функцию `narrationModifier` следует объявить как переменную верхнего уровня (а не внутри функции `narrate`). Внесите это изменение и обновите объявление переменной `narrationModifier`, чтобы текст сообщения передавался как аргумент.

Листинг 8.6. Добавление параметра (Narrator.kt)

```
val narrationModifier: (String) -> String = { message ->
    val numExclamationPoints = 3
    message.uppercase() + "!".repeat(numExclamationPoints)
}

fun narrate(
    message: String
) {
    val narrationModifier: () -> String = {
        val numExclamationPoints = 3
        message.uppercase() + "!".repeat(numExclamationPoints)
    }
    println(narrationModifier(message))
}
```

Чтобы указать, что лямбда-функция получает `String`, мы заключаем тип аргумента в круглые скобки типа функции:

```
val narrationModifier: (String) -> String = { message ->
```

Имя строкового параметра указывается внутри функции, сразу же после открывающей фигурной скобки:

```
val narrationModifier: (String) -> String = { message ->
```

Лямбда-выражения, предоставляющие имена параметров подобным способом, отделяют их от тела функции оператором-стрелкой (`->`).

Снова запустите `NyetHack.kt`. Вывод не изменится, но ваше лямбда-выражение теперь само получает аргумент `message` вместо того, чтобы читать его из аргумента `narrate`.

Помните функцию `count`, которая может получать функцию, применяемую к каждому символу строки? Эта функция является *предикатным* аргументом с именем `predicate` и типом `(Char) -> Boolean` — иначе говоря, функцией, которая получает аргумент `Char` и возвращает `Boolean`. Типы функций и лямбда-выражения часто встречаются в стандартной библиотеке Kotlin.

Идентификатор `it`

В определении лямбда-выражения, которое принимает ровно один аргумент, вместо определения имени параметра можно использовать удобную альтернативу — идентификатор `it`. В лямбда-выражениях с одним параметром допустимо использовать как `it`, так и именованный параметр.

Удалите имя параметра и стрелку в начале лямбда-выражения `narrationModifier` и используйте вместо них идентификатор `it`.

Листинг 8.7. Использование идентификатора `it` (`Narrator.kt`)

```
val narrationModifier: (String) -> String = { message →  
    val numExclamationPoints = 3  
    message.uppercase() + "!" .repeat(numExclamationPoints)  
    it.uppercase() + "!" .repeat(numExclamationPoints)  
}  
  
fun narrate(  
    message: String  
) {  
    println(narrationModifier(message))  
}
```

Запустите `Narrator.kt` и убедитесь в том, что все работает, как прежде.

Ключевое слово `it` удобно тем, что не требует имени переменной; с другой стороны, оно недостаточно ясно описывает представляемые данные. Мы рекомендуем при работе с более сложными лямбда-выражениями или с вложенными лямбда-функциями использовать именованные параметры, чтобы сохранить рассудок — не только ваш, но и тех, кто будет читать ваш код впоследствии.

С другой стороны, `it` очень хорошо подходит для коротких выражений. Например, приведенный выше вызов функции `count` для подсчета букв `s` в слове `Mississippi` можно записать более компактно:

```
"Mississippi".count({ it == 's' })
```

Из-за простоты этого примера его логика ясна даже без имени аргумента.

Получение нескольких аргументов

Синтаксис `it` можно использовать в лямбда-функциях, получающих один аргумент, но он недопустим при нескольких аргументах. Тем не менее лямбда-функции могут получать несколько именованных аргументов.

Допустим, вы хотите изменить `narrationModifier`, чтобы тон сообщения показывал отношение рассказчика к повествованию. Выполните код из листинга 8.8 в REPL. (Для простоты не будем вносить это изменение в `NyetHack`.)

Листинг 8.8. Получение второго аргумента (REPL)

```
val loudNarration: (String, String) -> String = { message, tone ->
    when (tone) {
        "excited" -> {
            val numExclamationPoints = 3
            message.uppercase() + "!".repeat(numExclamationPoints)
        }
        "sneaky" -> {
            "$message. The narrator has just blown Madrigal's cover.".uppercase()
        }
        else -> message.uppercase()
    }
}

println(loudNarration("Madrigal cautiously tip-toes through the hallway",
"sneaky"))
```

Нажмите Command-Return (Control-Enter), чтобы выполнить этот код. Вывод в REPL должен выглядеть так:

```
MADRIGAL CAUTIOUSLY TIP-TOES THROUGH THE HALLWAY. THE NARRATOR HAS JUST BLOWN
MADRIGAL'S COVER.
```

Это лямбда-выражение объявляет два параметра, `message` и `tone`, и получает два аргумента при вызове. Так как для выражения определено более одного параметра, идентификатор `it` в нем использоваться не может.

В отличие от обычных функций, лямбда-выражения не могут иметь аргументы по умолчанию. Тип функции — единственная информация, которую Kotlin хранит для таких функций, и в тип функции невозможно включить аргумент по умолчанию, как для обычных функций. Аналогичным образом именованные аргументы нельзя использовать с лямбда-выражениями.

Поддержка автоматического определения типов

Правила автоматического определения типов в языке Kotlin применяются к типам функций точно так же, как и к типам переменных, которые мы ранее рассматривали в книге. Если при объявлении переменной ей в качестве значения присваивается лямбда-выражение, явное определение типа не требуется.

Это означает, что лямбда-функцию без аргументов, которую мы написали выше:

```
val narrationModifier: () -> String = {
    val numExclamationPoints = 3
    message.uppercase() + "!".repeat(numExclamationPoints)
}
```


также можно записать без определения типа:

```
val narrationModifier = {  
    val numExclamationPoints = 3  
    message.uppercase() + "!".repeat(numExclamationPoints)  
}
```

Автоматическое определение типа работает, даже когда лямбда-функция получает один или более аргументов, но есть один нюанс. Компилятору нужно помочь определить типы параметров лямбда-выражения. При использовании механизма автоматического определения типов необходимо указать имена и типы всех параметров в определении лямбда-выражения. Это также означает, что сокращенная запись `it` не может использоваться при автоматическом определении типов.

Обновите переменную `narrationModifier` для использования автоматического определения типов, включив тип параметра.

Листинг 8.9. Использование автоматического определения типов в `narrationModifier` (`Narrator.kt`)

```
val narrationModifier: (String) -> String = {  
    val narrationModifier = { message: String ->  
        val numExclamationPoints = 3  
        it.uppercase() + "!".repeat(numExclamationPoints)  
        message.uppercase() + "!".repeat(numExclamationPoints)  
    }  
  
    fun narrate(  
        message: String  
    ) {  
        println(narrationModifier(message))  
    }  
}
```

Запустите `NyetHack.kt` и убедитесь, что все работает, как прежде.

В сочетании с неявным возвратом автоматическое определение типа функции может затруднить чтение лямбда-выражения. Но если лямбда-выражения достаточно простые и ясные, автоматическое определение типов помогает сделать код более лаконичным.

Более эффективные лямбда-выражения

Взгляните еще раз на файл `Narrator.kt`:

```
val narrationModifier = { message: String ->  
    val numExclamationPoints = 3  
    message.uppercase() + "!".repeat(numExclamationPoints)  
}
```

```
fun narrate(
    message: String
) {
    println(narrationModifier(message))
}
```

Возможно, внимательный читатель заметил один разочаровывающий факт: все, что мы сделали к настоящему моменту, доступно и без лямбда-выражений. Ту же логику можно выразить одной неанонимной функцией:

```
fun narrate(
    message: String
) {
    val numExclamationPoints = 3
    println(message.uppercase() + "!".repeat(numExclamationPoints))
}
```

Не падайте духом: наша работа не напрасна. Лямбда-выражения по-настоящему проявляют себя, когда требуется повлиять на поведение функции. Теперь, когда мы разобрались с основами лямбда-выражений, мы можем реализовать разные варианты настроения для рассказчика в NyetHack.

Выполним ряд изменений: `narrationModifier` преобразуем в `var`, чтобы значение можно было изменить, и удалим настроение рассказчика по умолчанию. Затем создадим новую функцию с именем `changeNarratorMood`, которая случайным образом присваивает переменной новое значение. Для этого необходимо импортировать в программу тип `Random` и его функцию `nextInt`.

Листинг 8.10. Добавление новых вариантов настроения (Narrator.kt)

```
import kotlin.random.Random
import kotlin.random.nextInt

val narrationModifier = { message: String ->
    val numExclamationPoints = 3
    message.uppercase() + "!".repeat(numExclamationPoints)
}

var narrationModifier: (String) -> String = { it }
fun narrate(
    message: String
) {
    println(narrationModifier(message))
}

fun changeNarratorMood() {
    val mood: String
    val modifier: (String) -> String
    when (Random.nextInt(1..4)) {
        1 -> {
            mood = "loud"
```

```

        modifier = { message ->
            val numExclamationPoints = 3
            message.uppercase() + "!".repeat(numExclamationPoints)
        }
    }
    2 -> {
        mood = "tired"
        modifier = { message ->
            message.lowercase().replace(" ", "... ")
        }
    }
    3 -> {
        mood = "unsure"
        modifier = { message ->
            "$message?"
        }
    }
} else -> {
    mood = "professional"
    modifier = { message ->
        "$message."
    }
}
}

narrationModifier = modifier
narrate("The narrator begins to feel $mood")
}

```

Обновите функцию `main` в `NyetHack.kt`, чтобы использовать новые варианты настроения.

Листинг 8.11. Изменение настроения рассказчика (NyetHack.kt)

```

fun main() {
    narrate("A hero enters the town of Kronstadt. What is their name?")
    val heroName = readLine() ?: ""

    changeNarratorMood()
    narrate("$heroName heads to the town square")
}

```

Запустите `NyetHack` несколько раз. Введите произвольное имя героя. Вызов `Random.nextInt(1..4)` возвращает случайное число от 1 до 4. В зависимости от того, какое число было выбрано, вы получите один из следующих вариантов вывода.

Если генератор случайных чисел возвращает 1:

```

A hero enters the town of Kronstadt. What is their name?
Madrigal
THE NARRATOR BEGINS TO FEEL LOUD!!!
MADRIGAL HEADS TO THE TOWN SQUARE!!!

```

Если генератор случайных чисел возвращает 2:

```
A hero enters the town of Kronstadt. What is their name?  
Madrigal  
the... narrator... begins... to... feel... tired  
madrigal... heads... to... the... town... square
```

Если генератор случайных чисел возвращает 3:

```
A hero enters the town of Kronstadt. What is their name?  
Madrigal  
The narrator begins to feel unsure?  
Madrigal heads to the town square?
```

Если генератор случайных чисел возвращает 4:

```
A hero enters the town of Kronstadt. What is their name?  
Madrigal  
The narrator begins to feel professional.  
Madrigal heads to the town square.
```

Теперь `narrate` использует лямбда-выражения в полной мере. Настроение рассказчика можно произвольно изменять, присваивая `narrationModifier` нужное значение. Эту классную возможность не удастся обеспечить без лямбда-выражений: поведение `narrate` можно изменять без изменения реализации.

Такой чрезвычайно мощный прием часто встречается в стандартной библиотеке Kotlin. Вам стоит взять его на вооружение при создании кода.

Определение функции, которая получает функцию

Вы уже видели, как лямбда-выражения способны изменять поведение функций из стандартной библиотеки. Например, функция `count` может получать лямбда-выражение, которое влияет на подсчет символов. Лямбда-выражения также можно передавать вашим собственным функциям.

Параметр функции способен принимать аргументы любого типа, даже те, которые сами являются функциями. Параметр типа функции определяется так же, как и параметр любого другого типа, — в круглых скобках после имени функции с указанием соответствующего типа. Чтобы увидеть, как это работает, добавьте в `narrate` новую функцию для переопределения настроения рассказчика, чтобы иметь возможность применить форматирование для одного конкретного случая.

Добавьте в функцию `narrate` из файла `Narrator.kt` параметр с именем `modifier` и типом `(String) -> String`. Так как параметр предполагается использовать только для относительно редких специальных вызовов, определите для него значение по умолчанию — `narrationModifier`, с которым вы работали до сих пор.

Листинг 8.12. Добавление аргумента-функции в narrate (Narrator.kt)

```
import kotlin.random.Random
import kotlin.random.nextInt

var narrationModifier: (String) -> String = { it }

fun narrate(
    message: String,
    modifier: (String) -> String = { narrationModifier(it) }
) {
    println(narrationModifier(message))
    println(modifier(message))
}
...
```

Параметр `modifier`, добавленный в `narrate`, представляет функцию, которая получает `String` и возвращает `String`. Она служит той же цели, что и `narrationModifier`, но может переопределяться для конкретного вызова `narrate`.

Чтобы опробовать новое лямбда-выражение, добавим немного цвета в мир NyetHack — пусть приветственное сообщение «A hero enters the town of Kronstadt. What is their name?» выводится желтым цветом. Многие терминалы, включая встроенный в IntelliJ, поддерживают простейшее оформление текста с использованием *управляющих последовательностей ANSI*. Управляющие последовательности ANSI уходят корнями в 1970-е годы; с их помощью ANSI-совместимый терминал может предоставлять (чуть более) расширенные средства вывода.

Мы сейчас используем две управляющие последовательности. На первый взгляд они кажутся устрашающими, но не пугайтесь — мы объясним, что делает каждая. Обновим функцию `main` в `NyetHack.kt`, чтобы первое сообщение выводилось желтым цветом.

Листинг 8.13. Вывод приветствия желтым цветом (NyetHack.kt)

```
fun main() {
    narrate("A hero enters the town of Kronstadt. What is their name?",
           { message ->
               // Выводит сообщение желтым цветом
               "\u001b[33;1m$message\u001b[0m"
           })
    val heroName = readLine() ?: ""

    changeNarratorMood()
    narrate("$heroName heads to the town square")
}
```

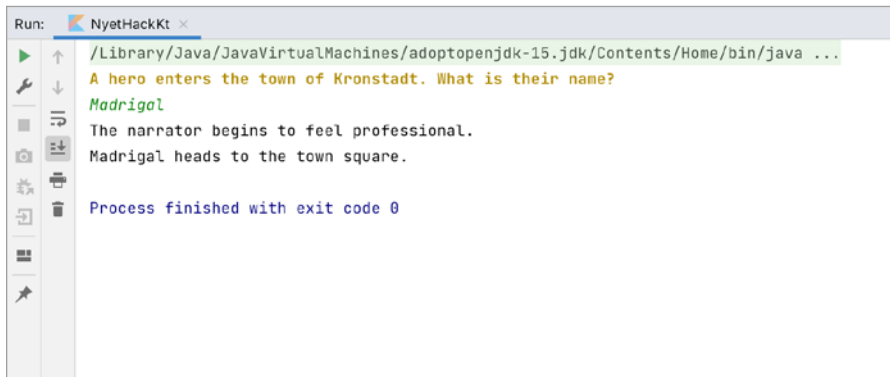
Первая управляющая последовательность ANSI, `\u001b[33;1m`, указывает, что весь следующий за ней текст должен быть желтым. `\u001b` использует синтаксис

экранированных последовательностей Юникода, представленный в главе 6. Квадратная скобка (`[`) показывает начало команды, а `33;1` согласно цветовой схеме терминала соответствует ярко-желтому цвету текста. Последняя часть команды `m` означает, что вы изменяете текстовые режимы. Собирая все воедино, получаем «Изменить стиль текста с текущей точки, чтобы для вывода использовался ярко-желтый цвет».

Так как мы не хотим, чтобы *весь* текст стал желтым, стиль необходимо изменить после приветственного сообщения. Для этого используем служебную последовательность ANSI `\u001b[0m`. Две команды различаются только символами, стоящими между квадратной скобкой (`[`) и `m`. Во второй команде значение `0` возвращает текст к режиму вывода по умолчанию.

Запустите NyetHack и восхититесь ярко-желтыми буквами в первой строке вывода (или см. рис. 8.2 — текст желтый, поверьте на слово!).

Если вы задали для IDE или терминала цветовую схему, отличную от используемой по умолчанию, текст может выводиться другим цветом. Это нормально. Если первая строка напечатана другим цветом, не таким, как остальной текст в консоли, то ваш код работает правильно.



```
Run: NyetHackKt x
/Library/Java/JavaVirtualMachines/adoptopenjdk-15.jdk/Contents/Home/bin/java ...
A hero enters the town of Kronstadt. What is their name?
Madrigal
The narrator begins to feel professional.
Madrigal heads to the town square.

Process finished with exit code 0
```

Рис. 8.2. NyetHack в цвете

Сокращенный лямбда-синтаксис

Когда функция получает тип функции в последнем параметре, круглые скобки вокруг аргумента лямбда-выражения можно опустить. Таким образом, приводившийся ранее пример:

```
"Mississippi".count({ it == 's' })
```

можно записать без круглых скобок:

```
"Mississippi".count { it == 's' }
```

Такой синтаксис более чистый, а вы чуть быстрее добираетесь до важнейших составляющих вызова функции.

Подобное упрощение возможно только при передаче лямбда-выражения в последнем аргументе функции. При написании функций объявляйте параметры с типами функций в последнюю очередь, чтобы вызывающая сторона могла воспользоваться этим синтаксисом.

В NyetHack сокращенная запись может быть применена с функцией. Функция `narrate` ожидает получить два аргумента: строку и функцию. Проведите рефакторинг кода, чтобы аргументы, которые не являются функциями, были заключены в круглые скобки, а последний аргумент — функция — находился вне круглых скобок.

Листинг 8.14. Передача завершающего лямбда-аргумента (NyetHack.kt)

```
fun main() {
    narrate("A hero enters the town of Kronstadt. What is their name?", { message ->
    narrate("A hero enters the town of Kronstadt. What is their name?") { message ->
        // Выводит сообщение желтым цветом
        "\u001b[33;1m$message\u001b[0m"
    }
    }
    val heroName = readLine() ?: ""
    ...
}
```

В реализации самой функции `main` ничего не изменилось; изменился только ее вызов.

Сокращенный синтаксис позволяет писать более чистый код, и мы будем при-
менять его в книге везде, где это возможно.

Встраиваемые функции

Лямбда-функции полезны, потому что они дают больше гибкости при написании программ. Однако за гибкость приходится платить.

Когда вы объявляете лямбда-функцию, она представляется в JVM экземпляром объекта независимо от того, на какой платформе выполняется код Kotlin. Кроме того, во время выполнения ваша программа выделяет память для всех переменных, доступных в лямбда-функции, а это увеличивает затраты памяти. Проще говоря, лямбда-функции повышают расход памяти, что может отразиться на производительности. А потери производительности следует избегать.

К счастью, есть возможность включить оптимизацию, которая избавит от лишних затрат памяти при использовании лямбда-функций в качестве аргументов. Такая возможность называется *встраиванием* (inlining). Встраивание избавляет

от необходимости использовать экземпляр объекта и выделять память для переменных в лямбда-функции.

Чтобы встроить лямбда-функцию, отметьте функцию, которой она передается, ключевым словом `inline`. Добавьте ключевое слово `inline` в определение функции `narrate` из `Narrator.kt`.

Листинг 8.15. Использование ключевого слова `inline` (`Narrator.kt`)

```
...
inline fun narrate(
    message: String,
    modifier: (String) -> String = { narrationModifier(it) }
) {
    println(modifier(message))
}
...
```

После того как ключевое слово `inline` будет добавлено, вместо вызова `narrate` с экземпляром объекта лямбда-функции компилятор скопирует и вставит тело функции в точку вызова. Посмотрите на скомпилированный байт-код функции `main` в `NyetHack.kt`, где происходит вызов (теперь встраиваемой) функции `narrate`:

```
...
public static final void main() {
    String message = "A hero enters the town of Kronstadt.
        What is their name?";
    String var2 = "\u001b[33;1m" + message + "\u001b[0m";
    System.out.println(var2);
    String var10000 = ConsoleKt.readLine();
    if (var10000 == null) {
        var10000 = "";
    }

    String heroName = var10000;
    NarratorKt.changeNarratorMood();
    String message$iv = heroName + " heads to the town square";
    String var4 = (String)NarratorKt.getNarrationModifier().invoke(message$iv);
    System.out.println(var4);
}
...
```

Вместо вызова функции `narrate` ее тело напрямую вставляется в функцию `main`, избавляя от необходимости передавать лямбда-функцию (и создавать еще один экземпляр объекта).

Встраивание функций — простой способ повышения производительности в приложениях, и мы рекомендуем применять его при любой возможности для функций, получающих лямбда-функции.

Все делается так просто, что возникает вопрос: почему функции не встраиваются по умолчанию? В некоторых ситуациях объявление функции с ключевым словом `inline` невозможно. Есть два случая, когда функцию нельзя объявить как встраиваемую.

- Функция является рекурсивной (то есть вызывает сама себя). Рекурсивные функции не могут встраиваться, потому что они должны встраиваться сами в себя, что приведет к бесконечному циклу копирования и вставки.
- Функция использует другие функции или переменные с более ограниченной областью видимости, чем у встраиваемой функции. Тела функций должны полностью вставляться повсюду, где вызывается встраиваемая функция. На вставленный код распространяются те же правила видимости, что и на обычный код. Это означает: если у вас имеется публичная встраиваемая функция, все используемые в ней функции и переменные тоже должны быть публичными. Тем не менее вы можете создать приватную встраиваемую функцию, которая использует приватные функции и переменные.

Существуют и другие причины для пометки функций как встраиваемых помимо лямбда-выражений. Пример мы приведем в главе 18.

Лямбда-выражения и стандартная библиотека Kotlin

Итак, теперь вы знаете основы объявления и вызова функций, использующих лямбда-выражения. Давайте подробнее рассмотрим стандартную библиотеку Kotlin, чтобы понять, как лямбда-выражения применяются в языке. Включите в функцию `main` вызов функции `require`, которая была впервые представлена в главе 7.

Листинг 8.16. Возвращение к `require` (NyetHack.kt)

```
fun main() {
    narrate("A hero enters the town of Kronstadt. What is their name?") { message ->
        // Выводит сообщение желтым цветом
        "\u001b[33;1m$message\u001b[0m"
    }
    val heroName = readLine().?: ""

    val heroName = readLine()
    require(heroName != null && heroName.isNotEmpty()) {
        "The hero must have a name."
    }

    changeNarratorMood()
    narrate("$heroName heads to the town square")
}
```

Как вы уже догадались, фигурные скобки после вызова `require` определяют лямбда-выражение. Этому конкретному лямбда-выражению присвоено имя `lazyMessage` и тип `() -> Any`.

`require` проверяет условие, переданное в первом аргументе. Если условие ложно, выдается исключение `IllegalArgumentException` с сообщением, сгенерированным лямбда-выражением. (Кстати говоря, лямбда-выражение `lazyMessage` не является обязательным. Если значение не передается, по умолчанию используется пустая строка.) В данном случае лямбда-выражение применяется для оптимизации: строка вычисляется только в том случае, если она необходима для выдачи исключения.

Функциональность `lazyMessage` не уникальна для `require`. Собственно, она доступна почти для всех предусловий, о которых мы говорили в главе 7, включая `requireNotNull`, `check`, `checkNotNull` и `assert`. (Функция `error` занимает особое место: это единственная функция проверки предусловий, не имеющая ленивого сообщения. Вместо этого она получает сообщение в `String`.)

В добавление к оптимизации производительности в стандартной библиотеке Kotlin есть ряд функций, позволяющих выражать сложные алгоритмы в одной строке кода.

Допустим, вы хотите присвоить игроку титул, зависящий от его имени, например «Повелитель гласных» (`The Master of Vowels`), если имя содержит много гласных, или «Узнаваемый» (`Identifiable`), если имя состоит из цифр. Тип `String` в Kotlin включает несколько функций с лямбда-аргументами, которые могут пригодиться для принятия подобных решений.

Добавьте в `main` функцию `createTitle`. Сначала проверьте, содержит ли имя игрока более четырех гласных, и если содержит, назначьте ему титул `The Master of Vowels`.

Листинг 8.17. Определение `createTitle` (`NyetHack.kt`)

```
fun main() {
    narrate("A hero enters the town of Kronstadt. What is their name?") { message ->
        // Выводит сообщение желтым цветом
        "\u001b[33;1m$message\u001b[0m"
    }
    val heroName = readLine()
    require(heroName != null && heroName.isNotEmpty()) {
        "The hero must have a name."
    }
    changeNarratorMood()
    narrate("$heroName heads to the town square")
    narrate("$heroName, ${createTitle(heroName)}, heads to the town square")
}

private fun createTitle(name: String): String {
```

```
return when {  
    name.count { it.lowercase() in "aeiou" } > 4 -> "The Master of Vowel"  
    else -> "The Renowned Hero"  
}  
}
```

Запустите NyetHack и введите имя с большим количеством гласных, например Aurelia. Результат должен выглядеть примерно так (в зависимости от текущего настроения рассказчика):

```
A hero enters the town of Kronstadt. What is their name?  
Aurelia  
The narrator begins to feel professional.  
Aurelia, The Master of Vowels, heads to the town square.
```

Другая полезная строковая функция — `all` — проверяет, что каждый символ строки удовлетворяет заданному предикату (вроде `(Char) -> Boolean`). Также существует функция `none`, которая проверяет обратное условие — что ни один символ строки не удовлетворяет заданному предикату.

Определим еще два титула для игрока. Если имя игрока состоит только из цифр, ему должен быть присвоен титул `The Identifiable` (Узнаваемый), а если оно не содержит ни одной буквы — титул `The Witness Protection Member` (Участник программы по защите свидетелей). Конечно, цифры тоже не являются буквами, поэтому в выражении `when` необходимо тщательно упорядочить условия, чтобы все титулы назначались по задуманным вами правилам.

Листинг 8.18. Добавление титулов (NyetHack.kt)

```
...  
private fun createTitle(name: String): String {  
    return when {  
        name.all { it.isDigit() } -> "The Identifiable"  
        name.none { it.isLetter() } -> "The Witness Protection Member"  
        name.count { it.lowercase() in "aeiou" } > 4 -> "The Master of Vowels"  
        else -> "The Renowned Hero"  
    }  
}
```

Протестируйте свой код с входными данными 11, **** и т. д., чтобы убедиться в том, что игрокам присваиваются правильные титулы.

Функции `all`, `none` и `count` позволяют компактно выполнять такие проверки. Без этих функций разработчику пришлось бы самостоятельно реализовать эти классы (вероятно, для этого потребуется 5–10 строк кода на каждое условие).

Стандартная библиотека Kotlin включает множество функций, которые делают ваш код более компактным и понятным. Вы больше узнаете об этих функциях и получите некоторое представление о способах функционального программирования в главе 11.

Для любознательных: ссылки на функции

До сих пор вы объявляли лямбда-выражения для передачи функции в виде аргумента другой функции. Сделать это можно иначе: передать *ссылку на функцию*. Ссылка на функцию преобразует обычную функцию, определенную с ключевым словом `fun`, в значение с типом функции. Ссылку на функцию можно использовать везде, где допускается лямбда-выражение.

Допустим, вы хотите выделить из кода лямбда-выражение, выводящее текст желтым цветом. Для этого реализация лямбда-выражения извлекается в отдельную функцию и применяется синтаксис ссылки на функцию (выделен в примере кода ниже):

```
fun main() {
    narrate(
        "A hero enters the town of Kronstadt. What is their name?",
        ::makeYellow
    )
    ...
}

private fun makeYellow(message: String) = "\u001b[33;1m$message\u001b[0m"

private fun createTitle(name: String): String {
    ...
}
```

Чтобы получить ссылку на функцию, примените оператор `::` к имени этой функции. Ссылки на функции могут пригодиться во многих ситуациях. Если у вас есть именованная функция, соответствующая требованиям к параметру, который должен получать аргумент-функцию, ссылка на функцию позволит использовать ее вместо определения лямбда-выражения. А может быть, вы захотите передать в аргументе функцию из стандартной библиотеки Kotlin. Ссылки на функции позволяют решать такие задачи еще компактнее, чем лямбда-выражения.

Для любознательных: захват лямбда-выражений

В языке Kotlin лямбда-функция может изменять переменные и ссылаться на них вне своей области видимости. Лямбда-функция *захватывает* переменные, объявленные за ее пределами, то есть сохраняет ссылки на эти переменные, как было показано на примере первого лямбда-выражения в функции `narrate`.

Для демонстрации такого свойства лямбда-функций давайте добавим еще одно настроение рассказчика в функцию `changeNarratorMood`:

Листинг 8.19. Изменение переменных из лямбда-функции (Narrator.kt)

```

...
fun changeNarratorMood() {
    val mood: String
    val modifier: (String) -> String

    when (Random.nextInt(1..4)) {
    when (Random.nextInt(1..5)) {
        ...
        3 -> {
            mood = "unsure"
            modifier = { message ->
                "$message?"
            }
        }
        4 -> {
            var narrationsGiven = 0
            mood = "like sending an itemized bill"
            modifier = { message ->
                narrationsGiven++
                "$message.\n(I have narrated $narrationsGiven things)"
            }
        }
        else -> {
            mood = "professional"
            modifier = { message ->
                "$message."
            }
        }
    }

    narrationModifier = modifier
    narrate("The narrator begins to feel $mood")
}

```

Новая переменная настроения учитывает количество вариантов повествования. Прежде чем выполнять код, ненадолго остановитесь и спросите себя: что будет делать этот код? Чтобы подтвердить свои догадки, запустите NyetHack (возможно, программу придется пару раз перезапустить, чтобы добраться до нового настроения рассказчика). Результат будет выглядеть примерно так:

```

A hero enters the town of Kronstadt. What is their name?
Madrigal
The narrator begins to feel like sending an itemized bill.
(I have narrated 1 things)
Madrigal, The Renowned Hero, heads to the town square.
(I have narrated 2 things)

```

Ваша догадка подтвердилась? Разберемся, что здесь происходит.

Хотя переменная `narrationsGiven` определяется за пределами лямбда-функции, последняя может обращаться к переменной и изменять ее. Таким образом, значение `narrationsGiven` увеличивается с 0 до 1 и с 1 до 2.

Задание: новые титулы и настроения

NyetHack в настоящее время поддерживает пять вариантов настроения и четыре титула, которые могут назначаться игроку, но полету фантазии нет предела. Проявите творческий подход и добавьте любые новые варианты настроения и титулы на свое усмотрение.

Вот несколько возможных идей для настроения.

- Ленивое: рассказчик выдает только первую часть выводимого сообщения. (Подсказка: воспользуйтесь функциями `take` или `substring` типа `String`.)
- Таинственное: рассказчик использует шифр leet, заменяя буквы похожими цифрами или знаками. Например, L заменяем на 1; E — на 3; T — на 7. (Подсказка: присмотритесь к функции `replace` типа `String`. Есть версия функции `replace`, которая получает во втором параметре лямбда-функцию.)
- Поэтическое: рассказчик делает между словами паузы (вставляем разрывы строк), чтобы придать тексту поэтический стиль. К сожалению, из-за сложности языка и размера у рассказчика вряд ли получится шедевр. (Подсказка: возможных решений много, но как вариант можно объединить функцию `replace` с классом `Random`, который мы уже использовали ранее в этой главе.)

А вот несколько идей для титулов, которые можно назначать игрокам.

- «Выдающийся»: назначается игроку, если все буквы в его имени записаны в верхнем регистре.
- «Пространный»: назначается игроку, если в его имени слишком много букв (пороговое значение для «слишком много» выберите сами).
- «Носитель палиндрома»: назначается игроку, если его имя является палиндромом. (Подсказка: обратите внимание на функцию `reverse` класса `String`. Не забудьте, что в строках учитывается регистр символов.)

9. Списки и множества

Работа с группой связанных значений важна для многих программ. Например, программа может оперировать списками книг, достопримечательностей на туристическом маршруте, блюд в меню или остатками денег на счетах постоянных посетителей таверны. *Коллекции* позволяют работать с группами значений и передавать их как аргументы в функции.

В следующих двух главах мы рассмотрим наиболее часто применяемые типы коллекций: `List` (список), `Set` (множество) и `Map` (ассоциативный массив). Как многие другие типы переменных, с которыми мы познакомились в главе 2, списки, множества и ассоциативные массивы делятся на две категории: изменяемые и доступные только для чтения. В этой главе мы поговорим о списках и множествах.

Ваш герой в `NyetHack` только что отправился в дальнее путешествие в город Кронштадт. Чтобы отдохнуть и пообщаться с жителями города, он решает посетить местную таверну. Модель таверны в `NyetHack` мы создадим при помощи коллекций.

Когда вы завершите работу, таверна обзаведется меню с солидным списком блюд, а также кучей посетителей, жаждущих потратить деньги. Но прежде чем браться за построение модели таверны, необходима небольшая подготовка. Создайте новый файл с именем `Tavern.kt` и задайте имя хозяина и название таверны.

Листинг 9.1. Основание таверны (`Tavern.kt`)

```
private const val TAVERN_MASTER = "Taernyl"
private const val TAVERN_NAME = "$TAVERN_MASTER's Folly"

fun visitTavern() {

}
```

Теперь снова обновите функцию `main` в файле `NyetHack.kt`. В конец функции будет вставлен вызов `visitTavern`. А поскольку имя героя используется в `NyetHack` постоянно, его следует сделать переменной верхнего уровня. Кроме того, чтобы функция `main` не усложнялась и не разрасталась чрезмерно, мы выделим приветственное сообщение и обработку ввода для имени героя в отдельную функцию.

В этой главе программу `NyetHack` придется запускать неоднократно. Вводить имя героя каждый раз было бы утомительно, как и терпеть перепады настройки

рассказчика. Чтобы сосредоточиться на разработке, мы прокомментируем реализацию `readLine` (вместо этого присвоим имени игрока значение по умолчанию), а также вызов `changeNarratorMood`.

Тем не менее удалять этот код не стоит. Мы уберем комментарий в главе 19, когда будем доводить `NyetHack` до ума.

Листинг 9.2. Подготовка функции `main` (`NyetHack.kt`)

```
var heroName: String = ""

fun main() {
narrate("A hero enters the town of Kronstadt. What is their name?")
    { message ->
        // Выводит сообщение желтым цветом
        "\u001b[33;1m$message\u001b[0m"
    }

    val heroName = readLine()
    require(heroName != null && heroName.isNotEmpty()) {
        "The hero must have a name."
    }
    heroName = promptHeroName()
    changeNarratorMood()
    // changeNarratorMood()
    narrate("$heroName, ${createTitle(heroName)}, heads to the town square")
    visitTavern()
}

private fun promptHeroName(): String {
    narrate("A hero enters the town of Kronstadt. What is their name?") { message ->
        // Выводит message желтым цветом
        "\u001b[33;1m$message\u001b[0m"
    }
    /*val input = readLine()
    require(input != null && input.isNotEmpty()) {
        "The hero must have a name."
    }
    return input*/
    println("Madrigal")
    return "Madrigal"
}
...
```

Здесь вы видите новый способ комментирования кода. Уже знакомая вам конструкция `//` выводит комментарий в одной строке. С маркерами `/*` и `*/` все символы, заключенные между ними, интерпретируются как комментарий, даже если он занимает несколько строк. Хотя комментарии обоих видов пишутся для пояснения работы кода, программисты также часто используют комментарии, чтобы закрыть код, который временно не должен выполняться.

Запустите NyetHack, чтобы проверить наличие ошибок. Результат должен выглядеть так:

```
A hero enters the town of Kronstadt. What is their name?  
Madrigal  
Madrigal, The Renowned Hero, heads to the town square
```

Пришло время реализовать `visitTavern`, чтобы таверна могла распахнуть двери перед своими первыми посетителями.

Списки

Список (`List`) содержит упорядоченную коллекцию значений и может содержать дубликаты. Это самая распространенная разновидность коллекций, что объясняется относительным удобством использования и общей простотой по сравнению с другими структурами данных. Благодаря упорядоченности списки отлично подходят для хранения такой информации, как очередь посетителей таверны или список блюд в меню.

Чтобы открыть таверну, в файле `Tavern.kt` добавьте список посетителей с помощью функции `listOf`. Она возвращает список, доступный только для чтения, с элементами, полученными из аргументов. Создайте список из трех посетителей.

Листинг 9.3. Создание списка посетителей (Tavern.kt)

```
private const val TAVERN_MASTER = "Taernyl"  
private const val TAVERN_NAME = "$TAVERN_MASTER's Folly"  
  
fun visitTavern() {  
    narrate("$heroName enters $TAVERN_NAME")  
  
    val patrons: List<String> = listOf("Eli", "Mordoc", "Sophie")  
    println(patrons)  
}
```

Снова запустите NyetHack. Результат должен выглядеть так:

```
A hero enters the town of Kronstadt. What is their name?  
Madrigal  
Madrigal, The Renowned Hero, heads to the town square  
Madrigal enters Taernyl's Folly  
[Eli, Mordoc, Sophie]
```

До этого момента вы создавали переменные разных типов, просто объявляя их. Но чтобы получить коллекцию, нужно выполнить два действия: создать коллекцию (в данном случае это список посетителей) и добавить в нее содержимое (имена посетителей). Kotlin предоставляет функции, которые позволяют делать это одновременно, такие как `listOf`.

После того как список будет создан, рассмотрим тип `List` подробнее.

Хотя механизм автоматического определения типов распознает списки, мы включили информацию о типе — `val patrons: List<String>`, чтобы она была видна в контексте обсуждения. Угловые скобки в `List<String>` обозначают *параметризованный тип*. Он сообщает компилятору тип значений, которые будут содержаться в списке, — в нашем случае это `String`. Изменение параметра типа изменяет тип значений, которые компилятор разрешит хранить в списке.

Если вы попытаетесь поместить целое число в `patrons`, компилятор этого не допустит. Попробуйте добавить число в объявленный список.

Листинг 9.4. Добавление целого числа в список строк (Tavern.kt)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")
    val patrons: List<String> = listOf("Eli", "Mordoc", "Sophie", 1)
    println(patrons)
}
```

IntelliJ предупредит, что целочисленное значение не соответствует ожидаемому типу `String`. Параметр типа необходимо использовать с `List`, потому что сам тип `List` является *обобщенным* (generic). Это означает, что список может хранить данные любого типа, включая текстовые — строки (как в случае с `patrons`) или символы; числовые значения — целые и числа с плавающей точкой; и даже данные новых типов, определяемых пользователем. (Об обобщенных типах подробнее рассказано в главе 18.)

Отмените последнее изменение с помощью команды отмены в IntelliJ (Command-Z (Ctrl-Z)) или удалите целое число из списка.

Листинг 9.5. Корректировка списка (Tavern.kt)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons: List<String> = listOf("Eli", "Mordoc", "Sophie", 1)
    println(patrons)
}
```

Обращение к элементам списка

К любому элементу списка можно обратиться по его индексу при помощи функции `get` или — более распространенный способ — с помощью оператора `[ ]`. Индексирование списков начинается с нуля, так что строка "Eli" в настоящее время хранится в `patrons` с индексом 0, а строка "Sophie" — с индексом 2.

Измените код `Tavern.kt`, чтобы он выводил только имя первого посетителя. Также удалите явную информацию о типе из объявления `patrons`. Теперь, когда вы узнали, каким типом параметризуется список `List`, воспользуемся механизмом автоматического определения типов для написания более чистого кода.

Листинг 9.6. Обращение к первому посетителю (`Tavern.kt`)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons: List<String> = listOf("Eli", "Mordoc", "Sophie")
    println(patrons[0])
}
```

Запустите `NyetHack`. Вы увидите в консоли имя первого посетителя, `Eli`.

Для обращения к элементу списка можно также использовать функцию `get`, получающую индекс. Вызов `patrons.get(0)` работает так же, как `patrons[0]`, — оба вызова возвращают имя первого посетителя в списке.

Границы индексов и безопасные обращения по индексу

Обращения к элементу по индексу требуют осторожности, потому что попытка получить элемент с несуществующим индексом — допустим, четвертый элемент в списке, содержащем только три элемента, — приведет к возникновению исключения `ArrayIndexOutOfBoundsException`.

Попробуйте сделать это в `Kotlin REPL`. (Можно скопировать первую строку кода из `Tavern.kt`.)

Листинг 9.7. Обращение по несуществующему индексу (`REPL`)

```
val patrons = listOf("Eli", "Mordoc", "Sophie")
patrons[4]
```

В результате вы получите исключение `ArrayIndexOutOfBoundsException` с сообщением о том, что индекс 4 выходит за границу списка.

Так как обращение к элементу по индексу может привести к выдаче исключения, `Kotlin` предоставляет функции безопасного обращения по индексу, которые позволят решить эту проблему. Например, есть удобные функции для обращения к первому или последнему элементу:

```
patrons.first() // Eli
patrons.last()  // Sophie
```

Другие функции могут указывать действие, которое должно выполняться вместо выдачи исключения, если номер индекса выходит за границу списка. Так, одна

из функций безопасного обращения по индексу `getOrElse` получает два аргумента: первый — запрашиваемый индекс (в круглых, а не в квадратных скобках), второй — лямбда-функция, которая генерирует значение по умолчанию вместо исключения, если запрашиваемого индекса нет в границах списка.

Попробуйте ввести в REPL следующий пример:

Листинг 9.8. Тестирование `getOrElse` (REPL)

```
patrons.getOrElse(4) { "Unknown Patron" }  
"Unknown Patron"
```

В этот раз результат — `Unknown Patron` (неизвестный посетитель). Так как запрашиваемый индекс отсутствует в списке, для получения значения по умолчанию было вызвано лямбда-выражение.

Другая функция безопасного доступа, `getOrNull`, возвращает `null` вместо исключения. При использовании `getOrNull` вы должны решить, что делать со значением `null`, — как было показано в главе 7. Один из вариантов — связать `null` со значением по умолчанию. Попробуйте использовать `getOrNull` с оператором `?:` в REPL.

Листинг 9.9. Тестирование `getOrNull` (REPL)

```
patrons.getOrNull(4) ?: "Unknown Patron"  
"Unknown Patron"
```

Вы снова получите результат `Unknown Patron`.

В Kotlin есть много «`orNull`»-версий функций для работы с коллекциями. Например, существуют функции `firstOrNull` и `lastOrNull`, которые возвращают `null` при пустом списке. Помните о подобных граничных случаях при работе со списками и другими коллекциями.

Проверка содержимого списка

В таверне есть темные углы и секретные помещения. К счастью, у хозяина заведения острый глаз, и он скрупулезно записывает всех приходящих и уходящих. Если спросить у него, есть ли сейчас в таверне конкретный посетитель, то он сможет ответить, заглянув в список.

Измените код `Tavern.kt` и используйте функцию `contains` для проверки присутствия определенного посетителя.

Листинг 9.10. Проверка присутствия посетителя (`Tavern.kt`)

```
...  
fun visitTavern() {  
    narrate("$heroName enters $TAVERN_NAME")  
  
    val patrons = listOf("Eli", "Mordoc", "Sophie")
```

```
println(patrons[0])

val eliMessage = if (patrons.contains("Eli")) {
    "$TAVERN_MASTER says: Eli's in the back playing cards"
} else {
    "$TAVERN_MASTER says: Eli isn't here"
}
println(eliMessage)
}
```

Запустите NyetHack. Так как `patrons` содержит "Eli", вы увидите ответ хозяина заведения в конце консольного вывода: Taernyl says: Eli's in the back playing cards.

Кстати говоря, функция `contains` выполняет структурное сравнение элементов в списке, как и оператор структурного равенства. (Если вы забыли, чем структурное равенство отличается от ссылочного, обратитесь к разделу «Сравнение строк» в главе 6.)

Также можно воспользоваться функцией `containsAll`, чтобы проверить присутствие сразу нескольких посетителей. Измените код, чтобы узнать у хозяина заведения, присутствуют ли Sophie и Mordoc одновременно.

Листинг 9.11. Проверка нескольких посетителей (Tavern.kt)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons = listOf("Eli", "Mordoc", "Sophie")

    val eliMessage = if (patrons.contains("Eli")) {
        "$TAVERN_MASTER says: Eli's in the back playing cards"
    } else {
        "$TAVERN_MASTER says: Eli isn't here"
    }
    println(eliMessage)

    val othersMessage = if (patrons.containsAll(listOf("Sophie", "Mordoc"))) {
        "$TAVERN_MASTER says: Sophie and Mordoc are seated by the stew kettle"
    } else {
        "$TAVERN_MASTER says: Sophie and Mordoc aren't with each other right now"
    }
    println(othersMessage)
}
```

Запустите NyetHack. Вы увидите следующий вывод:

```
...
Madrigal enters Taernyl's Folly
Taernyl says: Eli's in the back playing cards
Taernyl says: Sophie and Mordoc are seated by the stew kettle
```

Изменение содержимого списка

Если посетитель приходит или уходит из таверны, внимательный хозяин заведения должен добавить или убрать имя посетителя из переменной `patrons`. В настоящее время это невозможно.

`listOf` возвращает список, доступный только для чтения, в котором нельзя менять содержимое: вы не сможете добавить, убрать, обновить или заменить данные. Доступные только для чтения списки — хорошее решение, потому что они предотвращают досадные ошибки (например, хозяин заведения не выгонит посетителя на улицу, случайно убрав его имя из списка).

Доступность списка только для чтения не имеет ничего общего с ключевым словом `val` или `var`, использованным для объявления переменной списка. Заменяв `val` в объявлении переменной `patrons` (как сейчас) на `var`, вы не сделаете список доступным для изменения. Вам просто будет разрешено присвоить переменной `patrons` другое значение, то есть создать новый список.

Изменяемость списка определяется *типом* списка, который и определяет, возможно или нет изменять элементы. Поскольку посетители постоянно приходят и уходят, мы должны поменять тип `patrons`, чтобы разрешить обновление. В языке Kotlin список с возможностью изменения элементов называется *изменяемым списком*. Он имеет тип `MutableList`, и для его создания необходимо вызвать функцию `mutableListOf`.

Измените код `Tavern.kt` и используйте `mutableListOf` вместо `listOf`. Изменяемые списки поддерживают множество функций для добавления, удаления и обновления содержимого. Смоделируйте приход и уход при помощи функций `add` и `remove`.

Листинг 9.12. Создание изменяемого списка посетителей (Tavern.kt)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons = listOf("Eli", "Mordoc", "Sophie")
    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    ...
    println(othersMessage)

    narrate("Eli leaves the tavern")
    patrons.remove("Eli")
    narrate("Alex enters the tavern")
    patrons.add("Alex")
    println(patrons)
}
```

`List` также предоставляет функции для быстрого переключения между версиями (изменяемой и доступной только для чтения): `toList` и `toMutableList`. Например, вы можете создать версию изменяемого списка `patrons`, доступную только для чтения, вызвав `toList`:

```
val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
val readOnlypatrons = patrons.toList()
```

Запустите `NyetHack`. В консоли должно появиться следующее сообщение:

```
...
Eli leaves the tavern
Alex enters the tavern
[Mordoc, Sophie, Alex]
```

Новый элемент `Alex` добавлен в конец списка. Также можно добавить имя посетителя в конкретное место списка. Например, если VIP-гость придет в таверну, хозяин заведения может пропустить его вне очереди.

Добавьте VIP-гостя — пусть это будет посетитель с таким же именем Алекс (`Alex`) — в начало списка посетителей. (Алекс хорошо известен в городе и пользуется такими привилегиями, как обслуживание вне очереди, что, конечно, не нравится другому Алексу.) Список может содержать несколько элементов с одинаковым значением (как, например, два посетителя с одинаковыми именами), поэтому добавление еще одного Алекса — не проблема для списка.

Листинг 9.13. Добавление еще одного посетителя `Alex` (`Tavern.kt`)

```
...
fun visitTavern() {
    ...
    narrate("Eli leaves the tavern")
    patrons.remove("Eli")
    narrate("Alex enters the tavern")
    patrons.add("Alex")
    println(patrons)
    narrate("Alex (VIP) enters the tavern")
    patrons.add(0, "Alex")
    println(patrons)
}
```

Снова запустите `NyetHack`. В консоли должно появиться следующее сообщение:

```
...
[Mordoc, Sophie, Alex]
Alex (VIP) enters the tavern
[Alex, Mordoc, Sophie, Alex]
```

Допустим, важный гость `Alex` решил сменить имя на `Alexis`. Исполнить его желание можно, изменив `patrons` при помощи оператора присваивания (`[] =`), то есть присвоив строке с первым индексом новое значение.

Листинг 9.14. Модификация изменяемого списка оператором присваивания (Tavern.kt)

```
...
fun visitTavern() {
    ...
    narrate("Eli leaves the tavern")
    patrons.remove("Eli")
    narrate("Alex enters the tavern")
    patrons.add("Alex")
    narrate("Alex (VIP) enters the tavern")
    patrons.add(0, "Alex")
    patrons[0] = "Alexis"
    println(patrons)
}
```

Запустите NyetHack.kt. Вы увидите, что список `patrons` обновился и содержит новое имя `Alexis`.

```
...
[Mordoc, Sophie, Alex]
Alex (VIP) enters the tavern
[Alexis, Mordoc, Sophie, Alex]
```

Функции, которые изменяют содержимое изменяемых списков, называют *мутаторами*. В табл. 9.1 перечислены самые распространенные мутаторы для списков.

Таблица 9.1. Мутаторы изменяемых списков

Функция	Описание	Примеры
<code>[]=</code> (оператор присваивания)	Присваивает значение по индексу; выдает исключение, если индекс не существует	<pre>val patrons = mutableListOf("Eli", "Mordoc", "Sophie") patrons[4] = "Reggie" IndexOutOfBoundsException</pre>
<code>add</code>	Добавляет элемент в конец списка, увеличивая размер на один элемент	<pre>val patrons = mutableListOf("Eli", "Mordoc", "Sophie") patrons.add("Reggie") [Eli, Mordoc, Sophie, Reggie] patrons.size 4</pre>
<code>add</code> (по индексу)	Добавляет элемент в список по индексу, увеличивая размер на один элемент. Выдает исключение, если индекс не существует	<pre>val patrons = mutableListOf("Eli", "Mordoc", "Sophie") patrons.add(0, "Reggie") [Reggie, Eli, Mordoc, Sophie] patrons.add(5, "Sophie") IndexOutOfBoundsException</pre>

Функция	Описание	Примеры
<code>addAll</code>	Добавляет все элементы из другой коллекции, если они того же типа	<pre>val patrons = mutableListOf("Eli", "Mordoc", "Sophie") patrons. addAll(listOf("Reginald", "Alex")) <i>[Eli, Mordoc, Sophie, Reginald, Alex]</i></pre>
<code>+=</code> (оператор сложения с присваиванием)	Добавляет элемент или коллекцию элементов в список	<pre>mutableListOf("Eli", "Mordoc", "Sophie") += "Reginald" <i>[Eli, Mordoc, Sophie, Reginald]</i> mutableListOf("Eli", "Mordoc", "Sophie") += listOf("Alex", "Shruti") <i>[Eli, Mordoc, Sophie, Alex, Shruti]</i></pre>
<code>-=</code> (оператор вычитания с присваиванием)	Удаляет элемент или коллекцию элементов из списка	<pre>mutableListOf("Eli", "Mordoc", "Sophie") -= "Eli" <i>[Mordoc, Sophie]</i> val patrons = mutableListOf("Eli", "Mordoc", "Sophie") patrons -= listOf("Eli", "Mordoc") <i>[Sophie]</i></pre>
<code>clear</code>	Удаляет все элементы из списка	<pre>mutableListOf("Eli", "Mordoc", Sophie).clear() <i>[]</i></pre>
<code>remove</code>	Удаляет элемент из списка	<pre>val patrons = mutableListOf("Eli", "Mordoc", "Sophie") patrons.remove("Mordoc") <i>[Eli, Sophie]</i></pre>
<code>removeAll</code>	Удаляет все элементы другой коллекции из списка	<pre>val patrons = mutableListOf("Eli", "Mordoc", "Sophie") patrons.removeAll(listOf("Eli", "Mordoc")) <i>[Sophie]</i></pre>

Итерация

Хозяин заведения приветствует каждого посетителя, потому что это положительно сказывается на репутации таверны. Списки поддерживают множество разных функций, позволяющих выполнить некоторое действие для каждого элемента списка. Эта возможность называется *итерацией*.

Для выполнения итерации со списком можно воспользоваться циклом `for`. Цикл работает по принципу «для каждого элемента в списке сделай то-то». Вы задаете имя элемента, а компилятор Kotlin автоматически определяет его тип.

Измените код `Tavern.kt`, чтобы он выводил сообщение с приветствием для каждого посетителя. (Также удалите код, который модифицирует и выводит `patrons`, чтобы убрать лишнее из вывода в консоли.)

Листинг 9.15. Перебор элементов списка `patrons` в цикле `for` (`Tavern.kt`)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    ...
    println(othersMessage)

    narrate("Eli leaves the tavern")
    patrons.remove("Eli")
    narrate("Alex enters the tavern")
    patrons.add("Alex")
    println(patrons)
    narrate("Alex (VIP) enters the tavern")
    patrons.add(0, "Alex")
    patrons[0] = "Alexis"
    println(patrons)

    for (patron in patrons) {
        println("Good evening, $patron")
    }
}
```

Запустите `NyetHack`, и хозяин заведения поприветствует каждого посетителя по имени:

```
...
Good evening, Eli
Good evening, Mordoc
Good evening, Sophie
```

В этом случае, из-за того что `patrons` принадлежит типу `MutableList<String>`, `patron` получит тип `String`. В теле цикла `for` любой код, воздействующий на `patron`, будет применен ко всем элементам в `patrons`.

Обратите внимание на ключевое слово `in`:

```
for (patron in patrons) { ... }
```

Оно задает объект, по которому происходит перебор в цикле `for`.

Во многих языках, включая Java, C, C# и JavaScript, этот синтаксис цикла называется циклом `foreach`. Возможно, вам знаком другой синтаксис, который в коде Java и C обычно имеет вид

```
for (int i = 0; i < patrons.size; i++)
```

В Kotlin этот синтаксис не поддерживается. Единственная разновидность циклов `for`, поддерживаемая в Kotlin, — циклы `foreach`. Если вам не нужно перебирать конкретный набор чисел, используйте тип `Range`, о котором речь шла в главе 3. Интервалы поддерживают различные виды настройки и многие особенности поведения, присущие традиционным циклам `for`.

```
// Выводит имя каждого посетителя
for (i in 0 until patrons.size) {
    println(patrons[i])
}
// Выводит имена посетителей через одного в обратном порядке
for (i in patrons.size - 1 downTo 0 step 2) {
    println(patrons[i])
}
```

Возможно, вас интересует: не приведет ли такая разновидность циклов к потерям производительности? Не бойтесь. В процессе компиляции кода Kotlin компилятор оптимизирует циклы `foreach` в традиционные циклы `for`. Компилятору также хватает сообразительности, чтобы реализовать циклы `foreach` на базе `Range` без создания экземпляра `Range` во время выполнения.

Цикл `for` прост и легко читается, но существует и другой вариант: `List` и `MutableList` также поддерживают функцию `forEach`.

Функция `forEach` перебирает все элементы списка — последовательно от начала до конца — и передает каждый элемент лямбда-выражению, переданному в аргументе.

Заменим цикл `for` функцией `forEach`.

Листинг 9.16. Перебор списка `patrons` в `forEach` (Tavern.kt)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    ...
    println(othersMessage)
```

```

for (patron in patrons) {
    println("Good evening, $patron")
}
patrons.forEach { patron ->
    println("Good evening, $patron")
}
}

```

Запустите NyetHack, и вы увидите тот же вывод, что и раньше. Цикл `for` и функция `forEach` функционально эквивалентны.

Цикл `for` и функция `forEach` работают с индексами неявно. Если вам требуется получить индекс каждого элемента в списке, используйте `forEachIndexed`. Измените код `Tavern.kt` и используйте функцию `forEachIndexed`, чтобы вывести место каждого посетителя в очереди.

Листинг 9.17. Вывод позиции строки в списке с помощью `forEachIndexed` (`Tavern.kt`)

```

...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    ...
    println(othersMessage)

    patrons.forEach { patron ->
        patrons.forEachIndexed { index, patron ->
            println("Good evening, $patron - you're #${index + 1} in line")
        }
    }
}

```

Снова запустите NyetHack, чтобы увидеть посетителей в очереди:

```

...
Good evening, Eli - you're #1 in line
Good evening, Mordoc - you're #2 in line
Good evening, Sophie - you're #3 in line

```

В общем случае мы предпочитаем функцию `forEach` циклам. На наш взгляд, она лучше читается и идеально работает в сочетании с операциями функционального программирования, которые мы рассмотрим в главе 11. Кроме того, когда логика перебора требует индекс каждого элемента в списке, функция `forEachIndexed` является самым элегантным решением.

Функции `forEach` и `forEachIndexed` доступны и для некоторых других типов Kotlin. Эта категория типов называется `Iterable` (итерируемые) и включает `List`, `Set`, `Map`, `IntRange` (диапазоны типа `0..9`, которые мы показали в главе 3), а также другие типы коллекций. Итерируемые типы поддерживают итерацию — другими словами, они позволяют выполнить перебор хранимых элементов и совершить некоторые действия с каждым из них.

Некоторые другие типы Kotlin, например `String` также поддерживают эти пространственные функции перебора, хотя они и не реализованы как `Iterable`. Такие функции заслуживают вашего внимания. В главе 11 мы продемонстрируем некоторые эффективные приемы, часто используемые с типами `Iterable` для работы с данными в вашем коде.

Итак, хозяин заведения поприветствовал гостей и может принимать заказы. Допустим, каждый посетитель хочет заказать фирменный напиток `Dragon's Breath`. Для этого создайте новую функцию `placeOrder` и вызовите ее из лямбда-выражения, передаваемого функции `forEachIndexed`. В результате все посетители из списка закажут напиток.

Листинг 9.18. Моделирование нескольких заказов (Tavern.kt)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    ...
    println(othersMessage)

    patrons.forEachIndexed { index, patron ->
        println("Good evening, $patron - you're #${index + 1} in line")
        placeOrder(patron, "Dragon's Breath")
    }
}

private fun placeOrder(patronName: String, menuItemName: String) {
    narrate("$patronName speaks with $TAVERN_MASTER to place an order")
    narrate("$TAVERN_MASTER hands $patronName a $menuItemName")
}
```

Запустите `NyetHack` и посмотрите, как таверна оживает с приходом трех посетителей, заказывающих `Dragon's Breath`:

```
...
Good evening, Eli - you're #1 in line
Eli speaks with Taernyl to place an order
Taernyl hands Eli a Dragon's Breath
Good evening, Mordoc - you're #2 in line
Mordoc speaks with Taernyl to place an order
Taernyl hands Mordoc a Dragon's Breath
Good evening, Sophie - you're #3 in line
Sophie speaks with Taernyl to place an order
Taernyl hands Sophie a Dragon's Breath
```

Итерируемые коллекции поддерживают разнообразные функции, позволяющие определять действия, которые необходимо выполнить для каждого элемента коллекции. Об `Iterable` и других функциях итераций мы подробнее расскажем в главе 11.

Чтение файла в список

Разнообразие украшает жизнь, и хозяин заведения знает, что посетители приветствуют богатое меню. На данный момент Dragons' Breath — единственное, что предлагается в таверне. Настало время это исправить и расширить меню.

Чтобы сэкономить ваше время, мы подготовили меню в текстовом файле, который можно загрузить в NyetHack. Файл содержит несколько названий блюд и напитков, а также дополнительные метаданные для каждого из них — они понадобятся в следующей главе.

Сначала создайте новую папку для данных: щелкните правой кнопкой мыши на проекте NyetHack в окне инструментов проекта и выберите **New ▸ Directory** (рис. 9.1). Присвойте папке имя **data**.

Загрузите данные меню по ссылке bignerdranch.com/tavern-menu-data/ и сохраните в созданной вами папке **data** в файле с именем **tavern-menu-data.txt**.

Теперь можно добавить в **Tavern.kt** код для чтения текста из файла в строку. Чтобы упростить работу с меню, используйте функцию **split**. Она возвращает список **List<String>**, в котором каждый элемент соответствует части **String** между разделителями (в данном случае символами новой строки).

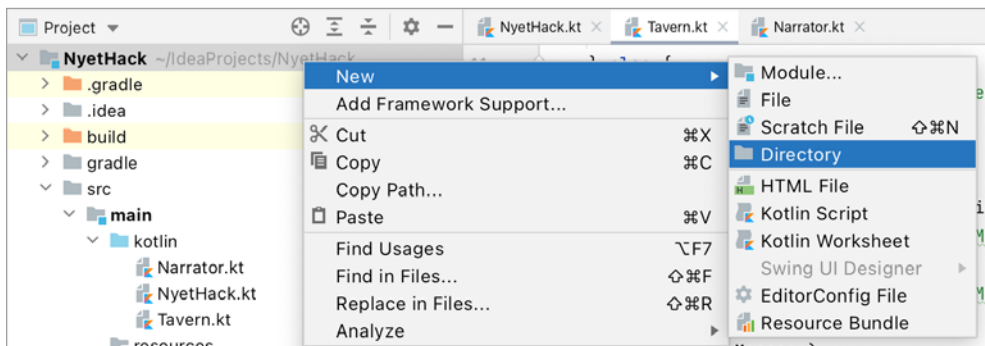


Рис. 9.1. Создание нового каталога

При внесении этих изменений не забудьте добавить в начало файла **Tavern.kt** инструкцию **import java.io.File**. (Также не забудьте, что тип **File** относится к Java API, поэтому его можно использовать только при программировании для Kotlin/JVM. Для других платформ необходимо выбрать API той платформы, для которой компилируется код.)

Листинг 9.19. Чтение данных меню из файла (Tavern.kt)

```
import java.io.File
```

```
private const val TAVERN_MASTER = "Taernyl"
private const val TAVERN_NAME = "$TAVERN_MASTER's Folly"

private val menuData = File("data/tavern-menu-data.txt")
    .readText()
    .split("\n")
...

```

Тип `java.io.File` используется для работы с конкретным файлом по указанному пути.

Функция `readText` в `File` возвращает содержимое файла в виде строки. Затем вызывается функция `split`, которая разбивает содержимое файла по символу перевода строки (представлен экранированной последовательностью `'\n'`) и возвращает его как список.

Теперь вызовите `forEachIndexed` для `menuData`, чтобы вывести все элементы списка вместе с их индексами.

Листинг 9.20. Вывод разнообразного меню (Tavern.kt)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")

    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    ...
    println(othersMessage)

    patrons.forEachIndexed { index, patron ->
        println("Good evening, $patron - you're #${index + 1} in line")
        placeOrder(patron, "Dragon's Breath")
    }

    menuData.forEachIndexed { index, data ->
        println("$index : $data")
    }
}
...

```

Запустите `NyetHack`. Вы увидите содержимое меню, загруженное в список:

```
...
0 : shandy,Dragon's Breath,5.91
1 : elixir,Shirley's Temple,4.12
2 : meal,Goblet of LaCroix,1.22
3 : desert dessert,Pickled Camel Hump,7.33
4 : elixir,Iced Boilermaker,11.22
5 : deserved dessert,Hard Day's Work Ice Cream,3.21
6 : meal,Bite of Lembas Bread,0.59

```

Деструктуризация

Список `menuData` содержит дополнительные метаданные о позициях в меню таверны, включая вид блюда и его стоимость. Эта информация пригодится при оплате заказа (этим мы займемся в главе 10), но пока нам понадобятся только названия.

Создайте второй список для хранения названий блюд. При этом будут использоваться две новые синтаксические конструкции; мы поговорим о них после того, как вы введете код.

Листинг 9.21. Парсинг названий в меню (Tavern.kt)

```
...
private val menuData = File("data/tavern-menu-data.txt")
    .readText()
    .split("\n")

private val menuItems = List(menuData.size) { index ->
    val (type, name, price) = menuData[index].split(",")
    name
}

fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")
    narrate("There are several items for sale:")
    println(menuItems)

    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    ...
menuData.forEachIndexed { index, data ->
    println("$index : $data")
}
}
```

Для создания списка `menuItems` используется *конструктор* `List` вместо функции `listOf`. Конструкторы мы рассмотрим в главе 13, а пока считайте его обычной функцией, которая возвращает `List`. Конструктор `List` получает два аргумента: размер (тип `Int`) и функцию инициализации для заполнения списка.

Взгляните на лямбда-выражение, которое передается конструктору `List`. Сначала оно разбивает одну из строк из файла с меню по всем позициям, в которых находятся запятые. Результатом разбиения становится список, который выглядит примерно так:

```
["shandy", "Dragon's Breath", "5.91"]
```

Для заполнения списка `menuItems` требуется второй элемент списка (в данном случае `"Dragon's Breath"`). Хотя для получения элемента с индексом 1 можно использовать операции `get`, упоминавшиеся ранее, здесь мы применим другой прием — *деструктуризацию*.

Список предоставляет возможность *деструктуризации* до пяти первых элементов. Деструктуризация позволяет объявить несколько переменных и присвоить им значения в одном выражении. В данном случае этот прием мы используем для разделения элементов данных меню:

```
val (type, name, price) = menuData[index].split(',')

```

Это объявление присваивает первые три элемента списка, возвращенного функцией `split`, строковым значениям с именами `type`, `name` и `price`.

Снова запустите `NyetHack`. Теперь названия блюд будут выводиться в виде списка вместо нумерованной последовательности:

```
...
Madrigal enters Taernyl's Folly
There are several items for sale:
[Dragon's Breath, Shirley's Temple, Goblet of LaCroix, Pickled Camel Hump,
  Iced Boilermaker, Hard Day's Work Ice Cream, Bite of Lembas Bread]
...

```

Также можно выполнить избирательную деструктуризацию элементов списка, используя символ `_` для пропуска ненужных элементов. Так как инициализатору списка не нужно знать ни тип элемента, ни цену, пропустим их при помощи `_`.

Листинг 9.22. Пропуск деструктуризованных элементов (Tavern.kt)

```
...
private val menuItems = List(menuData.size) { index ->
    val (type, name, price) = menuData[index].split(",")
    val (_, name, _) = menuData[index].split(",")
    name
}
...

```

Снова запустите `NyetHack` и убедитесь, что вывод не изменился.

После того как данные меню будут загружены, а вы сможете легко обращаться к элементам меню, каждый посетитель получит возможность выбрать любое блюдо из меню при размещении заказа.

Листинг 9.23. Размещение случайных заказов (Tavern.kt)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")
    narrate("There are several items for sale:")
    println(menuItems)

    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    ...
    println(othersMessage)
}

```

```
patrons.forEachIndexed { index, patron ->
    println("Good evening, $patron - you're #${index + 1} in line")
    placeOrder(patron, "Dragon's Breath")
    placeOrder(patron, menuItems.random())
}
}
...
```

Запустите NyetHack. Вы увидите, что каждый посетитель таверны заказывает из меню случайное блюдо.

Множества

Списки, как вы видели, позволяют хранить повторяющиеся элементы (и поддерживают упорядочение элементов, поэтому дубликаты легко выявить по их позициям). Но иногда нужна коллекция, гарантирующая уникальность каждого элемента. Для этого можно использовать множества (*Set*).

Множества имеют много общего со списками. Они используют те же итерационные функции, а также бывают изменяемыми или доступными только для чтения.

Но есть два важных отличия между списками и множествами: элементы множества уникальны, и множества не базируются на индексах, потому что не предусматривают какого-то определенного порядка размещения. (Тем не менее вы все равно можете читать элементы по конкретному индексу, к чему мы еще вернемся.)

Создание множества

По аналогии с созданием списков функцией `listOf` множество `Set` можно создать с помощью функции `setOf`. Попробуйте создать множество в REPL.

Листинг 9.24. Создание множества (REPL)

```
val planets = setOf("Mercury", "Venus", "Earth")
planets
["Mercury", "Venus", "Earth"]
```

Если попытаться дважды добавить в множество одну и ту же планету, из двух дублей сохранится только один.

Листинг 9.25. Попытка создания множества с дубликатом (REPL)

```
val planets = setOf("Mercury", "Venus", "Earth", "Earth")
planets
["Mercury", "Venus", "Earth"]
```

Дубликат "Earth" был исключен из множества.

Как и список, множество позволяет проверить присутствие конкретного элемента с помощью `contains` или `containsAll`. Также можно воспользоваться ключевым словом `in` как сокращением для вызова функции `contains` (для `containsAll` такого сокращения не существует). Попробуйте выполнить функцию `contains` в REPL.

Листинг 9.26. Проверка названий планет (REPL)

```
planets.contains("Pluto")  
false
```

```
"Earth" in planets  
true
```

Множество не индексирует свое содержимое — это означает, что оно не поддерживает встроенный оператор `[]` для обращения к элементам по индексу. Тем не менее можно запросить элемент по определенному индексу с помощью функции, которая использует итерации для решения задачи. Чтобы обратиться к третьей планете в множестве с помощью функции `elementAt`, введите в REPL следующий код.

Листинг 9.27. Поиск третьей планеты (REPL)

```
planets.elementAt(2)  
Earth
```

Такое решение работает, но имейте в виду, что обращение по индексу в множестве работает на порядок медленнее, чем в списке. Это связано с внутренней реализацией `elementAt`. Когда функция `elementAt` вызывается для множества, она последовательно перебирает его элементы, пока не достигнет заданного индекса. Это означает, что в большом множестве обращение к элементу с большим индексом будет происходить медленнее, чем обращение по индексу в списке. По этой причине, если вам нужны обращения по индексу, используйте список, а не множество.

Множество имеет изменяемую версию (как вы скоро увидите), но не имеет функций-мутаторов, использующих индексы, — как функция `add(index, element)` в типе `List`.

С другой стороны, множества обладают уникальным преимуществом перед списками: проверка присутствия элемента выполняется очень быстро независимо от размера множества. Во внутренней реализации `Set` использует внутреннее упорядочение элементов, которое позволяет мгновенно находить элементы.

Вернемся к таверне. Список хорошо подходит для отслеживания очереди, в которой посетители ожидают обслуживания. Но если вам достаточно знать, кто находится в таверне, вероятно, множество будет более подходящим вариантом — при условии, что имена посетителей уникальны.

Добавление элементов в множество

Владелец таверны хочет убрать систему очередности, но при этом ему все равно нужно знать, кто находится в таверне. Для решения этой задачи обновим `NyetHack`, чтобы имена посетителей хранились в `MutableSet` вместо `MutableList`. Так как у многих людей имена совпадают, нам также придется хранить фамилии посетителей.

Для этого в `NyetHack` полные имена посетителей будут генерироваться случайным образом по списку имен и фамилий. Добавьте в `Tavern.kt` список фамилий и используйте `repeat` для генерирования 10 случайных комбинаций имен и фамилий. Для добавления сгенерированных полных имен в множество можно воспользоваться оператором `+=`. (Также хорошо работает функция `add`, но мы будем использовать оператор.)

Удалите два вызова `forEachIndexed`, которые отвечали за приветствие посетителей и создание заказов из меню. Замените их выводом имен посетителей в таверне; чтобы вывод лучше читался, используйте функцию `joinToString`, которая выполняет конкатенацию всех элементов в коллекции и разделяет их запятыми. Наконец, пусть трое случайных посетителей сделают заказы.

Листинг 9.28. Генерирование случайных посетителей (Tavern.kt)

```
private const val TAVERN_MASTER = "Taernyl"
private const val TAVERN_NAME = "$TAVERN_MASTER's Folly"

private val firstNames = setOf("Alex", "Mordoc", "Sophie", "Tariq")
private val lastNames = setOf("Ironfoot", "Fernsworth", "Baggins", "Downstrider")

private val menuData = File("data/tavern-menu-data.txt")
    .readText()
    .split("\n")
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")
    narrate("There are several items for sale:")
    println(menuItems)
    narrate(menuItems.joinToString())

    val patrons = mutableListOf("Eli", "Mordoc", "Sophie")
    val patrons: MutableSet<String> = mutableSetOf()
    repeat(10) {
        patrons += "${firstNames.random()} ${lastNames.random()}"
    }

    val eliMessage = if (patrons.contains("Eli")) {
        "$TAVERN_MASTER says: Eli's in the back playing cards"
    }
```

```

    } else {
        "$TAVERN_MASTER says: Eli isn't here"
    }
    println(eliMessage)

    val othersMessage = if (patrons.containsAll(listOf("Sophie", "Mordoc"))) {
        "$TAVERN_MASTER says: Sophie and Mordoc are seated by the stew kettle"
    } else {
        "$TAVERN_MASTER says: Sophie and Mordoc aren't with each other right now"
    }
    println(othersMessage)

    patrons.forEachIndexed { index, patron ->
        println("Good evening, $patron - you're #${index + 1} in line")
        placeOrder(patron, menuItems.random())
    }

    narrate("$heroName sees several patrons in the tavern:")
    narrate(patrons.joinToString())

    repeat(3) {
        placeOrder(patrons.random(), menuItems.random())
    }
}
...

```

Запустите NyetHack. Вы увидите случайные имена посетителей в выводе. Они необязательно будут совпадать с приведенными ниже, но общая идея такая же:

```

...
Madrigal enters Taernyl's Folly
There are several items for sale:
Dragon's Breath, Shirley's Temple, Goblet of LaCroix, Pickled Camel Hump,
    Iced Boilermaker, Hard Day's Work Ice Cream, Bite of Lembas Bread
Madrigal sees several patrons in the tavern:
Alex Downstrider, Sophie Downstrider, Mordoc Fernsworth, Tariq Downstrider,
    Mordoc Baggins, Mordoc Ironfoot, Alex Fernsworth, Alex Baggins,
    Tariq Fernsworth, Tariq Baggins
Alex Fernsworth speaks with Taernyl to place an order
Taernyl hands Alex Fernsworth a Shirley's Temple
Mordoc Ironfoot speaks with Taernyl to place an order
Taernyl hands Mordoc Ironfoot a Hard Day's Work Ice Cream
Alex Downstrider speaks with Taernyl to place an order
Taernyl hands Alex Downstrider a Hard Day's Work Ice Cream

```

Мы упоминали ранее, что `MutableSet`, как и `MutableList`, поддерживает добавление и удаление элементов, но у него нет мутаторов с доступом по индексу. В табл. 9.2 перечислены некоторые часто применяемые мутаторы для `MutableSet`.

Таблица 9.2. Мутаторы изменяемых множеств

Функция	Описание	Примеры
add	Добавляет элемент в множество	<code>mutableSetOf(1, 2).add(3)</code> <code>[1, 2, 3]</code>
addAll	Добавляет все элементы другой коллекции в множество	<code>mutableSetOf(1, 2).addAll(listOf(1, 5, 6))</code> <code>[1, 2, 5, 6]</code>
+= (оператор сложения с присваиванием)	Добавляет элемент или коллекцию элементов в множество	<code>mutableSetOf(1, 2) += 3</code> <code>[1, 2, 3]</code> <code>mutableSetOf(1, 2) += listOf(1, 3, 5, 5)</code> <code>[1, 2, 3, 5]</code>
-= (оператор вычитания с присваиванием)	Удаляет элемент или коллекцию элементов из множества	<code>mutableSetOf(1, 2, 3) -= 3</code> <code>[1, 2]</code> <code>mutableSetOf(1, 2, 3) -= listOf(2, 3)</code> <code>[1]</code>
remove	Удаляет элемент из множества	<code>mutableSetOf(1, 2, 3).remove(1)</code> <code>[2, 3]</code>
removeAll	Удаляет из множества все элементы, перечисленные в другой коллекции	<code>mutableSetOf(1, 2).removeAll(listOf(1, 3))</code> <code>[2]</code>
clear	Удаляет все элементы из множества	<code>mutableSetOf(1, 2).clear()</code> <code>[]</code>

Запустите `NyetHack` несколько раз, обращая внимание на имена посетителей таверны. Вы увидите, что иногда в таверне оказывается менее 10 посетителей. Это происходит при генерировании повторяющихся имен, так как множества не допускают наличия дубликатов.

Цикл `while`

Если вы хотите быть уверены, что в момент прихода Мадригала в таверне будет ровно 10 посетителей, при заполнении множества придется действовать иначе. Существует другая управляющая конструкция, идеально подходящая для этой задачи: цикл `while`.

Цикл `for` удобно использовать, когда код должен быть выполнен для каждого элемента последовательности. Но он плохо подходит для случаев, когда цикл завершается по достижении некоторого состояния, а не после определенного числа итераций. Для таких ситуаций лучше использовать цикл `while`.

Цикл `while` действует по принципу «пока условие истинно, выполнять код в блоке». Обновите `Tavern.kt`, чтобы генерировать имена посетителей в цикле `while`, пока в множестве не появится ровно 10 элементов.

Листинг 9.29. Генерирование ровно 10 посетителей (`Tavern.kt`)

```
...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")
    narrate("There are several items for sale:")
    narrate(menuItems.joinToString())

    val patrons: MutableSet<String> = mutableSetOf()
    repeat(10) {
        while (patrons.size < 10) {
            patrons += "${firstNames.random()} ${lastNames.random()}"
        }
        ...
    }
}
```

Выполните `NyetHack` еще несколько раз. Теперь вы всегда будете видеть ровно 10 посетителей в таверне.

Цикл `while` требует некоторого основания для хранения состояния цикла. В данном случае мы пользуемся тем обстоятельством, что `patrons` автоматически отслеживает свой размер. В зависимости от вашей цели также можно реализовать собственные счетчики или условия, которыми вы будете управлять вручную.

Объединение циклов `while` с другими управляющими конструкциями (например, условными командами, представленными в главе 3) позволяет создать более сложные состояния. Рассмотрим следующий пример:

```
var isTavernOpen = true
var isClosingTime = false
while (isTavernOpen) {
    if (isClosingTime) {
        isTavernOpen = false
    }
    println("Having a grand old time!")
}
```

Здесь цикл `while` продолжает повторяться до тех пор, пока `isTavernOpen` истинно; состояние представлено логическим значением.

Это очень мощный инструмент, но он может быть опасен. Подумайте, что произойдет, если `isTavernOpen` никогда не примет ложное значение. Или если вы хотите

сгенерировать 20 посетителей, но случайно сгенерировать можно только 16 уникальных имен. В обоих случаях цикл `while` «зациклится», и программа зависнет или продолжит выполняться бесконечно. Будьте осторожны с применением цикла `while`.

Преобразование коллекций

В `NyetHack` вы создали изменяемое множество уникальных имен посетителей, передавая элементы из списка в множество один за другим. Также можно преобразовать список в множество и обратно с помощью функций `toSet` и `toList` (или их изменяемых аналогов: `toMutableSet` и `toMutableList`). В частности, существует распространенный прием — вызов `toSet`, отбрасывающий неуникальные элементы в списке. (Попробуйте поэкспериментировать в REPL.)

Листинг 9.30. Преобразование списка в множество (REPL)

```
listOf("Eli Baggins", "Eli Baggins", "Eli Ironfoot").toSet()  
[Eli Baggins, Eli Ironfoot]
```

Если после преобразования списка в множество для удаления дубликатов вам нужны быстрые обращения по индексу, множество можно преобразовать обратно в список.

Листинг 9.31. Преобразование множества обратно в список (REPL)

```
val patrons = listOf("Eli Baggins", "Eli Baggins", "Eli Ironfoot")  
    .toSet()  
    .toList()  
patrons  
[Eli Baggins, Eli Ironfoot]  
  
patrons[0]  
Eli Baggins
```

Задача удаления дубликатов, а потом возвращения к доступу по индексу настолько часто встречается, что в Kotlin есть для этого специальная функция с именем `distinct` в типе `List`.

Листинг 9.32. Вызов `distinct` (REPL)

```
val patrons = listOf("Eli Baggins", "Eli Baggins", "Eli Ironfoot").distinct()  
patrons  
[Eli Baggins, Eli Ironfoot]  
  
patrons[0]  
Eli Baggins
```

Множества полезны для представления наборов данных с уникальными элементами. В следующей главе мы закончим обзор типов коллекций в языке Kotlin, познакомив вас с ассоциативными массивами.

Для любознательных: типы массивов

Многие языки программирования, включая Kotlin, поддерживают примитивные определения массивов `Array`. Такие массивы намного проще типов коллекций, представленных в этой главе. Они не поддерживают автоматическое изменение размеров, они всегда изменяемы, и они перезаписывают значения в массиве, вместо того чтобы выделять место для них.

Довольно часто, особенно при использовании платформенного кода, вам придется передавать `Array` вместо коллекций.

Допустим, вы пытаетесь вызвать функцию, которая имеет следующую сигнатуру в Kotlin:

```
fun displayPlayerAges(playerAges: IntArray)
```

Функция `displayPlayerAges` ожидает получить параметр `IntArray`. В других языках программирования, таких как C и Java, он часто имеет вид `int[] playerAges`. Для получения `IntArray` (который компилируется в `int[]`) можно вызвать `displayPlayerAges` в следующем виде:

```
val playerAges: IntArray = intArrayOf(34, 27, 14, 52, 101)
displayPlayerAges(playerAges)
```

Обратите внимание на тип `IntArray` и вызов функции `intArrayOf`. Так же как `List`, тип `IntArray` представляет последовательность элементов, а именно целых чисел. `IntArray`, в отличие от `List`, при компиляции преобразуется в примитивный массив. Kotlin использует нативный тип массива на платформе, для которой предназначен ваш код, поэтому вам стоит знать об этой полезной возможности, если вы собираетесь часто пользоваться средствами совместимости.

Преобразовать коллекцию Kotlin в соответствующий элементарный Java-массив можно с помощью встроенных функций. Например, список целых чисел преобразовывают в `IntArray` при помощи функции `toIntArray`, поддерживаемой типом `List`. Это позволит вам преобразовать коллекцию в примитивный массив `int`, что может быть очень полезно при вызове функций Java или нативных функций:

```
val playerAges: List<Int> = listOf(34, 27, 14, 52, 101)
displayPlayerAges(playerAges.toIntArray())
```

В табл. 9.3 перечислены типы массивов и функции, их создающие.

В общем случае придерживайтесь типов коллекций (таких, как `List`), если у вас нет веских причин поступать иначе, например, для обеспечения совместимости с Java-кодом. Коллекции Kotlin — в большинстве случаев хороший вариант, потому что они дают возможность выбора между доступностью только для чтения и изменяемостью и при этом обладают широкими возможностями.

Таблица 9.3. Типы массивов

Тип массива	Создающая функция
<code>IntArray</code>	<code>intArrayOf</code>
<code>DoubleArray</code>	<code>doubleArrayOf</code>
<code>LongArray</code>	<code>longArrayOf</code>
<code>ShortArray</code>	<code>shortArrayOf</code>
<code>ByteArray</code>	<code>byteArrayOf</code>
<code>FloatArray</code>	<code>floatArrayOf</code>
<code>BooleanArray</code>	<code>booleanArrayOf</code>
<code>Array</code> ¹	<code>arrayOf</code>

¹ `Array` компилируется в примитивный массив, способный хранить элементы любого ссылочного типа.

Для любознательных: «только для чтения» vs «неизменяемый»

Во всей книге мы использовали термин «только для чтения» вместо «неизменяемый» за исключением пары случаев, но не объяснили почему. Время пришло. «Неизменяемый» неправильно отражает суть коллекций Kotlin (и некоторых других типов), потому что они все-таки способны изменяться. Рассмотрим несколько примеров списков.

Перед вами объявление двух списков `List`. Переменные, ссылающиеся на списки, доступны только для чтения, потому что объявлены как `val`. Но элементом каждого из этих списков является изменяемый список.

```
val x = listOf(mutableListOf(1, 2, 3))
val y = listOf(mutableListOf(1, 2, 3))

x == y
true
```

Пока ничего особенного.

Переменным `x` и `y` присвоены одинаковые значения, и API типа `List` не предоставляет никаких функций для добавления, удаления или повторного присваивания конкретного элемента. Тем не менее эти списки содержат изменяемые списки, и *их* содержимое *может* изменяться:

```
val x = listOf(mutableListOf(1, 2, 3))
val y = listOf(mutableListOf(1, 2, 3))
```

```
x[0].add(4)
```

```
x == y  
false
```

Структурное сравнение `x` и `y` вернуло `false`, потому что содержимое `x` изменилось. Должен ли неизменяемый список себя так вести? По нашему мнению, не должен.

Вот еще один пример:

```
var myList: List<Int> = listOf(1, 2, 3)  
(myList as MutableList)[2] = 1000  
myList  
[1, 2, 1000]
```

В этом примере `myList` был приведен к типу `MutableList`, то есть компилятору было приказано рассматривать `myList` как изменяемый список, несмотря на то что он был создан с помощью `listOf`. (О приведении типов мы подробнее расскажем в главах 15 и 17.) Приведение типа позволило изменить третье значение в `myList`. И мы снова видим не то поведение, которое можно ожидать от чего-то, названного «неизменяемым».

`List` в Kotlin не является строго неизменяемым — вы сами выбираете, использовать ли его в неизменяемой манере. «Неизменяемость» `List` в Kotlin — это не вполне точное определение, и что бы вы ни делали, не забывайте об этом.

Для любознательных: выражение `break`

Циклы `for` и `while` прекращают выполнение при достижении некоторого состояния. Для цикла `for` это отсутствие дальнейших элементов для перебора, для цикла `while` — результат `false`, полученный при вычислении условия, переданного циклу `while`.

Также для выхода из цикла можно воспользоваться выражением `break`. В следующем примере цикл `while` повторяется, пока значение `isTavernOpen` истинно. Вместо того чтобы изменять значение `isTavernOpen` на `false`, можно выполнить команду `break`, которая немедленно прервет цикл:

```
var isTavernOpen = true  
var isClosingTime = false  
while (isTavernOpen) {  
    if (isClosingTime) {  
        break  
    }  
  
    println("Having a grand old time!")  
}
```

Команда `break` не прерывает выполнение программы или функции. Она всего лишь останавливает цикл, в котором была вызвана, а выполнение программы продолжается.

Команда `break` позволяет легко выйти из цикла, но часто за это приходится расплачиваться удобочитаемостью кода. Особенно заметную путаницу `break` создает во вложенных циклах: какой именно цикл прерывается? (Правильный ответ: самый внутренний цикл, в котором находится `break`. Но читатели вашего кода, не знающие этого, останутся в недоумении.)

Сложные циклы часто проще читаются, когда вся управляющая логика располагается в самом условии цикла. В противном случае вам, возможно, придется прочитать все тело цикла, чтобы понять истинный смысл происходящего. В предыдущем примере тот же цикл можно выразить так:

```
var isTavernOpen = true
var isClosingTime = false
while (isTavernOpen && !isClosingTime) {
    println("Having a grand old time!")
}
```

Команды `break` время от времени встречаются в стандартной библиотеке Kotlin и других библиотеках, которые вы добавляете в проект, но мы рекомендуем как следует подумать, прежде чем использовать их в вашем коде.

Для любознательных: метки `return`

В главе 8 мы показали разные способы использования лямбда-функций. Вы можете передавать ссылки на функции или возвращать их из функций, но чаще всего лямбда-выражения используют при вызове функций из стандартной библиотеки Kotlin (таких, как `forEach`).

Лямбда-выражения возвращают результат неявно, без ключевого слова `return`. Ключевое слово `return` можно включить в лямбда-выражение, но результат вас удивит. Рассмотрим функцию, которая выводит буквы английского алфавита, пропуская гласные:

```
fun printConsonants() {
    ('a'..'z').forEach { letter ->
        if ("aeiou".contains(letter)) {
            return
        }
        print(letter)
    }
}
```

Какой результат будет получен при выполнении `printConsonants`? (Попробуйте сделать это в Kotlin REPL.)

Вместо того чтобы пропустить итерацию, выражение `return` выходит из функции `printConsonants`. И хотя `return` выполняется внутри лямбда-выражения, выход осуществляется из внешней области видимости `printConsonants`. Так как «a» является гласной и первой буквой английского алфавита, `printConsonants` вернет управление до того, как что-либо будет выведено.

Если вы хотите вернуть управление из лямбда-выражения, переданного `forEach`, вам придется более явно указать, в какую точку происходит возврат. Для этого можно воспользоваться метками *return*.

```
fun printConsonants() {  
    ('a'..'z').forEach letters@{ letter ->  
        if ("aeiou".contains(letter)) {  
            return@letters  
        }  
  
        print(letter)  
    }  
}
```

В измененной версии функции `printConsonants` лямбда-выражению, переданному `forEach`, назначается метка `letters@`. Здесь `letters` — имя метки, а символ `@` означает, что это метка связывается со следующим за ней лямбда-выражением.

Назначив метку команде `return`, вы можете вернуться в любую из областей видимости ваших локальных функций.

В данном случае `return@letters` просто переходит к следующей итерации цикла `forEach`, но метки также можно использовать для возвращения значений:

```
return@numbers 17
```

Их допустимо применять и к циклам, проясняя смысл команды `break`. Рассмотрим следующий код, в котором вложенные циклы используются для вывода всех возможных номеров версий некоторого программного продукта:

```
prefixLoop@for(prefix in listOf("alpha", "beta")) {  
    var number = 0  
    numbersLoop@while (number < 10) {  
        val identifier = "$prefix $number"  
        if (identifier == "beta 3") {  
            break@prefixLoop  
        }  
        number++  
    }  
}
```

Будьте внимательны при использовании меток, так как изменение области видимости без контекста в некоторых случаях усложняет понимание. Тем не менее при правильном подходе метки — весьма полезный инструмент для управления программной логикой.

Задание: форматированный вывод меню таверны

Первое впечатление — самое важное, и здесь трудно переоценить роль меню. В этом задании мы сгенерируем презентабельную версию меню. Напечатайте названия блюд с прописной буквы и выровняйте по левому краю. Включите в меню цены, выровняйте их по десятичной точке. Отформатируйте текст в аккуратный блок.

Вывод должен выглядеть приблизительно так:

```
*** Welcome to Taernyl's Folly ***
Dragon's Breath.....5.91
Shirley's Temple.....4.12
Goblet of LaCroix.....1.22
Pickled Camel Hump.....7.33
Iced Boilermaker.....11.22
Hard Day's Work Ice Cream....3.21
Bite of Lembas Bread.....0.59
```

Подсказка: чтобы подсчитать число точек-заполнителей для каждой строки, выберите из списка позиций меню самую длинную строку.

Задание: улучшенное форматирование меню таверны

На основе кода из предыдущего задания сгенерируйте меню, которое также группирует элементы в списке по типу блюд. Вы должны получить следующий вывод:

```
*** Welcome to Taernyl's Folly ***
    ~[shandy]~
Dragon's Breath.....5.91
    ~[elixir]~
Iced Boilermaker.....11.22
Shirley's Temple.....4.12
    ~[meal]~
Goblet of LaCroix.....1.22
Bite of Lembas Bread.....0.59
    ~[desert dessert]~
Pickled Camel Hump.....7.33
    ~[deserved dessert]~
Hard Day's Work Ice Cream....3.21
```

10. Ассоциативные массивы

Третий из наиболее часто применяемых типов коллекций в языке Kotlin — тип `Map`, или *ассоциативный массив* (его также называют *словарем*). Тип `Map` имеет много общего с типами `Set` и `List`: все они группируют наборы элементов, по умолчанию доступны только для чтения, используют параметризованные типы для передачи компилятору информации о типе содержимого и поддерживают итерации.

Отличие от списков и множеств в том, что ассоциативные массивы состоят из пар «ключ — значение», а вместо обращения по целочисленному индексу предоставляют доступ по ключу указанного типа. Ключи уникальны, они однозначно определяют значения в ассоциативном массиве, тогда как значения не обязаны быть уникальными. С этой точки зрения ассоциативные массивы обладают одним из свойств множеств: они гарантируют уникальность ключей подобно элементам множеств.

Создание ассоциативного массива

Подобно спискам и множествам, ассоциативные массивы создаются с помощью функций — `mapOf` или `mutableMapOf`. Мы будем использовать `mapOf` для создания ассоциативного массива, представляющего количество золота у каждого посетителя в кошельке (синтаксис аргумента мы объясним далее). А пока будем контролировать средства в кошельке только у героя игры и хозяина таверны; к остальным посетителям вернемся позже.

Создайте первый ассоциативный массив в `Tavern.kt`. (Синтаксис аргумента объясним ниже.)

Листинг 10.1. Создание ассоциативного массива, доступного только для чтения (`Tavern.kt`)

```
...
fun visitTavern() {
    ...
    val patrons: MutableSet<String> = mutableSetOf()
    val patronGold = mapOf(
        TAVERN_MASTER to 86.00,
        heroName to 4.50
    )
    while (patrons.size < 10) {
```

```

    patrons += "${firstNames.random()} ${lastNames.random()}"
}

println(patronGold)

narrate("$heroName sees several patrons in the tavern:")
...
}
...

```

Ключи ассоциативного массива должны быть одного типа. Значения тоже должны быть одного типа, но типы ключей и значений могут быть разными. Итак, создаем ассоциативный массив со строковыми ключами и дробными значениями. Мы положились на автоматическое определение типов, но при желании явно показать типы ключей и значений можно объявить ассоциативный массив так: `val patronGold: Map<String, Double>`.

Запустите NyetHack, чтобы увидеть получившийся массив.

```

...
Madrigal enters Taernyl's Folly
There are several items for sale:
Dragon's Breath, Shirley's Temple, Goblet of LaCroix, Pickled Camel Hump,
Iced Boilermaker, Hard Day's Work Ice Cream, Bite of Lembas Bread
{Taernyl=86.0, Madrigal=4.5}
Madrigal sees several patrons in the tavern:
...

```

При выводе ассоциативный массив заключается в фигурные скобки, а списки и множества — в квадратные.

Для определения каждого элемента (ключа и значения) в ассоциативном массиве используется синтаксис `to`:

```

val patronGold = mapOf(
    TAVERN_MASTER to 86.00,
    heroName to 4.50
)

```

Может показаться, что `to` — ключевое слово, но на самом деле это особая разновидность функций — так называемая *инфиксная функция*; ее можно вызывать без точки и круглых скобок, в которые заключаются аргументы. Также допустимо использовать вызов `heroName.to(4.50)`, но сокращенная запись предпочтительнее, особенно при создании ассоциативных массивов функцией `mapOf`. Эту разновидность функций более подробно мы рассмотрим в главе 19. Функция `to` преобразует операнды слева и справа в пару (`Pair`) — тип, представляющий группу из двух элементов.

Ассоциативные массивы строятся с использованием пар «ключ — значение». Но существует и другой способ определения элементов ассоциативного массива, как показано ниже. (Опробуйте эту возможность в REPL.)

Листинг 10.2. Определение ассоциативного массива с использованием типа Pair (REPL)

```
val patronGold = mapOf(
    Pair("Taernyl", 86.00),
    Pair("Madrigan", 4.50)
)
println(patronGold)
{Taernyl=86.0, Madrigal=4.5}
```

Тем не менее синтаксис ассоциативного массива с функцией `to` выглядит аккуратнее.

Ранее мы уже упоминали, что ключи в ассоциативном массиве должны быть уникальными. Но что случится, если попробовать повторно добавить элемент с уже существующим ключом? Добавьте пару с ключом "Madrigan" в REPL.

Листинг 10.3. Попытка добавления дубликата ключа (REPL)

```
val patronGold = mapOf(
    "Taernyl" to 86.00,
    "Madrigan" to 4.50,
    "Madrigan" to 20.00
)
println(patronGold)
{Taernyl=86.0, Madrigal=20.0}
```

По аналогии с тем, как множества требуют уникальности всех элементов, ключи ассоциативного массива должны быть уникальными. Если вы попытаетесь добавить другую пару, ключ которой уже существует в ассоциативном массиве, исходная пара будет заменена новой парой «ключ — значение».

Доступ к значениям в ассоциативном массиве

Получить доступ к значению в массиве можно по его ключу. Для массива `patronGold` мы используем строковый ключ, чтобы получить доступ к балансам (количеству золота в кошельках) посетителей таверны.

Листинг 10.4. Доступ к индивидуальным золотым балансам (Tavern.kt)

```
...
fun visitTavern() {
    ...
    while (patrons.size < 10) {
        patrons += "${firstNames.random()} ${lastNames.random()}"
    }

    println(patronGold)
    println(patronGold["Madrigan"])
```

```
println(patronGold["Taernyl"])
println(patronGold["Eli"])

narrate("$heroName sees several patrons in the tavern:")
...
}
...
```

Запустите NyetHack и выведите баланс для трех заданных персонажей:

```
...
{Taernyl=86.0, Madrigal=4.5}
4.5
86.0
null
...
```

Обратите внимание: вывод содержит только значения без ключей. Кроме того, поиск "Eli" вернул null, потому что ключ Eli не был добавлен в ассоциативный массив patronGold.

По аналогии с другими коллекциями, Kotlin предоставляет функции для обращения к значениям, хранящимся в ассоциативном массиве. В табл. 10.1 перечислены некоторые наиболее часто применяемые функции и их действие.

Таблица 10.1. Функции обращения к элементам ассоциативного массива

Функция	Описание	Примеры
<code>[]</code> (оператор обращения по индексу)	Возвращает значение для указанного ключа или null, если ключ не существует	<code>patronGold["Reginald"]</code> <i>null</i>
<code>getValue</code>	Возвращает значение для указанного ключа; выдает исключение, если ключ не существует	<code>patronGold.getValue("Reggie")</code> <i>NoSuchElementException</i>
<code>getOrElse</code>	Возвращает значение для указанного ключа или значение по умолчанию, используя переданное значение	<code>patronGold. getOrElse("Reginald", 0.0)</code> <i>0.0</i>
<code>getOrElse</code>	Возвращает значение для указанного ключа или вычисляет значение по умолчанию с использованием анонимной функции	<code>patronGold.getOrElse("Reggie") { patronName -> if (patronName == "Jane") 4.0 else 0.0 }</code> <i>0.0</i>

Добавление записей в ассоциативный массив

Ваш ассоциативный массив, хранящий сведения о средствах посетителей, представляет кошельки только Madrigal и Taernyl, но не других посетителей, которые были сгенерированы динамически. Настало время исправить это, заменив `patronGold` на `MutableMap`.

В цикле `while`, генерирующем имена посетителей, добавьте в массив запись, чтобы в кошельке каждого посетителя, входящего в таверну, оказалось 6 золотых монет.

Кроме того, удалите ранее созданные записи, потому что с этого момента ключи будут содержать полные имена, включая фамилии.

Листинг 10.5. Заполнение изменяемого ассоциативного массива (Tavern.kt)

```
...
fun visitTavern() {
    ...
    val patrons: MutableSet<String> = mutableSetOf()
    val patronGold = mapOf(
    val patronGold = mutableMapOf(
        TAVERN_MASTER to 86.00,
        heroName to 4.50
    )
    while (patrons.size < 10) {
        patrons += "${firstNames.random()} ${lastNames.random()}"
        val patronName = "${firstNames.random()} ${lastNames.random()}"
        patrons += patronName
        patronGold += patronName to 6.0
    }
    println(patronGold)
    println(patronGold["Madrigal"])
    println(patronGold["Taernyl"])
    println(patronGold["Eli"])

    narrate("$heroName sees several patrons in the tavern:")
    narrate(patrons.joinToString())
    ...
}
```

Оператор `+=` типа `Map` добавляет в ассоциативный массив запись для каждого нового ключа. (Также можно воспользоваться функцией `put`, которая работает точно так же.) Если пара с таким ключом уже существует, оператор `+=` заменяет ее. Мы уже показывали ранее аналогичный пример поведения `Map` при попытке создания ассоциативного массива с дубликатами ключей.

В табл. 10.2 перечислены наиболее популярные функции для корректировки содержимого изменяемого ассоциативного массива.

Таблица 10.2. Мутаторы изменяемого ассоциативного массива

Функция	Описание	Примеры
= (оператор присваивания)	Добавляет или обновляет значение для указанного ключа	<pre>val patronGold = mutableMapOf("Mordoc" to 6.0) patronGold["Mordoc"] = 5.0 {Mordoc=5.0}</pre>
+= (оператор сложения с присваиванием)	Добавляет или обновляет одну или несколько записей в зависимости от операнда справа, который может быть записью или ассоциативным массивом	<pre>val patronGold = mutableMapOf("Mordoc" to 6.0) patronGold += "Eli" to 5.0 {Mordoc=6.0, Eli=5.0} patronGold += mapOf("Eli" to 7.0, "Mordoc" to 1.0, "Sophie" to 4.5) {Mordoc=1.0, Eli=7.0, Sophie=4.5}</pre>
put	Добавляет или обновляет значение для указанного ключа	<pre>val patronGold = mutableMapOf("Mordoc" to 6.0) patronGold.put("Mordoc", 5.0) {Mordoc=5.0}</pre>
putAll	Добавляет все пары «ключ — значение», переданные в аргументе	<pre>val patronGold = mutableMapOf("Mordoc" to 6.0) patronGold.putAll(listOf("Jebediah" to 5.0, "Sahara" to 6.0)) patronGold["Sahara"] 6.0</pre>
getOrPut	Добавляет запись с указанным ключом, если она не существует, и возвращает результат; в противном случае возвращает существующее значение	<pre>val patronGold = mutableMapOf<String, Double>() patronGold.getOrPut("Randy") {5.0} 5.0 patronGold.getOrPut("Randy") {10.0} 5.0</pre>

Функция	Описание	Примеры
remove	Удаляет запись и возвращает значение	<pre>val patronGold = mutableMapOf("Mordoc" to 5.0) val mordocBalance = patronGold.remove("Mordoc") {} print(mordocBalance) 5.0</pre>
-- (оператор вычитания с присваиванием)	Удаляет одну или несколько записей	<pre>val patronGold = mutableMapOf("Mordoc" to 6.0, "Jebediah" to 1.0, "Sophie" to 8.0, "Tariq" to 4.0) patronGold -= listOf("Mordoc", "Sophie") {Jebediah=1.0, Tariq=4.0}</pre>
clear	Удаляет все записи	<pre>mutableMapOf("Mordoc" to 6.0, "Jebediah" to 1.0).clear() {}</pre>

Изменение значений в ассоциативном массиве

В настоящее время функция `performPurchase` не взимает с посетителей плату за заказы, но такая модель ведения бизнеса совершенно нежизнеспособна. Чтобы завершить покупку, необходимо вычесть цену заказа из суммы в кошельке посетителя и прибавить ее к сумме в кошельке хозяина таверны. Ассоциативный массив `patronGold` связывает значение золотого баланса с ключом — именем посетителя. Чтобы изменить сумму в кошельке посетителя, следует записать новое значение после завершения покупки.

Пока что будем использовать в таверне модель «все по одной цене», то есть все блюда в меню стоят одинаково. После того как мы начнем взимать оплату с посетителей, мы изменим модель и будем брать цены из файла `tavern-menu-data.txt`.

Обновите функцию `performPurchase`, чтобы в ней применялся ассоциативный массив `patronGold` для отслеживания денежных потоков; не забудьте об отклонении заказов, на которые у посетителя таверны не хватает денег. Кроме того, выводите

на печать ассоциативный массив до и после покупки, чтобы убедиться в правильности расчетов с посетителями.

Наконец, обслуживание 10 посетителей идет живее, но вывод NyetHack становится слишком объемным. Отрегулируйте количество посетителей так, чтобы с ними было удобнее управляться.

Листинг 10.6. Обновление значений в patronGold (Tavern.kt)

```
fun visitTavern() {
    ...
    while (patrons.size < 10) {
        while (patrons.size < 5) {
            val patronName = "${firstNames.random()} ${lastNames.random()}"
            patrons += patronName
            patronGold += patronName to 6.0
        }
        narrate("$heroName sees several patrons in the tavern:")
        narrate(patrons.joinToString())

        println(patronGold)
        repeat(3) {
            placeOrder(patrons.random(), menuItems.random(), patronGold)
        }
        println(patronGold)
    }

    private fun placeOrder(patronName: String, menuItemName: String) {
    private fun placeOrder(
        patronName: String,
        menuItemName: String,
        patronGold: MutableMap<String, Double>
    ) {
        val itemPrice = 1.0

        narrate("$patronName speaks with $TAVERN_MASTER to place an order")
        if (itemPrice <= patronGold.getOrElse(patronName, 0.0)) {
            narrate("$TAVERN_MASTER hands $patronName a $menuItemName")
            narrate("$patronName pays $TAVERN_MASTER $itemPrice gold")
            patronGold[patronName] = patronGold.getValue(patronName) - itemPrice
            patronGold[TAVERN_MASTER] = patronGold.getValue(TAVERN_MASTER) + itemPrice
        } else {
            narrate("$TAVERN_MASTER says, \"You need more coin for a $menuItemName\")
        }
    }
}

Запустите NyetHack. Вы увидите случайные заказы и содержимое кошельков посетителей до и после заказов:
...
{Taernyl=86.0, Madrigal=4.5, Alex Fernsworth=6.0, Tariq Downstrider=6.0,...}
Alex Fernsworth speaks with Taernyl to place an order
Taernyl hands Alex Fernsworth a Shirley's Temple
```

```
Alex Fernsworth pays Taernyl 1.0 gold
Tariq Downstrider speaks with Taernyl to place an order
Taernyl hands Tariq Downstrider a Goblet of LaCroix
Tariq Downstrider pays Taernyl 1.0 gold
...
{Taernyl=89.0, Madrigal=4.5, Alex Fernsworth=5.0, Tariq Downstrider=5.0,...}
```

Преобразования между списками и ассоциативными массивами

Следующая задача в NyetHack — списание реальной стоимости заказов со счетов посетителей. Для этого можно создать еще один ассоциативный массив, ключи в котором — названия блюд в меню, а значения — цены. Ранее проблема парсинга названий блюд из меню была решена созданием нового списка `menuItems` с использованием конструктора `List`. У ассоциативных массивов `Map` эквивалентного конструктора нет, но можно создать ассоциативный массив другими способами.

По аналогии с тем, как список преобразуется в множество и наоборот, списки также можно преобразовать в ассоциативные массивы функцией `toMap`. Но не все так просто. Функция `toMap` доступна только для списков, содержащих `Pair`. Таким образом, `toMap` можно вызвать для `List<Pair<String, Double>>`, но не для `List<String>`, например. (Попытка вызвать функцию `toMap` для списков, содержащих что-либо другое, но не `Pair`, аналогична вызову функции, которая не была объявлена.)

Чтобы создать ассоциативный массив для цен в меню, мы сначала создадим список с парами «ключ — значение», представляющими названия и цены блюд. Мы воспользуемся конструктором `List`, представленным в главе 9, для динамического заполнения этой коллекции данными из файла меню. Так как список содержит пары «ключ — значение», его можно преобразовать в ассоциативный массив вызовом `toMap`.

Листинг 10.7. Преобразование списка в ассоциативный массив (Tavern.kt)

```
...
private val menuItems = List(menuData.size) { index ->
    val (_, name, _) = menuData[index].split(",")
    name
}

private val menuItemPrices: Map<String, Double> = List(menuData.size) { index ->
    val (_, name, price) = menuData[index].split(",")
    name to price.toDouble()
}.toMap()
...
```

Этот код очень похож на код инициализации списка `menuItems`. Он перебирает элементы меню и использует те же средства деструктуризации, которые ранее использовались при чтении значений из меню. В отличие от инициализации `menuItems`, мы создаем пары в списке и добавляем вызов `toMap` после построения списка. Это позволяет динамически читать данные из файла данных меню. В противном случае для использования `mapOf` понадобился бы код следующего вида:

```
mapOf(
    "Dragon's Breath" to 5.91,
    "Shirley's Temple" to 4.12,
    "Goblet of LaCroix" to 1.22,
    "Pickled Camel Hump" to 7.33,
    "Iced Boilermaker" to 11.22
    "Hard Day's Work Ice Cream" to 3.21
    "Bite of Lembas Bread" to 0.59
)
```

С таким ассоциативным массивом можно обновить `placeOrder` для применения правильных цен.

Листинг 10.8. Взимание полной стоимости блюд в меню (Tavern.kt)

```
...
private fun placeOrder(
    patronName: String,
    menuItemName: String,
    patronGold: MutableMap<String, Double>
) {
    val itemPrice = 1.0
    val itemPrice = menuItemPrices.getValue(menuItemName)

    narrate("$patronName speaks with $TAVERN_MASTER to place an order")
    ...
}
```

Запустите `NyetHack`. Вывод должен выглядеть примерно так:

```
...
{Taernyl=86.0, Madrigal=4.5, Mordoc Baggins=6.0, Sophie Ironfoot=6.0,...}
Mordoc Baggins speaks with Taernyl to place an order
Taernyl serves Mordoc Baggins a Goblet of LaCroix
Mordoc Baggins pays Taernyl 1.22 gold
Sophie Ironfoot speaks with Taernyl to place an order
Taernyl hands Sophie Ironfoot a Shirley's Temple
Sophie Ironfoot pays Taernyl 4.12 gold
...
{Taernyl=97.25, Madrigal=4.5, Mordoc Baggins=4.78, Sophie Ironfoot=1.88,...}
```

В меню есть еще один атрибут данных, обозначающий категорию: еда или напиток. Эту информацию можно использовать для генерирования более правильных

формулировок в выводе, предназначенном для пользователя. Вместо обобщенного «hands» (приносит) иногда будут использоваться более точные глаголы «serves» (подает) или «pours» (наливает). Создайте другой ассоциативный массив для категорий меню, а затем воспользуйтесь выражением `when` в `placeOrder` для настройки вывода.

Листинг 10.9. Парсинг категорий позиций в меню (Tavern.kt)

```
...
private val menuItemPrices: Map<String, Double> = List(menuData.size) { index ->
    val (_, name, price) = menuData[index].split(",")
    name to price.toDouble()
}.toMap()

private val menuItemTypes: Map<String, String> = List(menuData.size) { index ->
    val (type, name, _) = menuData[index].split(",")
    name to type
}.toMap()
...
private fun placeOrder(
    patronName: String,
    menuItemName: String,
    patronGold: MutableMap<String, Double>
) {
    val itemPrice = menuItemPrices.getValue(menuItemName)
    narrate("$patronName speaks with $TAVERN_MASTER to place an order")
    if (itemPrice <= patronGold.getOrElse(patronName, 0.0)) {
        narrate("$TAVERN_MASTER hands $patronName a $menuItemName")
        val action = when (menuItemTypes[menuItemName]) {
            "shandy", "elixir" -> "pours"
            "meal" -> "serves"
            else -> "hands"
        }
        narrate("$TAVERN_MASTER $action $patronName a $menuItemName")
        narrate("$patronName pays $TAVERN_MASTER $itemPrice gold")
        patronGold[patronName] = patronGold.getValue(patronName) - itemPrice
        patronGold[TAVERN_MASTER] = patronGold.getValue(TAVERN_MASTER) + itemPrice
    } else {
        narrate("$TAVERN_MASTER says, \"You need more coin for a $menuItemName\")
    }
}
```

Запустите `NyetHack` еще несколько раз и убедитесь, что в выводе используется правильная терминология для описания действий хозяина таверны.

```
...
Tariq Fernsworth speaks with Taernyl to place an order
Taernyl serves Tariq Fernsworth a Goblet of LaCroix
Tariq Fernsworth pays Taernyl 1.22 gold
Sophie Fernsworth speaks with Taernyl to place an order
Taernyl pours Sophie Fernsworth a Dragon's Breath
```

```
Sophie Fernsworth pays Taernyl 5.91 gold
Tariq Baggins speaks with Taernyl to place an order
Taernyl says, "You need more coin for a Iced Boilermaker"
{Taernyl=87.22, Madrigal=4.5, Tariq Fernsworth=4.78, Tariq Baggins=6.0,
Sophie Baggins=6.0, Alex Fernsworth=6.0, Sophie Fernsworth=0.0899999999999986,
Sophie Ironfoot=6.0, Alex Baggins=6.0, Alex Downstrider=6.0,
Mordoc Fernsworth=6.0, Tariq Ironfoot=6.0}
```

Перебор элементов ассоциативного массива

Вы обновили балансы посетителей таверны, и теперь остается только одна задача — отформатировать результаты более презентабельно. Сейчас балансы выводятся, но результаты перегружены техническими подробностями. Пожалуй, вам захочется получить вывод без квадратных скобок, знаков равенства и с округлением, чтобы в выводе не было значений вида `0.0899999999999986`, что обусловлено потерей точности представления с плавающей точкой. Форматирование можно улучшить перебором ассоциативного массива с использованием `forEach`.

Добавьте в конец файла `Tavern.kt` новую функцию с именем `displayPatronBalances`, которая перебирает ассоциативный массив `patronGold` и выводит баланс (отформатированный до двух знаков в дробной части, как в главе 5) для каждого посетителя. Функция должна вызываться в конце функции `visitTavern`.

Листинг 10.10. Вывод информации о средствах в кошельках посетителей (Tavern.kt)

```
...
fun visitTavern() {
    ...
    narrate("$heroName sees several patrons in the tavern:")
    narrate(patrons.toString())

    println(playerGold)
    repeat(3) {
        placeOrder(patrons.random(), menuItems.random(), patronGold)
    }
    println(playerGold)
    displayPatronBalances(patronGold)
}
...
private fun displayPatronBalances(patronGold: Map<String, Double>) {
    narrate("$heroName intuitively knows how much money each patron has")
    patronGold.forEach { (patron, balance) ->
        narrate("$patron has ${"%0.2f".format(balance)} gold")
    }
}
```

Этот вызов `forEach` очень похож на те, которые использовались со списками и множествами в главе 9, но обратите внимание на два нюанса в только что до-

бавленном коде. Во-первых, аргумент лямбда-выражения использует деструктуризацию. Вызов `forEach` получает лямбда-выражение с типом `(Pair<String, Double>) -> Unit`. Обычно для обращения к ключу и значению элемента ассоциативного массива используются записи `it.first` и `it.second`, но здесь деструктуризация в списке параметров лямбда-выражения применена для получения более понятного кода.

Во-вторых, параметр функции `displayPatronBalances` объявлен с типом `Map` вместо `MutableMap`. Почему? Потому что остаток средств в кошельке посетителей не нужно изменять при выводе.

Когда ваша функция получает коллекцию, но не собирается вносить в нее изменения, рекомендуется отдавать предпочтение типам коллекций, доступным только для чтения. Это предотвращает случайное изменение ассоциативного массива, когда вы собираетесь только читать его. Кроме того, такая практика позволяет другим частям программы вызывать функцию независимо от того, используют они `Map` или `MutableMap`.

Но тогда возникает вопрос: почему Kotlin позволяет передать `MutableMap` как аргумент с типом `Map`? В конце концов, `MutableMap` и `Map` — разные типы.

Kotlin позволяет выполнить *приведение типа* `MutableMap` к `Map`. С приведением типа ваша программа может интерпретировать значения так, словно их тип отличается от того, с которым они были определены или объявлены. `MutableMap` базируется на функциональности `Map`, что позволяет интерпретировать ассоциативный массив как более общий тип (`Map`) по сравнению с фактическим типом (`MutableMap`). В этом конкретном сценарии приведение типа выполняется неявно при вызове `displayPatronBalances`.

Следует помнить, что приведение типа — всего лишь альтернативная интерпретация существующего значения, поэтому изменения исходного ассоциативного массива (`MutableMap`) в остальных местах кода будут распространяться на все, что хранит ссылку на этот ассоциативный массив, независимо от приведения типа. Чтобы предотвратить такие обновления, создайте копию, доступную только для чтения, вызовом `toMap` для `MutableMap` вместо приведения типа.

Тот же прием работает с `MutableList` и `MutableSet`, что позволяет интерпретировать любую коллекцию как ее разновидность, доступную только для чтения. Однако переход в обратном направлении не работает, потому что нельзя утверждать, что каждый экземпляр `Map` также является экземпляром `MutableMap`. Если вы хотите безопасно преобразовать экземпляр `Map`, доступный только для чтения, в `MutableMap`, используйте функцию `toMutableMap`. Помните, что при этом вы получаете копию исходного ассоциативного массива, и вставки и удаления не влияют на исходную копию, доступную только для чтения. (Если вы хотите выполнить небезопасную операцию приведения типа, то ознакомьтесь со средствами ручного приведения типа, которые описаны в главах 15 и 17.)

Запустите NyetHack и наблюдайте за тем, как посетители таверны общаются с хозяином, заказывают блюда из меню и оплачивают свои заказы:

```
A hero enters the town of Kronstadt. What is their name?
Madrigal
Madrigal, The Renowned Hero, heads to the town square
Madrigal enters Taernyl's Folly
There are several items for sale:
Dragon's Breath, Shirley's Temple, Goblet of LaCroix, Pickled Camel Hump,
    Iced Boilermaker, Hard Day's Work Ice Cream, Bite of Lembas Bread
Madrigal sees several patrons in the tavern:
Tariq Ironfoot, Alex Baggins, Alex Fernsworth, Sophie Ironfoot, Tariq
    Downstrider
Alex Fernsworth speaks with Taernyl to place an order
Taernyl pours Alex Fernsworth a Shirley's Temple
Alex Fernsworth pays Taernyl 4.12 gold
Alex Fernsworth speaks with Taernyl to place an order
Taernyl says, "You need more coin for a Shirley's Temple"
Tariq Ironfoot speaks with Taernyl to place an order
Taernyl pours Tariq Ironfoot a Dragon's Breath
Tariq Ironfoot pays Taernyl 5.91 gold
Madrigal intuitively knows how much money each patron has
Taernyl has 96.03 gold
Madrigal has 4.50 gold
Tariq Ironfoot has 0.09 gold
Alex Baggins has 6.00 gold
Alex Fernsworth has 1.88 gold
Sophie Ironfoot has 6.00 gold
Tariq Downstrider has 6.00 gold
```

В двух последних главах вы научились работать с типами коллекций Kotlin List, Set и Map.

Возможности разных коллекций мы сравнили в табл. 10.3.

Таблица 10.3. Сводка коллекций Kotlin

Тип коллекции	Упорядочение	Уникальность	Хранятся	Поддержка деструктуризации
List	Да	Нет	Элементы	Да
Set	Нет	Да	Элементы	Нет
Map	Нет	Ключи	Пары «ключ — значение»	Нет

Так как коллекции по умолчанию доступны только для чтения, для изменения их содержимого необходимо явно создать изменяемую коллекцию (или преобразовать коллекцию, доступную только для чтения, в изменяемую разновидность) — тем самым предотвращается случайное добавление или удаление элементов.

В следующей главе вы больше узнаете о средствах функционального программирования, которые позволяют еще эффективнее работать с типами коллекций Kotlin.

Задание: составные заказы

В настоящее время посетителям таверны доступен заказ только одной позиции меню. Если кто-то захочет заказать еду и напиток, ему придется разместить два разных заказа. Доработайте систему заказов таверны, чтобы каждый гость мог заказывать сразу несколько блюд и напитков.

Посетители включают в заказ от 1 до 3 позиций, при этом как количество, так и блюдо выбирается случайным образом. Если у гостя не хватает денег на весь заказ, то он должен быть отклонен, и гость ничего получит, пока не сделает новый заказ. Возможно, посетителям стоит дать больше денег, чтобы они могли позволить себе более дорогие заказы.

(Обязательно создайте копию NyetHack перед выполнением задания. В следующей главе мы покажем разнообразные варианты использования коллекций, несовместимые с изменениями, которые вы внесете при выполнении этого задания.)

11. Основы функционального программирования

В каждом языке программирования для разработчика есть несколько парадигм. Две самые популярные и известные парадигмы, реализуемые в Kotlin, — объектно-ориентированное и функциональное программирование. Вы больше узнаете об объектном программировании в части IV нашей книги, а эту главу мы посвятили функциональному программированию.

Kotlin поддерживает несколько стилей программирования, и для оптимального решения задач стили можно смешивать. Некоторые языки (например, Haskell) поддерживают *только* функциональное программирование. Такие функциональные языки обычно используются в академических кругах, а не в коммерческих продуктах, но их концепции и многие приемы применяются в других языках, таких как JavaScript и Swift.

В главе 8 вы узнали о функциях, которые получают другую функцию в параметре, о функциях, которые возвращают функцию в результате своего выполнения, и типах функций, которые позволяют определять функции как значения. Функции, которые получают функцию как параметр или возвращают ее, называются *функциями высшего порядка*. Функциональное программирование основано на данных, возвращаемых небольшим количеством функций высшего порядка, разработанных главным образом для работы с коллекциями. Функции высшего порядка проектируются так, чтобы простые функции можно было объединять для построения сложного поведения.

В этой главе мы покажем некоторые средства функционального программирования, поддерживаемые Kotlin, а также объясним идеи, заложенные в парадигму функционального программирования. Хотя этот материал мы показываем на примере Kotlin, многие идеи — и даже некоторые имена функций — также встречаются в других языках программирования.

Мы будем использовать функциональное программирование для обновления `Tavern.kt`, а конкретнее — для работы с коллекциями, что сделает кодовую базу Tavern более мощной и компактной. Мы изучим три концепции функционального программирования: *преобразование*, *фильтрацию* и *комбинирование* данных.

Преобразование данных

Первая категория функций в функциональном программировании — преобразователи. *Функция-преобразователь* перебирает элементы коллекции и изменяет каждый из них с помощью *функции преобразования*, заданной в аргументе. Затем функция-преобразователь возвращает измененную коллекцию.

Наиболее часто используются два преобразователя — `map` и `flatMap`.

`map`

Функция-преобразователь `map` перебирает элементы коллекции, для которой вызвана, и к каждому применяет функцию преобразования. Результатом является коллекция преобразованных элементов. (Напомним, что речь идет о функции `map`; не путайте ее с функцией `mapOf` или типом `Map`.)

Функции `map` и `flatMap` обычно используются для преобразования заданного набора данных в другое представление тех же значений. Например, следующий код извлекает пункты меню из файла `tavern-menu-data.txt`:

```
private val menuData = File("data/tavern-menu-data.txt")
    .readText()
    .split("\n")
private val menuItems = List(menuData.size) { index ->
    val (_, name, _) = menuData[index].split(",")
    name
}
```

Этот код работает, но вычисление `menuItems` выглядит довольно запутанно. Мы создаем новый список с таким же количеством элементов, как у `menuData`, после чего заполняем новый список посредством перебора `menuData`. Для решения подобных задач чаще используется `map`.

Обновите свое объявление `menuItems`, чтобы в нем использовалась функция `map`, как показано в листинге 11.1. Мы объясним, как работает `map`, после реализации этих изменений.

Листинг 11.1. Использование функции `map` (Tavern.kt)

```
...
private val menuData = File("data/tavern-menu-data.txt")
    .readText()
    .split("\n")

private val menuItems = List(menuData.size) { index ->
    val (_, name, _) = menuData[index].split(",")
    name
}
```

```
private val menuItems: List<String> = menuData.map { menuEntry: String ->
    val (_, name, _) = menuEntry.split(",")
    name
}
...
```

Запустите NyetHack и убедитесь в том, что вывод не изменился:

```
...
There are several items for sale:
Dragon's Breath, Shirley's Temple, Goblet of LaCroix, Pickled Camel Hump,
    Iced Boilermaker, Hard Day's Work Ice Cream, Bite of Lembas Bread
...
```

Как и прежде, `menuItems` присваивается список с названиями всех блюд в таверне. Но сейчас при запуске программы `map` создает список для элементов, возвращенных функцией преобразования. Затем функция преобразования применяется к каждому элементу (в данном случае читает второй элемент из каждого значения в `menuData`) и присоединяет каждое преобразованное значение к списку с сохранением порядка исходной коллекции. Наконец, `map` возвращает новый список преобразованных значений.

Взгляните еще раз на использованные типы. `menuData` начинается с `List<String>`, а лямбда-выражение, передаваемое `map`, возвращает `String`. Здесь преобразование дает значения того же типа, что, безусловно, допустимо и очень часто встречается при использовании функции преобразования.

Однако функция `map` способна выполнять преобразования к произвольному типу. Например, у вас может быть список чисел, закодированных в строковом виде. Если вы хотите преобразовать все строки в числа с плавающей точкой, можно воспользоваться функцией `map` в сочетании с функцией `toDouble()`:

```
val numbers: List<String> = listOf("1.0", "2.0", "3.0")
["1.0", "2.0", "3.0"]

val numbersAsDoubles: List<Double> = numbers.map { it.toDouble() }
[1.0, 2.0, 3.0]
```

При первом вызове `map` для преобразования предоставляется именованный параметр, включая информацию типа: `menuEntry: String`. Это сделано для наглядности; ни имя, ни объявление типа обязательным не является. Также можно воспользоваться идентификатором `it` или опустить информацию о типе, так как Kotlin способен автоматически его определить.

Чтобы увидеть, как такой вызов выглядит в реальном приложении, удалите явное объявление типа, но оставьте имя, так как оно содержит информативный контекст.

Листинг 11.2. Автоматическое определение типа для map (Tavern.kt)

```
...
private val menuItems: List<String> = menuData.map { menuEntry: String ->
    val (_, name, _) = menuEntry.split(",")
    name
}
...
```

Подобная комбинация выразительности и компактности очень полезна в приложениях любых размеров, и map применяется во многих кодовых базах.

Обновите определения menuItemPrices и menuItemTypes, чтобы в них также использовалась функция map.

Листинг 11.3. Использование map для создания Map (Tavern.kt)

```
...
private val menuItems = menuData.map { menuEntry ->
    val (_, name, _) = menuEntry.split(",")
    name
}

private val menuItemPrices: Map<String, Double> = List(menuData.size) { index ->
    val (_, name, price) = menuData[index].split(",")
private val menuItemPrices = menuData.map { menuEntry ->
    val (_, name, price) = menuEntry.split(",")
    name to price.toDouble()
}.toMap()

private val menuItemTypes: Map<String, String> = List(menuData.size) { index ->
    val (type, name, _) = menuData[index].split(",")
private val menuItemTypes = menuData.map { menuEntry ->
    val (type, name, _) = menuEntry.split(",")
    name to type
}.toMap()
...
```

Несмотря на свое название, функция map возвращает список List с парами Pairs для menuItemPrices и menuItemTypes. Вот почему вам все равно придется использовать вызовы toMap.

Запустите NyetHack и убедитесь, что в выводе ничего не изменилось.

associate

При изменении кода вы, возможно, заметили, что среда IntelliJ добавила волнистые линии под вызовами map. Тем самым IntelliJ деликатно просит вашего внимания.

Наведите указатель мыши на любой вызов map и посмотрите, что вам предлагает IntelliJ (рис. 11.1).

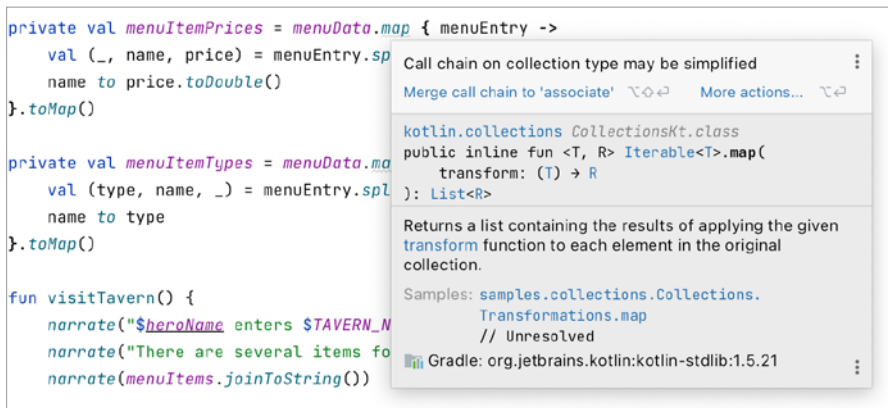


Рис. 11.1. Советы по функциональному программированию

Здесь IntelliJ предлагает использовать другую функцию — `associate`. Ее вызов эквивалентен вызову `вашаКоллекция.map { ключ to значение }.toMap()`. Опробуйте предлагаемую возможность: либо щелкните на `Merge call chain to 'associate'` во всплывающем окне, либо внесите изменения вручную.

Листинг 11.4. Использование функции `associate` (Tavern.kt)

```
...
private val menuItemPrices = menuData.map { menuEntry ->
    val (_, name, price) = menuEntry.split(",")
    name to price.toDouble()
}.toMap()

private val menuItemTypes = menuData.map { menuEntry ->
    val (type, name, _) = menuEntry.split(",")
    name to type
}.toMap()
...
```

У Kotlin в запасе много операций функционального программирования — слишком много, чтобы рассказывать о них в этой книге. Здесь мы ограничимся наиболее важными. Объединяя их друг с другом, вы сможете реализовать более сложные операции.

Иногда можно написать цепочку операций, которая, оказывается, уже доступна в стандартной библиотеке, как в нашем примере. IntelliJ обнаруживает такие варианты и предлагает использовать встроенную версию. Это отличный способ изучения новых операций функционального программирования при работе с коллекциями. Не пренебрегайте такими рекомендациями от IntelliJ!

Деструктуризация средствами функционального программирования

Списки `menuItems`, `menuItemPrices` и `menuItemTypes` в значительной степени перекрываются — все они разбивают строки из файла меню `menuData` и извлекают конкретные компоненты. Так как строки `menuData` всегда разбиваются, вызов `split(",")` можно переместить прямо в `menuData` при помощи `map`. Это также позволит вам использовать синтаксис деструктуризации с лямбда-выражениями, о которых мы говорили в предыдущей главе.

Внесите следующее изменение, которое сделает лямбда-выражения еще более компактными и выразительными.

Листинг 11.5. Деструктуризация с `map` (Tavern.kt)

```
...
private val menuData = File("data/tavern-menu-data.txt")
    .readText()
    .split("\n")
    .map { it.split(",") }

private val menuItemNames = menuData.map { menuEntry ->
    val (_, name, _) = menuEntry.split(",")
    name
}
private val menuItemNames = menuData.map { (_, name, _) -> name }

private val menuItemPrices = menuData.associate { menuEntry ->
    val (_, name, price) = menuEntry.split(",")
    name to price.toDouble()
}
private val menuItemPrices = menuData.associate { (_, name, price) ->
    name to price.toDouble()
}

private val menuItemTypes = menuData.associate { menuEntry ->
    val (type, name, _) = menuEntry.split(",")
    name to type
}
private val menuItemTypes = menuData.associate { (type, name, _) ->
    name to type
}
...
```

Новый вызов `split` включается внутрь вызова `map`, потому что мы хотим разбить каждую из строк, полученных в результате предыдущего разбиения, которое вернуло список. Если попытаться применить `split(",")` непосредственно к результату `split("\n")`, вы получите сообщение об ошибке компиляции.

В Kotlin подобные функциональные операции очень часто встречаются в двух-трех или даже в одной строке кода. Сравните свой обновленный код с предыдущей версией. Хотя для написания нового кода необходимо знать функции `map` и `associate`, он яснее выражает намерения разработчика. По сравнению с другими подходами

API функционального программирования в Kotlin часто способствует написанию более прозрачного и намного более компактного кода.

Снова запустите NyetHack и убедитесь, что программа выполняется без ошибок.

flatMap

Следующая функция-преобразователь называется `flatMap`. Функция `flatMap` работает практически идентично `map`, не считая того, что она сглаживает преобразованные значения: получает коллекцию коллекций (скажем, список списков строк) и возвращает коллекцию вложенного типа (список строк в данном случае).

В настоящий момент функция `flatMap` не используется. Однако хозяин таверны хочет пометить один из пунктов меню как блюдо дня, и оно должно выбираться из любимых блюд посетителей.

У большинства посетителей всего одно любимое блюдо, но у кого-то их может быть несколько. Например, посетитель `Alex Ironfoot` равнодушен к сладкому и всегда выбирает в меню все десерты. Создайте новую функцию для определения любимых блюд каждого посетителя.

Листинг 11.6. Определение любимых блюд посетителей (Tavern.kt)

```
import java.io.File
import kotlin.random.Random
import kotlin.random.nextInt
...
fun visitTavern() {
    ...
}

private fun getFavoriteMenuItems(patron: String): List<String> {
    return when (patron) {
        "Alex Ironfoot" -> menuItems.filter { menuItem ->
            menuItemTypes[menuItem]?.contains("dessert") == true
        }
        else -> menuItems.shuffled().take(Random.nextInt(1..2))
    }
}

private fun placeOrder(
    patronName: String,
    menuItemName: String,
    patronGold: MutableMap<String, Double>
) {
    ...
}
...
```

Здесь мы видим три функции типа `List`: `filter`, `shuffled` и `take`. Первое условие, предназначенное для посетителя `Alex Ironfoot`, использует функцию `filter` для поиска всех десертов в меню. Эту функцию мы рассмотрим позже в разделе «Фильтрация данных».

Функция `shuffled` возвращает копию исходного списка со случайной перестановкой элементов. Функция `take` возвращает список, содержащий количество элементов, меньшее или равное заданному, — в данном случае один или два. В итоге для всех посетителей, кроме `Alex Ironfoot`, случайным образом выбирается одно или два блюда как любимые.

При выборе блюда дня хозяин должен знать, кто сейчас находится в таверне, чтобы учесть любимые блюда всех посетителей. Чтобы получить эти данные, программа связывает посетителей с их любимыми позициями в меню.

Листинг 11.7. Использование `map` для получения любимых блюд (`Tavern.kt`)

```
...
fun visitTavern() {
    ...
    narrate("$heroName sees several patrons in the tavern:")
    narrate(patrons.joinToString())

    val favoriteItems = patrons.map { getFavoriteMenuItems(it) }
    println("Favorite items: $favoriteItems")
    ...
}
...
```

Как вы думаете, что выведет этот код? Запустите `NyetHack` и проверьте свое предположение.

```
...
Favorite items: [[Pickled Camel Hump], [Dragon's Breath, Goblet of LaCroix],
                [Iced Boilermaker, Shirley's Temple], [Pickled Camel Hump]]
...
```

Вы ожидали именно этого?

Присмотритесь к квадратным скобкам в этом выводе: элементы вывода сгруппированы в списки. Теперь выделите переменную `favoriteItems` и нажмите `Control-Shift-P`, чтобы отобразить ее тип. Среда `IntelliJ` выводит подсказку с информацией о том, что переменная относится к типу `List<List<String>>`.

Чтобы понять, почему `favoriteItems` — это список списков, вспомните, как работает функция `map`. Возвращаемые значения лямбда-выражения, переданного `map`, используются непосредственно в полученном списке. В данном случае функция-преобразователь возвращает список — и вы получаете список списков.

Иногда подобное вложение списков удобно (`menuData` также относится к типу `List<List<String>>`, что нормально), но в других случаях дополнительное вложе-

ние только мешает работе с коллекциями и усложняет ее. Чтобы избавиться от вложения, необходимо выполнить сглаживание коллекции.

Известны два подхода к сглаживанию вложенных коллекций. Первый — применение функции `flatten`, которая удаляет вложение. Но если вы уже используете преобразование `map` (как в нашем случае), его можно объединить со сглаживанием при помощи `flatMap`. Попробуйте эту возможность и замените вызов `map` при определении любимых блюд посетителей.

Листинг 11.8. Использование `flatMap` для получения любимых блюд (Tavern.kt)

```
...
fun visitTavern() {
    ...
    val favoriteItems = patrons.map { getFavoriteMenuItems(it) }
    val favoriteItems = patrons.flatMap { getFavoriteMenuItems(it) }
    println("Favorite items: $favoriteItems")
    ...
}
...
```

Снова запустите `NyetHack`. На этот раз названия любимых блюд выводятся без вложенных списков:

```
...
Favorite items: [Pickled Camel Hump, Iced Boilermaker, Goblet of LaCroix,
                 Iced Boilermaker, Iced Boilermaker, Pickled Camel Hump]
...
```

Обратите внимание, как легко изменить операции, определенные с помощью функционального программирования. Здесь для исключения целого уровня вложенности из вывода достаточно заменить `map` на `flatMap`. Без этих средств функционального программирования вам пришлось бы переписывать алгоритмы (или присоединять новые алгоритмы к текущим), чтобы внести в вывод изменения, которые можно реализовать несколькими нажатиями клавиш в функциональном мире.

Теперь, когда у хозяина таверны есть список любимых блюд посетителей (а не список списков любимых блюд!), он готов выбрать блюдо дня. Для новой маркетинговой кампании мы используем функцию `random`.

Листинг 11.9. Выбор блюда дня (Tavern.kt)

```
...
fun visitTavern() {
    ...
    val favoriteItems = patrons.flatMap { getFavoriteMenuItems(it) }
    val itemOfDay = patrons.flatMap { getFavoriteMenuItems(it) }.random()
    println("Favorite items: $favoriteItems")
    narrate("The item of the day is the $itemOfDay")
    ...
}
```

```
}  
...
```

Запустите `NyetHack`. Вывод должен выглядеть так:

```
...  
The item of the day is the Goblet of LaCroix  
...
```

map vs flatMap

Функции `map` и `flatMap` очень похожи. Наверное, вас интересует, в какой ситуации следует использовать каждую функцию?

Выбирая между `map` и `flatMap`, спросите себя: «Должна ли моя функция-преобразователь возвращать коллекцию?» Если нет, используйте `map`. Если функция-преобразователь не возвращает `List` или другую коллекцию, невозможно применить `flatMap`, потому что она требует, чтобы ваше лямбда-выражение возвращало коллекцию элементов.

Если функция-преобразователь возвращает коллекцию элементов (а не отдельный элемент), возможно, стоит применять `flatMap` в зависимости от типа, который вы хотите получить. Если вам нужна коллекция коллекций, выбирайте `map`, если вы предпочитаете избавиться от внутреннего вложения — `flatMap`.

Фильтрация данных

Вторая категория функций в функциональном программировании — это фильтры. *Фильтр* принимает функцию, которая определяет, какие элементы должны войти в возвращаемый список. Функции-фильтры реализуются несколькими способами.

Например, функция `take` — ее мы уже встречали ранее — отбрасывает все элементы свыше заданного количества. Похожая функция `drop` отбрасывает заданное количество элементов от начала коллекции.

Тем не менее на практике чаще всего применяют функцию `filter`.

filter

Функция `filter` получает функцию-предикат, которая проверяет каждый элемент на соответствие условию и возвращает логическое значение. Если значение предиката — истина, то элемент будет добавлен в новую коллекцию, возвращаемую `filter`. Если значение предиката — ложь, то элемент не войдет в новую коллекцию.

Хозяин таверны заботится о репутации заведения и хочет поддерживать ее на высоком уровне. Он желает ввести правило, согласно которому посетитель, коше-

лек которого почти пуст, должен покинуть таверну. Удалите из списка посетителя, если у него остается менее 4 монет, используя функцию `filter` для обнаружения такого бедолаги.

Листинг 11.10. Поиск поиздержавшихся посетителей с использованием функции `filter` (Tavern.kt)

```
...
fun visitTavern() {
    ...
    displayPatronBalances(patronGold)

    val departingPatrons: List<String> = patrons
        .filter { patron -> patronGold.getOrDefault(patron, 0.0) < 4.0 }
    departingPatrons.forEach { patron ->
        narrate("$heroName sees $patron departing the tavern")
    }

    narrate("There are still some patrons in the tavern")
    narrate(patrons.joinToString())
}
```

Запустите `NyetHack` и обратите особое внимание на последнюю часть вывода:

```
...
Madrigal sees Tariq Baggins departing the tavern
There are still some patrons in the tavern
Sophie Fernsworth, Sophie Downstrider, Mordoc Downstrider, Tariq Baggins,
    Alex Downstrider
```

(Возможно, вам придется несколько раз запустить программу, пока посетитель не купит достаточно дорогое блюдо.)

Обратите внимание: хотя Мадригал видит, что некоторые посетители покидают таверну, их имена все еще хранятся в списке `patrons`. Дело в том, что `filter`, как и многие операции, включая `map`, возвращает *новый* список. Исходная коллекция остается неизменной независимо от того, изменяема она или нет.

Чтобы посетители действительно покидали таверну, необходимо удалить их имена как из `patrons`, так и из `patronGold`.

Листинг 11.11. Удаление имен посетителей, покинувших таверну (Tavern.kt)

```
...
fun visitTavern() {
    ...
    val departingPatrons: List<String> = patrons
        .filter { patron -> patronGold.getOrDefault(patron, 0.0) < 4.0 }
    patrons -= departingPatrons
    patronGold -= departingPatrons
    departingPatrons.forEach { patron ->
        narrate("$heroName sees $patron departing the tavern")
    }
```



```

    }

    narrate("There are still some patrons in the tavern")
    narrate(patrons.joinToString())
}
...

```

Снова запустите NyetHack и убедитесь, что посетители действительно покидают таверну.

```

...
Madrigal sees Tariq Downstrider departing the tavern
There are still some patrons in the tavern
Alex Ironfoot, Mordoc Fernsworth, Sophie Baggins

```

Комбинирование данных

Третья категория функций, используемых в функциональном программировании, — *комбинаторы*. Функции-комбинаторы берут несколько коллекций и объединяют их в одну новую. (Это отличается от поведения `flatMap`, которая получает одну коллекцию, содержащую другие коллекции.)

zip

Первый пример комбинатора — функция `zip`. При слиянии двух списков вызовом `zip` элементы этих списков объединяются в порядке их перечисления. Например, если применить `zip` к спискам `[1, 2, 3]` и `["a", "b", "c"]`, получится результат вида `["1a", "2b", "3c"]`.

Один из вариантов применения функции `zip` — генерирование имен посетителей. Вместо того чтобы случайно выбирать пары из имени и фамилии, пока не будут сгенерированы пять уникальных имен, можно воспользоваться `zip`, чтобы все посетители имели уникальные, случайным образом выбранные сочетания имени и фамилии.

Листинг 11.12. Объединение имен с использованием `zip` (Tavern.kt)

```

...
fun visitTavern() {
    narrate("$heroName enters $TAVERN_NAME")
    narrate("There are several items for sale:")
    narrate(menuItems.joinToString())

    val patrons: MutableSet<String> = mutableSetOf()
    val patrons: MutableSet<String> = firstNames.shuffled()
        .zip(lastNames.shuffled()) { firstName, lastName -> "$firstName $lastName"
}

    .toMutableSet()

```

```

val patronGold = mutableMapOf(
    TAVERN_MASTER to 86.00,
    heroName to 4.50
)
while (patrons.size < 5) {
    val patronName = "${firstNames.random()} ${lastNames.random()}"
    patrons += patronName
    patrons.forEach { patronName ->
        patronGold += patronName to 6.0
    }

    narrate("$heroName sees several patrons in the tavern:")
    narrate(patrons.joinToString())
    ...
}
...

```

Переданное лямбда-выражение сообщает `zip`, как следует объединять элементы двух коллекций. В данном случае имя через пробел присоединяется к фамилии, образуя полное имя. Функцию преобразования можно опустить; в этом случае `zip` вернет список пар. (Результат будет таким же, как при передаче аргумента `{ firstName, lastName -> firstName to lastName }`.)

Выполните измененный код. Полные имена посетителей снова будут генерироваться случайным образом, но на этот раз имена и фамилии будут уникальными. Например, посетителям могут быть присвоены полные имена *Sophie Downstrider*, *Alex Fernsworth*, *Tariq Baggins* и *Mordoc Ironfoot*.

Каждое из множеств `firstNames` и `lastNames` состоит из четырех элементов, поэтому `zip` возвращает список из четырех элементов. А если бы входные коллекции имели разные размеры? Функция `zip` возвращает список, размер которого соответствует размеру меньшей из двух входных коллекций. Когда каждый элемент меньшей входной коллекции будет объединен с парным элементом, остальные элементы большей входной коллекции игнорируются.

Так как мы теперь не добавляем элемент в `patronGold` при создании каждого посетителя по отдельности, функция `forEach` используется для перебора всех посетителей и предоставления им денег. Такое решение работает, но весь ассоциативный массив можно заполнить одним вызовом `mutableMapOf`. Удалите вызов `forEach`, как показано в листинге 11.13, и не забудьте добавить запятую после `heroName to 4.50`. Мы объясним новый синтаксис после того, как вы его введете.

Листинг 11.13. Расширение аргументов (Tavern.kt)

```

...
fun visitTavern() {
    ...
    val patrons: MutableSet<String> = firstNames.shuffled()
        .zip(lastNames.shuffled()) { firstName, lastName -> "$firstName $lastName"
}

```

```

        .toMutableSet()

    val patronGold = mutableMapOf(
        TAVERN_MASTER to 86.00,
        heroName to 4.50,
        *patrons.map { it to 6.00 }.toTypedArray()
    )
    patrons.forEach { patronName →
        patronGold += patronName to 6.0
    }

    narrate("$heroName sees several patrons in the tavern:")
    narrate(patrons.joinToString())
    ...
}
...

```

`mutableMapOf` получает переменное количество аргументов, о котором подробнее рассказано в разделе «Для любознательных: ключевое слово `vararg`» в конце главы. Обычно `List` не передают построителям коллекций — если только вы не хотите получить коллекцию списков. Вместо этого необходимо распаковать значения из коллекции в отдельные аргументы. Для этого мы воспользовались *оператором распаковки* (`*`).

С оператором распаковки элементы коллекции рассматриваются как отдельные параметры функции, получающей переменное количество аргументов. Одно из ограничений оператора распаковки заключается в том, что он работает только с `Array`, этим и объясняется необходимость также вызывать `toTypedArray`. И хотя применение оператора распаковки ограничивается очень узкой нишей, он весьма удобен, когда возникает необходимость строить коллекции подобным способом.

Почему именно функциональное программирование?

Вернемся к примеру использования `zip` в листинге 11.12. Представьте реализацию этого же задания без API функционального программирования. В Java, например, она могла бы выглядеть приблизительно так:

```

List<String> firstNames = Arrays.asList("Alex", "Mordoc", "Sophie", "Tariq");
List<String> lastNames = Arrays.asList("Ironfoot", "Fernsworth", "Baggins",
    "Downstrider");
Collections.shuffle(firstNames);
Collections.shuffle(lastNames);
List<String> patrons = new ArrayList<>();
for (int i = 0; i < firstNames.size; i++) {
    patrons.add(firstNames.get(i) + " " + lastNames.get(i));
}

```

Такой стиль называется *императивным программированием*. На первый взгляд императивная версия решает задачу примерно с тем же количеством строк кода, что и функциональная версия в листинге 11.12. Но при более внимательном рассмотрении можно заметить, что функциональный подход обладает рядом важных преимуществ, включая неявные *переменные-аккумуляторы*.

Переменные-аккумуляторы (например, `patrons`) неявно определяются в операциях функционального программирования, избавляя разработчика от необходимости создавать временные переменные для хранения промежуточных результатов. Результаты функциональных операций добавляются в накапливаемое значение автоматически, что снижает вероятность ошибки.

Именно благодаря этому обстоятельству в функциональную цепочку так легко добавить новые операции. Сравните с операциями в императивном стиле: без неявных переменных-аккумуляторов новые операции обычно требуют создания новых временных переменных, используемых в преобразованиях.

Другая причина, по которой в функциональные цепочки так легко добавить новые звенья: все функциональные операции предназначены для работы с итерируемыми коллекциями. Допустим, ассоциативный массив `patrons` необходимо отформатировать для представления заказов после построения массива. В императивном стиле для этого пришлось бы добавить следующий код:

```
List<String> formattedOrders = new ArrayList<>();
for (Map.Entry<String, String> favoriteOrder : customerFavorites.entrySet()) {
    formattedOrders.add(favoriteOrder.getKey() + " orders their favorite item - "
        + favoriteOrder.getValue());
}
```

Добавление нового аккумулятора и нового цикла `for`, заполняющего коллекцию данными, приводит к тому, что в программе появляется больше сущностей, больше состояний, за которыми надо следить.

В функциональном стиле дальнейшие операции легко добавляются в цепочку и не нуждаются в дополнительном состоянии. В функциональном стиле то же самое можно реализовать очень просто:

```
.map { "${it.key} orders their favorite menu item - ${it.value}" }
```

Последовательности

В главах 9 и 10 мы познакомили вас с типами коллекций `List`, `Set` и `Map`. Эти коллекции называются *готовыми* (*eager*). При создании экземпляра любого из этих типов он сразу содержит все значения элементов коллекции и предоставляет доступ к ним.

Есть и другой вид коллекций — *отложенные*. Термин «отложенный» (*lazy*) означает, что значение не создается до момента первого обращения к нему. Отложенные

типы коллекций, как и отложенная инициализация, позволяют увеличить производительность, особенно при работе с очень большими коллекциями, потому что значения в них создаются только по необходимости.

В Kotlin имеется встроенный тип отложенной коллекции `Sequence` (последовательность). Последовательности не поддерживают обращение к содержимому по индексам и не отслеживают свой размер. Более того, при работе с ними есть вероятность получить бесконечное количество значений, потому что нет ограничений на количество элементов, которое способна сгенерировать последовательность.

Для последовательности вы объявляете *функцию-итератор*, которая вызывается каждый раз, когда потребуется новое значение. Один из способов объявить последовательность и ее итератор — использовать встроенную в Kotlin функцию `generateSequence`. Она принимает начальное значение, которое станет отправной точкой для последовательности. При обращении к последовательности в функциональном стиле `generateSequence` вызовет указанный вами итератор для получения следующего значения. Например:

```
generateSequence(0) { it + 1 }  
    .onEach { println("The Count says: $it, ah ah ah!") }
```

Если запустить этот фрагмент кода, то функция `onEach` будет выполняться бесконечно.

Чем же хороша отложенная коллекция и когда стоит отдать ей предпочтение перед списком? Допустим, вы хотите написать блок кода для нахождения первых *N* простых чисел — скажем, 1000. Первая попытка может выглядеть так:

```
// Проверяет, является ли число простым  
fun isPrime(number: Int): Boolean {  
    (2 until number)  
        .map { divisor ->  
            if (number % divisor == 0) {  
                return false // Не простое число  
            }  
        }  
    return true  
}  
  
val listOfPrimes = (1..5000)  
    .toList()  
    .filter { isPrime(it) }  
    .take(1000)
```

Недостаток этой реализации в том, что вы не знаете заранее, сколько чисел придется проверить для получения 1000 простых чисел. Наша реализация берет наугад 5000 чисел, но на самом деле этого недостаточно. (Если вам интересно, будет найдено всего 669 простых чисел.)

Это идеальный случай для использования отложенной коллекции, потому что она не требует задавать количество элементов последовательности для проверки:

```
val oneThousandPrimes = generateSequence(3) { value ->
    value + 1
}.filter { isPrime(it) }
.take(1000)
```

В этом решении `generateSequence` последовательно создает новые значения, начиная с 3 (начальное значение) и прибавляя каждый раз по единице. Дальше с помощью расширения `isPrime` происходит фильтрация значений. Так продолжается, пока не будет создано 1000 элементов. Поскольку нет способа узнать, как много элементов проверено, отложенное создание новых элементов происходит до тех пор, пока условие функции `take` не будет выполнено.

В большинстве случаев коллекции, с которыми вы работаете, — небольшие, менее 1000 элементов. При этом беспокоиться о том, что лучше использовать для такого ограниченного набора элементов — последовательность или список, не имеет особого смысла, так как разница в производительности составит всего несколько наносекунд.

Но с огромными коллекциями, состоящими из сотен тысяч элементов, добиться улучшения производительности за счет смены типа коллекции вполне реально. В подобных случаях преобразовать список в последовательность довольно просто:

```
val listOfNumbers = (0 until 10000000).toList()
val sequenceOfNumbers = listOfNumbers.asSequence()
```

Парадигма функционального программирования может требовать постоянного создания новых коллекций, а последовательности предлагают масштабируемый механизм для работы с большими коллекциями.

В этой главе вы познакомились с базовыми инструментами функционального программирования, такими как `map`, `filter` и `zip`, которые упрощают работу с данными. Также вы научились использовать последовательности для эффективной работы с постоянно растущими объемами данных.

В следующей главе мы завершим обзор основных концепций функционального программирования рассказом о функциях областей видимости, входящих в Kotlin.

Для любознательных: профилирование

Тем, кому важна скорость выполнения, Kotlin предлагает полезные функции для профилирования производительности кода: `measureNanoTime` и `measureTimeInMillis`. Обе функции принимают лямбда-функцию как аргумент

и измеряют скорость выполнения кода внутри лямбда-функции. `measureNanoTime` возвращает время в наносекундах, а `measureTimeInMillis` возвращает время в миллисекундах.

Примените эти функции для измерения скорости выполнения:

```
val listInNanos = measureNanoTime {  
    // Цепочка функций для обработки списка  
}  
  
val sequenceInNanos = measureNanoTime {  
    // Цепочка функций для обработки последовательности  
}  
  
println("List completed in $listInNanos ns")  
println("Sequence completed in $sequenceInNanos ns")
```

В качестве эксперимента попробуйте оценить производительность примеров поиска простых чисел со списком и последовательностью. (Измените пример со списком, увеличив число проверяемых значений до 7919, чтобы удалось найти 1000 простых чисел.) Как изменилось время выполнения программы после замены списка последовательностью?

Для любознательных: агрегирование данных

Мы рассмотрели небольшое подмножество API функционального программирования в Kotlin. Однако существует еще одна группа функций — так называемые *агрегатные функции*, сводящие все содержимое коллекции в одно значение. Из этой группы наиболее известна функция `reduce`, но есть и похожая функция — `fold`.

reduce

Функция `reduce` накапливает все значения заданной коллекции в одно выходное значение. Функция `reduce` получает лямбда-выражение, которое имеет два параметра — *аккумулятор* (текущий накапливаемый результат) и следующее значение в коллекции. Возвращаемое значение лямбда-выражения используется как следующее значение для аккумулятора.

В некоторых ситуациях `reduce` вам очень пригодится. Например, рассмотрим таблицу посетителей: каждый посетитель делает собственный заказ, но кухня может обрабатывать группу заказов как один. Для выполнения такого преобразования выполните свертку вызовом `reduce`, то есть сведите заказы в один. Соответствующий код может выглядеть так:

```
val ordersAtTable: List<Order> = listOf(...)
val tableOrder: Order = ordersAtTable.reduce { acc, order -> acc + order }
```

(В этом примере используется тип `Order`. Такого типа не существует, но вы можете создать его самостоятельно. Вы научитесь определять собственные типы в части IV нашей книги.)

Обратите внимание, что `reduce` возвращает одиночный элемент вместо коллекции — итоговый аккумулярированный результат. Если список содержит только один элемент, то лямбда-выражение вызываться не будет и вызов вернет первое (и единственное) значение в коллекции.

fold

Для агрегирования значений также используется функция `fold`. Ее поведение напоминает `reduce`, но с несколькими важными отличиями.

Функции `fold` должно передаваться исходное значение аккумулятора (вместо первого значения из коллекции). Как и в случае с `reduce`, значение аккумулятора обновляется результатом лямбда-выражения, которое вызывается для каждого элемента. Важное преимущество `fold` перед `reduce` в том, что аккумулятору можно назначить любой тип по вашему усмотрению (аккумулятор `reduce` должен относиться к типу, содержащемуся в списке).

Допустим, посетители таверны обязаны платить налог и добавлять чаевые при оплате заказов (они чрезвычайно щедры и всегда после уплаты налога оставляют 20% за отличное обслуживание). Для определения стоимости заказа можно воспользоваться функцией `fold`:

```
val orderSubtotal = menuItemPrices.getOrDefault("Dragon's Breath", 0.0)

val salesTaxPercent = 5
val gratuityPercent = 20
val feePercentages: List<Int> = listOf(salesTaxPercent, gratuityPercent)

val orderTotal: Double = feePercentages.fold(orderSubtotal) { acc, percent ->
    acc * (1 + percent / 100.0)
}

println("Order subtotal: $orderSubtotal")
println("Order total: $orderTotal")
```

Исходное значение аккумулятора `orderSubtotal` (5.91 в случае `Dragon's Breath`) передается лямбда-выражению в `acc`, а первый процент от оплаты — налог (5%) — передается в `percent` (имя по умолчанию этого аргумента — `item`). Затем лямбда-выражение преобразует целочисленный процент в множитель, преобразует произведение для определения цены с учетом налога и возвращает обновленное накапливаемое значение.

В следующем вычислении новое накапливаемое значение — 6.2055 для `Dragon's Breath` — передается лямбда-выражению в `acc`, а второй процент — чаевые (20%) — передается в `percent`. Итоговое значение аккумулятора содержит результат (7.4466 для `Dragon's Breath`) и возвращается вызовом `fold`.

sumBy

Если у вас есть коллекция со значениями, которые требуется просуммировать, можно воспользоваться функцией `sumBy` (для суммирования целых чисел) или `sumByDouble` (для чисел с плавающей точкой). Например, если вы хотите просуммировать стоимость всех блюд в меню, воспользуйтесь `sumByDouble`:

```
val orderTotal = menuItems.sumByDouble { item ->
    menuItemPrices.getOrDefault(item, 0.0)
}
println("Order price: $orderTotal")
```

Функции `sumBy` и `sumByDouble` фактически представляют собой специализированные разновидности `fold`. Если вам нужно просто просуммировать значения, `sumBy` и `sumByDouble` опускают исходное значение и аккумулятор из кода — для удобочитаемости кода. Также обратите внимание, что функцию `sumBy` необязательно вызывать для коллекций с числовыми значениями. Здесь она вызывается для `List<String>`, и это не создает никаких проблем. Если лямбда-выражение возвращает числовое значение, значения можно агрегировать функциями `sumBy` или `sumByDouble`.

Подобно тому как `associate` упрощает вызов `map`, за которым следует вызов `toMap`, `sumBy` делает менее сложным вызов `fold`. Множество функций действуют как сокращенная запись или вспомогательная функция по сравнению с более общими вызовами `map`, `flatMap`, `filter` и `fold`. Обращайте внимание на такие функции и ознакомьтесь с полным списком операций функционального программирования, чтобы использовать их в своих программах. Например, API типа `List` можно найти на kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/.

Для любознательных: ключевое слово `vararg`

Как мы говорили ранее, функции — строители коллекций (такие, как `listOf` и `mapOf`) получают нужное содержимое коллекции в параметре. Этим функциям можно передавать произвольное число аргументов. Но все остальные функции, которые вы объявляли, имеют фиксированное количество параметров; передать лишний аргумент не удастся, потому что Kotlin не будет знать, что с ним делать. Как же работают строители коллекций?

Функции — строители коллекций, а также некоторые другие функции стандартной библиотеки получают переменное количество аргументов. Для этого

они используют ключевое слово `vararg`, которое показано в приведенной ниже сигнатуре функции `listOf`. (Здесь также вы увидите незнакомый синтаксис с `<T>`, о котором мы подробнее расскажем в главе 18.)

```
public fun <T> listOf(vararg elements: T): List<T> = ...
```

Параметры с ключевым словом `vararg` могут получать 0, 1 или несколько аргументов при вызове. Внутри тела функции `listOf` параметр `elements` в действительности имеет тип `Array`, о котором мы говорили в разделе «Для любознательных: типы массивов». Вы можете запросить количество элементов в `elements`, а также перебрать элементы или обратиться к элементу с конкретным индексом по аналогии работы с типами коллекций.

Вы также можете объявлять собственные параметры `vararg`. Проверьте эту возможность в REPL.

Листинг 11.14. Вывод переменного количества сообщений (REPL)

```
fun printAll(vararg messages: String) {  
    println("I have ${messages.size} things to say.")  
    messages.forEach { println(it) }  
}  
  
printAll("Hello, World!", "Madrigal has left the building.")  
I have 2 things to say.  
Hello, World!  
Madrigal has left the building.
```

На практике параметры `vararg` встречаются в стандартной библиотеке чаще, чем вы будете использовать их в своем коде. Тем не менее полезно знать о них, чтобы понимать, как работают строители коллекций. Также `vararg` предоставляет удобный синтаксис на случай, если вам придется объявить функцию, получающую группу элементов, и вы не хотите вручную упаковывать их в тип коллекции.

Для любознательных: Arrow.kt

В этой главе вы познакомились с инструментами функционального программирования, встроенными в стандартную библиотеку Kotlin, такими как `map`, `flatMap` и `filter`.

Kotlin является мультипарадигменным языком. Это означает, что он смешивает приемы объектно-ориентированного, императивного и функционального программирования. Если вы работали исключительно с функциональным языком, таким как Haskell, то знаете, что он предоставляет гораздо более продвинутые приемы функционального программирования, чем Kotlin.

Например, Haskell содержит тип `Maybe` — тип, который включает поддержку какого-то значения или ошибки. Применение типа `Maybe` позволяет сообщить об исключении (например, об ошибке парсинга числовых данных) без его факти-

ческой выдачи. Благодаря этому отпадает необходимость использовать логику `try/catch`.

Обрабатывать исключения без реализации логики `try/catch` очень удобно. Некоторые представляют `try/catch` как форму команды `GOTO`, которая усложняет понимание и сопровождение кода.

Значительное число средств функционального программирования, доступных в Haskell, можно добавить в Kotlin через специальные библиотеки вроде `Arrow.kt` (<http://arrow-kt.io/>).

Например, библиотека `Arrow.kt` включает подобие типа `Maybe` из Haskell с именем `Either`. Использование `Either` позволяет обработать операцию, которая может закончиться сбоем, не прибегая к исключениям и не требуя использовать команду `try/catch`.

В качестве примера рассмотрим функцию, которая преобразует ввод пользователя из строки в `Int`. Если пользователь введет допустимое число, оно будет преобразовано в значение `Int`, в противном случае функция выведет сообщение об ошибке.

При использовании `Either` код выглядит так:

```
fun parse(s: String): Either<NumberFormatException, Int> =
    if (s.matches(Regex("-?[0-9]+"))) {
        Either.Right(s.toInt())
    } else {
        Either.Left(NumberFormatException("$s is not a valid integer."))
    }

val x = parse("123")

val value = when(x) {
    is Either.Left -> when (x.a) {
        is NumberFormatException -> "Not a number!"
        else -> "Unknown error"
    }
    is Either.Right -> "Number that was parsed: ${x.b}"
}
```

Никаких исключений, никаких блоков `try/catch` — простая и понятная логика.

Задание: перестановка ключей и значений в ассоциативном массиве

Используя функциональные средства, изученные в этой главе, напишите функцию с именем `flipValues`, которая позволит менять местами ключ и значение в ассоциативном массиве. Например:

```
val gradesByStudent = mapOf("Josh" to 4.0, "Alex" to 2.0, "Jane" to 3.0)
{Josh=4.0, Alex=2.0, Jane=3.0}
flipValues(gradesByStudent)
{4.0=Josh, 2.0=Alex, 3.0=Jane}
```

Задание: поиск самого популярного пункта меню

В текущей версии блюдо дня случайным образом выбирается из списка любимых блюд. Если блюдо нравится хотя бы одному из посетителей, оно может быть выбрано как блюдо дня. Хозяин таверны предпочел бы, чтобы блюдо дня нравилось всем посетителям.

Обновите код, который определяет блюдо дня выбором позиции меню, чаще всего помеченной посетителями как любимое блюдо. При равной популярности можно вернуть любую из позиций. Чтобы усложнить задачу, попробуйте выполнить это вычисление без объявления новых переменных. (Подсказка: при этом вам пригодятся функции `fold` и `maxOf`. Также задумайтесь над тем, какая разновидность коллекции подойдет для подсчета вхождений каждой позиции меню.)

12. Функции области видимости

Функции области видимости (scope functions) — это универсальные вспомогательные функции из стандартной библиотеки Kotlin, которые помогают писать более выразительный и компактный код. В этой главе вы познакомитесь с шестью наиболее популярными функциями области видимости: `apply`, `let`, `run`, `with`, `also` и `takeIf` — и увидите, что конкретно они делают.

Сначала мы расскажем о каждой из этих функций, а затем одну из них применим в NyetHack.

Каждая функция области видимости вызывается для значения, которое обычно называется *получателем* (receiver). Оно получает лямбда-выражение, определяющее работу, которая должна быть выполнена с этим значением. Термин «получатель» возник потому, что функции области видимости Kotlin во внутренней реализации являются *функциями-расширениями*, а субъекты таких функций обычно называются получателями. Подробнее с функциями-расширениями, которые позволяют гибко определять дополнительные функции для разных типов, вы познакомитесь в главе 19.

apply

Начнем обзор функций области видимости с функции `apply`. Ее можно считать функцией настройки: она позволяет вызвать несколько функций для объекта-получателя и настроить его для дальнейшего использования. После применения лямбда-выражения `apply` возвращает настроенный получатель.

`apply` можно использовать для сокращения количества повторений при подготовке объекта к использованию. Например, вы можете создать сложные правила, чтобы определить элементы в ваших коллекциях. В NyetHack не все посетители могут вести себя случайным образом: скажем, у каждого жителя Кронштадта есть свое привычное время, когда он посещает таверну.

Один из способов реализовать такие сложные правила — создание коллекции `MutableList` и вручную заполнить список. Если вместо `MutableList` вы хотите получить список `List`, доступный только для чтения, вам также придется объявить вторую переменную:

```

val patrons: MutableList<String> = mutableListOf()
if (isAfterMidnight) { patrons.add("Sidney") }
if (isOpenMicNight) { patrons.add("Janet") }
if (isHappyHour) { patrons.add("Jamie") }
if (patrons.contains("Janet") || patrons.contains("Jamie")) { patrons.add("Hal"
}

val guestList: List<String> = patrons.toList()

```

Используя `apply`, ту же конфигурацию удастся реализовать с меньшим количеством повторений и без переменной `patrons`:

```

val guestList: List<String> = mutableListOf<String>().apply {
    if (isAfterMidnight) { add("Sidney") }
    if (isOpenMicNight) { add("Janet") }
    if (isHappyHour) { add("Jamie") }
    if (contains("Janet") || contains("Jamie")) { add("Hal" }
}.toList()

```

`apply` позволяет исключить имя переменной из каждого вызова функции, выполняемого для настройки получателя, потому что `apply` ограничивает *область видимости* каждого вызова функции внутри лямбда-функции тем получателем, для которого вызвана сама функция.

Такое поведение иногда называют *ограничением относительной области видимости* (relative scoping), потому что вызовы всех функций внутри лямбда-выражения рассматриваются относительно получателя. Также можно сказать, что функции *неявно вызываются* для получателя.

let

Еще одна часто используемая функция области видимости — это `let`, с которой вы уже познакомились в главе 7. `let` определяет переменную в области видимости, предоставленной лямбда-выражением, и передает имя получателя в аргументе. Это позволяет сослаться на получателя ключевым словом `it`, о котором вы узнали в главе 8.

Только что появившийся в городе Мадригал наверняка привлечет внимание. Посетитель таверны (скорее всего, первый увидевший героя) подойдет, представится и поприветствует его. Для вывода приветствия можно воспользоваться функцией `let`:

```

val patrons: List<String> = listOf(...)
val greeting = patrons.first().let {
    "$it walks over to Madrigal and says, \"Hi! I'm $it. Welcome to
Kronstadt!\""
}

```

Без `let` пришлось бы присвоить первый элемент переменной, чтобы программа помнила посетителя, который поприветствует героя первым:

```
val patrons: List<String> = listOf(...)
val friendlyPatron = patrons.first()
val greeting = "$friendlyPatron walks over to Madrigal and says, \"Hi! \" +
    \"I'm $friendlyPatron. Welcome to Kronstadt!\""
```

В сочетании с другими элементами синтаксиса Kotlin функция `let` дает дополнительные преимущества. Ключевое слово `let` мы впервые показали в главе 7 как часть механизма обработки `null`-безопасности:

```
censoredQuest?.let {
    println(
        """
        |$HERO_NAME approaches the bounty board. It reads:
        |   "$censoredQuest"
        |""".trimMargin()
    )
}
```

Вернемся к примеру, когда посетитель таверны приветствует героя. В зависимости от времени суток может оказаться, что Мадригал заходит в пустую таверну. Чтобы учесть такую возможность, следует использовать `firstOrNull` вместо `first`, чтобы обработать граничный случай без сбоя.

```
val patrons: List<String> = listOf(...)
val greeting = patrons.firstOrNull()?.let {
    "$it walks over to Madrigal and says, \"Hi! I'm $it. Welcome to
    Kronstadt!\""
} ?: "Nobody greets Madrigal because the tavern is empty"
```

Использование оператора безопасного вызова (`?.`) означает, что `let` выполняется в том и только в том случае, если получатель отличен от `null`. В данном примере это означает, что `let` выполняется, только если `firstOrNull` возвращает имя посетителя. Используя `let` с безопасным вызовом, можно гарантировать, что аргумент `it` отличен от `null` и в лямбда-выражении удастся безопасно выполнять операции без повторной проверки на `null`.

Сравните пример, приведенный выше, с версией, показанной ниже, в которой `let` не используется:

```
val patrons: List<String> = listOf(...)
val friendlyPatron = patrons.firstOrNull()
val greeting = if (friendlyPatron != null) {
    "$friendlyPatron walks over to Madrigal and says, \"Hi! \" +
    \"I'm $friendlyPatron. Welcome to Kronstadt!\""
} else {
    "Nobody greets Madrigal because the tavern is empty"
}
```

Эта версия функционально эквивалентна, но занимает больше места. Со структурой `if/else` переменная `friendlyPatron` используется трижды: один раз в условии и два раза для создания итоговой строки. А `let` позволяет опустить объявление промежуточных переменных.

Помните, что функция `let` сама по себе не гарантирует `null`-безопасности. `null`-безопасность обеспечивается комбинацией функции области видимости и безопасного вызова.

Функцию `apply` тоже используют с безопасным вызовом, но реже. Большинство программистов Kotlin, стремящихся обеспечить `null`-безопасность, выбирают `let`, но есть и такие, кто отдает предпочтение функции `run` (мы ее рассмотрим далее). У этих функций много общего, но возможно, какая-то вам понравится больше.

Функция `let` вызывается для любого получателя и возвращает результат вычисления предоставленного лямбда-выражения. В приведенном примере `let` вызывается для строки `patrons.firstOrNull()`. Лямбда-выражение, переданное `let`, принимает имя получателя, с которым оно вызывается, в своем единственном аргументе. Это позволяет обратиться к аргументу по идентификатору `it`.

Стоит отметить некоторые различия между `let` и `apply`: как было показано, `let` передает имя получателя предоставленному лямбда-выражению, тогда как `apply` ничего не передает. Кроме того, `apply` возвращает имя текущего получателя после завершения лямбда-выражения. С другой стороны, `let` возвращает последнюю строку лямбда-выражения (*лямбда-результат*).

run

Следующей в нашем списке функций области видимости идет `run`. Как и `apply`, функция `run` ограничивает относительную область видимости, но возвращает результат лямбда-выражения вместо имени самого получателя, как `let`.

Допустим, вы хотите отслеживать музыку, которая играет в таверне. Для этого можно написать такой код:

```
val tavernPlaylist = mutableListOf("Korobeiniki", "Kalinka", "Katyusha")
val nowPlayingMessage: String = tavernPlaylist.run {
    shuffle()
    "${first()} is currently playing in the tavern"
}
```

Функция `shuffle` неявно применяется к имени получателя — экземпляру `List`. Все происходит так же, как с уже знакомыми вам функциями `add` и `contains`, продемонстрированными с `apply`. Но в отличие от `apply`, функция `run` возвращает лямбда-результат — в данном случае сообщение с названием русской народной песни, которая в настоящий момент играет в таверне.

Кстати говоря, существует вторая разновидность `run`, которая не вызывается для получателя. Эта форма встречается намного реже, но мы приводим ее ради полноты информации:

```
val healthPoints = 90
val healthStatus = run {
    if (healthPoints == 100) "perfect health" else "has injuries"
}
```

with

Функция `with` имеет много общего с `run`. Она ведет себя так же, но использует другое условие вызова. В отличие от функций обратного вызова, о которых мы рассказали ранее, `with` требует, чтобы аргумент передавался в первом параметре (вместо вызова функции области видимости для типа получателя):

```
val nameTooLong = with("Polarcubis, Supreme Master of NyetHack") {
    length >= 20
}
```

Ранее функции области видимости вызывались для самого значения, например, `"Polarcubis".run{ ... }`. Однако `with` нарушает это соглашение. Функция `with` вызывается *на первом месте*, как в `with("Polarcubis") { ... }`.

Из-за такого расхождения с другими функциями области видимости мы рекомендуем использовать `run` вместо `with` — такое решение более последовательно, более привычно и лучше читается. Однако выбор между `run` и `with` — скорее стилистический. В зависимости от контекста можно отдать предпочтение как `run`, так и `with`:

```
val player: Player = ...
val monster: Goblin = ...

// Оба вызова компилируются в `player.fight(monster)`
with(player) { fight(monster) }
player.run { fight(monster) }
```

Эти вызовы работают одинаково и возвращают одинаковый результат.

also

Функция `also` похожа на `let`. Как и `let`, функция `also` передает имя получателя, для которого она вызывается, в аргументе лямбда-выражения. Однако у `let` и `also` есть важное различие: `also` возвращает имя получателя вместо лямбда-результата, как и функция `apply`.

Из-за этого функция `also` особенно удобна для добавления различных побочных эффектов от общего источника. В следующем примере `also` дважды вызывается для

выполнения двух разных операций: первая выводит имя файла, а вторая записывает содержимое файла в переменную `fileContents`.

```
var fileContents: List<String>
File("file.txt")
    .also { print(it.name) }
    .readLines()
    .also { fileContents = it }
```

Так как функция `also` возвращает имя получателя, а не результат лямбда-выражения, вы можете продолжить строить цепочку, присоединяя вызовы функций к имени первоначального получателя.

takeIf

Последняя функция области видимости, которую мы рассмотрим, — `takeIf` — немного отличается от других: она вычисляет условие, или предикат, заданное лямбда-выражением, которое возвращает истинное или ложное значение. Если условие истинно, `takeIf` возвращает имя получателя. Если условие ложно, она вернет `null`.

В следующем примере файл читается только в том случае, если файл существует:

```
val fileContents = File("myfile.txt")
    .takeIf { it.exists() }
    ?.readText()
```

Без `takeIf` запись получается более громоздкой:

```
val file = File("myfile.txt")
val fileContents = if (file.exists()) {
    file.readText()
} else {
    null
}
```

Вариант с `takeIf` не требует временной переменной `file` и явного возврата `null`. `takeIf` удобно использовать для проверки условия перед присваиванием значения переменной или продолжением работы. Концептуально функция `takeIf` напоминает оператор `if`, но у нее есть преимущество прямого вызова для экземпляра, что часто позволяет избавиться от временной переменной.

Обзор функций области видимости мы почти завершили, но у `takeIf` есть дополнительная функция, которую необходимо упомянуть ради полноты информации, — `takeUnless`. Функция `takeUnless` аналогична `takeIf`, не считая того, что при ложном условии она возвращает исходное значение. Следующий пример читает файл, если он не является скрытым (и возвращает `null` в ином случае):

```
val fileContents = File("myfile.txt").takeUnless { it.isHidden }?.readText()
```

Мы рекомендуем отдавать предпочтение `takeIf`, особенно для сложных условий, так как эта функция часто делает условия более понятными. Сравните следующие две формулировки:

- «Вернуть значение, если условие истинно» — `takeIf`.
- «Вернуть значение, кроме случая, когда условие истинно» — `takeUnless`.

Если вам пришлось на секунду задуматься, чтобы осознать вторую формулировку, то мы на вашей стороне — `takeUnless` предоставляет менее естественный способ описания логики, которую вы хотите выразить.

С простыми условиями (как в приведенном выше примере) `takeUnless` проблем не создает. Но в более сложных ситуациях понять логику этой функции будет труднее. Мы рекомендуем ограничить применение `takeUnless` очень короткими условиями, которые в противном случае пришлось бы инвертировать в лямбда-выражении вашего предиката.

Использование функций области видимости

Таблица 12.1 содержит краткую информацию о функциях области видимости Kotlin, описанных в этой главе.

Таблица 12.1. Функции области видимости

Функция	Передает имя получателя лямбда-выражению как аргумент?	Предоставляет относительную область видимости?	Что возвращает
<code>let</code>	Да	Нет	Лямбда-результат
<code>apply</code>	Нет	Да	Имя получателя
<code>run</code> ¹	Нет	Да	Лямбда-результат
<code>with</code> ²	Нет	Да	Лямбда-результат
<code>also</code>	Да	Нет	Имя получателя
<code>takeIf</code>	Да	Нет	Версию получателя, допускающую null
<code>takeUnless</code>	Да	Нет	Версию получателя, допускающую null

¹ Версия `run`, вызываемая без имени получателя (применяется реже), не передает его имя, не предоставляет относительную область видимости и возвращает лямбда-результат.

² `with` вызывается не в контексте получателя, например `"hello.with{...}"`. Вместо этого функция рассматривает первый аргумент как получателя, а второй как лямбда-выражение, например: `"with("hello"){...}"`. Это единственная функция области видимости, которая работает таким образом, и она используется реже других.

Как подсказывает название, функции области видимости лучше всего использовать для временного создания новой области видимости или изменения области видимости, в которой выполняется ваша программа. Каждый раз, когда вы применяете временную переменную, стоит подумать об использовании функции области видимости.

Вспомните NyetHack. Есть ли в программе фрагменты, которые прямо запрашиваются на применение функции области видимости? Так как мы используем операторы функционального программирования Kotlin, возможностей что-то улучшить за счет функций области видимости не так много.

Но взгляните на логику ухода посетителей в Tavern.kt:

```
val departingPatrons: List<String> = patrons
    .filter { patron -> patronGold.getDefault(patron, 0.0) < 4.0 }
patrons -= departingPatrons
patronGold -= departingPatrons
departingPatrons.forEach { patron ->
    narrate("$heroName sees $patron departing the tavern")
}
```

В этом блоке кода `departingPatrons` используется несколько раз, и функции области видимости позволяют его усовершенствовать. При желании здесь можно использовать любые функции области видимости, представленные в этой главе. Впрочем, мы рекомендуем `also` для инкапсуляции побочных эффектов. Откройте Tavern.kt и попробуйте сами.

Листинг 12.1. Использование `also` (Tavern.kt)

```
...
fun visitTavern() {
    ...
    displayPatronBalances(patronGold)

    val departingPatrons: List<String> = patrons
    patrons
    .filter { patron -> patronGold.getDefault(patron, 0.0) < 4.0 }
    patrons -= departingPatrons
    patronGold -= departingPatrons
    .also { departingPatrons ->
        patrons -= departingPatrons
        patronGold -= departingPatrons
    }
    departingPatrons.forEach { patron ->
    .forEach { patron ->
        narrate("$heroName sees $patron departing the tavern")
    }

    narrate("There are still some patrons in the tavern")
    narrate(patrons.joinToString())
}
...
```

Запустите `NyetHack` и убедитесь, что программа работает так же, как прежде.

Это изменение позволяет собрать всю логику ухода посетителей в одной команде. Также нам удалось полностью отказаться от переменной `departingPatrons`, так как к ней теперь не нужно обращаться позднее.

В этой главе вы узнали, как упростить код при помощи функций области видимости. Они дают возможность писать код не только лаконичный, но и передающий особый дух языка Kotlin. Мы будем использовать функции области видимости до конца нашей книги — везде, где это уместно.

В этой части вы научились работать с типами коллекций и освоили некоторые приемы функционального программирования, поддерживаемые Kotlin. А в главах следующей части мы сменим курс и займемся другой парадигмой — объектно-ориентированным программированием.

Часть IV

Объектно-ориентированное программирование

Следующие пять глав посвящены еще одной парадигме программирования — объектно-ориентированному программированию (ООП). ООП, появившееся в 1960-е годы, остается популярным, потому что предоставляет набор полезных средств для упрощения структуры программы. Это отличный способ преобразования частей кода в расширяемые компоненты, пригодные для повторного использования, — они называются *классами* и *объектами*.

Вы научитесь определять классы, инициализировать объекты, наследовать классы от других классов и интерфейсов, а также использовать специальные разновидности классов в Kotlin, такие как синглтоны и классы данных. К концу этой части вы превратите NyetHack в полноценную интерактивную игру — с комнатами, которые можно исследовать, и монстрами, с которыми можно сражаться.

13. Классы

Центральное место в объектно-ориентированном программировании занимают *классы*, которые определяют уникальные категории объектов, содержащихся в вашем коде. Классы определяют, какие данные будут хранить эти объекты и какую работу выполнять.

Чтобы сделать NyetHack объектно-ориентированным, определим уникальные типы объектов нашей программы и объявим для них классы. В этой главе мы добавим в NyetHack класс `Player`, который задает конкретные характеристики игрока.

Объявление класса

Класс можно объявить в отдельном файле или вместе с другими элементами, такими как функции и переменные. Объявление класса в отдельном файле дает возможность его расширения со временем — вместе с развитием программы, и именно так мы поступим в NyetHack. Создайте новый файл `Player.kt`: щелкните правой кнопкой мыши на папке `src/main/kotlin`, выберите команду `New ► Kotlin Class/File` и выберите вариант `File` при вводе имени. Затем объявите первый класс при помощи ключевого слова `class`.

Листинг 13.1. Объявление класса `Player` (`Player.kt`)

```
class Player
```

Класс часто объявляют в файле с соответствующим именем, но это необязательно. Можно определить несколько классов в одном файле, и скорее всего, именно так и стоит поступить, если все классы создаются для одной цели.

Итак, класс объявлен. Теперь надо заставить его работать.

Создание экземпляров

Объявление класса напоминает чертеж. Чертеж содержит информацию о том, как построить дом, но самого дома еще нет. Объявление класса `Player` работает похожим образом: пока что игрока нет, вы только нарисовали чертеж для его создания.

Когда вы начинаете новую игру в NyetHack, вызывается функция `main`, и одной из первых задач должно стать создание игрового персонажа. Чтобы сконструировать героя, которого можно использовать в NyetHack, сначала нужно создать его *экземпляр* вызовом *конструктора*. В `NyetHack.kt`, где переменные объявляются в функции `main`, создайте экземпляр `Player`, как показано ниже.

Листинг 13.2. Создание экземпляра `Player` (`NyetHack.kt`)

```
var heroName = ""
val player = Player()

fun main() {
    heroName = promptHeroName()

    // changeNarratorMood()
    narrate("$heroName, ${createTitle(heroName)}, heads to the town square")
    visitTavern()
}
...
```

Чтобы вызвать конструктор `Player`, следует указать имя класса, за которым следуют круглые скобки. Это действие создает экземпляр класса `Player`. Говорят, что переменная `player` теперь «содержит экземпляр класса `Player`». Каждое определение класса в Kotlin создает соответствующий тип, так что переменная `player` имеет тип `Player`.

Термин «конструктор» говорит сам за себя: он конструирует. Если конкретнее, он создает экземпляр и готовит его к использованию. Синтаксис вызова конструктора похож на вызов функции: в круглых скобках передаются аргументы для его параметров. О других способах создания экземпляров мы поговорим в главе 14.

Группировка логики объектов в коде при помощи классов способствует организации кода при масштабировании. С развитием NyetHack в программу будут добавляться новые классы, каждый из которых наделен собственными обязанностями.

Теперь, когда у вас есть экземпляр `Player`, что с ним можно сделать?

Функции класса

Объявление класса может включать два вида содержимого: *поведение* и *данные*. В NyetHack игрок должен уметь совершать разные действия: участвовать в сражении, перемещаться, произносить заклинания, проверять снаряжение и т. д. Вы определяете поведение класса, добавляя определения функций в тело класса. Объявленные внутри класса функции называют *функциями класса*.

Вы уже определили некоторые действия игрока в NyetHack. Теперь реорганизуем код и добавим в него новое поведение и данные, свойственные классу.

Начнем с определения первого поведения `Player` — добавим в класс `Player` функцию `castFireball`, позволяющую игроку произносить заклинание.

Листинг 13.3. Определение функции класса (`Player.kt`)

```
class Player {  
    fun castFireball(numFireballs: Int = 2) {  
        narrate("A glass of Fireball springs into existence (x$numFireballs)")  
    }  
}
```

Здесь *тело класса* `Player` определяется в паре фигурных скобок. Тело класса содержит определение поведения и данных класса — по аналогии с тем, как действия функции задаются в теле функции.

Зачем перемещать `castFireball` в `Player`? В `NyetHack` заклинание для получения дурманящего напитка произносит игрок — этот напиток нельзя получить без экземпляра `Player`, и бокал с `Fireball`¹ создает конкретный игрок, который вызвал `castFireball`. Когда мы определяем `castFireball` как функцию класса, вызываемую для конкретного экземпляра класса, мы следуем этой логике.

В `NyetHack.kt` добавьте вызов `castFireball` как вызов функции класса в `main`.

Листинг 13.4. Вызов функции класса (`NyetHack.kt`)

```
var heroName = ""  
val player = Player()  
fun main() {  
    heroName = promptHeroName()  
    // changeNarratorMood()  
    narrate("$heroName, ${createTitle(heroName)}, heads to the town square")  
    visitTavern()  
    player.castFireball()  
}  
...
```

Запустите `NyetHack` и убедитесь, что игрок успешно творит заклинание.

Видимость и инкапсуляция

Добавляя поведение в класс в виде функций класса (и данные в виде свойств класса, как мы увидим далее), вы создаете описание того, чем класс может быть и что может делать, и это описание доступно всем, у кого есть экземпляр класса.

По умолчанию любая функция или свойство без модификатора видимости будут доступны всем, то есть из любого файла или функции в программе. Так как вы не указали модификатор видимости для `castFireball`, она может быть вызвана откуда угодно.

¹ Fireball — канадский виски с корицей. — *Примеч. пер.*

В таких случаях, как с `castFireball`, требуется, чтобы другие части кода имели доступ к свойствам класса или вызывали функции класса. Но в программе могут быть иные функции класса или свойства, которые не должны быть доступны повсюду.

С ростом количества классов растет и сложность кодовой базы. Соккрытие деталей реализации, которые не должны быть доступны другим частям программы, помогает сохранить ясную логику кода, а сам код сделать лаконичным. В этом вам поможет ограничение видимости.

В то время как общедоступную, или публичную (`public`), функцию класса можно вызвать из любого места программы, приватная (`private`) функция доступна только в пределах класса, в котором объявлена. Ограничение видимости некоторых свойств и/или функций класса в объектно-ориентированном программировании называется *инкапсуляцией*. Согласно этой идее, класс должен выборочно предоставлять функции и свойства другим объектам для взаимодействия с ним. Все, что не нужно раскрывать, включая детали реализации функций и свойств, должно быть скрыто.

Например, хотя функция `castFireball` вызывается из `main`, функцию `main` не интересует, как реализована `castFireball`. Для нее важен только результат — появление стакана с `Fireball`. Поэтому хотя сама функция может быть доступна, детали ее реализации неважны для вызывающей стороны.

Проще говоря, создавая классы, раскрывайте подробности реализации, только если это необходимо.

В табл. 13.1 приводится список модификаторов видимости.

Таблица 13.1. Модификаторы видимости

Модификатор	Описание
<code>public</code> (по умолчанию)	Функция или свойство будут доступны вне класса. По умолчанию функции и свойства без модификатора видимости являются публичными
<code>private</code>	Функция или свойство будут доступны только внутри класса
<code>protected</code>	Функция или свойство будут доступны только внутри класса или подкласса
<code>internal</code>	Функция или свойство будут доступны внутри модуля

О ключевом слове `protected` мы расскажем в главе 15.

Модификаторы видимости работают одинаково независимо от того, для какой платформы предназначен код Kotlin. Если вам знаком язык Java, обратите внимание, что в Kotlin отсутствует уровень видимости, ограниченный рамками пакета. Причину мы объясним в разделе «Для любознательных: ограничение видимости рамками пакета» в конце главы.

Свойства класса

Определения функций класса описывают поведение, связанное с классом. Определения данных, чаще называемые *свойствами класса*, представляют собой атрибуты, необходимые для представления определенного состояния или характеристик класса. Например, свойства класса `Player` могут задавать имя игрока, состояние здоровья, расу, мировоззрение, пол и пр.

На данный момент имя игрока объявляется в функции `main`, но класс лучше подойдет для хранения такой информации. Обновите код `Player.kt`, добавив свойство `name`. (Значение `name` выглядит неряшливо, но как говорится, в этом безумии есть система. Просто введите код, показанный ниже.)

Листинг 13.5. Определение свойства `name` (`Player.kt`)

```
class Player {
    val name = "madrigal"

    fun castFireball(numFireballs: Int = 2) {
        narrate("A glass of Fireball springs into existence (x$numFireballs)")
    }
}
```

Вы добавили свойство `name` в тело класса `Player`. Оно включается как актуальные данные, которые должны содержаться в экземпляре `Player`. Обратите внимание, что свойство `name` объявлено как `val`. Как и переменные, свойства могут объявляться с ключевыми словами `var` и `val`, и это позволяет или не позволяет изменять хранящуюся в них информацию. Об изменяемости свойств мы поговорим позже в этой главе.

При конструировании экземпляра класса все его свойства должны получить значения. Это означает, что свойствам класса, в отличие от других переменных, необходимо присвоить исходные значения. Например, следующий код недопустим, потому что `name` не присваивается значение при объявлении:

```
class Player {
    var name: String
}
```

О нюансах инициализации свойств и классов мы расскажем в главе 14.

Исключите объявление `heroName` из `NyetHack.kt`.

Листинг 13.6. Удаление `heroName` из `main` (`NyetHack.kt`)

```
var heroName: String = ""
val player = Player()

fun main() {
    heroName = promptHeroName()
}
```

```

// changeNarratorMood()
narrate("$heroName, ${createTitle(heroName)}, heads to the town square")
narrate("${player.name}, ${createTitle(player.name)}, heads to the town
square")

visitTavern()
player.castFireball()
}
...

```

Теперь, когда `name` является свойством `Player`, мы используем *точечный синтаксис* для обращения к свойству `name` переменной `player` при вызове `narrate`. Точечный синтаксис используется для чтения и записи свойств и вызова функций для экземпляра объекта.

При попытке запустить программу в ее текущем состоянии вы получите несколько ошибок компилятора в `Tavern.kt`, и в каждом описании сообщается о «не-разрешенной ссылке `heroName`» (`Unresolved reference: heroName`). Откройте `Tavern.kt` и исправьте ошибки, заменяя обращения `heroName` на `player.name`.

Листинг 13.7. Разрешение ссылок на свойство `name` класса `Player` (`Tavern.kt`)

```

...
fun visitTavern() {
    narrate("$heroName{player.name} enters $TAVERN_NAME")
    narrate("There are several items for sale:")
    narrate(menuItems.joinToString())
    ...
    val patronGold = mutableMapOf(
        TAVERN_MASTER to 86.00,
        heroNameplayer.name to 4.50,
        *patrons.map { it to 6.00 }.toTypedArray()
    )
    narrate("$heroName{player.name} sees several patrons in the tavern:")
    narrate(patrons.joinToString())
    ...
    patrons.filter { patron -> patronGold.getOrDefault(patron, 0.0) < 4.0 }
        .also { departingPatrons ->
            patrons -= departingPatrons
            patronGold -= departingPatrons
        }
        .forEach { patron ->
            narrate("$heroName{player.name} sees $patron departing the tavern")
        }
    ...
}
...
private fun displayPatronBalances(patronGold: Map<String, Double>) {
    narrate("$heroName{player.name} intuitively knows how much money each patron
        has")
    patronGold.forEach { (patron, balance) ->

```

```
        narrate("$patron has ${"%0.2f".format(balance)} gold")
    }
}
```

Запустите NyetHack. Мадригал снова посещает таверну, как и прежде, но теперь мы обращаемся к свойству `name` из экземпляра класса `Player`, а не к переменной верхнего уровня в `NyetHack.kt`.

Позже в этой главе мы переработаем NyetHack для перемещения других данных, принадлежащих классу `Player`, в определение класса.

Get-методы и set-методы свойств

Свойства моделируют характеристики каждого экземпляра класса. Они также позволяют другим объектам взаимодействовать с данными в классе с использованием простого и компактного синтаксиса. Подобное взаимодействие происходит через методы свойств.

Для каждого определенного свойства Kotlin генерирует до трех компонентов: *поле*, *get-метод* (метод чтения, иначе *getter*) и при необходимости *set-метод* (метод записи, иначе *setter*). Поле — это то место, где хранятся данные для свойства. Объявить поле непосредственно в классе нельзя. Kotlin инкапсулирует поля, защищая данные в поле и открывая доступ к ним через геттеры и сеттеры.

Get-метод свойства определяет правила его чтения. Get-методы генерируются для всех свойств. Set-метод определяет правила присваивания значения свойству, поэтому он генерируется только для изменяемых свойств, другими словами, если свойство объявлено с ключевым словом `var`.

Представьте, что вы пришли в ресторан и в меню, помимо прочего, есть спагетти. Вы заказываете их, и официант приносит спагетти с сыром и соусом. Вам не нужен доступ на кухню, официант решает все вопросы сам, в том числе добавлять ли сыр и соус в заказанное блюдо. Ваши действия можно сравнить с вызывающим кодом, а официанта — с get-методом.

Вы посетитель ресторана и не хотите возиться с приготовлением спагетти. Вы хотите сделать заказ и получить его. А ресторану не нужно, чтобы вы слонялись по кухне, где перекладывали бы ингредиенты и посуду, как вам заблагорассудится. Именно так работает инкапсуляция.

Несмотря на то что методы свойств по умолчанию генерируются языком Kotlin, вы можете изменить поведение геттеров и сеттеров, если хотите конкретизировать, как должны осуществляться чтение и запись данных. Для этого следует написать пользовательские get- и set-методы.

Чтобы увидеть, как работают пользовательские get-методы, добавьте get-метод в определение свойства `name`. Он будет следить за тем, чтобы при обращении к этому свойству возвращалась строка, начинающаяся с прописной буквы.

Листинг 13.8. Определение пользовательского get-метода (Player.kt)

```
class Player {  
    val name = "madrigal"  
        get() = field.replaceFirstChar { it.uppercase() }  
  
    fun castFireball(numFireballs: Int = 2) {  
        narrate("A glass of Fireball springs into existence (x$numFireballs)")  
    }  
}
```

Объявляя свой get-метод для свойства, вы меняете его поведение при попытке прочитать значение. Так как `name` содержит имя собственное, то при обращении к нему должна возвращаться строка, начинающаяся с прописной буквы. Наш пользовательский get-метод гарантирует это.

Запустите `NyetHack` и убедитесь, что имя `Madrigal` пишется с прописной `M`.

Идентификатор `field` в примере ссылается на поле со значением, которое Kotlin автоматически поддерживает для свойства. Поле — это область памяти, из которой геттеры и сеттеры читают и куда записывают данные, представляющие свойство. Это как ингредиенты на кухне ресторана: вызывающая сторона никогда не видит исходные данные (ингредиенты), только то, что вернет get-метод. Более того, к полю можно обращаться только внутри get- и set-методов этого свойства.

При возвращении версии имени с прописной буквой содержимое самого поля не меняется. Если значение, присвоенное `name`, начинается со строчной буквы, как у нас в коде, оно останется таковым и после вызова get-метода.

Set-метод, напротив, *изменяет* поле свойства. Добавьте set-метод в определение свойства `name` и используйте в нем функцию `trim`, чтобы убрать начальные и конечные пробелы из передаваемого значения.

Листинг 13.9. Определение пользовательского set-метода (Player.kt)

```
class Player {  
    val name = "madrigal"  
        get() = field.replaceFirstChar { it.uppercase() }  
        set(value) {  
            field = value.trim()  
        }  
  
    fun castFireball(numFireballs: Int = 2) {  
        narrate("A glass of Fireball springs into existence (x$numFireballs)")  
    }  
}
```

Добавление set-метода в это свойство создает проблему, о которой предупредит IntelliJ (рис. 13.1).



Рис. 13.1. Свойства val доступны только для чтения

Свойство name объявлено как val, поэтому оно доступно только для чтения и не может быть изменено даже set-методом. Это защищает значения val от изменений без вашего согласия.

В своей подсказке IntelliJ сообщает важные сведения о set-методах: они вызываются для присваивания значений свойствам. Было бы нелогично (а на самом деле это ошибка) объявлять set-метод для свойства val, ведь если значение доступно только для чтения, то set-метод никогда не сможет выполнить свою работу.

Чтобы иметь возможность менять имя игрока, нужно заменить val на var в объявлении свойства name. (Обратите внимание: с этого момента мы будем показывать все изменения в коде во всех случаях, когда это возможно.)

Листинг 13.10. Переход к изменяемой форме name (Player.kt)

```

class Player {
    var name = "madrigal"
    get() = field.replaceFirstChar { it.uppercase() }
    set(value) {
        field = value.trim()
    }

    fun castFireball(numFireballs: Int = 2) {
        narrate("A glass of Fireball springs into existence ($numFireballs)")
    }
}

```

Теперь name можно изменять по правилам, указанным в set-методе, и предупреждения от IntelliJ исчезли.

Get-методы свойств вызываются в том же синтаксисе, что и другие переменные, с которыми вы уже знакомы. Set-методы свойства вызываются автоматически при попытке присвоить новое значение свойству с помощью оператора присваивания.

Попытайтесь изменить имя игрока вне класса Player в Kotlin REPL. Сначала вам придется перезагрузить REPL кнопкой  Build and restart, потому что среда

распознает изменение в `Player`. (Не забудьте включить пробел в строку имени, чтобы у `set`-метода была работа.)

Листинг 13.11. Изменение имени игрока (REPL)

```
val player = Player()
player.name = "estragon "
print(player.name + "TheBrave")
EstragonTheBrave
```

Здесь виден результат работы `get`- и `set`-методов для нового значения `name`.

Присваивание новых значений свойствам меняет состояние класса, которому они принадлежат. Если бы `name` все еще было объявлено с `val`, то в результате выполнения кода, который вы только что ввели в REPL, появилось бы следующее сообщение об ошибке:

```
error: val cannot be reassigned
```

Видимость свойств

Свойства отличаются от переменных, объявленных локально внутри функции. Свойства определяются на уровне класса. Как следствие, они могут быть доступны для других классов, если это позволяет их видимость. Слишком широкая видимость свойства порождает проблемы: если другие классы получают доступ к данным `Player`, они смогут вносить изменения в экземпляр класса `Player` по своему усмотрению.

Свойства предоставляют возможность детализированного управления чтением и записью данных через методы свойств. Все свойства имеют `get`-методы, а все `var`-свойства имеют `set`-методы независимо от того, определяете вы для них нестандартное поведение или нет. По умолчанию видимость методов свойств совпадает с видимостью самих свойств. То есть если свойство объявлено как общедоступное, то его методы тоже будут общедоступными.

А если вы решите открыть доступ к свойству, но не желаете раскрывать его `set`-метод? Объявите область видимости `set`-метода отдельно, сделав ее приватной.

Листинг 13.12. Скрытие `set`-метода `name` (`Player.kt`)

```
class Player {
    var name = "madrigal"
    get() = field.replaceFirstChar { it.uppercase() }
    private set(value) {
        field = value.trim()
    }

    fun castFireball(numFireballs: Int = 2) {
        narrate("A glass of Fireball springs into existence (x$numFireballs)")
    }
}
```

Теперь к `name` можно обращаться для чтения из любой части `NyetHack`, но изменить его сможет только сам экземпляр `Player`. Этот прием весьма полезен, когда требуется запретить изменение свойства другими частями вашего приложения.

Видимость `set`-метода не может быть шире видимости свойства, для которого он определяется. Можно ограничить доступ к свойству, определив более узкую видимость `get`-методов, но нельзя присвоить методам свойства более широкую видимость, чем объявлена для самого свойства.

Также можно применить модификатор видимости к `set`-методу без определения собственного поведения `set`. Для этого опустите круглые и фигурные скобки после ключевого слова `set`:

```
class Player {  
    var name = "madrigal"  
    private set  
    ...  
}
```

Как и прежде, это гарантирует, что свойство `name` останется доступным из любой части `NyetHack`, но изменить его сможет только сам экземпляр `Player` — без определения дополнительного поведения для изменений. Этот компактный синтаксис весьма полезен, когда вы хотите запретить внешние изменения переменной, но вам не нужно применять собственную логику в сеттере.

Что касается видимости `get`-метода, вы не сможете использовать модификаторы видимости, чтобы уровень видимости геттера отличался от уровня видимости свойства. (Указать модификатор видимости для геттера возможно, но он должен соответствовать видимости свойства, так что практического смысла в этом нет.)

Вычисляемые свойства

Ранее мы упоминали, что при объявлении свойства всегда создается поле, в котором хранится фактическое значение. Это, конечно, правда... Кроме одного случая — так называемых *вычисляемых свойств*. Вычисляемым называют свойство, для которого геттер (и сеттер, если свойство является `var`) переопределяется без использования поля. В таких случаях Kotlin не генерирует поле.

Рассмотрим функцию `createTitle` из файла `NyetHack.kt`. Она создает титул игрока и зависит от его имени; так что ее следовало бы поместить в класс `Player`. Но хотя ее можно было бы просто переместить в `Player`, стоит учесть один факт: титул — данные, а не поведение. Таким образом, его действительно следует размещать в свойстве, но он должен иметь возможность реагировать на изменения имени игрока. Вычисляемое свойство позволит хранить значение в свойстве, но при этом гарантировать, что оно всегда будет актуальным.

Добавьте вычисляемое свойство для титула игрока, а также новую функцию, которая позволяет игроку сменить имя.

Листинг 13.13. Добавление титула в Player (Player.kt)

```
class Player {

    var name = "madrigal"
    get() = field.replaceFirstChar { it.uppercase() }
    private set(value) {
        field = value.trim()
    }

    val title: String
    get() = when {
        name.all { it.isDigit() } -> "The Identifiable"
        name.none { it.isLetter() } -> "The Witness Protection Member"
        name.count { it.lowercase() in "aeiou" } > 4 -> "The Master of Vowels"
        else -> "The Renowned Hero"
    }

    fun castFireball(numFireballs: Int = 2) {
        narrate("A glass of Fireball springs into existence (x$numFireballs)")
    }

    fun changeName(newName: String) {
        narrate("$name legally changes their name to $newName")
        name = newName
    }
}
```

Опробуйте новое вычисляемое свойство в `NyetHack.kt` — как до, так и после того, как Мадригал подаст заявку на изменение своего имени.

Листинг 13.14. Использование вычисляемых свойств (NyetHack.kt)

```
val player = Player()

fun main() {
    narrate("${player.name} is ${player.title}")
    player.changeName("Aurelia")
    // changeNarratorMood()
    narrate("${player.name}, ${createTitle(player.name)}, heads to the town
    square")
    narrate("${player.name}, ${player.title}, heads to the town square")

    visitTavern()
    player.castFireball()
}

private fun promptHeroName(): String {
    ...
}

private fun createTitle(name: String): String {
    return when {
```

```

name.all { it.isDigit() } -> "The Identifiable"
name.none { it.isLetter() } -> "The Witness Protection Member"
name.count { it.lowercase() in "aeiou" } -> 4 -> "The Master of Vowels"
else -> "The Renowned Hero"
}
}

```

Запустите NyetHack. Вывод должен выглядеть примерно так:

```

Madrigal is The Renowned Hero
Madrigal legally changes their name to Aurelia
Aurelia, The Master of Vowels, heads to the town square
Aurelia enters Taernyl's Folly
...

```

Значение `title` вычисляется при каждом обращении к свойству. Оно не имеет исходного значения или значения по умолчанию, а также поля для хранения значения. Если имя игрока изменилось, значение автоматически обновится, чтобы титул соответствовал имени игрока.

В разделе «Для любознательных: более пристальный взгляд на свойства `var` и `val`» в конце этой главы мы подробнее рассмотрим реализацию свойств `var` и `val` и то, какой байт-код генерируется компилятором при их определении.

В следующей главе вы познакомитесь с другими способами создания экземпляров `Player`. Но прежде чем развивать приложение дальше, уделим немного времени пакетам.

Использование пакетов

Пакет — это нечто вроде папки для похожих объектов, которая обеспечивает логическую группировку файлов в проекте. Например, пакет `kotlin.collections` содержит классы для создания и управления списками и множествами. Пакеты способствуют организации проектов по мере их усложнения, а также предотвращают конфликты имен.

Чтобы создать пакет, щелкните правой кнопкой мыши на каталоге `src/main/kotlin` и выберите команду **New ► Package**. Когда вам будет предложено ввести имя пакета, назовите его `com.bignerdranch.nyethack`. (Вы вольны дать пакету любое имя, но мы предпочитаем стиль «обратного DNS», который хорошо масштабируется с количеством написанных вами приложений.)

Созданный пакет `com.bignerdranch.nyethack` является пакетом верхнего уровня для NyetHack. Включив ваши файлы в пакет верхнего уровня, вы предотвратите любые конфликты имен между типами, которые вы объявили, и типами, объявленными где-то еще, например во внешних библиотеках или в модулях. После добавления новых файлов можно создать и новые пакеты для упорядоченного хранения файлов.

Новый пакет `com.bignerdranch.nyethack` (который напоминает папку) отображается в окне инструментов проекта. Теперь можно переместить весь ваш код Kotlin в новый пакет.

Выделите все исходные файлы `Narrator.kt`, `NyetHack.kt`, `Player.kt` и `Tavern.kt`, используя `Command+щелчок` (`Ctrl+щелчок`). Перетащите файлы в папку `com.bignerdranch.nyethack`. На экране появится диалоговое окно **Move** (рис. 13.2). Убедитесь, что флажок **Update package directive** установлен, и нажмите кнопку **OK**.

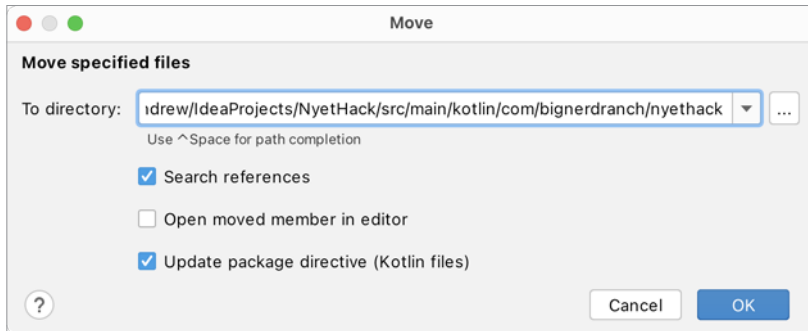


Рис. 13.2. Диалоговое окно Move

После завершения операции все файлы Kotlin попадут в пакет `com.bignerdranch.nyethack`, как показано на рис. 13.3. На диске среда IntelliJ создала папку `com/bignerdranch/nyethack`, которая теперь содержит весь код Kotlin. Хотя в ней существуют три вложенные папки, инструментальное окно проекта IntelliJ упрощает эту иерархию до одной папки для удобства навигации.

При выполнении этого рефакторинга IntelliJ также вставляет в начало каждого файла строку `package com.bignerdranch.nyethack`. Она сообщает Kotlin, какому пакету принадлежит этот файл. И хотя объявление пакета не обязательно соответствовать структуре папок, в которых файл хранится на диске, мы настоятельно рекомендуем следовать этой схеме.

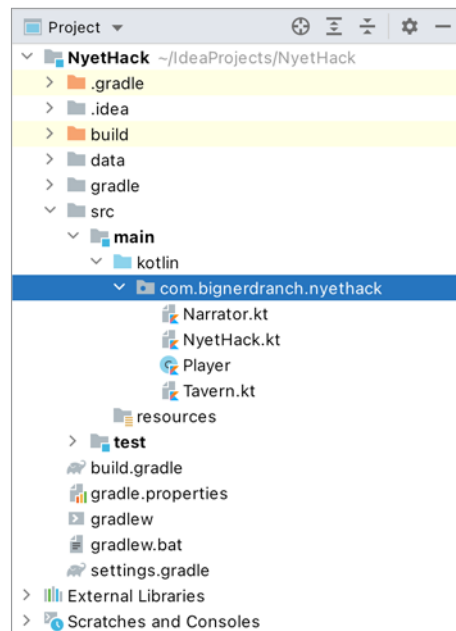


Рис. 13.3.
Пакет `com.bignerdranch.nyethack`

Организация кода с использованием классов, файлов и пакетов поможет сохранить ясность кода по мере его усложнения.

Для любознательных: более пристальный взгляд на свойства `var` и `val`

В этой главе вы узнали, что при определении свойств класса используются ключевые слова `val` и `var`: свойства `val` доступны только для чтения, свойства `var` — для записи.

Чтобы понять, как реализованы свойства классов, полезно взглянуть на декомпилированный байт-код JVM, а конкретнее — сравнить байт-код, сгенерированный для каждого свойства в зависимости от формы определения. Взгляните на приведенный ниже файл `Player.kt`.

```
package com.bignerdranch.nyethack

class Player {

    var name = "madrigal"
    get() = field.replaceFirstChar { it.uppercase() }
    private set(value) {
        field = value.trim()
    }

    val title: String
    get() = when {
        name.all { it.isDigit() } -> "The Identifiable"
        name.none { it.isLetter() } -> "The Witness Protection Member"
        name.count { it.lowercase() in "aeiou" } > 4 ->
            "The Master of Vowels"
        else -> "The Renowned Hero"
    }

    fun castFireball(numFireballs: Int = 2) {
        narrate("A glass of Fireball springs into existence (x$numFireballs)")
    }

    fun changeName(newName: String) {
        narrate("$name legally changes their name to $newName")
        name = newName
    }

}
```

А теперь взгляните на полученный декомпилированный байт-код (откройте декомпилятор командой **Tools ▶ Kotlin ▶ Show Kotlin Bytecode** и щелкните на кнопке **Decompile**). IntelliJ выведет полностью декомпилированную реализацию `Player.kt`. Мы слегка почистили вывод, чтобы подчеркнуть самое важное:

```
...
public final class Player {
    @NotNull
    private String name = "madrigan";

    @NotNull
    public final String getName() {
        // Возвращает значение this.name
    }

    private final void setName(String value) {
        // Присваивает this.name результат value.trim()
    }
    @NotNull

    public final String getTitle() {
        // Вычисляет выражение when для генерирования титулов
    }

    public final void castFireball(int numFireballs) {
        ...
    }

    // $FF: синтетический метод
    public static void castFireball$default(...) {
        ...
    }

    public final void changeName(@NotNull String newName) {
        // Вызывает setName(newName) и выводит сообщение
    }
}
```

Класс содержит (сверху вниз):

- поле для `name`, а также `get`- и `set`-методы (с именами `getName` и `setName`);
- `get`-метод `getTitle` для `title`;
- функцию `castFireball`;
- «синтетическую» функцию для аргумента по умолчанию функции `castFireball`;
- функцию `changeName`.

`name` содержит поле, `get`-метод и `set`-метод (потому что это `var`), тогда как `title` содержит только `get`-метод, потому что это вычисляемое значение `val`.

Теперь попробуйте преобразовать свойство `name` из `var` в `val`. (Закомментируйте код, выделенный в листинге, вместо того, чтобы удалять его, потому что это изменение следует отменить для дальнейшей работы в следующей главе.)

Листинг 13.15. Преобразование var в val (Player.kt)

```

...
class Player {
    var val name = "madrigal"
    get() = field.replaceFirstChar { it.uppercase() }
    /* private set(value) {
        field = value.trim()
    } */
    ...
    fun changeName(newName: String) {
        narrate("$name legally changes their name to $newName")
        // name = newName
    }
}

```

Просмотрите полученный декомпилированный байт-код, снова запустив декомпилятор. (Перечеркнутый фрагмент показывает, что пропало из кода.)

```

public final class Player {
    @NotNull
    private String name = "madrigal";

    @NotNull
    public final String getName() {
        // Возвращает значение this.name
    }

    private final void setName(String value) {
        // Присваивает this.name результат value.trim()
    }
    ...
}

```

Ключевое слово `var` отличается от `val` отсутствием `set`-метода.

Из этой главы вы узнали, что для свойства можно определить нестандартные `get`- и `set`-методы. Пользовательские реализации геттеров и сеттеров появляются непосредственно в функциях `getName`, `setName` и `getTitle`.

Вычисляемые свойства работают аналогичным образом. У свойства `title` есть только `get`-метод. Компилятор смог определить, что поле не требуется, так как оно не упоминается в геттере.

Эта специфическая особенность свойств — вычисление значения вместо чтения состояния поля — еще одна причина, по которой мы используем термин «доступный только для чтения» вместо «неизменяемый». Для примера рассмотрим класс `Dice`:

```

class Dice {
    val rolledValue
    get() = (1..6).random()
}

```


Результат чтения свойства `rolledValue` из `Dice` — это случайное значение в интервале от 1 до 6, определяемое каждый раз при обращении к свойству. Вряд ли это соответствует нашим представлениям о «неизменяемом».

Когда вы завершите анализ байт-кода, верните `Player.kt` в исходное состояние; для этого восстановите ключевое слово `var` для свойства `name` и сеттера.

Листинг 13.16. Восстановление Player (Player.kt)

```
...
class Player {
    val var name = "madrigal"
    get() = field.replaceFirstChar { it.uppercase() }
    /* private set(value) {
        field = value.trim()
    } */
    ...
    fun changeName(newName: String) {
        narrate("$name legally changes their name to $newName")
        // name = newName
    }
}
```

Для любознательных: защита от изменяемости

Если свойство класса одновременно изменяемое и имеет тип, допускающий `null`, необходимо проверить, что оно не равно `null`, прежде чем ссылаться на него. Например, игроки в `NyetHack` могут носить оружие, но оружие игрока также может быть равно `null` (например, игрок или еще не добыл оружия, или разоружен). Следующий код выводит название оружия, если оно есть у игрока:

```
class Weapon(val name: String)

class Player {
    var weapon: Weapon? = Weapon("Mjolnir")

    fun printWeaponName() {
        if (weapon != null) {
            println(weapon.name)
        }
    }
}

fun main() {
    Player().printWeaponName()
}
```

Как ни странно, этот код не компилируется. Чтобы понять почему, посмотрите на ошибку ниже (рис. 13.4).

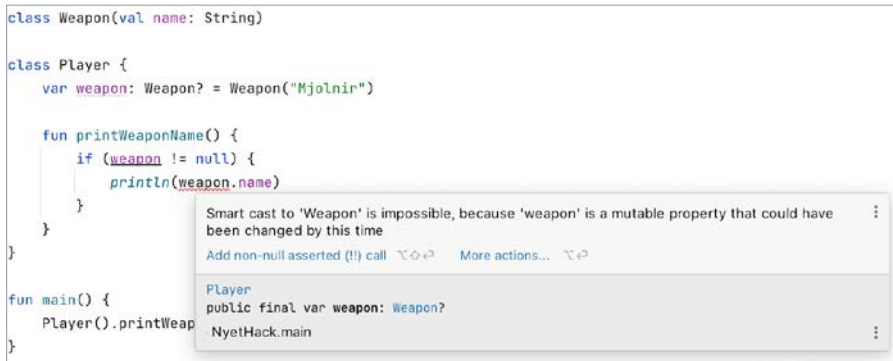


Рис. 13.4. Умное приведение типа к 'Weapon' невозможно

Компилятор прерывает компиляцию кода из-за возможности так называемого *состояния гонки*, о котором мы подробнее расскажем в главе 20. Состояние гонки возникает, если другая часть программы одновременно изменяет состояние вашего кода так, что это может привести к непредвиденным результатам.

В приведенном выше примере компилятор видит, что хотя `weapon` и проверяется на равенство `null`, все равно существует вероятность, что свойство `weapon` класса `Player` получит значение `null` между моментом проверки и моментом вывода названия оружия на экран.

Таким образом, в отличие от случаев, показанных в главе 7, умное приведение типа `weapon` при проверке на `null` невозможно. В предыдущих примерах переменные объявлялись внутри функций. Поскольку в любой момент времени только один вызов этой функции будет обладать доступом к таким переменным, то умное приведение типа возможно. Однако `weapon` — изменяемое свойство класса, поэтому оно может быть изменено во время выполнения функции.

Компилятор протестует, потому что он не может быть уверен, что значение `weapon` отлично от `null`. (Тем не менее умное приведение типа можно использовать со свойствами `val` при условии, что свойство определяется в вашем коде, не является вычисляемым и не может изменяться другими классами.)

Исправить эту проблему и защититься от `null` удастся при помощи функции области видимости, например функции `let`, о которой вы узнали в главе 12:

```
class Player {
    var weapon: Weapon? = Weapon("Mjolnir")

    fun printWeaponName() {
        weapon?.let {
            println(it.name)
        }
    }
}
```

Этот код компилируется благодаря функции области видимости `let`. Вместо ссылки на свойство класса код использует аргумент `it` функции `let`, который в данном случае является локальной переменной, существующей только внутри области видимости анонимной функции. Это означает, что переменная `it` гарантированно не будет изменена другими частями программы.

Нам удалось полностью избежать проблемы с умным приведением типа, так как вместо использования свойства, допускающего `null`, код использует локальную переменную, доступную только для чтения и не допускающую `null` (поскольку `let` вызывается после оператора безопасного вызова `weapon?.let`).

Для любознательных: ограничение видимости рамками пакета

В начале главы мы обсуждали уровни видимости `private` и `public`. Как вы уже знаете, в языке Kotlin классы, функции или свойства по умолчанию (без модификатора видимости) получают публичную видимость. Это означает, что ими могут пользоваться любые другие классы, функции или свойства в проекте.

Если вы имели дело с Java, то могли заметить, что уровень доступа по умолчанию отличается в этих языках: по умолчанию в Java назначается уровень видимости, ограниченный рамками пакета, то есть методы, поля и классы без модификатора видимости могут использоваться только классами из того же пакета.

Видимость, ограниченная рамками пакета, не только не используется в Kotlin по умолчанию — она не поддерживается вообще. В Kotlin отказались от этого подхода, потому что от него мало пользы. На практике такое ограничение легко обойти, создав соответствующий пакет и скопировав в него класс.

С другой стороны, в Kotlin есть то, чего нет в Java, — внутренний уровень видимости `internal`. Функции, классы или свойства с этим уровнем видимости доступны для других функций, классов и свойств внутри того же *модуля*. Модуль — это отдельная функциональная единица, которую можно выполнять, тестировать и отлаживать независимо.

Модули включают исходный код, сценарии сборки, модульные тесты, дескрипторы развертывания и т. д. `NyetHack` (не файл `NyetHack.kt`, а `NyetHack` верхнего уровня) — один модуль внутри вашего проекта, при этом любой проект IntelliJ может вмещать несколько модулей. Модули зависят от исходных файлов и ресурсов других модулей.

Уровень видимости `internal` помогает организовать совместный доступ к классам внутри модуля и сделать их недоступными из других модулей. Поэтому видимость `internal` хорошо подходит для создания библиотек на языке Kotlin.

14. Инициализация

В предыдущей главе вы узнали, как определять классы для представления объектов реального мира. В `NetHack` вы определили свойства и поведение для игрока. Несмотря на всю сложность, которую можно передать через свойства и функции классов, вы пока что видели лишь малую часть способов создания экземпляров класса.

Вспомним, как в прошлой главе объявлялся класс `Player`:

```
class Player {  
    ...  
}
```

Заголовок класса `Player` довольно прост, поэтому создать экземпляр `Player` тоже было просто:

```
val player = Player()
```

Как вы уже знаете, при вызове конструктора класса создается *экземпляр* класса — этот процесс известен также как *инстанцирование*. В этой главе мы рассмотрим различные способы *инициализации* классов и их свойств. Инициализируя переменную, свойство или экземпляр класса, вы присваиваете им начальные значения, что делает их готовыми к использованию. Далее мы познакомим вас с разными видами конструкторов, приемами инициализации свойств и даже научим изменять правила с помощью поздней и отложенной инициализации.

Замечание относительно терминологии: формально экземпляр класса *создается* (*инстанцируется*), когда для него выделяется память, а *инициализируется* — когда ему присваивается значение. Но на практике эти термины обычно употребляются немного иначе. Часто под *инициализацией* подразумевается все необходимое, чтобы подготовить переменную, свойство или экземпляр класса к использованию. В этой книге мы придерживаемся именно такого, более распространенного значения.

Конструкторы

`Player` теперь содержит особенности поведения и данные, которые вы определили. Например, вы указали свойство `name`:

```
var name = "madrigal"  
get() = field.replaceFirstChar { it.uppercaseChar() }
```

```
private set(value) {
    field = value.trim()
}
```

В текущей реализации каждый игрок начинает игру под именем «madrigal» и должен изменить свое имя в дальнейшем, что может быть сопряжено с бюрократической волокитой в NyetHack. Лучше сразу создать экземпляра `Player` с правильным именем.

И тут в дело вступает *главный конструктор*. Конструктор позволяет при его вызове определить начальные значения, необходимые для создания экземпляра класса. Эти значения затем можно использовать для инициализации свойств, объявленных внутри класса.

Главный конструктор

Как и функция, конструктор определяет ожидаемые параметры, для которых должны передаваться аргументы. Чтобы задать все, что необходимо экземпляру `Player` для корректной работы, объявите главный конструктор в заголовке `Player`. Измените код в `Player.kt` и добавьте возможность передачи имени игрока через главный конструктор. Также запросите дополнительную информацию об игроке, которая может пригодиться при построении новой функциональности в NyetHack.

Листинг 14.1. Определение главного конструктора (Player.kt)

```
package com.bignerdranch.nyethack

class Player(
    initialName: String,
    hometown: String,
    healthPoints: Int,
    isImmortal: Boolean
) {
    var name = "madrigal" initialName
    get() = field.replaceFirstChar { it.uppercaseChar() }
    private set(value) {
        field = value.trim()
    }

    val hometown = hometown

    var healthPoints = healthPoints

    val isImmortal = isImmortal

    val title: String
    get() = ...

    ...
}
```

Теперь для того, чтобы создать экземпляр `Player`, передайте аргументы, соответствующие параметрам, добавленным в конструктор. Благодаря этому, например, можно не ограничивать себя жестко заданным значением для свойства `name`, а передавать его как аргумент главному конструктору `Player`.

Обратите внимание: имена параметров конструктора совпадают или почти совпадают с именами свойств объекта. Такая схема выбора имен широко используется, но необязательна. Если вы хотите четко обозначить, к чему относится ссылка — к параметру конструктора или к свойству, присвойте им разные имена.

Если вы хотите убедиться, что обращаетесь к нужному свойству или переменной, подведите текстовый курсор к соответствующему имени и нажмите `Command-B` (`Ctrl-B`), чтобы перейти к его определению. Также можно сделать `Command-щелчок` (`Ctrl-щелчок`) на имени переменной. Если Kotlin читает значение из аргумента конструктора, IntelliJ переведет вас к параметру. Если же значение читается из свойства, вы перейдете к объявлению свойства.

Этот прием также годится для перехода к функциям и определениям класса. Кроме того, он часто упрощает навигацию по проекту и просмотр реализации некоторого класса или функции.

Чтобы использовать новый конструктор, измените вызов конструктора `Player` в `main` и включите в него новую информацию. Также удалите код, изменяющий имя игрока при запуске `NyetHack`, и добавьте информацию об игроке в описание.

(Мы разбили первую строку описания на две, чтобы она поместилась на странице книги; вам следует вводить ее в одной строке.)

Листинг 14.2. Вызов главного конструктора (`NyetHack.kt`)

```
package com.bignerdranch.nyethack

val player = Player("Jason", "Jacksonville", 100, false)

fun main() {
    narrate("${player.name} is ${player.title}")
    player.changeName("Aurelia")
    // changeNarratorMood()
    val mortality = if (player.isImmortal) "an immortal" else "a mortal"
    narrate("${player.name} of ${player.hometown}, ${player.title},
        heads to the town square")
    narrate("${player.name}, $mortality, has ${player.healthPoints} health points")

    visitTavern()
    player.castFireball()
}
...
```

Только подумайте, как много возможностей добавил главный конструктор в `Player`: раньше игрок в `NyetHack` всегда получал имя `Madrigal` и не мог изменять

атрибуты, доступные только для чтения (родной город, статус бессмертия и т. д.). Теперь игрок может выбрать любое имя, город, количество очков здоровья и любой статус бессмертия — никакие данные в классе `Player` не задаются жестко.

Запустите `NyetHack` и убедитесь в том, что вывод не изменился.

```
Jason of Jacksonville, The Renowned Hero, heads to the town square
Jason, a mortal, has 100 health points
Jason enters Taernyl's Folly
...
```

Объявление свойств в главном конструкторе

Обратите внимание на однозначную связь параметров конструктора в `Player` и свойств класса: в программе есть параметр для каждого свойства класса, которое требуется инициализировать при создании игрока.

Для свойств, использующих `get`- и `set`-методы по умолчанию, Kotlin позволяет указать параметр и свойство в одном определении, без создания временной переменной. `name` использует пользовательские `get`- и `set`-методы, поэтому для него такой подход не годится. Тем не менее это возможно для других свойств `Player`.

Измените класс `Player` и объявите `hometown`, `healthPoints` и `isImmortal` свойствами в главном конструкторе `Player`.

Листинг 14.3. Определение свойств в главном конструкторе (`Player.kt`)

```
class Player(
    initialName: String,
    val hometown: String,
    var healthPoints: Int,
    val isImmortal: Boolean
) {
    var name = initialName
        get() = field.replaceFirstChar { it.uppercaseChar() }
        private set(value) {
            field = value.trim()
        }

    val hometown = hometown

    var healthPoints = healthPoints

    val isImmortal = isImmortal
    ...
}
```

Для каждого параметра конструктора вы указываете, будет ли он изменяемым или доступным только для чтения. Определяя параметры в конструкторе с помо-

щью ключевых слов `val` и `var`, вы объявляете соответствующие свойства класса `val` или `var`, а также параметры, для которых конструктор ожидает аргументы. Также каждому свойству неявно присваивается значение, которое передается в качестве аргумента.

Дублирование кода усложняет внесение изменений. Обычно мы предпочитаем именно этот способ объявления свойств класса, потому что он ведет к меньшему дублированию. Этот прием нельзя использовать для свойства `name`, потому что оно имеет нестандартные `get`- и `set`-методы, но в других случаях объявление свойства в главном конструкторе часто оказывается лучшим решением.

Дополнительные конструкторы

Конструкторы бывают двух видов: главные и дополнительные. Конструктор, который мы определяли выше, — главный. Определяя главный конструктор, вы говорите: «Эти параметры обязательны для любого экземпляра этого класса». Также можно задать дополнительный конструктор, который определяет альтернативные способы создания экземпляра, соответствующие требованиям главного конструктора.

Дополнительный конструктор должен вызывать либо главный конструктор, передавая ему все требуемые аргументы, либо другой дополнительный конструктор, который следует тому же правилу. Например, вы знаете, что в большинстве случаев игрок начнет игру со 100 очками здоровья и он смертен. Вы можете объявить дополнительный конструктор, автоматически устанавливающий эти значения. Добавьте дополнительный конструктор в `Player`.

Листинг 14.4. Определение дополнительного конструктора (`Player.kt`)

```
class Player(  
    initialName: String,  
    val hometown: String,  
    var healthPoints: Int,  
    val isImmortal: Boolean  
) {  
    ...  
    val title: String  
        get() = when {  
            ...  
        }  
  
    constructor(name: String, hometown: String) : this(  
        initialName = name,  
        hometown = hometown,  
        healthPoints = 100,  
        isImmortal = false  
    )  
    ...  
}
```


Можно определить несколько дополнительных конструкторов для разных комбинаций параметров. Наш дополнительный конструктор вызывает главный конструктор с полным набором параметров. Ключевое слово `this` в данном случае ссылается на экземпляр класса, для которого объявлен конструктор. Конкретно в этом случае `this` вызывает другой конструктор, определенный внутри класса, — главный конструктор.

Так как дополнительный конструктор предоставляет значения по умолчанию для `healthPoints` и `isImmortal`, вам не придется передавать аргументы для этих параметров при его вызове. Вызовите дополнительный конструктор для `Player` из `NyetHack.kt` вместо главного конструктора.

Листинг 14.5. Вызов дополнительного конструктора (NyetHack.kt)

```
...
val player = Player("Jason", "Jacksonville", 100, false)

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

В дополнительном конструкторе можно определить также логику инициализации — код, выполняемый в момент создания экземпляра класса. Например, добавьте выражение, которое заметно улучшает здоровье игрока, если его зовут `Jason`.

Листинг 14.6. Добавление логики в дополнительный конструктор (Player.kt)

```
class Player(
    initialName: String,
    val hometown: String,
    var healthPoints: Int,
    val isImmortal: Boolean
) {
    ...
    constructor(name: String, hometown: String) : this(
        initialName = name,
        hometown = hometown,
        healthPoints = 100,
        isImmortal = false
    ) {
        if (name.equals("Jason", ignoreCase = true)) {
            healthPoints = 500
        }
    }
    ...
}
```

В дополнительном конструкторе удобно определять альтернативную логику, которая должна выполняться при создании экземпляра. Логика дополнительного конструктора применяется только при его вызове — она не распространяется на другие конструкторы класса. По этой причине дополнительные конструкторы не могут использоваться для определения свойств, как главные конструкторы. Свойства класса определяются только в главном конструкторе или на уровне класса.

Запустите `NyetHack` и убедитесь, что герой смертен и у него много очков здоровья; это доказывает, что дополнительный конструктор `Player` был вызван из `NyetHack.kt`.

Аргументы по умолчанию

При определении конструктора также можно указать значения по умолчанию, которые получают параметры, если их аргументы не были заданы. Вы уже видели аргументы по умолчанию в контексте функций. В случае с главными и дополнительными конструкторами они работают абсолютно так же. Например, установите для `homeTown` значение по умолчанию `"Neversummer"` в главном конструкторе.

Листинг 14.7. Определение аргумента по умолчанию в конструкторе (`Player.kt`)

```
class Player(  
    initialName: String,  
    val hometown: String = "Neversummer",  
    var healthPoints: Int,  
    val isImmortal: Boolean  
) {  
    ...  
    constructor(name: String, hometown: String) : this(  
        initialName = name,  
        hometown = hometown,  
        healthPoints = 100,  
        isImmortal = false  
    ) {  
        if (name.equals("Jason", ignoreCase = true)) {  
            healthPoints = 500  
        }  
    }  
    ...  
}
```

Так как вы удалили аргумент из дополнительного конструктора, вам также придется соответствующим образом обновить параметры игрока.

Листинг 14.8. Использование аргумента по умолчанию (Nyethack.kt)

```
...
val player = Player("Jason", "Jacksonville")

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

Выполните свой код и убедитесь в том, что даже если `Jason` — фанат `Jaguars`, он теперь родом из города `Neversummer`, а не `Jacksonville`¹. С аргументами по умолчанию и дополнительными конструкторами появляются разные способы определения комбинаций аргументов, допустимых для конструктора. Для класса `Player` возможные варианты выглядят так:

```
Player("Jason", "Jacksonville", 40, true)
```

Главный конструктор со всеми заданными параметрами.

```
Player("Madrigal", healthPoints = 40, isImmortal = false)
```

Главный конструктор с родным городом `Neversummer` в аргументе по умолчанию и с остальными заданными параметрами.

```
Player("Estragon")
```

Дополнительный конструктор с заданным именем, родным городом `Neversummer`, 100 очками здоровья и статусом смертного, заданными в аргументах по умолчанию.

Именованные аргументы

Чем больше аргументов по умолчанию, тем больше появляется вариантов для вызова конструктора. Чем больше вариантов, тем больше неоднозначности. Поэтому Kotlin позволяет использовать в вызове конструктора именованные аргументы, аналогичные именованным аргументам в вызовах функций.

В синтаксисе именованных аргументов указывается имя параметра, соответствующее каждому аргументу, что улучшает удобочитаемость. Сравните два варианта создания экземпляра `Player`:

```
val player = Player(
    initialName = "Madrigal",
    hometown = "Neversummer",
    healthPoints = 40,
    isImmortal = true
)

val player = Player("Madrigal", "Neversummer", 40, true)
```

¹ Jacksonville Jaguars — профессиональный футбольный клуб. — *Примеч. ред.*

Какой вариант, по-вашему, понятнее? Если вы выбрали первый, то мы с вами солидарны.

Синтаксис именованных аргументов особенно полезен, когда имеется несколько параметров одного типа. В классе `Player` не возникнет вопросов, то ли игрок с именем `Madison` происходит из города `Austin`, то ли игрок с именем `Austin` происходит из города `Madison` — именованные параметры все объясняют.

Кроме устранения неоднозначности, именованные аргументы дают еще одно преимущество: они позволяют передавать аргументы в функцию или конструктор в произвольном порядке. При передаче неименованных аргументов необходимо тщательно соблюдать порядок, в каком они указаны в конструкторе.

Возможно, вы заметили, что дополнительный конструктор, который был написан для `Player`, использует именованные аргументы, сходные с теми, что мы видели в главе 4:

```
class Player(  
    initialName: String,  
    val hometown: String = "Neversummer",  
    var healthPoints: Int,  
    val isImmortal: Boolean  
) {  
    ...  
    constructor(name: String, hometown: String) : this(  
        initialName = name,  
        healthPoints = 100,  
        isImmortal = false  
    ) { ...  
    }  
    ...  
}
```

Если вам надо передать в конструктор или в функцию несколько аргументов, мы рекомендуем применять именованные аргументы. С ними проще понять, какие аргументы какому параметру передаются.

Блок инициализации

Помимо главного и дополнительных конструкторов, в Kotlin можно указать для класса *блок инициализации*. Он позволяет настроить переменные или значения, а также выполнить их проверку, то есть убедиться, что конструктору передаются допустимые аргументы. Код в блоке инициализации выполняется сразу после создания экземпляра класса.

Например, при создании к игроку предъявляется ряд требований: он должен начать игру хотя бы с одним очком здоровья, а имя не должно быть пустым.

Воспользуйтесь блоком инициализации, обозначенным ключевым словом `init`, для проверки этих требований.

Листинг 14.9. Определение блока инициализации (Player.kt)

```
class Player(  
    initialName: String,  
    val hometown: String = "Neversummer",  
    var healthPoints: Int,  
    val isImmortal: Boolean  
) {  
    ...  
    val title: String  
        get() = when {  
            ...  
        }  
  
    init {  
        require(healthPoints > 0) { "healthPoints must be greater than zero" }  
        require(name.isNotBlank()) { "Player must have a name" }  
    }  
  
    constructor(name: String) : this(  
        initialName = name,  
        healthPoints = 100,  
        isImmortal = false  
    ) { ...  
    }  
    ...  
}
```

Код в блоке инициализации вызывается при создании экземпляра класса, какой бы конструктор при этом ни был вызван, главный или дополнительный. Если хотя бы одно из условий не выполнится, выдается исключение `IllegalArgumentException` (вы можете проверить это в Kotlin REPL, передав `Player` другие параметры).

Эти требования сложно инкапсулировать в конструкторе или в объявлении свойства. При желании можно разместить присваивание значений свойствам в блоках инициализации. Обычно это делается только тогда, когда исходное значение не может быть вычислено в одном выражении, и такой инструмент неплохо иметь в запасе.

Блоки инициализации также используют для присваивания значений свойствам, если вы хотите отделить объявление от присваивания. Данная возможность особенно полезна, если вам нужно определить сложную логику с несколькими командами для вычисления исходного значения свойства. Например, для вычисления исходного состояния снаряжения игрока можно использовать блок следующего вида:

```
class Player {  
    ...
```

```

val inventory: List<String>

init {
    val baseInventory = listOf("waterskin", "torches")
    val classInventory = when (playerClass) {
        "archer" -> listOf("arrows")
        "wizard" -> listOf("arcane staff", "spellbook")
        "rogue" -> listOf("lockpicks", "crowbar")
        else -> emptyList()
    }
    inventory = baseInventory + classInventory
}
}

```

Порядок инициализации

Вы узнали, как инициализировать свойства и добавлять логику инициализации разными способами: объявлять параметры в главном конструкторе, инициализировать при объявлении, в дополнительном конструкторе или в блоке инициализации. Одно и то же свойство может использоваться в нескольких инициализациях, поэтому порядок их выполнения очень важен.

Рассмотрим следующий класс `Villager`, который представляет жителей города Кронштадта:

```

class Villager(val name: String, val hometown: String) {
    val personality: String
    val race = "Dwarf"
    var age = 50
    private set
    init {
        println("initializing villager")
        personality = "Outgoing"
    }
    constructor(name: String) : this(name, "Bavaria") {
        age = 99
    }
}

```

Допустим, вы конструируете экземпляр дополнительным конструктором класса, для чего используется вызов `Villager("Estragon")`. В каком порядке выполняются эти выражения?

Чтобы получить ответ на этот вопрос, полезно проанализировать порядок инициализации полей и вызовов методов в декомпилированном байт-коде Java. На рис. 14.1 класс `Villager` изображен слева. Сокращенный декомпилированный байт-код Java справа показывает итоговый порядок инициализации.

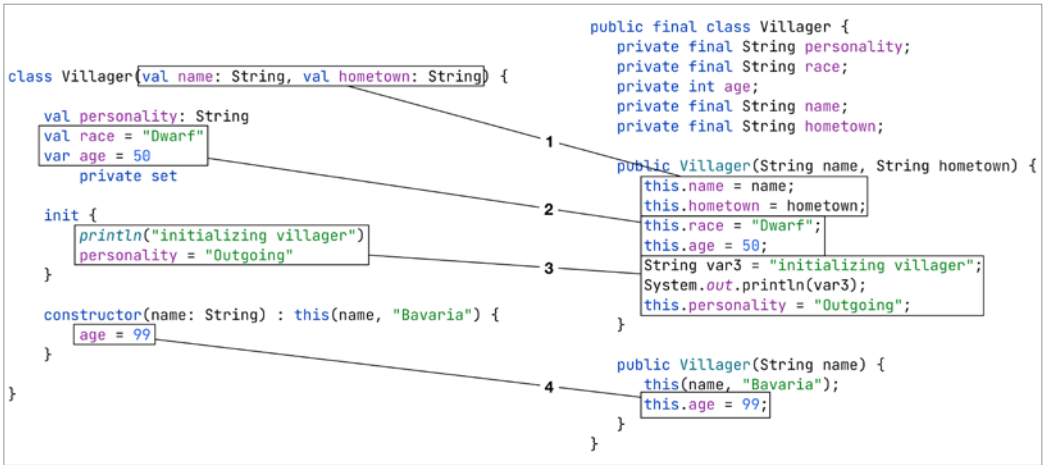


Рис. 14.1. Порядок инициализации класса Villager (декомпилированный байт-код)

Итак, инициализация выполняется в следующем порядке.

- Свойства, объявленные в главном конструкторе (1: name и hometown).
- Блоки init и присваивания значений свойствам в порядке их объявления (2: свойства race и age; 3: вызов println и присваивание personality).
- Инициализаторы дополнительных конструкторов (4: присваивание age = 99).

(Порядок инициализации блоков init и присваивания значений свойствам зависит от порядка, в котором они указаны. Если бы блок init был указан до появления race и age, то он выполнялся бы до присваивания значений этим свойствам.)

Задержка инициализации

Независимо от способа объявления, свойство класса должно инициализироваться в момент создания экземпляра класса. Это правило является важной частью системы защиты от null в Kotlin и гарантирует инициализацию действительными значениями всех свойств, не поддерживающих null, при вызове конструктора класса. После создания объекта можно сразу же сослаться на любое его свойство внутри или вне класса.

Несмотря на его важность, это правило можно обойти. Зачем? Дело в том, что вы не всегда контролируете, как и когда происходит вызов конструктора. Это достаточно часто происходит с фреймворками и библиотеками, такими как Android, Spring и JUnit.

Поздняя инициализация

Иногда вы оказываетесь в ситуации, когда свойство не может быть инициализировано при инициализации объекта, потому что его значение должно быть получено в будущем. Это может быть связано с вашим фреймворком (например, Android-приложения содержат компоненты, которые инициализируются при вызове функции `onCreate`, а не в конструкторе) или же с архитектурой вашего приложения.

Традиционно проблема решалась так: свойству присваивалось значение `null` для инициализации, а затем, когда это становилось возможным, свойство получало реальное значение. Такое решение работает, но у него есть неприятная особенность: при каждом обращении к свойству приходится проверять, не равно ли оно `null`.

Иногда значение, допускающее `null`, имеет смысл. Но в других ситуациях свойство после инициализации всегда имеет значение — и требования `null`-безопасности Kotlin быстро начинают раздражать.

Именно в таких случаях важное значение приобретает *поздняя инициализация*, которая позволяет обойти правила инициализации в Kotlin.

В любое объявление `var`-свойства можно добавить ключевое слово `lateinit`. С ним компилятор Kotlin знает, что при инициализации класса инициализацию свойства можно пропустить.

```
class Arena {
    var isArenaOpen = false
    lateinit var opponentName: String

    fun prepareArena() {
        isArenaOpen = true
        opponentName = getWillingCombatants().random()
    }

    private fun getWillingCombatants() =
        listOf("Cornelius", "Cheryl", "Ralph", "Deborah")
}
```

Это полезный инструмент, но применять его следует с осторожностью. Не все переменные заслуживают пометки `lateinit`. Например, свойства класса `Player`, скорее всего, объявлять с `lateinit` не стоит, потому что вся информация об игроке может быть получена до создания экземпляра класса.

Тем не менее иногда поздней инициализации не избежать. Взгляните на переменную `player` в `NyetHack.kt`:

```
val player = Player("Jason")

fun main() {
    // changeNarratorMood()
```



```

        val mortality = if (player.isImmortal) "an immortal" else "a mortal"
        narrate("${player.name} of ${player.hometown}, ${player.title},
            heads to the town square")
        narrate("${player.name}, $mortality, has ${player.healthPoints}
            health points")

        visitTavern()
        player.castFireball()
    }
    ...

```

Ключевое слово `lateinit` также можно использовать со свойствами верхнего уровня, и `player` оказывается хорошим кандидатом для поздней инициализации. Но почему мы рекомендуем так поступить? Разве экземпляра `player` не должен создаваться сразу же после запуска `NyetHack`?

Помните логику запроса имени у пользователя в `promptHeroName`? Мы не использовали эту функцию с момента введения класса `Player`, но в конце концов это придется сделать.

Значение `name` должно запрашиваться до создания `player`, что создает серьезное затруднение: как инициализировать `player` в `main`, но сохранить значение как свойство верхнего уровня? Именно здесь поздняя инициализация по-настоящему проявляет себя.

Пометьте `player` ключевыми словами `lateinit` `var` и восстановите вызов функции, чтобы запросить имя у игрока. (Но оставьте подробности `promptHeroName` закомментированными, потому что нам еще предстоит многое сделать в `NyetHack` и не хотелось бы вводить имя при каждом тестировании кода.)

Листинг 14.10. Использование `lateinit` (`NyetHack.kt`)

```

val player = Player("Jason")
lateinit var player: Player

fun main() {
    narrate("Welcome to NyetHack!")
    val playerName = promptHeroName()
    player = Player(playerName)
    // changeNarratorMood()
    val mortality = if (player.isImmortal) "an immortal" else "a mortal"
    narrate("${player.name} of ${player.hometown}, ${player.title},
        heads to the town square")
    narrate("${player.name}, $mortality, has ${player.healthPoints} health points")
    visitTavern()
    player.castFireball()
}

private fun promptHeroName(): String {
    narrate("A hero enters the town of Kronstadt. What is their name?") { message ->
        // Prints the message in yellow
    }
}

```

```

        "\u001b[33;1m$message\u001b[0m"
    }

    /*val input = readLine()
    require(input != null && input.isNotEmpty()) {
        "The hero must have a name."
    }

    return input*/
    println("Madrigal")
    return "Madrigal"
}

```

Запустите NyetHack и убедитесь в том, что новая логика инициализации работает так, как предполагалось:

```

Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Madrigal
Madrigal of Neversummer, The Renowned Hero, heads to the town square
...

```

Используя `lateinit`, вы по сути сообщаете компилятору: «Я обязуюсь присвоить значение этой переменной до первой попытки обратиться к ней». Если переменная `lateinit` будет инициализирована до первого обращения, проблем нет. Чтобы увидеть, что произойдет при нарушении этой договоренности, попробуйте обратиться к `player` до присваивания.

Листинг 14.11. Нарушение обязательств перед компилятором (NyetHack.kt)

```

lateinit var player: Player

fun main() {
    narrate("Welcome to NyetHack, ${player.name}!")
    val playerName = promptHeroName()
    player = Player(playerName)
    ...
}
...

```

Компилятор не протестует против этого кода. Но при выполнении NyetHack программа аварийно завершается с исключением `UninitializedPropertyAccessException`, и вы увидите следующий вывод:

```

Exception in thread "main" kotlin.UninitializedPropertyAccessException:
lateinit
property player has not been initialized
    at com.bignerdranch.nyethack.NyetHackKt.getPlayer(NyetHack.kt:3)
    at com.bignerdranch.nyethack.NyetHackKt.main(NyetHack.kt:6)
    at com.bignerdranch.nyethack.NyetHackKt.main(NyetHack.kt)

```

В любой момент, когда вы попытаетесь прочитать свойство `lateinit` перед присваиванием, в программе происходит такая ошибка. При необходимости можно проверить, была ли инициализирована переменная с поздней инициализацией:

```
lateinit var player: Player

fun main() {
    if (::player.isInitialized) {
        narrate("Welcome to NyetHack, ${player.name}!")
    }
    val playerName = promptHeroName()
    player = Player(playerName)
    ...
}
```

Вы можете вызывать `isInitialized` каждый раз, когда есть сомнения, что переменная `lateinit` была инициализирована, чтобы избежать исключения `UninitializedPropertyAccessException`. Тем не менее `isInitialized` следует использовать экономно, например, не следует добавлять эту проверку к каждой переменной с поздней инициализацией. Если вы используете `isInitialized` слишком часто, скорее всего, это означает, что лучше задействовать тип, допускающий `null`.

Чтобы исправить ошибку, отмените последнее изменение, прежде чем идти дальше.

Листинг 14.12. Соблюдение обязательств перед компилятором (NyetHack.kt)

```
lateinit var player: Player

fun main() {
    narrate("Welcome to NyetHack, ${player.name}!")
    val playerName = promptHeroName()
    player = Player(playerName)
    ...
}
...
```

Хотя ключевое слово `lateinit` иногда необходимо, у него есть свои ограничения. `lateinit` может использоваться только со свойствами `var`, потому что невозможно гарантировать, что значение свойства может быть задано только один раз, и для `var`-свойств с `lateinit` нельзя определять нестандартные методы чтения и записи. `lateinit` также не применяется, если свойство относится к типу `Boolean`, `Char` или любому числовому типу (включая `Int`, `Double` и `UInt`). Во внутренней реализации свойства `var` с `lateinit` реализуются значения `null`, и Kotlin не позволяет присваивать этим примитивным типам `null` во время выполнения.

Отложенная инициализация

Поздняя инициализация — не единственный способ перенести инициализацию на будущее. Инициализацию переменной также можно отложить до первого обращения к ней. Эта концепция называется *отложенной (lazy) инициализацией*, и несмотря на название¹, она способна значительно повысить эффективность кода.

Большинство свойств, которые мы инициализировали в этой главе, были довольно простыми — одиночные объекты (такие, как `String`), которые вычисляются почти мгновенно. Но некоторые свойства содержат более сложные значения. Они могут требовать создания нескольких объектов или выполнения сложных вычислений при инициализации (например, читать данные из файла).

Если инициализация вашего свойства требует выполнения подобных вычислений или класс не требует немедленной готовности свойства, отложенная инициализация станет хорошим решением.

Предположим, игрок может получить пророчество о подвигах, которые ему предстоит совершить. Качественное пророчество способна дать лишь хорошая гадалка, а найти ее непросто. Необязательно сообщать пророчество каждому игроку, потому что оно сбудется независимо от того, знает игрок о нем или нет. Кроме того, пророчества неотвратимы: если игрок получил пророчество, изменить свою судьбу он уже не сможет.

Отложенная инициализация реализует именно эту схему. Если пророчество вычисляется по отложенному принципу, игрок не получит его до того момента, когда оно ему понадобится. А после того, как игрок получил пророчество, он запомнит его и сможет моментально вернуть по следующему запросу.

Чтобы увидеть эти концепции в действии, добавьте в `Player` новое свойство `prophecy`, а также новую функцию с именем `propheSize`.

Листинг 14.13. Отложенное получение свойств (Player.kt)

```
class Player(  
    initialName: String,  
    val hometown: String = "Neversummer",  
    var healthPoints: Int,  
    val isImmortal: Boolean  
) {  
    ...  
    val title: String  
        get() = when {  
            ...  
        }  
  
    val prophecy by lazy {
```

¹ lazy — ленивый (англ.) — *Примеч. перев.*

```

narrate("$name embarks on an arduous quest to locate a fortune teller")
Thread.sleep(3000)
narrate("The fortune teller bestows a prophecy upon $name")
"An intrepid hero from $hometown shall some day " + listOf(
    "form an unlikely bond between two warring factions",
    "take possession of an otherworldly blade",
    "bring the gift of creation back to the world",
    "best the world-eater"
).random()
}
...
fun changeName(newName: String) {
    narrate("$name legally changes their name to $newName")
    name = newName
}

fun prophesize() {
    narrate("$name thinks about their future")
    narrate("A fortune teller told Madrigal, \"$prophecy\"")
}
}

```

Взгляните на новый синтаксис `by lazy`. Ключевое слово `by` означает, что свойство реализуется с использованием делегата свойства (подробности — в разделе «Для любознательных: делегаты свойств» в конце главы). Функция `lazy` — делегат, который определяет поведение отложенной инициализации.

Свойство `prophecy` остается неинициализированным до первого обращения к нему. В этот момент выполняется весь код в лямбда-функции `lazy`. Очень важно, что этот код выполняется только один раз — при первом обращении к делегируемому свойству (в данном случае `prophecy`) из `prophesize`. Будущие обращения к отложенному свойству будут использовать кэшированный результат (строку, возвращенную лямбда-функцией) вместо повторного выполнения затратных вычислений.

Внутри лямбда-выражения содержится вызов `Thread.sleep(3000)`. Эта функция заставляет ваш код приостановиться на 3000 миллисекунд (то есть на 3 секунды). И хотя добавлять для `Thread` команду `import` не нужно, следует помнить, что `Thread` является классом `Java`.

Чтобы увидеть отложенную инициализацию в действии, вставьте пару вызовов `prophesize` в `main`.

Листинг 14.14. Использование отложенного свойства (NyetHack.kt)

```

...
fun main() {
    narrate("Welcome to NyetHack!")
    val playerName = promptHeroName()
    player = Player(playerName)
}

```

```

// changeNarratorMood()
player.propheSize()
val mortality = if (player.isImmortal) "an immortal" else "a mortal"
narrate("${player.name} of ${player.hometown}, ${player.title},
        heads to the town square")
narrate("${player.name}, $mortality, has ${player.healthPoints} health points")

visitTavern()
player.castFireball()
player.propheSize()
}
...

```

Запустите NyetHack и посмотрите на вывод. Он будет выглядеть примерно так (но обратите внимание, что программа приостановится на 3 секунды только один раз, хотя чтение из `prophecy` выполняется дважды).

```

Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Madrigal
Madrigal thinks about their future
Madrigal embarks on an arduous quest to locate a fortune teller
The fortune teller bestows a prophecy upon Madrigal
A fortune teller told Madrigal, "An intrepid hero from Neversummer shall some
    day take possession of an otherworldly blade"
Madrigal of Neversummer, The Renowned Hero, heads to the town square
...
A glass of Fireball springs into existence (x2)
Madrigal thinks about their future
A fortune teller told Madrigal, "An intrepid hero from Neversummer shall some
    day take possession of an otherworldly blade"

```

Отложенная инициализация полезна, но она сопряжена с дополнительными затратами (в отношении как производительности, так и объема кода), поэтому применяйте ее только тогда, когда требуются сложные вычисления. Если свойства вычисляются тривиально, избыточное применение отложенной инициализации может ухудшить производительность. Но для свойств, инициализация которых сопряжена с высокими затратами, применение отложенной инициализации дает отличную возможность отложить работу до того момента, когда она станет необходимой, что может повысить скорость отклика программы.

Итак, вы узнали все об инициализации объектов в Kotlin. Чаще всего вы будете использовать самый простой способ: вызов конструктора и получение ссылки на экземпляр класса для дальнейшей работы. Но у вас есть и другие варианты инициализации, и понимание этих вариантов поможет вам писать более красивый и эффективный код.

В следующей главе вы познакомитесь с наследованием — принципом объектно-ориентированного программирования, который позволяет родственным классам совместно использовать данные и поведение.

Для любознательных: подводные камни инициализации

Ранее в главе мы показали, что при использовании блоков инициализации очень важен порядок — вы должны принять меры, чтобы все свойства, используемые в блоке, были инициализированы раньше объявления блока инициализации. Следующий пример демонстрирует проблему упорядочения блоков инициализации.

```
class Player() {
    init {
        val healthBonus = health.times(3)
    }

    val health = 100
}

fun main() {
    Player()
}
```

Этот код не компилируется, потому что свойство `health` не инициализировано перед его использованием в блоке `init`. Как мы отмечали ранее, когда свойство используется внутри блока `init`, инициализация свойства должна произойти до обращения к нему. Если объявить `health` перед блоком инициализации, код скомпилируется:

```
class Player() {
    val health = 100

    init {
        val healthBonus = health.times(3)
    }
}

fun main() {
    Player()
}
```

Есть пара похожих, но более коварных сценариев развития событий, которые могут преподнести неприятный сюрприз неопытному программисту. Например, в следующем коде объявляется свойство `name`, а затем функция `firstLetter` читает первый символ свойства:

```
class Player() {
```

```
val name: String

private fun firstLetter() = name[0]
init {
    println(firstLetter())
    name = "Madrigal"
}

fun main() {
    Player()
}
```

Этот код компилируется, потому что компилятор видит, что свойство `name` инициализируется в блоке `init` — подходящем месте для присваивания начального значения.

Но при выполнении этого кода вы получите ошибку времени выполнения (исключение `NullPointerException` на JVM, `TypeError` в JS, ошибку сегментации в нативном приложении), потому что функция `firstLetter` (которая использует свойство `name`) вызывается раньше, чем свойство `name` получит начальное значение в блоке `init`.

Компилятор не проверяет порядок инициализации свойств и вызовов функций, которые их используют в блоке `init`. Прежде чем определять блок `init`, который вызывает функции, обращающиеся к свойствам, убедитесь, что свойства были инициализированы до вызовов функций. Если `name` инициализировать до вызова `firstLetter`, код компилируется и выполняется без ошибки:

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

    private fun firstLetter() = name[0]

    init {
        name = "Madrigal"
        println(firstLetter())
    }
}

fun main() {
    Player()
}
```

Еще один нетривиальный сценарий мы показываем в следующем примере, где инициализируются два свойства:

```
class Player(name: String) {
    val playerName: String = initPlayerName()
    val name: String = name
```



```
        private fun initPlayerName() = name
    }

    fun main() {
        println(Player("Madrigal").playerName)
    }
```

И снова код компилируется, так как компилятор видит, что все свойства инициализируются. Но при выполнении выводится имя `null` — совсем не то, чего вы ожидали.

В чем проблема? Когда `playerName` инициализируется функцией `initPlayerName`, компилятор предполагает, что свойство `name` уже инициализировано, но при вызове `initPlayerName` выясняется, что это не так.

В этом случае снова важен порядок инициализации. Инициализацию двух свойств надо поменять местами. Если это сделать, класс `Player` компилируется, а при обращении к свойству `name` возвращается действительное значение:

```
class Player(name: String) {
    val name: String = name
    val playerName: String = initPlayerName()

    private fun initPlayerName() = name
}

fun main() {
    println(Player("Madrigal").playerName)
}
```

Для любознательных: делегаты свойств

Отложенная инициализация реализуется в Kotlin с использованием механизма, называемого *делегированием*. Делегаты определяют шаблоны поведения свойств.

Использование делегата объявляется ключевым словом `by`. Стандартная библиотека Kotlin включает ряд уже реализованных делегатов: `lazy`, `observable`, `vetoable` и `notNull`.

Используя ключевое слово `by`, вы приказываете Kotlin использовать реализацию `get` и `set`, предоставленную используемым делегатом. Также это означает, что при использовании делегата свойства невозможно определить пользовательский геттер или сеттер.

На практике `lazy` безусловно применяется намного чаще других делегатов. Другие встроенные делегаты редко встречаются в кодовых базах реальных продуктов, хотя некоторые библиотеки для Kotlin, такие как Kotlin и Jetpack Compose, определяют собственные делегаты, которые вам тоже могут пригодиться.

Также можно подумать об определении собственных делегатов, если вам часто приходится писать в своем коде одинаковые пользовательские геттеры и сеттеры. Для этого обратите внимание на интерфейсы `ReadOnlyProperty` (если вы хотите написать делегат для свойств `val`) и `ReadWriteProperty` (если вы хотите написать делегат для свойств `var` и `val`). Вам придется реализовать один из этих интерфейсов. О том, как это делается, мы расскажем в главе 17.

Задание: загадка Экскалибура

В главе 13 вы узнали, что для свойства можно определить свои методы чтения и записи. Теперь, когда вы знаете, как инициализируются свойства и их классы, мы приготовили вам загадку.

У каждого великого меча есть имя. Объявите класс с именем `Sword` в Kotlin REPL, который подтверждает это.

Листинг 14.15. Определение класса `Sword` (REPL)

```
class Sword(name: String) {
    var name = name
    get() = "The Legendary $field"
    set(value) {
        field = value.lowercase().reversed().capitalize()
    }
}
```

Что выведет следующий код, который создает экземпляр `Sword` и обращается к свойству `name`? (Попробуйте ответить до того, как проверите в REPL.)

Листинг 14.16. Обращение к `name` (REPL)

```
val sword = Sword("Excalibur")
println(sword.name)
```

Что выведет следующий код после изменения `name`?

Листинг 14.17. Присваивание `name` (REPL)

```
sword.name = "Gleipnir"
println(sword.name)
```

Наконец, добавьте в `Sword` блок инициализации, который инициализирует `name`.

Листинг 14.18. Добавление блока инициализации (REPL)

```
class Sword(name: String) {
    var name = name
```

```
    get() = "The Legendary $field"
    set(value) {
        field = value.lowercase().reversed().capitalize()
    }

    init {
        this.name = name
    }
}
```

Что будет выведено теперь, после создания экземпляра `Sword` и обращения к `name`?

Листинг 14.19. Повторное обращение к `name` (REPL)

```
val sword = Sword("Excalibur")
println(sword.name)
```

Это задание проверяет ваши знания об инициализации и пользовательских методах чтения и записи свойств.

15. Наследование

Наследование — принцип объектно-ориентированного программирования, используемый для определения иерархических отношений между типами. В этой главе мы при помощи наследования организуем совместное использование данных и поведения родственными классами.

Чтобы получить представление о наследовании, рассмотрим пример, не имеющий прямого отношения к программированию. У легковых и грузовых автомобилей много общего: колеса, двигатель и т. д. Но есть и различия. Использование наследования позволяет объединить одинаковые черты в общий класс **Venicle** (автомобиль), что даст возможность не реализовывать повторно **Wheel** (колеса) и **Engine** (двигатель) для легковых и грузовых автомобилей. Легковые и грузовые автомобили унаследуют эти общие признаки, а дальше для каждого из них будут определяться уникальные признаки.

В NyetHack мы применим наследование для создания ряда комнат, по которым будет перемещаться игрок.

Объявление класса Room

Начнем с создания в пакете `com.bignerdranch.nyethack` нового файла с именем `Room.kt`. В `Player.kt` вы создали пустой файл и добавили объявление класса. На этот раз IntelliJ сделает объявление за вас. Для этого при создании следует выбрать тип файла **Class**:

```
class Room {  
}
```

В принципе, для создания файлов классов вы можете использовать любой способ.

Теперь `Room.kt` содержит новый класс с именем `Room`, это один квадрат в плоскости координат NyetHack. Позже вы объявите конкретный вид комнаты в классе, который наследует характеристики `Room`.

Класс `Room` имеет одно свойство `name` и две функции, `description` и `enterRoom`. Функция `description` возвращает строку с описанием комнаты (пока описание состоит просто из названия комнаты), `enterRoom` определяет поведение комнаты и выводит на консоль описание того, что видит или испытывает герой внутри комнаты. Такими чертами должны обладать все комнаты в NyetHack.

Добавьте определение класса `Room` в `Room.kt`.

Листинг 15.1. Объявление класса Room (Room.kt)

```
class Room(val name: String) {
    fun description() = name
    fun enterRoom() {
        narrate("There is nothing to do here")
    }
}
```

В файле NyetHack.kt протестируйте новый класс Room; для этого создайте экземпляр Room при запуске игры в main и выведите результат функции description. Заодно удалите вызов visitTavern — мы переработаем код таверны позднее в этой главе для использования нового класса Room.

Листинг 15.2. Вывод описания комнаты (NyetHack.kt)

```
...
fun main() {
    narrate("Welcome to NyetHack!")
    val playerName = promptHeroName()
    player = Player(playerName)
    // changeNarratorMood()
    player.prophesize()

    var currentRoom = Room("The Foyer")
    val mortality = if (player.isImmortal) "an immortal" else "a mortal"
    narrate("${player.name} of ${player.hometown}, ${player.title},
        heads to the town square
        is in ${currentRoom.description()}")
    narrate("${player.name}, $mortality, has ${player.healthPoints} health points")
    currentRoom.enterRoom()

    visitTavern()
    player.castFireball()
    player.prophesize()
}
...
```

Запустите NyetHack. На консоли появится следующий вывод:

```

Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Madrigal
Madrigal thinks about their future
Madrigal embarks on an arduous quest to locate a fortune teller
The fortune teller bestows a prophecy upon Madrigal
A fortune teller told Madrigal, "An intrepid hero from Neversummer shall
some
    day bring the gift of creation back to the world"
Madrigal of Neversummer, The Renowned Hero, is in The Foyer
Madrigal, a mortal, has 100 health points
There is nothing to do here
```

```

A glass of Fireball springs into existence (x2)
Madrigal thinks about their future
A fortune teller told Madrigal, "An intrepid hero from Neversummer shall
some
    day bring the gift of creation back to the world"

```

Пока неплохо... Но скучно. Кто захочет проводить время в прихожей (foyer)? Настало время приключений! Мадригалу из Неверсаммера пора отправляться в другие места.

Создание подкласса

Подкласс обладает всеми чертами наследуемого класса, который также называют *родительским классом*, или *суперклассом*.

Например, жителям NyetHack не помешает городская площадь. Городская площадь — это разновидность комнаты `Room` со своими особенностями, характерными только для площадей (например, при входе игрока выводится приветственное сообщение). Класс `TownSquare` можно объявить дочерним по отношению к `Room`, так как у них много общего, а затем описать, чем `TownSquare` отличается от `Room`.

Но прежде чем объявлять класс `TownSquare`, надо изменить класс `Room`, чтобы ему можно было наследовать.

Не каждый класс, который вы напишете, сможет стать частью иерархии. Более того, по умолчанию классы закрыты для расширения, то есть наследование для них запрещено. Чтобы от класса можно было наследовать, его надо отметить ключевым словом `open`.

Добавьте ключевое слово `open` в `Room`, чтобы можно было создавать подклассы.

Листинг 15.3. Разрешение наследования от класса `Room` (`Room.kt`)

```

open class Room(val name: String) {

    fun description() = name

    fun enterRoom() {
        narrate("There is nothing to do here")
    }
}

```

Теперь, когда класс `Room` отмечен как доступный для наследования, создайте класс `TownSquare` в `Room.kt`, объявив его подклассом `Room`. Для этого используйте оператор `:`, как в следующем примере.

Листинг 15.4. Объявление класса `TownSquare` (`TownSquare.kt`)

```

class TownSquare : Room("The Town Square")

```

Объявление класса `TownSquare` содержит имя класса слева от оператора `:` и вызов конструктора справа. Вызов конструктора указывает на то, какой родительский конструктор нужно вызвать для создания экземпляра `TownSquare` и какие аргументы ему передать. В данном случае `TownSquare` — это разновидность `Room` с названием `"Town Square"`.

Но мы хотим, чтобы у городской площади было не только название. Другой способ добавить отличия подкласса от предка — это *переопределение* (overriding). В главе 13 мы говорили, что свойства представляют данные класса, а функции — его поведение. Подклассы могут переопределять (определять собственные реализации) для данных и функций.

`Room` имеет две функции — `description` и `enterRoom`. В `TownSquare` должна присутствовать своя реализация `enterRoom` для выражения народом радости, когда герой выходит на городскую площадь.

Переопределите `enterRoom` в `TownSquare`, используя ключевое слово `override`.

Листинг 15.5. Переопределение `enterRoom` (`TownSquare.kt`)

```
class TownSquare : Room("The Town Square") {
    override fun enterRoom() {
        narrate("The villagers rally and cheer as the hero enters")
    }
}
```

Когда вы будете переопределять `enterRoom`, IntelliJ пожалуется на ключевое слово `override` (рис. 15.1).

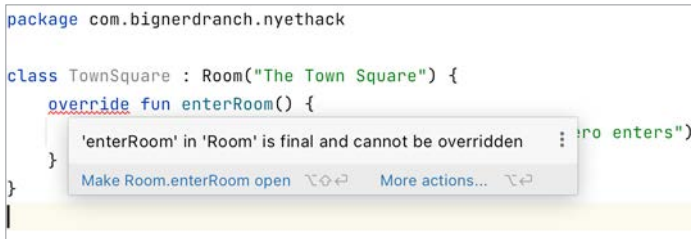


Рис. 15.1. `enterRoom` не может переопределяться

Среда IntelliJ, как всегда, права: есть проблема. Ключевым словом `open` нужно пометить не только класс `Room`, но также функцию `enterRoom`, чтобы ее можно было переопределить.

Сделайте функцию `enterRoom` в классе `Room` доступной для переопределения.

Листинг 15.6. Разрешение переопределения для функции `enterRoom` (`Room.kt`)

```
open class Room(val name: String) {
```

```
fun description() = name

open fun enterRoom() {
    narrate("There is nothing to do here")
}
}
```

Теперь вместо сообщения по умолчанию (*There is nothing to do here*) экземпляр `TownSquare` выведет на экран ликование жителей, когда герой появляется на площади. Это реализуется вызовом `enterRoom`.

В главе 12 вы научились управлять видимостью свойств и функций, используя модификаторы видимости. Свойства и функции по умолчанию общедоступны. Вы также можете сделать их доступными только внутри класса, в котором они объявлены, установив уровень видимости `private`.

Модификатор доступа `protected` — это третий вариант, ограничивающий уровень видимости члена класса самим классом *и* любыми его подклассами.

Добавьте свойство с модификатором `protected` и именем `status` в `Room`.

Листинг 15.7. Объявление свойства `protected` (`Room.kt`)

```
open class Room(val name: String) {

    protected open val status = "Calm"

    fun description() = "$name (Currently: $status)"

    open fun enterRoom() {
        narrate("There is nothing to do here")
    }
}
```

`status` определяет общее состояние комнаты (например, представляет опасность, захламлена, вызывает ужас или еще что-то). Оно выводится в описании, чтобы игрок знал, чего ожидать при входе в комнату. В реализации `Room` по умолчанию не происходит ничего интересного, отсюда и значение по умолчанию «*There is nothing to do here*» (Здесь нечего делать).

Подклассы `Room` могут изменять `status`, чтобы выразить, насколько опасна (или не очень) конкретная комната, но в целом свойство `status` должно быть инкапсулировано в `Room` и его подклассах. Это идеальный случай для использования ключевого слова `protected`: свойство должно быть доступно только классу, в котором оно объявлено, и его подклассам.

Чтобы переопределить свойство `status` в `TownSquare`, нужно использовать ключевое слово `override`, как в случае с функцией `enterRoom`.

Листинг 15.8. Переопределение status (TownSquare.kt)

```
class TownSquare : Room("The Town Square") {  
    override val status = "Bustling"  
  
    override fun enterRoom() {  
        narrate("The villagers rally and cheer as the hero enters")  
    }  
}
```

Подклассы способны не только переопределять свойства и функции суперклассов, но и объявлять свои.

Городская площадь NyetHack, например, отличается от других комнат наличием башни с колоколом, возвещающим о важных событиях. Добавьте в TownSquare функцию private с именем ringBell и переменную private с именем bellSound. bellSound содержит строку, которая описывает удар в колокол, а функция ringBell, вызываемая в enterRoom, возвращает строку, которая объявляет о появлении героя на городской площади.

Листинг 15.9. Добавление нового свойства и функции в подкласс (TownSquare.kt)

```
class TownSquare : Room("The Town Square") {  
    override val status = "Bustling"  
    private var bellSound = "GWONG"  
  
    override fun enterRoom() {  
        narrate("The villagers rally and cheer as the hero enters")  
        ringBell()  
    }  
  
    fun ringBell() {  
        narrate("The bell tower announces the hero's presence: $bellSound")  
    }  
}
```

Для TownSquare доступны все свойства и функции, объявленные и в TownSquare, и в Room. Однако для Room недоступны свойства и функции, объявленные в TownSquare (такие, как ringBell).

Протестируйте функцию enterRoom, обновив переменную currentRoom в NyetHack.kt для создания экземпляра TownSquare.

Листинг 15.10. Вызов реализации функции подкласса (NyetHack.kt)

```
...  
fun main() {  
    narrate("Welcome to NyetHack!")  
    val playerName = promptHeroName()  
    player = Player(playerName)  
    // changeNarratorMood()
```

```

player.prophesize()

var currentRoom: Room = Room("The Foyer") TownSquare()
val mortality = if (player.isImmortal) "an immortal" else "a mortal"
narrate("${player.name} of ${player.hometown}, ${player.title},
        is in ${currentRoom.description()}")
narrate("${player.name}, $mortality, has ${player.healthPoints} health points")
currentRoom.enterRoom()
player.castFireball()
player.prophesize()
}
...

```

Снова запустите `NyetHack.kt`. В консоли появятся следующие строки:

```

...
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
  (Currently: Bustling)
Madrigal, a mortal, has 100 health points
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
A glass of Fireball springs into existence (x2)
...

```

Обратите внимание, что переменная `currentRoom` в `NyetHack.kt` до сих пор имеет тип `Room`, несмотря на то что сам экземпляр имеет тип `TownSquare`, а его функция `enterRoom` существенно изменилась по сравнению с реализацией в `Room`. Вы явно указали тип `Room` для `currentRoom`, поэтому переменная может ссылаться на комнату любого типа, несмотря на то что `currentRoom` был присвоен результат, возвращаемый конструктором `TownSquare`. Так как `TownSquare` является подклассом `Room`, этот синтаксис допустим.

Также можно определить подкласс подкласса (более глубокая иерархия). Если создать класс `Piazza` наследованием от `TownSquare`, то `Piazza` также будет типом `TownSquare` и типом `Room`. Глубина наследования ограничивается только здравым смыслом для организации кодовой базы. (И конечно же, вашим воображением.)

Вызов разных версий `enterRoom` в зависимости от класса объекта — пример идеи объектно-ориентированного программирования, которая называется *полиморфизмом*.

Полиморфизм — стратегия для упрощения структуры программы. Он позволяет повторно использовать функции, описывающие общие черты поведения группы классов (например, что происходит, когда игрок заходит в комнату), а также изменять поведение под уникальные потребности класса (например, ликующая толпа в `TownSquare`).

Определяя класс `TownSquare` как подкласс `Room`, мы объявили новую реализацию `enterRoom`, которая переопределила версию в `Room`. Теперь при вызове метода

`enterRoom` объекта `currentRoom` будет вызываться версия `enterRoom` для `TownSquare`. Соответственно, никаких изменений в `Nyethack.kt` вносить не придется.

Рассмотрим следующий заголовок функции:

```
fun drawBlueprint(room: Room)
```

`drawBlueprint` получает `Room` в параметре. Она также может получить любой подкласс `Room`, потому что любой подкласс будет обладать всеми характеристиками `Room`. Полиморфизм позволяет писать функции, которым важны только возможности класса, а не их реализации.

Открывать функции для переопределения весьма полезно, но есть и побочный эффект. Когда вы переопределяете функцию в Kotlin, переопределяющая функция в подклассе по умолчанию открыта для переопределения (если сам подкласс помечен ключевым словом `open`).

Что делать, если это нежелательно? Давайте рассмотрим пример с `TownSquare`. Допустим, вы хотите, чтобы любой подкласс `TownSquare` мог менять свое описание `description`, но не поведение `enterRoom`.

Добавьте ключевое слово `final`, чтобы запретить возможность переопределения функции. Откройте `TownSquare` и добавьте ключевое слово `final` в определение функции `enterRoom`, чтобы никакой подкласс не мог переопределить ликование жителей, когда герой приходит на городскую площадь.

Листинг 15.11. Объявление функции с ключевым словом `final` (`TownSquare.kt`)

```
open class TownSquare : Room("The Town Square") {
    override val status = "Bustling"
    private var bellSound = "GWONG"

    final override fun enterRoom() {
        narrate("The villagers rally and cheer as the hero enters")
        ringBell()
    }

    fun ringBell() {
        narrate("The bell tower announces the hero's presence: $bellSound")
    }
}
```

Теперь любой подкласс `TownSquare` сможет переопределить функцию `description`, но не `enterRoom`, потому что перед ней стоит ключевое слово `final`.

Как вы уже видели при первой попытке переопределения `enterRoom`, функции по умолчанию недоступны для переопределения, если они не были унаследованы от класса `open`. Добавление ключевого слова `final` к унаследованной функции гарантирует, что она не будет переопределена, даже если класс, в котором она объявлена, имеет модификатор `open`.

Итак, вы познакомились с тем, как использовать наследование, чтобы обеспечить совместное использование данных и поведения родственными классами. Также вы увидели, как использовать ключевые слова `open`, `final` и `override` для управления возможностью переопределения. Требуя явного использования ключевых слов `open` и `override`, Kotlin побуждает согласиться с наследованием. Это снижает риск раскрытия классов, не предназначенных для создания подклассов, и не позволит вам или кому-то другому переопределить функции там, где это не предусмотрено.

Проверка типов

Нельзя сказать, что NyetHack — невероятно сложная программа. Тем не менее ее завершенная кодовая база может включать множество классов и подклассов. Можно очень старательно выбирать имена для своих переменных, но все равно вы не всегда будете уверены в том, какой тип имеет та или иная переменная во время выполнения программы. Избавиться от сомнений в типе объекта вам поможет оператор `is`.

Опробуйте его в Kotlin REPL. Создайте экземпляр `Room`. (Возможно, вам понадобится импортировать `Room` в REPL командой `import com.bignerdranch.nyethack.Room`. Если вы используете автозаполнение при вводе объявления переменной, то IntelliJ сделает это за вас.)

Листинг 15.12. Создание экземпляра Room (REPL)

```
var room = Room("Foyer")
```

Проверьте, является ли `room` экземпляром класса `Room`, при помощи оператора `is`.

Листинг 15.13. Проверка room is Room (REPL)

```
room is Room  
true
```

Тип объекта слева от оператора `is` сравнивается с типом, указанным справа. Выражение возвращает `Boolean: true`, если типы совпадают, и `false`, если нет.

Теперь проверьте, является ли `room` экземпляром класса `TownSquare`.

Листинг 15.14. Проверка room is TownSquare (REPL)

```
room is TownSquare  
false
```

`room` имеет тип `Room`, который является родительским классом для `TownSquare`. Но объект `room` не является экземпляром `TownSquare`.

Попробуйте еще одну переменную, на этот раз с типом `TownSquare`.

Листинг 15.15. Проверка TownSquare (REPL)

```
var townSquare = TownSquare()
townSquare is TownSquare
true

townSquare is Room
true
```

`townSquare` имеет тип `TownSquare`, а также тип `Room`. Напомним, что именно эта идея делает полиморфизм возможным.

Если вам нужно узнать тип переменной, то проверка типа — самое простое решение. Используя проверку типов и условные команды, можно организовать ветвление логики. Но не забывайте о том, как полиморфизм влияет на эту логику.

Например, создайте выражение `when` в Kotlin REPL, которое возвращает `Room` или `TownSquare` в зависимости от типа переменной.

Листинг 15.16. Проверка типа в ветвлении с условием (REPL)

```
var className: String = when(townSquare) {
    is TownSquare -> "TownSquare"
    is Room -> "Room"
    else -> throw IllegalArgumentException()
}
print(className)
TownSquare
```

Первое условие в выражении `when` определяется как истинное, потому что `townSquare` — это тип `TownSquare`. Второе условие тоже истинное, потому что `townSquare` также относится к типу `Room`. Но для вас это уже неважно, так как первое условие удовлетворено и второе проверяться просто не будет. Запустите этот код, и в консоли появится строка `TownSquare`.

А теперь поменяем ветви местами.

Листинг 15.17. Проверка типов с обратным порядком условий (REPL)

```
var className: String = when(townSquare) {
    is Room -> "Room"
    is TownSquare -> "TownSquare"
    else -> throw IllegalArgumentException()
}
print(className)
Room
```

Запустите код. Теперь в консоли появится строка `Room`, поскольку первое условие определено как истинное.

При ветвлении по типу объекта важен порядок следования условий.

Иерархия типов в языке Kotlin

Если суперкласс не указан явно, все классы в Kotlin наследуют от общего суперкласса `Any`. Это означает, что каждый тип в языке в конечном итоге наследует от `Any`. В случае класса `TownSquare` формируется иерархия классов, показанная на рис. 15.2.

`Any` также можно использовать для определения функций, получающих аргументы любого типа. Допустим, в `NyetHack` источником благословения могут стать объекты двух типов: ранее благословенный игрок или комната с названием `The Fount of Blessings`. Функция `printIsSourceOfBlessings`, проверяющая, может ли объект стать источником благословения, получает аргумент типа `Any` и использует проверку типа для условного ветвления по типу переданного аргумента.

```
fun printIsSourceOfBlessings(any: Any) {  
    val isSourceOfBlessings: Boolean = if (any is Player) {  
        any.title == "The Blessed"  
    } else {  
        (any as Room).name == "The Fount of Blessings"  
    }  
  
    println("$any is a source of blessings: $isSourceOfBlessings")  
}
```

В этом коде встречаются некоторые новые концепции, которые мы рассмотрим в ближайших разделах. А пока заметим, что поскольку каждый объект является подклассом `Any`, в `printIsSourceOfBlessings` можно передавать аргументы любого типа. Такая гибкость полезна, но иногда она не позволяет сразу же начать работу с аргументом. В этом примере используется приведение типа (по аналогии с тем, как это делалось в главе 10), чтобы справиться с ненадежным аргументом `Any`.

Приведение типа

Применяя приведение типа, вы приказываете Kotlin во время выполнения программы работать с объектом так, как будто он является экземпляром другого типа. Это позволяет либо перекрывать часть функциональности, приводя объекты к типу суперкласса, либо обращаться к расширенной функциональности (функциям и свойствам) с приведением к типу подкласса.

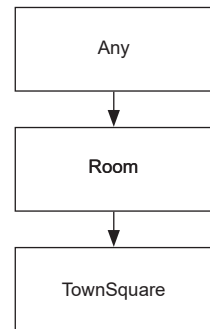


Рис. 15.2. Иерархия типов `TownSquare`

В функции `printIsSourceOfBlessings` условная команда проверяет тип аргумента `any`, сравнивая его с типом `Player`. Если условие не выполняется, то управление передается в ветвь `else`.

Код в ветви `else` ссылается на переменную `name`:

```
fun printIsSourceOfBlessings(any: Any) {
    val isSourceOfBlessings: Boolean = if (any is Player) {
        any.isBlessed
    } else {
        (any as Room).name == "Fount of Blessings"
    }
    println("$any is a source of blessings: $isSourceOfBlessings")
}
```

Оператор `as` используется для приведения значений к разным типам. По сути, он показывает: «В контексте этого выражения с переменной `any` следует работать так, как если бы она относилась к типу `Room`». Выражение в данном случае — это ссылка на свойство `name` класса `Room`, значение которого сравнивается со строкой `"Fount of Blessings "`.

Приведение типов — это сила, а с силой приходит ответственность: применять его следует осторожно. Приведение к суперклассу исходного значения — например, `Room` к `Any` или `Int` к `Number` — всегда безопасно. Другой пример мы показывали в главе 10, когда неявно приводили экземпляр `MutableMap` к `Map`. Как мы говорили, `MutableMap` расширяет `Map`.

Приведение в `printIsSourceOfBlesings` работает, но оно небезопасно. Почему? В `NyetHack` есть всего три класса — `Room`, `Player` и `TownSquare`, поэтому логично предположить, что если какой-то объект не игрок (имеет тип, отличный от `Player`), значит, он имеет тип `Room`. Но в стандартную библиотеку и платформенные API (например, в стандартную библиотеку Java и генерируемые связывания для пользователей Kotlin/JS и Kotlin/Native) входит множество классов, не говоря уже о классах, которые вы определите в будущем в своем собственном коде.

Приведение не работает, если тип экземпляра несовместим с типом, к которому его надо привести. Например, строка `String` не имеет ничего общего с `Int`, поэтому приведение `String` к `Int` породит исключение `ClassCastException`, которое вызывает аварийное завершение программы. (Помните, что приведение и преобразование — это разные вещи. Некоторые строки можно преобразовать в целые числа, но строковый тип `String` нельзя привести к числовому типу `Int`.)

Оператор приведения типа *пытается* привести любую переменную к любому типу, но вы должны позаботиться о том, чтобы это значение *можно было* привести к такому типу. Вы можете избежать рисков небезопасного преобразования при помощи умного приведения типа и оператора безопасного приведения типа (вы узнаете о них в разделе «Для любознательных: оператор безопасного приведения типа» в конце этой главы).

Если небезопасное приведение типа неизбежно, учтите, что в программе может произойти сбой. Лучше избегать приведения типов, если вы не уверены, что оно пройдет успешно.

Умное приведение типа

Один из способов убедиться в успехе приведения типа заключается в том, чтобы предварительно проверить тип приводимой переменной. Вернемся к первой ветви условного выражения в `printIsSourceOfBlessings`:

```
fun printIsSourceOfBlessings(any: Any) {
    val isSourceOfBlessings: Boolean = if (any is Player) {
        any.isBlessed
    } else {
        (any as Room).name == "Fount of Blessings"
    }

    println("$any is a source of blessings: $isSourceOfBlessings")
}
```

Согласно условию, для выполнения этой ветви аргумент `any` должен иметь тип `Player`. Внутри ветви код ссылается на свойство `isBlessed` экземпляра `any`. `isBlessed` — это свойство класса в `Player`, а не `Any`, как же возможно такое обращение без приведения типа?

На самом деле здесь выполняется умное приведение. Вы уже познакомились с ним в главе 7.

Компилятор Kotlin достаточно умен, чтобы определить: если проверка типа `any is Player` в условной команде выполнена успешно, то внутри ветви `any` можно считать экземпляром `Player`. Зная, что в этой ветви приведение `any` к `Player` всегда будет успешным, компилятор позволяет отбросить синтаксис приведения и просто обращаться к свойству `isBlessed` класса `Player`.

Один из способов превратить небезопасное приведение типа в `printIsSourceOfBlessings` в безопасное основан на включении дополнительной проверки с умным приведением типа:

```
fun printIsSourceOfBlessings(any: Any) {
    val isSourceOfBlessings: Boolean = if (any is Player) {
        any.isBlessed
    } else if (any is Room) {
        any.name == "Fount of Blessings"
    } else {
        false
    }

    println("$any is a source of blessings: $isSourceOfBlessings")
}
```


Такая функция уже не приводит к сбою, и теперь вы можете безопасно получать аргумент `any` независимо от его типа. Для этого мы воспользовались сообразительностью компилятора Kotlin, который способен автоматически и безопасно приводить типы посредством умного приведения. Другой способ основан на использовании оператора безопасного приведения типа, о котором мы расскажем ниже.

Рефакторинг кода таверны

Теперь, когда у нас есть механизм описания областей в `NyetHack`, вернемся к коду `Tavern`. Все поведение таверны в настоящее время определяется свойствами верхнего уровня и функциями. Стоит выделить немного времени и переработать код таверны в класс, расширяющий `Room`.

Начнем с определения нового класса `Tavern` в существующем файле `Tavern.kt`.

Листинг 15.18. Создание класса `Tavern` (`Tavern.kt`)

```
...
private val menuItemTypes = menuData.associate { (type, name, _) ->
    name to type
}

class Tavern : Room(TAVERN_NAME) {
    override val status = "Busy"
    override fun enterRoom() {
    }
}

fun visitTavern() {
    ...
}
...
```

Теперь скопируйте реализацию `visitTavern` в функцию `enterRoom` класса `Tavern`.

Листинг 15.19. Реализация `enterRoom` (`Tavern.kt`)

```
...
class Tavern : Room(TAVERN_NAME) {
    override val status = "Busy"

    override fun enterRoom() {
        narrate("${player.name} enters $TAVERN_NAME")
        narrate("There are several items for sale:")
        narrate(menuItems.joinToString())

        val patrons: MutableSet<String> = firstNames.shuffled()
            .zip(lastNames.shuffled()) { firstName, lastName ->
                "$firstName $lastName"
            }
    }
}
```

```

        .toMutableSet()

    val patronGold: MutableMap<String, Double> = mutableMapOf(
        TAVERN_MASTER to 86.00,
        player.name to 4.50,
        *patrons.map { it to 6.00 }.toTypedArray()
    )

    narrate("${player.name} sees several patrons in the tavern:")
    narrate(patrons.joinToString())

    val itemOfDay = patrons.flatMap { getFavoriteMenuItems(it) }.random()
    narrate("The item of the day is the $itemOfDay")

    repeat(3) {
        placeOrder(patrons.random(), menuItems.random(), patronGold)
    }
    displayPatronBalances(patronGold)

    patrons
        .filter { patron -> patronGold.getOrDefault(patron, 0.0) < 4.0 }
        .also { departingPatrons ->
            patrons -= departingPatrons
            patronGold -= departingPatrons
        }
        .forEach { patron ->
            narrate("${player.name} sees $patron departing the tavern")
        }

    narrate("There are still some patrons in the tavern")
    narrate(patrons.joinToString())
}
...

```

Скопируйте функцию `placeOrder` в новый класс `Tavern`.

Листинг 15.20. Копирование `placeOrder` в `Tavern` (`Tavern.kt`)

```

...
class Tavern : Room(TAVERN_NAME) {
    override val status = "Busy"

    override fun enterRoom() {
        ...
    }

    private fun placeOrder(
        patronName: String,
        menuItemName: String,
        patronGold: MutableMap<String, Double>
    ) {

```

```

val itemPrice = menuItemPrices.getValue(menuItemName)

narrate("$patronName speaks with $TAVERN_MASTER to place an order")
if (itemPrice <= patronGold.getDefault(patronName, 0.0)) {
    val action = when (menuItemTypes[menuItemName]) {
        "shandy", "elixir" -> "pours"
        "meal" -> "serves"
        else -> "hands"
    }

    narrate("$TAVERN_MASTER $action $patronName a $menuItemName")
    narrate("$patronName pays $TAVERN_MASTER $itemPrice gold")
    patronGold[patronName] = patronGold.getValue(patronName) - itemPrice
    patronGold[TAVERN_MASTER] = patronGold.getValue(TAVERN_MASTER) +
        itemPrice
} else {
    narrate("$TAVERN_MASTER says, \"You need more coin
        for a $menuItemName\")
}
}
}
...

```

После копирования этих двух функций в класс `Tavern` можно удалить функции верхнего уровня `visitTavern` и `placeOrder`. (Будьте внимательны: `getFavoriteMenuItems` удалять не нужно.)

Листинг 15.21. Удаление неиспользуемых функций (Tavern.kt)

```

...
class Tavern : Room(TAVERN_NAME) {
    override val status = "Busy"

    override fun enterRoom() {
        ...
    }

    private fun placeOrder(
        patronName: String,
        menuItemName: String,
        patronGold: MutableMap<String, Double>
    ) {
        ...
    }
}

fun visitTavern() {
    narrate("${player.name} enters $TAVERN_NAME")
    narrate("There are several items for sale:")
    narrate(menuItems.joinToString())
    ...
}

```

```

        narrate("There are still some patrons in the tavern")
        narrate(patrons.joinToString())
    }
    ...
private fun placeOrder(
    patronName: String,
    menuItemName: String,
    patronGold: MutableMap<String, Double>
){
    val itemPrice = menuItemPrices.getValue(menuItemName)

    narrate("$patronName speaks with $TAVERN_MASTER to place an order")
    if (itemPrice <= patronGold.getDefault(patronName, 0.0)){
        ...
    }else{
        ...
    }
}
}
...

```

Чтобы более эффективно использовать новый класс, Tavern должен хранить информацию о посетителях и содержимом их кошельков в свойстве класса. Внесите это изменение и удалите аргумент patronGold функции placeOrder.

Листинг 15.22. Выделение информации о посетителях и содержимом их кошельков в свойства (Tavern.kt)

```

...
class Tavern : Room(TAVERN_NAME) {
    val patrons: MutableSet<String> = firstNames.shuffled()
        .zip(lastNames.shuffled()) { firstName, lastName -> "$firstName $lastName"
    }
    .toMutableSet()

    val patronGold: MutableMap<String, Double> = mutableMapOf(
        TAVERN_MASTER to 86.00,
        player.name to 4.50,
        *patrons.map { it to 6.00 }.toTypedArray()
    )

    override val status = "Busy"

    override fun enterRoom() {
        narrate("${player.name} enters $TAVERN_NAME")
        narrate("There are several items for sale:")
        narrate(menuItems.joinToString())

        val patrons: MutableSet<String> = firstNames.shuffled()
            .zip(lastNames.shuffled()) { firstName, lastName ->
                "$firstName $lastName"
            }
    }
}

```

```

        .toMutableSet()

    val patronGold: MutableMap<String, Double> = mutableMapOf(
        TAVERN_MASTER to 86.00,
        player.name to 4.50,
        *patrons.map { it to 6.00 }.toTypedArray()
    )

    narrate("${player.name} sees several patrons in the tavern:")
    narrate(patrons.joinToString())

    val itemOfDay = patrons.flatMap { getFavoriteMenuItems(it) }.random()
    narrate("The item of the day is the $itemOfDay")

    repeat(3) {
        placeOrder(patrons.random(), menuItems.random(), patronGold)
    }
    ...
}

private fun placeOrder(
    patronName: String,
    menuItemName: String,
    patronGold: MutableMap<String, Double>
) {
    ...
}
}
...

```

Убедимся в том, что рефакторинг прошел успешно и не изменил поведение таверны. При запуске NyetHack герой должен входить в таверну, а не на городскую площадь.

Листинг 15.23. Тестирование Tavern после рефакторинга (NyetHack.kt)

```

...
fun main() {
    narrate("Welcome to NyetHack!")
    val playerName = promptHeroName()
    player = Player(playerName)
    // changeNarratorMood()
    player.prophesize()

    var currentRoom: Room = TownSquare() Tavern()
    ...
}
...

```

Запустите NyetHack. В консоли должен появиться уже знакомый вывод таверны:

```
...
Madrigal of Neversummer, The Renowned Hero, is in Taernyl's Folly
(Currently: Busy)
Madrigal, a mortal, has 100 health points
Madrigal enters Taernyl's Folly
There are several items for sale:
Dragon's Breath, Shirley's Temple, Goblet of LaCroix, Pickled Camel Hump,
    Iced Boilermaker, Hard Day's Work Ice Cream, Bite of Lembas Bread
Madrigal sees several patrons in the tavern:
Mordoc Downstrider, Alex Baggins, Sophie Fernsworth, Tariq Ironfoot
The item of the day is Bite of Lembas Bread
Mordoc Downstrider speaks with Taernyl to place an order
Taernyl says, "You need more coin for a Iced Boilermaker"
Sophie Fernsworth speaks with Taernyl to place an order
Taernyl pours Sophie Fernsworth a Dragon's Breath
Sophie Fernsworth pays Taernyl 5.91 gold
Alex Baggins speaks with Taernyl to place an order
Taernyl says, "You need more coin for a Iced Boilermaker"
Madrigal intuitively knows how much money each patron has
Taernyl has 91.91 gold
Madrigal has 4.50 gold
Mordoc Downstrider has 6.00 gold
Alex Baggins has 6.00 gold
Sophie Fernsworth has 0.09 gold
Tariq Ironfoot has 6.00 gold
Madrigal sees Sophie Fernsworth departing the tavern
There are still some patrons in the tavern
Mordoc Downstrider, Alex Baggins, Tariq Ironfoot
A glass of Fireball springs into existence (x2)
Madrigal thinks about their future
A fortune teller told Madrigal, "An intrepid hero from Neversummer shall some
    day form an unlikely bond between two warring factions"
```

В следующей главе мы добавим функциональность, позволяющую герою переходить из комнаты в комнату. При желании Мадригал сможет входить в комнату несколько раз. Некоторые уже реализованные особенности поведения необходимо обновить для этой новой реальности. Так, блюдо дня не должно меняться в течение дня — в контексте игрового времени NyetHack это означает, что оно должно оставаться неизменным до выхода и перезапуска NyetHack. Для этого следует переместить переменную `itemOfDay` из функции `enterRoom`, чтобы ее значение сохранялось в программе. Также следует переместить сообщение о блюде дня, чтобы оно выводилось вместе с меню.

Листинг 15.24. Сохранение блюда дня (Tavern.kt)

```

...
class Tavern : Room(TAVERN_NAME) {
    ...
    val itemOfDay = patrons.flatMap { getFavoriteMenuItems(it) }.random()

    override val status = "Busy"

    override fun enterRoom() {
        narrate("${player.name} enters $TAVERN_NAME")
        narrate("There are several items for sale:")
        narrate(menuItems.joinToString())
        narrate("The item of the day is the $itemOfDay")

        narrate("${player.name} sees several patrons in the tavern:")
        narrate(patrons.joinToString())

        val itemOfDay = patrons.flatMap { getFavoriteMenuItems(it) }.random()
        narrate("The item of the day is the $itemOfDay")
        ...
    }
    ...
}
...

```

Пожалуй, вывод таверны занимает слишком много места. Чтобы сократить вывод `NyetHack`, сделаем так, чтобы при входе героя в таверну заказ поступал только от одного посетителя. Также мы доказали, что деньги правильно переходят со счета на счет, так что учетная система всеведущего героя уже не требуется. Удалите функцию `displayPatronBalances`, она больше не понадобится.

Мадригал никогда не должен входить в пустую таверну, а это может произойти, если все посетители покинут таверну, истратив свое золото. Чтобы предотвратить такую возможность, удалите логику, которая заставляет посетителей уходить из таверны, когда на их счету остается мало средств.

Листинг 15.25. Обновление учета средств в таверне (Tavern.kt)

```

...
class Tavern : Room(TAVERN_NAME) {
    ...
    override fun enterRoom() {
        narrate("${player.name} enters $TAVERN_NAME")
        narrate("There are several items for sale:")
        narrate(menuItems.joinToString())
        narrate("The item of the day is $itemOfDay")

        narrate("${player.name} sees several patrons in the tavern:")
        narrate(patrons.joinToString())
    }
}

```

```

repeat(3){
  placeOrder(patrons.random(), menuItems.random())
}
displayPatronBalances()

patrons
  .filter { patron → patronGold.getOrDefault(patron, 0.0) < 4.0 }
  .also { departingPatrons →
    patrons -= departingPatrons
    patronGold -= departingPatrons
  }
  .forEach { patron →
    narrate("${player.name} sees $patron departing the tavern")
  }
  narrate("There are still some patrons in the tavern")
  narrate(patrons.joinToString())
}
...
}
...
private fun displayPatronBalances(patronGold: Map<String, Double>) {
  narrate("${player.name} intuitively knows how much money each patron has")
  patronGold.forEach { (patron, balance) →
    narrate("$patron has ${"%0.2f".format(balance)} gold")
  }
}

```

Запустите NyetHack после внесения всех изменений. Результат должен выглядеть примерно так:

```

...
Madrigal, a mortal, has 100 health points
Madrigal enters Taernyl's Folly
There are several items for sale:
Dragon's Breath, Shirley's Temple, Goblet of LaCroix, Pickled Camel Hump, Iced
  Boilermaker, Hard Day's Work Ice Cream, Bite of Lembas Bread
The item of the day is Iced Boilermaker
Madrigal sees several patrons in the tavern:
Mordoc Downstrider, Tariq Ironfoot, Alex Baggins, Sophie Fernsworth
Tariq Ironfoot speaks with Taernyl to place an order
Taernyl says, "You need more coin for a Iced Boilermaker"
A glass of Fireball springs into existence (x2)
Madrigal thinks about their future
A fortune teller told Madrigal, "An intrepid hero from Neversummer shall some
  day bring the gift of creation back to the world"

```

Итак, мы провели рефакторинг кода таверны для использования нового класса `Room`. Хотите спросить, почему мы храним `getFavoriteMenuItems` и большинство свойств таверны на уровне файла, вместо того чтобы переместить их в класс? Они существуют за пределами класса, потому что не должны изменяться с реализацией таверны. Каждый экземпляр таверны должен содержать одни и те же пункты меню,

а любимые блюда посетителей не должны изменяться в зависимости от того, в какую конкретную таверну заходит посетитель.

С другой стороны, свойства меню и функцию `getFavoriteMenuItems` можно переместить в класс `Tavern`. Такое решение абсолютно допустимо, и возможно, оно оправданно при написании класса `Tavern` с самого начала (вместо проведения рефакторинга). Каждый из способов абсолютно допустим, и в вашей воле выбрать тот вариант, который вам больше нравится. Объявления верхнего уровня можно свободно смешивать с классами в одном файле, что открывает дополнительные возможности для группировки кода.

В этой главе мы показали, как использовать подклассы для совместного использования поведения в разных классах. В следующей главе вы познакомитесь с другими типами классов, включая классы данных, перечисления и `object` (класс с одним экземпляром), когда будете добавлять цикл игры в `NyetHack`.

Для любознательных: Any

При выводе значения переменной в консоль вызывается функция с именем `toString`, которая определяет, как значение должно выглядеть в консоли. Для некоторых типов это просто: например, значение строки выражается через значение `String`. Но для других типов это не так очевидно.

`Any` включает абстрактные определения функций вроде `toString` — они поддерживаются реализацией целевой платформы, для которой предназначен ваш проект.

Заглянув в исходный код класса `Any`, вы увидите в нем следующее:

```
/**
 * Корень иерархии классов Kotlin.
 * Класс [Any] является суперклассом каждого класса Kotlin.
 */
public open class Any {
    public open operator fun equals(other: Any?): Boolean
    public open fun hashCode(): Int
    public open fun toString(): String
}
```

Определение класса не содержит никакого определения функции `toString`. Так где же определяется функция `toString` и что она возвращает, допустим, при вызове для `Player`?

Вспомните последнюю строку, выводимую `printIsSourceOfBlessings` в консоль:

```
fun printIsSourceOfBlessings(any: Any) {
    val isSourceOfBlessings: Boolean = if (any is Player) {
        any.isBlessed
    } else {
        (any as Room).name == "Fount of Blessings"
    }
}
```

```
println("$any is a source of blessings: $isSourceOfBlessings")
}
```

Если вызвать `printIsSourceOfBlessings` и передать ей игрока, находящегося под благословением, результат будет выглядеть примерно так:

```
Player@71efa55d is a source of blessings: true
```

`Player@71efa55d` — результат выполнения реализации `toString` по умолчанию класса `Any`. Kotlin использует эту реализацию для JVM и для платформенных приложений (Kotlin/JS вместо этого выдаст строку `[object Object]`). Вы можете переопределить `toString` в своем классе `Player`, чтобы она возвращала что-то более понятное для человека.

Тип `Any` — один из способов обеспечения платформенной независимости Kotlin; он служит абстракцией, представляющей типичный суперкласс для каждой конкретной платформы (например, JVM). То есть когда целевой платформой является JVM, для `toString` в `Any` выбирается реализация `java.lang.Object.toString`, но при компиляции в JavaScript ее реализация будет отличаться. Наличие такой абстракции означает, что вам необязательно знать особенности каждой системы, в которой будет запускаться ваш код. Просто положитесь на тип `Any`.

Для любознательных: оператор безопасного приведения типа

Ранее в этой главе мы представили оператор `as` как механизм приведения типов. Мы также упоминали, что приведение типа иногда оказывается небезопасным и может привести к исключению `ClassCastException` при выполнении недопустимого приведения во время запуска программы.

Кроме оператора `as`, также существует оператор безопасного приведения типа `as?`. У `as?` очень много общего с `as`, но с одним принципиальным отличием: при выполнении недопустимого приведения типа `as?` возвращает `null` вместо исключения. Вы можете убедиться в этом в REPL.

Листинг 15.26. Безопасное и небезопасное приведение типа (REPL)

```
5 as String
```

```
ClassCastException: class Integer cannot be cast to class String
```

```
5 as? String
```

```
null
```

Оператор `as?` чрезвычайно удобен для объединения проверки типа с приведением типа в одной команде. Если приведение типа завершается неудачей, вы можете воспользоваться знакомыми приемами `null`-безопасности из главы 7 для реализации резервного поведения вместо сбоя.

16. Объекты, классы данных и перечисления

В трех предыдущих главах вы научились применять объектно-ориентированное программирование для построения значимых отношений между объектами. Несмотря на разнообразие вариантов инициализации, все классы, с которыми вы работали до этого момента, объявлялись ключевым словом `class`. В этой главе мы познакомим вас с *объявлениями объектов*, а также с другими типами классов: *вложенными классами* (nested classes), *классами данных* (data classes) и *классами-перечислениями* (enum classes). Как вы увидите далее, каждый из них имеет свой синтаксис объявления и уникальные характеристики.

К концу главы наш герой сможет перемещаться из комнаты в комнату в мире NyetHack. Также мы улучшим структуру программы и подготовим ее к изменениям, которыми займемся в следующих главах.

Ключевое слово `object`

В главе 14 вы научились конструировать классы. Конструктор класса возвращает экземпляр класса; конструктор разрешается вызвать любое количество раз, чтобы создать любое количество экземпляров.

Например, в NyetHack может быть сколько угодно игроков, поскольку конструктор `Player` можно вызвать столько раз, сколько потребуется. Для `Player` это вполне целесообразно — мир NyetHack достаточно велик, чтобы вместить нескольких игроков.

Но представьте, что вы решили создать класс `Game`, который будет следить за текущим состоянием игры. Наличие нескольких экземпляров класса `Game` в игре нежелательно. Ведь каждый из них способен хранить свое состояние, что, скорее всего, приведет к их десинхронизации.

Если вам необходим один экземпляр с непротиворечивым состоянием, существующий на протяжении всего времени работы программы, объявите *синглтон* (singleton). С ключевым словом `object` вы сообщаете, что класс будет ограничен единственным экземпляром — синглтоном. Экземпляр такого класса создается автоматически при первом обращении к нему. Этот экземпляр существует на всем

протяжении работы программы, и при каждом следующем обращении будет возвращаться первоначальный экземпляр.

Есть три способа применения ключевого слова `object`: для *объявления объектов* (object declarations), для *объектов-выражений* (object expressions) и для *объектов-компаньонов* (companion objects). Мы обозначим рамки применения каждого из них в следующих разделах.

Объявления объекта

Объявления объекта полезны для организации и управления состоянием, особенно когда надо поддерживать какое-то состояние на протяжении всей работы программы. Объект `Game` будет объявлен именно для этой цели.

Создание класса `Game` через объявление объекта также предоставит удобное место для определения цикла игры и сделает функцию `main` в `NyetHack.kt` более чистой. Разделение кода на классы и объявление объектов способствует тому, чтобы поддерживать организованную структуру кода.

Объявите объект `Game` в `NyetHack.kt`, используя объявление объекта.

Листинг 16.1. Объявление объекта `Game` (`NyetHack.kt`)

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

private fun promptHeroName(): String {
    ...
}

object Game {
}
```

В этом примере мы решили добавить класс `Game` в существующий файл `NyetHack.kt`, так как он является важной служебной конструкцией, влияющей на работу игры. Но при желании `Game` можно разместить в отдельном файле. (В этом случае мы рекомендуем присвоить файлу имя `Game.kt`.) В Kotlin поддерживается гибкий подход к содержимому файлов, поэтому выбор за вами.

Функция `main` в `NyetHack.kt` должна служить исключительно для запуска игрового процесса. Вся игровая логика будет сосредоточена внутри объекта `Game`, существующего в единственном экземпляре.

Так как экземпляр синглтона создается автоматически, нет необходимости добавлять свой конструктор с кодом инициализации. Вместо этого достаточно определить блок инициализации, выполняющий все необходимое для инициализации

вашего объекта. Добавим такой блок в объект `Game`, который выведет на консоль приветствие при его создании.

Листинг 16.2. Добавление блока `init` в `Game` (`Game.kt`)

```
...
object Game {
    init {
        narrate("Welcome, adventurer")
    }
}
```

Запустите `NyetHack`. Приветствие не выводится, потому что объект `Game` не был инициализирован. А не инициализирован он из-за того, что мы к нему пока не обращались.

Чтобы обратиться к объекту, нужно сослаться на одно из его свойств или функций. Чтобы запустить инициализацию `Game`, объявим и вызовем функцию с именем `play`. В `play` мы разместим основной цикл игры `NyetHack`.

Добавьте функцию `play` в `Game` и вызовите ее из `main`. Функция объекта вызывается с использованием имени объекта, в котором она объявлена, а не экземпляра класса, как в случае с функциями класса.

Листинг 16.3. Вызов функции из объявления объекта (`NyetHack.kt`)

```
...
fun main() {
    ...
    player.castFireball()
    player.prophesize()

    Game.play()
}

private fun promptHeroName(): String {
    ...
}

object Game {
    init {
        narrate("Welcome, adventurer")
    }

    fun play() {
        while (true) {
            // Игра в NyetHack
        }
    }
}
```

Объект `Game` не только инкапсулирует состояние игры, но и выполняет цикл игры, получающий и обрабатывающий команды игрока. Ваш цикл игры примет форму цикла `while`, что сделает игру `NyetHack` более интерактивной. `while` имеет простое условие `true`, которое поддерживает работу цикла, пока приложение не будет закрыто.

Пока что функция `play` не делает ничего. Скоро она разобьет игровой процесс `NyetHack` на раунды: в каждом из них в консоль будут выводиться состояние игрока и прочая информация о мире, после чего данные пользователя будут вводиться с помощью функции `readLine`.

Посмотрите на игровую логику `main` и подумайте, где она должна располагаться в `Game`. Например, нет смысла создавать новую комнату `currentRoom` в начале каждого раунда, поэтому эта часть игровой логики должна находиться в `Game`, а не в `play`. Объявите `currentRoom` как `private`-свойство `Game`. Затем переместите вызовы `narrate` и `enterRoom` в цикл `while` функции `play`, потому что эта логика должна выполняться как часть игры: мы оповещаем игрока о том, что происходит вокруг.

Листинг 16.4. Перемещение логики в `Game` (`NyetHack.kt`)

```
...
fun main() {
    narrate("Welcome to NyetHack!")
    val playerName = promptHeroName()
    player = Player(playerName)
    // changeNarratorMood()
    player.prophesize()

    var currentRoom: Room = Tavern()
    val mortality = if (player.isImmortal) "an immortal" else "a mortal"
    narrate("${player.name} of ${player.hometown}, ${player.title},
           is in ${currentRoom.description()}")
    narrate("${player.name}, $mortality, has ${player.healthPoints} health points")
    currentRoom.enterRoom()

    player.castFireball()
    player.prophesize()

    Game.play()
}

private fun promptHeroName(): String {
    ...
}

object Game {
    private var currentRoom: Room = TownSquare()
```

```

init {
    narrate("Welcome, adventurer")
}

fun play() {
    while (true) {
        // Игра в NyetHack
        narrate("${player.name} of ${player.hometown}, ${player.title},
            is in ${currentRoom.description()}")
        currentRoom.enterRoom()
    }
}
}

```

Переместив код из `main` в функцию `play` из `Game`, вы сохраните все необходимое для настройки игрового цикла в объекте `Game`.

Что осталось в `main`? Инициализация игрока, изменение настроения рассказчика, двойное пророчество, состояние здоровья игрока и применение заклинания.

Инициализация игрока и настроение рассказчика могут остаться в `main`, но игрок не должен творить заклинания или размышлять о своем будущем вне цикла игры. Аналогичным образом сообщение о здоровье игрока должно выводиться в начале игры — в блоке `init` функции `Game`. Удалите вызовы `castFireball` и `prophesize`, затем переместите сообщение о здоровье в `Game`, чтобы завершить выделение логики игры в новый класс.

Листинг 16.5. Улучшение `main` (`NyetHack.kt`)

```

fun main() {
    narrate("Welcome to NyetHack!")
    val playerName = promptHeroName()
    player = Player(playerName)
    // changeNarratorMood()
    player.prophesize()

    val mortality = if (player.isImmortal) "an immortal" else "a mortal"
    narrate("${player.name}, $mortality, has ${player.healthPoints} health points")

    player.castFireball()
    player.prophesize()

    Game.play()
}

...
object Game {
    ...
    init {
        narrate("Welcome, adventurer")
        val mortality = if (player.isImmortal) "an immortal" else "a mortal"
    }
}

```

```

        narrate("${player.name}, $mortality, has ${player.healthPoints} health
            points")
    }
    ...
}

```

Если запустить NyetHack.kt прямо сейчас, цикл будет повторяться бесконечно, потому что он ничем не прерывается. Последний шаг, по крайней мере сейчас, — это прием пользовательского ввода из консоли с использованием функции `readLine`. Вспомните, что функция `readLine` приостанавливает выполнение и ожидает пользовательского ввода. Затем она возобновляет выполнение, возвращая полученные данные.

Добавьте вызов `readLine` в цикл игры, чтобы получить пользовательский ввод.

Листинг 16.6. Получение пользовательского ввода (NyetHack.kt)

```

...
object Game {
    ...
    fun play() {
        while (true) {
            narrate("${player.name} of ${player.hometown}, ${player.title},
                is in ${currentRoom.description()}")
            currentRoom.enterRoom()

            print("> Enter your command: ")
            println("Last command: ${readLine()}")
        }
    }
}

```

Попробуйте запустить NyetHack и ввести команду:

```

Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Madrigal
Welcome, adventurer
Madrigal, a mortal, has 100 health points
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command: fight
Last command: fight
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command:

```

Введенный текст также выводится. Отлично: введенные данные передаются игре.

Объекты-выражения

Определение классов с ключевым словом `class` полезно тем, что оно вводит в кодовую базу новые концепции. Определив класс с именем `Room`, вы сообщили, что в `NyetHack` есть комнаты. А определив подкласс `TownSquare`, вы указали, что существует особая разновидность комнат — городские площади.

Но иногда объявление нового именованного класса выглядит излишеством. Например, в некоторых случаях нужен экземпляр класса, немного отличающегося от уже существующего, и этот экземпляр предполагается использовать лишь однажды. Более того, он будет настолько временным, что ему даже имя не требуется.

Еще один вариант использования ключевого слова `object`: объекты-выражения. Рассмотрим следующий пример:

```
val abandonedTownSquare = object : TownSquare() {
    override fun enterRoom() {
        narrate("The hero anticipated applause, but no one is here...")
    }
}
```

Этот объект-выражение определяет подкласс `TownSquare` (по аналогии с тем, как мы определяли подклассы в главе 13) и возвращает его экземпляр. Новый подкласс переопределяет функцию `enterRoom`, чтобы никто не приветствовал появление героя. Тело объекта-выражения работает так же, как тело класса: в нем можно переопределять и создавать новые свойства и функции по своему усмотрению.

Этот класс соблюдает многие правила класса `object`. Хотя объекты-выражения не являются синглтонами, они остаются классами, предназначенными для одноразового применения. Вы не сможете создать второй экземпляр этого класса, потому что вам недоступен его конструктор — и даже соответствующий класс.

Объекты-выражения создают особую разновидность классов, называемую *анонимным классом*. Эта концепция очень похожа на анонимные функции (которые мы называли лямбда-функциями). Так как анонимные классы определяются без ключевого слова `class`, использовать их как тип невозможно. Также не удастся обратиться к свойствам и функциям, объявленным в анонимном классе, за пределами функции, в которой он был создан.

С другой стороны, объявление объекта определяет новый тип для соответствующего синглтона. Вспомните объект `Game`: к его функциям и свойствам можно обращаться в любой точке кода (если доступ не ограничивается модификаторами видимости).

Объекты-выражения также имеют существенно меньшую область видимости, чем класс `object`, и, как результат, объект-выражение получает некоторые атрибуты

в зависимости от места его объявления. При объявлении на уровне файла объект-выражение инициализируется немедленно. При объявлении внутри другого класса он инициализируется при инициализации вмещающего класса.

Объекты-компаньоны

Если вы хотите добавить поведение в класс, к которому можно обращаться как при наличии экземпляра класса, так и без него, то можно воспользоваться *объектом-компаньоном*. Объекты-компаньоны объявляются внутри другого объявления класса с помощью модификатора `companion`. У класса не может быть больше одного объекта-компаньона.

Объект-компаньон определяет синглтон как объект класса, который был описан ранее. Если у класса имеется объект-компаньон, класс может вести себя и как обычный класс, и как объект класса.

В следующем примере используется объект-компаньон, определенный для класса `Player`:

```
class Player(...) {
    constructor(saveFileBytes: ByteArray) : this(...)

    companion object {
        private const val SAVE_FILE_NAME = "player.dat"

        fun fromSaveFile() = Player(File(SAVE_FILE_NAME).readBytes())
    }
}
```

Класс `Player` содержит объект-компаньон с одной функцией `fromSaveFile`. Если вы будете вызывать `fromSaveFile` из любой другой точки кодовой базы, экземпляр `Player` вам для этого не понадобится:

```
val player = Player.fromSaveFile()
```

При необходимости объекты-компаньоны также могут иметь собственную логику инициализации. Инициализатор объекта-компаньона вызывается при инициализации вмещающего его класса либо при прямом обращении к одной из его функций или свойств. Но сколько бы раз ни создавался экземпляр `Player`, будет существовать только один экземпляр его объекта-компаньона.

В табл. 16.1 сравниваются три способа определения объектов в коде с ключевым словом `object`.

Если вы будете понимать, когда и как используются объявления объектов, объекты-выражения и объекты-компаньоны, это облегчит вам их применение. А это, в свою очередь, поможет писать хорошо организованный и хорошо масштабируемый код.

Таблица 16.1. Применение ключевого слова object

Синтаксис	Описание
<pre>object Game{ val player: Player = ... } Game.player</pre>	Объявления объектов могут располагаться везде, где можно объявлять классы. Объявления объектов определяют класс-синглтон. Это полезно, если вы хотите инкапсулировать поведение в классе, но при этом в любой момент времени должен существовать только один экземпляр класса.
<pre>val singleUseRoom = object : Room(name = "Pocket Dimension") { override fun enterRoom() { narrate("Madrigal doesn't think she's in Kronstadt anymore") } } singleUseRoom.enterRoom() <i>Madrigal doesn't think she's in Kronstadt anymore</i></pre>	Объекты-выражения обычно используются как аргументы функций или значения, присваиваемые переменным. Объекты-выражения определяют и создают одноразовый экземпляр, расширяющий другой тип. Это полезно, если вы хотите создать экземпляр класса и переопределить часть его поведения без выделения полноценного класса.
<pre>class SpellBook(val spells: List<String> { companion object { fun createDefault(): SpellBook = SpellBook(listOf("Thundersurge", "Arcane Ammunition", "Reverse Damage")) } } val spells = SpellBook.createDefault()</pre>	Объекты-компаньоны определяются внутри другого класса. Объекты-компаньоны представляют собой классы-синглтоны, связанные с другим классом. Если вам нужен класс, для которого возможно создание нескольких экземпляров, но при этом вы также хотите определять поведение, доступное без экземпляра класса, такое глобальное поведение можно определить в объекте-компаньоне.

Вложенные классы

Не все классы, определенные внутри других классов, объявляются без имени. Также можно использовать ключевое слово `class` для объявления именованного класса, *вложенного* в другой класс. В этом разделе вы объявите новый класс `GameInput`, вложенный в объект `Game`.

Теперь, когда мы определили игровой цикл, можно заняться анализом команд, которые вводит пользователь. `NyetHack` — это текстовая игра, управляемая командами пользователя, для чтения которых используется функция `readLine`. При проверке команды, введенной пользователем, важно, во-первых, проверить допустимость команды и, во-вторых, правильно обработать команду, состоящую из нескольких частей. Например, `move east` (иди на восток): слово `move` преобразуется в вызов функции `move`, а слово `east` — в вызов `move` в виде аргумента, определяющего направление движения.

Рассмотрим эти два требования. Начнем с анализа команд, состоящих из нескольких частей. Логiku отделения команды от ее аргументов мы поместим в класс `GameInput`.

Создайте внутри `Game` `private`-класс для реализации этой абстракции.

Листинг 16.7. Определение вложенного класса (`NyetHack.kt`)

```
...
object Game {
    ...
    private class GameInput(arg: String?) {
        private val input = arg ?: ""
        val command = input.split(" ")[0]
        val argument = input.split(" ").getOrNull(1) { "" }
    }
}
```

Почему `GameInput` объявлен приватным и вложен в `Game`? Дело в том, что класс `GameInput` непосредственно связан только с объектом `Game` и не должен быть доступен из других точек `NyetHack`. Объявляя вложенный класс `GameInput` приватным, вы сообщаете, что `GameInput` может использоваться только внутри `Game` и не должен загромождать его общедоступный API.

В классе `GameInput` определяются два свойства: для команды и для аргумента. Для этого вызывается функция `split`, которая разбивает входную строку по символу пробела, а затем `getOrNull` для получения второго элемента из списка, созданного `split`. Если элемент с указанным индексом не существует, `getOrNull` вернет пустую строку.

Итак, появилась возможность разбивать команды на части. Теперь можно переходить к построению инфраструктуры обработки команд.

Для этого мы воспользуемся выражением `when` для построения коллекции допустимых команд `Game`. Добавьте в `GameInput` функцию `processCommand`. Функция использует выражение `when` для организации ветвления по командам, вводимым пользователем.

К обработке команд мы вернемся позже, а пока добавим резервное действие для случая, когда пользователь вводит недействительную команду. Не забудьте «причесать» ввод пользователя, вызвав `lowercase` для введенной команды.

Листинг 16.8. Определение функции во вложенном классе (NyetHack.kt)

```
...
object Game {
    ...
    private class GameInput(arg: String?) {
        private val input = arg ?: ""
        val command = input.split(" ")[0]
        val argument = input.split(" ").getOrNull(1) { "" }

        fun processCommand() = when (command.lowercase()) {
            else -> narrate("I'm not sure what you're trying to do")
        }
    }
}
```

Пора применить `GameInput` на практике. Замените вызов `readLine` в `Game.play` версией, использующей класс `GameInput`.

Листинг 16.9. Использование `GameInput` (NyetHack.kt)

```
...
object Game {
    ...
    fun play() {
        while (true) {
            narrate("${player.name} of ${player.hometown}, ${player.title},
                is in ${currentRoom.description()}")
            currentRoom.enterRoom()

            print("> Enter your command: ")
            println("Last command: ${readLine()}")
            GameInput(readLine()).processCommand()
        }
    }
    ...
}
```

Запустите `NyetHack`. (Если вы обычно пользовались кнопкой запуска на панели инструментов в верхней части окна IntelliJ, возможно, вы заметите, что ее сейчас нет. Дело в том, что игра все еще ожидает ввода от последнего запуска. Используйте-

те кнопку остановки с перезапуском , заменившую кнопку запуска на панели инструментов, или кнопку запуска рядом с `main`. В открывшемся временном окне выберите команду `Stop and rerun`.)

Теперь при любом вводе срабатывает ответ «неизвестная команда»:

```
Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Madrigal
Welcome, adventurer
Madrigal, a mortal, has 100 health points
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command: fight
I'm not sure what you're trying to do
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command:
```

Явный прогресс: мы ограничили ввод командами, входящими в небольшой (пока пустой) белый список. Далее в этой главе мы добавим команду `move`, и тогда `GameInput` станет немного полезнее.

Но прежде чем вы сможете перемещаться по миру `NyetHack`, надо создать мир, в котором есть еще что-то, кроме городской площади.

Классы данных

Первый шаг на пути построения мира для героя — это создание системы координат для перемещения. Система координат будет использовать основные направления движения, а также класс с именем `Coordinate` для представления изменения направления.

`Coordinate` — простой тип и хороший кандидат для определения в форме *класса данных*. Как можно понять из названия, классы данных спроектированы специально для хранения данных и предлагают широкие возможности для работы с данными.

Создайте новый файл `Navigation.kt` и добавьте в него `Coordinate` как класс данных, используя ключевое слово `data`. У `Coordinate` в конструкторе должны быть определены два свойства: координата `x` и координата `y`.

Листинг 16.10. Определение класса данных (`Navigation.kt`)

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

Для отслеживания текущего положения игрока на карте мира добавьте в объект `Game` свойство с именем `currentPosition`.

Листинг 16.11. Отслеживание позиции игрока (NyethHack.kt)

```
...
object Game {
    private var currentRoom: Room = TownSquare()
    private var currentPosition = Coordinate(0, 0)
    ...
}
```

В главе 15 вы узнали, что все классы в Kotlin являются потомками одного класса — `Any`. В `Any` определена группа функций, которые можно вызвать для любого экземпляра. В эту группу входят функции `toString`, `equals` и `hashCode`.

`Any` предоставляет для этих функций реализации по умолчанию, но как вы уже могли заметить, они не всегда удобны. Классы данных предоставляют свои реализации функций, которые иногда лучше соответствуют требованиям вашего проекта. В этом разделе мы поговорим об этих функциях, а также о некоторых других преимуществах использования классов данных для представления данных в кодовой базе.

toString

Реализация `toString` по умолчанию возвращает для класса малопонятную строку. Возьмем для примера класс `Player`. Он определяется как обычный класс, и вызов `toString` для экземпляра в Kotlin/JVM и Kotlin/Native вернет строку вида:

```
Player@3527c201
```

Реализация по умолчанию возвращает строку вида *ИмяКласса@хешКод* — имя класса, за которым следует фактически случайное шестнадцатеричное число.

В своем классе вы можете переопределить `toString`, как любую другую открытую функцию. Но классы данных избавляют от этой работы, предлагая реализацию по умолчанию. Для `Coordinate` эта реализация вернет строку вида:

```
Coordinate(x=1, y=0)
```

Так как `x` и `y` — это свойства, объявленные в главном конструкторе `Coordinate`, они используются для представления `Coordinate` в текстовой форме. (Свойства, объявленные за пределами конструктора, не включены в вывод.) Реализация `toString` в классах данных значительно полезнее реализации по умолчанию в `Any`.

equals и hashCode

Как вы думаете, какой результат вернет следующее выражение:

```
Room("The Haunted Mines") == Room("The Haunted Mines")
```

Как ни странно, оно вернет `false`. По умолчанию объекты сравниваются по ссылкам, так как это реализация по умолчанию функции `equals` в `Any`. Так как эти два экземпляра существуют независимо друг от друга, то на них будут указывать разные ссылки, и поэтому они не равны.

Возможно, вам захочется считать две комнаты с одинаковыми названиями равными. Реализуйте проверку равенства, переопределив `equals` в своем классе, и сравнивайте свойства, а не ссылки на память. Вы уже видели, как классы вроде `String` используют этот подход для сравнения по значению.

И снова классы данных выполняют за вас эту работу, используя свою реализацию `equals`, которая сравнивает свойства, объявленные в главном конструкторе. Если объявить `Coordinate` как класс данных, выражение `Coordinate(1, 0) == Coordinate(1, 0)` будет возвращать `true`, так как значения свойств `x` и `y` обоих экземпляров равны.

Классы данных также предоставляют реализацию функции `hashCode`. Каждый раз, когда вы переопределяете функцию `equals`, вы также должны предоставить соответствующее переопределение `hashCode`, чтобы избежать появления в программе трудно локализуемых ошибок. `hashCode` возвращает числовое представление объекта; среди прочего, функция используется коллекциями `Set` и `Map` для быстрого поиска.

Функция `hashCode` подчиняется двум правилам. Первое: если два объекта равны на основании функции `equals`, они должны иметь одинаковые хеш-коды. Второе: хеш-код объекта не должен изменяться, пока не изменится одно из его свойств.

В общем случае мы не рекомендуем писать собственную реализацию `hashCode`. Когда вы переопределяете `equals` и вам необходимо предоставить соответствующую реализацию `hashCode`, среда `IntelliJ` может сгенерировать реализацию за вас. О том, как это делается, мы расскажем в разделе «Для любознательных: определение структурного сравнения» в конце главы.

Наконец, если вам когда-либо понадобится реализация `equals`, то прежде чем объявлять ее самостоятельно, подумайте, не будет ли класс данных более подходящим вариантом.

copy

Кроме более удобных реализаций функций из `Any`, классы данных также предоставляют функцию, которая позволяет легко создать копию объекта — возможно, с изменением значений.

`Coordinate` определяется как класс данных со свойствами `x` и `y`, доступными только для чтения. Напрямую изменять координаты невозможно, так что если вы захотите изменить их, вам придется создать новый объект. Для этого можно снова вызвать конструктор с нужными значениями, но функция `copy` позволяет

выполнить операцию более компактно и не указывать значения, которые вы не хотите изменять. Таким образом, вы можете создать копию координатной точки.

Создание копии в той же позиции `val duplicatedCoordinate = coordinate.copy()`

Создание копии у левого края карты `val leftCoordinate = coordinate.copy(x = 0)`

Создание копии у верхнего края карты `val topCoordinate = coordinate.copy(y = 0)`

Многие разработчики на Kotlin считают правильным использовать в классах данных только свойства `val`. Этот подход предотвращает такие ошибки, как состояние гонки, а также неожиданное поведение, связанное с изменяемостью. Для классов данных, имеющих указанную структуру, применение функции `copy` для модификации данных приложения практически неизбежно.

Деструктуризация объявлений

Еще одно преимущество классов данных — поддержка автоматической деструктуризации данных.

Ранее вы уже видели пример деструктуризации с такими типами, как `Pair` и `List` (глава 11). Под внешней оболочкой деструктуризация опирается на функции с именами `component1`, `component2` и т. д. Каждая функция предназначена для извлечения части данных, которую вы хотите вернуть. Классы данных автоматически определяют эти функции для каждого свойства, объявленного в главном конструкторе.

В деструктуризации нет ничего таинственного: класс данных просто выполняет дополнительную работу, чтобы сделать класс деструктурированным. Вы можете сделать любой класс деструктурированным, добавив функцию-оператор `component`, как в следующем примере:

```
class PlayerScore(val experience: Int, val level: Int) {
    operator fun component1() = experience
    operator fun component2() = level
}
```

```
val (experience, level) = PlayerScore(1250, 5)
```

Объявив `Coordinate` как класс данных, вы сможете получить свойства, определенные в главном конструкторе `Coordinate`, следующим образом:

```
val (x, y) = Coordinate(1, 0)
```

В этом примере `x` имеет значение 1, так как `component1` возвращает значение первого свойства, объявленного в главном конструкторе `Coordinate`, а `y` имеет значение 0, потому что `component2` возвращает второе свойство, объявленное в главном конструкторе `Coordinate`.

Эти особенности подчеркивают важность использования классов данных в представлении простых объектов, предназначенных для хранения данных (как `Coordinate`). Классы, которые часто сравниваются или копируются и содержание которых часто выводится, особенно хорошо подходят для создания классов данных.

Тем не менее у классов данных существует ряд ограничений. Классы данных:

- должны иметь главный конструктор хотя бы с одним параметром;
- требуют, чтобы каждый параметр главного конструктора имел пометку `var` или `val`;
- не могут быть объявлены с ключевыми словами `abstract`, `open`, `sealed`, `inner`.

Если вашему классу не требуются функции `toString`, `copy`, `equals`, `hashCode` или `componentN`, то его объявление как класса данных не даст никаких преимуществ. А если вам нужна своя реализация `equals` (например, использующая только определенные свойства для сравнения, а не все), классы данных вам не подойдут, потому что в автоматически сгенерированную функцию `equals` включаются все свойства.

Мы расскажем о переопределении `equals` и других функций в своих типах позже в этой же главе в разделе «Перегрузка операторов».

Классы-перечисления

Класс-перечисление, или `enum`, — специальный тип класса, у которого все возможные значения *перечисляются* в теле класса.

В `NyetHack` мы используем перечисление для представления множества из четырех доступных для движения игрока направлений, а именно четырех сторон света. Для этого добавим перечисление с именем `Direction` в `Navigation.kt`.

Листинг 16.12. Определение перечисления (`Navigation.kt`)

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

enum class Direction {
    North,
    East,
    South,
    West
}
```

Перечисления более выразительны, чем другие типы констант (например, строки). Для ссылок на значения перечисляемых типов следует указать имя класса перечисления, точку и имя, например, вот так:

```
Direction.EAST
```

Пара слов об именах: имена перечисления обычно записываются по схеме `PascalCase` или только прописными буквами по схеме `SNAKE_CASE`. Оба варианта

допустимы, и вы можете выбрать тот, который вам больше подходит. В частности, разработчики, переходящие на Kotlin с Java, имеют возможность использовать схему `SNAKE_CASE`, которая действует в Java. В этой книге мы будем придерживаться схемы `PascalCase`.

Возможности перечислений не ограничиваются простым объявлением конкретных значений. Чтобы использовать `Direction` для представления направления движения в `NyetHack`, определите для каждого типа в `Direction` соответствующее изменение `Coordinate` при перемещении игрока в этом направлении.

Когда герой передвигается в мире, в соответствии с направлением движения должны изменяться его координаты `x` и `y`. Например, если герой перемещается на восток, координата `x` должна увеличиться на 1, а `y` — на 0. Если игрок перемещается на юг, координата `x` должна увеличиться на 0, а `y` — на 1.

Добавьте в перечисление `Direction` главный конструктор, определяющий свойство `coordinate`. Так как вы добавили параметр в конструктор перечисления, он должен вызываться в объявлении каждого перечисленного типа в `Direction` с передачей соответствующего значения `Coordinate`.

Листинг 16.13. Определение конструктора перечисления (Navigation.kt)

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

enum class Direction(
    private val directionCoordinate: Coordinate
) {
    North(Coordinate(0, -1)),
    East(Coordinate(1, 0)),
    South(Coordinate(0, 1)),
    West(Coordinate(-1, 0))
}
```

Перечисления, как и другие классы, могут содержать объявления функций.

Добавьте в `Direction` функцию с именем `updateCoordinate`, которая будет изменять положение игрока в зависимости от направления его движения (не забудьте добавить точку с запятой, отделяющую объявления значений перечисления от объявления функции).

Листинг 16.14. Определение функции в перечислении (Navigation.kt)

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

enum class Direction(
    private val directionCoordinate: Coordinate
) {
    North(Coordinate(0, -1)),
    East(Coordinate(1, 0)),
    South(Coordinate(0, 1)),
```

```
West(Coordinate(-1, 0));

fun updateCoordinate(coordinate: Coordinate) =
    Coordinate(
        x = coordinate.x + directionCoordinate.x,
        y = coordinate.y + directionCoordinate.y
    )
}
```

Функции должны вызываться для перечисляемых значений, а не для самого класса перечисления, поэтому вызов `updateCoordinate` будет выглядеть так:

```
var currentPosition = Coordinate(5, 2)
currentPosition = Direction.East.updateCoordinate(currentPosition)
```

Перечисления также обладают рядом встроенных полезных возможностей. Как и классы данных, классы перечислений содержат реализации `equals`, `hashCode` и `toString`. Также перечисления содержат два специфических свойства — `name` и `ordinal`.

Свойство перечислений `name` представляет имя элемента перечисления в коде, а `ordinal` содержит целое число (`Int`), задающее его позицию в объявлении перечисления — аналог индекса. Таким образом, для `Direction` у элемента `North` свойство `name` содержит "North", а свойство `ordinal` — 0; у элемента `East` свойство `name` содержит "East", а свойство `ordinal` — 1, и т. д. Будьте внимательны, используя свойства в вашей программе. Очень легко случайно изменить эти значения в ходе рефакторинга объявлений перечислений, что может вызвать нежелательное поведение.

Существуют две функции, которые могут вызываться для самого класса перечисления, — `values` и `valueOf`. `values` возвращает `Array` всех объявленных значений перечисления. `valueOf` возвращает элемент, имя которого совпадает со входным значением. (Если такого элемента не существует, выдается исключение.) Эти две функции можно использовать для поиска конкретных значений при динамическом вводе, чем мы займемся позже.

Перегрузка операторов

Вы уже знаете, что встроенные типы языка Kotlin поддерживают множество операций, а некоторые даже адаптируют эти операции в зависимости от представляемых ими данных. Возьмем для примера функцию `equals` и связанный с ней оператор `==`. С их помощью можно проверить, равны ли два экземпляра числового типа, содержат ли строки одинаковую последовательность символов и являются ли значения свойств двух экземпляров класса данных, объявленных в главном конструкторе, эквивалентными. Аналогично, функция `plus` и оператор `+` складывают два числа, добавляют одну строку в конец другой и добавляют элементы из одного списка в другой.

При работе с пользовательскими типами компилятор Kotlin не всегда знает, как применить к ним встроенные операторы. Например, если мы говорим, что

два экземпляра `Player` равны, то что при этом имеется в виду? Если вы хотите использовать встроенные операторы со своими типами, вам придется переопределить функции-операторы, чтобы подсказать компилятору, как они должны быть реализованы с вашими типами. Этот процесс называется *перегрузкой операторов*.

Преимущества перегрузки операторов были продемонстрированы в главах 9 и 10. Вместо того чтобы вызывать функцию `get` напрямую для извлечения элемента из списка, мы использовали *оператор обращения по индексу* `[]`. Лаконичность синтаксиса Kotlin определяется именно такими маленькими удобствами, как использование `spellList[3]` вместо `spellList.get[3]`.

`Coordinate` — первоочередной кандидат на улучшение через перегрузку операторов. Перемещение героя по миру происходит через сложение свойств двух экземпляров `Coordinate`. Вместо того чтобы определять эту работу в `Direction`, можно перегрузить оператор `plus` для `Coordinate`.

Чтобы реализовать такую возможность в `Navigation.kt`, добавьте функцию с модификатором `operator`.

Листинг 16.15. Перегрузка оператора `plus` (`Navigation.kt`)

```
data class Coordinate(val x: Int, val y: Int) {
    operator fun plus(other: Coordinate) = Coordinate(x + other.x, y + other.y)
}
...
```

Теперь просто используйте оператор сложения (+), чтобы прибавить один экземпляр `Coordinate` к другому. Давайте сделаем это в `Direction`.

Листинг 16.16. Использование перегруженного оператора (`Navigation.kt`)

```
data class Coordinate(val x: Int, val y: Int) {
    operator fun plus(other: Coordinate) = Coordinate(x + other.x, y + other.y)
}

enum class Direction(
    private val directionCoordinate: Coordinate
) {
    North(Coordinate(0, -1)),
    East(Coordinate(1, 0)),
    South(Coordinate(0, 1)),
    West(Coordinate(-1, 0));

    fun updateCoordinate(coordinate: Coordinate) =
        Coordinate(
            x = coordinate.x + directionCoordinate.x,
            y = coordinate.y + directionCoordinate.y
        )
        coordinate + directionCoordinate
}
```

В табл. 16.2 перечислены ряд операторов, которые могут перегружаться в Kotlin.

Таблица 16.2. Распространенные операторы

Оператор	Имя функции	Назначение
+	plus	Складывает два объекта
++	inc	Увеличивает значение объекта на 1
+=	plusAssign	Складывает два объекта и присваивает результат первому
-	minus	Вычитает один объект из другого
--	dec	Уменьшает значение объекта на 1
-=	minusAssign	Вычитает один объект из другого и присваивает результат первому
*	times	Умножает один объект на другой
/	div	Делит один объект на другой
==	equals	Возвращает true, если два объекта равны, и false, если нет
>	compareTo	Возвращает true, если объект слева от оператора больше объекта в правой части, в противном случае возвращает false
[]	get	Возвращает элемент из коллекции по индексу
..	rangeTo	Создает объект, представляющий интервал
in	contains	Возвращает true, если объект присутствует в коллекции
()	invoke	Выполняет функцию, как если бы значение было лямбда-выражением

Эти операторы могут перегружаться в любом классе, но делать это стоит, только если в этом есть смысл. Хотя вы *можете* связать собственную логику с оператором сложения в классе `Player`, задайте себе сначала вопрос: «Как следует понимать “игрок плюс игрок”?» Попробуйте ответить, прежде чем перегружать оператор.

Исследуем мир NyetHack

Теперь, когда вы построили цикл игры и установили систему координат, настало время добавить в NyetHack побольше комнат для исследования.

Для создания карты мира вам понадобится список, включающий все комнаты. Более того, так как игрок может перемещаться в двух измерениях, вам понадобится список, содержащий внутри два других списка — по одному для каждого ряда комнат.

Первый список комнат будет включать, с запада на восток: городскую площадь (начало игры), таверну и заднюю комнату таверны. Второй список комнат будет включать длинный коридор и просто комнату. Третий список содержит подземелье. Эти списки мы сохраним в четвертом списке с именем `worldMap`.

На рис. 16.1 изображен план расположения комнат, или карта мира.

Городская площадь (0,0)	Таверна (1,0)	Задняя комната (2,0)
Длинный коридор (0,1)	Просто комната (1,1)	
Подземелье (0,2)		

Рис. 16.1. Карта мира NyetHack

Добавьте свойство `worldMap` в `Game` с набором комнат для исследования героем.

Листинг 16.17. Определение карты мира в NyetHack (NyetHack.kt)

```
...
object Game {
    private val worldMap = listOf(
        listOf(TownSquare(), Tavern(), Room("Back Room")),
        listOf(Room("A Long Corridor"), Room("A Generic Room")),
        listOf(Room("The Dungeon"))
    )

    private var currentRoom: Room = TownSquare()
    private var currentPosition = Coordinate(0, 0)
    ...
}
```

Теперь комнаты находятся на своих местах, и настало время добавить команду перемещения и дать игроку возможность двигаться в мире NyetHack. Добавьте функцию с именем `move`, которая получает аргумент `Direction` и обновляет `currentRoom` и `currentPosition`, если игрок может передвигаться в этом направлении.

Листинг 16.18. Определение функции `move` (NyetHack.kt)

```

...
object Game {
    ...
    fun play() {
        while (true) {
            narrate("${player.name} of ${player.hometown}, ${player.title},
                is in ${currentRoom.description()}")
            currentRoom.enterRoom()

            print("> Enter your command: ")
            GameInput(readLine()).processCommand()
        }
    }

    fun move(direction: Direction) {
        val newPosition = direction.updateCoordinate(currentPosition)
        val newRoom = worldMap.getOrNull(newPosition.y)?.getOrNull(newPosition.x)

        if (newRoom != null) {
            narrate("The hero moves ${direction.name}")
            currentPosition = newPosition
            currentRoom = newRoom
        } else {
            narrate("You cannot move ${direction.name}")
        }
    }
    ...
}

```

Функция `getOrNull` используется для определения того, соответствуют ли координаты комнате на карте. Если игрок пытается выйти за пределы карты, поиск вернет `null` и попытка перемещения будет отклонена.

Также обратите внимание на то, что в коде используется свойство перечисления `name`. Как правило, это значение не предназначено для вывода в консоль, потому что имена перечисления не всегда понятны для пользователя. Такое использование имен также может усложнить перевод программы на другие языки. Но для простоты мы пока будем использовать `name`.

В более сложных проектах мы рекомендуем добавить в перечисление `Direction` свойство `directionName`. Если ваша программа также потребует интернационализации, вы можете реализовать нестандартное свойство с использованием локализационных API вашей платформы, чтобы получать правильную строку в зависимости от местоположения пользователя.

Функция `move` должна вызываться при вводе игроком команды «move» («идти»). Вы реализуете это поведение с помощью класса `GameInput`, который мы написали ранее в этой главе:

Листинг 16.19. Реализация команды move (NyetHack.kt)

```

...
object Game {
    ...
    private class GameInput(arg: String?) {
        private val input = arg ?: ""
        val command = input.split(" ")[0]
        val argument = input.split(" ").getOrElse(1) { "" }

        fun processCommand() = when (command.lowercase()) {
            "move" -> {
                val direction = Direction.values()
                    .firstOrNull { it.name.equals(argument, ignoreCase = true) }
                if (direction != null) {
                    move(direction)
                } else {
                    narrate("I don't know what direction that is")
                }
            }
            else -> narrate("I'm not sure what you're trying to do")
        }
    }
}

```

Выполнение кода начинается с преобразования аргумента в `Direction`. Мы используем функцию `values` класса перечисления и находим соответствующее значение `Direction` (без учета регистра). Если направление с подходящим именем не найдено, поиск вернет `null` и рассказчик выразит свое недоумение. В противном случае вы получаете экземпляр `Direction`, который может использоваться при вызове `move`.

Попробуйте запустить `NyetHack` и переместиться. Результат будет выглядеть примерно так:

```

...
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command: move east
The hero moves East
Madrigal of Neversummer, The Renowned Hero, is in Taernyl's Folly
(Currently: Busy)
Madrigal enters Taernyl's Folly
...

```

Вот и все. Теперь вы можете двигаться в мире `NyetHack`. В этой главе вы научились пользоваться некоторыми разновидностями классов. Кроме ключевого слова `class`, для представления данных вы можете использовать объявления

объектов (синглтоны), классы данных и перечисления. Выбор правильных средств делает отношения между объектами более ясными.

В следующей главе мы рассмотрим интерфейсы и абстрактные классы (то есть механизмы объявления протоколов, которые должны будут поддерживаться вашими классами), когда вы добавите в `NyetHack` возможность битвы.

Для любознательных: объявление структурного сравнения

Представим класс `Weapon`, который обладает свойствами `name` и `type`:

```
open class Weapon(val name: String, val type: String)
```

Предположим, вы хотите, чтобы оператор равенства (`==`) считал два разных экземпляра оружия структурно эквивалентными, если значения их свойств `name` и `type` структурно равны. По умолчанию, как было сказано ранее в этой главе, оператор `==` проверяет равенство ссылок, поэтому следующее выражение вернет `false`:

```
open class Weapon(val name: String, val type: String)

println(Weapon("ebony kris", "dagger") == Weapon("ebony kris", "dagger")) //
False
```

Из этой главы вы узнали, что классы данных могут решить эту проблему. Для этого нужна реализация `equals`, которая принимает решения о равенстве на основании свойств, объявленных в главном конструкторе. Но класс `Weapon` не является (и не может быть) классом данных, потому что это базовый класс для видов оружия (модификатор `open`). Классы данных не могут быть суперклассами.

Тем не менее, как упоминалось в разделе «Перегрузка операторов», вы можете предоставить собственные реализации `equals` и `hashCode` для структурного сравнения экземпляров классов.

Эта задача встречается настолько часто, что в IntelliJ есть операция `Generate`, добавляющая переопределения функций; она вызывается командой меню `Code ▸ Generate (Command-N [Alt-Insert])`. При выборе этой команды на экране появляется диалоговое окно `Generate` (рис. 16.2).

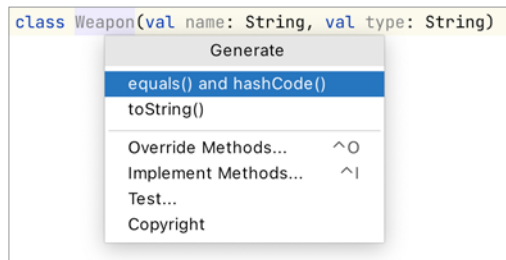


Рис. 16.2. Диалоговое окно `Generate`

В диалоговом окне выберите `equals()` и `hashCode()`.

При генерировании функций `equals` и `hashCode` можно указать свойства, которые будут использоваться для структурного сравнения двух экземпляров вашего объекта (рис. 16.3). Установите флажки `name` и `type`.

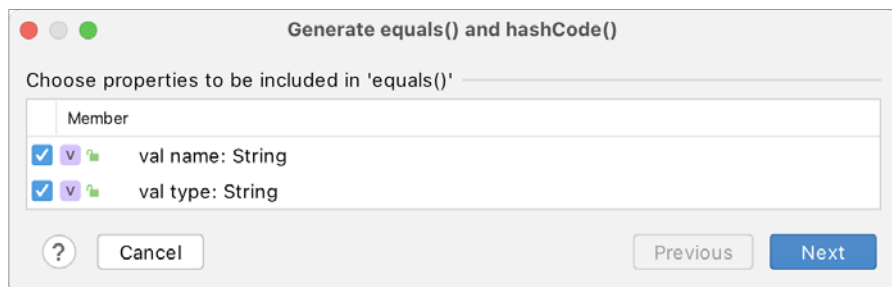


Рис. 16.3. Выбор свойств, включаемых в переопределение `equals`

IntelliJ добавляет в класс функции `equals` и `hashCode`, основываясь на вашем выборе:

```
open class Weapon(val name: String, val type: String) {
    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other !is Weapon) return false

        other as Weapon

        if (name != other.name) return false
        if (type != other.type) return false

        return true
    }
    override fun hashCode(): Int {
        var result = name.hashCode()
        result = 31 * result + type.hashCode()
        return result
    }
}
```

Переопределенная функция `equals`, которая была сгенерирована IntelliJ, выполняет структурное сравнение свойств, выбранных в команде **Generate**. Если какие-либо из свойств не будут структурно равны, то сравнение дает результат `false`. В противном случае возвращается `true`.

При наличии этих переопределений сравнение двух видов оружия дает результат `true`, если совпадают значения их свойств `name` и `type`:

```
Weapon("Mjolnir", "hammer") == Weapon("Mjolnir", "hammer") // True
```

Как упоминалось ранее, при переопределении функции `equals` также необходимо предоставить соответствующее переопределение `hashCode`. Как правило, свойства, используемые в `equals`, должны хешироваться совместно для формирования этого хеш-кода. Среда IntelliJ выполняет хеширование за вас, для чего она берет хеш-код каждого свойства, умножает его на произвольное простое число и вычисляет сумму произведений.

Для любознательных: алгебраические типы данных

Алгебраические типы данных (Algebraic Data Types, ADT) позволяют представлять закрытое множество возможных подтипов, которые могут быть ассоциированы с заданным типом. Перечисления являются простейшей формой ADT.

Представьте класс `Student`, который имеет три ассоциативных состояния, зависящих от статуса зачисления: `NotEnrolled` (не зачислен), `Active` (активный) и `Graduated` (выпущен).

Используя класс перечисления, с которым вы только что познакомились, можно смоделировать эти три состояния для класса `Student` следующим образом:

```
enum class StudentStatus {
    NotEnrolled,
    Active,
    Graduated
}

class Student(var status: StudentStatus)

fun main() {
    val student = Student(StudentStatus.NotEnrolled)
}
```

Также можно написать функцию, которая генерирует сообщение о статусе студента:

```
fun studentMessage(status: StudentStatus): String {
    return when (status) {
        StudentStatus.NotEnrolled -> "Please choose a course."
    }
}
```

Одно из преимуществ перечислений и других ADT заключается в том, что компилятор способен проверить, обработаны ли все возможные варианты, потому что ADT — это закрытое множество всех возможных типов. Реализация `studentMessage` не обрабатывает типы `Active` и `Graduated`, поэтому компилятор выдаст ошибку (рис. 16.4).

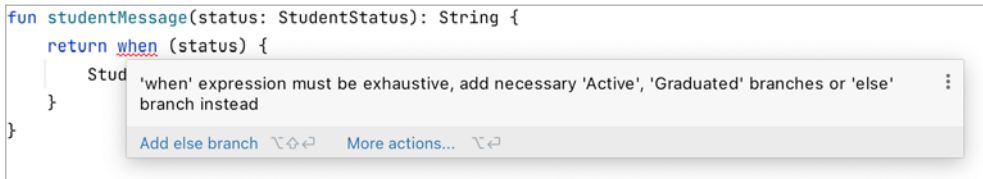


Рис. 16.4. Добавление необходимых вариантов

Компилятор будет удовлетворен, если в коде присутствуют явные обращения ко всем типам либо они указаны в ветви `else`:

```
fun studentMessage(status: StudentStatus): String {
    return when (status) {
        StudentStatus.NotEnrolled -> "Please choose a course."
        StudentStatus.Active -> "Welcome, student!"
        StudentStatus.Graduated -> "Congratulations!"
    }
}
```

Для более сложных ADT можно использовать *изолированные* (sealed) классы, которые позволяют реализовать более хитроумные объявления. Изолированные классы позволяют определить ADT, похожие на перечисления, но с возможностью большего контроля в отношении конкретных подтипов.

Допустим, активному студенту должен быть назначен идентификатор курса. Можно добавить соответствующее свойство в определение перечисления, например, так:

```
enum class StudentStatus {
    NotEnrolled,
    Active,
    Graduated;
    var courseId: String? = null // Используется только для Active
}
```

У такого решения два недостатка. Так как свойство используется только в варианте `Active`, в других вариантах появляются два ненужных состояния `null`. Кроме того, Kotlin создает только один экземпляр каждого перечисляемого значения, которое совместно используется в программах, — по аналогии с синглтоном `Game` (его вы создали при объявлении объекта). Если вы настроите свою программу подобным образом, все студенты будут совместно использовать один экземпляр `Active`. Если все студенты не посещают один и тот же курс (или у вас только один студент), такое представление работать не будет.

Для моделирования статусов студентов лучше использовать изолированный класс:

```
sealed class StudentStatus {  
    object NotEnrolled : StudentStatus()  
    data class Active(val courseId: String) : StudentStatus()  
    object Graduated : StudentStatus()  
}
```

Изолированный класс `StudentStatus` имеет ограниченное количество подклассов. Изолированные классы можно использовать для создания подклассов только другими классами, объявленными в том же пакете и кодовой базе, что и определение самого изолированного класса.

Из-за ограничений на расширение изолированных классов компилятору Kotlin известны все возможные реализации на стадии компиляции. Это позволяет ему проверить полноту выражения `when` без обязательной ветви `else` — так же, как для перечислений.

Ключевое слово `object` используется для состояний, не содержащих идентификатор курса, так как у этих состояний нет никаких вариаций. С другой стороны, статус `Active` определяется как обычный класс (а конкретно — класс данных), потому что у него могут быть другие экземпляры: идентификатор курса изменяется для каждого студента.

Использование нового изолированного класса в `when` позволит вам прочесть значение `courseID` из класса `Active` через умное приведение типа:

```
fun main() {  
    val student = Student(StudentStatus.Active("Kotlin101"))  
    studentMessage(student.status)  
}  
fun studentMessage(status: StudentStatus): String {  
    return when (status) {  
        is StudentStatus.NotEnrolled -> "Please choose a course!"  
        is StudentStatus.Active -> "You are enrolled in: ${status.courseId}"  
        is StudentStatus.Graduated -> "Congratulations!"  
    }  
}
```

Для любознательных: классы-значения

Кроме разновидностей классов, описанных в этой главе, в Kotlin существуют *классы-значения*. Классы-значения удобны, когда вы хотите создать новый класс, действующий как другая интерпретация существующего типа. Допустим, вам требуется несколько единиц для измерения расстояний на карте NyetHack. Можно воспользоваться классами-значениями и определить типы для миль и километров:

```
@JvmInline  
value class Kilometers(private val kilometers: Double) {  
    operator fun plus(other: Kilometers) =  
        Kilometers(kilometers + other.kilometers)
```

```
        fun toMiles() = kilometers / 1.609
    }
    @JvmInline
    value class Miles(private val miles: Double) {
        operator fun plus(other: Miles) =
            Miles(miles + other.miles)

        fun toKilometers() = miles * 1.609
    }
```

Иметь отдельный тип для конкретной единицы измерения очень удобно, потому что у вас появляется возможность кодировать единицы в типах вашей программы. Кроме того, вы таким образом предотвращаете возможность случайного суммирования миль с километрами. Но у таких классов есть недостаток: создание экземпляра любого класса сопряжено с лишними затратами памяти. Классы-значения позволяют обойти эту проблему.

Когда вы используете классы-значения в своем коде, они заменяются типом, который они инкапсулируют (в данном случае `Double`). После этого все функции класса (такие, как `plus`, `toMiles` и `toKilometers`) компилируются в статические функции, которые могут вызываться без экземпляра класса-значения.

В результате вы получаете безопасность типов, присущую объявлению нестандартных классов-обертки для существующих типов, без дополнительных затрат. Впрочем, у классов-значений есть свои ограничения.

- Главный конструктор класса-значения должен получать только один аргумент, который непременно объявляется как `val`-свойство. Это значение будет встраиваться в каждой точке использования класса.
- Классы-значения не могут объявлять дополнительные свойства, не имеющие резервных полей. Тем не менее разрешены вычисляемые свойства `var` или `val`.
- Классы-значения не могут переопределять `equals` или `hashCode`. Kotlin использует реализации из инкапсулированного значения.
- Классы-значения не могут помечаться модификатором `open` и наследовать от другого класса. (Но они способны реализовать интерфейсы, о которых мы расскажем в следующей главе.)

Задание: другие команды

С изменениями, внесенными в этой главе, герой уже не может творить заклинания или предугадывать судьбу. Чтобы исправить этот недостаток, добавьте две новые команды, `cast fireball` и `prophesize`, которые будут вызывать соответствующие функции `Player`.

Кроме того, игроки, скорее всего, в какой-то момент захотят завершить игру NyetHack, но сейчас это сделать невозможно. Добавьте еще одну команду, которая активизируется при вводе строки «quit» или «exit», чтобы игра NyetHack выводила прощальное сообщение и завершала работу. Подсказка: вспомните, что на данный момент ваш цикл `while` выполняется бесконечно. Суть задачи в том, чтобы завершать цикл при определенном условии.

Задание: реализация карты мира

Вы помните, что в самом начале мы говорили: NyetHack не будет содержать ASCII-графику? После завершения этого задания — будет!

Игроки иногда могут заблудиться в обширном мире NyetHack, но, к счастью, вы можете подарить им волшебную карту королевства. Реализуйте команду `map`, которая выводит текущее положение игрока в мире. Если игрок находится в таверне, вывод должен быть примерно таким:

```
> Enter your command: map
0 X 0
0 0
0
```

X обозначает комнату, в которой сейчас находится игрок.

Задание: колокольный звон

Добавьте команду «ring» в NyetHack, чтобы игрок мог ударить в колокол на городской площади любое количество раз.

Подсказка: объявите функцию `ringBell` общедоступной (`public`).

Подписывайся на самый топовый канал по Kotlin:
<https://t.me/KotlinSenior>

17. Интерфейсы и абстрактные классы

В этой главе вы научитесь объявлять и использовать *интерфейсы* и *абстрактные классы* Kotlin.

Интерфейс позволяет перечислить общие свойства и поведение, которые должны поддерживаться подмножеством классов в программе, но без указания того, как они должны быть реализованы. Эта особенность — *что без как* — полезна, если наследование неточно отражает отношения между классами. Используя интерфейсы, группа классов может иметь общие свойства и функции без наследования от общего суперкласса, причем классы в группе не являются подклассами друг друга.

Вы также познакомитесь с абстрактными классами — гибридом между классами и интерфейсами. Абстрактные классы похожи на интерфейсы: они позволяют определить *что без как*, но объявляют конструкторы и способны выполнять роль суперкласса.

Эти новые идеи позволят добавить интересную возможность в NyetHack: теперь, когда наш герой может перемещаться, мы добавим систему ведения боя, чтобы герой получил возможность расправиться со злодеями, которых он встретит в ходе игры.

Определение интерфейса

Чтобы определить, как должно протекать сражение, сначала создадим интерфейс с перечнем функций и свойств игровых сущностей, которые могут участвовать в схватках. Игрок будет сражаться с гоблинами, но наша система боя будет годиться для любых противников.

Создайте новый файл с именем `Creature.kt` в пакете `com.bignerdranch.nyethack` (для предотвращения конфликтов имен) и определите интерфейс `Fightable` с использованием ключевого слова `interface`.

Листинг 17.1. Определение интерфейса (`Creature.kt`)

```
interface Fightable {  
    val name: String  
    var healthPoints: Int
```

```

val diceCount: Int
val diceSides: Int

fun takeDamage(damage: Int)

fun attack(opponent: Fightable)
}

```

Объявление интерфейса определяет общие черты сущностей, которые могут принимать участие в схватках в NyetHack. Существа, способные сражаться, используют несколько игровых костей, чтобы определить величину ущерба (сумма очков, выпавших на всех игровых костях), нанесенного противнику. Такие существа также должны иметь имя, очки здоровья `healthPoints` и реализацию двух функций: `takeDamage` и `attack`.

Четыре свойства `Fightable` не имеют инициализаторов, а функции `takeDamage` и `attack` не имеют тела. Интерфейс не предоставляет инициализаторы или тела функций. Запомните: интерфейс определяет «что», а не «как».

Итак, интерфейс `Fightable` используется как тип параметра `opponent`, который получает функция `attack`. Интерфейс, как и класс, можно использовать в качестве типа параметра.

Для функции, указывающей тип параметра, важно, что аргумент может делать, а не как его поведение реализовано. Это одна из сильных сторон интерфейса: можно определить набор требований, общих для классов, которые в остальном не имеют ничего общего.

Реализация интерфейса

Чтобы использовать интерфейс, необходимо реализовать его в классе. Для этого нужно, во-первых, объявить, что класс реализует интерфейс, и, во-вторых, предоставить реализации всех свойств и функций, указанных в интерфейсе.

Чтобы объявить, что класс `Player` реализует интерфейс `Fightable`, используйте оператор `:`, как показано в листинге 17.2.

Листинг 17.2. Реализация интерфейса (Player.kt)

```

class Player(
    initialName: String,
    val hometown: String = "Neversummer",
    override var healthPoints: Int,
    val isImmortal: Boolean
) : Fightable {

    override var name = initialName
    get() = field.replaceFirstChar { it.uppercaseChar() }
    private set(value) {

```

```
        field = value.trim()
    }
    ...
}
```

(Смысл добавленных ключевых слов `override` мы объясним далее.)

Когда вы добавляете интерфейс `Fightable` в `Player`, IntelliJ сообщает об отсутствии свойств и функций. Предупреждение об отсутствии реализации функций и свойств в `Player` поможет вам соблюсти правила `Fightable`, а IntelliJ поможет реализовать все необходимое.

Щелкните правой кнопкой мыши на `Player` и выберите `Generate... ▸ Implement Methods...`, а затем в диалоговом окне `Implement Members` (рис. 17.1) выберите `diceCount`, `diceSides` и `takeDamage`. (Об `attack` речь пойдет в следующем разделе.)

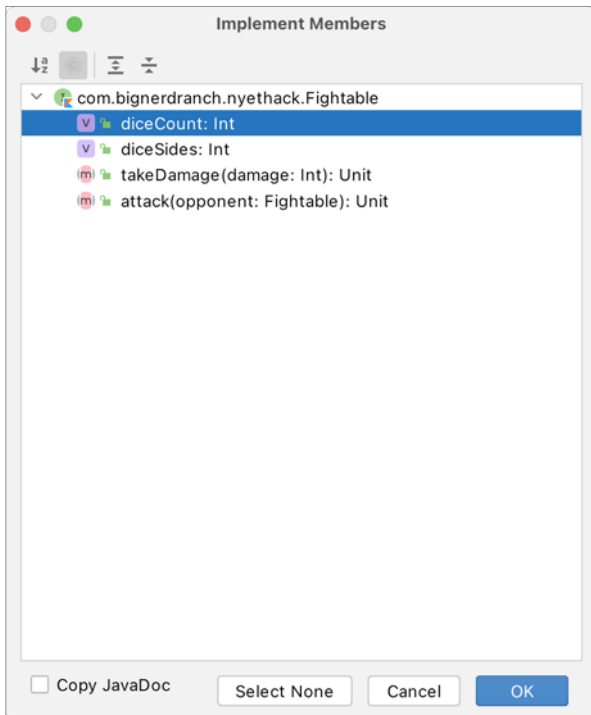


Рис. 17.1. Реализация членов `Fightable`

После того как IntelliJ сгенерирует реализации для класса `Player`, переставьте их так, как показано в листинге 17.3, чтобы сгруппировать свойства с функциями.

Листинг 17.3. Генерирование заглушек (Player.kt)

```
class Player(  
    initialName: String,  
    val hometown: String = "Neversummer",  
    override var healthPoints: Int,  
    val isImmortal: Boolean  
) : Fightable {  
    ...  
    val prophecy by lazy {  
        ...  
    }  
  
    override val diceCount: Int  
        get() = TODO("Not yet implemented")  
  
    override val diceSides: Int  
        get() = TODO("Not yet implemented")  
  
    init {  
        require(healthPoints > 0) { "healthPoints must be greater than zero" }  
        require(initialName.isNotBlank()) { "Player must have a name" }  
    }  
    ...  
    fun prophesize() {  
        narrate("$name thinks about their future")  
        narrate("A fortune teller told Madrigal, \"$prophecy\"")  
    }  
  
    override fun takeDamage(damage: Int) {  
        TODO("Not yet implemented")  
    }  
}
```

Реализации функций, добавленных в `Player`, пока представляют собой простые заглушки. Вам нужно заменить их настоящими реализациями. (Кстати, обратите внимание на функцию `TODO`, знакомую вам по обсуждению типа `Nothing` в главе 4. Тут она показана в действии.) Как только вы реализуете `diceCount`, `diceSides` и `takeDamage` (а также позаботитесь об `attack`), `Player` будет удовлетворять интерфейс `Fightable` и сможет участвовать в сражениях.

Обратите внимание, что во всех реализациях свойств и функций, включая `healthPoints` и `name`, используется ключевое слово `override`. Это может стать неожиданностью: все-таки вы не заменяете реализацию этих свойств в `Fightable`. Тем не менее все реализации свойств и функций интерфейса должны быть отмечены словом `override`.

С другой стороны, ключевое слово `open` не требуется при объявлении функций в интерфейсе. Это связано с тем, что все свойства и функции, добавленные в ин-

терфейс, должны неявно иметь модификатор доступа `open`, иначе их реализация не будет иметь никакого смысла. В конце концов, интерфейс определяет, что нужно делать, а как — это уже ответственность классов, реализующих его.

Замените вызовы `TODO` в `diceCount`, `diceSides` и `takeDamage` соответствующими значениями и функциями.

Листинг 17.4. Реализация интерфейса (Player.kt)

```
class Player(  
    initialName: String,  
    val hometown: String = "Neversummer",  
    override var healthPoints: Int,  
    val isImmortal: Boolean  
) : Fightable {  
    ...  
    override val diceCount: Int = 3  
        get() = TODO("Not yet implemented")  
    override val diceSides: Int = 4  
        get() = TODO("Not yet implemented")  
    ...  
    override fun takeDamage(damage: Int) {  
        TODO("Not yet implemented")  
        if (!isImmortal) {  
            healthPoints -= damage  
        }  
    }  
}
```

`diceCount` и `diceSides` реализуются целыми числами. Функция `takeDamage` уменьшает здоровье игрока на количество повреждений, выпавших при броске костей, только если игрок не является бессмертным (в этом случае ему нельзя причинить урон обычными средствами). `takeDamage` будет вызываться функцией `attack`, которую мы зададим в следующем разделе.

Реализация по умолчанию

Мы уже неоднократно говорили: интерфейсы определяют, что надо реализовать, а не как. Тем не менее есть возможность предоставить реализацию по умолчанию для методов чтения свойств и функций в интерфейсе. Классы, определяющие интерфейс, могут использовать реализацию по умолчанию или определить свою.

Предоставьте реализацию по умолчанию для `attack` в `Fightable`. Она должна вычислять сумму очков, выпавших на всех игровых костях, и наносить соответствующий урон противнику.

Листинг 17.5. Определение реализации по умолчанию (Creature.kt)

```
import kotlin.random.Random

interface Fightable {
    val name: String
    var healthPoints: Int
    val diceCount: Int
    val diceSides: Int

    fun takeDamage(damage: Int)

    fun attack(opponent: Fightable) {
        val damageRoll = (0 until diceCount).sumOf {
            Random.nextInt(diceSides + 1)
        }
        narrate("$name deals $damageRoll to ${opponent.name}")
        opponent.takeDamage(damageRoll)
    }
}
```

Теперь, когда `attack` имеет метод чтения по умолчанию, любой класс, реализующий интерфейс `Fightable`, может отказаться от своего определения функции `attack`. Все ошибки исчезают из проекта; `Player` теперь полностью реализует `Fightable`. Запустите программу и убедитесь в том, что она работает, как и прежде.

Вы также можете предоставить реализации по умолчанию для свойств из интерфейса, но они ограничиваются вычисляемыми свойствами — интерфейсу не разрешается выделять память для резервных полей. Не каждому свойству или функции нужна уникальная реализация в каждом классе, поэтому определение реализации по умолчанию — это хороший способ убрать дубликаты из кода.

Абстрактные классы

Абстрактные классы предоставляют еще один способ организации классов. Экземпляры абстрактных классов никогда не создаются напрямую. Их цель — предложить реализации функций через наследование подклассам, которые *могут* иметь экземпляры.

Абстрактный класс объявляется присоединением ключевого слова `abstract` в начало определения класса. Кроме реализованных функций, абстрактные классы могут включать *абстрактные функции* — объявления функций без реализации.

Настало время задать для игрока противника, с которым он будет сражаться в `NyetHack`. Добавьте абстрактный класс с именем `Monster` в `Creature.kt`. `Monster` реализует интерфейс `Fightable`, а это значит, что он должен иметь свойство `healthPoints` и функцию `takeDamage`. («А как же другие свойства `Fightable`?» — спросите вы. Совсем скоро мы к ним вернемся.)

Листинг 17.6. Определение абстрактного класса (Creature.kt)

```
interface Fightable {
    val name: String
    var healthPoints: Int
    val diceCount: Int
    val diceSides: Int

    fun takeDamage(damage: Int)

    fun attack(opponent: Fightable) {
        val damageRoll = (0 until diceCount).sumOf {
            Random.nextInt(diceSides + 1)
        }
        narrate("$name deals $damageRoll to ${opponent.name}")
        opponent.takeDamage(damageRoll)
    }
}

abstract class Monster(
    override val name: String,
    val description: String,
    override var healthPoints: Int
) : Fightable {
    override fun takeDamage(damage: Int) {
        healthPoints -= damage
    }
}
```

Класс `Monster` объявляется как абстрактный, поскольку он станет базисным для создания конкретных монстров в игре. Экземпляры `Monster` никогда создаваться не будут, да это и невозможно. Вместо этого создаются экземпляры подклассов `Monster`: конкретные существа — гоблины, призраки, драконы, то есть разные воплощения абстрактного монстра.

Снова присмотритесь к конструктору `Monster`. Свойства `name` и `healthPoints` определяются в конструкторе с ключевым словом `override`. Эти два свойства наследуются от интерфейса `Fightable`, и класс `Monster` может обращаться к ним независимо от того, присутствуют ли они в самом классе. Эти свойства можно не включать в класс `Monster`, но их объявление имеет два значительных преимущества.

Первое преимущество объявления переопределяемых свойств в конструкторе в том, что подклассы класса `Monster` теперь могут реализовать свойства, передавая значение конструктору вместо объявления собственных переопределений свойств. Это позволяет использовать чуть более компактный синтаксис в подклассах.

Второе преимущество не столь очевидно. Когда-нибудь в будущем герой может заключить мир с монстрами, и вы захотите обновить свой класс `Monster`, чтобы он не реализовал интерфейс `Fightable`. Тем не менее знать имя и количество очков здоровья монстра все равно полезно, даже если сейчас вы с ним дружите. В таком

сценарии монстры по-прежнему будут обладать этими свойствами, потому что они объявлены в самом классе. (Только не забудьте удалить ключевое слово `override`.)

Определив `Monster` как абстрактный класс, мы получили шаблон для создания любых монстров в `NyetHack`: монстр должен обладать именем и описанием, а также — пока! — удовлетворять критериям интерфейса `Fightable`.

Теперь создадим в `Creature.kt` первую конкретную версию абстрактного класса `Monster` — подкласс `Goblin`.

Листинг 17.7. Создание подклассов из абстрактного класса (`Creature.kt`)

```
interface Fightable {  
    ...  
}  
  
abstract class Monster(  
    override val name: String,  
    val description: String,  
    override var healthPoints: Int  
) : Fightable {  
    override fun takeDamage(damage: Int) {  
        healthPoints -= damage  
    }  
}  
  
class Goblin(  
    name: String = "Goblin",  
    description: String = "A nasty-looking goblin",  
    healthPoints: Int = 30  
) : Monster(name, description, healthPoints)
```

Так как `Goblin` — это подкласс `Monster`, он обладает всеми свойствами и функциями, которые есть у `Monster`.

Если вы попытаетесь скомпилировать код сейчас, то это не удастся, потому что свойства `diceCount` и `diceSides` объявлены в интерфейсе `Fightable`, но не реализованы в `Goblin` (и не имеют реализации по умолчанию).

Но `Goblin` не реализует интерфейс `Fightable` явно, так почему же он должен реализовать его требования?

Подкласс по умолчанию разделяет всю функциональность со своим суперклассом. Это утверждение истинно независимо от того, какой разновидностью класса является суперкласс. Если класс реализует интерфейс, то его подкласс также должен удовлетворять требованиям интерфейса.

`Monster` не обязан отвечать всем требованиям интерфейса `Fightable`, несмотря на то что реализует его, потому что это абстрактный класс и он никогда не будет иметь экземпляров. Но его подклассы должны реализовать все требования `Fightable` либо через наследование от `Monster`, либо сами по себе.

Удовлетворите требования `Fightable`, добавив недостающие свойства в `Goblin`.

Листинг 17.8. Реализация свойств в подклассе абстрактного класса (Creature.kt)

```
interface Fightable {
    ...
}

abstract class Monster(
    override val name: String,
    val description: String,
    override var healthPoints: Int
) : Fightable {
    ...
}

class Goblin(
    name: String = "Goblin",
    description: String = "A nasty-looking goblin",
    healthPoints: Int = 30
) : Monster(name, description, healthPoints) {
    override val diceCount = 2
    override val diceSides = 8
}
```

Снова запустите свою программу и убедитесь в том, что она компилируется, как и ожидалось.

Возможно, вы заметили сходство между абстрактными классами и интерфейсами: и те и другие способны объявлять функции и свойства без реализации. В чем же тогда разница?

Во-первых, интерфейс не может определить конструктор. Во-вторых, интерфейсы не могут запретить наследникам переопределять поведение по умолчанию, и у них есть ограничения относительно того, какие свойства и функции могут помечаться `private`. В-третьих, класс (или подкласс) имеет возможность *расширять* только один абстрактный класс, но он способен реализовать несколько интерфейсов.

Хорошее правило: если нужна категория поведения или свойств, общая для объектов, но неудобная для наследования, используйте интерфейс. Если наследование имеет смысл, но вам не нужен конкретный класс-предок, используйте абстрактный класс. (А если вы хотите создавать экземпляр класса-предка, то лучше использовать обычный класс.)

Сражение в NyetHack

Для добавления сражений в NyetHack придется использовать все наши знания об объектно-ориентированном программировании.

В некоторых комнатах NyetHack героя будет ждать монстр, которого удастся победить наиболее жестоким способом из всех возможных: превращением его в `null`.

Создайте новую разновидность `MonsterRoom` для определения комнаты, в которой может находиться монстр. Добавьте в новый класс `MonsterRoom` свойство `monster` с типом `Monster?`, допускающим `null`, и инициализируйте его экземпляром `Goblin`. Также обновите описание `Room`, чтобы игрок знал, есть ли в комнате монстр, жаждущий сражения.

Листинг 17.9. Определение комнат с монстрами (Room.kt)

```
open class Room(val name: String) {

    protected open val status = "Calm"

    open fun description() = "$name (Currently: $status)"

    open fun enterRoom() {
        narrate("There is nothing to do here")
    }
}

open class MonsterRoom(
    name: String,
    var monster: Monster? = Goblin()
) : Room(name) {

    override fun description() =
        super.description() + " (Creature: ${monster?.description ?: "None"})"

    override fun enterRoom() {
        if (monster == null) {
            super.enterRoom()
        } else {
            narrate("Danger is lurking in this room")
        }
    }
}
```

Обратите внимание на вызовы `super.description()` и `super.enterRoom()`. Ключевое слово `super` используется для вызова суперкласса и обращения к непереопределенному поведению функции или свойства. Здесь оно используется для вызова реализаций `description` и `enterRoom` класса `Room` и построения функциональности на их основе при реализации `MonsterRoom`.

Новый класс `MonsterRoom` хранит информацию о притаившихся чудовищах в свойстве с именем `monster`. Если свойство содержит `null`, значит, монстр побежден. В противном случае герою еще предстоит сражение.

Вы инициализировали `monster`, свойство типа `Monster?`, создав объект типа `Goblin`. Комната может содержать экземпляр любого подкласса `Monster`, а `Goblin` является подклассом `Monster`. Это пример полиморфизма. Если создать еще

один класс, наследующий от `Monster`, его также можно использовать в комнатах `NyetHack`.

Чтобы воспользоваться новым типом `MonsterRoom`, необходимо стратегически изменить свойства некоторых комнат в `NyetHack`. Обновите свойство `worldMap` в `Game`, чтобы добавить врагов в наиболее подозрительных частях города.

Листинг 17.10. Размещение монстров (NyetHack.kt)

```
...
object Game {
    private val worldMap = listOf(
        listOf(TownSquare(), Tavern(), Room("Back Room")),
        listOf(MonsterRoom("A Long Corridor"), Room("A Generic Room")),
        listOf(MonsterRoom("The Dungeon"))
    )
    ...
}
```

После размещения монстров добавьте в `Game` функцию с именем `fight`.

Листинг 17.11. Определение функции `fight` (NyetHack.kt)

```
...
object Game {
    ...
    fun move(direction: Direction) {
        ...
    }

    fun fight() {
        val monsterRoom = currentRoom as? MonsterRoom
        val currentMonster = monsterRoom?.monster
        if (currentMonster == null) {
            narrate("There's nothing to fight here")
            return
        }

        while (player.healthPoints > 0 && currentMonster.healthPoints > 0) {
            player.attack(currentMonster)
            if (currentMonster.healthPoints > 0) {
                currentMonster.attack(player)
            }
            Thread.sleep(1000)
        }

        if (player.healthPoints <= 0) {
            narrate("You have been defeated! Thanks for playing")
            exitProcess(0)
        } else {
            narrate("${currentMonster.name} has been defeated")
        }
    }
}
```

```

        monsterRoom.monster = null
    }
}

private class GameInput(arg: String?) {
    ...
}

```

Прежде всего `fight` проверяет значение `monster`. Если оно равно `null`, то сражаться не с кем, и тогда выводится соответствующее сообщение. Если монстр присутствует в комнате и у игрока и монстра есть хотя бы по одному очку здоровья, проводится раунд сражения.

Если значение `healthPoints` игрока падает до нуля, игра завершается, для чего вызывается функция `exitProcess` — функция стандартной библиотеки Kotlin, которая завершает выполняемый экземпляр вашей программы. (Функция доступна в Kotlin/JVM и Kotlin/Native, но не Kotlin/JS.) Чтобы получить доступ к этой функции, необходимо импортировать `kotlin.system.exitProcess`.

Если значение `healthPoints` монстра падает до нуля, то монстр погибает, остается только `null`.

В каждом раунде сражения функция `attack` вызывается для монстра и для игрока. Одна и та же функция `attack` может вызываться как для `Monster`, так и для `Player`, потому что оба класса реализуют интерфейс `Fightable`.

Чтобы протестировать новую систему сражений, добавьте в `GameInput` команду `fight`, которая вызывает функцию `fight`.

Листинг 17.12. Добавление команды `fight` (NyetHack.kt)

```

...
object Game {
    ...
    private class GameInput(arg: String?) {
        private val input = arg ?: ""
        val command = input.split(" ")[0]
        val argument = input.split(" ").getOrElse(1) { "" }

        fun processCommand() = when (command.lowercase()) {
            "fight" -> fight()
            "move" -> {
                val direction = Direction.values()
                    .firstOrNull { it.name.equals(argument, ignoreCase = true) }
                if (direction != null) {
                    move(direction)
                } else {
                    narrate("I don't know what direction that is")
                }
            }
        }
    }
}

```

```
        else -> narrate("I'm not sure what you're trying to do")
    }
}
```

Запустите `NyetHack.kt`. Попробуйте переместиться на юг в длинный коридор, начните сражение командой `fight`. Случайность, введенная в функцию `attack` в интерфейсе `Fightable`, означает, что каждый раз, когда вы переходите в новую комнату и начинаете сражение, вы будете получать разные результаты.

```
Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Madrigal
Welcome, adventurer
Madrigal, a mortal, has 100 health points
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command: move south
The hero moves South
Madrigal of Neversummer, The Renowned Hero, is in A Long Corridor
(Currently: Calm) (Creature: A nasty-looking goblin)
Danger is lurking in this room
> Enter your command: fight
Madrigal deals 9 to Goblin
Goblin deals 13 to Madrigal
Madrigal deals 8 to Goblin
Goblin deals 7 to Madrigal
Madrigal deals 5 to Goblin
Goblin deals 12 to Madrigal
Madrigal deals 6 to Goblin
Goblin deals 6 to Madrigal
Madrigal deals 11 to Goblin
Goblin has been defeated
Madrigal of Neversummer, The Renowned Hero, is in A Long Corridor
(Currently: Calm) (Creature: None)
There is nothing to do here
> Enter your command:
```

В этой главе вы использовали интерфейсы, чтобы определить, что необходимо монстру (или игроку) для участия в сражении, а также абстрактные классы, чтобы создать базовый класс для всех монстров в `NyetHack`. Эти инструменты позволяют построить отношения между классами, которые задают, *что* может делать класс, но не *как*.

Многие идеи объектно-ориентированного программирования, с которыми вы познакомились в предыдущих главах, служат одной цели: использовать средства языка `Kotlin` для создания масштабируемой кодовой базы, которая раскрывает только нужное и скрывает все остальное.

За время путешествий в мире NyetHack вы многого добились: заложили фундамент условных конструкций и функций, определили собственные классы для представления объектов в мире, построили игровой цикл для получения ввода от игрока и даже создали мир с монстрами, с которыми можно сражаться. Поздравляем!

В следующей главе вы познакомитесь с обобщениями — средствами языка, которые позволяют определять классы, работающие со многими типами.

Задание: дополнительные монстры

На данный момент вы заложили основу для создания разнообразных противников в NyetHack, но пока игроку встречаются только гоблины. Расширьте мир NyetHack — определите новых монстров и разместите их в комнатах случайным образом, чтобы каждая партия в NyetHack дарила новые эмоции.

Например, попробуйте добавить классы `Draugr`, `Werewolf` и `Dragon`. Продумайте, сколько здоровья должно иметь каждое существо и сколько бросков игровых костей оно должно сделать, исходя из его силы. (Точные числа выбирайте сами, но разумно предположить, что у дракона должно быть намного больше здоровья, чем у гоблина.) При размещении монстров по комнатам добавьте разреженность в процесс генерации случайных размещений. Например, драконы в NyetHack — существа мифические и не должны появляться в таких местах, как длинный коридор рядом с городской площадью.

Возможно, вам придется добавить в NyetHack новые комнаты, чтобы герой имел возможность сразиться со всеми монстрами и даже погибнуть в сражении.

Часть V

Kotlin для опытных программистов

В предыдущей части книги вы познакомились с базовым синтаксисом Kotlin, функциональным и объектно-ориентированным программированием. У Kotlin в запасе есть еще несколько средств, которые построены на основе этих концепций и развивают их. В этой части книги мы рассмотрим три темы: обобщения, функции-расширения и свойства-расширения, а также сопрограммы.

18. Обобщения

В главе 9 мы рассказали, что список может содержать элементы любого типа: целые числа, строки и даже новые типы, определенные вами:

```
val listOfInts: List<Int> = listOf(1, 2, 3)
val listOfStrings: List<String> = listOf("string one", "string two")
val listOfRooms: List<Room> = listOf(Room(), TownSquare())
```

Списки могут хранить элементы любого типа благодаря *обобщениям* — системе типов, которая позволяет функциям и типам работать с типами, которые еще не известны вашему компилятору. Обобщения расширяют область повторного использования определений классов, потому что позволяют вашим определениям работать со многими типами.

В этой главе вы научитесь создавать обобщенные классы и функции, работающие с обобщенными параметрами типов. Мы добавим класс с именем `LootBox`. В каждой комнате `NyetHack` мы поместим сундук с сокровищами, которые игрок сможет забрать.

Определение обобщенных типов

Обобщенный тип — это класс, конструктор которого принимает входные данные любого типа (хотя для типа можно установить ограничения, как будет показано позднее в этой главе). Начнем с определения собственного обобщенного типа.

Создайте в `NyetHack` новый файл Kotlin с именем `Loot.kt`. В нем определите класс `LootBox` с *параметром обобщенного типа* для его содержимого, а также с `private`-свойством `contents`, в котором хранится награда (`item`).

Листинг 18.1. Создание обобщенного класса (`Loot.kt`)

```
class LootBox<T>(var contents: T)
```

Вы определяете класс `LootBox` (сундук)¹ и делаете его обобщенным, указывая параметр обобщенного типа для использования с классом; он обозначается `T` и заключается в угловые скобки (`< >`). Параметр обобщенного типа `T` определяет тип награды.

¹ В видеоиграх виртуальный ящик с наградами часто называют «лутбокс». — *Примеч. ред.*

Параметр обобщенного типа часто обозначается именно так — одной буквой *T* (сокращение от *type*), хотя с таким же успехом можно использовать любую букву или слово. Также иногда буквой *K* задают имя ключей (*key*), *V* — значения (*value*), *E* — элементы (*element*) и *R* — результаты (*result*). Если используют обобщения, которые не подходят под эти категории, обычно применяют обозначения *T*, *U* и *V*. При желании также можно вводить полные слова для имен типов в обобщениях.

Главный конструктор класса `LootBox` принимает награду любого типа в главном конструкторе (`var contents: T`) и сохраняет его в свойстве, как показано в сокращенном синтаксисе в главе 14.

В сундуки можно поместить различные награды. Создайте три разновидности: фетровые шляпы (`Fedora`), самоцветы (`Gemstones`) и ключи (`Key`) (с ними герой сможет попасть в новые, недоступные ранее области).

Листинг 18.2. Определение `loot` (`Loot.kt`)

```
class LootBox<T>(var contents: T)

class Fedora(
    val name: String,
    val value: Int
)

class Gemstones(
    val value: Int
)

class Key(
    val name: String
)
```

Пришло время опробовать новый класс `LootBox`. Создайте два сундука в функции `main`.

Листинг 18.3. Создание сундуков (`NyetHack.kt`)

```
...
fun main() {
    narrate("Welcome to NyetHack!")
    val playerName = promptHeroName()
    player = Player(playerName)
    // changeNarratorMood()

    val lootBoxOne: LootBox<Fedora> = LootBox(Fedora("a generic-looking fedora", 15))
    val lootBoxTwo: LootBox<Gemstones> = LootBox(Gemstones(150))
}
```

```
Game.play()  
}  
...
```

Так как вы сделали класс `LootBox` обобщенным, можно ввести только одно объявление класса для разных видов сундуков с наградами: одни сундуки — со шляпами, другие — с самоцветами, и т. д.

Обратите внимание на объявления переменных `LootBox`:

```
val lootBoxOne: LootBox<Fedora> = LootBox(Fedora("a generic-looking fedora", 15))  
val lootBoxTwo: LootBox<Gemstones> = LootBox(Gemstones(150))
```

Угловые скобки в объявлении типа переменной указывают, какой тип награды может хранить конкретный экземпляр `LootBox`.

Обобщенные типы, как и другие в Kotlin, поддерживают механизм автоматического определения типов. Для наглядности мы явно указали типы, но их можно опустить, так как каждая переменная инициализируется значением конкретного типа. В своем коде вы можете не включать информацию о типе, если она не нужна. Не стесняйтесь и смело удаляйте ее, если хочется. Тогда объявления сундуков будут выглядеть так:

```
val lootBoxOne = LootBox(Fedora("a generic-looking fedora", 15))  
val lootBoxTwo = LootBox(Gemstones(150))
```

Обобщенные функции

Обобщенные параметры типа также работают и с функциями. И это хорошая новость, так как в данный момент игрок не может забрать награду из сундука.

Настало время это исправить. Добавьте функцию, которая позволяет игроку достать награду, если он еще не забрал ее. Чтобы контролировать, не был ли сундук открыт ранее, добавим свойство `isOpen`.

Листинг 18.4. Добавление функции `takeLoot` (`Loot.kt`)

```
class LootBox<T>(var contents: T) {  
    var isOpen = false  
    private set  
  
    fun takeLoot(): T? {  
        return contents.takeIf { !isOpen }  
            .also { isOpen = true }  
    }  
}  
...
```

В примере объявляется обобщенная функция `takeLoot`, которая возвращает `T?` — допускающую версию `null` обобщенного типа `T`, определяющего вид на-

грады. Если бы функция `takeLoot` была объявлена вне `LootBox`, тип `T` оказался бы недоступен, так как `T` привязан к определению класса `LootBox`. Но как мы покажем в следующем разделе, обобщенные функции не требуют класса для использования параметра обобщенного типа.

Попробуйте забрать содержимое `lootBoxOne` в функции `main`, используя новую функцию `takeLoot`. Точнее, попробуйте сделать это дважды.

Листинг 18.5. Тестирование обобщенной функции `takeLoot` (NyetHack.kt)

```
...
fun main() {
    ...
    val lootBoxOne: LootBox<Fedora> = LootBox(Fedora("a generic-looking fedora",
15))
    val lootBoxTwo: LootBox<Gemstones> = LootBox(Gemstones(150))

    repeat(2) {
        narrate(
            lootBoxOne.takeLoot()?.let {
                "The hero retrieves ${it.name} from the box"
            } ?: "The box is empty"
        )
    }
    ...
}
...
```

Примените функцию области видимости `let` (с ней вы познакомились в главе 12) для вывода сообщения об открытии `lootBoxOne`. Напомним, что `let` получает аргумент (к которому можно обращаться по идентификатору `it`) со значением получателя, для которого она была вызвана. Так как для `lootBoxOne` тип `T` известен (он объявлен явно — это `Fedora`), мы знаем, что возвращаемый результат `takeLoot` — и `it` — имеет тип `Fedora`.

Запустите `NyetHack`. В выводе сообщается, что герой успешно достал награду из сундука и сундук опустел.

```
Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Madrigal
The hero retrieves a generic-looking fedora from the box
The box is empty
Welcome, adventurer
...
```

Ограничения обобщений

В текущей версии в сундук можно положить все что угодно. Ситуация оставляет желать лучшего — не хотелось бы, чтобы в сундуке оказался монстр. Для гарантии

того, что в сундук можно складывать только награды, а не что-нибудь нехорошее, укажем ограничение обобщенного типа.

Для начала добавим абстрактный класс `Loot` и интерфейс с именем `Sellable` в `Loot.kt`. Затем объявим классы наград подклассами нового суперкласса, как показано в листинге 18.6. Ключи не будут `Sellable` (продаваемыми) — никто не захочет покупать ключ неизвестного назначения.

Листинг 18.6. Добавление суперклассов (`Loot.kt`)

```
...
abstract class Loot {
    abstract val name: String
}

interface Sellable {
    val value: Int
}

class Fedora(
    override val name: String,
    override val value: Int
) : Loot(), Sellable

class Gemstones(
    override val value: Int
) : Loot(), Sellable {
    override val name = "sack of gemstones worth $value gold"
}

class Key(
    override val name: String
) : Loot()
```

Теперь добавим ограничение в объявление параметра обобщенного типа `LootBox`, чтобы с `LootBox` могли использоваться только наследники класса `Loot`.

Листинг 18.7. Ограничение параметра обобщенного типа суперклассом `Loot` (`Loot.kt`)

```
class LootBox<T : Loot>(var contents: T) {
    ...
}
...
```

Если не задать ограничение для обобщенного типа, Kotlin неявно использует ограничение `Any?`, которое означает, что с обобщенным классом может использоваться любой тип (допускающий или не допускающий `null`). В примере мы добавили ограничение для обобщенного типа `T`, заданное в виде `:Loot`. Теперь в сундук можно положить только те награды, которые являются наследниками класса `Loot`.

Запустите `NyetHack` и убедитесь, что код выполняется без ошибок и выводит тот же результат, что и прежде.

Возможно, у вас возник вопрос: зачем вообще нужен параметр `T`? Почему просто не использовать тип `Loot`? Параметр `T` позволяет `LootBox` обращаться к награде определенного типа и одновременно допускает хранение в сундуке любой разновидности `Loot`. То есть в `LootBox` вместо `Loot` будет храниться, например, тип `Fedora`. И конкретный тип `Fedora` определяется с помощью `T`.

Объявив тип награды `Loot`, вы точно так же смогли бы поместить в сундук только наследники `Loot`, но информация о том, что в сундуке хранится награда `Fedora`, была бы утрачена. Например, при использовании конкретного типа `Loot` следующий код не сможет компилироваться:

```
val lootBox: LootBox<Loot> = LootBox(Fedora("a dazzling fuchsia fedora", 15))
val fedora: Fedora = lootBox.contents // Несоответствие типов.
// Требуется Fedora, обнаружен Loot
```

И вам уже не удастся узнать, что сундук `LootBox` хранит что-то, отличное от `Loot`. Используя ограничение типов, возможно ограничить содержимое сундука до `Loot` и сохранить подтип награды в сундуке.

Для обобщенных типов допустимо определять более сложные ограничения. Вероятно, в какой-то момент герой захочет обменять свои самоцветы и шляпы на деньги. Добавьте новый класс с именем `DropOffBox` для создания магазинчика, где ценности можно продать — с уплатой 30% комиссионных.

Листинг 18.8. Использование нескольких ограничений (`Loot.kt`)

```
class LootBox<T : Loot>(var contents: T) {
    ...
}

class DropOffBox<T> where T : Loot, T : Sellable {
    fun sellLoot(sellableLoot: T): Int {
        return (sellableLoot.value * 0.7).toInt()
    }
}

abstract class Loot {
    abstract val name: String
}

interface Sellable {
    val value: Int
}
...
```

Чтобы задать ограничения в новом коде, для обобщенного типа `T` используем ключевое слово `where`. Заданное ограничение указывает, что тип `T` должен расши-

рять `Loot` и реализовать `Sellable`. Это ограничение не позволяет `DropOffBox` принимать ключи (бесполезные для всех, кроме их владельца и потенциальных воров) и другие предметы, которые могут появиться в игре в будущем (такие предметы можно продавать, но они не будут наградами).

В теле функции доступны все функции и свойства, объявленные для `Loot` и `Sellable`, потому что тип `T` гарантированно расширяет оба типа.

Позже в этой главе мы создадим экземпляр `DropOffBox` на городской площади, чтобы герой получил возможность продать свои сокровища после возвращения из странствий. А пока протестируем новый магазинчик в REPL и убедимся, что в нем обменивают награды на деньги, как и ожидалось, при условии, что предлагаемые героем вещи пригодны для продажи. (Возможно, вам придется перезагрузить REPL кнопкой `Build and restart` в левой части окна.)

Листинг 18.9. Использование класса с несколькими ограничениями (REPL)

```
import com.bignerdranch.nyethack.*

val hatDropOffBox = DropOffBox<Fedora>()
hatDropOffBox.sellLoot(Fedora("a sequin-covered fedora", 20))
14

hatDropOffBox.sellLoot(Gemstones(100))
error: type mismatch: inferred type is Gemstones but Fedora was expected
hatDropOffBox.sellLoot(Gemstones(100))
      ^
```

in и out

Для дальнейшей настройки параметров обобщенного типа Kotlin предоставляет ключевые слова `in` и `out`. Чтобы посмотреть, как они влияют на обобщенные классы, попробуйте выполнить в REPL следующий код:

Листинг 18.10. Попытка повторного присвоения `lootBox` (REPL)

```
var fedoraBox: LootBox<Fedora> = LootBox(Fedora("a generic-looking fedora", 15))
var lootBox: LootBox<Loot> = LootBox(Gemstones(150))

lootBox = fedoraBox
error: type mismatch: inferred type is LootBox<Fedora> but
      LootBox<Loot> was expected
lootBox = fedoraBox
      ^
```

Результат, наверное, вас удивил. Компилятор не позволит заново присвоить `lootBox` значение `fedoraBox`.

Казалось бы, такое присваивание должно быть допустимо. `Fedora` все-таки является наследником `Loot`, и переменной типа `Loot` вполне можно присвоить экземпляр `Fedora`:

```
var loot: Loot = Fedora("a generic-looking fedora", 15) // Без ошибок
```

Чтобы понять, где кроется проблема, разберемся, что могло бы произойти, если бы такое присваивание выполнялось успешно.

Если компилятор разрешит присвоить экземпляр `fedoraBox` переменной `lootBox`, то `lootBox` будет указывать на `fedoraBox` и появится возможность взаимодействовать с наградой в `fedoraBox` так, будто это `Loot`, а не `Fedora` (потому что тип `lootBox` — это `LootBox<Loot>`).

Например, `Gemstones` — это один из подтипов `Loot`, поэтому было бы возможно присвоить `Gemstones` свойству `lootBox.contents` (которое указывает на `fedoraBox`).

```
var fedoraBox: LootBox<Fedora> = LootBox(Fedora("a generic-looking fedora", 15))
var lootBox: LootBox<Loot> = LootBox(Gemstones(150))
lootBox = fedoraBox
lootBox.contents = Gemstones(200)
```

Теперь допустим, что вы попытались обратиться к `fedoraBox.contents`, ожидая получить `Fedora`:

```
var fedoraBox: LootBox<Fedora> = LootBox(Fedora("a generic-looking fedora",
15))
var lootBox: LootBox<Loot> = LootBox(Gemstones(150))

lootBox = fedoraBox
lootBox.contents = Gemstones(200)
val myFedora: Fedora = fedoraBox.contents
```

Компилятор столкнется с несоответствием типа — `fedoraBox.contents` имеет тип `Gemstones`, а не `Fedora`. При выполнении программы в этом гипотетическом сценарии происходит исключение `ClassCastException`. Именно из-за возникновения этой проблемы компилятор изначально запретил присваивание.

Решение — ключевые слова `in` и `out`.

В объявлении класса `lootBox` добавьте ключевое слово `out` и измените `contents` с `var` на `val`.

Листинг 18.11. Добавление ключевого слова `out` (`Loot.kt`)

```
class LootBox<out T : Loot>(var val contents: T) {
    ...
}
```

Теперь попробуйте снова выполнить последний фрагмент кода, введенный в REPL (из листинга 18.10). На этот раз REPL не выводит никаких сообщений об ошибках, а следовательно, код откомпилировался успешно.

Что же изменилось?

Параметру обобщенного типа можно назначить две роли: *производитель* и *потребитель*. Роль производителя подразумевает, что обобщенный параметр доступен для чтения (но не для записи), а роль потребителя означает, что обобщенный параметр доступен для записи (но не для чтения).

Добавив ключевое слово `out` в `LootBox<out T>`, вы указали, что обобщенный параметр — производитель. То есть он доступен для чтения, но не для записи. Это означает, что объявление `contents` с ключевым словом `var` более не разрешено, потому что `LootBox` станет производителем *и* потребителем типа `T`.

Вспомните: когда переменной `contents` можно было присвоить другое значение, в некоторых сценариях ее тип менялся, что приводило к неожиданным ошибкам при извлечении награды из сундука. Сделав обобщение производителем, вы сообщили компилятору, что эта проблема больше не появится: поскольку обобщенный параметр теперь производитель, а не потребитель, переменная `contents` никогда не будет изменяться.

Kotlin разрешит присвоить экземпляру `fedoraBox` переменной `lootBox`, потому что это безопасно: свойство `contents` экземпляра `lootBox` теперь имеет тип `Fedora`, а не `Loot` и не сможет измениться. Kotlin разрешает такое приведение типов только в том случае, если тип помечен ключевым словом `out`. И компилятор поддерживает ограничение, требующее, чтобы любой тип с ключевым словом `out` использовался только как возвращаемый тип, но не входной.

Кстати, списки `List` тоже являются производителями. В Kotlin в объявлении `List` параметр обобщенного типа отмечен ключевым словом `out`:

```
public interface List<out E> : Collection<E>
```

(При этом класс `MutableList` не является ни производителем, ни потребителем. Он получает данные на входе и выводит данные в коллекцию, и это означает, что вы не сможете безопасно привести `MutableList<Fedora>` к `MutableList<Loot>`.)

Класс `DropOffBox` является потребителем: он получает значения обобщенного типа, но не выводит их.

Обобщенный параметр типа класса `DropOffBox` помечен ключевым словом `in`, чтобы он имел обратный эффект при приведении экземпляров `DropOffBox`. Вместо возможности приведения `DropOffBox<Fedora>` к `DropOffBox<Loot>` вам будет разрешено присвоить значение `DropOffBox<Loot>` переменной `DropOffBox<Fedora>` — но не наоборот. Кроме того, обобщенные типы, помеченные ключевым словом `in`, не сохраняются в свойствах, так как чтение из свойства считается выводом значения, нарушающим правило потребителя.

Обновите класс `DropOffBox`, чтобы в нем использовалось ключевое слово `in`.

Листинг 18.12. Пометка DropOffBox ключевым словом in (Loot.kt)

```
...
class DropOffBox<in T> where T : Loot, T : Sellable {
    fun sellLoot(sellableLoot: T): Int {
        return (sellableLoot.value * 0.7).toInt()
    }
}
...
```

Прежде чем вы начнете использовать новые возможности приведения типов, необходимо объявить и другие типы наград. Создайте новый абстрактный класс с именем `Hat`, который послужит базовым классом для разных видов головных уборов. Объявите класс `Fedora` как расширение `Hat` и добавьте новый тип `Fez` (феска), чтобы расширить ассортимент головных уборов.

Листинг 18.13. Добавление новых головных уборов (Loot.kt)

```
...
abstract class Hat : Loot(), Sellable

class Fedora(
    override val name: String,
    override val value: Int
) : Loot(), Sellable Hat()

class Fez(
    override val name: String,
    override val value: Int
) : Hat()
...
```

Расширение ассортимента привлекает внимание странствующего торговца, который тоже заглядывает в магазинчик `DropOffBox`.

Изначально торговец покупает все виды головных уборов. Если рынок внезапно окажется завален фесками, торговец ограничивает `DropOffBox`, чтобы принимать только шляпы. С ключевым словом `in` эту ситуацию теперь можно отлично смоделировать.

Выполните следующий код в REPL:

Листинг 18.14. Ключевое слово in при приведении типа (REPL)

```
import com.bignerdranch.nyethack.*

val hatDropOffBox: DropOffBox<Hat> = DropOffBox()
val fedoraDropOffBox: DropOffBox<Fedora> = hatDropOffBox

fedoraDropOffBox.sellLoot(Fedora("one-of-a-kind fedora", 1000))
700
```

Присваивание стало возможным, потому что теперь компилятор уверен, что вы уже никогда не сможете произвести `Not` из `DropOffBox` для `Fedora` и таким образом избежите возможных исключений `ClassCastException`. Так как головные уборы никогда не покидают стен магазинчика `DropOffBox`, компилятор может сделать вывод, что такое присваивание безопасно.

Кстати, возможно, вы слышали о терминах *ковариантность* (covariance) и *контравариантность* (contravariance), которые описывают то, что делают `out` и `in`. По нашему мнению, эти описания весьма туманны, поэтому мы постараемся их избегать. Мы упомянули их на всякий случай, если вы вдруг столкнетесь с такими терминами. Теперь вы будете знать: если слышите «ковариантность», то речь идет об `out`, а если «контравариантность» — то об `in`.

Добавление наград в NyetHack

Необходимая основа заложена, и мы займемся тем, как распределить награды в `NyetHack`, чтобы герой мог собрать их и продать. Для начала дадим игроку карманы для хранения наград.

Листинг 18.15. Добавление места для хранения (Player.kt)

```
class Player(  
    initialName: String,  
    val hometown: String = "Neversummer",  
    override var healthPoints: Int,  
    val isImmortal: Boolean  
) : Fightable {  
    ...  
    val prophecy by lazy {  
        ...  
    }  
  
    val inventory = mutableListOf<Loot>()  
  
    var gold = 0  
    ...  
}
```

Теперь добавьте в `LootBox` объект-компаньон и создайте функцию `random`, которая будет случайным образом генерировать сундуки с различными наградами.

Листинг 18.16. Формирование случайных наград (Loot.kt)

```
class LootBox<out T : Loot>(val contents: T) {  
    var isOpen = false  
    private set  
  
    fun takeLoot(): T? {
```

```

        return contents.takeIf { !isOpen }
            .also { isOpen = true }
    }

    companion object {
        fun random(): LootBox<Loot> = LootBox(
            contents = when (Random.nextInt(1..100)) {
                in 1..5 -> Fez("fez of immaculate style", 150)
                in 6..10 -> Fedora("fedora of knowledge", 125)
                in 11..15 -> Fedora("stunning teal fedora", 75)
                in 16..30 -> Fez("ordinary fez", 15)
                in 31..50 -> Fedora("ordinary fedora", 10)
                else -> Gemstones(Random.nextInt(50..100))
            }
        )
    }
}
...

```

(Не забудьте импортировать `kotlin.random.Random` и `kotlin.random.nextInt`.)

Эта функция случайным образом создает сундуки с самоцветами или ценными головными уборами.

Разобравшись с генерированием наград, можно добавить сундук в каждую комнату в NyetHack. Для этого добавьте общедоступное свойство в класс `Room`.

Листинг 18.17. Добавление сундуков в Room (Room.kt)

```

open class Room(val name: String) {

    protected open val status = "Calm"
    open val lootBox: LootBox<Loot> = LootBox.random()
    ...
}

```

Чтобы игрок получил возможность находить в комнатах награды, мы используем тип `Loot` — наиболее общий класс предметов, которые могут храниться в сундуках. И хотя `Room` указывает, что `lootBox` способен содержать любую разновидность наград, в подклассах допустимо отойти от этого правила, потому что свойство помечено ключевым словом `open`.

Предположим, в таверне всегда стоит сундук с ключом, который может пригодиться позднее. Переопределите свойство `lootBox` класса `Tavern`.

Листинг 18.18. Переопределение обобщенного свойства (Tavern.kt)

```

...
class Tavern : Room(TAVERN_NAME) {
    ...
    override val status = "Busy"
}

```

```

    override val lootBox: LootBox<Key> =
        LootBox(Key("key to Nogartse's evil lair"))
    ...
}
...

```

Новое свойство объявляется в виде `LootBox<Key>`, что расходится с типом `LootBox<Loot>` в родительском классе. Такое стало возможным, потому что обобщенный тип `T` у `LootBox` объявлен с `out`. Компилятор может безопасно преобразовать `Key` в `Loot` (и как следствие, `LootBox<Key>` в `LootBox<Loot>`), поэтому такое переопределение разрешено. Если теперь прочитать `LootBox` из экземпляра `Tavern`, компилятор поймет, что вы получите `Key`, так что код откомпилируется и будет успешно выполняться:

```

val tavern = Tavern()
val key: Key? = tavern.lootBox.takeLoot()

```

Когда мы закончим с размещением сундуков, а у героя появятся карманы, можно переходить к реализации новой команды «take loot» (получи награду). Заодно удалите код сундуков из `main`, так как сундуки теперь размещаются в комнатах `NyetHack`.

Листинг 18.19. Реализация команды получения наград (NyetHack.kt)

```

...
fun main() {
    narrate("Welcome to NyetHack!")
    val playerName = promptHeroName()
    player = Player(playerName)
    // changeNarratorMood()

    val lootBoxOne: LootBox<Fedora> = LootBox(Fedora("a generic-looking fedora", 15))
    val lootBoxTwo: LootBox<Gemstones> = LootBox(Gemstones(150))

    repeat(2){
        narrate{
            lootBoxOne.takeLoot()?.let {
                "The hero retrieves ${it.name} from the box"
            } ?: "The box is empty"
        }
    }

    Game.play()
}
...
object Game {
    ...
    fun takeLoot() {

```

```

    val loot = currentRoom.lootBox.takeLoot()
    if (loot == null) {
        narrate("${player.name} approaches the loot box, but it is empty")
    } else {
        narrate("${player.name} now has a ${loot.name}")
        player.inventory += loot
    }
}

private class GameInput(arg: String?) {
    ...
    fun processCommand() = when (command.lowercase()) {
        "fight" -> fight()
        "move" -> ...
        "take" -> {
            if (argument.equals("loot", ignoreCase = true)) {
                takeLoot()
            } else {
                narrate("I don't know what you're trying to take")
            }
        }
        else -> narrate("I'm not sure what you're trying to do")
    }
    ...
}
}

```

Опробуйте новую команду в разных комнатах NyetHack. В таверне всегда должен храниться ключ к логову главного злодея, но все остальные комнаты должны содержать случайные награды. Вывод выглядит примерно так.

```

Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Madrigal
Welcome, adventurer
Madrigal, a mortal, has 100 health points
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command: take loot
Madrigal now has a sack of gemstones worth 57 gold
...
> Enter your command:

```

Теперь герой имеет возможность собирать награды, и мы разместим магазинчики, где герой будет продавать добытые ценности за игровые деньги. Разместите на городской площади две торговые точки: для головных уборов и для самоцветов. Затем создайте в TownSquare новую функцию с именем `sellLoot`, которая выберет правильный магазин для продажи.

Листинг 18.20. Размещение магазинчиков (TownSquare.kt)

```

open class TownSquare : Room("The Town Square") {
    override val status = "Bustling"
    private var bellSound = "GWONG"
    val hatDropOffBox = DropOffBox<Hat>()
    val gemDropOffBox = DropOffBox<Gemstones>()

    final override fun enterRoom() {
        narrate("The villagers rally and cheer as the hero enters")
        ringBell()
    }

    fun ringBell() {
        narrate("The bell tower announces the hero's presence: $bellSound")
    }

    fun <T> sellLoot(
        loot: T
    ): Int where T : Loot, T : Sellable {
        return when (loot) {
            is Hat -> hatDropOffBox.sellLoot(loot)
            is Gemstones -> gemDropOffBox.sellLoot(loot)
            else -> 0
        }
    }
}

```

Чтобы игрок мог воспользоваться новыми торговыми точками, создайте еще одну команду sell loot.

Листинг 18.21. Продажа наград (NyetHack.kt)

```

...
object Game {
    ...
    fun sellLoot() {
        when (val currentRoom = currentRoom) {
            is TownSquare -> {
                player.inventory.forEach { item ->
                    if (item is Sellable) {
                        val sellPrice = currentRoom.sellLoot(item)
                        narrate("Sold ${item.name} for $sellPrice gold")
                        player.gold += sellPrice
                    } else {
                        narrate("Your ${item.name} can't be sold")
                    }
                }
                player.inventory.removeAll { it is Sellable }
            }
            else -> narrate("You cannot sell anything here")
        }
    }
}

```

```

}

private class GameInput(arg: String?) {
    ...
    fun processCommand() = when (command.lowercase()) {
        ...
        "take" -> {
            if (argument.equals("loot", ignoreCase = true)) {
                takeLoot()
            } else {
                narrate("I don't know what you're trying to take")
            }
        }
        "sell" -> {
            if (argument.equals("loot", ignoreCase = true)) {
                sellLoot()
            } else {
                narrate("I don't know what you're trying to sell")
            }
        }
        else -> narrate("I'm not sure what you're trying to do")
    }
    ...
}
}

```

В программе есть две функции с именем `sellLoot`: `TownSquare` содержит функцию `sellLoot` для продажи заданной награды в подходящем магазинчике, а `Game` содержит другую функцию `sellLoot`, которая продает добычу героя в зависимости от того, в какой комнате тот находится. Если герой находится на городской площади, то функция `sellLoot` класса `Game` вызывает функцию `sellLoot` класса `TownSquares`, после чего удаляет все предметы, которые можно продать, из карманов героя. В противном случае награды не могут быть проданы в текущем местоположении игрока.

Система наград готова. Протестируйте ее, посещая разные комнаты в `NyetHack`, собирая ценности и обменивая их на деньги. Результат должен выглядеть примерно так:

```

Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Madrigal
Welcome, adventurer
Madrigal, a mortal, has 100 health points
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command:take loot
Madrigal now has a sack of gemstones worth 70 gold
Madrigal of Neversummer, The Renowned Hero, is in The Town Square

```

```

    (Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command:move east
The hero moves East
Madrigal of Neversummer, The Renowned Hero, is in Taernyl's Folly
    (Currently: Busy)
...
> Enter your command:take loot
Madrigal now has a key to Nogartse's evil lair
...
> Enter your command:move west
The hero moves West
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
    (Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command:sell loot
Sold sack of gemstones worth 70 gold for 49 gold
Your key to Nogartse's evil lair can't be sold
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
    (Currently: Bustling)
The villagers rally and cheer as the hero enters
The bell tower announces the hero's presence: GWONG
> Enter your command:

```

В этой главе мы рассказали, как использовать обобщения для расширения возможностей классов в языке Kotlin. Вы узнали, как ограничивать обобщенные типы и использовать ключевые слова `in` и `out` для определения роли производителя и потребителя обобщенного параметра.

В следующей главе мы рассмотрим расширения, позволяющие совместно использовать функции и свойства без наследования. Мы воспользуемся ими для улучшения кодовой базы NyetHack.

Для любознательных: ключевое слово `reified`

Иногда полезно знать конкретный тип обобщенного параметра. Сделать это можно с помощью ключевого слова `reified`.

Предположим, герой отправился за наградами определенного типа. Например, он может собрать все головные уборы, которые ему попадутся, игнорируя остальные виды наград. Функция `takeLootOfType` пытается выразить эту логику:

```

class LootBox<out T : Loot>(val contents: T) {
    var isOpen = false
    private set

    fun takeLoot(): T? {

```



```

        return contents.takeIf { !isOpen }
            .also { isOpen = true }
    }

    fun <U> takeLootOfType(): U? {
        return if (contents is U) {
            takeLoot() as U
        } else {
            null
        }
    }
    ...
}

```

```

val lootBox = LootBox.random()
val loot = lootBox.takeLootOfType<Hat>()

```

Если вы введете этот код, то обнаружите, что он не работает. IntelliJ выделит параметр типа `U` как ошибку (рис. 18.1).

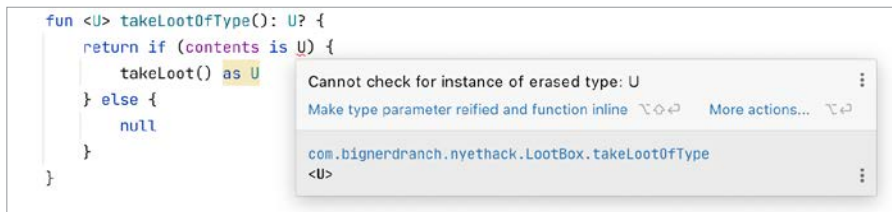


Рис. 18.1. Нельзя проверить экземпляр стертого типа

Информация об обобщенных типах обычно доступна только во время компиляции. Когда вы компилируете свой код, Kotlin проверяет, что обобщенные типы соответствуют использованию класса, а затем убирает информацию обобщенных типов из откомпилированного кода. На практике это означает, что `LootBox<Hat>` и `LootBox<Gemstones>` компилируются в `LootBox` без дополнительной информации о типе. Данное явление называется *стиранием типов* (type erasure) — это означает, что информация об обобщенном типе недоступна во время выполнения.

Так как проверка производится во время выполнения, откомпилированная программа не содержит достаточной информации, чтобы узнать тип `U`. Однако Kotlin предоставляет возможность обойти это ограничение при помощи ключевого слова `reified`. Используя его с обобщенным типом, вы сообщаете компилятору, что обобщенный тип должен быть доступен во время выполнения. Использование ключевого слова `reified` также требует, чтобы функция была встраиваемой.

Чтобы избежать стирания типов в `takeLootOfType`, а также устранить ошибку компилятора, можно добавить ключевые слова `inline` и `reified`.

```
inline fun <reified U> takeLootOfType(): U? {  
    return if (contents is U) {  
        takeLoot() as U  
    } else {  
        null  
    }  
}
```

Теперь проверка типа `contents is U` возможна, потому что информация о типе сохранена. Компилятор Kotlin сохраняет информацию типа, встраивая функцию и заменяя обобщенный тип фактическим. Так как функция должна быть помечена ключевым словом `inline`, на нее распространяются все ограничения и потенциальные проблемы встраиваемых функций, о которых мы рассказали в главе 8.

Ключевое слово `reified` позволяет быстро и эффективно проверить тип обобщенного параметра.

19. Расширения

Расширения позволяют добавить функциональность типу без явного изменения определения типа. Применяйте расширения с пользовательскими типами, а также с типами, которые вам неподконтрольны, — `List`, `String` и другими из стандартной библиотеки Kotlin.

Расширения — это альтернатива наследованию. Они хорошо подходят для добавления функциональности в тип, если определение класса недоступно или класс не имеет модификатора `open`, разрешающего создавать подклассы.

Стандартная библиотека Kotlin часто использует расширения. Например, функции области видимости, о которых мы рассказывали в главе 12, определяются как расширения, и в этой главе мы покажем несколько примеров их объявления.

Определение функции-расширения

Первое расширение, с которым мы познакомимся, позволит добавить любую порцию энтузиазма в произвольную строку. Определите его в новом файле `Extensions.kt` в `NyetHack`.

Листинг 19.1. Добавление расширения для типа `String` (`Extensions.kt`)

```
fun String.addEnthusiasm(enthusiasmLevel: Int = 1) =  
    this + "!".repeat(enthusiasmLevel)
```

Функции-расширения объявляются так же, как и другие функции, но с одним отличием: определяя функцию расширения, вы также указываете тип, которому расширение добавляет функциональность, — он известен как *тип-получатель*. (Вспомните, что в главе 12 субъект расширения назывался получателем.) Для функции `addEnthusiasm` в качестве типа-получателя указывается `String`.

Тело функции `addEnthusiasm` представляет собой одно выражение, которое возвращает строку: содержимое `this` с несколькими восклицательными знаками в зависимости от значения аргумента `enthusiasmLevel` (по умолчанию 1). Ключевое слово `this` ссылается на экземпляр объекта-получателя, для которого вызвана функция-расширение (в этом случае экземпляр — `String`).

Теперь можно вызвать функцию `addEnthusiasm` для любого экземпляра `String`. Испытайте новую функцию-расширение, объявив строку в функции `main` и вызвав для нее функцию-расширение `addEnthusiasm` с выводом результата.

Листинг 19.2. Вызов нового расширения для экземпляра объекта-получателя `String` (`Extensions.kt`)

```
...
object Game {
    ...
    fun fight() {
        val monsterRoom = currentRoom as? MonsterRoom
        val currentMonster = monsterRoom?.monster
        if (currentMonster == null) {
            narrate("There's nothing to fight here")
            return
        }

        var combatRound = 0
        val previousNarrationModifier = narrationModifier
        narrationModifier = { it.addEnthusiasm(enthusiasmLevel = combatRound) }

        while (player.healthPoints > 0 && currentMonster.healthPoints > 0) {
            combatRound++

            player.attack(currentMonster)
            if (currentMonster.healthPoints > 0) {
                currentMonster.attack(player)
            }
            Thread.sleep(1000)
        }
        narrationModifier = previousNarrationModifier

        if (player.healthPoints <= 0) {
            narrate("You have been defeated! Thanks for playing.")
            exitProcess(0)
        } else {
            narrate("${currentMonster.name} has been defeated!")
            monsterRoom.monster = null
        }
    }
    ...
}
```

Запустите `Extensions.kt` и посмотрите, добавляет ли функция-расширение восклицательные знаки в строку, как и предполагалось.

```
...
Madrigal of Neversummer, The Renowned Hero, is in A Long Corridor
(Currently: Calm) (Creature: A nasty-looking goblin)
Danger is lurking in this room
```

```

> Enter your command:fight
Madrigal deals 6 to Goblin!
Goblin deals 8 to Madrigal!
Madrigal deals 5 to Goblin!!
Goblin deals 8 to Madrigal!!
Madrigal deals 2 to Goblin!!!
Goblin deals 8 to Madrigal!!!
Madrigal deals 6 to Goblin!!!!
Goblin deals 6 to Madrigal!!!!
Madrigal deals 5 to Goblin!!!!!
Goblin deals 8 to Madrigal!!!!!!
Madrigal deals 4 to Goblin!!!!!!!
Goblin deals 4 to Madrigal!!!!!!!
Madrigal deals 6 to Goblin!!!!!!!
Goblin has been defeated
Madrigal of Neversummer, The Renowned Hero, is in A Long Corridor
(Currently: Calm) (Creature: None)
There is nothing to do here
> Enter your command:

```

Можно ли создать подкласс `String`, чтобы добавить эту возможность экземплярам `String`? В IntelliJ просмотрите исходный код объявления `String`, нажав клавишу Shift дважды, чтобы открыть диалоговое окно **Search Everywhere**. Выполните поиск файла `String.kt`, в котором определен тип. Заголовок выглядит так:

```

public class String : Comparable<String>, CharSequence {
    ...
}

```

Так как ключевое слово `open` отсутствует в определении класса `String`, вы не сможете создать подкласс `String` для добавления новых возможностей через наследование. Как мы говорили ранее, расширения — хороший вариант, если вы хотите добавить функциональности в класс, которым не можете управлять или который нельзя использовать для создания подкласса.

Объявление расширения для суперкласса

Расширения не полагаются на наследование, но их можно сочетать с наследованием. Попробуйте сделать это в `Extensions.kt`: объявите для типа `Any` расширение с именем `print`. Так как расширение объявлено для `Any`, оно будет доступно для всех типов.

Листинг 19.3. Расширение `Any` (`Extensions.kt`)

```

fun String.addEnthusiasm(enthusiasmLevel: Int = 1) =
    this + "!".repeat(enthusiasmLevel)

fun Any.print() {
    println(this)
}

```

Опробуйте новую функцию `print` в REPL.

Листинг 19.4. Функция `print` доступна для всех подтипов (REPL)

```
import com.bignerdranch.nyethack.*

"Hello from String!".print()
42.print()
Hello from String!
42
```

Обобщенные функции-расширения

А что, если вы хотите вывести строку "Madrigal has left the building" как до, так и после вызова `addEnthusiasm`?

Для этого нужно добавить в функцию `print` возможность вызова цепочкой. Вы уже знаете, что функции могут объединяться в цепочки, если они возвращают объект-получатель или иной объект, для которого допустимо вызывать последующие функции.

Обновите `print` для вызова цепочкой.

Листинг 19.5. Изменение `print` для вызова цепочкой (Extensions.kt)

```
fun String.addEnthusiasm(enthusiasmLevel: Int = 1) =
    this + "!".repeat(enthusiasmLevel)

fun Any.print(): Any {
    println(this)
    return this
}
```

Теперь попробуйте вызвать функцию `print` дважды — до и после `addEnthusiasm`.

Листинг 19.6. Функция `print` вызывается дважды (REPL)

```
"Madrigal has left the building".print().addEnthusiasm().print()
error: unresolved reference. None of the following candidates is applicable
because of receiver type mismatch:
public fun String.addEnthusiasm(enthusiasmLevel: Int = ...): String defined in
com.bignerdranch.nyethack
"Madrigal has left the building".print().addEnthusiasm().print()
^
```

Код не компилируется. Первый вызов `print` был разрешен, а вызов `addEnthusiasm` нет. Чтобы понять почему, взгляните на возвращаемый тип `addEnthusiasm`:

```
fun Any.print(): Any
```

Так как функция `print` возвращает тип `Any`, информация о типе получателя теряется и вы не получаете обратно `String`. Хотя проблему можно решить приведением типа, лучше, если функция `print` будет возвращать тот же тип, для которого она вызывалась, — например `String` при вызове для экземпляра `String`.

Чтобы решить эту проблему, можно сделать функцию `print` обобщенной и указать, что она возвращает тот же тип, для которого была вызвана. Обновите вашу функцию, чтобы в качестве получателя она использовала обобщенный тип вместо `Any`.

Листинг 19.7. Обобщение `print` (`Extensions.kt`)

```
...
fun <T> AnyT.print():-AnyT {
    println(this)
    return this
}
```

Теперь, когда расширение использует параметр обобщенного типа `T` в качестве получателя и возвращает `T` вместо `Any`, информация о конкретном типе объекта-получателя передается далее по цепочке вызовов.

Снова выполните код из листинга 19.6. На этот раз вывод будет выглядеть так:

```
Madrigal has left the building
Madrigal has left the building!
```

Новая обобщенная функция-расширение работает с любым типом, а также хранит информацию о типе. Расширения, использующие обобщенные типы, позволяют писать функции, которые могут работать с самыми разными типами в программе.

Расширения для обобщенных типов имеются и в стандартной библиотеке Kotlin. Например, посмотрите на объявление функции `let`:

```
public inline fun <T, R> T.let(block: (T) -> R): R {
    return block(this)
}
```

`let` определяется как обобщенная функция-расширение, что позволяет ей работать со всеми типами. Она принимает лямбда-функцию, которая получает объект-получатель в качестве аргумента (`T`) и возвращает `R` — некоторый новый тип, который возвращается лямбда-выражением.

Обратите внимание на ключевое слово `inline`, о котором мы рассказывали в главе 8. Тот же совет, что мы давали раньше, применим и здесь: объявите функцию-расширение встраиваемой, если она получает лямбда-выражение, — это уменьшает затраты памяти.

Операторные функции-расширения

В главе 16 вы узнали ключевое слово `operator` и использовали его для реализации оператора сложения. Функции-расширения можно объединять с ключевым словом `operator`, чтобы предоставлять реализации операторов Kotlin для типов, которые вам неподконтрольны.

Вспомните тип `Coordinate`, который используется для обращения к текущей комнате в следующей строке кода:

```
worldMap.getOrNull(newPosition.y)?.getOrNull(newPosition.x)
```

Синтаксис индексирования списка получается слишком громоздким. Почистите его, определив другую функцию расширения, чтобы получить правильную комнату `Room` из `worldMap` для заданного экземпляра `Coordinate`.

Листинг 19.8. Определение операторной функции-расширения

```
...
fun <T> T.print(): T {
    println(this)
    return this
}

operator fun List<List<Room>>.get(coordinate: Coordinate) =
    getOrNull(coordinate.y)?.getOrNull(coordinate.x)
```

Это позволит вам использовать оператор индексирования (`[]`) с `worldMap` для получения следующей комнаты (вместо двух вызовов `get`).

Листинг 19.9. Использование операторной функции, определенной как расширение (NyetHack.kt)

```
...
object Game {
    ...
    fun move(direction: Direction) {
        val newPosition = direction.updateCoordinate(currentPosition)
        val newRoom = worldMap.getOrNull(newPosition.y)?.getOrNull(newPosition.x)
        val newRoom = worldMap[newPosition]
        ...
    }
}
...
```

Новый синтаксис более явно выражает в коде ваши намерения и скрывает подробности реализации внутри функции-расширения. Кроме того, если другой функции понадобится найти комнату по координатам, она сможет это сделать без повторного объявления той же логики.

Запустите NyetHack и убедитесь, что в новой версии вы можете перемещаться между комнатами.

С функциями-расширениями можно использовать *инфиксные функции*. Модификатор функций `infix` используется с функциями-расширениями или функциями классов, имеющими ровно один параметр.

Если функция помечена ключевым словом `infix`, это позволяет опустить точку перед именем функции и не заключать аргумент в круглые скобки.

Вы уже использовали инфиксные функции в своем коде. Взгляните на определение функции `to`, с которой мы познакомили вас в главе 10:

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

Вы можете вызвать функцию `to` любым способом, из-за чего она внешне почти не отличается от операторов языка.

Полный синтаксис	<code>playerName.to(hometown)</code>

Инфиксная запись	<code>playerName to hometown</code>

Многие разработчики избегают объявления собственных инфиксных функций. Тем не менее такая возможность существует, если вы считаете, что она улучшит удобочитаемость кода, или захотите создать новую функцию, которая выглядит как равноправная языковая конструкция. Чтобы увидеть, как это делается, создайте функцию-расширение с именем `move`, которая упростит изменение `Coordinate`.

Листинг 19.10. Объявление инфиксной функции-расширения (Extensions.kt)

```
...
operator fun List<List<Room>>.get(coordinate: Coordinate) =
    getOrNull(coordinate.y)?.getOrNull(coordinate.x)

infix fun Coordinate.move(direction: Direction) =
    direction.updateCoordinate(this)
```

Теперь используйте функцию-расширение — и инфиксную запись — в классе `Game`.

Листинг 19.11. Использование инфиксной функции-расширения (NyetHack.kt)

```
...
object Game {
    ...
    fun move(direction: Direction) {
        val newPosition = direction.updateCoordinate(currentPosition)
        val newPosition = currentPosition move direction
        val newRoom = worldMap[newPosition]

        if (newRoom != null) {
            ...
        }
    }
}
```

```

        } else {
            ...
        }
    }
    ...
}

```

Свойства-расширения

Кроме функций-расширений, добавляющих новые возможности, также можно объявлять свойства-расширения. Добавьте еще одно расширение для `String` в `Extensions.kt`; на этот раз это будет свойство-расширение, подсчитывающее гласные в строке.

Листинг 19.12. Добавление свойства-расширения (Extensions.kt)

```

fun String.addEnthusiasm(enthusiasmLevel: Int = 1) =
    this + "!".repeat(enthusiasmLevel)

val String.numVowels
    get() = count { it.lowercase() in "aeiou" }
...

```

Чтобы испытать свое новое свойство-расширение, обновите условие «Master of Vowels» в `Player`.

Листинг 19.13. Использование свойства-расширения (Player.kt)

```

class Player(
    initialName: String,
    val hometown: String = "Neversummer",
    override var healthPoints: Int,
    val isImmortal: Boolean
) : Fightable {
    ...
    val title: String
        get() = when {
            name.all { it.isDigit() } -> "The Identifiable"
            name.none { it.isLetter() } -> "The Witness Protection Member"
            name.count { it.lowercase() in "aeiou" } > 4 -> "The Master of Vowels"
            name.numVowels > 4 -> "The Master of Vowels"
            else -> "The Renowned Hero"
        }
    ...
}

```

Прежде чем тестировать свойство-расширение, необходимо внести еще одно изменение. В главе 9 мы закомментировали код, в котором игроку предлагалось ввести имя при запуске `NyetHack`. Тогда мы упомянули, что этот код вернется в программу в конце работы над `NyetHack`, — и сейчас этот момент настал.

Снова обновите `promptHeroName`, чтобы выводился запрос имени героя. А заодно раскомментируйте вызов `changeNarratorMood`.

Листинг 19.14. Запрос имени (NyetHack.kt)

```
...
fun main() {
    ...
    // changeNarratorMood()

    Game.play()
}

private fun promptHeroName(): String {
    narrate("A hero enters the town of Kronstadt. What is their name?") { message
->
        // Выводит сообщение желтым цветом
        "\u001b[33;1m$message\u001b[0m"
    }

    /*val input = readLine()
    require(input != null && input.isNotEmpty()) {
        "The hero must have a name."
    }

    return input*/
    println("Madrigal")
    return "Madrigal"
}
...

```

Запустите NyetHack. Когда вы получите запрос на имя, введите имя с множеством гласных, например Aurelia. Вы увидите, что герою присваивается титул «Master of Vowels», как и прежде:

```
Welcome to NyetHack!
A hero enters the town of Kronstadt. What is their name?
Aurelia
THE NARRATOR BEGINS TO FEEL LOUD!!!
WELCOME, ADVENTURER!!!
AURELIA, A MORTAL, HAS 100 HEALTH POINTS!!!
AURELIA OF NEVERSUMMER, THE MASTER OF VOWELS, IS IN THE TOWN SQUARE
(CURRENTLY: BUSTLING)!!!
THE VILLAGERS RALLY AND CHEER AS THE HERO ENTERS!!!
THE BELL TOWER ANNOUNCES THE HERO'S PRESENCE: GWONG!!!
> Enter your command:
```

Вспомните: в главе 13 мы говорили, что свойства класса (кроме вычисляемых свойств) имеют поля, в которых хранятся фактические данные, и для них автоматически создаются `get`-методы, а при необходимости — и `set`-методы. У свойств-расширений не может быть полей, поэтому они должны быть вычисляемыми.

Каждое свойство-расширение должно определять функцию `get` и (для `var`) `set`, вычисляющую значение, которое должно быть возвращено для свойства.

Например, следующее объявление недопустимо:

```
val String.numberOfWords = 10
error: extension property cannot be initialized because it has no backing field
```

Зато можно объявить свойство-расширение `numberOfWords` с `get`-методом для `val numberOfWords`.

Расширения для типов, допускающих null

Расширение также можно определить для типа, допускающего `null`. Объявление расширения для типа, допускающего `null`, позволит обрабатывать значение `null` в теле функции расширения, а не в точке вызова.

Добавьте в `Extensions.kt` расширение для `Room`, допускающее `null`, которое возвращает комнату где-то за пределами города Кронштадта.

Листинг 19.15. Добавление расширения для типа, допускающего `null` (`Extensions.kt`)

```
...
infix fun Coordinate.move(direction: Direction) =
    direction.updateCoordinate(this)

fun Room?.orEmptyRoom(name: String = "the middle of nowhere"): Room =
    this ?: Room(name)
```

Используйте функцию-расширение в функции `move`, чтобы игрок мог выйти за пределы города.

Листинг 19.16. Вызов расширения для типа, допускающего `null` (`NyetHack.kt`)

```
...
object Game {
    ...
    fun move(direction: Direction) {
        val newPosition = currentPosition move direction
        val newRoom = worldMap[newPosition].orEmptyRoom()

        if (newRoom != null) {
            narrate("The hero moves ${direction.name}")
            currentPosition = newPosition
            currentRoom = newRoom
        } else {
            narrate("You cannot move ${direction.name}")
        }
    }
    ...
}
```

Запустите NyetHack и попробуйте переместиться на север. Вы увидите, что герой не ограничен границами созданного мира, ему доступна пустота, окружающая город. (При этом вы также сможете исследовать другие комнаты в NyetHack, как и прежде.)

```
...
Welcome, adventurer?
Madrigal, a mortal, has 100 health points?
Madrigal of Neversummer, The Renowned Hero, is in The Town Square
(Currently: Bustling)?
The villagers rally and cheer as the hero enters?
The bell tower announces the hero's presence: GWONG?
> Enter your command: move north
The hero moves North?
Madrigal of Neversummer, The Renowned Hero, is in the middle of nowhere
(Currently: Calm)?
There is nothing to do here?
> Enter your command:
```

Расширения: как это устроено

Функции-расширения, или свойства-расширения, вызываются так же, как обычные, но они не определяются внутри класса, который расширяют, и не используют механизм наследования для добавления новых возможностей. Так как же тогда расширения реализованы в JVM?

Чтобы понять, как расширения работают в JVM, заглянем в байт-код, сгенерированный компилятором Kotlin для расширения, и переведем его на язык Java.

Подведите текстовый курсор к `Extensions.kt`, откройте окно инструментария байт-кода Kotlin, а затем выберите в меню команду **Tools ▶ Kotlin ▶ Show Kotlin Bytecode** либо введите запрос «show Kotlin bytecode» в диалоговом окне **Search Everywhere** (двойное нажатие Shift).

В окне байт-кода Kotlin щелкните на кнопке **Decompile** вверху слева, чтобы открыть новую вкладку с Java-версией байт-кода, сгенерированного из `Extensions.kt`. Найдите эквивалентный байт-код расширения `addEnthusiasm`, которое вы определили для `String`.

```
@NotNull
public static final String addEnthusiasm(@NotNull String $this$addEnthusiasm,
                                         int enthusiasmLevel) {
    Intrinsics.checkNotNullParameter($this$addEnthusiasm,
"$this$addEnthusiasm");
    return $this$addEnthusiasm +
        StringsKt.repeat((CharSequence)!",", enthusiasmLevel);
}
```

Расширения в Kotlin представляют собой статические функции, которым получатель передается в первом аргументе. Компилятор подставляет вызов функции `addEnthusiasm`. Это позволяет вам объявлять функции-расширения для каждого типа в Kotlin.

Побочный эффект такой реализации заключается в том, что функции-расширения не способны переопределять функции базового типа и не могут обращаться к приватным свойствам и функциям класса. Если вы создадите расширение с такой же сигнатурой, как у свойства или функции самого класса, компилятор отдаст предпочтение определению базового типа перед расширением. Чтобы убедиться в этом, попробуйте выполнить следующий код в REPL.

Листинг 19.17. Обработка функций-расширений (REPL)

```
val String.length
    get() = -999

"Madrigal has left the building".length
30
```

Выражение возвращает значение `30`, что свидетельствует об использовании встроенного свойства `length` вместо расширения. Будьте внимательны при объявлении расширений, чтобы избежать конфликтов с базовым типом. Мы рекомендуем использовать разные имена, но функцию можно перегрузить функцией-расширением при условии различия в параметрах.

Видимость расширений

В этой главе мы не объявляли расширения с модификатором видимости, так что они неявно являются публичными. Такие расширения доступны из любой точки кодовой базы. С течением времени может оказаться, что расширяемые классы загромаждаются множеством добавляемых новых функций и свойств.

Чтобы этого не происходило, стоит выбрать обычную функцию вместо расширения, но во многих ситуациях расширения эффективнее упрощают ваш код. Другое возможное решение основано на добавлении модификаторов видимости к расширениям — по аналогии с обычными функциями и свойствами.

Пометка расширения модификатором `private` не позволяет использовать расширение за пределами файла, в котором оно определено. Возьмем определенную вами функцию `List<List<Room>>.get`. Допустим, вы хотите, чтобы она была доступной только из `Game`, где задается логика навигации и `worldMap`. Чтобы ограничить доступ к функции и избежать загромождения списка рекомендаций автозаполнения в IDE, функцию можно переместить в `NyetHack.kt` и пометить ее модификатором `private`.

Как вы, возможно, уже поняли, функции-расширения допустимо объявлять почти со всеми модификаторами функций, которые могут использоваться при определении функций внутри класса. Важным исключением являются ключевые слова `abstract` и `open`, потому что переопределить функцию-расширение нельзя. Практически любую функцию можно оформить в виде функции-расширения при условии, что ей не нужно обращаться к внутренней реализации класса и она не нуждается в переопределении.

Расширения в стандартной библиотеке Kotlin

Большая часть функциональности стандартной библиотеки Kotlin реализована через функции-расширения и свойства-расширения.

Загляните, например, в файл `Strings.kt` (обратите внимание: `Strings`, а не `String`), выполнив поиск по имени в диалоговом окне `Search Everywhere`:

```
public inline fun CharSequence.trim(predicate: (Char) -> Boolean): CharSequence
{
    var startIndex = 0
    var endIndex = length - 1
    var startFound = false

    while (startIndex <= endIndex) {
        val index = if (!startFound) startIndex else endIndex
        val match = predicate(this[index])

        if (!startFound) {
            if (!match)
                startFound = true
            else
                startIndex += 1
        }
        else {
            if (!match)
                break
            else
                endIndex -= 1
        }
    }

    return subSequence(startIndex, endIndex + 1)
}
```

Просмотрев этот файл из стандартной библиотеки, вы обнаружите, что он состоит из расширений типа `String`. Фрагмент выше, например, объявляет функцию-расширение `trim`, которая используется для удаления символов из строки.

Имена файлов в стандартной библиотеке, которые содержат расширения, часто начинаются с имени расширяемого типа и заканчиваются на `-s`. Просматривая список файлов в стандартной библиотеке, вы увидите другие файлы, имена которых следуют этому соглашению. `Sequences.kt`, `Ranges.kt`, `Maps.kt` — лишь некоторые примеры файлов, которые добавляют новые возможности в стандартную библиотеку путем расширения соответствующих типов.

Повсеместное применение функций-расширений для определения базовой функциональности API — одна из причин, почему стандартная библиотека Kotlin при такой компактности (около 1,4 Мбайт) предлагает столь широкие возможности. Расширения очень экономно расходуют память, потому что единственное определение способно добавлять новые возможности в множество типов.

В этой главе вы узнали о расширениях, предлагающих альтернативный способ наследования общего поведения. И на этом мы прощаемся с `NyetHack`. В предыдущих главах вы познакомились со многими возможностями Kotlin: лямбда-выражениями, коллекциями, классами, наследованием, обобщениями и функциями-расширениями. В следующей главе мы создадим новый проект для изучения сопрограмм, что позволит нам выполнять асинхронные задачи в Kotlin.

Для любознательных: литералы функций с получателями

Литералы функций эффективно используются с синтаксисом расширений. Чтобы понять, что мы имеем в виду, говоря о литералах функций с получателями, рассмотрим определение `apply` — функции из главы 12:

```
public inline fun <T> T.apply(block: T.() -> Unit): T {
    block()
    return this
}
```

Вспомните, что функция `apply` позволяет задать свойства некоторого экземпляра получателя лямбда-выражением, переданным в аргументе, например:

```
val finalBossRoom = EvilLair().apply {
    lairOwner = "Nogartse"
    securityFeatures = listOf("moat", "lasers", "confusing interior layout")
    prepareBattleMusic()
}
```

Таким образом появляется возможность избежать явного вызова каждой функции для получателя. Вместо этого вы вызываете их неявно в лямбда-выражении. Волшебство `apply` достигается определением литерала функции с получателем.

Снова обратившись к определению `apply`, посмотрите, как задается параметр функции с именем `block`:

```
public inline fun <T> T.apply(block: T.() -> Unit): T {
    block()
    return this
}
```

Параметр функции `block` не только является лямбда-выражением, но и задается как расширение обобщенного типа `T` выражением `T.() -> Unit`. Это обстоятельство позволяет лямбда-выражению, которое вы определяете, также неявно получить доступ к свойствам и функциям экземпляра получателя.

Получатель лямбда-выражения, заданный как расширение, также является экземпляром, для которого вызывается `apply`, что открывает доступ к функциям и свойствам экземпляра получателя в теле лямбда-выражения.

Подобный стиль позволяет создавать так называемые предметно-ориентированные языки, или DSL (Domain Specific Languages). Это стиль API, где можно выразить функции и особенности контекста получателя, который вы настраиваете с помощью лямбда-выражений, объявленных для доступа к этим функциям. Например, фреймворк `Exposed` от JetBrains (github.com/JetBrains/Exposed) широко использует стиль DSL для API, позволяющего определять запросы SQL.

В `NyetHack` можно добавить функцию, которая использует такой стиль для настройки комнаты со злобным гоблином. (При желании добавьте эту возможность в проект `NyetHack` в порядке эксперимента.)

```
inline fun MonsterRoom.configurePitGoblin(
    block: MonsterRoom.(Goblin) -> Goblin
): MonsterRoom {
    val goblin = block(Goblin("Pit Goblin", description = "An Evil Pit Goblin"))
    monster = goblin
    return this
}
```

Расширение `Room` получает лямбда-выражение с получателем `Room`. В результате свойства `Room` становятся доступными в определяемом вами лямбда-выражении, так что гоблина можно настроить через свойства получателя `Room`. Например, вызов функции-расширения в классе `Game` может выглядеть так:

```
object Game {
    ...

    fun configureCurrentRoom() {
        val monsterRoom = currentRoom as? MonsterRoom ?: return
    }
}
```

```

        monsterRoom.configurePitGoblin { goblin ->
            goblin.healthPoints = when {
                "Haunted" in name -> 60
                "Dungeon" in name -> 45
                "Town Square" in name -> 15
                else -> 30
            }
            goblin
        }
    }
}

```

Задание: расширение рамок

Следующий пример — маленькая программа, позволяющая вывести произвольную строку в красивой ASCII-рамке, которую можно распечатать и повесить на стену:

```

fun frame(name: String, padding: Int, formatChar: String = "*"): String {
    val greeting = "$name!"
    val middle = formatChar
        .padEnd(padding)
        .plus(greeting)
        .plus(formatChar.padStart(padding))
    val end = (0 until middle.length).joinToString("") { formatChar }
    return "$end\n$middle\n$end"
}

```

В этом задании вам понадобятся знания о расширениях. Попробуйте преобразовать функцию `frame` в расширение, которое можно использовать с любой строкой. А вот и пример того, что у вас должно получиться:

```

print("Welcome, Madrigal".frame(5))
*****
*   Welcome, Madrigal   *
*****

```

Подписывайся на самый топовый канал по Kotlin:
<https://t.me/KotlinSenior>

20. Сопрограммы

Приложения могут выполнять разнообразные задачи, в том числе и подключаться к внешним ресурсам. Например, иногда требуется, чтобы приложение скачивало данные, обращалось к базам данных или к веб-службам. Эти операции полезны, но они требуют времени для выполнения.

Чтобы не заставлять пользователей ждать завершения продолжительных операций, следует перевести эту работу в *фоновый режим*. Если этого не сделать, программа *блокируется*, то есть перестает реагировать на любые другие события, и все выглядит так, словно она зависла. *Сопрограммы* позволяют выполнять работу в фоновом режиме, или, другими словами, *асинхронно*.

В Kotlin 1.3 появилась стабильная поддержка сопрограмм, но сами сопрограммы не являются чем-то новым или специфическим для Kotlin. Концепция сопрограмм возникла в 1950-е годы, и они были реализованы во многих языках программирования. В основе лежит идея, что функции должны быть способны *приостанавливаться* до завершения продолжительной операции.

Многие другие языки программирования для выполнения асинхронной работы полагаются исключительно на концепцию *программных потоков* (threads), или просто потоков. Программные потоки управляют выполнением вашей программы. Каждый поток выполняет собственную последовательность инструкций в том порядке, в каком они были объявлены. Первичный поток, управляющий операциями, с которыми пользователь взаимодействует напрямую, называется *главным потоком*, или *UI-потоком*.

Традиционно разработчики перемещают долго выполняющиеся операции, например выполнение сетевых запросов, в фоновые потоки. Так главный поток освобождается для своих собственных задач, таких как прорисовка пользовательского интерфейса приложения. В этой модели асинхронных вычислений главный поток запускает фоновый поток для выполнения сетевого запроса. После того как сетевой запрос будет выполнен, фоновый поток возвращает данные главному потоку.

Такая реализация обладает рядом недостатков. Операции с потоками осуществляются через низкоуровневый API, что усложняет работу с ними. Также при прямых операциях с потоками очень легко совершить ошибку, а ошибки часто приводят к неэффективному расходованию ресурсов или неожиданным фатальным сбоям приложения.

Сопрограммы предоставляют высокоуровневый и более безопасный набор инструментов для построения асинхронного кода. Под капотом сопрограммы Kotlin используются потоки для параллельного выполнения работы, но обычно вам не приходится вникать в такие подробности.

Когда сопрограмма применяется для какой-либо задачи, например создания сетевых запросов, код, выполняющий запрос, приостанавливается на время выполнения задачи. В это время другие части вашей программы могут исполняться; тем самым главный поток освобождается, а программа реагирует на действия пользователя. После завершения сетевого запроса приостановленный код возобновляет работу с точки останова.

За кулисами Kotlin сохраняет и восстанавливает состояние функции между приостановкой вызовов. Это позволяет временно выгрузить исходный вызов функции из памяти до того, как он будет готов к продолжению.

Благодаря такой оптимизации сопрограммы расходуют ресурсы намного эффективнее нативных потоков. Как мы покажем в этой главе, сопрограммы по своему поведению очень близки к синхронному коду, с которым вы уже работали в книге.

Итак, наш герой Мадригал изрядно потрудился и ему отчаянно нужен отпуск. Для этого он заказывает билет на тропический остров, где нет гоблинов. Чтобы он оказался на борту самолета вовремя, мы применим в новом проекте сопрограммы для загрузки данных о полете и о том, когда являться на посадку.

К концу главы наш проект будет взаимодействовать с двумя веб-службами — для получения информации о рейсе и о статусе героя в программе лояльности авиакомпании TaernylAir, что влияет на выбор места в процессе посадки. Эта информация объединяется в один объект `FlightStatus`. В главе 21 мы будем генерировать инструкции по посадке, используя потоки данных. Наконец, в главе 22 мы ускорим получение данных при помощи *каналов* — механизма передачи данных между сопрограммами.

Блокирующие вызовы

Первая веб-служба, к которой мы обратимся, доступна по адресу `kotlin-book.bignerdranch.com/2e/flight`. При запросе информации служба возвращает данные о рейсе, разделенные запятыми, с атрибутами `flightNumber` (номер рейса), `originAirport` (аэропорт вылета), `destinationAirport` (аэропорт прибытия) и `departureTimeInMinutes` (время вылета).

Откройте URL в своем браузере. (Будьте терпеливы. Страница загружается с пятисекундной задержкой, пока система обращается к своей огромной базе данных рейсов.) Вы увидите страницу с одной строкой, которая выглядит примерно так:

```
JS1112,UJH,WUI,On Time,88
```

Несколько раз перезагрузите страницу, чтобы увидеть разные ответы. При каждой загрузке страницы генерируются случайные данные. В этой главе мы построим клиент, который принимает ответы этого API и выводит их в консоль.

Создайте новый проект Kotlin с именем `TaernylAir`. Используйте шаблон `Application` из группы `JVM`, не забудьте задать значение `Project JDK`. Также проследите, чтобы была выбрана система сборки `Gradle Groovy`. Позднее в этой главе мы изменим файлы, относящиеся к системе сборки, поэтому для других систем сборки приведенное описание не подойдет.

В новом проекте создайте новый файл с именем `FlightFetcher.kt`. В нем определите две константы, `BASE_URL` и `FLIGHT_ENDPOINT`, для конечных точек API.

Также создайте новую функцию с именем `fetchFlight`, которая возвращает строку с представлением данных, полученных от конечной точки. Kotlin включает в класс `URL` языка Java функцию-расширение `readText`, которая поддерживает подключение к базовой конечной точке API, буферизацию данных и преобразование данных в строку — все, что нам понадобится.

Наконец, вызовите `fetchFlight` из новой функции `main` и выведите результат.

Листинг 20.1. Получение данных рейса (`FlightFetcher.kt`)

```
private const val BASE_URL = "http://kotlin-book.bignerdranch.com/2e"
private const val FLIGHT_ENDPOINT = "$BASE_URL/flight"

fun main() {
    val flight = fetchFlight()
    println(flight)
}

fun fetchFlight(): String = URL(FLIGHT_ENDPOINT).readText()
```

Выполните функцию `main` в `FlightFetcher.kt`. Вы заметите, что для возвращения данных этому вызову требуется некоторое время (зависит от скорости вашего подключения к интернету). После завершения добавьте команды отладочного вывода, чтобы контролировать, когда запрос запускается и завершается.

Листинг 20.2. Хронометраж запроса (`FlightFetcher.kt`)

```
...
fun main() {
    println("Started")
    val flight = fetchFlight()
    println(flight)
    println("Finished")
}

fun fetchFlight(): String = URL(FLIGHT_ENDPOINT).readText()
```

Снова запустите функцию `main` и проследите за консольным выводом. Обратите внимание на промежуток времени между выводом сообщений `Started` и `Finished`.

Из-за задержки конечной точки вызов `fetchFlight` вернет данные приблизительно через пять секунд. Поток напоминает конвейер, на котором выполняется серия операций; на время ожидания получения данных от `fetchFlight` поток блокируется — он не может быть использован ни для какой другой работы, пока не освободится.

Пять секунд — это долго, но ответы веб-служб нередко занимают много времени, особенно при плохом сетевом подключении или большом теле ответа. Чтобы совершать подобные длительные вызовы, не тратя времени пользователя (и не испытывая его терпения), следует переместить работу в отдельный поток. Это позволит выполнять другую работу во время выполнения продолжительной операции.

Включение сопрограмм

Сопрограммы, как и потоки, предоставляют механизм выполнения асинхронной, потенциально продолжительной работы в фоновом режиме. В отличие от потоков, сопрограммы могут выполняться и ожидать завершения другой работы без блокирования потока, в котором они запущены.

Kotlin не включает встроенную поддержку сопрограмм. Чтобы использовать их в проекте, необходимо добавить в состав зависимостей проекта библиотеку, в которой сопрограммы определены. Зависимостями управляет Gradle — система сборки, выбранная нами при создании проекта `TaetnylAir`. Проекты Gradle в основном настраиваются в файлах с расширением `.gradle`. В этих файлах хранится информация об используемой версии Kotlin, необходимых проекту зависимостях, настройках, определяющих генерирование вывода в программах, и многом другом.

Пока достаточно зарегистрировать одну зависимость в Gradle. Найдите и откройте файл `build.gradle` в каталоге проекта верхнего уровня, добавьте в него зависимость `Coroutines`.

Листинг 20.3. Включение поддержки сопрограмм (build.gradle)

```
plugins {  
    id 'org.jetbrains.kotlin.jvm' version '1.5.21'  
}  
  
group = 'com.bignerdranch'  
version = '1.0-SNAPSHOT'  
  
repositories {  
    mavenCentral()  
}
```

```
dependencies {
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.1"
    testImplementation 'org.jetbrains.kotlin:kotlin-test'
}

test {
    useJUnit()
}

compileKotlin {
    kotlinOptions.jvmTarget = '1.8'
}

compileTestKotlin {
    kotlinOptions.jvmTarget = '1.8'
}
```

Добавив новую запись в файл `build.gradle`, щелкните на кнопке  **Load Gradle Changes**, которая появится в верхнем правом углу экрана, для синхронизации файлов Gradle.

Кстати говоря, библиотека `Coroutines` также поддерживает каналы и потоки данных, которые мы используем в следующих главах.

Строители сопрограмм

Строитель сопрограммы — функция, создающая новую сопрограмму. Многие строители также запускают сопрограмму сразу же после ее создания. В библиотеке `Coroutines` вы найдете несколько готовых строителей сопрограмм.

Самым популярным из них является `launch` — функция, определенная в расширении класса `CoroutineScope`. Немного позже мы рассмотрим тему областей видимости, а пока используем подкласс `GlobalScope`.

Чтобы запустить новую сопрограмму, упакуйте вызов `fetchFlight` в вызов функции `launch`, определенной в `GlobalScope`.

Листинг 20.4. Запуск сопрограммы (FlightFetcher.kt)

```
private const val BASE_URL = "http://kotlin-book.bignerdranch.com/2e"
private const val FLIGHT_ENDPOINT = "$BASE_URL/flight"

fun main() {
    println("Started")
    GlobalScope.launch {
        val flight = fetchFlight()
        println(flight)
    }
    println("Finished")
}

fun fetchFlight(): String = URL(FLIGHT_ENDPOINT).readText()
```

Функция `launch` получает один аргумент — лямбда-выражение с определением работы, которая должна выполняться без блокирования текущего потока.

Запустите новую версию приложения. На этот раз сообщения `Started` и `Finished` выводятся, но процесс завершится до того, как ваш запрос вернет данные. Что происходит?

Вы создали сопрограмму вызовом функции `launch`. Она запускает работу, заданную в новой сопрограмме, немедленно.

Область видимости сопрограммы определяется объектом `CoroutineScope`, где она определяется. Вы вызвали `launch` для `GlobalScope`, поэтому этот класс является областью видимости вашей сопрограммы и определяет, как долго она будет выполняться. `GlobalScope` — это практически неуправляемая реализация `CoroutineScope`: все сопрограммы объединяются в общий *пул потоков*.

Возможно, вы заметили предупреждение, которое появилось в коде при использовании `GlobalScope`. Этот класс может создать проблемы при некорректном использовании. Например, если запустить сопрограмму в `GlobalScope`, когда пользователь входит в некоторую часть приложения, но забыть отменить ее при выходе, связанные с сопрограммой ресурсы не будут освобождены и ваша сопрограмма продолжит поглощать память в фоновом режиме.

Из-за этого `GlobalScope` считается «чувствительным» API. IntelliJ пытается направить вас к другим API, которые считаются более безопасными. Чтобы в этом разобраться, исследуем концепцию области видимости сопрограмм.

Области видимости сопрограмм

Каждый строитель запускает свои сопрограммы внутри *области видимости сопрограммы*. Область видимости сопрограммы управляет выполнением кода сопрограммы: подготовкой ее к выполнению, ее отменой, выбором потока, используемого для выполнения кода, и т. д. На данный момент мы используем `GlobalScope` как область видимости сопрограммы для выполнения сетевых запросов.

`GlobalScope` предоставляет простые средства для запуска сопрограмм, поэтому этот класс хорошо подходит для первого примера. Но мы не рекомендуем использовать его в реальных приложениях, так как он способен создать проблемы. Мы в этом уже убедились: `GlobalScope` не поддерживает процесс достаточно долго, чтобы дождаться возвращения запроса и вывода результата.

До настоящего момента мы говорили об асинхронном коде так, словно блокирования всегда нужно избегать, но на самом деле это не так. Иногда код *должен* блокироваться для завершения критической задачи. В `TaarnylAir` блокирование `fetchFlight` нежелательно, но функция `main` определенно должна блокироваться до завершения работы в `fetchFlight`. Это гарантирует, что у `fetchFlight` будет возможность вернуть данные.

Функция `runBlocking` — строитель сопрограммы, который блокирует свой поток до того момента, пока выполнение сопрограммы не завершится. Функцию `runBlocking` используют для запуска сопрограмм, которые все должны завершиться до возобновления работы. Примеры такого рода мы покажем в главах 21 и 22.

Упакуйте функцию `launch` в `main` в вызов `runBlocking`. Теперь, когда вы используете область видимости сопрограммы этого строителя, можно отказаться от использования `GlobalScope`.

Листинг 20.5. Использование блокирующего строителя сопрограммы (`FlightFetcher.kt`)

```
...
fun main() {
    println("Started")
    GlobalScope.launch {
        runBlocking {
            println("Started")
            launch {
                val flight = fetchFlight()
                println(flight)
            }
            println("Finished")
        }
    }
    println("Finished")
}
fun fetchFlight(): String = URL(FLIGHT_ENDPOINT).readText()
```

Запустите `TaarnylAir`. Вывод должен выглядеть примерно так:

```
Started
Finished      // пауза
CE7902,FVY,CLI,On Time,116
```

Сначала выводится сообщение `Started`, так как его вызов `println` выполняется первым, а функция `println` не приостанавливает выполнение. Затем идет вызов `fetchFlight`, но так как он был запущен в отдельной сопрограмме, второй вызов `println` не ожидает его завершения. На получение данных рейса требуется намного больше времени, чем для вывода на консоль, поэтому сообщение `Finished` предшествует появлению данных о рейсе.

Структурированный параллелизм

Во внутренней реализации каждый экземпляр `CoroutineScope` содержит контекст сопрограммы `CoroutineContext`. Контекст сопрограммы — это набор правил, определяющих, как сопрограмма должна выполняться, а область видимости сопрограммы — наблюдатель, следящий за выполнением сопрограммы в соответствии с этими правилами. `CoroutineContext` также делится на меньшие компоненты, определяющие правила выполнения. Наиболее часто в качестве элементов,

образующих контекст сопрограммы, используются `Job` и `CoroutineDispatcher` (рис. 20.1).

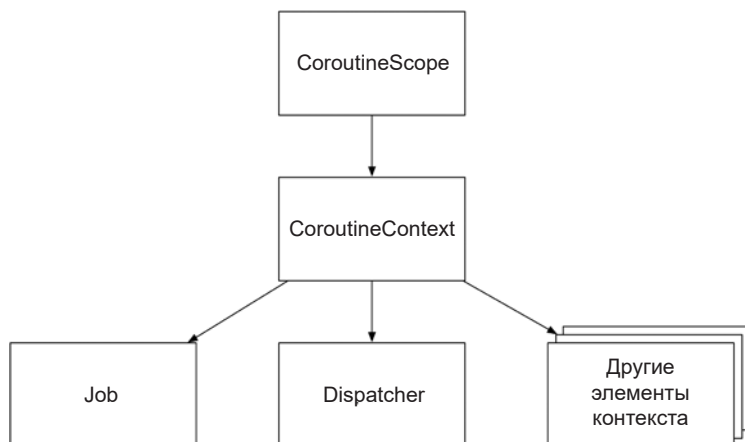


Рис. 20.1. Анатомия `CoroutineScope`

Экземпляр `Job` сопрограммы хранит информацию о состоянии сопрограммы (например, выполняется ли она в настоящее время). Также он предоставляет средства для досрочной отмены сопрограммы. Каждый строитель возвращает соответствующий экземпляр `Job` при запуске сопрограммы. Это означает, что вам доступна информация о выполнении и вы можете вручную отменить сопрограмму перед завершением выполнения. Ручная отмена особенно удобна, если требуется остановить продолжительную фоновую задачу по инициативе пользователя, например, если тот покидает страницу во время загрузки или отменяет передачу файла.

Диспетчер `CoroutineDispatcher` отвечает за запуск сопрограммы. В общем случае это подразумевает планирование времени запуска или перемещение работы сопрограммы в нужный поток. В вашем распоряжении есть несколько готовых вариантов диспетчера, и вы также можете создать собственный диспетчер. Наиболее часто используемые встроенные диспетчеры находятся в классе с именем `Dispatchers`.

<code>Dispatchers.Default</code>	Рекомендуется для работы общего характера и операций, сопряженных с высокой вычислительной нагрузкой. Базируется на пуле потоков, размер которого ограничивается количеством ядер у процессора устройства.
<code>Dispatchers.IO</code>	Рекомендуется для операций, относящихся к вводу-выводу. Базируется на пуле потоков с большим количеством потоков.
<code>Dispatchers.Main</code>	Выполняет код в главном потоке (UI-потоке). На JVM требуется дополнительная зависимость для Android, JavaFX и Swing, чтобы обозначить, какой поток является главным. На других платформах ведет себя так же, как <code>Dispatchers.Default</code> .

`Dispatchers.Unconfined` — Указывает, что выбор потока, в котором будет выполняться эта работа, ни на что не влияет. Библиотека `Coroutines` продолжит выполнение сопрограммы в потоке, который она уже использует.

Каждый вариант `CoroutineScope` имеет собственный контекст для выполнения сопрограммы, но вы также можете изменить контекст сопрограммы для части работы. Для этой цели используют функцию `withContext`.

Опробуйте эту возможность, заставив функцию `fetchFlight` выполняться с `Dispatchers.IO`. Функцию необходимо пометить ключевым словом `suspend`, смысл которого мы объясним позднее.

Листинг 20.6. Использование контекста сопрограммы (FlightFetcher.kt)

```
...
fun main() {
    runBlocking {
        println("Started")
        launch {
            val flight = fetchFlight()
            println(flight)
        }
        println("Finished")
    }
}

suspend fun fetchFlight(): String = withContext(Dispatchers.IO) {
    URL(ENDPOINT).readText()
}
```

Теперь работа функции `fetchFlight` выполняется в потоке под управлением `Dispatchers.IO`. Сопрограмма, запущенная в `main`, выполняется в главном потоке, который блокируется до завершения всех сопрограмм.

При переключении контекста сопрограммы функцией `withContext` новый контекст объединяется с предыдущим. В результате новый контекст наследует от контекста родительской сопрограммы.

В приведенном примере для нового контекста был назначен другой диспетчер, который переопределяет диспетчер из контекста родительской сопрограммы (не объединяется с ним!).

Очень важно, что дочерние сопрограммы также наследуют родительский экземпляр задания `Job` в дополнение к получению собственного. В отличие от диспетчера, родительское задание не переопределяется — оба задания могут управлять выполнением сопрограммы. Если родительское задание отменяется, то задания всех дочерних сопрограмм (и их потомки) также будут отменены.

Эта концепция, называемая *структурированным параллелизмом*, часто встречается при использовании сопрограмм в Kotlin. Поведение наследования контекстов

сопрограмм также можно увидеть при запуске сопрограммы внутри другой сопрограммы, например при вызове `launch` внутри `runBlocking`.

Структурированный параллелизм представляет парадигму организации сопрограмм. Вы оцените его, когда начнете чаще пользоваться сопрограммами. И хотя структурированный параллелизм проявляется не только в сопрограммах Kotlin, они являются одной из самых известных его реализаций.

Чтобы вызвать `withContext`, необходимо пометить `fetchFlight` модификатором приостановки `suspend`. Приостанавливаемые функции (такие, как `withContext`) могут вызываться только из других приостанавливаемых функций или из строителя сопрограмм. Функция `withContext` приостанавливается на то время, пока сетевые взаимодействия происходят в другом потоке. После завершения сетевого запроса `withContext` продолжает выполнение с исходным диспетчером, и функция `fetchFlight` продолжится с той точки, в которой она была приостановлена.

Есть много причин для пометки функций как приостанавливаемых. В ближайших двух главах мы как раз и займемся созданием собственных приостанавливаемых функций.

В той точке, где ваша функция `main` вызывает `fetchFlight`, IntelliJ добавляет значок . Этот же значок встречается в строке с вызовом `withContext`. Он указывает, что в этой строке вызывается приостанавливаемая функция, и напоминает, где сопрограмма может приостанавливаться и возобновляться.

Запустите `TaernylAir` и убедитесь, что вывод не изменился. Передача данных по сети будет происходить в потоке ввода-вывода, но так как вы ожидаете завершения сетевого запроса, прежде чем выполнять какую-либо другую работу, никаких улучшений в поведении приложения вы не заметите.

Использование клиента HTTP

Проект `TaernylAir` теперь способен загружать информацию о рейсах, но этого недостаточно, чтобы герой оказался в самолете. Существуют жесткие правила, регулирующие посадку пассажиров. Например, выход на посадку закрывается за 15 минут до вылета, и все пассажиры к этому моменту должны быть на борту.

Кроме того, герой участвует в программе лояльности. Каждый уровень дает пассажиру определенный приоритет при посадке, а уровни зависят от того, сколько миль пассажир налетал с авиакомпанией.

Чтобы вывести точную посадочную информацию для пассажира, необходимо запросить информацию об уровне лояльности с конечной точки `kotlin-book.bignerdranch.com/2e/loyalty`.

Но прежде чем приступать к добавлению новых вызовов API, взгляните еще раз на функцию `fetchFlight`. Среда IntelliJ выделила использование класса `URL`; если навести на него указатель мыши, появляется предупреждение о некорректном

вызове блокирующего метода. Проблема в том, что URL и `readText` блокируют поток внутри сопрограммы. Блокирование потока не позволяет сопрограмме приостановиться, а это приводит к связыванию ресурсов, которые могли бы использоваться библиотекой `Coroutines` для выполнения сопрограмм, готовых к возобновлению работы.

На практике это не повлияет на ваш код, потому что вы используете `withContext(Dispatchers.IO)`. Диспетчер `Dispatchers.IO` спроектирован для работы, связанной с вводом-выводом, и в его распоряжении находится большое количество потоков, поэтому блокировка одного из потоков в пуле `Dispatchers.IO` вряд ли создаст проблемы. Тем не менее вам все равно стоит прислушаться к совету IDE и обновить реализацию, чтобы она использовала клиент HTTP с поддержкой приостановки.

Есть несколько библиотек, хорошо работающих с сопрограммами (включая популярную библиотеку `Retrofit`). В нашем проекте мы задействуем `Ktor`; эта сетевая библиотека является частью `Kotlinx` — богатого набора дополнительных библиотек от `JetBrains`, расширяющих базовую функциональность языка `Kotlin` и его стандартной библиотеки. Используемая нами библиотека `Coroutines` также является частью `Kotlinx`.

У библиотек `Kotlinx` есть одна уникальная особенность: они отлично совместимы с `Kotlin Multiplatform`, поэтому рекомендуются для проектов, которые предполагают совместное использование кода на разных платформах. Вы больше узнаете об этих средствах код-шеринга в главе 24.

Чтобы включить поддержку `Ktor` в проекте, снова обновите файл `build.gradle` и включите в него необходимые зависимости.

Листинг 20.7. Добавление зависимости `Ktor` (`build.gradle`)

```
...
dependencies {
    implementation "org.jetbrains.kotlin:kotlinx-coroutines-core:1.5.1"
    implementation "io.ktor:ktor-client-core:1.6.2"
    implementation "io.ktor:ktor-client-cio:1.6.2"
    testImplementation 'org.jetbrains.kotlin:kotlin-test'
}
...
```

Первая зависимость, объявленная в этом фрагменте, позволяет использовать клиент HTTP с `Ktor`. Вторая относится к ядру, которое применяет `Ktor` для выполнения сетевых запросов. Есть несколько ядер, из которых вы можете выбрать нужное; `CIO` (`Coroutine-based I/O`) — простая библиотека, которая поддерживает JVM и не имеет других зависимостей.

После обновления файла `build.gradle` необходимо щелкнуть на кнопке  **Load Gradle Changes**, чтобы подключить новые зависимости.

После настройки зависимостей Ktor обновите функцию `fetchFlight`, чтобы она использовала `HttpClient` вместо `URL`. Также можно удалить вызов `withContext`, потому что Ktor автоматически перемещает сетевой запрос в фоновый поток и приостанавливается до его завершения. (Кроме того, необходимо добавить команды `import` для классов `HttpClient` и `CIO`, а также функции `get`. Не забудьте добавить правильные директивы `import`, в противном случае вы можете столкнуться с неожиданными ошибками.)

Листинг 20.8. Переход на `HttpClient` (`FlightFetcher.kt`)

```
import io.ktor.client.HttpClient
import io.ktor.client.engine.cio.CIO
import io.ktor.client.request.get
import kotlinx.coroutines.*

...
suspend fun fetchFlight(): String = withContext(Dispatchers.IO) {
    URL(FLIGHT_ENDPOINT).readText()
    val client = HttpClient(CIO)
    return client.get<String>(FLIGHT_ENDPOINT)
}
```

Перезапустите приложение и убедитесь, что оно по-прежнему загружает информацию о рейсе.

После рефакторинга вызовов API можно реализовать второй вызов API, который будет загружать информацию о статусе программы лояльности. Создайте другую константу для новой конечной точки и запросите информацию.

Листинг 20.9. Вызов второй конечной точки (`FlightFetcher.kt`)

```
private const val BASE_URL = "http://kotlin-book.bignerdranch.com/2e"
private const val FLIGHT_ENDPOINT = "$BASE_URL/flight"
private const val LOYALTY_ENDPOINT = "$BASE_URL/loyalty"

...
suspend fun fetchFlight(): String {
    val client = HttpClient(CIO)
    return val flightResponse = client.get<String>(FLIGHT_ENDPOINT)
    val loyaltyResponse = client.get<String>(LOYALTY_ENDPOINT)

    return "$flightResponse\n$loyaltyResponse"
}
```

Снова запустите приложение. Результат должен выглядеть так:

```
Started
Finished
VA4520,RXF,PBY,On Time,95
Platinum,90781,9218
```

Пока неплохо. Но что делать с этими данными? Чтобы приложение стало более удобным для пользователя и с целью улучшить структуру кода, создайте новый класс `FlightStatus` в отдельном файле. Этот класс будет парсить результаты, полученные от конечных точек, и хранить информацию о рейсе и пассажире, используя приложение для получения посадочной информации. Такую модель мы более широко применим в главе 21.

Листинг 20.10. Настройка `FlightStatus` (`FlightStatus.kt`)

```
data class FlightStatus(  
    val flightNumber: String,  
    val passengerName: String,  
    val passengerLoyaltyTier: String,  
    val originAirport: String,  
    val destinationAirport: String,  
    val status: String,  
    val departureTimeInMinutes: Int  
) {  
  
    companion object {  
        fun parse(  
            flightResponse: String,  
            loyaltyResponse: String,  
            passengerName: String  
        ): FlightStatus {  
            val (flightNumber, originAirport, destinationAirport, status,  
                departureTimeInMinutes) = flightResponse.split(",")  
  
            val (loyaltyTierName, milesFlown, milesToNextTier) =  
                loyaltyResponse.split(",")  
  
            return FlightStatus(  
                flightNumber = flightNumber,  
                passengerName = passengerName,  
                passengerLoyaltyTier = loyaltyTierName,  
                originAirport = originAirport,  
                destinationAirport = destinationAirport,  
                status = status,  
                departureTimeInMinutes = departureTimeInMinutes.toInt()  
            )  
        }  
    }  
}
```

Когда новый класс `FlightStatus` будет готов, обновите функцию `fetchFlight`, чтобы она возвращала новый тип и запрашивала имя пассажира.

Листинг 20.11. Парсинг данных рейса (FlightFetcher.kt)

```

...
fun main() {
    runBlocking {
        println("Started")
        launch {
            val flight = fetchFlight("Madrigal")
            println(flight)
        }
        println("Finished")
    }
}

suspend fun fetchFlight(passengerName: String): String FlightStatus {
    val client = HttpClient(CIO)
    val flightResponse = client.get<String>(FLIGHT_ENDPOINT)
    val loyaltyResponse = client.get<String>(LOYALTY_ENDPOINT)

    return "$flightResponse\n$loyaltyResponse"
    return FlightStatus.parse(
        passengerName = passengerName,
        flightResponse = flightResponse,
        loyaltyResponse = loyaltyResponse
    )
}

```

Снова запустите приложение. Вывод должен измениться; теперь он выглядит приблизительно так:

```

Started
Finished
FlightStatus(flightNumber=YY8272, passengerName=Madrigal,
    passengerLoyaltyTier=Gold, originAirport=GWX, destinationAirport=LFX,
    status=On Time, departureTimeInMinutes=66)

```

Итак, у авиакомпании есть вся информация, чтобы направить пассажиров на посадку. Но реализация оставляет желать лучшего.

Хотя приостанавливаемые функции используются для предотвращения блокировки главного потока, все команды по-прежнему выполняются последовательно. Сопрограмма просто указывает, что работу можно приостановить, дождаться завершения некоторой операции, а затем возобновить. При этом команды функции выполняются по одной и в объявленном вами порядке.

В `fetchFlight` выдаются два сетевых запроса, но из-за того, каким образом вы их объявили, запрос статуса лояльности будет отправлен только после завершения загрузки информации о рейсе.

Ранее мы упоминали, что конечная точка `flight` имеет 5-секундную задержку. Конечная точка `loyalty` тоже — 2-секундную. Чтобы дополнительная задержка

стала более заметной, добавьте еще пару команд отладочного вывода в функцию `fetchFlight`.

Листинг 20.12. Вывод ответов (`FlightFetcher.kt`)

```
...
suspend fun fetchFlight(passengerName: String): FlightStatus {
    val client = HttpClient(CIO)

    println("Started fetching flight info")
    val flightResponse = client.get<String>(FLIGHT_ENDPOINT).also {
        println("Finished fetching flight info")
    }

    println("Started fetching loyalty info")
    val loyaltyResponse = client.get<String>(LOYALTY_ENDPOINT).also {
        println("Finished fetching loyalty info")
    }

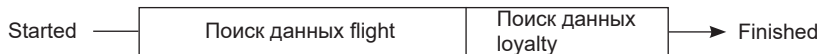
    println("Combining flight data")
    return FlightStatus.parse(...)
}
```

Снова запустите приложение. Вывод должен начинаться так:

```
Started
Finished
Started fetching flight info
Finished fetching flight info
Started fetching loyalty info
Finished fetching loyalty info
Combining flight data
FlightStatus(...)
```

На рис. 20.2 изображена временная диаграмма сравнения двух последовательных и параллельных сетевых запросов к конечным точкам `flight` и `loyalty`.

Последовательные



Параллельные

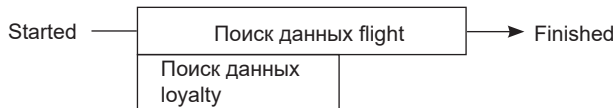


Рис. 20.2. Последовательные vs параллельные запросы

При параллельном выполнении поиск данных о статусе лояльности начнется и завершится до того, как информация о рейсе вернет управление с результатом. Если удастся структурировать сетевые запросы подобным образом, функция `fetchFlight` будет выполняться быстрее. Проще всего сделать это при помощи ключевых слов `async` и `await`.

async и await

`async` — строитель сопрограмм, который используют как альтернативу для `launch`. Как и `launch`, `async` получает в аргументе лямбда-выражение с другим приостанавливаемым кодом.

Две функции различаются прежде всего возвращаемыми типами: `launch` возвращает `Job`, а `async` — экземпляр `Deferred`.

Эти два класса похожи, более того, `Deferred` наследует от `Job`. Но кроме включения информации о статусе сопрограммы, `Deferred` также получает доступ к значению, возвращаемому сопрограммой. `Deferred` — значение, которое не всегда присутствует в данный момент, но будет доступно позже. Экземпляр `Deferred`, возвращаемый `async`, получит значение, возвращенное лямбда-выражением, как только оно будет готово.

Такая возможность чрезвычайно полезна, потому что работа, передаваемая `async`, запускается и выполняется независимо от остального кода сопрограммы. Вы можете столкнуться только с остаточной задержкой, когда понадобится обратиться к значению.

Чтобы обратиться к отложенному значению, вызовите `await` с `Deferred`. Функция `await` также является приостанавливающей: при вызове она немедленно возвращает результат, если работа завершилась; в противном случае приостанавливается до того момента, когда значение будет готово.

Используя `async` и `await`, обновите `fetchFlight` для параллельного выполнения сетевых запросов. Также добавьте небольшую задержку перед слиянием результатов, чтобы вам было проще увидеть, что они выполняются одновременно.

Листинг 20.13. Использование `async` и `await` (`FlightFetcher.kt`)

```
...
suspend fun fetchFlight(passengerName: String): FlightStatus = coroutineScope {
    val client = HttpClient(CIO)

    println("Started fetching flight info")
    val flightResponse = async {
        println("Started fetching flight info")
        client.get<String>(FLIGHT_ENDPOINT).also {
            println("Finished fetching flight info")
        }
    }
}
```

```
}

println("Started fetching loyalty info")
val loyaltyResponse = async {
    println("Started fetching loyalty info")
    client.get<String>(LOYALTY_ENDPOINT).also {
        println("Finished fetching loyalty info")
    }
}

delay(500)
println("Combining flight data")
return FlightStatus.parse(
    passengerName = passengerName,
    flightResponse = flightResponse.await(),
    loyaltyResponse = loyaltyResponse.await()
)
}
```

Строители сопрограмм, такие как `async` и `launch`, могут вызываться только для объекта `CoroutineScope`. К сожалению, модификатор `suspend` функции не предоставляет прямого доступа к области видимости, в которой выполняется функция. Для получения области видимости сопрограммы приходится действовать более творчески.

Ранее для получения `CoroutineScope` мы использовали `GlobalScope`, `runBlocking` и `withContext`. Ни один из этих способов в данном случае не годится, так как они либо не позволяют пользоваться структурированным параллелизмом, либо приводят к неожиданным побочным эффектам. Если вы используете `GlobalScope` и сопрограмма, вызвавшая функцию, отменяется, то у любой работы, переданной `GlobalScope`, ресурсы освобождены не будут. `withContext` может заставить вашу функцию использовать диспетчер, отличный от диспетчера исходной области видимости вызова. А `runBlocking` вообще не позволит вашей функции приостановиться, что нежелательно.

Функции `fetchFlight` необходимо разрешить приостановку; она должна выполняться с тем же диспетчером, для которого была вызвана; и она должна останавливаться, если используется с заданием, которое было отменено. `coroutineScope` — единственный вариант, удовлетворяющий всем этим требованиям, поэтому мы его и применяем. При вызове `coroutineScope` создается новая область видимости сопрограммы, но она наследует диспетчеру области видимости вызова и будет добавлена как потомок `Job` области видимости вызова. Добавление в качестве потомка означает, что она будет остановлена в случае отмены родительской области видимости.

Этот вариант идеально подходит для наших целей, так как он предоставляет доступ к строителю сопрограммы `async` при сохранении диспетчеров и соблюде-

нии области видимости жизненного цикла, заданной функцией, которая вызвала `fetchFlight`.

Снова запустите `TaernylAir`. Вывод будет выглядеть примерно так, как показано ниже; из него видно, что вы используете мощь сопрограмм для параллельного выполнения нескольких сетевых запросов.

```
Started
Finished
Started fetching flight info
Started fetching loyalty info
Combining flight data
Finished fetching loyalty info
Finished fetching flight info
FlightStatus(flightNumber=GM2813, passengerName=Madrigal,
passengerLoyaltyTier=Platinum, originAirport=AJA, destinationAirport=IEE,
status=Canceled, departureTimeInMinutes=52)
```

В этой главе мы написали асинхронный код, который позволяет приложению выполнять параллельную работу с использованием официальной библиотеки Kotlin `Coroutines`. В следующей главе вы пополните свой инструментарий сопрограмм новыми средствами: мы займемся тем, как передать пассажирам информацию о посадке.

Для любознательных: состояние гонки

Каждый раз, когда в программе встречается параллельное выполнение кода, необходимо предотвратить *состояние гонки*. Это ситуация, в которой ваша программа ведет себя некорректно из-за того, что инструкции выполняются в незапланированный момент времени или в непредусмотренном порядке.

Чтобы увидеть состояние гонки в действии, представьте систему бронирования билетов, которая должна эффективно проводить регистрацию пассажиров и подсчитывать их количество. Если система обрабатывает 1000 рейсов по 75 пассажиров на каждом, вы решаете применить отдельную сопрограмму для каждого рейса. Реализовать такую схему позволяет код, показанный ниже. Опробуйте его в REPL.

(Вам будет предложено запустить REPL в контексте разных модулей вашего проекта. Выберите `TaernylAir.main` — для других модулей библиотека `Coroutines` окажется недоступной.)

Листинг 20.14. Состояние гонки при регистрации пассажиров (REPL)

```
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking
```

```
val passengersPerFlight = 75
val numberOfFlights = 1000
var checkedInPassengers = 0

runBlocking {
    repeat(numberOfFlights) {
        launch(Dispatchers.Default) {
            checkedInPassengers += passengersPerFlight
        }
    }
}

println(checkedInPassengers)
74325
```

Несколько раз выполните этот фрагмент кода и наблюдайте за выводом. В описанном сценарии регистрацию проходят 75 000 пассажиров. Результат совпадает с вашим выводом?

В зависимости от производительности вашего компьютера вы будете получать разные числа при каждом запуске кода, но они окажутся меньше ожидаемых 75 000. (Если вы постоянно получаете результат 75 000, попробуйте увеличить количество рейсов в системе.) Почему это происходит?

Когда вы используете оператор `+=` в `checkedInPassengers += passengersPerFlight`, программа должна выполнить три действия: сначала она читает значения в `checkedInPassengers` и `passengersPerFlight`, затем суммирует их, после чего записывает полученное значение в область памяти, в которой хранится переменная `checkedInPassengers`.

Все эти действия занимают некоторое время, и при работе с одной переменной может оказаться, что разные потоки будут находиться на разных стадиях операции. В результате они начнут вмешиваться в работу друг друга.

Для предотвращения такой проблемы возможно несколько способов:

- синхронное выполнение работы без использования многопоточного выполнения;
- изменение структуры кода, чтобы потокам не приходилось обращаться к общим изменяемым значениям;
- синхронизация потоков, которая гарантирует, что в любой момент времени только один из них сможет работать с переменной `checkedInPassengers`;
- хранение `checkedInPassengers` в структуре данных, безопасной по отношению к потокам.

У каждого из этих способов есть достоинства и недостатки. В данном примере мы рекомендуем последний вариант: использование потоково-безопасных структур данных.

В Kotlinx есть библиотека AtomicFU. Как и Ktor или сама библиотека Coroutines, AtomicFU является мультиплатформенной и может использоваться с JVM и на других платформах.

Чтобы настроить AtomicFU для работы в вашем проекте, необходимо зарегистрировать плагин в файле `build.gradle`. Внесите следующее изменение.

Листинг 20.15. Добавление плагинов AtomicFU (`build.gradle`)

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlinx:atomicfu-gradle-plugin:0.16.2"
    }
}

plugins {
    id 'org.jetbrains.kotlin.jvm' version '1.5.21'
}

apply plugin: 'kotlinx-atomicfu'
...
```

После загрузки изменений Gradle вам станет доступна функция `atomic`, используемая для объявления атомарных ссылок. Атомарная ссылка — потоково-безопасная структура данных, которая выполняет сложные операции (такие, как серия «чтение—увеличение—запись», упоминавшаяся ранее) в *атомарном* режиме — это означает, что операция выполняется как единое целое.

Атомарные структуры данных отлично подходят для хранения данных в задачах такого рода, потому что они гарантируют, что данные не будут потеряны при попытке изменить одни и те же значения из нескольких потоков. Попробуйте сделать это в REPL. (Постройте программу и перезапустите REPL, чтобы актуализировать изменения.) Единственное изменение по сравнению с кодом проекта — объявление `checkedInPassengers`. Теперь в нем используется ключевое слово `val`, а функция `atomic` упаковывает значение в `AtomicReference`.

Листинг 20.16. Использование функции `atomic` (REPL)

```
import kotlinx.atomicfu.atomic
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

val passengersPerFlight = 75
val numberOfFlights = 1000
val checkedInPassengers = atomic(0)

runBlocking {
    repeat(numberOfFlights) {
```

```
        launch(Dispatchers.Default) {  
            checkedInPassengers += passengersPerFlight  
        }  
    }  
}  
  
println(checkedinPassengers)  
75000
```

Выполните этот код несколько раз. Вы увидите, что теперь все 75 000 пассажиров стабильно проходят регистрацию; это означает, что состояние гонки устранено.

О потоковой безопасности можно рассказать еще многое, но это уже выходит за рамки нашей книги. Атомарные ссылки хорошо работают в описанной ситуации, где несколько потоков должны записывать данные в одно поле, но это не панацея.

Иногда требуется блокировать целые разделы кода, чтобы в опасной секции мог работать только один поток. Практика, когда к некоторой части кода может обращаться только один поток, называется *взаимоисключающим выполнением*. Существуют разные механизмы решения этой задачи, от мьютексов и семафоров до функции `synchronized` и платформенно-зависимых API.

Попробуйте избежать ситуаций гонки, ограничивая размер общего изменяемого состояния, используемого несколькими потоками. Мы также рекомендуем отслеживать возможные состояния гонки в вашем коде и выявлять их в ходе отладки, если окажется, что сложный многопоточный алгоритм ведет себя не так, как ожидалось.

Для любознательных: Kotlin на стороне сервера

В этой главе вы использовали Ktor как клиент HTTP для потребления данных от конечных точек `flight`. Возможности Ktor этим не ограничиваются: Ktor также предоставляет фреймворк веб-сервера для JVM. Собственно, хостинг конечных точек `kotlin-book.bignerdranch.com/2e`, которые использовались в примерах этой главы, обеспечивался средствами Ktor.

Если вас интересует тема запуска собственного сервера на базе Kotlin и Ktor, посетите официальный веб-сайт по адресу `ktor.io`. Для справки ниже мы приводим упрощенную версию функции `main` для сервера, с которым вы взаимодействовали в этой главе.

```
fun main() = embeddedServer(Netty, port = 8080) {  
    routing {  
        get("/") {  
            call.respondRedirect(  
                url = "https://bignerdranch.com/books/"  
            )  
        }  
  
        route("/2e") {
```

```
        get("flight") {
            delay(5000)
            call.respond(
                status = HttpStatusCode.OK,
                message = FlightSchedule.random().toString()
            )
        }

        get("loyalty") {
            delay(2000)
            call.respond(
                status = HttpStatusCode.OK,
                message = LoyaltyStatus.random().toString()
            )
        }
    }
}.start(wait = true)
```

Задание: никаких отмен

Отмена рейса нарушает планы героя на отпуск. В этом задании отклоняйте все ответы со статусом `Canceled`. Выполняйте новые запросы, пока не получите рейс, который либо осуществляется по графику, либо (к разочарованию героя) задерживается.

21. Потоки данных

Вам как разработчику приходится принимать множество решений. Будут ли разные компоненты взаимодействовать напрямую? Должны ли источники данных предоставить возможность другим компонентам подписываться на публикацию изменений? Должны ли данные стабильно двигаться в одном направлении? Какую бы стратегию вы ни выбрали, очень важно, чтобы ваши решения были осознанными.

В этой главе мы рассмотрим встроенную поддержку потоков данных в Kotlin. Вы узнаете, как пересылать данные внутри приложения. *Потоки данных* (flows) поддерживают возможность подписки и помогают создавать приложения с однонаправленной передачей информации, которые обновляются в зависимости от изменения состояния.

Часто такие решения строятся на базе объектно-ориентированных парадигм, применявшихся в `NyetHack`, и это способствует проектированию классов, в большей степени изолированных друг от друга. Если часть вашей программы должна знать об изменениях, происходящих в другой точке кода, она может зарегистрировать себя для получения новых значений состояния в момент их появления, а не запрашивать обновленные значения, когда она посчитает, что значение могло измениться.

Потоки данных также реализуют многие операции функционального программирования, о которых мы рассказали в главе 11. В результате в вашем распоряжении оказывается мощный набор инструментов для работы с данными.

Мы будем обновлять приложение `TaernylAir` для мониторинга нескольких рейсов. Для каждого рейса будет выводиться информация о том, когда открывается посадка, когда начинается посадка для группы лояльности, к которой принадлежит пассажир, когда посадка завершается и когда самолет взлетает.

В основе модели лежат события, происходящие в ответ на другие события. К концу этой главы приложение будет содержать два блока информации: какой рейс отслеживается и сведения о нем.

Для решения задачи можно воспользоваться императивным подходом с приемами, которые мы уже применяли в книге. Однако императивное решение труднее разделять на части, чтобы в будущем добавлять функциональность. Чтобы повысить модульность приложения, мы напишем реактивный код с использованием потоков данных.

Создание потоков данных

Потоки используются в Kotlin для представления асинхронной передачи данных. На верхнем уровне потоки данных поддерживают два действия: генерирование и потребление. При потреблении потока данных вы регистрируете слушатель, который будет получать все данные, генерируемые потоком. Поток может сгенерировать за свой период жизни любое количество элементов данных, а может не генерировать их вообще.

Конкретные подробности генерирования и потребления зависят от того, как был создан поток данных. Иногда с потоком данных можно оперировать напрямую и приказывать ему сгенерировать конкретное значение. В других случаях функциональность генерирования может оказаться недоступной и данные из потока можно будет только потреблять.

В этой главе мы покажем примеры двух разновидностей потоков данных с разным поведением.

Потоки данных встроены в библиотеку Coroutines, уже добавленную в проект TaarnylAir, так что они готовы к использованию. Чтобы создать первый поток данных, создайте новый файл с именем `FlightWatcher.kt` и добавьте в него три новые функции: функцию `main` для организации задач отслеживания рейсов, вторую функцию для отслеживания статуса рейса и вывода обновлений статуса и третью — для получения всех отслеживаемых рейсов.

(При вводе этого кода необходимо импортировать класс `kotlinx.coroutines.flow.Flow` и функцию `kotlinx.coroutines.flow.flow()`.)

Листинг 21.1. Использование функции `flow` (`FlightWatcher.kt`)

```
fun main() {
    runBlocking {
        println("Getting the latest flight info...")
        val flights = fetchFlights()
        val flightDescriptions = flights.joinToString {
            "${it.passengerName} (${it.flightNumber})"
        }
        println("Found flights for $flightDescriptions")
        flights.forEach {
            watchFlight(it)
        }
    }
}

fun watchFlight(initialFlight: FlightStatus) {
    val currentFlight: Flow<FlightStatus> = flow {
        var flight = initialFlight
        repeat(5) {
```

```
        emit(flight)
        delay(1000)
        flight = flight.copy(
            departureTimeInMinutes = flight.departureTimeInMinutes - 1
        )
    }
}

suspend fun fetchFlights(
    passengerNames: List<String> = listOf("Madrigal", "Polarcubis")
) = passengerNames.map { fetchFlight(it) }
```

Первая половина новой функции `main` похожа на ту, что мы использовали в главе 20; функция `fetchFlight` вызывается для каждого пассажира с использованием строителя сопрограммы `runBlocking`.

Функция `watchFlight` выглядит интереснее. Мы определяем поток данных, который генерирует новые данные рейса. Каждую секунду значение `departureTimeInMinutes` для рейса уменьшается, так как до вылета остается все меньше времени.

Вспомните функцию построения последовательностей `generateSequence` из главы 11. Когда с последовательностью, создаваемой с исходным значением, выполняет операцию другая функция, `generateSequence` вызывает итератор для определения следующего производимого значения. Как и `generateSequence`, `flow` в отложенном режиме генерирует в некотором порядке элементы, которые потребляются другим компонентом.

Обратите внимание на вызов `emit` в `watchFlight`. Он показывает, как задаются элементы, которые должны отправляться потребителю потока данных. Эта функция доступна только внутри лямбда-выражения, которое передается `flow`.

Функция `emit` является приостанавливаемой, как и функция `delay`, которую мы использовали выше. Но мы не добавили модификатор `suspend` в сигнатуру функции `watchFlight`, что дает нам подсказку о том, что здесь происходит: во внутренней реализации `flow` создает область видимости сопрограммы, в которой выполняется содержимое лямбда-выражения. Это и позволяет нам вызывать приостанавливаемые функции внутри лямбда-выражения. Область видимости сопрограммы, используемая потоком данных, создается в начале потребления потока данных и закрывается, когда поток данных перестает генерировать элементы или эти элементы перестают потребляться.

В программе присутствуют две функции `main`, `FlightWatcher.kt` и `FlightFetcher.kt`. У компилятора с этим проблем не возникает, и программа прекрасно работает. Но для предотвращения неоднозначности можно удалить функцию `main` в `FlightFetcher.kt`, которая стала ненужной.

Листинг 21.2. Удаление старой функции main (FlightFetcher.kt)

```
private const val BASE_URL = "http://kotlin-book.bignerdranch.com/2e"
private const val FLIGHT_ENDPOINT = "$BASE_URL/flight"
private const val LOYALTY_ENDPOINT = "$BASE_URL/loyalty"

fun main() {
    runBlocking {
        println("Started")
        launch {
            val flight = fetchFlight("Madrigal")
            println(flight)
        }
        println("Finished")
    }
}
...
```

Если сейчас запустить TaernylAir, код внутри лямбда-выражения потока данных выполняться не будет. Поток данных *пассивен*, то есть он не генерирует данные. Он не будет вычислять тело лямбда-выражения, пока вы не начнете потреблять данные от него. (*Активный* поток данных выполняет код и генерирует значения, даже если у него нет потребителей.)

Для реализации возможностей потока данных используется функция `collect`. Пока мы ограничимся моделированием первых пяти минут посадки на борт после получения информации о рейсе. Позднее в этой главе мы обновим логику потока данных, чтобы вы имели возможность отслеживать рейс до момента вылета.

Листинг 21.3. Потребление данных рейса (FlightWatcher.kt)

```
...
suspend fun watchFlight(initialFlight: FlightStatus) {
    val passengerName = initialFlight.passengerName
    val currentFlight: Flow<FlightStatus> = flow {
        var flight = initialFlight
        repeat(5) {
            emit(flight)
            delay(1000)
            flight = flight.copy(
                departureTimeInMinutes = flight.departureTimeInMinutes - 1
            )
        }
    }

    currentFlight
        .collect {
            println("$passengerName: $it")
        }
}
...
```

В отличие от `flow`, функция `collect` является приостанавливаемой. Чтобы иметь возможность вызвать `collect`, необходимо добавить в `watchFlight` модификатор `suspend`. Сама функция `collect` приостанавливается до того момента, когда `flow` перестанет генерировать элементы. Чтобы продемонстрировать эту возможность, добавьте отладочный вывод после вызова `collect`.

Листинг 21.4. Ожидание завершения `flow` (`FlightWatcher.kt`)

```
...
suspend fun watchFlight(initialFlight: FlightStatus) {
    ...
    currentFlight
        .collect {
            println("$passengerName: $it")
        }

    println("Finished tracking $passengerName's flight")
}
```

Запустите функцию `main` из `FlightWatcher.kt`. Вывод должен выглядеть примерно так:

```
Getting the latest flight info...
...
Finished fetching flight info
Found flights for Madrigal (RD0475), Polarcubis (WG2393)
Madrigal: FlightStatus(flightNumber=RD0475, ..., departureTimeInMinutes=110)
Madrigal: FlightStatus(flightNumber=RD0475, ..., departureTimeInMinutes=109)
Madrigal: FlightStatus(flightNumber=RD0475, ..., departureTimeInMinutes=108)
Madrigal: FlightStatus(flightNumber=RD0475, ..., departureTimeInMinutes=107)
Madrigal: FlightStatus(flightNumber=RD0475, ..., departureTimeInMinutes=106)
Finished tracking Madrigal's flight
Polarcubis: FlightStatus(flightNumber=WG2393, ..., departureTimeInMinutes=30)
Polarcubis: FlightStatus(flightNumber=WG2393, ..., departureTimeInMinutes=29)
Polarcubis: FlightStatus(flightNumber=WG2393, ..., departureTimeInMinutes=28)
Polarcubis: FlightStatus(flightNumber=WG2393, ..., departureTimeInMinutes=27)
Polarcubis: FlightStatus(flightNumber=WG2393, ..., departureTimeInMinutes=26)
Finished tracking Polarcubis's flight
```

Пока неплохо. Программа выводит обновления за первые пять минут, но вывод оставляет желать лучшего. Пассажира не получают информации о том, может ли он подняться на борт самолета, а вывод плохо читается. Прежде чем имитировать процесс посадки далее, попробуем улучшить вывод.

Чтобы реализовать логику, определяющую время посадки, создайте два новых класса перечислений в существующем файле `FlightStatus.kt`.

Листинг 21.5. Добавление информации о лояльности и времени посадки (FlightStatus.kt)

```
data class FlightStatus(
    ...
) {
    ...
}

enum class LoyaltyTier(
    val tierName: String,
    val boardingWindowStart: Int
) {
    Bronze("Bronze", 25),
    Silver("Silver", 25),
    Gold("Gold", 30),
    Platinum("Platinum", 35),
    Titanium("Titanium", 40),
    Diamond("Diamond", 45),
    DiamondPlus("Diamond+", 50),
    DiamondPlusPlus("Diamond++", 60)
}

enum class BoardingState {
    FlightCanceled,
    BoardingNotStarted,
    WaitingToBoard,
    Boarding,
    BoardingEnded
}
```

Перечисление `LoyaltyTier` определяет статус клиентов компании по программе лояльности. С каждым уровнем лояльности надо связать время, когда пассажиры могут взойти на борт. Информация — интервалы минут до вылета — хранится в `boardingWindowStart`. Например, пассажиры с уровнем лояльности `Gold` могут садиться в самолет за 30 минут до вылета.

Чтобы использовать первое из новых перечислений, обновите `FlightStatus` — значение `passengerLoyaltyTier` будет храниться в типе `LoyaltyTier` вместо `String`.

Листинг 21.6. Парсинг `LoyaltyTier` (FlightStatus.kt)

```
data class FlightStatus(
    val flightNumber: String,
    val passengerName: String,
    val passengerLoyaltyTier: String LoyaltyTier,
    val originAirport: String,
    val destinationAirport: String,
    val status: String,
    val departureTimeInMinutes: Int
) {
```

```

companion object {
    fun parse(
        flightResponse: String,
        loyaltyResponse: String,
        passengerName: String
    ): FlightStatus {
        val (flightNumber, originAirport, destinationAirport, status,
            departureTimeInMinutes) = flightResponse.split(",")
        val (loyaltyTierName, milesFlown, milesToNextTier) =
            loyaltyResponse.split(",")
        return FlightStatus(
            flightNumber = flightNumber,
            passengerName = passengerName,
            passengerLoyaltyTier = loyaltyTierName,
            passengerLoyaltyTier = LoyaltyTier.values()
                .first { it.tierName == loyaltyTierName },
            originAirport = originAirport,
            destinationAirport = destinationAirport,
            status = status,
            departureTimeInMinutes = departureTimeInMinutes.toInt()
        )
    }
}
}
...

```

Теперь у нас есть вся информация для определения статуса пассажира в любой заданный момент времени. Определите в `FlightStatus` новое свойство `boardingStatus` для вычисления значения `BoardingState` пассажира. Необходимо определить четыре дополнительных свойства: `isFlightCanceled`, `hasBoardingStarted`, `isBoardingOver` и `isEligibleToBoard`. Помните, что посадка начинается за 60 минут до вылета — на борт поднимаются пассажиры с уровнем `Diamond++` — и завершается за 15 минут до вылета.

Листинг 21.7. Вычисление `BoardingState` (`FlightStatus.kt`)

```

data class FlightStatus(
    ...
) {

    val isFlightCanceled: Boolean
        get() = status.equals("Canceled", ignoreCase = true)

    val hasBoardingStarted: Boolean
        get() = departureTimeInMinutes in 15..60

    val isBoardingOver: Boolean
        get() = departureTimeInMinutes < 15

    val isEligibleToBoard: Boolean

```

```

        get() = departureTimeInMinutes in 15..passengerLoyaltyTier.
            boardingWindowStart

    val boardingStatus: BoardingState
    get() = when {
        isFlightCanceled -> BoardingState.FlightCanceled
        isBoardingOver -> BoardingState.BoardingEnded
        isEligibleToBoard -> BoardingState.Boarding
        hasBoardingStarted -> BoardingState.WaitingToBoard
        else -> BoardingState.BoardingNotStarted
    }
    ...
}
...

```

После определения новых свойств для `FlightStatus` можно вернуться к функции `watchFlight`. Замените `repeat` циклом `while`, чтобы данные обновлялись вплоть до вылета. Также измените формат вывода, чтобы он был более удобным для пассажиров.

Листинг 21.8. Улучшение вывода для отслеживания рейсов (FlightWatcher.kt)

```

...
suspend fun watchFlight(initialFlight: FlightStatus) {
    val passengerName = initialFlight.passengerName
    val currentFlight: Flow<FlightStatus> = flow {
        var flight = initialFlight
        repeat(5){
            while (flight.departureTimeInMinutes >= 0 && !flight.isFlightCanceled) {
                emit(flight)
                delay(1000)
                flight = flight.copy(
                    departureTimeInMinutes = flight.departureTimeInMinutes - 1
                )
            }
        }
    }
    currentFlight
        .collect {
            val status = when (it.boardingStatus) {
                FlightCanceled -> "Your flight was canceled"
                BoardingNotStarted -> "Boarding will start soon"
                WaitingToBoard -> "Other passengers are boarding"
                Boarding -> "You can now board the plane"
                BoardingEnded -> "The boarding doors have closed"
            } + " (Flight departs in ${it.departureTimeInMinutes} minutes)"
            println("$passengerName: $it $status")
        }
    println("Finished tracking $passengerName's flight")
}
...

```


(Чтобы использовать значения `BoardingState` без префикса `BoardingState.`, добавьте команду `import BoardingState.*` в начало файла `FlightWatcher.kt`.)

Запустите `TaernylAir`. Вывод должен выглядеть примерно так:

```
...
Found flights for Madrigal (0A9084), Polarcubis (YJ8056)
Madrigal: Other passengers are boarding (Flight departs in 34 minutes)
Madrigal: Other passengers are boarding (Flight departs in 33 minutes)
Madrigal: Other passengers are boarding (Flight departs in 32 minutes)
Madrigal: Other passengers are boarding (Flight departs in 31 minutes)
Madrigal: You can now board the plane (Flight departs in 30 minutes)
Madrigal: You can now board the plane (Flight departs in 29 minutes)
Madrigal: You can now board the plane (Flight departs in 28 minutes)
Madrigal: You can now board the plane (Flight departs in 27 minutes)
...
Madrigal: The boarding doors have closed (Flight departs in 0 minutes)
Finished tracking Madrigal's flight
...
```

То, что мы сделали, существенно упрощает жизнь пассажиров. Теперь они видят состояние процесса посадки и получают обновления вплоть до вылета рейса. На следующей стадии мы улучшим поведение `TaernylAir` для отслеживания нескольких рейсов (при этом в любой момент времени мониторится только один рейс). Чтобы пользователь лучше представлял общую картину, приложение будет записывать логи, показывающие, сколько рейсов осталось для отслеживания.

MutableStateFlow

В зависимости от работы, которую выполняет приложение, может оказаться, что возможности строителя `flow` сильно ограничены. Только переданное вами лямбда-выражение может генерировать элементы в поток данных. Такое решение хорошо подходит для обновления времени вылета рейса, так как в этом случае данные с течением времени обновляются автоматически. Однако на состояние других частей приложения должны влиять сразу несколько компонентов.

Чтобы пользователям было проще отслеживать несколько рейсов, приложение `TaernylAir` должно сообщать, сколько рейсов находится в очереди на отслеживание. Состояние, которое следует отслеживать в данном случае, — это количество рейсов на посадку. Однако строитель `flow` плохо подходит для этой задачи, поскольку подсчитывать это количество в одном лямбда-выражении крайне неудобно.

Вместо написания сложной логики просто сохраним счетчик рейсов в `MutableStateFlow`.

`MutableStateFlow` — реализация `Flow`, которая приносит огромную пользу при отслеживании состояния приложения. Она позволяет создать поток данных, значение которому можно переписать вручную после его создания. Начнем с объявления `MutableStateFlow` для количества рейсов.

Листинг 21.9. Объявление `MutableStateFlow` (`FlightWatcher.kt`)

```
fun main() {
    runBlocking {
        ...
        println("Found flights for $flightDescriptions")
        val flightsAtGate = MutableStateFlow(flights.size)

        flights.forEach {
            watchFlight(it)
        }
    }
}
...

```

Обратите внимание на различия между потоками данных `flightsAtGate` и `currentFlight`. Используя `flow` с `currentFlight`, вы определяли каждое значение, которое поток данных будет генерировать на протяжении своего срока жизни. С другой стороны, конструктор `MutableStateFlow` получает исходное значение. Если немедленно начать потреблять информацию из этого потока данных, вы сразу получите значение `flights.size`. (В текущем коде это значение всегда равно 2.)

Кроме того, `MutableStateFlow` является активным потоком данных, тогда как поток данных `currentFlight` — пассивный. `MutableStateFlow` всегда «живой», но если генерировать в поток, не имеющий потребителей, значения не будут использованы до вызова `collect`.

Чтобы извлечь максимум пользы из `MutableStateFlow` (особенно из изменяемости), вы можете обновить значение, хранимое внутри потока. При этом достигаются две цели. Поток данных генерирует новое значение для всех активных потребителей. Это означает, что каждый компонент, зарегистрированный для наблюдения за значением, будет немедленно оповещен о новом значении. Кроме того, поток данных запомнит последнее значение. Если новый компонент начнет потреблять данные из потока, он немедленно получит самое свежее значение, записанное в поток.

Обновите функцию `main`, чтобы значение счетчика `flightsAtGate` уменьшалось по завершении мониторинга данных по рейсу.

Листинг 21.10. Запись в `MutableStateFlow` (`FlightWatcher.kt`)

```
fun main() {
    runBlocking {
        ...
        val flightsAtGate = MutableStateFlow(flights.size)

        flights.forEach {
            watchFlight(it)
        }
    }
}

```

```

        flightsAtGate.value = flightsAtGate.value - 1
    }
}
...

```

В отличие от функции `emit`, которую мы представляли ранее, метод записи свойства `value` экземпляра `MutableStateFlow` не является приостанавливаемой функцией. Любая часть вашего кода, который может обратиться к потоку данных `flightsAtGate`, имеет полномочия для отправки информации этому потоку данных, даже если код, который отправляет новое значение, не выполняется в сопрограме. Этот нюанс важен, когда вы определяете свойство уровня файла или уровня класса, содержащее `MutableStateFlow`.

(Также для уменьшения значения можно использовать синтаксис `flightsAtGate.value--`, но мы приводим здесь полный синтаксис, дабы продемонстрировать, что значение можно читать и записывать без приостановки.)

С большой силой приходит большая ответственность. Возможно, стоит ограничить круг тех, кому разрешено задавать количество рейсов. Для этого можно воспользоваться интерфейсом `StateFlow`.

По аналогии с отношениями `List/MutableList` интерфейс `StateFlow` представляет собой аналог `MutableStateFlow`, доступный только для чтения. Это еще один пример особого отношения к неизменяемости в Kotlin.

Один из распространенных паттернов при использовании `MutableStateFlow` — предоставление доступа к версии, доступной только для чтения, в публичном свойстве, в то время как изменяемая версия остается приватной. Следующий пример демонстрирует реализацию этого паттерна с двумя свойствами.

```

private val _boardingPass: MutableStateFlow<BoardingPass> =
    mutableStateFlow(BoardingPass())
val boardingPass: StateFlow<BoardingPass>
    get() = _boardingPass

fun refreshBoardingPass() {
    _boardingPass.value = BoardingPass()
}

```

(Если у вас есть как приватная, так и публичная версия свойства, использующая разные типы, принято снабжать приватное свойство префиксом `_`.)

С потоком данных `flightsAtGate` так поступать не нужно, потому что его область видимости уже ограничена функцией. Но для другой функциональности, такой как вывод посадочных талонов с номерами рейсов, которые могут неожиданно измениться, стоит объявить `MutableStateFlow` за пределами функции и заблокировать его изменяемость.

Чтобы увидеть, как отслеживается количество рейсов, необходимо потреблять поток данных `flightsAtGate`. Добавьте еще один вызов `collect` перед тем, как пас-

сажиры начнут занимать места. (Вероятно, вы заметили, что в этом примере есть одна проблема. Не беспокойтесь, вскоре мы ее исправим.)

Листинг 21.11. Добавление второго вызова collect (FlightWatcher.kt)

```
fun main() {
    runBlocking {
        println("Getting the latest flight info...")
        val flights = fetchFlights()
        val flightDescriptions = flights.joinToString {
            "${it.passengerName} (${it.flightNumber})"
        }
        println("Found flights for $flightDescriptions")

        val flightsAtGate = MutableStateFlow(flights.size)
        flightsAtGate
            .collect { flightCount ->
                println("There are $flightCount flights being tracked")
            }
        println("Finished tracking all flights")

        flights.forEach {
            watchFlight(it)
            flightsAtGate.value = flightsAtGate.value - 1
        }
    }
}
...

```

Снова запустите приложение. Вы увидите, что после вывода сообщения `There are 2 flights being tracked` ничего не происходит. В чем дело?

Вспомните, что `collect` является приостанавливаемой функцией — она приостанавливается до завершения потока данных. Это и создает проблему, потому что потоку данных `flightsAtGate` не было приказано завершиться. Более того, класс `MutableStateFlow` завершаться не умеет.

В данном случае необходимо параллельное потребление из двух потоков данных. Для этого переместите вызов `collect` в собственные сопрограммы. Используя функцию `launch`, снова обновите `main`, чтобы предотвратить проблему взаимной блокировки.

Листинг 21.12. Параллельное отслеживание потоков данных (FlightWatcher.kt)

```
fun main() {
    runBlocking {
        ...
        val flightsAtGate = MutableStateFlow(flights.size)
        launch {
            flightsAtGate
                .collect { flightCount ->

```

```

        println("There are $flightCount flights being tracked")
    }
    println("Finished tracking all flights")
}

launch {
    flights.forEach {
        watchFlight(it)
        flightsAtGate.value = flightsAtGate.value - 1
    }
}
}
...

```

Снова запустите `FlightWatcher.kt`. На этот раз вывод будет включать количество отслеживаемых рейсов:

```

...
Found flights for Madrigal (ER9618), Polarcubis (M07737)
There are 2 flights being tracked
...
Madrigal: The boarding doors have closed (Flight departs in 3 minutes)
Madrigal: The boarding doors have closed (Flight departs in 2 minutes)
Madrigal: The boarding doors have closed (Flight departs in 1 minutes)
Madrigal: The boarding doors have closed (Flight departs in 0 minutes)
There are 1 flights being tracked
...
Polarcubis: The boarding doors have closed (Flight departs in 3 minutes)
Polarcubis: The boarding doors have closed (Flight departs in 2 minutes)
Polarcubis: The boarding doors have closed (Flight departs in 1 minutes)
Polarcubis: The boarding doors have closed (Flight departs in 0 minutes)
Finished tracking all flights

```

Отслеживание рейсов теперь работает так, как ожидалось, но в `TaernylAir` осталась одна проблема. После вывода завершающего сообщения `Finished tracking all flights` программа не завершается. На следующей стадии мы обновим код, чтобы приложение `TaernylAir` завершалось после вылета последнего рейса.

Завершение потоков данных

Итак, мы продемонстрировали примеры потоков данных, которые завершаются автоматически, и потоков, неспособных к завершению. Вы получили в распоряжение некоторые инструменты, способные влиять на способ завершения потока.

Когда сопрограмма отменяется перед завершением потока, потребитель перестает получать генерируемые данные, а поток переходит в спящее состояние, если у него нет других потребителей. Для изменения этого поведения можно воспользоваться *оператором*. Операторы влияют на то, как поток данных генерирует

элементы. (Хотя термины звучат одинаково, эти операторы отличаются от уже известных вам операторов `+`, `-` и т. д.)

В зависимости от того, какой оператор вы используете, можно пропускать, добавлять или изменять элементы в потоке данных или — для ваших конкретных целей — влиять на способ завершения потока. Ниже перечислены операторы, влияющие на завершение потока; некоторые из них могут быть вам знакомы по API для работы с коллекциями и последовательностями в Kotlin. Эти операторы определяются в файле с именем `Limit.kt` библиотеки `Coroutines`.

take	Получает параметр <code>Int</code> и потребляет элементы до количества, заданного в этом параметре, после чего завершается.
takeWhile	Потребляет сгенерированные элементы до тех пор, пока не будет сгенерирован элемент, не удовлетворяющий заданному предикату, после чего завершается.
drop	Получает параметр <code>Int</code> и игнорирует сгенерированные элементы из потока данных до количества, заданного в параметре, после чего передает другие значения для потребления.
dropWhile	Игнорирует сгенерированные элементы до тех пор, пока не будет сгенерирован элемент, не удовлетворяющий заданному предикату, после чего передает другие значения для потребления.

Эти функции также доступны для типов коллекций и последовательностей, о которых мы рассказывали в главе 11. Более того, многие операторы потоков данных также существуют как операции функционального программирования для типов коллекций и последовательностей Kotlin. Часто используемые операторы описаны в документации `Flow`, доступной по адресу kotlinlang.org/docs/flow.html.

Чтобы приложение `TaernylAir` могло завершиться, необходимо обновить `flightsAtGate`, чтобы данные переставали потребляться при достижении счетчиком нуля. Для этого воспользуемся оператором `takeWhile`.

Листинг 21.13. Отмена бесконечного потока данных (`FlightWatcher.kt`)

```
fun main() {
    runBlocking {
        ...
        val flightsAtGate = MutableStateFlow(flights.size)
        launch {
            flightsAtGate
                .takeWhile { it > 0 }
                .collect { flightCount ->
                    println("There are $flightCount flights being tracked")
                }
            println("Finished tracking all flights")
        }
    }
}
```

```

    ...
}
}
...

```

Запустите `FlightWatcher.kt` и немного подождите. Вы увидите сообщение `Process finished with exit code 0`, которое означает, что потоки данных завершаются правильно.

`Flow` также содержит функцию `onCompletion`, которая позволяет задать действие, выполняемое при завершении потока. Измените код потоков данных, переместив два вызова `println` в вызов `onCompletion`.

Листинг 21.14. Использование `onCompletion` (`FlightWatcher.kt`)

```

fun main() {
    runBlocking {
        ...
        val flightsAtGate = MutableStateFlow(flights.size)
        launch {
            flightsAtGate
                .takeWhile { it > 0 }
                .onCompletion {
                    println("Finished tracking all flights")
                }
                .collect { flightCount ->
                    println("There are $flightCount flights being tracked")
                }
            println("Finished tracking all flights")
        }
        ...
    }
}

suspend fun watchFlight(initialFlight: FlightStatus) {
    ...
    currentFlight
        .onCompletion {
            println("Finished tracking $passengerName's flight")
        }
        .collect {
            val status = when (it.boardingStatus) {
                FlightCanceled -> "Your flight was canceled"
                BoardingNotStarted -> "Boarding will start soon"
                WaitingToBoard -> "Other passengers are boarding"
                Boarding -> "You can now board the plane"
                BoardingEnded -> "The boarding doors have closed"
            } + " (Flight departs in ${it.departureTimeInMinutes} minutes)"
            println("$passengerName: $status")
        }
        println("Finished tracking $passengerName's flight")
    }
}
...

```

Так как функция `collect` уже приостанавливается до завершения, это изменение не повлияет на поведение кода. (При желании вы можете запустить программу снова, чтобы убедиться в этом.) Это всего лишь удобный способ упростить логику и связать побочные эффекты с самим потоком данных.

Преобразования потоков данных

В главе 11 вы узнали о функции `map`, которая преобразует каждый элемент коллекции на основании заданного преобразующего лямбда-выражения. Эта функция также является оператором, который может использоваться с `Flow`. Попробуйте ее на `currentFlight`, чтобы переместить часть логики за пределы лямбда-выражения `collect`.

Листинг 21.15. Преобразование значений в потоке данных (FlightWatcher.kt)

```
...
suspend fun watchFlight(initialFlight: FlightStatus) {
    ...
    currentFlight
        .map { flight ->
            when (flight.boardingStatus) {
                FlightCanceled -> "Your flight was canceled"
                BoardingNotStarted -> "Boarding will start soon"
                WaitingToBoard -> "Other passengers are boarding"
                Boarding -> "You can now board the plane"
                BoardingEnded -> "The boarding doors have closed"
            } + " (Flight departs in ${flight.departureTimeInMinutes} minutes)"
        }
        .onCompletion {
            println("Finished tracking $passengerName's flight")
        }
        .collect { status ->
            val status = when (it.boardingStatus) {
                FlightCanceled -> "Your flight was canceled"
                BoardingNotStarted -> "Boarding will start soon"
                WaitingToBoard -> "Other passengers are boarding"
                Boarding -> "You can now board the plane"
                BoardingEnded -> "The boarding doors have closed"
            } + " (Flight departs in ${it.departureTimeInMinutes} minutes)"
            println("$passengerName: $status")
        }
    }
    ...
}
```

Функция `map` ведет себя так же, как в коде главы 11. Каждое значение, которое генерируется этим потоком данных, будет преобразовано перед потреблением. Потребитель получает из потока только преобразованные значения.

Многие преобразующие функции, известные вам по стандартной библиотеке Kotlin (особенно с типами коллекций), также встречаются при работе с Flow. Например, доступны функции `flatMap`, `filter` и `zip`. (Кстати говоря, многие разработчики используют термин «оператор» и «преобразующая функция» как синонимы, потому что у них много общего. Однако во внутренней реализации Flow использует собственные операторы, которые строятся по образцу преобразующих функций коллекций.)

Чтобы определить нестандартное преобразование для вашего потока данных, вы можете воспользоваться функцией преобразования. Например, поведение оператора `map` можно имитировать в коде (преобразующем температуры по шкале Цельсия в температуры по шкале Кельвина) следующим образом:

```
suspend fun observeTemperature(
    val temperatureInCelsius: Flow<Int>
) {
    temperatureInCelsius.convertToKelvin()
        .collect { temperatureInKelvin ->
            println("The current temperature is $temperatureInKelvin"
        )
    }

    fun Flow<Int>.convertToKelvin(): Flow<Double> =
        transform<Int, Double> { temperatureInCelsius ->
            emit(temperatureInCelsius + 273.15)
        }
}
```

Этот код вызывает `emit` внутри лямбда-аргумента `transform`. По аналогии с применением строителя `flow`, здесь вы генерируете элементы для любого компонента, потребляющего преобразованное значение. Если вы захотите убрать температуры, лежащие ниже абсолютного нуля, обновите преобразование:

```
fun Flow<Int>.convertToKelvin(): Flow<Double> =
    transform<Int, Double> { temperatureInCelsius ->
        val temperatureInKelvin = temperatureInCelsius + 273.15
        if (temperatureInKelvin >= 0) {
            emit(temperatureInKelvin)
        }
    }
}
```

Это преобразование эквивалентно отображению с последующей фильтрацией. Оно преобразует значение к шкале Кельвина, но выдает только неотрицательные температуры.

Почти все преобразования, которые вам потребуется выполнять, либо встроены в библиотеку Coroutines (где также публикуется Flow), либо могут быть реализованы комбинацией операторов. Мы рекомендуем поглубже изучить стандартную библиотеку, чтобы узнать о других доступных операторах. Но если их окажется недостаточно, вы всегда можете создать собственные.

Обработка ошибок в потоках данных

Теперь вы знаете, как потоки данных могут завершаться корректно, но они также могут завершаться и аномально с выдачей исключения. Когда это происходит, исключение будет передаваться, пока не достигнет вашей функции `collect`. По умолчанию `collect` перезапускает все неперехваченные исключения в потоке данных. Если вы не организуете обработку ошибок на месте, это может привести к аварийному завершению приложения.

Чтобы проверить сказанное, добавим предусловие, которое проверяет, что ни один из отслеживаемых пассажиров не внесен в черный список авиакомпании. Обновите функцию `main`, чтобы увидеть, что происходит при попытке отследить рейс для злодея `Nogartse`, которому запретили летать самолетами после предпринятой им попытки уничтожить мир.

Листинг 21.16. Выдача исключения в потоке данных (FlightWatcher.kt)

```
val bannedPassengers = setOf("Nogartse")

fun main() {
    runBlocking {
        println("Getting the latest flight info...")
        val flights = fetchFlights(listOf("Nogartse"))
        val flightDescriptions = flights.joinToString {
            "${it.passengerName} (${it.flightNumber})"
        }
        println("Found flights for $flightDescriptions")
        ...
    }
}

suspend fun watchFlight(initialFlight: FlightStatus) {
    val passengerName = initialFlight.passengerName

    val currentFlight: Flow<FlightStatus> = flow {
        require(passengerName !in bannedPassengers) {
            "Cannot track $passengerName's flight. They are banned from the
airport."
        }

        var flight = initialFlight
        while (flight.departureTimeInMinutes >= 0 && !flight.isFlightCanceled) {
            emit(flight)
            delay(1000)
            flight = flight.copy(
                departureTimeInMinutes = flight.departureTimeInMinutes - 1
            )
        }
    }
    ...
}
```

Снова запустите свое приложение. Вы увидите, что оно завершается аварийно с выдачей трассировки стека:

```
...
Exception in thread "main" java.lang.IllegalArgumentException: Cannot track
Nogartse's flight. They are banned from the airport.
    at FlightWatcherKt$watchFlight$currentFlight$1.invokeSuspend
      (FlightWatcher.kt:41)
    at FlightWatcherKt$watchFlight$currentFlight$1.invoke(FlightWatcher.kt)
    at FlightWatcherKt$watchFlight$currentFlight$1.invoke(FlightWatcher.kt)
    at kotlinx.coroutines.flow.SafeFlow.collectSafely(Builders.kt:61)
    at kotlinx.coroutines.flow.AbstractFlow.collect(Flow.kt:212)
...
```

Если вы хотите обрабатывать исключения, к вашим услугам несколько инструментов. Первый вариант — упаковка функции `collect` в блок `try/catch`:

```
try {
    currentFlight.collect { println("Got flight data: $it") }
} catch (e: IllegalArgumentException) {
    // Логика восстановления после ошибки
}
```

`Flow` также содержит оператор `catch`, который может перехватывать ошибки в потоке с потенциальной возможностью восстановления работоспособности приложения. Оператор `catch` перехватывает все типы исключений, появляющиеся в потоке. Он получает в аргументе лямбда-выражение, которое определяет, что поток данных должен делать после перехвата исключения. Лямбда-выражение ведет себя так же, как то, что работало с `transform`, и для него доступны те же функции `emit`.

Поток данных завершится после того, как лямбда-выражение вернет управление, а блок `catch` перестанет генерировать элементы. Если вы выберете вариант с включением оператора `catch`, код будет выглядеть примерно так:

```
currentFlight
    .catch { throwable ->
        throwable.printStackTrace()
        emit(/* Резервное значение */)
    }
    .collect { println("Got flight data: $it") }
```

Для исключения, с которым мы работаем, завершение с ошибкой для пассажиров из черного списка, вероятно, допустимо, потому что такие клиенты обслуживаться не должны. Обновите функцию `main`, чтобы она снова отслеживала рейсы для героя.

Листинг 21.17. Предотвращение аварийного завершения (FlightWatcher.kt)

```
...
fun main() {
    runBlocking {
        println("Getting the latest flight info...")
    }
}
```

```

        val flights = fetchFlights(listOf("Nogartse"))
        val flightDescriptions = flights.joinToString {
            "${it.passengerName} (${it.flightNumber})"
        }
        println("Found flights for $flightDescriptions")
        ...
    }
}
...

```

Запустите `FlightWatcher.kt` и убедитесь, что приложение выполняется без ошибок.

Независимо от того, внедряете вы функциональное программирование или ищете решение для более последовательного обмена данными между классами в объектно-ориентированном приложении, вы увидите, что классы `Flow` очень полезны. Богатый набор предоставляемых ими операторов дополняет средства из стандартной библиотеки Kotlin. Мы рекомендуем поглубже изучить эти операторы, чтобы понять, чем они могут быть вам полезны.

Для любознательных: `SharedFlow`

В этой главе мы описали два способа создания потоков данных: с использованием строителя `flow` и типа `MutableStateFlow`. Оба хорошо подходят для подавляющего большинства сценариев, но для особых случаев существует другой класс — `MutableSharedFlow` (и его аналог, доступный только для чтения, — `SharedFlow`).

`MutableSharedFlow` — более общая версия `MutableStateFlow`. Собственно, `MutableStateFlow` наследует от `MutableSharedFlow`. Для чего же он нужен?

Допустим, вы хотите потреблять данные из потока данных с двумя потребителями. Первый потребитель добавляется сразу же после создания потока, но второй появится только по прошествии некоторого времени. Если вы использовали функцию `flow` для построения потока, то заметите, что оба потребителя получили разные значения, а их данные рассинхронизированы.

Опробуйте следующий пример в REPL.

Листинг 21.18. Попытка совместного использования со строителем `flow` (REPL)

```

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

runBlocking {
    val numbersFlow = flow {
        (1..5).forEach {
            delay(1000)
            emit(it)
        }
    }
}

```

```

launch {
    numbersFlow.collect { println("Collector 1: Got $it"\n) }
}

launch {
    delay(2200)
    numbersFlow.collect { println("Collector 2: Got $it"\n) }
}
}
Collector 1: Got 1
Collector 1: Got 2
Collector 1: Got 3
Collector 2: Got 1
Collector 1: Got 4
Collector 2: Got 2
Collector 1: Got 5
Collector 2: Got 3
Collector 2: Got 4
Collector 2: Got 5

```

Каждый потребитель потока данных, построенного с использованием строителя `flow`, обладает собственным состоянием, и новые потребители ведут себя так, как если бы поток данных никогда не потреблялся. Если вы захотите отправить новому потребителю новые значения (но не старые!), вы увидите, что текущий инструмент плохо подходит для этой задачи. Как вы уже видели, при использовании строителя `flow` второй потребитель начнет с самого начала и рассинхронизируется с предыдущим потребителем. А при попытке использовать `MutableStateFlow` вы увидите, что хотя коллекции синхронизированы, второй потребитель немедленно получит последнее сгенерированное значение, когда начнет потреблять данные, что нас не устраивает, потому что никакие старые значения потребляться не должны.

А вот `MutableSharedFlow` идеально подойдет для этой работы.

Как следует из названия, `MutableSharedFlow` обеспечивает совместное использование генерируемых данных между всеми потребителями. Как и `MutableStateFlow`, он всегда активен и никогда не завершается. Но потребители `MutableSharedFlow` получают только данные, отправленные после того, как они начали потребление.

Опробуйте `MutableSharedFlow` в REPL вместо строителя `flow`.

Листинг 21.19. Использование `MutableSharedFlow` (REPL)

```

runBlocking {
    val numbersFlow = MutableSharedFlow<Int>()
    launch {
        numbersFlow.collect { println("Collector 1: Got $it"\n) }
    }
    launch {
        delay(2200)
        numbersFlow.collect { println("Collector 2: Got $it"\n) }
    }
}

```

```
(1..5).forEach {  
    delay(1000)  
    numbersFlow.emit(it)  
}  
}  
Collector 1: Got 1  
Collector 1: Got 2  
Collector 1: Got 3  
Collector 2: Got 3  
Collector 1: Got 4  
Collector 2: Got 4  
Collector 1: Got 5  
Collector 2: Got 5
```

Теперь потребители остаются синхронизированными и получают одни и те же значения.

Следует заметить, что потребление начинается до генерирования каких-либо элементов в поток. `SharedFlow`, как мы уже говорили, находится в активном состоянии, как и `MutableStateFlow`. Но в отличие от `MutableStateFlow`, `SharedFlow` не буферизуется и не отправляет заново старые значения новым потребителям. Если поставить вызов `emit` перед вызовами `collect`, ни один из потребителей не получит первые сгенерированные данные.

При желании это поведение можно изменить. Конструктор `MutableSharedFlow` получает три параметра. Первый параметр — `replay` — определяет количество значений, которые необходимо запомнить и отправить потребителям, которые будут добавлены в будущем (по умолчанию 0).

Второй параметр — `extraBufferCapacity` — определяет дополнительные значения, которые могут буферизоваться в том случае, если поток в настоящее время не имеет потребителей (он также равен 0 по умолчанию, чтобы значения терялись при отсутствии потребителей).

Третий параметр — `onBufferOverflow` — используется только в том случае, если значение `extraBufferCapacity` не менее 1. Этот параметр управляет поведением при вызове `emit` для потока данных, когда его буфер заполнен. По умолчанию происходит приостановка до того момента, пока буфер не опустеет.

Поведение `MutableStateFlow` можно моделировать с использованием `MutableSharedFlow`; для этого следует присвоить `replay` значение 1, а другим аргументам оставить значения по умолчанию.

В общем случае для создания потоков данных чаще всего применяют `MutableStateFlow` и строитель `flow`. Если вы строите поток данных для одного потребителя и можете определить жизненный цикл сгенерированных элементов в одном месте, выбирайте строитель `flow.MutableStateFlow` хорошо работает в большинстве остальных случаев. Но если вы обнаружите, что поведение `MutableStateFlow` вас ограничивает, воспользуйтесь `MutableSharedFlow`.

22. Каналы

В предыдущей главе мы использовали класс `Flow` для построения потоков данных с меняющейся во времени информацией о рейсах. Потоки данных позволяют хранить состояние приложения в свойстве, состояние которого можно отслеживать и реагировать на эти изменения. Но иногда взаимодействовать друг с другом должны не два компонента, а две *сопрограммы*.

Если вы хотите передавать сообщения между двумя сопрограммами, вам необходимо знать, что у `Flow` есть некоторые ограничения. `Flows` не гарантирует отправителю, что сгенерированное значение будет потреблено. И если потребителей несколько, все они могут получить одно значение. В некоторых случаях значение должно быть получено только одной сопрограммой, а этого трудно добиться при помощи потока данных.

Если вы хотите решить проблему передачи данных между сопрограммами, вам безусловно стоит обратить внимание на каналы. *Канал* (`channel`) — коммуникационный путь, у которого есть получатели и отправители. Когда отправитель выдает сообщение в канал, он должен ожидать, пока сообщение доберется до получателя. Аналогично, когда получатель хочет получить сообщение из канала, он должен ожидать, пока отправитель поместит его в канал.

Получатель в каждый момент времени может принять только одно сообщение. Если сразу несколько получателей ожидают сообщение, то придет оно только одному из них — остальные будут ждать. (Если вам известна концепция блокирующей очереди, вы заметите, что у каналов с ней много общего.)

Существует много разных вариантов использования такого механизма. В этой главе мы с помощью каналов ускорим загрузку набора отслеживаемых рейсов.

Распределение работы с использованием каналов

Чтобы ускорить загрузку данных о рейсах, мы снова увеличим количество сетевых запросов, которые выдаются параллельно нашим приложением. Ранее для этой задачи мы применяли ключевые слова `async` и `await`. Такое решение хорошо работало, потому что параллельных запросов было немного и они комбинировались достаточно тривиальным образом.

Однако использование `async` и `await` для получения информации о рейсах имеет свои недостатки. На верхнем уровне проблемы сводятся к способу управления запросами: `async` и `await` выполняют сетевые запросы как можно быстрее. Если требуется мониторить много рейсов, то одновременно отправляется множество сетевых запросов, что вызывает перегрузку сети или сервера.

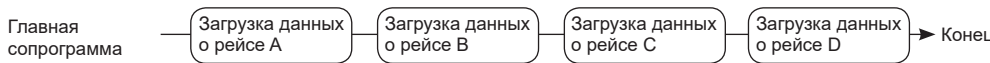
Мы ограничимся выборкой данных из двух рейсов. Для каждого рейса будем выводить информацию о самом рейсе и уровень лояльности — и эти два сетевых запроса мы уже выполним одновременно с использованием `async` и `await`, значит, теперь приложение будет выдавать до четырех сетевых запросов параллельно (вместо двух в предыдущей версии). Это ускорит отображение информации о полетах и гарантирует, что приложение не будет перегружать серверы и сетевое подключение пользователя.

Реализовать такое параллельное выполнение задач непросто. Как узнать, сколько рейсов загружаются в настоящее время? Когда вы сможете начать работать над следующим рейсом? Какие средства позволят вам разумно выбрать условия для запуска следующего запроса?

Для координации загрузки данных о рейсах мы используем три сопрограммы. Первая отвечает за производство и делегирование работы. Вторая и третья сопрограммы будут *работниками* (*workers*): их назначение — ожидать запроса со стороны производителя на загрузку данных о рейсе.

Когда производитель выдаст запрос, одна из двух сопрограмм-работников потребит его и приступит к получению данных о рейсе. Другой работник будет ожидать следующего запроса. Последовательность операций показана на рис. 22.1, где она сравнивается с текущей реализацией, использующей одну сопрограмму в цикле `foreach` для последовательного получения каждого рейса.

Текущая схема: цикл `foreach`



В конце главы: несколько работников

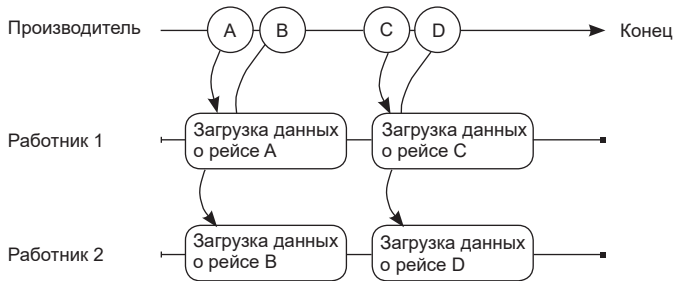


Рис. 22.1. Схемы с одним и несколькими работниками

Описание кажется запутанным, но вы увидите, что каналы предоставляют средства для компактной реализации взаимодействий между тремя сопрограммами. Результат: данные набора рейсов будут загружаться приблизительно в два раза быстрее, потому что два работника загружают их параллельно.

Отправка данных в канал

Начнем с производителя. Это будет отдельная сопрограмма, которая помещает запросы на получение данных о рейсах в канал. Обновите функцию `fetchFlights` в файле `FlightWatcher.kt`; добавьте канал для запросов на выполнение работы. Функция возвращает временный результат, а мы продолжим заниматься расширением реализации.

Листинг 22.1. Отправка запросов по каналу (FlightWatcher.kt)

```
...
suspend fun fetchFlights(
    passengerNames: List<String> = listOf("Madrigal", "Polarcubis")
) = passengerNames.map { fetchFlight(it) }
): List<FlightStatus> = coroutineScope {
    val passengerNamesChannel = Channel<String>()
    launch {
        passengerNames.forEach {
            passengerNamesChannel.send(it)
        }
    }

    emptyList()
}
```

Не забудьте импортировать класс `Channel` из `kotlinx.coroutines`.

Созданный вами канал `passengerNamesChannel` содержит запросы на выполнение работы. Сами запросы хранят имена пассажиров для загружаемых рейсов — входные данные для функции `fetchFlights` из главы 20.

После создания канала запускается сопрограмма-производитель. Сопрограмма помещает все названия рейсов в канал.

Обратите внимание на вызов `send`, который среда IntelliJ помечает как приостанавливаемую функцию. `send` приостанавливается до того момента, когда отправленное значение будет получено другой сопрограммой. Если запустить код прямо сейчас, то "Madrigal" станет единственным значением, которое будет отправлено в канал. Сейчас ничто не получает значения из канала, поэтому вызов `send` будет приостановлен навсегда. Следующая задача — создать работника, получающего значения из канала.

Получение данных из канала

Когда производитель будет готов, можно переходить к созданию первого работника. Добавьте функцию, определяющую поведение работника, а затем запустите ее в отдельной сопрограмме.

Листинг 22.2. Получение данных из канала (FlightWatcher.kt)

```
...
suspend fun fetchFlights(
    passengerNames: List<String> = listOf("Madrigal", "Polarcubis")
): List<FlightStatus> = coroutineScope {
    val passengerNamesChannel = Channel<String>()

    launch {
        passengerNames.forEach {
            passengerNamesChannel.send(it)
        }
    }

    launch {
        fetchFlightStatuses(passengerNamesChannel)
    }

    emptyList()
}

suspend fun fetchFlightStatuses(
    fetchChannel: Channel<String>
) {
    val passengerName = fetchChannel.receive()
    val flight = fetchFlight(passengerName)
    println("Fetched flight: $flight")
}
```

`receive` — еще одна приостанавливаемая функция. Как и `send`, она приостанавливается до отправки значения в канал. Если две сопрограммы пытаются одновременно получить данные из одного канала, значение получит только одна из них.

Неизвестно, какая сопрограмма получит значение в такой ситуации, но в данной главе это несущественно. Для нас важно только то, что работа выполняется лишь один раз, а каналы справляются с ней автоматически.

Запустите `TaernylAir`. Результат выглядит примерно так:

```
Getting the latest flight info...
Started fetching flight info
Started fetching loyalty info
Combining flight data
Finished fetching loyalty info
Finished fetching flight stats
Fetched flight: FlightStatus(flightNumber=MK1737, passengerName=Madrigal, ...)
```

Мы получили информацию только об одном рейсе, но хотим отслеживать данные о двух пассажирах. Кроме того, программа никогда не завершит моделирование и не прервет выполнение по собственной инициативе. Она останется в таком состоянии навсегда. Нажмите кнопку остановки , чтобы завершить ее выполнение. Что происходит?

Функция `receive` возвращает только одно значение из канала. Работник должен продолжать обработку запросов, пока остается незавершенная работа, поэтому одного вызова `receive` недостаточно. Одно из возможных решений — продолжать получение элементов из канала в цикле `for`, пока они не будут исчерпаны.

Добавьте цикл `for` в `fetchFlightStatuses`:

Листинг 22.3. Получение всех значений из канала (FlightWatcher.kt)

```
...
suspend fun fetchFlightStatuses(
    fetchChannel: Channel<String>
) {
    val passengerName = fetchChannel.receive()
    for (passengerName in fetchChannel) {
        val flight = fetchFlight(passengerName)
        println("Fetched flight: $flight")
    }
}
```

Снова запустите TaernylAir. Вывод будет выглядеть примерно так:

```
Getting the latest flight info...
Started fetching flight info
Started fetching loyalty info
Combining flight data
Finished fetching loyalty info
Finished fetching flight stats
Fetched flight: FlightStatus(flightNumber=GZ2871, passengerName=Madrigal, ..)
Started fetching flight info
Started fetching loyalty info
Combining flight data
Finished fetching loyalty info
Finished fetching flight stats
Fetched flight: FlightStatus(flightNumber=EH0675, passengerName=Polarcubis,
...)
```

Теперь данные о рейсах загружаются как для `Madrigal`, так и для `Polarcubis`, но моделирование все еще не запускается. Кроме того, результаты не будут загружаться быстрее, так как работник только один и код работает без дополнительного параллелизма.

Прежде чем добавлять второго работника, необходимо избавиться от некоторых проблем. Самая большая — работник ничего не делает после получения данных

о рейсе. Результат просто выводится, а потом отбрасывается. Работнику нужно куда-то поместить готовые результаты.

Для этого нам нужен еще один канал, который будет отвечать за передачу полученных данных о рейсах. Он будет содержать готовые объекты `FlightStatus`.

Обновите функции `fetchFlights` и `fetchFlightStatuses`, чтобы создать второй канал, отправить в него обработанные рейсы и реализовать возвращаемое значение `fetchFlights`.

Листинг 22.4. Отправка результатов от работника (FlightWatcher.kt)

```
...
suspend fun fetchFlights(
    passengerNames: List<String> = listOf("Madrigal", "Polarcubis")
): List<FlightStatus> = coroutineScope {
    val passengerNamesChannel = Channel<String>()
    val fetchedFlightsChannel = Channel<FlightStatus>()

    launch {
        passengerNames.forEach {
            passengerNamesChannel.send(it)
        }
    }

    launch {
        fetchFlightStatuses(passengerNamesChannel, fetchedFlightsChannel)
    }

    emptyList()
    fetchedFlightsChannel.toList()
}

suspend fun fetchFlightStatuses(
    fetchChannel: Channel<String>,
    resultChannel: Channel<FlightStatus>
) {
    for (passengerName in fetchChannel) {
        val flight = fetchFlight(passengerName)
        println("Fetched flight: $flight")
        resultChannel.send(flight)
    }
}
```

Чтобы функция `fetchFlightStatuses` лучше читалась, можно воспользоваться типами `SendChannel` и `ReceiveChannel`. Как нетрудно догадаться, интерфейс `SendChannel` способен только отправлять значения в канал, а `ReceiveChannel` — только получать элементы из канала.

`Channel` реализует оба интерфейса. Экземпляр `Channel` можно привести к любому из этих типов для сужения его функциональности. Это иногда полезно, чтобы

исключить случайную отправку значений или чтение значений из неподходящего канала. Обновите функцию `fetchFlightStatuses`.

Листинг 22.5. Сужение функциональности канала (FlightWatcher.kt)

```
...
suspend fun fetchFlightStatuses(
    fetchChannel: ReceiveChannel<String>,
    resultChannel: SendChannel<FlightStatus>
) {
    for (passengerName in fetchChannel) {
        val flight = fetchFlight(passengerName)
        println("Fetched flight: $flight")
        resultChannel.send(flight)
    }
}
```

Новые типы гарантируют, что входные и выходные данные перемещаются в нужном направлении. Если теперь работник попытается отправить новую работу вне очереди вызовом `send` для `fetchChannel`, компилятор сообщит об ошибке, потому что функция `send` недоступна.

Снова запустите `TaernylAir`. Хотя внутренняя реализация была улучшена, вы увидите уже знакомое поведение: появится информация о двух рейсах, но моделирование так и не начинается, а программа переходит в бесконечное ожидание. Чтобы понять, что же пошло не так, необходимо разобраться, как закрыть канал.

Заккрытие канала

В главе 21 мы показали, что потоки данных могут завершаться, — это означает, что они перестают генерировать элементы. Завершение потока данных может происходить несколькими способами. В случае строителя `flow`, получающего лямбда-выражение, поток неявно завершается при возврате из лямбда-выражения.

Потоки могут также закрываться, чтобы дальнейшая отправка или получение значений по каналу стали невозможными. Но каналы, которые мы создавали до сих пор, не обладали возможностью неявного закрытия. Требовалось явно вызвать `close` для канала, когда генерирование элементов завершилось.

Вам понадобятся два вызова `close`. Первый будет относиться к `passengerNamesChannel`. Закрытие этого канала позволит работникам завершиться. Если канал не закрылся, ваш цикл `for` приостанавливается на неопределенное время, ожидая отправки другого значения по каналу.

Также необходимо вызвать `close` для `fetchedFlightsChannel`. Закрытие этого канала сообщает функции `toList()`, что все элементы были получены. Это позволяет списку завершиться, а функции `fetchFlights` — вернуть управление.

Добавьте два вызова `close` в `fetchFlights`.

Листинг 22.6. Заккрытие каналов (FlightWatcher.kt)

```

...
suspend fun fetchFlights(
    passengerNames: List<String> = listOf("Madrigal", "Polarcubis")
): List<FlightStatus> = coroutineScope {
    val passengerNamesChannel = Channel<String>()
    val fetchedFlightsChannel = Channel<FlightStatus>()

    launch {
        passengerNames.forEach {
            passengerNamesChannel.send(it)
        }
        passengerNamesChannel.close()
    }

    launch {
        fetchFlightStatuses(passengerNamesChannel, fetchedFlightsChannel)
        fetchedFlightsChannel.close()
    }

    fetchedFlightsChannel.toList()
}
...

```

Снова запустите TaernylAir. Программа загружает данные по обоим рейсам, и моделирование начинается. Кроме того, после окончания моделирования программа завершается (а не приостанавливается навсегда). Вывод совпадает с тем, который был приведен в конце предыдущей главы.

Вспомните, что мы ставили целью параллельную загрузку данных о двух рейсах. Так как сейчас работник только один, на этой стадии вы не заметите никакого выигрыша в скорости. И хотя кажется, что мы не продвинулись вперед, на самом деле мы подготовили все необходимое для добавления второго работника.

Объединение заданий

Производитель и первый работник готовы, можно добавлять второго. Добавьте параметр для управления количеством работников, которых разрешено использовать, затем запустите дополнительные сопрограммы-работники. Кроме того, добавьте еще двух людей в список пассажиров, чтобы увеличить количество рейсов для моделирования.

Листинг 22.7. Запуск нескольких работников (FlightWatcher.kt)

```

...
suspend fun fetchFlights(
    passengerNames: List<String> = listOf("Madrigal", "Polarcubis",
        "Estragon", "Taernyl1"),

```

```
    numberOfWorkers: Int = 2
): List<FlightStatus> = coroutineScope {
    val passengerNamesChannel = Channel<String>()
    val fetchedFlightsChannel = Channel<FlightStatus>()

    launch {
        passengerNames.forEach {
            passengerNamesChannel.send(it)
        }
        passengerNamesChannel.close()
    }

    repeat(numberOfWorkers) {
        launch {
            fetchFlightStatuses(passengerNamesChannel, fetchedFlightsChannel)
            fetchedFlightsChannel.close()
        }
    }

    fetchedFlightsChannel.toList()
}
...

```

Запустите TaarnylAir после внесения изменений. Приложение начинает загружать данные о рейсах, но после четвертого рейса завершается со следующей ошибкой:

```
ClosedSendChannelException: Channel was closed
```

Дело в том, что `fetchedFlightsChannel` закрывается слишком рано. Работник, завершающийся первым, также закрывает канал, что делает невозможной отправку значений от других работников.

Чтобы решить эту проблему, необходимо дождаться завершения всех работников перед закрытием `fetchedFlightsChannel`. Так как время выполнения сетевых запросов неизвестно заранее, а количество запросов также является переменной величиной, неизвестно, какой работник завершится первым. Нам необходимо следить, какие работники еще выполняются, и дождаться их завершения.

К счастью, вам уже известен инструмент, который поможет в этом. Вспомните, что строители сопрограмм, например `launch` возвращают объект задания `Job`. Задания содержат информацию о состоянии сопрограммы и позволяют выполнять такие операции, как досрочная отмена сопрограммы.

У заданий также имеется приостанавливаемая функция — `join`. При вызове она приостанавливается до завершения задания. Если задание уже завершено, то `join` не приостанавливается, и оставшаяся часть кода продолжит выполняться.

В Kotlin также включена функция-расширение для `List<Job>` с именем `joinAll`, которая приостанавливается и ожидает завершения всех заданий в списке. При отслеживании заданий, связанных с работниками, вы можете воспользоваться

функцией `joinAll`, чтобы дождаться завершения всех работников перед закрытием `fetchFlightsChannel`.

Обновите логику создания работников, чтобы запоминать задания работников и дожидаться их завершения, прежде чем закрывать `fetchFlightsChannel`.

Листинг 22.8. Объединение заданий (FlightWatcher.kt)

```
...
suspend fun fetchFlights(
    passengerNames: List<String> = listOf("Madrigal", "Polarcubis",
        "Estragon", "Taernyl"),
    numberOfWorkers: Int = 2
): List<FlightStatus> = coroutineScope {
    val passengerNamesChannel = Channel<String>()
    val fetchedFlightsChannel = Channel<FlightStatus>()

    launch {
        passengerNames.forEach {
            passengerNamesChannel.send(it)
        }
        passengerNamesChannel.close()
    }

    repeat(numberOfWorkers) {
        launch {
            fetchFlightStatuses(passengerNamesChannel, fetchedFlightsChannel)
            fetchedFlightsChannel.close()
        }
    }
    launch {
        (1..numberOfWorkers).map {
            launch {
                fetchFlightStatuses(passengerNamesChannel, fetchedFlightsChannel)
            }
        }.joinAll()
        fetchedFlightsChannel.close()
    }
    fetchedFlightsChannel.toList()
}
...
```

Снова запустите приложение. Оно получает данные для всех четырех пассажиров и начинает моделирование, как показано ниже. Из вывода ясно, что в приложении выполняются несколько сетевых запросов одновременно.

```
Getting the latest flight info...
Started fetching flight info
Started fetching loyalty info
```



```
Started fetching flight info
Started fetching loyalty info
Combining flight data
Combining flight data
Finished fetching loyalty info
Finished fetching loyalty info
Finished fetching flight stats
Finished fetching flight stats
Fetched flight: FlightStatus(flightNumber=RX7759, passengerName=Madrigal, ...)
Fetched flight: FlightStatus(flightNumber=UC4790, passengerName=Polarcubis,
...)
...
Fetched flight: FlightStatus(flightNumber=TF3942, passengerName=Taernyl, ...)
Fetched flight: FlightStatus(flightNumber=RD2604, passengerName=Estragon, ...)
Found flights for Madrigal (RX7759), Polarcubis (UC4790), Taernyl (TF3942),
    Estragon (RD2604)
There are 4 flights being tracked
Madrigal: Other passengers are boarding (Flight departs in 28 minutes)
...
Madrigal: The boarding doors have closed (Flight departs in 0 minutes)
Finished tracking Madrigal's flight
There are 3 flights being tracked
Polarcubis: Boarding will start soon (Flight departs in 111 minutes)
...
Polarcubis: The boarding doors have closed (Flight departs in 0 minutes)
Finished tracking Polarcubis's flight
There are 2 flights being tracked
Taernyl: You can now board the plane (Flight departs in 55 minutes)
...
Taernyl: The boarding doors have closed (Flight departs in 0 minutes)
Finished tracking Taernyl's flight
There are 1 flights being tracked
Estragon: Your flight was canceled (Flight departs in 45 minutes)
Finished tracking Estragon's flight
Finished tracking all flights
```

(Если рейс пассажира отменен или отправляется немедленно, вы увидите только одно сообщение о статусе рейса, за которым следует текст «finished tracking flight» — «отслеживание рейса завершено». Также вы можете заметить, что пассажиры отслеживаются в несколько ином порядке. Перебор пассажиров выполняется в том порядке, в каком завершилась загрузка данных, и он может отличаться от порядка их объявления.)

Обратите внимание: в начале вывода две идентичные записи следуют друг за другом, то есть данные о двух рейсах загружаются параллельно. Проведя хронометраж, вы увидите, что теперь загрузка занимает около 10 секунд (на надежном сетевом подключении); до корректировки кода она занимала 20 секунд.

Такое применение каналов называется расхождением (*fan-out*), потому что канал расширяется при переходе от производителя к рабочим. Каналы также используют для создания сценария схождения (*fan-in*), когда несколько сопрограмм отправляют данные в один канал.

На верхнем уровне каналы и потоки данных очень похожи. И те и другие используются с сопрограммами и выдают значения потребителям с течением времени. Но они применяются в разных ситуациях.

Подумайте, как бы выглядел ваш код, если бы вы попытались реализовать свою логику расхождения с использованием `Flow` вместо `Channel`. Когда вы отправляете значение во `Flow`, нет никаких гарантий относительно того, сколько потребителей получают значение. Если ни один компонент не потребляет данные из `Flow`, значение может вообще никуда не попасть. А если потребителей несколько, то все они могут получить одно и то же значение. Для нашей реализации с расхождением ситуация не идеальна, потому что мы хотим, чтобы каждый запрос отправлялся только одному работнику.

С другой стороны, каналы плохо подойдут для потоков с данными статуса полета, созданными в главе 21. Эти потоки содержат данные и состояние, относящиеся к приложению. Если нескольким частям приложения требуется доступ к одним и тем же данным, например к информации о посадке для Мадригала, то вам нужно, чтобы каждый из потребителей получил одни и те же значения. А иногда — например, представим, что Мадригал перешел к другому терминалу, на котором его текущий рейс не выводится, — эта информация не нужна, и для `Flow` не будет потребителей.

Вероятно, вы будете использовать `Flow` чаще, чем `Channel`, но оба инструмента бесценны. Каналы идеально подходят в тех ситуациях, когда у вас есть две независимые сопрограммы, между которыми необходимо наладить безопасное взаимодействие. Для всех других обозримых потоков данных и для моделирования состояния приложения стоит обращаться к `Flow`.

Приложение `TaepnylAir` завершено. Теперь пассажирам доступна вся информация, необходимая для посадки, и пользователи, отслеживающие несколько рейсов, не потратят много времени.

В следующей части книги вы узнаете, как `Kotlin` взаимодействует с кодом `Java`, благодаря чему код `Kotlin` можно включать в существующий проект `Java`. Вы также познакомитесь с мультиплатформенными средствами `Kotlin`, позволяющими использовать его код за пределами JVM.

Для любознательных: другие особенности поведения каналов

Существует несколько разновидностей каналов. Тип канала определяет поведение функций `send` и `receive`.

Встречные каналы (rendezvous)

При вызове конструктора `Channel` без аргументов по умолчанию будет получен *встречный канал* (rendezvous channel). Именно такую разновидность каналов мы использовали в этой главе.

Встречный канал не имеет буфера. Когда вы отправляете элемент во встречный канал, функция `send` приостанавливается до того момента, когда получатель вызовет `receive` для получения значения. Аналогичным образом, если вы вызовете `receive` до `send`, код приостановится при отправке значения в канал. (Почему встречные? Вызовы `send` и `receive` как бы ожидают возможности встретиться друг с другом.)

Буферизованные каналы (buffered)

При создании канала также можно задать размер буфера. В результате вы получаете *буферизованный канал* (buffered channel).

Буферизованный канал создается двумя способами; они показаны ниже:

```
val defaultBufferedChannel = Channel(BUFFERED)
val bufferSize = 5
val bufferedChannel = Channel(bufferSize)
```

В первом примере для `Channel` используется размер буфера, установленный по умолчанию для среды выполнения (обычно 64). Во втором примере выбран нестандартный размер буфера: переданное значение `Int` указывает, сколько элементов можно разместить в буфере.

Если при работе с буферизованным каналом буфер не заполнен, то при вызове `send` значение помещается в буфер, а вызов `send` может не приостанавливаться. Если `send` вызывается при заполненном буфере, вы получаете то же поведение, как у встречного канала: вызов `send` приостанавливается до того момента, когда значение будет извлечено из буфера вызовом `receive`.

Буфер функционирует как очередь, работающая по принципу FIFO (First-In-First-Out, то есть «первым пришел, первым вышел».) Если буфер не пуст, то получатель немедленно получает самое старое значение из буфера, без приостановки. Если буфер пуст, то вызов `receive` приостанавливается до отправки значения в канал.

Неограниченные каналы (unlimited)

Неограниченные каналы (unlimited channels) — это буферизованные каналы с неограниченным размером буфера. Если программа располагает достаточным объемом памяти, элементы могут добавляться в буфер канала. Если памяти не хватает, в программе произойдет фатальный сбой.

На практике это означает, что каждый вызов `send` для канала будет завершаться немедленно, без приостановки. (Функция `send` остается приостанавливаемой и должна вызываться из сопрограммы, но без реальной приостановки.) Получатели ведут себя так же, как с буферизованными каналами: если буфер не пуст, получатель немедленно получает самое старое значение из буфера. Если буфер пуст, вызов `receive` приостанавливается до отправки значения.

Чтобы создать неограниченный канал, передайте константу `Channel.UNLIMITED` в аргументе конструктора `Channel`:

```
val unlimitedChannel = Channel(UNLIMITED)
```

Каналы с заменой (conflated)

Еще одна — и последняя — разновидность каналов, которые может создавать конструктор `Channel`, — так называемые *каналы с заменой* (conflated channels).

Канал с заменой позволяет буферизовать один элемент и заменяет элемент в буфере новыми значениями вместо приостановки. Данная возможность полезна в том случае, если с появлением нового значения прежнее содержимое буфера теряет актуальность. Например, если канал отправляет события в ответ на нажатие пользователем кнопки в программе, вас интересует только самое последнее нажатие.

Как и с неограниченными каналами, функция `send` никогда не приостанавливается. Функция `receive` ведет себя так же, как с другими буферизованными каналами: если значение находится в буфере, то оно используется и удаляется из буфера (так как буфер содержит только одно значение, любое хранимое значение по определению является самым старым). Если в буфере нет значения, `receive` приостанавливается до отправки `send`.

Чтобы получить канал с заменой, передайте константу `Channel.CONFLATED` в аргументе конструктора `Channel`:

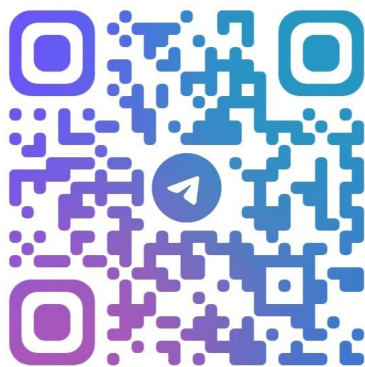
```
val conflatedChannel = Channel(CONFLATED)
```

Существуют и другие аргументы, с которыми можно поэкспериментировать при создании канала. Для каналов с буфером фиксированного размера (не включая канал с заменой) можно задать альтернативное поведение того, что должно происходить при заполнении буфера.

Например, рассмотрим буферизованный канал, используемый для обработки нажатий клавиш или перемещений мыши. Если события передаются каналу быстрее, чем обрабатываются, вы можете задать стратегию, определяющую, что должно происходить при заполнении буфера событиями. Один из вариантов — аргумент `BufferOverflow.DROP_LATEST`, который блокирует дальнейшую отправку событий в канал при заполнении буфера.

Как правило, встречных или буферизованных каналов будет достаточно для большинства ваших потребностей, особенно с учетом возможностей настройки буферизованных каналов.

Подписывайся на самый топовый канал по Kotlin:
<https://t.me/KotlinSenior>



@KOTLINSENIOR

Часть VI

Совместимость и мультиплатформенные приложения

До сих пор мы использовали в книге Kotlin/JVM для написания кода Kotlin, который выполняется на виртуальной машине Java. Но во введении мы упоминали, что Kotlin также может работать вне JVM. Kotlin/JS позволяет писать код Kotlin для веб-приложений, а Kotlin/Native позволяет коду Kotlin работать на таких платформах, как iOS, macOS, Windows, Linux и т. д.

В этой части книги речь пойдет о том, как средства взаимодействия Kotlin работают на разных платформах. Сначала мы расскажем о совместимости Kotlin с Java, упрощающей интеграцию Kotlin в существующие кодовые базы. Затем мы напишем мультиплатформенное приложение, предназначенное для JVM, настольных систем macOS и веб-приложения, — с единой кодовой базой и совместным использованием кода Kotlin на разных платформах.

23. Совместимость с Java

Есть много разных причин для изучения Kotlin. Java-разработчикам, например, Kotlin предлагает более современный и безопасный язык для существующих проектов. Если вы принадлежите к их числу, надеемся, что книга вдохновит вас использовать Kotlin для улучшения ваших Java-проектов.

Другие изучают Kotlin, чтобы применять его в проектах, написанных исключительно на этом языке. Но даже в таких случаях у программистов время от времени возникает необходимость во взаимодействии с Java. Возможно, вы работаете с фреймворком, написанным на Java, или используете библиотеку, которая предъявляет определенные требования к структуре вывода программы. Понимание проблем совместимости позволит вам достойно выходить из трудных ситуаций.

До сих пор в книге использовалась платформа Kotlin/JVM; это означает, что Kotlin компилируется в байт-код Java. Так как этот байт-код не отличается от того, что генерируется для обычного кода Java, Kotlin совместим с Java, то есть может выполняться вместе с кодом Java.

Пожалуй, это одна из самых главных особенностей языка программирования Kotlin. Полная совместимость с Java означает, что файлы Kotlin и файлы Java могут существовать в одном проекте. Вам удастся вызывать методы Java из Kotlin, а функции Kotlin — из Java либо использовать существующие библиотеки и фреймворки Java в Kotlin (два важных примера — фреймворки Android и Spring).

Полная совместимость с Java подразумевает, что кодовую базу можно постепенно переводить с Java на Kotlin. Если вы не можете или не хотите полностью переписать существующий проект на языке Kotlin, рассмотрите Kotlin для разработки будущих проектов. Возможно, вы захотите преобразовать только те файлы Java, которые получают наибольшее преимущество от перехода на Kotlin. В этом случае, вероятно, это будет полезно для модельных объектов или юнит-тестов.

В этой главе мы расскажем, как совмещаются файлы Java и Kotlin, а также о чем стоит помнить при написании кода, чтобы он был совместимым.

Взаимодействие с классом Java

Для примеров этой главы создайте в IntelliJ новый проект с именем Interop. Последовательность действий такая же, как ранее: выберите шаблон `Application` и нужный

вариант в списке **Project JDK**. Шаблон включает все, что необходимо для объявления кода Java наряду с кодом Kotlin.

Проект **Interop** будет содержать два файла: **Hero.kt** (файл с кодом на Kotlin, который представляет героя из игры **NyetHack**) и **Jhava.java** (класс Java, который представляет монстра из другого измерения).

В этой главе вы напишете код на двух языках — Kotlin и Java. Даже если у вас нет опыта в Java, не бойтесь: примеры на Java будут вам понятны, учитывая уже полученный опыт работы с Kotlin.

Начнем с создания папки для кода Java. Найдите папку **main** на панели проекта IntelliJ (она находится в **src**). Щелкните правой кнопкой мыши на папке **main** и выберите команду **New ► Directory**. Присвойте папке имя **java**. (IntelliJ предлагает такое имя по умолчанию, подтвердите его двойным щелчком.)

Код Java следует хранить в только что созданной вами папке **java**. Если разместить его в используемой по умолчанию папке **kotlin**, то код будет проигнорирован и не будет использоваться в вашем проекте. (А вот код Kotlin может храниться в папке **java**, его поведение от этого не изменится. Такая возможность полезна для разработчиков, желающих постепенно перейти на Kotlin с Java.)

Когда папка **java** будет создана, щелкните на ней правой кнопкой мыши и выберите команду **New ► Java Class**. Введите имя нового класса **Jhava**. В классе **Jhava** определите метод с именем **utterGreeting**, возвращающий строку.

Листинг 23.1. Объявление класса и метода в Java (Jhava.java)

```
public class Jhava {  
    public String utterGreeting() {  
        return "BLARGH";  
    }  
}
```

Теперь создайте новый файл Kotlin с именем **Hero.kt** в **src/main/kotlin**. Определите в нем функцию **main** и объявите переменную для монстра **val adversary**, экземпляр **Jhava**.

Листинг 23.2. Объявление функции main и переменной adversary в Kotlin (Hero.kt)

```
fun main() {  
    val adversary = Jhava()  
}
```

Свершилось! Написав строку кода на Kotlin, вы создали объект Java — и преодолели барьер между двумя языками. Как видите, вызвать код на Java из Kotlin очень просто.

Но нам есть еще что показать. Для проверки выведите приветственный рык монстра **Jhava**.

Листинг 23.3. Вызов метода Java в Kotlin (Hero.kt)

```
fun main() {  
    val adversary = Jhava()  
    println(adversary.utterGreeting())  
}
```

Вы создали объект Java и вызвали его метод, причем сделали все это из Kotlin. Запустите **Hero.kt**. Вы увидите, как монстр приветствует героя (**BLARGH**) в консоли.

Kotlin создавали с прицелом на максимально удобную совместимость с Java. Но разработчики позаботились и о ряде усовершенствований, недоступных в Java. Придется ли вам отказаться от них при интеграции этих двух языков? Конечно нет! Зная о различиях между языками и используя аннотации, доступные на обеих сторонах, вы сможете взять все лучшее, что предлагает Kotlin.

Совместимость и null

Добавим еще один метод в **Jhava** с именем `determineFriendshipLevel` (уровень дружелюбия). `determineFriendshipLevel` должен возвращать значение типа `String`, но так как монстр недружелюбен, метод возвращает `null`.

Листинг 23.4. Возвращение null из метода Java (Jhava.java)

```
public class Jhava {  
    public String utterGreeting() {  
        return "BLARGH";  
    }  
  
    public String determineFriendshipLevel() {  
        return null;  
    }  
}
```

Вызовите этот новый метод из **Hero.kt**, сохранив значение дружелюбия монстра в `val`. Выведите это значение в консоль. Помня, что громкий рык монстра выводился прописными буквами, уровень дружелюбия мы выведем строчными буквами.

Листинг 23.5. Вывод уровня дружелюбия (Hero.kt)

```
fun main() {  
    val adversary = Jhava()  
    println(adversary.utterGreeting())  
  
    val friendshipLevel = adversary.determineFriendshipLevel()  
    println(friendshipLevel.lowercase())  
}
```

Запустите файл `Hero.kt`. Хотя компилятор не сообщает ни о каких проблемах, сразу после запуска программа завершится с ошибкой:

```
BLARGH
Exception in thread "main" java.lang.NullPointerException: friendshipLevel must
not be null
    at HeroKt.main(Hero.kt:6)
    at HeroKt.main(Hero.kt)
```

В Java все объекты могут иметь значение `null`. Вызывая метод Java, такой как `determineFriendshipLevel`, мы видим, что он возвращает `String`, но это не значит, что возвращаемое значение будет соответствовать правилам языка Kotlin в отношении `null`.

Так как все объекты в Java могут быть равны `null`, то безопаснее предположить, что все значения имеют тип с поддержкой `null`, если явно не указано обратное. Следование этому правилу обеспечит безопасность, но код получится более объемным, так как вам придется обрабатывать каждую возможность `null` для каждой переменной Java, на которую вы ссылаетесь.

В `Hero.kt` переместите текстовый курсор к `FriendshipLevel` и нажмите `Ctrl-Shift-P`, чтобы вывести информацию о типе. IntelliJ сообщит, что метод возвращает значение `String!`. Восклицательный знак показывает, что возвращаемым значением может быть `String` или `String?`. Компилятор Kotlin не знает, какое значение будет получено из Java — строка или `null`.

Этот неоднозначный возвращаемый тип называется *платформенным типом*. Платформенные типы не имеют синтаксического смысла: они встречаются только в IDE или другой документации. Вы не сможете определить платформенный тип в коде Kotlin. Он существует только как механизм обеспечения совместимости.

С платформенными типами иногда трудно работать, потому что они скрывают фактическую допустимость значения `null`, о котором идет речь. К счастью, программисты на Java могут писать код, дружественный по отношению к Kotlin, который более явно выражает допустимость `null` с использованием аннотаций допустимости. Чтобы явно объявить, что `determineFriendshipLevel` может вернуть значение `null`, добавьте аннотацию `@Nullable` в заголовок метода.

Листинг 23.6. Объявление допустимости null для возвращаемого значения (Jhava.java)

```
public class Jhava {
    public String utterGreeting() {
        return "BLARGH";
    }
}
```

```
@Nullable
public String determineFriendshipLevel() {
    return null;
}
}
```

(Вам следует импортировать `org.jetbrains.annotations.Nullable`, что и предложит сделать IntelliJ.)

`@Nullable` предупреждает пользователя этого API, что метод может (но не обязан) вернуть `null`. Компилятор Kotlin понимает значение этой аннотации. Вернитесь в `Hero.kt`: вы увидите, что теперь IntelliJ предупреждает вас о прямом вызове `toLowerCase` для `String?`.

Замените прямой вызов безопасным.

Листинг 23.7. Обработка null с помощью оператора безопасного вызова (`Hero.kt`)

```
fun main() {
    val adversary = Jhava()
    println(adversary.utterGreeting())

    val friendshipLevel = adversary.determineFriendshipLevel()
    println(friendshipLevel?.lowercase())
}
```

Запустите `Hero.kt`. Теперь значение `null` будет выведено в консоль.

Так как `friendshipLevel` содержит `null`, возможно, вы захотите определить уровень дружелюбия по умолчанию. Используйте оператор `?:` (объединения с `null`), чтобы вернуть значение по умолчанию в тех случаях, когда `friendshipLevel` содержит `null`.

Листинг 23.8. Определение значения по умолчанию с помощью оператора `?:` (`Hero.kt`)

```
fun main() {
    val adversary = Jhava()
    println(adversary.utterGreeting())

    val friendshipLevel = adversary.determineFriendshipLevel()
    println(friendshipLevel?.lowercase() ?: "It's complicated")
}
```

Запустив `Hero.kt`, вы получите сообщение `It's complicated`.

Аннотация `@Nullable` сообщала, что метод может вернуть `null`. Указать, что значение точно будет отличным от `null`, можно с помощью аннотации `@NotNull`. Эта удобная аннотация дает возможность пользователю API не беспокоиться о том, что возвращаемое значение может быть `null`. Приветствие монстра `Jhava` не должно быть `null`, поэтому добавьте аннотацию `@NotNull` в заголовок метода `utterGreeting`.

Листинг 23.9. Аннотация определяет, что возвращаемое значение отлично от null (Jhava.java)

```
public class Jhava {  
  
    @NotNull  
    public String utterGreeting() {  
        return "BLARGH";  
    }  
  
    @Nullable  
    public String determineFriendshipLevel() {  
        return null;  
    }  
}
```

(Не забудьте импортировать аннотации.)

Аннотации `@Nullable` и `@NotNull` можно использовать для уточнения контекста возвращаемых значений, параметров и даже полей.

Kotlin предоставляет разные инструменты для обработки значений `null`, включая возможность запрета обычным типам принимать значение `null`. Самый распространенный источник ошибок со значениями `null` в Kotlin — это взаимодействие с кодом на Java, так что будьте аккуратны при вызове кода Java из Kotlin.

Соответствие типов

Между типами Kotlin и Java часто существует прямое соответствие. `String` в Kotlin также остается `String`, когда компилируется в Java. Это означает, что `String`, возвращаемый методом Java, можно использовать в Kotlin, как если бы это значение было получено непосредственно в Kotlin.

Однако есть типы, не имеющие прямых аналогов, например, базовые типы. Как мы уже упоминали в разделе «Для любознательных: примитивные типы Java в Kotlin» главы 2, Java представляет базовые типы данных через *примитивы*. Примитивы в Java — это не объекты, однако в Kotlin объектами являются все типы, включая базовые. Несмотря на это, компилятор Kotlin может отображать примитивные типы Java в наиболее похожие типы в Kotlin.

Чтобы понять, как происходит такое преобразование, добавьте целочисленное значение `hitPoints` в `Jhava`. Целое число представляется объектом типа `Int` в Kotlin и примитивом `int` в Java.

Листинг 23.10. Объявление `int` в Java (Jhava.java)

```
public class Jhava {  
  
    public int hitPoints = 52489112;  
}
```

```
@NotNull
public String utterGreeting() {
    return "BLARGH";
}

@Nullable
public String determineFriendshipLevel() {
    return null;
}
}
```

Теперь получим ссылку на `hitPoints` в `Hero.kt`.

Листинг 23.11. Обращение к полю Java из Kotlin (Hero.kt)

```
fun main() {
    val adversary = Jhava()
    println(adversary.utterGreeting())

    val friendshipLevel = adversary.determineFriendshipLevel()\
    println(friendshipLevel?.lowercase() ?: "It's complicated")

    val adversaryHitPoints: Int = adversary.hitPoints
}
```

Несмотря на то что поле `hitPoints` объявлено в `Jhava` как `int`, у вас не возникло проблем при обращении к нему как к `Int`. (Мы не используем автоматическое определение типов в примере только для того, чтобы показать, как происходит преобразование типов. Явные объявления типов не нужны для интеграции языков: объявление `val adversaryHitPoints = adversary.hitPoints` работает так же, и автоматически определенным типом будет `Int`.)

Теперь, когда у вас есть ссылка на целое число, вы можете вызывать с ним функции. Чтобы убедиться, что здоровье монстра не превышает 100, воспользуйтесь функцией `coerceAtMost`.

Листинг 23.12. Вызов функции для поля Java из Kotlin (Hero.kt)

```
fun main() {
    ...
    val adversaryHitPoints: Int = adversary.hitPoints
    println(adversaryHitPoints.coerceAtMost(100))
}
```

Запустите `Hero.kt` для вывода очков здоровья противника. В консоли появится значение `100`.

В Java нельзя вызывать методы для простых типов. В Kotlin целое число `adversaryHitPoints` является объектом типа `Int`, и вы можете вызывать его функции.

В качестве еще одного примера преобразования типов выведите имя класса `Java`, которому принадлежит `adversaryHitPoints`.

Листинг 23.13. Вывод имени класса Java (Hero.kt)

```
fun main() {  
    ...  
    val adversaryHitPoints: Int = adversary.hitPoints  
    println(adversaryHitPoints.coerceAtMost(100))  
    println(adversaryHitPoints.javaClass)  
}
```

Если вы запустите `Hero.kt`, то увидите `int` в консоли. Хотя вы можете вызывать функции типа `Int` для `adversaryHitPoints`, во время работы программы переменная является примитивом `int`. А теперь давайте вспомним байт-код из главы 2. В нем все типы Kotlin преобразуются в свои аналоги Java. Kotlin дает вам мощь объектов и сохраняет производительность примитивов, когда они понадобятся.

Get-методы, set-методы и совместимость

Kotlin и Java по-разному работают с переменными уровня класса. Java использует поля и обычно предоставляет доступ к ним через методы доступа и изменения (аксессоры и мутаторы). Свойства в Kotlin, как вы уже видели, ограничивают доступ к полям и автоматически предлагают аксессоры и мутаторы.

В прошлом разделе вы добавили публичное поле `hitPoints` в `Jhava`. Этот пример продемонстрировал отображение типов, но он нарушает принципы инкапсуляции, — не самое лучшее решение. В Java к полям надо обращаться или изменять их с помощью геттеров и сеттеров. Get-методы используются для извлечения данных, а set-методы — для их изменения.

Объявите поле `hitPoints` приватным и добавьте get-метод, чтобы `hitPoints` можно было прочитать, но нельзя изменить.

Листинг 23.14. Объявление поля в Java (Jhava.java)

```
public class Jhava {  
  
    public private int hitPoints = 52489112;  
  
    @NotNull  
    public String utterGreeting() {  
        return "BLARGH";  
    }  
  
    @Nullable  
    public String determineFriendshipLevel() {  
        return null;  
    }  
  
    public int getHitPoints() {  
        return hitPoints;  
    }  
}
```

Вернитесь к `Hero.kt`. Код все еще компилируется. В главе 13 мы говорили о том, что Kotlin генерирует геттеры и сеттеры, но синтаксис их использования выглядит так, как если бы вы обращались к переменным напрямую.

Хотя метод `getHitPoints` можно вызвать напрямую, желательно использовать такой синтаксис обращения к свойствам, как сейчас. Это позволяет применить тот же синтаксис, что и для классов Kotlin, с сохранением инкапсуляции. Так как `getHitPoints` имеет префикс `get`, префикс в Kotlin можно опустить и обращаться к `hitPoints` напрямую. Эта особенность стирает барьер между Kotlin и Java.

Это относится и к `set`-методам. На данный момент наш герой и монстр `Jhava` уже хорошо знакомы и хотят развивать отношения. Герой желал бы расширить словарный запас монстра. Переместите приветствие монстра в поле и добавьте для него `get`- и `set`-методы, чтобы у героя появилась возможность изменить приветствие и обучить монстра человеческой речи.

Листинг 23.15. Замена `greeting` в Java (`Jhava.java`)

```
public class Jhava {
    private int hitPoints = 52489112;
    private String greeting = "BLARGH";

    @NotNull
    public String utterGreeting() {
        return "BLARGH" + greeting;
    }

    public String getGreeting() {
        return greeting;
    }

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

В `Hero.kt` измените `adversary.greeting`.

Листинг 23.16. Настройка поля `Jave` из Kotlin (`Hero.kt`)

```
fun main() {
    ...
    val adversaryHitPoints: Int = adversary.hitPoints
    println(adversaryHitPoints.coerceAtMost(100))
    println(adversaryHitPoints.javaClass)

    adversary.greeting = "Hello, Hero."
    println(adversary.utterGreeting())
}
```


Для изменения поля в Java вместо вызова его `set`-метода можно использовать синтаксис присваивания. Преимущества синтаксиса Kotlin доступны вам даже при работе с Java API. Впрочем, следует помнить об одной небольшой проблеме, присущей `set`-методам. Автоматическое преобразование работает для всех методов с префиксом `get`, но этот механизм требует, чтобы у каждого сеттера был соответствующий геттер. Если для поля доступен только сеттер, вам не удастся использовать синтаксис обращения к свойству.

Запустите `Hero.kt` и посмотрите, научил ли герой монстра новым словам.

За пределами класса

Kotlin предлагает разработчикам большую гибкость в выборе формата для кода. Файл Kotlin может одновременно содержать на верхнем уровне и классы, и функции, и переменные. В Java файл может представлять только один класс. Как в таком случае функции верхнего уровня, объявленные в Kotlin, выглядят в Java?

Расширим межвидовое общение, добавив ответ героя. В `Hero.kt` объявите функцию с именем `makeProclamation` за пределами функции `main`.

Листинг 23.17. Объявление функции верхнего уровня в Kotlin (`Hero.kt`)

```
fun main() {  
    ...  
}  
  
fun makeProclamation() = "Greetings, beast!"
```

Нам понадобится способ вызова этой функции из Java, поэтому добавьте метод `main` в `Jhava`.

Листинг 23.18. Объявление метода `main` в Java (`Jhava.java`)

```
public class Jhava {  
  
    private int hitPoints = 52489112;  
    private String greeting = "BLARGH";  
  
    public static void main(String[] args) {  
  
    }  
    ...  
}
```

В этом методе `main` выведите значение, возвращаемое `makeProclamation`, ссылаясь на функцию как на статический метод класса `HeroKt`.

Листинг 23.19. Обращение к функции высшего порядка Kotlin из Java (Jhava.java)

```
public class Jhava {
    ...
    public static void main(String[] args) {
        System.out.println(HeroKt.makeProclamation());
    }
    ...
}
```

Функции верхнего уровня, определенные в Kotlin, представлены в Java как статические методы и вызываются соответствующим образом. `makeProclamation` определяется в `Hero.kt`, поэтому компилятор Kotlin создаст класс с именем `HeroKt` и поместит в него эту функцию как статический метод.

Чтобы взаимодействия между `Hero.kt` и `Jhava.java` выглядели более гладкими, можно изменить имя создаваемого класса, добавив в начало `Hero.kt` аннотацию `@file:JvmName`.

Листинг 23.20. Объявление имени откомпилированного класса с помощью `@file:JvmName` (`Hero.kt`)

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

fun main() {
    ...
}

fun makeProclamation() = "Greetings, beast!"
```

Теперь в `Jhava` можно использовать более понятный вызов функции `makeProclamation`.

Листинг 23.21. Ссылка на переименованную функцию верхнего уровня Kotlin из Java (`Jhava.java`)

```
public class Jhava {
    ...
    public static void main(String[] args) {
        System.out.println(HeroKt.makeProclamation());
    }
    ...
}
```

Запустите `Jhava.java` (при помощи кнопки запуска на полях редактора, потому что сохраненная конфигурация запуска относится к `Hero.kt`), чтобы увидеть ответ вашего героя. Аннотации JVM, такие как `@file:JvmName`, дают возможность прямо определять, какой код Java будет сгенерирован из исходного кода на Kotlin.

Еще одна важная аннотация JVM — `@JvmOverloads`. Параметры по умолчанию языка Kotlin дают возможность использовать более простой подход взамен громоздкого многократного объявления перегруженных версий метода. Что это означает на практике? Следующий пример отвечает на этот вопрос.

Добавьте новую функцию с именем `handOverFood` в `Hero.kt`.

Листинг 23.22. Добавление функции с параметрами по умолчанию (`Hero.kt`)

```
...
fun makeProclamation() = "Greetings, beast!"

fun handOverFood(leftHand: String = "berries", rightHand: String = "beef") {
    println("Mmmm... you hand over some delicious $leftHand and $rightHand.")
}
```

Герой предлагает кое-что из еды в функции `handOverFood`, и вызывающая сторона может выбирать, какую еду он будет держать в правой, а какую в левой руке, либо оставить выбор по умолчанию — мясо и ягоды (`beef` и `berries`). Kotlin позволяет осуществить такой выбор без усложнения кода.

В Java, где параметры по умолчанию отсутствуют, того же эффекта можно добиться только перегрузкой методов:

```
public static void handOverFood(String leftHand, String rightHand) {
    System.out.println("Mmmm... you hand over some delicious " +
        leftHand + " and " + rightHand + ".");
}

public static void handOverFood(String leftHand) {
    handOverFood(leftHand, "beef");
}

public static void handOverFood() {
    handOverFood("berries", "beef");
}
```

Для перегрузки методов в Java требуется гораздо больше кода, чем параметры по умолчанию в Kotlin. Кроме того, один из вариантов вызова функции Kotlin не получится воссоздать в Java — это использование значения по умолчанию для `leftHand` с передачей значения для `rightHand`. Именованные аргументы в Kotlin делают возможным такой вызов: `handOverFood(rightHand= "cookies")` выведет результат `"Mmmm... you hand over some delicious berries and cookies"`. Но Java не поддерживает именованные аргументы и поэтому не различает методы, вызываемые с одинаковым числом параметров (если только эти параметры не относятся к разным типам).

Как вы вскоре увидите, аннотация `@JvmOverloads` обеспечивает автоматическое создание трех соответствующих Java методов, поэтому пользователи Java не сильно ущемлены.

Монстр Jhava терпеть не может вегетарианскую пищу. Ему больше понравились бы пицца или стейк, а не ягоды. Добавьте метод с именем `offerFood` в `Jhava.java`, он дает герою возможность предложить монстру именно такую еду.

Листинг 23.23. Только одна сигнатура метода (`Jhava.java`)

```
public class Jhava {
    ...
    public int getHitPoints() {
        return hitPoints;
    }

    public void offerFood() {
        Hero.handOverFood("pizza");
    }
}
```

Вызов `handOverFood` приводит к ошибке компиляции, потому что в Java не существует концепции параметров по умолчанию. Как следствие, версия `handOverFood` с одним параметром в Java не существует. Чтобы убедиться в этом, загляните в декомпилированный байт-код Java для `handOverFood`:

```
public static final void handOverFood(@NotNull String leftHand,
                                     @NotNull String rightHand) {
    Intrinsics.checkNotNullParameter(leftHand, "leftHand");
    Intrinsics.checkNotNullParameter(rightHand, "rightHand");
    String var2 = "Mmmm... you hand over some delicious " +
        leftHand + " and " + rightHand + '.';
    boolean var3 = false;
    System.out.println(var2);
}
```

В Kotlin имеются способы избежать перегрузки методов, а в Java такая роскошь отсутствует. Аннотация `@JvmOverloads` поможет пользователям вашего API в Java, предоставив перегруженные версии функций Kotlin. Добавьте аннотацию для `handOverFood` в `Hero.kt`.

Листинг 23.24. Добавление `@JvmOverloads` (`Hero.kt`)

```
...
fun makeProclamation() = "Greetings, beast!"

@JvmOverloads
fun handOverFood(leftHand: String = "berries", rightHand: String = "beef") {
    println("Mmmm... you hand over some delicious $leftHand and $rightHand.")
}
```

Вызов `handOverFood` в `Jhava.offerFood` больше не приводит к ошибке, так как теперь происходит вызов версии `handOverFood`, существующей в Java. В этом можно убедиться, если посмотреть на декомпилированный байт-код Java:

```
@JvmOverloads
public static final void handOverFood(@NotNull String leftHand,
                                     @NotNull String rightHand) {
    Intrinsics.checkNotNullParameter(leftHand, "leftHand");
    Intrinsics.checkNotNullParameter(rightHand, "rightHand");
    String var2 = "Mmm... you hand over some delicious " +
        leftHand + " and " + rightHand + '.';
    boolean var3 = false;
    System.out.println(var2);
}

@JvmOverloads
public static final void handOverFood(@NotNull String leftHand) {
    handOverFood$default(leftHand, (String)null, 2, (Object)null);
}

@JvmOverloads
public static final void handOverFood() {
    handOverFood$default((String)null, (String)null, 3, (Object)null);
}
```

Метод с одним параметром определяет первый параметр в функции Kotlin — `leftHand`. При вызове этого метода второй параметр получит значение по умолчанию.

Чтобы проверить, как работает предложение еды монстру, вызовем `offerFood` в `Hero.kt`.

Листинг 23.25. Тестирование `offerFood` (`Hero.kt`)

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

fun main() {
    ...
    adversary.greeting = "Hello, Hero."
    println(adversary.utterGreeting())

    adversary.offerFood()
}
fun makeProclamation() = "Greetings, beast!"
...
```

Запустите `Hero.kt` и убедитесь, что герой передал пищу и стейк.

Если вы проектируете API, который можно использовать из Java, не забывайте применять аннотацию `@JvmOverloads`, чтобы сделать свой API для разработчиков на Java таким же гибким, как и для разработчиков на Kotlin.

Существуют еще две JVM-аннотации для использования в коде на Kotlin, который будет взаимодействовать с кодом Java, и обе они имеют отношение к классам. У `Hero.kt` пока нет реализации класса, поэтому добавим новый класс с именем `Spellbook`. Добавьте в `Spellbook` свойство `spells` — список строк с названиями заклинаний.

Листинг 23.26. Объявление класса `Spellbook` (`Hero.kt`)

```
...
@JvmOverloads
fun handOverFood(leftHand: String = "berries", rightHand: String = "beef") {
    println("Mmmm... you hand over some delicious $leftHand and $rightHand.")
}

class Spellbook {
    val spells = listOf("Magic Ms. L", "Lay on Hans")
}
```

Как вы уже знаете, Java и Kotlin работают с переменными уровня класса совершенно по-разному: Java использует поля с методами чтения и записи, а Kotlin предлагает свойства со вспомогательными полями. Как результат, в Java вы можете обращаться к полям напрямую, а в Kotlin путь к полю лежит через методы доступа, даже если синтаксис обращения выглядит одинаково.

Поэтому ссылка на `spells` — свойство `Spellbook` — в Kotlin такова:

```
val spellbook = Spellbook()
val spells = spellbook.spells
```

А вот обращение к `spells` в Java:

```
Spellbook spellbook = new Spellbook();
List<String> spells = spellbook.getSpells();
```

В Java не удастся обойтись без вызова `getSpells` из-за отсутствия прямого доступа к полю `spells`. Но можно применить аннотацию `@JvmField` к свойству Kotlin, чтобы разрешить пользователям Java использовать поле и избавить их от необходимости вызывать `get`-метод. Добавьте `JvmField` в `spells`, чтобы открыть прямой доступ к `spells` для Jhava.

Листинг 23.27. Применение аннотации `@JvmField` (`Hero.kt`)

```
...
@JvmOverloads
fun handOverFood(leftHand: String = "berries", rightHand: String = "beef") {
    println("Mmmm... you hand over some delicious $leftHand and $rightHand.")
}

class Spellbook {
    @JvmField
    val spells = listOf("Magic Ms. L", "Lay on Hans")
}
```

```
    val spells = listOf("Magic Ms. L", "Lay on Hans")
}
```

Теперь в методе `main` `Jhava.java` вы можете напрямую обратиться к `spells` и вывести каждое заклинание.

Листинг 23.28. Обращение к полю Kotlin напрямую из Java (`Jhava.java`)

```
...
public static void main(String[] args) {
    System.out.println(Hero.makeProclamation());

    System.out.println("Spells:");
    Spellbook spellbook = new Spellbook();
    for (String spell : spellbook.spells) {
        System.out.println(spell);
    }
}
...
```

Запустите `Jhava.java` и убедитесь, что все заклинания из книги заклинаний выводятся в консоль.

Вы также можете использовать `@JvmField` для статического представления значений в объектах-компаньонах. Вспомните главу 16, в которой мы говорили, что объекты-компаньоны объявляются внутри класса и инициализируются, когда инициализируется вмещающий класс либо при обращении к их свойствам или функциям. Добавим объект-компаньон, содержащий единственное значение `maxSpellCount` в `Spellbook`.

Листинг 23.29. Добавление объекта-компаньона в `Spellbook` (`Hero.kt`)

```
...
class Spellbook {
    @JvmField
    val spells = listOf("Magic Ms. L", "Lay on Hans")

    companion object {
        var maxSpellCount = 10
    }
}
```

Теперь попробуем обратиться к `maxSpellCount` из метода `main` в `Jhava`, используя синтаксис Java для обращения к статическим значениям.

Листинг 23.30. Обращение к статическому значению в Java (`Jhava.java`)

```
public static void main(String[] args) {
    System.out.println(Hero.makeProclamation());

    System.out.println("Spells:");
```

```

    Spellbook spellbook = new Spellbook();
    for (String spell : spellbook.spells) {
        System.out.println(spell);
    }

    System.out.println("Max spell count: " + Spellbook.maxSpellCount);
}
...

```

Код не компилируется. Почему? Потому что для обращения к членам объекта-компаньона из Java необходимо сначала получить сам объект, а затем вызвать метод чтения свойства:

```

System.out.println("Max spell count: " +
    Spellbook.Companion.getMaxSpellCount());

```

@JvmField сделает это за вас. Добавьте аннотацию @JvmField к maxSpellCount во вспомогательном объекте Spellbook.

Листинг 23.31. Добавление аннотации @JvmField к члену объекта-компаньона (Hero.kt)

```

...
class Spellbook {
    @JvmField
    val spells = listOf("Magic Ms. L", "Lay on Hans")

    companion object {
        @JvmField
        var maxSpellCount = 10
    }
}

```

После добавления аннотации код Jhava.java скомпилируется, и вы сможете обратиться к maxSpellCount как к любому другому полю в Java. Запустите Jhava.kt и убедитесь, что в консоль выводится максимальное количество заклинаний.

Несмотря на то что Kotlin и Java обрабатывают поля по-разному, @JvmField позволяет открыть поля и обеспечить эквивалентный способ доступа к вашим структурам из Java.

Функции, определенные в объектах-компаньонах, имеют похожие проблемы при обращении к ним из Java — к ним надо обращаться через ссылку на объект-компаньон. Аннотация @JvmStatic действует подобно @JvmField, позволяя получить прямой доступ к функциям, объявленным в объекте-компаньоне. Определите

в объекте-компаньоне класса `Spellbook` функцию с именем `getSpellbookGreeting`. `getSpellbookGreeting` возвращает функцию, которая будет вызвана при вызове `getSpellbookGreeting`.

Листинг 23.32. Применение `@JvmStatic` к функции (`Hero.kt`)

```
...
class Spellbook {
    @JvmField
    val spells = listOf("Magic Ms. L", "Lay on Hans")

    companion object {
        @JvmField
        var maxSpellCount = 10

        @JvmStatic
        fun getSpellbookGreeting() = println("I am the Great Grimoire!")
    }
}
```

Теперь добавьте вызов `getSpellbookGreeting` в `Jhava.java`.

Листинг 23.33. Вызов статического метода в Java (`Jhava.java`)

```
...
public static void main(String[] args) {
    System.out.println(Hero.makeProclamation());

    System.out.println("Spells:");
    Spellbook spellbook = new Spellbook();
    for (String spell : spellbook.spells) {
        System.out.println(spell);
    }

    System.out.println("Max spell count: " + Spellbook.maxSpellCount);

    Spellbook.getSpellbookGreeting();
}
...
```

Запустите `Jhava.java` и убедитесь, что в консоли выводится приветствие книги заклинаний.

Несмотря на отсутствие статических методов в Kotlin, многие его конструкции компилируются в статические переменные и методы. Применение аннотации `@JvmStatic` позволит вам точнее определять, как разработчики Java должны взаимодействовать с вашим кодом.

Исключения и совместимость

Герой научил монстра Jhava своему языку, и монстр теперь хочет пожать ему руку в знак дружбы... а может, и нет. Добавим метод `extendHandInFriendship` в `Jhava.java`.

Листинг 23.34. Выдача исключения в Java (Jhava.java)

```
public class Jhava {  
    ...  
    public void offerFood() {  
        Hero.handOverFood("pizza");  
    }  
  
    public void extendHandInFriendship() throws Exception {  
        throw new Exception();  
    }  
}
```

Вызовем этот метод в `Hero.kt`.

Листинг 23.35. Вызов метода, порождающего исключение (Hero.kt)

```
@file:JvmName("Hero")  
  
fun main() {  
    ...  
    adversary.offerFood()  
  
    adversary.extendHandInFriendship()  
}  
...
```

Запустите этот код, и вы увидите, как во время выполнения выдается исключение. Ведь доверять монстру — не очень разумно.

Вспомните, что в Kotlin все исключения — непроверяемые. Вызывая `extendHandInFriendship`, вы вызвали метод, выдающий исключение. В этом случае вы точно знали, что такое может случиться. В других случаях это не всегда очевидно. Постарайтесь хорошенько разобраться с Java API, с которым вы взаимодействуете из Kotlin.

Упакуйте вызов `extendHandInFriendship` в блок `try/catch`, чтобы воспрепятствовать коварным проискам монстра.

Листинг 23.36. Обработка исключения в try/catch (Hero.kt)

```
@file:JvmName("Hero")  
  
fun main() {  
    ...  
    adversary.offerFood()  
}
```

```

    try {
        adversary.extendHandInFriendship()
    } catch (e: Exception) {
        println("Begone, foul beast!")
    }
}
...

```

Запустите `Hero.kt`, чтобы увидеть, увернулся ли герой от подлой атаки монстра.

Вызов функции Kotlin из Java требует дополнительных знаний о том, как обрабатываются исключения. Все исключения в Kotlin, как мы уже говорили, непроверяемые. Но в Java это не так — там исключения проверяемые, и они должны обрабатываться, чтобы устранить риск сбоя. Как это влияет на вызов функций Kotlin из Java?

Чтобы это узнать, добавьте в `Hero.kt` функцию `acceptApology`. Настало время расплаты для монстра.

Листинг 23.37. Выдача непроверяемого исключения (Hero.kt)

```

...
@JvmOverloads
fun handOverFood(leftHand: String = "berries", rightHand: String = "beef") {
    println("Mmmm... you hand over some delicious $leftHand and $rightHand.")
}

fun acceptApology() {
    throw IOException()
}

class Spellbook {
    ...
}

```

(Не забудьте импортировать `java.io.IOException`.)

Теперь вызовите `acceptApology` из `Jhava.java`.

Листинг 23.38. Выдача исключения в Java (Jhava.java)

```

public class Jhava {
    ...
    public void apologize() {
        try {
            Hero.acceptApology();
        } catch (IOException e) {
            System.out.println("Caught!");
        }
    }
}

```

Монстр `Jhava` достаточно умен, чтобы заподозрить героя в обмане, и упаковывает вызов `acceptApology` в блок `try/catch`. Но компилятор Java предупреждает, что исключение `IOException` никогда не выдается в блоке `try`, то есть в `acceptApology`. Почему так? Ведь `acceptApology` явно выдает исключение.

Чтобы понять причину, необходимо рассмотреть декомпилированный байт-код Java:

```
public static final void acceptApology() {  
    throw (Throwable)(new IOException());  
}
```

Как видите, `IOException` точно выдается в этой функции, но в ее сигнатуре нет ничего, что свидетельствовало бы о необходимости проверять `IOException`. Именно поэтому компилятор Java уверенно заявляет, что `acceptApology` не выбрасывает `IOException` при вызове из Java — он просто не знает об этом.

К счастью, существует аннотация `@Throws`, которая решает эту проблему. Когда вы используете `@Throws`, вы включаете информацию об исключении, которое выбрасывает функция. Добавьте аннотацию `@Throws` в `acceptApology`, чтобы добавить в байт-код Java эту важную информацию.

Листинг 23.39. Использование аннотации `@Throws` (Hero.kt)

```
...  
@Throws(IOException::class)  
fun acceptApology() {  
    throw IOException()  
}  
  
class Spellbook {  
    ...  
}
```

Теперь взгляните на получившийся байт-код Java:

```
public static final void acceptApology() throws IOException {  
    throw (Throwable)(new IOException());  
}
```

Аннотация `@Throws` добавляет ключевое слово `throws` в Java-версию `acceptApology`. Вернитесь в `Jhava.java` и убедитесь, что компилятор Java теперь удовлетворен, так как понял, что `acceptApology` выбрасывает исключение `IOException`, которое необходимо проверить.

Аннотация `@Throws` сглаживает идеологическую разницу между Java и Kotlin в отношении проверки исключений. Если вы пишете Kotlin API, который может вызываться из Java, используйте эту аннотацию, чтобы потребитель API мог корректно обработать любое выданное исключение.

Функциональные типы в Java

Функциональные типы и лямбда-выражения — это новшество языка программирования Kotlin, представляющее рациональный способ взаимодействия между компонентами. Их компактный синтаксис возможен благодаря оператору `->`, но этот синтаксис недоступен в версиях до Java 8.

Как же выглядят эти функциональные типы при вызове из Java? Ответ кажется обманчиво простым: в Java функциональные типы представлены интерфейсами с именами `FunctionN`, где `N` — количество принимаемых аргументов.

Чтобы убедиться в этом, добавьте в `Hero.kt` функциональный тип с именем `translator`: он получает строку, преобразует символы в строчные, первую букву делает прописной и выводит результат.

Листинг 23.40. Определение функционального типа `translator` (`Hero.kt`)

```
fun main() {  
    ...  
}  
  
val translator = { utterance: String ->  
    println(utterance.lowercase().replaceFirstChar { it.uppercase() })  
}  
  
fun makeProclamation() = "Greetings, beast!"
```

`translator` определяется так же, как другие функциональные типы из главы 8. Он имеет тип `(String)-> Unit`. Чтобы увидеть, как этот тип будет выглядеть в Java, сохраните экземпляр `translator` в переменную в `Jhava`.

Листинг 23.41. Сохранение функционального типа в переменной Java (`Jhava.java`)

```
public class Jhava {  
    ...  
    public static void main(String[] args) {  
        ...  
        Spellbook.getSpellbookGreeting();  
  
        Function1<String, Unit> translator = Hero.getTranslator();  
    }  
}
```

(Не забудьте импортировать тип `Kotlin.Unit`; убедитесь, что выбрали его из стандартной библиотеки Kotlin. Также необходимо импортировать `kotlin.jvm.functions.Function1`.)

Эта функция имеет тип `Function1<String, Unit>`. `Function1` — это базовый тип, потому что `translator` имеет только один параметр. `String` и `Unit` используются как параметры типов потому, что `translator` получает параметр типа `String` и возвращает тип `Unit` из Kotlin.

Существуют 23 интерфейса `Function`, начиная с `Function0` и заканчивая `Function22`. Каждый из них содержит единственную функцию `invoke`. `invoke` служит для вызова функционального типа, поэтому каждый раз, когда вам потребуется вызвать функциональный тип, вы используете для этого `invoke`. Вызовите `translator` в `Jhava`.

Листинг 23.42. Вызов функционального типа в Java (`Jhava.java`)

```
public class Jhava {  
    ...  
    public static void main(String[] args) {  
        ...  
        Function1<String, Unit> translator = Hero.getTranslator();  
        translator.invoke("TRUCE");  
    }  
}
```

Запустите `Jhava.kt` и убедитесь, что `Truce` выводится в консоль.

Функциональные типы полезны в Kotlin, но следует помнить, как они представлены в Java. Компактный, динамичный синтаксис Kotlin, который вы наверняка оценили, превращается в громоздкую конструкцию в Java. Если ваш код доступен для классов Java (например, как часть API), то лучше отказаться от функциональных типов. Но если вас устраивает более объемный синтаксис, то просто знайте, что функциональные типы Kotlin доступны в Java.

Совместимость Java и Kotlin является прочной основой развития Kotlin. Она дает возможность использовать из Kotlin существующие фреймворки, такие как Android и Spring, и взаимодействовать со старыми кодовыми базами на Java, что позволит вам постепенно внедрить Kotlin в своих проектах.

К счастью, интеграция Kotlin и Java практически не вызывает затруднений (несколько исключений все-таки есть). Умение писать код на Kotlin, удобный для Java, и наоборот, окупится с лихвой, если вы планируете использовать Kotlin/JVM.

В следующей главе вы узнаете о мультиплатформенной разработке — методологии, позволяющей написать код однократно, а затем использовать его совместно в приложениях, работающих на разных платформах.

24. Знакомство с Kotlin Multiplatform

В этой книге написанный вами код на Kotlin выполнялся в JVM. Исполнительная среда Android строится на базе JVM, поэтому ваш код будет также выполняться на смартфонах и планшетах Android.

Kotlin/JVM — не единственный вариант. Мы уже несколько раз упоминали о том, что Kotlin является мультиплатформенным языком. Помимо поддержки JVM, код Kotlin также работает на других платформах, таких как macOS (без выполнения внутри JVM), iOS и веб-платформах (через JavaScript).

Если вы относитесь к числу опытных разработчиков, то при упоминании кросс-платформенных фреймворков вы обычно представляете себе такие инструменты, как React Native, Flutter и Xamarin. У всех них есть нечто общее: одна кодовая база должна компилироваться и непосредственно выполняться на каждой платформе, поддерживаемой вашей программой, вплоть до пользовательского интерфейса. Теоретически замечательно, но на практике ситуация с кросс-платформенными решениями выглядит не столь идиллически.

Каждый из этих инструментов требует использования собственных UI-фреймворков и API. Если вы уже владеете каким-нибудь нативным фреймворком, вам придется изучить совершенно новый набор API, прежде чем вы сможете писать кросс-платформенный код с одним из этих инструментов. И вы зависите от специалистов, занимающихся сопровождением фреймворка, которые добавляют поддержку новых возможностей для использования в нативных приложениях. Если кросс-платформенные средства, выбранные вами, не реализуют те или иные возможности, необходимые для построения приложения, вам придется ожидать официальной поддержки или же самостоятельно заняться интеграцией с платформенными API.

Кроме того, кросс-платформенные приложения часто реализуют некоторую функциональность только на уровне конкретной платформы. Это означает, что если вы, например, собираетесь создать приложения под iOS и Android, нельзя исключать, что в итоге вам придется написать код для трех платформ: кросс-платформенного фреймворка, функциональности iOS и функциональности Android. Таким образом,

вам придется изучать не один кросс-платформенный инструментарий или две нативные платформы, а три разных фреймворка!

В основе Kotlin Multiplatform лежит другая идея — возможность построить один проект Kotlin, компилируемый для разных платформ.

В следующих трех главах мы исследуем средства мультиплатформенной разработки в Kotlin. Вначале создадим новый мультиплатформенный проект, который будет компилироваться для разных целевых платформ. Затем напишем код для Kotlin/Native и Kotlin/JS и разберем, чем он отличается от кода для Kotlin/JVM.

Что такое Kotlin Multiplatform?

Kotlin Multiplatform не пытается заменять существующий фреймворк. Он предоставляет возможность совместно использовать код Kotlin на разных платформах. Кодовые базы, использующие Kotlin Multiplatform, обычно содержат несколько наборов исходного кода (то есть групп кода, которые являются частью одного проекта).

Наш новый проект будет содержать четыре набора исходного кода:

- общий код;
- JVM;
- macOS;
- JavaScript.

Большая часть кода должна находиться в наборе общего кода. Когда вы компилируете код для конкретной целевой платформы, компилятор объединяет общий код с кодом для этой платформы. Таким образом, например, при компиляции для macOS вы получаете общий код и код для macOS. Отношения между наборами исходного кода и выходными двоичными файлами показаны на рис. 24.1.

В главе 1 мы упоминали, что многие Kotlin API считаются общими API и работают на всех целевых платформах, поддерживаемых Kotlin. В справочной документации API из стандартной библиотеки Kotlin перечислены платформы, поддерживаемые каждым конкретным API. С этими встроенными функциями и классами вы можете ознакомиться на kotlinlang.org/api/latest/jvm/stdlib/.

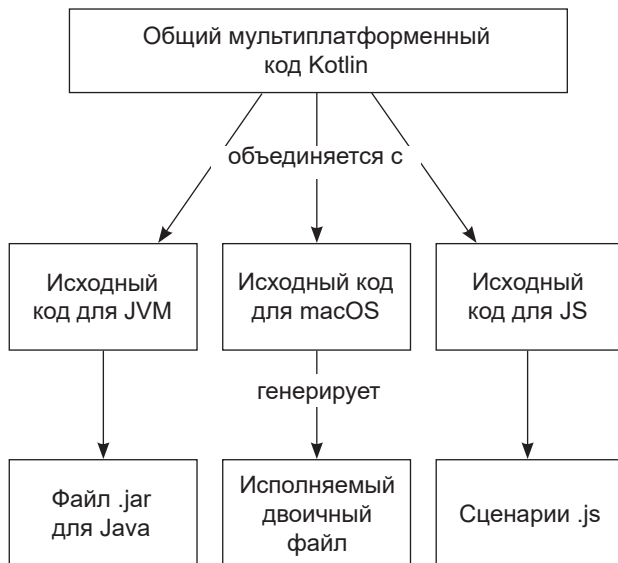


Рис. 24.1. Компиляция наборов исходного кода в двоичную форму

Планирование мультиплатформенного проекта

Первый шаг построения мультиплатформенного приложения — планирование кода, который должен быть общим. При использовании Kotlin Multiplatform мы рекомендуем сосредоточиться на написании общего кода, относящегося к бизнес-логике, то есть логике, неотъемлемо связанной с вашим приложением и задачами предметной области. К этой категории может относиться парсинг данных, сложные алгоритмы и другие виды логики, позволяющие вашему приложению решить поставленную задачу.

И хотя технически возможно полностью определять пользовательские интерфейсы в коде Kotlin Multiplatform, мы не рекомендуем так поступать. Разные платформы, для которых вы пишете код, с большой вероятностью будут использовать разные UI-фреймворки, что усложняет совместное использование UI-компонентов. Попытка строить пользовательские интерфейсы в Kotlin Multiplatform, скорее всего, кончится тем, что вы будете строить их на Kotlin вместо того, чтобы *совместно* использовать UI-код.

Мы рекомендуем объявлять пользовательские интерфейсы на нативном языке платформы. В этом случае вы сможете воспользоваться всем официальным UI-инструментарием, который будет работать намного эффективнее Kotlin

и избавит вас от многих проблем. Очень кстати, что Kotlin является официальным языком Android; это означает, что вы сможете уверенно создавать Android-UI.

Если вы хотите пойти по пути совместного использования бизнес-логики только в UI-приложениях, мы рекомендуем построить *библиотеку* Kotlin Multiplatform вместо *приложения*. Такая библиотека представляет собой заранее откомпилированный двоичный файл со всеми API, определенными в вашем проекте Kotlin Multiplatform. Она не может выполняться сама по себе и не содержит функции `main`.

Далее вашу библиотеку можно импортировать в проект для соответствующей платформы при помощи официального инструментария. IntelliJ содержит ряд шаблонов, которые помогут вам это осуществить, но они в основном ориентированы на совместное использование мобильного кода.

Совместное использование кода в мобильных приложениях мы более подробно рассмотрим в главе 25.

Здесь же мы построим простое приложение, а не библиотеку. Это более подходящий способ, потому что мы не будем создавать сложный UI; наше приложение будет выполняться в терминале (или на простой веб-странице в случае кода JavaScript). Однако принципы, о которых мы расскажем, применимы как к мультиплатформенным приложениям, так и к библиотекам.

Первый мультиплатформенный проект

Итак, после долгого перелета наш герой приземлился на тропическом острове. Теперь можно расслабиться и насладиться заслуженным отдыхом. Однако возникла проблема, которую никто не предвидел: здесь не принимают валюту, которая есть у героя, — золотые монеты!

Необходимо срочно конвертировать золото в дублоны.

Курс обмена постоянно и непредсказуемо меняется. Для управления обменом валюты мы построим мультиплатформенное приложение, которое будет выводить текущий курс, запрашивать у пользователя нужное количество дублонов и возвращать сумму в золотых монетах.

Создайте новый проект при помощи мастера **New Project**, как это делалось ранее. Но на этот раз в качестве шаблона выберите вариант **Application** из раздела **Multiplatform**, как показано на рис. 24.2. Как обычно, выберите в группе **Build System** вариант **Gradle Groovy** и нужную версию **Project JDK**. Присвойте новому проекту имя **Doubloons4Gold**.

Во втором окне мастера убедитесь, что в списках **Test framework** и **Template** выбран вариант **None**, как показано на рис. 24.3. Также обратите внимание на раздел в левой части окна. Здесь можно определить целевые платформы для мультиплатформенного приложения и настроить иерархию проекта. В папке **mainModule** должен содержаться один вариант **Common**, в котором будет сохраняться общий код.

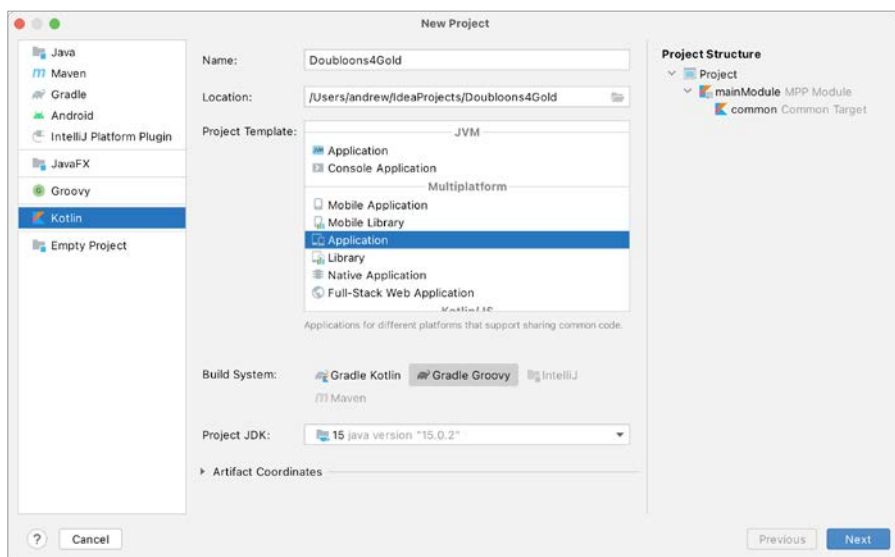


Рис. 24.2. Создание проекта Kotlin Multiplatform

Пока оставьте эти параметры без изменений. Другие целевые платформы мы определим позднее.

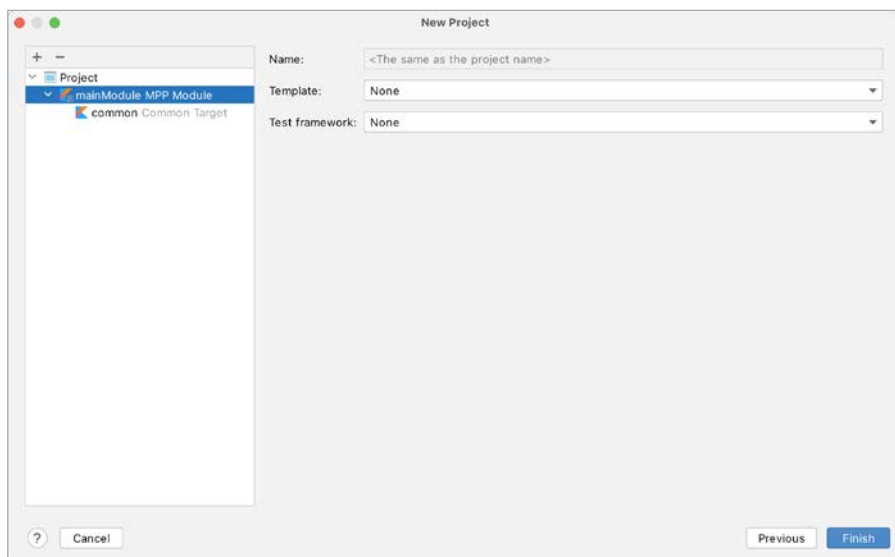


Рис. 24.3. Выбор шаблона проекта

Щелкните на кнопке **Finish**, чтобы подтвердить выбор и создать новый проект. IntelliJ выполняет начальное построение проекта, и вы с ходу получаете ошибку компиляции:

```
Please initialize at least one Kotlin target in 'Doubloons4Gold (:)'.
```

Как следует из сообщения, необходимо определить целевую платформу, для которой должно компилироваться ваше приложение.

Определение целевой платформы Kotlin/JVM

Добавим в проект поддержку целевой платформы JVM. Это поможет вам освоить Kotlin Multiplatform в процессе доработки концепций — этот способ вам уже знаком.

Конечно, если в своем проекте вы хотите ориентироваться только на Kotlin/JVM, лучше создать проект Kotlin/JVM, как мы делали ранее, — без использования шаблона Multiplatform. В следующих двух главах мы определим цели Kotlin/Native и Kotlin/JS — тогда приложение Doubloons4Gold станет действительно мультиплатформенным.

Чтобы добавить новую целевую платформу в код, необходимо внести изменения в файл **build.gradle** проекта. Как мы говорили в главе 20, этот файл содержит информацию о том, как строить проект. Ранее вы редактировали этот файл для объявления зависимостей проекта. Теперь мы будем редактировать его, чтобы передать плагину Kotlin Multiplatform информацию о тех платформах, которые мы намерены поддерживать.

Откройте файл **build.gradle**. Он должен выглядеть примерно так:

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.5.21'
}

group = 'com.bignerdranch'
version = '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

kotlin {
    sourceSets {
        commonMain {
        }
        commonTest {
            dependencies {

```

```

        implementation kotlin('test')
    }
}
}
}

```

Чтобы указать, что вы собираетесь поддерживать конкретную платформу, добавьте в блок `kotlin` соответствующую запись для JVM.

Листинг 24.1. Поддержка JVM (build.gradle)

```

...
kotlin {
    jvm()
    sourceSets {
        ...
    }
}
}

```

Существует много целевых платформ для проекта Kotlin Multiplatform. В табл. 24.1 перечислены наиболее часто используемые.

Таблица 24.1. Часто используемые целевые платформы Kotlin Multiplatform

Платформа	Определение в build.gradle	Описание
JVM	<code>jvm()</code>	Для всех систем, на которых работает Java, и устройств Android
Android	<code>android()</code>	Для приложений Android, выполняемых под управлением исполнительной среды Android в Java
iOS	<code>iosArm64()</code>	Для 64-разрядных устройств ARM iOS, включая iPhone 5s и далее
Эмулятор iOS	<code>iosX64()</code>	Используется эмулятором iOS при выполнении на Mac
Компьютеры macOS	<code>macosX64()</code>	Для компьютеров macOS на базе Intel ¹
JavaScript	<code>js()</code>	Компилирует код Kotlin в JavaScript, для Node.js или для браузеров

¹ Машины с Apple Silicon также могут выполнять эти двоичные файлы под управлением Rosetta 2, но прямая компиляция для Mac с процессором, разработанным Apple, на момент написания книги не поддерживалась.

В следующих двух главах мы будем использовать целевые платформы macOS и JavaScript. Если вы захотите больше узнать о других целевых платформах или об их настройке, обратитесь к документации на kotlinlang.org/docs/mpp-dsl-reference.html#targets.

Каждый раз, когда вы редактируете файлы сборки Gradle, изменения вступят в силу только после синхронизации с IntelliJ. Как и в главе 20, щелкните на кнопке **Gradle sync** , появляющейся во всплывающей подсказке в правом верхнем углу редактора. Ошибка, упомянутая выше, должна исчезнуть.

Итак, вы объявили, что код будет компилироваться для JVM. Теперь можно создать папку для кода, относящегося к Java-разновидности вашего приложения. Щелкните правой кнопкой мыши на папке `src` и выберите команду **New ▸ Directory**. IntelliJ предлагает часто используемые имена папок. Выберите в меню вариант `jvmMain/kotlin` (или введите его вручную и нажмите Return).

Поздравляем — ваша мультиплатформенная программа теперь компилируется для JVM. Остается лишь построить конвертер валют.

Определение общего кода

Основная логика `Doubloons4Gold` определяется в каталоге `commonMain`. Это позволит вам однократно написать части приложения, специфические для вашей задачи (а не платформы), включая курс обмена, подсказки для пользователя и запросы на ввод. Затем эти реализации вы сможете использовать на всех платформах без изменений. (Более того, если вы случайно попытаетесь использовать платформенно-зависимый API в этом каталоге, среда IntelliJ сообщит об ошибке.)

Начнем с создания в папке `src/commonMain/kotlin` нового файла с именем `Converter.kt`. Создайте функцию `convertCurrency` для выполнения конвертации.

Листинг 24.2. Конвертация валют (`commonMain/kotlin/Converter.kt`)

```
val pricePerDoubloon = Random.nextDouble(0.75, 1.5)

fun convertCurrency() {
    println("The current exchange rate is $pricePerDoubloon per doubloon")

    println("How many doubloons do you want?")
    val numberOfDoubloons = readLine()?.toDoubleOrNull()

    if (numberOfDoubloons == null) {
        println("Sorry, I don't know how many doubloons that is.")
    } else {
        val cost = pricePerDoubloon * numberOfDoubloons
        println("$numberOfDoubloons doubloons will cost you $cost")
    }
}
```

В программу нужно импортировать `kotlin.random.Random`.

Этот код задает базовую логику вашего приложения. Он будет совместно использоваться на всех платформах, поддерживаемых `Doubloons4Gold`. Но прежде чем этот код заработает, необходимо добавить точку входа, из которой будет вызываться новая функция `convertCurrency`.

Хотя каждая из точек входа `Doubloons4Gold` вызывает `convertCurrency`, мы определим отдельную точку входа для каждой целевой платформы. Создание разных точек входа на уровне приложения необязательно, но оно упростит запуск приложения Kotlin/JVM из IntelliJ. Кроме того, это обеспечит необходимую гибкость при запуске версии для конкретной платформы, если потребуется добавить дополнительную логику инициализации только для одной из платформ.

Создайте в `jvmMain/kotlin` новый файл с именем `Main.kt` и добавьте функцию `main`. Она выводит сообщение о том, что `Doubloons4Gold` выполняется в JVM (это упростит отслеживание новых целевых платформ по мере их определения), и вызывает `convertCurrency`.

Листинг 24.3. Определение точки входа для JVM (`jvmMain/kotlin/Main.kt`)

```
fun main() {  
    println("Hello from Kotlin/JVM!")  
    convertCurrency()  
}
```

На полях рядом с определением функции `main` должен появиться значок запуска. Запустите функцию `main` и введите какое-нибудь значение, чтобы убедиться, что конвертер работает. Вывод должен выглядеть примерно так:

```
Hello from Kotlin/JVM!  
The current exchange rate is 1.380843418388969 per doubloon  
How many doubloons do you want?  
50  
50.0 doubloons will cost you 69.04217091944845
```

expect и actual

Пока все неплохо. Но станет еще лучше, если в `Doubloons4Gold` будет реализовано форматирование денежных сумм, чтобы вывод выглядел примерно так:

```
Hello from Kotlin/JVM!  
The current exchange rate is $1.38 per doubloon  
How many doubloons do you want?  
50  
50.0 doubloons will cost you $69.04
```

Это форматирование должно применяться на всех платформах, поддерживаемых приложением, поэтому его следует определить в общей кодовой базе. Создайте

в `commonMain/kotlin` новый файл с именем `Currency.kt` и задайте функцию-расширение для `Double` с именем `formatAsCurrency`.

Как следует из имени, эта функция преобразует `double` в строку, отформатированную для вывода денежных единиц пользователя. А пока оставьте реализацию как задачу `TODO`. Вскоре мы объясним, как ее реализовать.

Листинг 24.4. Форматирование валют (`commonMain/kotlin/Currency.kt`)

```
fun Double.formatAsCurrency(): String = TODO("Not implemented")
```

После того, как вы подготовите заглушку для функции `formatAsCurrency`, обновите `convertCurrency` для форматирования строк с денежными суммами. (Текст разделен на две строки, чтобы он поместился на печатной странице; у вас он должен занимать одну строку.)

Листинг 24.5. Использование отформатированных денежных сумм (`commonMain/kotlin/Converter.kt`)

```
val pricePerDoubloon = Random.nextDouble(0.75, 1.5)

fun convertCurrency() {
    println("The current exchange rate is
           ${pricePerDoubloon.formatAsCurrency()} per doubloon")

    println("How many doubloons do you want?")
    val numberOfDoubloons = readLine()?.toDoubleOrNull()

    if (numberOfDoubloons == null) {
        println("Sorry, I don't know how many doubloons that is.")
    } else {
        val cost = pricePerDoubloon * numberOfDoubloons
        println("$numberOfDoubloons doubloons will cost you
               ${cost.formatAsCurrency()}")
    }
}
```

Теперь вернемся к функции `formatAsCurrency`. В главе 5 упоминалось, что в Java существует класс `NumberFormat`, который делает именно то, что нам нужно: он форматирует число по заданному шаблону, например, как денежную сумму. В других проектах вам были доступны все API языка Java, так что реализация функции могла бы выглядеть так:

```
fun Double.formatAsCurrency(): String =
    NumberFormat.getCurrencyInstance().format(this)
```

Но хотя ваша программа в настоящее время предназначена только для JVM, общий код может использовать только API, доступные для целевых платформ, для которых он будет компилироваться. `Doubloons4Gold` также компилируется

для macOS и JavaScript, и ни на одной из этих платформ нет класса `NumberFormat`, соответствующего API языка Java. А значит, общий код не использует API, специфические для Java.

С другой стороны, общий код зависит от функции `formatAsCurrency`, так что перемещение ее определения в набор `jvmMain` недопустимо. Самостоятельно реализовать логику форматирования — слишком хлопотно (и так поступать не рекомендуется). Что же делать?

Для решения подобных проблем в Kotlin есть два ключевых слова для мультиплатформенных проектов: `expect` и `actual`. В общем коде функции свойства и классы можно пометить модификатором `expect`. При этом реализация опускается по аналогии с тем, как определяются функции для интерфейса (см. главу 17).

Пометьте функцию `formatAsCurrency` модификатором `expect` и удалите ее реализацию.

Листинг 24.6. Объявление функции с модификатором `expect` (`commonMain/kotlin/Currency.kt`)

```
expect fun Double.formatAsCurrency(): String = TODO("Not implemented")
```

Объявляя функцию `expect`, вы сообщаете компилятору, что ее реализация отличается на разных платформах. Компилятор будет ожидать реализацию этой функции для каждой целевой платформы, которую вы определите.

Теперь IntelliJ предупреждает об ошибке в функции `formatAsCurrency`. В сообщении говорится, что ожидаемая функция `'formatAsCurrency'` не имеет фактического объявления в модуле `Doubloons4Gold.jvmMain` для JVM. IntelliJ оповещает вас, что вы объявили функцию с ключевым словом `expect`, но не предоставили ее реализацию для всех платформ, для которых предназначен ваш код (в данном случае только JVM).

Чтобы исправить ошибку компиляции, необходимо определить реализацию `formatAsCurrency` в наборе `jvmMain`. Создайте в `jvmMain/kotlin` новый файл с именем `Currency.kt` и реализуйте преобразование с использованием класса `Java NumberFormat`.

Листинг 24.7. Объявление функции с ключевым словом `actual` (`jvmMain/kotlin/Currency.kt`)

```
import java.text.NumberFormat

actual fun Double.formatAsCurrency(): String =
    NumberFormat.getCurrencyInstance().format(this)
```

Новая функция определяется с ключевым словом `actual`. Оно сообщает компилятору Kotlin, что определение является *фактической* реализацией ожидаемой функции.

Все, что помечено ключевым словом `expect` в общем коде, должно иметь подходящее определение `actual` для всех целевых платформ. На стадии компиляции компилятор Kotlin объединяет общий исходный код с исходным кодом, предназначенным для целевых платформ. Использование `expect` и `actual` определяет API, который будет доступен для всех целевых платформ, но реализация которого не разрешена до окончания компиляции.

При определении функции, класса или свойства с ключевым словом `actual` на полях редактора появляется флаг . Если щелкнуть на этом флаге, вы перейдете к объявлению `expect`, соответствующему фактической реализации. Аналогичным образом флаг  появляется на полях рядом с API, определенным с ключевым словом `expect`. Если щелкнуть на этом флаге, IntelliJ выводит меню со всеми фактическими реализациями. Это позволяет легко перейти к нужной платформенно-зависимой реализации в вашем коде.

(Кстати говоря, хотя `expect`- и `actual`-версии `formatAsCurrency` определяются в файлах с именем `Currency.kt`, имена файлов необязательно совпадают. Тем не менее у разработчиков принято использовать одинаковые имена, чтобы было проще ориентироваться в коде. Определения `expect` и `actual` должны размещаться в одном пакете и иметь одинаковые сигнатуры, чтобы компилятор мог правильно связать два определения.)

Снова запустите `Doublings4Gold`. Результат должен выглядеть так:

```
Hello from Kotlin/JVM!
The current exchange rate is $1.06 per doubloon
How many doublings do you want?
50
50.0 doublings will cost you $53.07
```

Разработчики часто используют `expect` и `actual` так, как показано здесь: создается общий набор API, работающий на всех целевых платформах; тем самым процесс разработки оптимизируется за счет увеличения объема общей логики. Однако вызов платформенного кода — не единственная причина для использования этих ключевых слов. Они также могут применяться для определения разных вариантов поведения на конкретных платформах, даже когда реализацию можно определить в общем наборе исходного кода.

Функция `convertCurrency` использует функции `println` и `readLine` для вывода информации и получения ввода от пользователя. Эти знакомые функции хорошо работают в коде JVM, и они будут также работать в версии приложения для Kotlin/Native, но не для JavaScript.

В JavaScript функции `println` и `readLine` доступны для кода Kotlin, но они используются только как средства разработчика. Эти инструменты не предназначены для пользователей веб-сайта. Вам понадобится другой способ визуализации контента на странице.

Так как для JavaScript нужен другой вариант поведения, вам придется определять поведение для каждой платформы в отдельности. Для этого мы снова используем ключевые слова `expect` и `actual`. Начните с создания в папке `commonMain/kotlin` нового файла с именем `InputOutput.kt`. В файле определите две функции `expect` с именами `output` и `getInput`.

Листинг 24.8. `expect` с платформенно-зависимым поведением (`commonMain/kotlin/InputOutput.kt`)

```
expect fun output(message: String)
```

```
expect fun getInput(prompt: String): String
```

Теперь предоставьте соответствующие фактические реализации для JVM. Создайте второй файл `InputOutput.kt` в `jvmMain/kotlin` и реализуйте те же две функции с `println` и `readLine`.

Листинг 24.9. Поведение, специфическое для Java (`jvmMain/kotlin/InputOutput.kt`)

```
actual fun output(message: String) = println(message)
```

```
actual fun getInput(prompt: String): String {  
    output(prompt)  
    return readLine() ?: ""  
}
```

После того как у вас появятся новые платформенно-зависимые функции `output` и `getInput`, вы сможете использовать их в функции `convertCurrency`. Откройте файл `Converter.kt`. Чтобы быстро заменить все вхождения `println` на `output`, откройте меню поиска/замены IntelliJ клавишами `Command-R` (`Ctrl-R`). В первом поле введите исходный текст `println`. Во втором поле укажите текст замены `output`. Щелкните на кнопке `Replace All`, чтобы произвести замену по всему файлу.

Листинг 24.10. Замена `println` на `output` (`commonMain/kotlin/Converter.kt`)

```
val pricePerDoubloon = Random.nextDouble(0.75, 1.5)  
  
fun convertCurrency() {  
    println output("The current exchange rate is  
        ${pricePerDoubloon.formatAsCurrency()} per doubloon")  
  
    println output("How many doubloons do you want?")  
    val numberOfDoubloons = readLine()?.toDoubleOrNull()  
  
    if (numberOfDoubloons == null) {  
        println output("Sorry, I don't know how many doubloons that is.")  
    } else {  
        val cost = pricePerDoubloon * numberOfDoubloons
```

```

        println output("$numberOfDoubloons doubloons will cost you
                        ${cost.formatAsCurrency()}")
    }
}

```

Закройте меню замены кнопкой X в правом верхнем углу, после чего вручную замените вхождения `readLine` на `getInput`.

Листинг 24.11. Использование `getInput` (`commonMain/kotlin/Converter.kt`)

```

val pricePerDoubloon = Random.nextDouble(0.75, 1.5)

fun convertCurrency() {
    output("The current exchange rate is
           ${pricePerDoubloon.formatAsCurrency()} per doubloon")

    val input = output getInput("How many doubloons do you want?")
    val numberOfDoubloons = readLine() input.toDoubleOrNull()

    if (numberOfDoubloons == null) {
        output("Sorry, I don't know how many doubloons that is.")
    } else {
        val cost = pricePerDoubloon * numberOfDoubloons
        output("$numberOfDoubloons doubloons will cost you
                ${cost.formatAsCurrency()}")
    }
}

```

Внесите те же изменения в функции `main`.

Листинг 24.12. Использование функции `output` (`jvmMain/kotlin/Main.kt`)

```

fun main() {
    println output("Hello from Kotlin/JVM!")
    convertCurrency()
}

```

Снова запустите `Doubloons4Gold`. Вывод не изменится, но теперь у вас появляется необходимая гибкость для настройки поведения программы на уровне конкретных платформ.

В этой главе мы сделали первые шаги для построения мультиплатформенного приложения. Мы объявили общий код и предоставили реализации платформенно-зависимых подробностей вашей программы для Java.

На данный момент дополнительная настройка не приносит особой пользы — того же эффекта удалось бы добиться и без объявления функций `expect` и `actual`, если вы хотите ориентироваться только на JVM. Но в следующих двух главах наши усилия окупятся — `Doubloons4Gold` сможет поддерживать больше платформ без изменения логики в файле `Converter.kt`.

25. Kotlin/Native

После мультиплатформенного обустройства приложение Doubloons4Gold готово к расширению на новые целевые платформы. В этой главе мы добавим поддержку macOS.

Если вы работали именно на Mac для изучения примеров книги, может возникнуть резонный вопрос: «Зачем это нужно, я же и так использовал macOS?» Действительно, так как ваши приложения компилировались для Java, они будут работать везде, где работает Java, включая Mac с установкой Java. Но как насчет Mac *без* установленной Java?

Выбирая macOS в качестве целевой платформы, вы строите приложение, которое способно выполняться в macOS без промежуточной исполнительный среды — Java или любой другой. При этом вы также получаете полный доступ к встроенным фреймворкам и API macOS, благодаря чему можете делать то, что не удалось бы сделать только на уровне Java.

Прежде чем идти дальше, мы должны предупредить: многие платформы, поддерживаемые Kotlin/Native, устанавливают ограничения для хоста (то есть компьютера, который будет использоваться для компиляции вашего кода).

В частности, если вы хотите поддерживать macOS, iOS или любую другую платформу Apple, вам придется использовать компьютер с macOS. Когда вы попытаетесь внести изменения, о которых пойдет речь в этой главе, без использования macOS, вы неизбежно столкнетесь с множеством проблем.

Плагин Kotlin Multiplatform блокирует компиляцию для целевых платформ, не поддерживаемых хостом, и редактирование в IntelliJ будет серьезно ограничено. Любые классы и функции, являющиеся частью платформенных API macOS, останутся невидимыми для IntelliJ, из-за чего автозаполнение, быстрые исправления, средства рефакторинга и т. д. будут работать некорректно или не будут работать вообще.

Если у вас нет доступа к компьютеру с macOS, просто прочитайте эту главу, не внося изменений в проект.

Кроме того, компилятор Kotlin/Native требует установки Xcode, официальной IDE компании Apple. Использовать Xcode необязательно, но среда должна быть установлена, чтобы компилятор Kotlin/Native мог выполнить свою работу. В противном случае при попытке построить проект, предназначенный для одной из платформ Apple, вы получите ошибку компиляции.

Чтобы проверить, установлена ли на вашем компьютере среда Xcode, откройте App Store и выполните поиск Xcode. Соответствующий раздел магазина показан на рис. 25.1. Если вы видите кнопку **Open**, все нормально. Если нет, начните загрузку или обновление Xcode — и будьте терпеливы. Xcode — очень большое приложение, и его загрузка займет много времени.

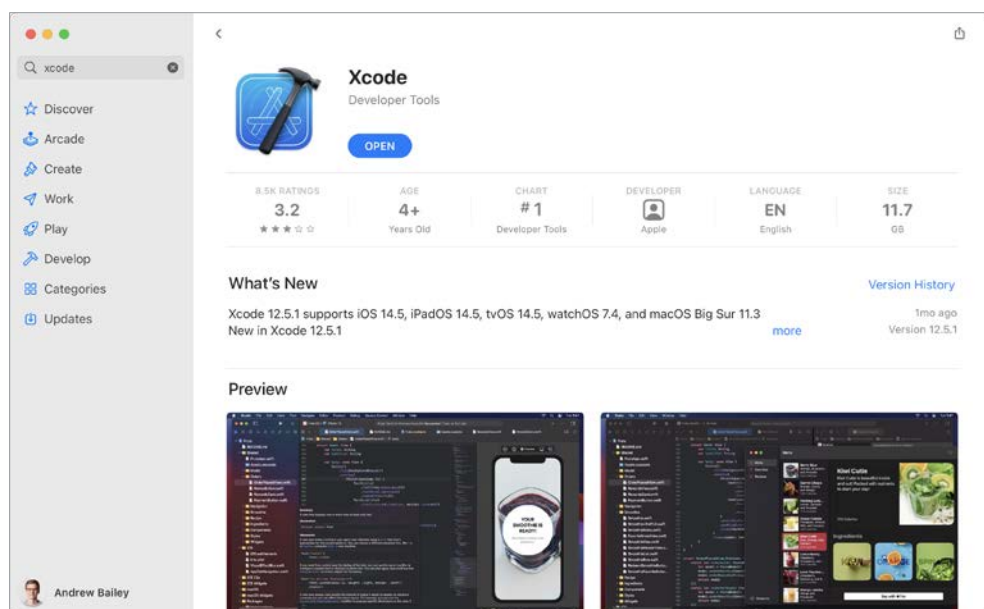


Рис. 25.1. Описание Xcode в App Store

После установки среды Xcode запустите ее. Вам необходимо установить средства разработчика Xcode для режима командной строки, а это происходит при первом запуске Xcode. После подтверждения лицензии Xcode и установки этих компонентов появится заставка, показанная на рис. 25.2.

Если вы увидите такое окно, значит, среда Xcode готова к использованию, а у компилятора Kotlin/Native имеется все необходимое для построения macOS-приложений. Закройте Xcode. Эта среда вам больше не понадобится (если только вы не решите заняться разработкой для Swift или iOS).

И еще одно замечание напоследок: совместимость Kotlin с платформами Apple реализуется через язык Objective-C, а не Swift. Objective-C — более старый язык программирования, берущий начало в далеком 1984 году; именно этот язык заложил основу для программных продуктов Apple. Swift — более современный язык Apple, который получил широкое распространение в сообществе iOS и macOS.

Хотя в настоящее время Kotlin/Native поддерживает только совместимость на уровне Objective-C, Swift также совместим с Objective-C. Это означает, что Kotlin/Native можно использовать вместе со Swift, правда, некоторые преобразования выглядят довольно неуклюже.

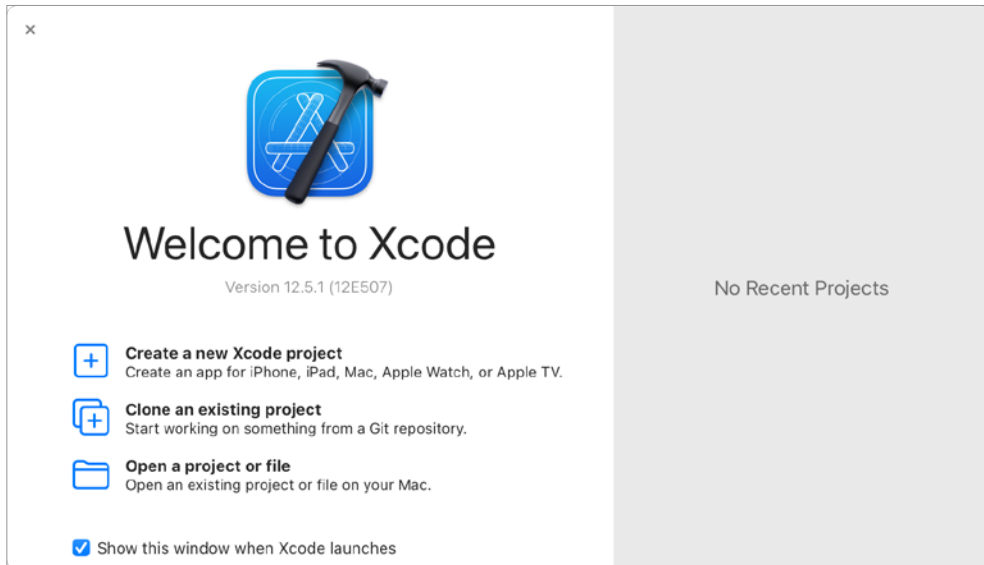


Рис. 25.2. Заставка Xcode

Для этого проекта нам не придется писать код Swift или Objective-C, поэтому сейчас этот факт неважен. Но о нем следует помнить, если вы намерены поддерживать Kotlin/Native в своих Swift-приложениях.

Объявление целевой платформы macOS

После того как мы провели всю необходимую подготовку, можно переходить к добавлению целевой платформы macOS в Doubloons4Gold.

Откройте файл `build.gradle` и объявите новую целевую платформу для macOS. Смысл нового кода мы объясним ниже.

Листинг 25.1. Объявление целевой платформы macOS (`build.gradle`)

```
...
kotlin {
    jvm()
    macOSX64 {
        binaries {
```

```

        executable()
    }
}
sourceSets {
    ...
}
}

```

Каждая платформа, поддерживаемая Kotlin Multiplatform, предоставляет свои параметры компиляции проекта. Для целевых платформ Kotlin/Native, таких как macOS, в этих параметрах можно настроить формат двоичного вывода. Есть несколько вариантов, причем вы можете выбрать несколько сразу.

<code>executable</code>	Создает исполняемый двоичный файл, который можно запускать напрямую.
<code>test</code>	Создает исполняемый двоичный файл, который может использоваться для запуска модульных тестов.
<code>sharedLib,</code> <code>staticLib</code>	Создает двоичный файл для использования вашего кода как библиотеки. Это полезно, если вы хотите применить свой код Kotlin Multiplatform в другом нативном проекте, не на Kotlin. Нативные библиотеки бывают общие и статические. Различия между ними мы не будем рассматривать в книге, но если вы знакомы с нативной разработкой, то вероятно, уже знаете, какая разновидность вам нужна.
<code>framework</code>	Создает фреймворк Objective-C, который можно применять для использования вашего кода как проекта (из Xcode). Этот вариант годится только для платформ, на которых работает Objective-C, включая macOS и iOS; обычно его следует выбирать вместо <code>sharedLib</code> и <code>staticLib</code> , если это возможно.

В файле `build.gradle` мы указали, что хотим создать исполняемый двоичный файл. Это позволит запустить программу напрямую.

Вам нужно указывать двоичные файлы только для целевых платформ Kotlin/Native. JVM и JavaScript не различают библиотеки и исполняемые файлы — в обоих случаях используется один формат вывода. Но в случае Native-платформ это не так, поэтому нужно предоставить соответствующую информацию, чтобы гарантировать корректную компиляцию и использование вашего кода.

После выбора целевой платформы macOS снова выполните синхронизацию Gradle, чтобы среда IntelliJ подключила новейшие изменения. По завершении синхронизации Gradle (что может занять минуту или две) создайте новый каталог для исходного кода `src/macosX64Main/kotlin`. (IntelliJ предлагает его по умолчанию, когда вы открываете меню **New Directory** в `src`.)

Итак, теперь ваш код официально поддерживает целевую платформу macOS и у вас имеется каталог для размещения кода macOS. Все готово к написанию вашего первого кода для macOS.

Написание нативного кода на Kotlin

Если попытаться откомпилировать проект прямо сейчас, компилятор выдаст ошибку для каждой из функций `expect`, потому что вы не предоставили фактические реализации для новой целевой платформы macOS. Также необходимо определить точку входа, чтобы приложение `Doubloons4Gold` нормально функционировало в macOS.

Начнем с реализации функции `formatAsCurrency`. Создайте новый файл Kotlin с именем `Currency.kt` — на этот раз в папке `macosX64Main/kotlin`.

В новом файле реализуем форматирование денежных сумм, которое было настроено для JVM. При этом используем пока незнакомые вам API, поставляемые с macOS. Подробнее о коде мы расскажем ниже.

Листинг 25.2. Нативное форматирование числа (`macosX64Main/kotlin/Currency.kt`)

```
import platform.Foundation.*

actual fun Double.formatAsCurrency(): String {
    val formatter = NSNumberFormatter().apply {
        setNumberStyle(NSNumberFormatterCurrencyStyle)
        setLocale(NSLocale.currentLocale)
    }

    val number = NSNumber(this)
    return formatter.stringFromNumber(number)!!
}
```

Первая строка, `import platform.Foundation.*`, импортирует фреймворк `Foundation` компании Apple. `Foundation` предоставляет API для всех платформ Apple для операций с числами, датами, файлами, форматирования данных, загрузки URL и многого другого. (В `Foundation` API используется префикс `NS`, который несколько раз встречается в этом коротком коде; `NS` происходит от `NeXTSTEP` — предка macOS, появившегося в 1980-х годах.)

Затем создается экземпляр класса `NSNumberFormatter` — аналога Java-класса `NumberFormat` в `Foundation`.

`NSNumberFormatter` передаются два параметра: стиль и локаль (региональные настройки). Эти параметры задаются внутри блока `apply`. Это та же функция `apply`, о которой мы говорили в главе 12. Напомним, что `apply` позволяет вызывать функции для получателя, чтобы настроить его; здесь функция ведет себя так же, хотя она и вызывается для класса, не определенного в Kotlin.

Стиль задается элементом перечисления. Для обращения к стилю денежных величин используется запись `NSNumberFormatterCurrencyStyle`. Чтобы задать локаль, мы получаем региональные настройки пользователя из свойства `currentLocale` класса `NSLocale`, а затем передаем его функции `setLocale`. В этой точке конфигурация `NSNumberFormatter` полностью настроена.

Чтобы преобразовать значение `Double` в отформатированную строку, мы сначала упаковываем его в `NSNumber` — базовый числовой тип в Objective-C. Затем вызываем функцию `stringFromNumber` для выполнения преобразования.

Эта функция возвращает `null`, если значение, содержащееся в `NSNumber`, не может быть отформатировано как денежная сумма (например, если `NSNumber` представляет `Boolean`). Так как мы форматируем `Double`, такая ситуация исключена, поэтому мы используем принудительное преобразование возвращаемого значения `String?` к нужному возвращаемому типу `String`.

Если вам интересно, версия этой функции-расширения на Swift выглядит так:

```
import Foundation

extension Double {
    func formatAsCurrency() -> String {
        let formatter = NumberFormatter()
        formatter.numberStyle = .currency
        formatter.locale = .current

        let number = NSNumber(value: self)
        return formatter.string(from: number)!
    }
}
```

А для самых любознательных — на Objective-C она будет выглядеть так:

```
- (NSString *)formatAsCurrency:(NSNumber *)value {
    NSNumberFormatter *formatter = [[NSNumberFormatter alloc] init];
    formatter.numberStyle = NSNumberFormatterCurrencyStyle;
    formatter.locale = NSLocale.currentLocale;
    return [formatter stringFromNumber:value];
}
```

Есть еще две функции, которым требуется фактическая реализация: `output` и `getInput`. В главе 24 мы упоминали, что `println` и `readLine` относятся к средствам разработчика в JavaScript, и мы объявили функции `output` и `getInput` с `expect`, чтобы отразить это отличие. Но целевая платформа macOS использует те же реализации, что и для Java.

Найдите файл `jvmMain/kotlin/InputOutput.kt` в окне инструментов проекта. Выделите его и нажмите `Command-C` (`Ctrl-C`), чтобы скопировать. Затем выделите каталог `macosX64Main/kotlin` и нажмите `Command-V` (`Ctrl-V`), чтобы вставить файл. В открывшемся диалоговом окне подтвердите, что среда IntelliJ должна вставить копию файла в каталог `macosX64Main/kotlin` (рис. 25.3). Когда все будет готово, щелкните на кнопке `ОК`, чтобы среда IntelliJ создала копию файла.



Рис. 25.3. Копирование файла между наборами исходного кода

IntelliJ открывает скопированный файл в редакторе. Убедитесь, что его содержимое совпадает с реализацией, скопированной из исходного набора JVM:

```
actual fun output(message: String) = println(message)

actual fun getInput(prompt: String): String {
    output(prompt)
    return readLine() ?: ""
}
```

Запуск приложения Kotlin/Native

После определения фактических реализаций для всех функций с `expected` проект `Doubloons4Gold` будет компилироваться. Но вы еще не определили точку входа для целевой платформы `macOS`, так что при попытке запуска программа не знает, какой именно код нужно выполнять, и не запустится.

Чтобы исправить ситуацию, вам понадобится еще одна функция `main` — на этот раз из набора исходного кода `macOSX64`. Создайте в папке `macOSX64Main/kotlin` новый файл `Main.kt` и определите в нем функцию `main`. Чтобы отличать приложение Kotlin/Native от приложения Kotlin/JVM, включите выходное сообщение с объявлением платформы. (При желании скопируйте файл `Main.kt` из `jvmMain` в `macOSX64Main` и внесите соответствующие изменения в выходную строку.)

Листинг 25.3. Определение функции `main` для `macOS` (`macOSX64Main/kotlin/Main.kt`)

```
fun main() {
    output("Hello from Kotlin/Native!")
    convertCurrency()
}
```

Когда вы ранее объявляли новые функции `main`, среда IntelliJ автоматически добавляла рядом с ними значок запуска на полях редактора. Но взгляните на свою

новую функцию: значок запуска не появился. Среда IntelliJ не обнаруживает точки входа в коде, не предназначенном для JVM, поэтому запускать программу на целевой платформе macOS следует несколько по-иному.

Вместо того чтобы запускать код средствами IntelliJ, необходимо действовать напрямую через Gradle.

Откройте инструментальное окно **Gradle**, щелкнув на вкладке  **Gradle** в правом верхнем углу IDE. Откроется панель, изображенная на рис. 25.4.

В этом окне отображается иерархия всех задач, которые может выполнять Gradle, — иначе говоря, все отдельные действия сборки для вашего проекта, о которых Gradle знает. Эти задачи сгруппированы по категориям.

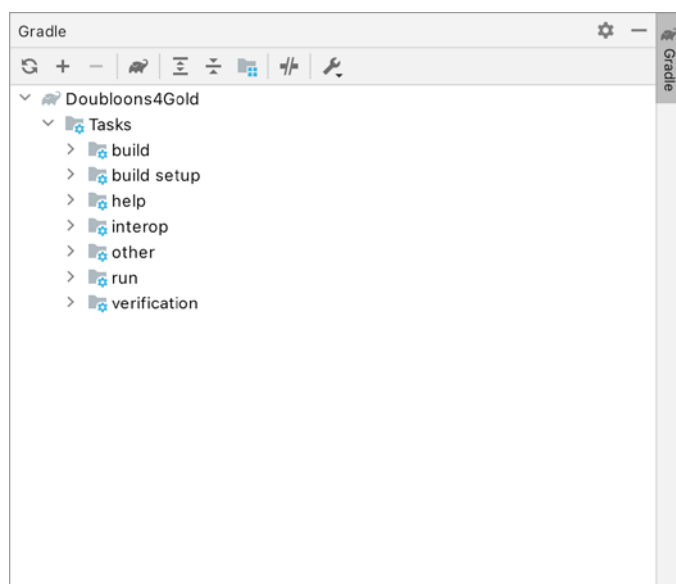


Рис. 25.4. Инструментальное окно Gradle

В инструментальном окне **Gradle** перейдите к группе **Doubloons4Gold/Tasks/run** и сделайте двойной щелчок на **runReleaseExecutableMacosX64**, чтобы построить и запустить код для macOS. Если вы хотите построить свой проект, не выполняя его, используйте задачу с именем **compileKotlinMacosX64** в группе **build**.

Обычно это требуется сделать только один раз для каждой конфигурации запуска в проекте. IntelliJ запоминает конфигурации, и вы можете переключаться между ними при помощи раскрывающегося списка рядом с кнопкой запуска на панели инструментов. После выполнения этой задачи в раскрывающемся списке будет выбран вариант **Doubloons4Gold[runReleaseExecutableMacosX64]** — он сообщает, какая конфигурация будет использована при нажатии кнопки запуска.

Gradle строит и запускает проект macOS. После исполнения вывод должен выглядеть так, как показано ниже. В нижней части выводится информация о сборке, которую мы включили для справки.

```
...
> Task :runReleaseExecutableMacosX64
Hello from Kotlin/Native!
The current exchange rate is $1.14 per doubloon
How many doubloons do you want?
Sorry, I don't know how many doubloons that is.

BUILD SUCCESSFUL in 611ms
3 actionable tasks: 1 executed, 2 up-to-date
2:57:36 PM: Task execution finished 'runReleaseExecutableMacosX64'.
```

Из-за способа выполнения кода в Gradle у вас, вероятно, не будет возможности предоставить входные данные. Это нормально. На следующем этапе мы найдем откомпилированный исполняемый файл на компьютере и запустим его вручную.

Вывод Kotlin/Native

Когда код Kotlin компилируется для macOS, компилятор генерирует файл `kexe` (Kotlin Executable). Этот файл можно запустить из терминала, как любую другую программу. В отличие от приложений, которые мы создавали ранее, такой двоичный файл можно выполнять на компьютерах Mac, на которых не установлена исполнительная среда Java. Это упрощает установку приложения и делает его доступным для более широкого круга пользователей macOS.

После построения проекта Gradle создает двоичный файл приложения в папке `build/bin/macosX64/releaseExecutable`. Найдите эту папку в инструментальном окне проекта. Щелкните правой кнопкой мыши на каталоге `releaseExecutable` и выберите команду `Open In ▸ Terminal`. В нижней части окна IntelliJ откроется панель терминала (рис. 25.5) с текущим каталогом `releaseExecutable`, который вы только что выбрали.



Рис. 25.5. Интегрированный терминал IntelliJ

Чтобы запустить приложение с полноценной интерактивностью, введите следующую команду и нажмите Return:

```
./Doubloons4Gold.kexe
```

Doubloons4Gold запускается в интегрированном терминале IntelliJ, и теперь вы сможете нормально взаимодействовать с приложением. Сообщение `Hello from Kotlin/Native!` указывает, что вы действительно создали и выполнили программу, отличную от JVM-версии. Чтобы убедиться, что программа ведет себя так же, как в JVM-версии, введите значение для конвертации:

```
> ./Doubloons4Gold.kexe
Hello from Kotlin/Native!
The current exchange rate is $1.06 per doubloon
How many doubloons do you want?
20
20.0 doubloons will cost you $21.15
>
```

С последней проверкой приложение macOS завершено. Ненадолго остановитесь и подумайте, что бы вам пришлось делать для поддержки нескольких платформ без Kotlin Multiplatform. Возможно, пришлось бы реализовать функцию `convertCurrency` несколько раз для каждой платформы, и не исключено, что на разных языках для разных платформ.

Если позже вы захотели бы изменить функциональность программы, то потребовалось бы внести изменения в несколько разных проектов, портируя эти изменения в нескольких языках. Сравните с проектом Kotlin Multiplatform, где вы написали базовую логику однократно и реализовали минимум платформенно-зависимой логики.

И хотя настройка этого проекта была несколько сложнее, чем обычно, и потребовалось дублировать сигнатуры функций, в долгосрочной перспективе такой подход сэкономит немало времени и сил по сравнению с трудозатратами на написание кода для каждой отдельной платформы, которую вы захотите поддерживать.

В следующей главе мы продолжим совершенствовать приложение Doubloons4Gold и добавим в него поддержку JavaScript.

Для любознательных: Kotlin Multiplatform Mobile

Kotlin/Native чаще всего используется для приложений iOS. С другой стороны, поскольку Kotlin является официальным языком для Android, он хорошо знаком многим командам мобильной разработки. Все больше разработчиков выбирают Kotlin как для iOS, так и для Android-приложений. Такое использование Kotlin Multiplatform называется Kotlin Multiplatform Mobile, или KMM.

Условие для работы с КММ заключается в создании трех кодовых областей: общего кода на Kotlin, кода Android-приложения (также на Kotlin) и кода iOS-приложения на Swift. Этот подход отличается от используемого в большинстве кросс-платформенных мобильных фреймворков. При применении КММ ваш мультиплатформенный код должен быть библиотекой, содержащей бизнес-логику, то есть он должен реализовать только алгоритмы и платформенно-независимую часть (вместо построения всего пользовательского интерфейса).

Таким образом вы получаете доступ к сильным сторонам двух систем. Ключевая логика программируется только один раз и совместно используется между платформами, обеспечивая согласованное поведение обоих приложений. Если в этой логике встретится ошибка, вы сможете исправить ее на обеих платформах одновременно.

Вы также получаете доступ к нативным инструментариям целевых платформ. Это означает, что вы можете пользоваться преимуществами таких библиотек, как SwiftUI и Jetpack Compose; каждую библиотеку допустимо подключать только при нативной разработке на официальном языке платформы.

Впрочем, у КММ есть свои недостатки — с ними чаще сталкиваются iOS-, чем Android-разработчики. У последних есть преимущество — обращаться как к функциональности Android, так и к общим возможностям знакомого языка и IDE. А вот iOS-разработчикам нужно изучать Kotlin в дополнение к Swift. Кроме того, им придется переключаться между Xcode и IntelliJ, когда та или иная функциональность потребует обращения как к общей библиотеке Kotlin, так и к рабочему пространству iOS.

Инструментарий для Kotlin Multiplatform и Kotlin/Native все еще находится в стадии развития, и у него есть свои недостатки. В частности, взаимодействие между Kotlin и Swift требует промежуточного уровня, который Kotlin не всегда предоставляет в виде, привычном для разработчиков Swift.

Например, приостанавливаемые функции предоставляются коду Swift через функции обратного вызова (callbacks), что усложняет их применение за пределами Kotlin. Если вы планируете использовать сопрограммы, каналы или потоки данных, то скорее всего, вам также придется создавать обертки для упрощения работы с асинхронным кодом в iOS.

При всех своих недостатках КММ при правильном использовании может стать чрезвычайно мощным инструментом, особенно если ваша команда имеет больше опыта в разработке для Android, чем для iOS. Если вас заинтересует идея совместного использования кода, мы рекомендуем опробовать КММ. Но начинать лучше с малого, потому что вы рискуете избыточно усложнить приложение, что перевесит преимущества от совместного использования кода.

Для любознательных: другие нативные платформы

При написании кода Kotlin/Native для macOS и iOS у вас появляется возможность использовать API из Objective-C. Эти API находятся на гораздо более высоком уровне, чем API, с которыми приходится иметь дело при вызове кода, написанного на низкоуровневых языках (таких, как C или C++). Однако Kotlin поддерживает вызов кода низкоуровневых языков, если вам это понадобится.

При работе с низкоуровневым нативным кодом вам почти сразу же потребуется вызывать API с использованием таких средств, как указатели и ручное выделение памяти. Эта тема выходит за рамки книги, но в Kotlin такая возможность имеется. Также существует ряд правил относительно взаимодействия Kotlin и C, включая соответствия типов и функций между языками. Если эта тема вас интересует, посмотритесь к программе `cinterop`.

Windows — пример платформы на базе C с API более низкого уровня, чем те, с которыми вы познакомились в этой главе. Если вам *действительно* интересно, ниже мы приводим реализацию `formatAsCurrency` для Windows. В этом фрагменте показаны многие средства взаимодействия с C.

Если эти концепции вам незнакомы — не страшно. Мы привели их, просто чтобы мотивировать вас на дальнейшее получение знаний, но для освоения языка C или C++ требуется отдельная книга.

```
import kotlinx.cinterop.*
import platform.windows.GetCurrencyFormatEx
import platform.windows.LOCALE_NAME_USER_DEFAULT
import platform.windows.WCHARVar

actual fun Double.formatAsCurrency(): String {
    // Ввод преобразуется в неформатированную строку
    // для использования с GetCurrencyFormatEx
    val numberToFormat = toString()

    return memScoped {
        // Определение локального контекста, выбранного пользователем
        val userLocaleName = LOCALE_NAME_USER_DEFAULT
            ?.reinterpret<UShortVar>()
            ?.toKString()

        // Определение количества символов в отформатированном выводе
        val length = GetCurrencyFormatEx(userLocaleName, 0u, numberToFormat,
            null, null, 0)

        // Выделение памяти для отформатированного вывода
        val output = allocArray<WCHARVar>(length)
```



```
// Форматирование ввода как денежной суммы
// с сохранением результата в массиве output
GetCurrencyFormatEx(userLocaleName, 0u, numberToFormat, null, output,
                    length)

// Вывод из символьного массива преобразуется
// в строку Kotlin и возвращается функцией
output.toKString()
    }
}
```

Подписывайся на самый топовый канал по Kotlin:
<https://t.me/KotlinSenior>

26. Kotlin/JS

Последняя задача в Doubloons4Gold — добавление еще одной целевой платформы, на сей раз для веб. Поддержка веб-браузеров позволяет выполнять код практически где угодно. Существуют разные способы реализации на веб-платформе; мы воспользуемся Kotlin/JS для компиляции программы Kotlin в файлы JavaScript, которые будут запускаться в клиентских браузерах.

Для веб-браузера интерфейс Doubloons4Gold немного изменится. Вариант с использованием `readLine` и `println` уже не подходит, так как эти функции требуют взаимодействия со средствами разработчика в JavaScript. Поэтому мы построим базовый пользовательский интерфейс внутри веб-страницы.

Объявление поддержки Kotlin/JS

Прежде всего следует объявить в коде новую целевую платформу — JavaScript. Для этого мы снова обновим файл `build.gradle`.

Листинг 26.1. Объявление целевой платформы JavaScript (`build.gradle`)

```
...
kotlin {
    jvm()
    macosX64() {
        binaries {
            executable()
        }
    }
    js {
        browser()
        binaries {
            executable()
        }
    }
    sourceSets {
        ...
    }
}
...
```

Единственное новшество в этом фрагменте — вызов функции `browser`. Он сообщает, что вы намерены выполнять JavaScript в браузере. Также можно использовать вызов `nodejs`, чтобы проект компилировался для Node.js — автономной исполнительной среды JavaScript, которую обычно используют для построения веб-серверов.

Проведите синхронизацию Gradle после внесения изменений.

И снова проект не компилируется, потому что функции `expect` не имеют фактических реализаций в наборе исходного кода JavaScript. Нашему коду для JavaScript необходим собственный каталог (как и коду для macOS и JVM). Щелкните правой кнопкой мыши на каталоге `src`, выберите команду `New ► Directory` и дважды щелкните на `jsMain/kotlin`.

При запуске на веб-странице поведение `Doubloons4Gold` заметно меняется. Вместо консольного вывода контент будет динамически вставляться в веб-страницу в процессе выполнения кода. Кроме того, для получения ввода используются запросы браузера (вместо консольного ввода).

Разобьем задачу на более простые этапы и начнем с подстановки заглушек для всех функций `expect`, чтобы программа компилировалась. Пока эти заглушки перенаправляют вывод в консоль разработчика и предоставляют значение, имитирующее ввод от пользователя. Мы вернемся к этому чуть позже.

Начнем с создания нового файла Kotlin с именем `InputOutput.kt` в `jsMain/kotlin`. Предоставим фактические реализации для `output` и `getInput`.

Листинг 26.2. Заглушки для ввода/вывода (`jsMain/kotlin/InputOutput.kt`)

```
actual fun output(message: String) {  
    println(message)  
}  
  
actual fun getInput(prompt: String): String {  
    val input = "10.0"  
    return input  
}
```

Создайте в `jsMain/kotlin` новый файл Kotlin с именем `Currency.kt` и определите заглушку для функции `formatAsCurrency`. Пока она возвращает число в виде строки без дополнительного форматирования.

Листинг 26.3. Заглушка для форматирования денежных сумм (`jsMain/kotlin/Currency.kt`)

```
actual fun Double.formatAsCurrency(): String {  
    return toString()  
}
```

С функциями `actual` проект будет компилироваться. Впрочем, для его запуска понадобится точка входа — на этот раз для кода, который выполняется при загрузке сценария `Doubloons4Gold.js` в браузере.

Создайте в `jsMain/kotlin` файл Kotlin с именем `Main.kt`, из которого вызывается `convertCurrency`.

Листинг 26.4. Определение точки входа JavaScript (`jsMain/kotlin/Main.kt`)

```
fun main() {  
    output("Hello from Kotlin/JS!")  
    convertCurrency()  
}
```

Как и другие функции `main`, написанные ранее, она будет выполняться при загрузке страницы.

Прежде чем мы запустим приложение Kotlin/JS, осталось решить еще одну задачу. Код JavaScript не может выполняться браузером в пустоте — код необходимо разместить внутри веб-страницы. И эту страницу мы сейчас создадим.

Создайте в `jsMain` новый каталог с именем `resources`. (Вы можете щелкнуть правой кнопкой мыши на `src` и выбрать **New ► Directory**, чтобы получить список стандартных каталогов, как это мы делали ранее, или же щелкнуть правой кнопкой мыши на `jsMain` и ввести имя каталога вручную.) Каталог `resources` используется для файлов, необходимых программе, но не содержащих исполняемого кода.

В каталоге `resources` создайте файл `index.html`. (Не используйте шаблон IntelliJ для HTML-файлов; выберите команду **New ► File** и введите полное имя с расширением `.html`.) Этот файл содержит страницу по умолчанию, именно ее будет открывать браузер при запуске `Doubloons4Gold`. Включите в него простую разметку HTML.

Листинг 26.5. Создание лендинга (`jsMain/resources/index.html`)

```
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <title>Doubloons4Gold</title>  
  </head>  
  <body>  
    <h1>Doubloons4Gold</h1>  
    <script src="Doubloons4Gold.js"></script>  
  </body>  
</html>
```

Такая разметка HTML создает простейший лендинг. В теге `<head>` для страницы задается заголовок `Doubloons4Gold`, отображаемый в браузере. В теге `<body>` задается заголовок с тем же именем приложения. Тег `<script>` используется для импорта и выполнения приложения.

Так как тег `<script>` располагается в конце блока `<body>`, приложение будет загружаться и выполняться после того, как завершится загрузка HTML-разметки, определяющей страницу. Это гарантирует, что браузеру хватит времени для подготовки страницы, прежде чем он начнет выполнять какой-либо код. (Другие ресурсы, например графика, могут еще подгружаться, но обычно они не влияют на выполнение кода `Doubloons4Gold`.)

Скрипт `Doubloons4Gold.js` генерируется компилятором Kotlin/JS при построении проекта. Имя файла определяется названием проекта, но при желании его можно изменить. (За подробностями обращайтесь к описанию `outputFileName` в документации Kotlin по адресу kotlinlang.org/docs/home.html.)

Теперь можно запустить версию `Doubloons4Gold` для Kotlin/JS. Для этого вам придется воспользоваться Gradle, как и в случае построения и запуска кода для macOS.

Откройте окно **Gradle**, чтобы просмотреть задачи сборки для вашего проекта. Щелкните на кнопке со стрелкой, чтобы раскрыть категорию **other**, а затем дважды щелкните на задаче **jsRun**. Gradle строит вашу программу и запускает браузер, в котором открывается файл `index.html`.

При запуске браузера вы увидите новую вкладку — `Doubloons4Gold`. Также на странице выводится заголовок с тем же именем, а оставшаяся часть страницы пуста. Вывод направляется в консоль разработчика в IntelliJ.

Чтобы просмотреть вывод в браузере, следует перейти в режим *просмотра кода*. В большинстве браузеров для этого достаточно щелкнуть правой кнопкой мыши в любой точке страницы и выбрать команду меню **Inspect** (Просмотр кода страницы) или **Inspect Element** (Исследовать элемент).

Пользователям Safari сначала придется включить инструментарий разработчика. Для этого выполните команду **Safari ▸ Preferences...** (Настройки...) и перейдите на вкладку **Advanced** (Дополнения). Установите флажок **Show develop menu in menu bar** (Показывать меню «Разработчика» в строке меню) и закройте окно настроек. Теперь вы можете щелкнуть правой кнопкой мыши в окне браузера и выбрать команду **Inspect Element**. Откроется окно со средствами разработчика (рис. 26.1).

С открытым окном инспектора перейдите на вкладку **Console**. Консольный вывод должен включать знакомые сообщения о курсе и итоговой сумме, как показано ниже. (Если консоль пуста, возможно, следует обновить страницу для получения вывода.)

```
Hello from Kotlin/JS!
```

```
The current exchange rate is 1.377781434913309 per doubloon
```

```
10 doubloons will cost you 13.777814349133092
```

```
console.
```

```
kt:78:16
```

```
console.
```

```
kt:78:16
```

```
console.
```

```
kt:78:16
```

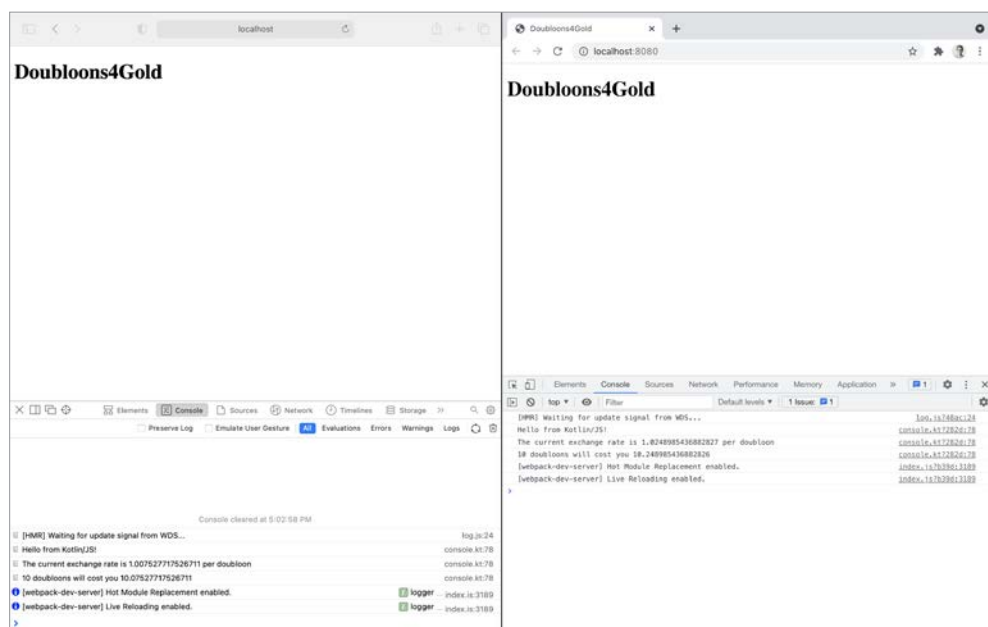


Рис. 26.1. Окно Developer Tools в Chrome и Safari

Вывод `convertCurrency` демонстрирует, что `Doubloons4Gold` действительно выполняется на веб-странице. Возможно, в области вывода будут присутствовать другие сообщения, относящиеся к запуску веб-сервера из Gradle. На все эти сообщения, не относящиеся к делу, можно не обращать внимания.

При запуске `Doubloons4Gold` на вашем компьютере стартует веб-сервер, который так и будет работать, даже если вы закроете вкладку в браузере, пока вы не прикажете ему остановиться. Если оставить его работающим, вероятно, он будет пытаться перезагружать страницу после каждого изменения в коде; это быстро начинает раздражать и мешает работе.

Каждый раз, когда вы запускаете `Doubloons4Gold` и возвращаетесь в IntelliJ, останавливайте программу вручную нажатием кнопки  на панели инструментов IntelliJ.

Взаимодействие с DOM

Итак, `Doubloons4Gold` выполняется в браузере, и на следующем этапе мы вернемся к функциям, для которых ранее создали заглушки, и улучшим процесс взаимодействия с пользователем — так, чтобы не нужно было обращаться к консоли разработчика. Вначале обновим реализацию `InputOutput.kt`

для представления контента на странице. Затем отформатируем денежные суммы в `formatAsCurrency`.

В функции `output` необходимо обновить код веб-страницы, чтобы в ней отображалась строка выводимого текста. Для этого вам придется взаимодействовать с DOM — набором API, который позволяет на программном уровне изменять HTML-страницы. Для упрощения задачи воспользуемся библиотекой HTML из Kotlinx, которая предоставляет удобный API для объявления элементов HTML в коде Kotlin.

После остановки `Doubloons4Gold` объявите зависимость для этой библиотеки в блоке `jsMain` файла `build.gradle`. (Незавершенные сборки могут помешать синхронизации с Gradle в IntelliJ.)

Листинг 26.6. Добавление зависимости для библиотеки HTML Kotlinx (и `build.gradle`)

```
...
kotlin {
    jvm()
    macOSX64() {
        binaries {
            executable()
        }
    }
    js {
        browser()
        binaries {
            executable()
        }
    }
    sourceSets {
        commonMain {
        }
        commonTest {
        }
        jsMain {
            dependencies {
                implementation "org.jetbrains.kotlinx:kotlinx-html-js:0.7.3"
            }
        }
    }
}
```

Это объявление несколько отличается от тех, которые встречались вам ранее. Так как манипуляции с DOM будут выполняться только в приложении Kotlin/JS, зависимость объявлена для набора исходного кода `jsMain`. Она применяется только к результатам этой сборки. Тем самым мы сокращаем итоговый размер других приложений, в которых эта зависимость использоваться не будет. Если вы добавите ее в блоки `commonMain` или `commonTest`, она будет доступна независимо от выбора целевой платформы.

Проведите синхронизацию с Gradle, чтобы среда IntelliJ знала о новой зависимости.

Теперь обновите функции `output` и `getInput`, чтобы добавить контент к вашей странице. Откройте файл `jsMain/kotlin/InputOutput.kt`.

Для получения доступа к DOM и внесению изменений на страницу используется свойство `document`. Это свойство объявляется в JavaScript и содержит множество функций и свойств, влияющих на визуализацию страницы. Для обращения к контенту `<body>` разметки HTML из кода Kotlin используется свойство `body` объекта `document`.

Для добавления нового контента используется `append` — функция, определяемая в JavaScript для свойства `body`. Впрочем, мы воспользуемся перегруженной версией из библиотеки HTML Kotlinx, только что добавленной в проект. Перегруженная версия использует лямбда-выражение для определения добавляемого контента, и она намного выразительнее и идиоматичнее встроенных API JavaScript.

Обновите функции `output` и `getInput` для добавления на страницу отформатированного текста. Синтаксис мы объясним далее.

Листинг 26.7. Добавление абзацев (`jsMain/kotlin/InputOutput.kt`)

```
import kotlinx.browser.document
import kotlinx.html.dom.append
import kotlinx.html.*

actual fun output(message: String) {
    val body = checkNotNull(document.body) {
        "Could not locate the <body> tag"
    }

    body.append {
        p { +message }
    }
    println(message)
}

actual fun getInput(prompt: String): String {
    val body = checkNotNull(document.body) {
        "Could not locate the <body> tag"
    }

    val input = "10.0"

    body.append {
        p {
            em { +prompt }
            +" "
            strong { +input }
        }
    }
}
```



```
    }  
    return input  
}
```

В блоках `body.append` вызываются функции (также из библиотеки HTML Kotlinx) для добавления контента и манипуляций с ним. Для добавления абзацев используется функция `p`; для курсивного оформления текста — функция `em` (сокращение от `emphasize`), а для оформления текста жирным шрифтом — функция `strong`. Имена этих функций соответствуют именам тегов HTML, которые добавляются на страницу (`<p>`, `<em>` и `<strong>`).

Чтобы указать, что строка должна вставляться в элемент в виде текста, снабдите ее префиксом «унарный плюс» (+).

Итак, `output` теперь добавляет в блок `body` содержимое переданного сообщения, а `getInput` добавляет в `body` абзац с текстом приглашения, набранным курсивом, за которым следует пробел и ввод, выделенный жирным шрифтом.

Снова запустите `Doubloons4Gold`.

Теперь веб-страница будет обновляться динамически. Она должна выглядеть примерно как на рис. 26.2.

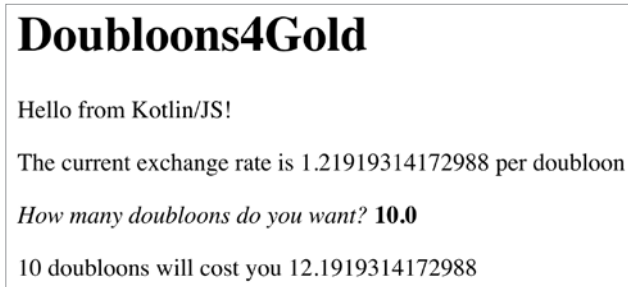


Рис. 26.2. Динамически обновляемая страница
(текст на странице: «Привет от Kotlin/JS! Курс обмена 1.2191... за дублон. Сколько дублонов вам нужно? — 10.0. Цена 10 дублонов будет 12.191....»)

После изменения модели DOM для отображения вывода на странице можно запросить входные данные у пользователя. Для этого нам понадобится еще одна переменная DOM — `window`.

У `window` имеется функция `prompt`, которая выводит временное окно с полем ввода; она же возвращает текст, набранный в поле. Мы воспользуемся этой функцией для получения от пользователя нужной суммы в дублонах.

Чтобы вывести курс обмена и итоговую сумму, воспользуемся функцией `alert` объекта `window`. Она отображает модальное окно, которое не получает от пользователя никаких данных и содержит только кнопку ОК для закрытия.

Вставьте вызовы `prompt` и `alert` для завершения реализаций `output` и `getInput`.

Листинг 26.8. Вывод информации для пользователя (jsMain/kotlin/InOutOutput.kt)

```
import kotlinx.browser.document
import kotlinx.browser.window
import kotlinx.html.dom.append
import kotlinx.html.*

actual fun output(message: String) {
    val body = checkNotNull(document.body) {
        "Could not locate the<body> tag"
    }

    body.append {
        p { +message }
    }

    println(message)
    window.alert(message)
}

actual fun getInput(prompt: String): String {
    val body = checkNotNull(document.body) {
        "Could not locate the <body> tag"
    }

    val input = "10-0" window.prompt(message = prompt, default = "") ?: ""

    body.append {
        p {
            em { +prompt }
            +" "
            strong { +input }
        }
    }

    return input
}
```

Снова запустите `Doubloons4Gold`. После того как страница загрузится, появляется модальное окно с сообщением `The current exchange rate is [курс] per doubloon`. После нажатия кнопки `OK` появится еще одно модальное окно с сообщением `How many doubloons do you want?` и полем для ввода.

Введите произвольное число и щелкните на кнопке `OK`. Откроется модальное окно с сообщением `[число] doubloons will cost you [сумма]`. При нажатии кнопки `OK` на странице появится информация о совершенном обмене.

(Из-за особенностей реализации функции `alert` и `DOM` ваша страница не перерисовывается от сообщения к сообщению. Перерисовка происходит после

возврата управления из функции `convertCurrency`. Вы можете дописать код для перерисовки страницы после вывода каждого модального окна, но это сделает код значительно более громоздким.)

Функции `output` и `getInput` завершены. Тем не менее `Doubloons4Gold` надо немного довести до ума. Когда ваша страница выводит денежную сумму, вывод отображается как дробное число без форматирования и без округления. Наша следующая задача — вызов API интернационализации в JavaScript для выполнения форматирования.

Ключевое слово `external`

В JavaScript существует ряд API интернационализации, упакованных в объект `Intl`. Мы используем класс `Intl.NumberFormat`, который умеет форматировать денежные суммы для заданной локали (региональных настроек). С документацией этого класса можно ознакомиться на developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl/NumberFormat.

Попробуйте использовать этот класс в файле `Currency.kt`. Не пытайтесь добавлять какие-либо команды `import`. После внесения изменений появятся ошибки, о которых мы поговорим далее.

Листинг 26.9. Использование неизвестного класса JavaScript (`jsMain/kotlin/Currency.kt`)

```
actual fun Double.formatAsCurrency(): String {  
    return toString()  
    val numberFormatter = Intl.NumberFormat()  
  
    return numberFormatter.format(this)  
}
```

В Kotlin/JVM и в Kotlin/Native реализованы средства совместимости, которые автоматически позволяют вызывать API, объявленные платформой и внешними библиотеками, одновременно пользуясь всеми преимуществами безопасности типов и средствами анализа, которые предоставляют Kotlin и IntelliJ. Kotlin/JS также содержит средства совместимости, которые позволяют вызывать код JavaScript из Kotlin (и наоборот). Однако вы не можете просто вызывать JavaScript и при этом иметь доступ ко всем возможностям Kotlin и IDE, как в случае с Kotlin/JVM и Kotlin/Native.

Чтобы обращаться к любому классу или функции, объявленным в JavaScript, и при этом наслаждаться преимуществами компилятора Kotlin и IDE, JavaScript API должен быть включен в стандартную библиотеку Kotlin/JS, определен во включаемой библиотеке Kotlin либо его заглушки должны быть объявлены вручную в вашем коде. `Intl.NumberFormat` не удовлетворяет ни одному из этих критериев, поэтому вам не удастся к нему обратиться.

В подобных ситуациях Kotlin даже не знает о существовании типа — таким образом, использование API нельзя свести к командам `import`, как мы делали для кода Kotlin/JVM и Kotlin/Native.

Чтобы решить эту проблему, необходимо сообщить Kotlin о классе `Intl` при помощи ключевого слова `external`. Создайте в папке `jsMain/kotlin` новый файл Kotlin с именем `Intl.kt`, в котором будут храниться заголовки функций и классов для `Intl.NumberFormat`. Эти заголовки передают Kotlin информацию об API, доступных из JavaScript.

Листинг 26.10. Объявление внешнего класса (`jsMain/kotlin/Intl.kt`)

```
external class Intl {  
    class NumberFormat {  
        constructor()  
  
        fun format(number: Number): String  
    }  
}
```

Помечая класс, функцию или свойство ключевым словом `external`, вы тем самым сообщаете Kotlin, что они являются *внешними*, то есть объявляются и реализуются в JavaScript. Внешние классы и функции не могут иметь реализаций в коде Kotlin. Они существуют исключительно для того, чтобы сообщить компилятору об API, определяемых в JavaScript. После только что сделанного объявления внешнего класса вам стали доступны конструкторы `Intl.NumberFormat` и функция `format`.

Ключевое слово `external` (или `extern` в некоторых языках) существует не только в Kotlin/JS, но в Kotlin/JS оно имеет особый смысл. Оно используется только для объявления API, не определенных в Kotlin, чтобы они могли вызываться из кода Kotlin. В дополнение к только что продемонстрированному способу применения ключевое слово `external` также задействуют для JVM при использовании JNI (Java Native Interface), но эта тема выходит за рамки нашей книги.

При использовании ключевого слова `external` компилятор доверяет вашим заявлениям о типах и функциях, которые вы объявляете. Вы отвечаете за то, чтобы эти определения не содержали опечаток и точно соответствовали определениям в JavaScript. В противном случае в программе возникнут ошибки времени выполнения, потому что вы пытаетесь обратиться к несуществующим API. Преимущество ключевого слова `external` таково: с того момента, как вы определите эти объявления (и тщательно проверите их), компилятор уверен, что вы не допустите опечаток в будущем. За дополнительной информацией о встроенных типах JavaScript обращайтесь к документации разработчика Mozilla на developer.mozilla.org/en-US/docs/Web/JavaScript/Reference.

У `Intl` и `Intl.NumberFormat` есть и другие API, которые доступны в JavaScript, но не включены в наше определение `external`. И это нормально — объявлять необходимо только те классы и функции, которые вы собираетесь вызывать из своего кода Kotlin. Любые функции и классы, не включенные в определение `external`, остаются неизвестными компилятору и не могут вызываться из кода Kotlin напрямую.

Запустите `Doubloons4Gold` и убедитесь, что проект теперь строится без ошибок. В выводе используется форматирование, принятое в `Intl.NumberFormat` по умолчанию для числовых значений, а результат выглядит примерно так:

```
The current exchange rate is 0.845 per doubloon
How many doubloons do you want? 1.5
1.5 doubloons will cost you 1.268
```

Числа, округленные до трех знаков в дробной части, выглядят намного приятнее. Это значимое улучшение, но мы хотим, чтобы класс `Intl.NumberFormat` был настроен для вывода денежной суммы. Это потребует еще некоторых усилий и применения средств совместимости с JavaScript.

Выполнение произвольного кода JavaScript

Чтобы указать, что числа должны форматироваться как денежные суммы, необходимо использовать другой конструктор вместо того, который мы использовали во внешнем определении `Intl.NumberFormat`. Новый конструктор будет получать два аргумента: локаль и опциональный набор спецификаций форматирования. Начните с указания параметра `locale`. Параметр `options` будет объявлен позже (пока мы используем набор параметров по умолчанию).

Листинг 26.11. Объявление второго конструктора (`jsMain/kotlin/Intl.kt`)

```
external class Intl {
    class NumberFormat {
        constructor()

        constructor(locale: String)

        fun format(number: Number): String
    }
}
```

Параметр `locale` задает локаль (региональные настройки) пользователя. Он влияет на такие настройки форматирования, как разделитель групп разрядов и разделитель дробной части. Для получения локали браузера можно воспользоваться свойством `navigator.language`.

Как и `Intl.NumberFormat`, свойство `navigator` изначально неизвестно Kotlin. Чтобы обратиться к нему, можно объявить дополнительные заглушки `external`,

чтобы сообщить Kotlin об этих API. Такое решение сработает, но для одноразовых вызовов базовых API оно получается слишком громоздким.

Вместо определения внешних интерфейсов в Kotlin также можно динамически исполнить произвольный фрагмент JavaScript в коде Kotlin. Задача решается вызовом функции `js`. Функция получает один аргумент `String` с кодом JavaScript, который требуется динамически исполнить. Чтобы опробовать эту возможность, добавим в `Currency.kt` новое свойство с именем `userLocale`.

Листинг 26.12. Выполнение произвольного кода JavaScript (`jsMain/kotlin/Currency.kt`)

```
private val userLocale: String
    get() = js("navigator.language") as String? ?: "en-US"

actual fun Double.formatAsCurrency(): String {
    val numberFormatter = Intl.NumberFormat(
        locale = userLocale
    )

    return numberFormatter.format(this)
}
```

Свойство `userLocale` выполняет код `navigator.language` и возвращает его значение. Если язык неизвестен, свойство равно `null`. В таком случае используется английский язык (США).

Такая встроенная реализация произвольного кода JavaScript имеет ряд недостатков. Прежде всего, компилятор Kotlin не может проверить код JavaScript, определенный этим способом. Вы должны сами проследить, чтобы в коде не было опечаток; в противном случае вы узнаете о них только при выполнении программы.

Kotlin также не может автоматически определить возвращаемый тип выражений, которые вычисляются функцией `js`. В данном случае мы явно преобразуем результат к `String?`. Вам как разработчику следует проследить, чтобы эта информация о типе соответствовала той, которая будет получена в ходе выполнения программы; иначе вас ждут сообщения об ошибках и неожиданное поведение программы.

Мы рекомендуем использовать функцию `js` осмотрительно. И все-таки в некоторых ситуациях встраивание низкоуровневого кода JS имеет смысл — обычно это разовые акции, когда усилия для определения внешних загрузок не окупаются выгодой от применения этих API. Если вам потребуется использовать свойство `navigator` или `language` в другой точке вашего кода, возможно, лучше определить внешние классы, а не полагаться на `js`.

Снова запустите `Doubloons4Gold` и убедитесь в том, что вывод не изменился.

Динамические типы

В главе 2 мы упоминали, что Kotlin использует статическую систему проверки типов. Это означает, что каждая переменная, параметр функции, возвращаемый тип и выражение в Kotlin имеют тип, известный на стадии компиляции. Он либо автоматически определяется компилятором, либо объявляется явно в коде Kotlin.

Статическая система проверки типов гарантирует, что вы обращаетесь к поведению, доступному для данных, с которыми вы работаете, — и это очень полезно. Многие средства редактирования, анализа и рефакторинга IntelliJ основаны на статической типизации.

Но в JavaScript применяется *динамическая* система типов: они проверяются на стадии выполнения, а не на стадии компиляции. Кроме того, в JavaScript нет встроенного синтаксиса объявления типа переменной или возвращаемого типа функции.

У динамической системы есть свои преимущества. В коде будет меньше ошибок, которые проявляются перед запуском, а это ускоряет разработку. Иногда динамические типы могут ускорить некоторые изменения в коде или рефакторинг, так как типы можно менять с минимальными правками в коде.

Однако динамические типы — палка о двух концах. Без участия компилятора, проверяющего правильность использования типов, разработчик может нечаянно внести изменения, нарушающие работоспособность кода. И эти ошибки проявятся только во время выполнения; это означает, что тестирование кода критично после изменений в типах возвращаемых значений или переменных.

При использовании Kotlin/JS язык сохраняет статическую типизацию, но у вас также появляется возможность применения динамических типов в коде. Для примера вернемся к функции `format`. В JavaScript вызов этой функции может выглядеть так:

```
const price = 25.68;
const locale = navigator.language ?? "en-US";
const formatOptions = { style: 'currency', currency: 'USD' };
const numberFormat = new Intl.NumberFormat(locale, formatOptions);

numberFormat.format(price);
```

Синтаксис немного непривычен, но у каждой конструкции в этом фрагменте существует прямой аналог Kotlin. Взгляните на объявление `formatOptions`: в правой части присваивания создается новый объект с двумя свойствами, `style` и `currency`. Так как JavaScript использует динамическую систему типов, класс `Intl.NumberFormat` может прочитать эти два свойства, несмотря на то что объект не расширяет конкретный базовый класс или интерфейс.

Вернемся к новому конструктору. Вы уже передали параметр `locale`, теперь требуется параметр `options`. Он определяет необязательные критерии,

влияющие на форматирование значения. Здесь можно объявить, что значение должно форматироваться как денежная сумма. Но к какому типу оно должно относиться?

Так как этот параметр используется динамически, нет никаких спецификаций, определяющих используемые типы. Значит, в качестве `options` можно передать любое значение независимо от того, является ли оно содержательным вводом или нет. В таких ситуациях допустимо воспользоваться специальным типом `dynamic`, доступным только в Kotlin/JS. Добавьте в конструктор дополнительный параметр с использованием нового типа.

Листинг 26.13. Объявление параметра `dynamic` (jsMain/kotlin/Intl.kt)

```
external class Intl {  
    class NumberFormat {  
        constructor()  
  
        constructor(locale: String, options: dynamic)  
  
        fun format(number: Number): String  
    }  
}
```

Тип `dynamic` можно использовать где угодно в коде Kotlin/JS, хотя мы рекомендуем применять его только как средство совместимости при вызове кода, написанного на JavaScript. (Кстати говоря, функция `js` возвращает тип `dynamic`; именно поэтому ее результат пришлось явно приводить к `String`?.)

Ранее упоминалось, что параметр `options` не является обязательным. В JavaScript, как и в Kotlin, существует концепция параметров по умолчанию. Чтобы предоставить доступ к параметрам по умолчанию в коде Kotlin, можно использовать значение `definedExternally`. Оно указывает, что значение существует, но имеет внешнее определение. Обновите конструктор `Intl.NumberFormat`, чтобы задать значение по умолчанию для параметра `options`.

Листинг 26.14. Объявление внешнего параметра по умолчанию (jsMain/kotlin/Intl.kt)

```
external class Intl {  
    class NumberFormat {  
        constructor()  
  
        constructor(locale: String, options: dynamic = definedExternally)  
  
        fun format(number: Number): String  
    }  
}
```

`definedExternally` часто применяют в ситуациях, когда внешняя функция имеет аргументы по умолчанию, но также и с переменными, определенными в JavaScript.

При желании можно использовать `definedExternally` и как тело внешней функции — при условии, что это единственная команда в теле.

Зададим для параметра `options` аргумент. Есть несколько способов создания объекта `options` в коде Kotlin, и один из них — посредством функции `js`. Если вы выберете этот путь, вызов функции будет выглядеть так:

```
val options: dynamic = js("{ style: 'currency', currency: 'USD' }")
```

Но чтобы продемонстрировать мощь динамических типов в Kotlin, мы определим объект внутри Kotlin и используем его динамически. Для объявления параметров в Kotlin можно воспользоваться объектом-выражением.

Мы выберем именно этот способ, указав в качестве формата вывода денежные суммы и доллары США в качестве денежной единицы. (В JavaScript нет API для определения предпочитаемой валюты. Если для вашего приложения это критично, предложите пользователю выбрать денежную единицу или определите ее автоматически на основании страны местонахождения.)

Листинг 26.15. Динамическое использование объекта (jsMain/kotlin/Currency.kt)

```
private val userLocale: String
    get() = js("navigator.language") as String? ?: "en-US"

actual fun Double.formatAsCurrency(): String {
    val numberFormatter = Intl.NumberFormat(
        locale = userLocale,
        options = object {
            @JsName("style") val style = "currency"
            @JsName("currency") val currency = "USD"
        }.asDynamic()
    )

    return numberFormatter.format(this)
}
```

К свойствам добавлены аннотации `@JsName`. Когда Kotlin компилирует код в файл JavaScript, он может минимизировать выходной код, чтобы сократить размер файла скрипта. Как следствие, имена переменных искажаются после компиляции. Это создает проблемы, потому что класс `Intl.NumberFormat` обрабатывает свойства по именам. Снабжая свойства аннотацией `@JsName`, вы сообщаете компилятору имена свойств, которые должны сохраниться после компиляции кода.

Также мы вызываем `asDynamic`, чтобы объект рассматривался как динамический тип. Это необязательно, поскольку параметр `options` уже имеет тип `dynamic`, но так в вашем коде более явно указано, что значение будет использоваться динамически. При вызове `asDynamic` значение преобразуется к типу `dynamic`.

Вы все еще можете вызывать функции и читать свойства значений `dynamic`, но будьте внимательны: IntelliJ и компилятор Kotlin никак не проверяют, что

вызываемые функции и свойства существуют. А компилятор Kotlin не пытается обрабатывать функции-расширения или свойства для динамических типов. То же самое относится и к функциям области видимости, так что если вы попытаетесь вызвать функцию (например, `let` или `apply`) для динамического типа, компилятор решит, что функция для типа существует.

Снова запустите `Doubloons4Gold`. На этот раз вывод должен выглядеть так:

```
The current exchange rate is $1.42 per doubloon
How many doubloons do you want? 25
25 doubloons will cost you $35.44
```

Приложение `Doubloons4Gold` готово. Герой может поменять свои монеты на местные дублоны, расслабиться и наслаждаться тропическими красотами.

Итак, что сделано в трех последних главах? Мы написали одно приложение, которое работает на трех разных платформах: JVM, macOS и веб-страницах. В каждом случае для создания соответствующего UI используются платформенно-зависимые API. И все это не требует переписывать критическую для бизнес-логики функцию `convertCurrency`.

Для любознательных: фреймворки клиентской части

Существует много популярных фреймворков для построения веб-UI, среди них наиболее популярны React, Vue и Angular. С технической точки зрения любые такие библиотеки можно использовать со средствами совместимости, о которых вы узнали в этой главе. JetBrains также предоставляет привязки Kotlin/JS для React, что избавляет вас от хлопот с самостоятельным построением уровня совместимости.

Тем не менее мы не рекомендуем использовать эти фреймворки из Kotlin напрямую. Использование JavaScript-фреймворка клиентской части внутри Kotlin часто приводит к созданию кода более сложного, чем эквивалентный код JavaScript. Кроме того, в интернете можно найти гораздо больше ресурсов и руководств по этим традиционным JavaScript-фреймворкам, чем по Kotlin.

Помните: мощь Kotlin Multiplatform в полной мере проявляется при совместном использовании приложениями критической логики. Вместо того чтобы писать веб-интерфейс на Kotlin, мы рекомендуем применить Kotlin/JS для построения библиотеки, которую можно использовать из другого приложения JavaScript. Если вы захотите больше узнать о вызове Kotlin из JavaScript, обратитесь к документации на kotlinlang.org/docs/js-to-kotlin-interop.html.

Задание: комиссионные при обмене валюты

В настоящее время Doubloons4Gold обменивает деньги по текущему курсу. Никакие комиссионные в цену не включаются, что вряд ли можно назвать выгодной бизнес-стратегией.

Реализуйте новую политику, в которой за каждую операцию обмена взимаются комиссионные 5%. Установите минимальный размер комиссионных \$5, чтобы клиентам было выгодно уменьшить количество операций обмена. После внесенных изменений вывод должен выглядеть примерно так:

```
Hello from Kotlin/JS!
The current exchange rate is $1.19 per doubloon
How many doubloons do you want?
5
5.0 doubloons is worth $5.96
It will cost $10.96 for 5.0 doubloons
Hello from Kotlin/JS!
The current exchange rate is $1.37 per doubloon
How many doubloons do you want?
100
100.0 doubloons is worth $137.27
It will cost $144.13 for 100.0 doubloons
```

Проследите за тем, чтобы политика затрагивала всех клиентов независимо от того, используют ли они веб-браузер, JVM или приложение macOS. В приложении не должно быть никаких лазеек или особых условий для каких-либо платформ.

Послесловие

Вот и все. Вы познакомились с основами языка программирования Kotlin. Можете себя поздравить.

С этого момента и начинается настоящая работа.

Что дальше?

Kotlin используется для решения многих задач — от обработки запросов на сервере до управления Android-приложением или организации совместного использования кода на разных платформах. После того как вы изучили материал этой книги, вы, скорее всего, уже решили, где и как будете применять полученные знания. Поэтому действуйте! Ведь только так можно научиться писать хороший код.

Если вы хотите погрузиться в документацию Kotlin, посетите сайт kotlinlang.org. Если нужны справочные материалы, рекомендуем книгу «Kotlin in Action»¹ (manning.com/books/kotlin-in-action).

Вам необязательно писать код в одиночку: сообщество Kotlin — очень активное и приветствует развитие языка. Kotlin — проект с открытым исходным кодом, поэтому если вы хотите следить за его развитием (или даже внести свой вклад), обращайтесь на GitHub: github.com/jetbrains/kotlin. Мы рекомендуем связаться с локальными пользовательскими группами Kotlin, а если в вашем окружении такой группы нет, создайте ее.

Бесстыдная самореклама

Если хотите подписаться на авторов этой книги, то сделать это можно в Twitter по адресам [@_andrewbailey](https://twitter.com/_andrewbailey) (Эндрю), [@mutexkid](https://twitter.com/mutexkid) (Джош) и [@drgreenhalgh](https://twitter.com/drgreenhalgh) (Дэвид).

Если захотите больше узнать про Big Nerd Ranch — загляните на bignerdranch.com. У нас есть масса других полезных учебников, которые можно найти по ссылке bignerdranch.com/books. Мы особо рекомендуем «Android Programming: The Big Nerd Ranch Guide»². Разработка на Android — это хороший способ применить на практике полученные знания о Kotlin.

¹ Исакова С., Жемеров Д. Kotlin в действии.

² Филлипс, Б., Стюарт, К., Марсикано, К. Android. Программирование для профессионалов. СПб., Издательство «Питер».

Мы также ведем курсы интенсивного обучения и разрабатываем приложения для клиентов. Если вы мечтаете придумать какой-нибудь гениальный код, Big Nerd Ranch готов вам помочь.

Спасибо вам

И напоследок мы просто говорим спасибо. Благодаря вам — да-да, именно *вам!* — эта книга стала возможна.

Мы надеемся, что вы получили такое же удовольствие, читая ее, как и мы — создавая. А теперь идите и напишите свое шедевральное приложение на Kotlin.

Глоссарий

Kotlin REPL	Инструмент IntelliJ IDEA, который позволяет протестировать код без создания файла или запуска готовой программы.
null	Несуществующее значение.
активный поток данных (hot flow)	Поток данных, который всегда находится в активном состоянии независимо от того, потребляются ли из него данные. См. <i>поток данных</i> , <i>пассивный поток данных</i> .
аргумент (argument)	Входные данные для функции.
аргумент, именованный (argument, named)	Аргумент функции, передаваемый с указанием имени параметра, которому соответствует данный аргумент.
аргумент, по умолчанию (argument, default)	Значение, присваиваемое аргументу функции, если при вызове функции не было передано фактическое значение.
байт-код (bytecode)	Низкоуровневый язык, используемый в виртуальной машине Java.
блок инициализации (initializer block)	Блок кода с префиксом <code>init</code> , который будет выполнен в ходе инициализации экземпляра объекта.
ветвь (branch)	Часть кода, выполняемая при определенном условии.
взаимоисключающий доступ (mutual exclusion)	Практика блокировки участка кода, чтобы он был доступен только для одного программного потока.
видимость (visibility)	Доступность элементов из других частей кода.
внешний код (external)	Код, объявленный и реализованный за пределами Kotlin.

возврат, неявный (return, implicit)	Возврат значения без явного использования команды <code>return</code> .
время выполнения (runtime)	Период времени, когда выполняется программа.
встраивание функции (function inlining)	Оптимизация работы компилятора, используемая для уменьшения затрат памяти у функций, получающих в качестве аргументов анонимные функции.
выражение (expression)	Сочетание значений, операторов и функций, которые создают новое значение.
геттер, get-метод (getter)	Функция, определяющая, как читается свойство.
декремент, оператор (decrement operator)	Уменьшает значение переменной на 1: (--).
делегат (delegate)	Способ определения свойства с использованием шаблона его поведения.
деструктуризация (destructuring)	Объявление и присваивание нескольким переменным по одному значению в одном выражении.
допускающий null (nullable)	С возможностью присваивания null.
зарезервированное ключевое слово (reserved keyword)	Слово, которое не может использоваться в качестве имени в коде.
значение, перечисления (value, enumerated)	Возможное значение экземпляра класса перечисления. См. <i>класс, перечисления</i> .
изменяемость (mutable)	Возможность изменения. См. <i>только для чтения</i> .
императивное программирование (imperative programming)	Парадигма программирования, когда серии исполняемых команд выполняются по очереди.
индекс (index)	Целое число, определяющее позицию элемента в серии.
инициализация (initialization)	Подготовка переменной, свойства или экземпляра класса к использованию.

инициализация, отложенная (initialization, lazy)	Инициализации переменной не происходит до момента первого обращения к ней.
инициализация, поздняя (initialization, late)	Инициализация переменной происходит с задержкой, но до первого обращения к ней.
инкапсуляция (encapsulation)	Принцип, допускающий видимость функций и свойств объекта другими объектами, только если это необходимо. Также процесс ограничения доступа к реализации функций и свойств с помощью модификаторов видимости.
инкремент, оператор (increment operator)	Увеличивает значение элемента, к которому он применяется, на 1: (++)
интервал (range)	Последовательный набор значений или символов.
интерполяция строк (string interpolation)	Использование строковых шаблонов.
интерфейс (interface)	Набор абстрактных функций и свойств, определяющий общие черты объектов, не связанных наследованием.
исключение (exception)	Нарушение хода выполнения программы, ошибка.
исключение, непроверяемое (exception, unchecked)	Исключение, которое может выдаваться без его обязательного перехвата try/catch или объявления того, что функция выдает исключение.
итерация (iteration)	Повторение процесса, например, для каждого элемента в интервале или коллекции.
канал (channel)	Механизм передачи значений между сопрограммами с использованием приостанавливаемых функций <code>send</code> и <code>receive</code> .
канал, буферизованный (channel, buffered)	Канал с буфером заданного размера.
канал, встречный (channel, rendezvous)	Канал без буфера.
канал, неограниченный (channel, unlimited)	Канал с буфером, размер которого не имеет заданной верхней границы.

канал, с заменой (channel, conflated)	Канал с буфером единичного размера, который заменяет существующее значение при отправке нового значения.
класс (class)	Определение категории объектов, представленных в коде.
класс, абстрактный (class, abstract)	Класс, экземпляры которого никогда не создаются. Используется для определения общих возможностей его подклассов.
класс, вложенный (class, nested)	Именованный класс, объявленный внутри другого класса.
класс, данных (class, data)	Класс, специально предназначенный для манипуляций с данными.
класс, изолированный (class, sealed)	Класс с определенным множеством подтипов, что позволяет компилятору проверить, содержит ли выражение <code>when</code> полный набор вариантов. См. <i>алгебраический тип данных</i> ; <i>класс, перечисления</i> .
класс, перечисления (class, enumerated)	Класс, определяющий коллекцию констант, которые называют перечисляемыми значениями: все экземпляры класса являются одним из объявленных значений. См. <i>класс, изолированный</i> ; <i>значение, перечисления</i> .
ковариантность (covariance)	Обозначение обобщенного параметра как производителя. См. <i>производитель</i> .
коллекция, готовая (collection, eager)	Коллекция, элементы которой доступны сразу после создания. См. <i>коллекция, отложенная</i> .
коллекция, отложенная (collection, lazy)	Коллекция, элементы которой создаются только при обращении к ним. См. <i>коллекция, готовая</i> ; <i>функция, итератор</i> .
команда (statement)	Инструкция в коде.
комментарий (code comment)	Пояснение в коде; комментарии игнорируются компилятором.

компаньон, объект (object, companion)	Объект, определенный в классе и помеченный модификатором <code>companion</code> ; объекты-компаньоны предоставляют доступ к своим членам с указанием имени внешнего класса. <i>См. объявление объекта; объектное выражение; синглет.</i>
компилируемый язык (compiled language)	Язык, исходный код на котором преобразуется компилятором в машинные команды перед выполнением. <i>См. компиляция; компилятор.</i>
компилятор (compiler)	Программа, выполняющая компиляцию. <i>См. компиляция.</i>
компиляция (compilation)	Преобразование исходного кода в низкоуровневый язык для создания выполняемой программы.
конкатенация строк (string concatenation)	Соединение двух или более строк в одну.
консоль (console)	Панель в окне IntelliJ IDEA, которая выводит информацию о действиях программы после ее запуска, а также ее выходные данные.
константа (constant)	Элемент, хранящий неизменное значение.
конструктор (constructor)	Специальная функция, которая готовит класс к использованию во время создания экземпляра.
конструктор, главный (constructor, primary)	Конструктор, определенный в заголовке класса; все остальные конструкторы класса должны делегировать часть работы главному конструктору.
контравариантность (contravariance)	Обозначение обобщенного параметра как потребителя. <i>См. потребитель.</i>
логический оператор (logical operator)	Функция или оператор, который выполняет логическую операцию над входными данными.
логический оператор И (logical 'and' operator)	Возвращает истинное значение, если истинны значения в левой и правой части: <code>(&&)</code> .

логический оператор ИЛИ (logical 'or' operator)	Возвращает истинное значение, если истинно хотя бы одно из значений в левой и правой части: (<code> </code>).
логический оператор НЕ (logical 'not' operator)	Возвращает истинное значение, если значение элемента, к которому применяется оператор, ложно, или ложное значение, если значение элемента истинно: (<code>!</code>).
лямбда-выражение (lambda expression)	Другой термин для обозначения анонимной функции. См. <i>функция, анонимная</i> .
лямбда-функция (lambda)	Другой термин для обозначения анонимной функции. См. <i>функция, анонимная</i> .
метод (method)	Функция в терминологии Java. См. <i>функция</i> .
модификатор видимости (visibility modifier)	Модификатор, добавляемый к функции или свойству для определения области их видимости.
модуль (module)	Отдельный блок функциональности, который можно независимо выполнять, тестировать и отлаживать.
накопитель, переменная (accumulator variable)	Временная переменная, используемая для вычисления результата операции с серией значений.
наследование (inheritance)	Принцип объектно-ориентированного программирования, в котором свойства и поведение класса наследуются его подклассами.
не допускающий null (non-nullable)	Без возможности присваивания null.
область видимости (scope)	Область программы, в которой сущность (например, переменная) доступна по имени.
область видимости сопрограммы (coroutine scope)	Экземпляр CoroutineScope, управляющий выполнением сопрограммы.
обобщения (generics)	Механизм системы типов, позволяющий функциям и типам работать с разными типами.

обобщенный параметр-тип (generic type parameter)	Параметр, задаваемый для обобщенного типа (например, <T>).
объектное выражение (object expression)	Синглтон без имени, созданный с ключевым словом <code>object</code> . См. <i>компаньон</i> , <i>объект</i> ; <i>объявление объекта</i> ; <i>синглтон</i> .
объявление объекта (object declaration)	Именованный синглтон, созданный с помощью ключевого слова <code>object</code> . См. <i>компаньон</i> , <i>объект</i> ; <i>объектное выражение</i> ; <i>синглтон</i> .
оператор (для потоков данных) (operator (for flows))	Функция, изменяющая способ генерирования объектов потоком данных.
оператор безопасного вызова (safe call operator)	Вызывает функцию, только если вызываемый элемент отличен от null: (?).
оператор безопасного приведения типа (safe cast operator)	Пытается привести один тип к другому и возвращает null, если приведение типа недействительно: (as?). См. <i>приведение типа</i> .
оператор вычитания с присваиванием (subtraction and assignment operator)	Вычитает значение в правой части из переменной в левой части: (-=).
оператор обращения по индексу (indexed access operator)	Возвращает значение элемента из коллекции по индексу: [].
оператор присваивания (assignment operator)	Присваивает значение в правой части элементу в левой части: (=).
оператор равенства ссылок (referential equality operator)	Определяет, ссылаются ли переменные в левой и правой части на один и тот же экземпляр: (===). См. <i>равенство</i> , <i>ссылок</i> .
оператор слияния с null (null coalescing operator)	Возвращает элемент в левой части, если он отличен от null; в противном случае возвращает элемент в правой части: (?:).

оператор сложения с присваиванием (addition and assignment operator)	Прибавляет значение в правой части к переменной в левой части: $(+=)$.
оператор сравнения (comparison operator)	Оператор, сравнивающий элементы в левой и в правой части.
оператор структурного равенства (structural equality operator)	Проверяет равенство значений в правой и левой части: $(==)$. См. <i>равенство, структурное</i> .
ошибка времени выполнения (runtime error)	Ошибка, возникающая после компиляции, во время выполнения программы.
ошибка времени компиляции (compile-time error)	Ошибка, возникающая во время компиляции. См. <i>компиляция</i> .
параметр (parameter)	Входные данные, необходимые функции.
параметрический тип (parameterized type)	Тип, указанный в точке вызова, который должен использоваться для обобщенного типа.
пассивный поток данных (cold flow)	Поток данных, который начинает работать только после начала потребления данных из него. См. <i>поток данных, активный поток данных</i> .
перегрузка оператора (operator overloading)	Определение реализации операторной функции для пользовательского типа.
перегрузка функции (function overloading)	Объявление двух или более реализаций функции с одинаковым именем и областью видимости, но с разными параметрами.
переменная (variable)	Элемент, содержащий значение; переменные бывают доступными только для чтения и изменяемыми.
переменная, локальная (variable, local)	Переменная, определяемая в области видимости функции.
переменная, уровня файла (variable, file-level)	Переменная, определяемая за пределами любой функции или класса.
переопределение (override)	Определение своей реализации для унаследованной функции или свойства.

платформенный тип (platform type)	Неоднозначные типы, возвращаемые Kotlin из кода Java; могут допускать или не допускать null.
подкласс (subclass)	Класс, объявленный с наследованием свойств другого класса.
поле (field)	Место хранения данных, связанных со свойством.
полиморфизм (polymorphism)	Способность использовать одинаковую именованную сущность (например, функцию) для получения разных результатов.
получатель (receiver)	Субъект применения функции-расширения.
поток данных (flow)	Путь передачи данных, на который можно подписаться.
потребитель (consumer)	Класс с обобщенным типом, который используется только для ввода данных класса, но не для вывода.
предикат (predicate)	Условие истина/ложь, передаваемое лямбда-функции для уточнения того, как должна выполняться работа.
приведение типа (casting)	Выполнение операций с объектом так, как если бы он был экземпляром другого типа.
проверка типов (type checking)	Подтверждение компилятором, что присвоенное переменной значение относится к правильному типу.
проверка типов, статическая (type checking, static)	Проверка типов, производимая в процессе ввода или компиляции кода (а не во время выполнения).
программный поток (thread)	Низкоуровневый компонент, выполняющий инструкции приложения в том порядке, в котором они записаны. См. <i>сопрограмма</i> .
программный поток, главный (thread, main)	Программный поток, управляющий представлением работы для пользователя; также называется UI-поток.
проект (project)	Весь исходный код программы, вместе с информацией о зависимостях и конфигурации.
производитель (producer)	Класс с обобщенным типом, который используется только для вывода данных класса, но не для ввода.

пул потоков (thread pool)	Набор потоков, доступных для распределения работы. См. <i>сопрограмма, программный поток</i> .
равенство, ссылок (equality, referential)	Две переменные считаются равными, если они ссылаются на один и тот же экземпляр. См. <i>равенство, структурное</i> .
равенство, структурное (equality, structural)	Две переменные считаются равными, если они содержат одинаковые значения. См. <i>равенство, ссылок</i> .
расширение (extension)	Свойство или функция, добавленные к объекту без применения наследования.
расширить (extend)	Добавить новую функциональность через наследование или реализацию интерфейса.
регулярное выражение (regular expression, regex)	Шаблон для поиска в тексте.
редактор (editor)	Главная область окна IntelliJ IDEA, в которой можно вводить и редактировать код.
рефакторинг (refactor)	Реорганизация кода без изменения его функциональности.
свойство класса (class property)	Атрибут класса, представляющий состояние или характеристики объекта.
свойство, встроенное (property, inline)	Свойство класса, определенное в главном конструкторе.
свойство, вычисляемое (property, computed)	Свойство, значение которого вычисляется при каждом обращении к нему.
сеттер, set-метод (setter)	Функция, определяющая, каким образом свойству присваивается значение.
символ Юникода (Unicode character)	Символ, определенный в стандарте Юникод.
синглтон (singleton)	Объект, объявленный с ключевым словом <code>object</code> ; синглтон ограничен одним экземпляром на протяжении выполнения программы.

система типов, статическая (type system, static)	Система, в которой компилятор помечает исходный код информацией о типах для проверки.
служебная последовательность (escape sequence)	Последовательность символов, экранированная для того, чтобы она имела смысл, отличный от содержащихся в ней литеральных символов.
сопрограмма (coroutine)	Возможность Kotlin, которая позволяет выполнять работу параллельно.
состояние гонки (race condition)	Условие, которое возникает, если поведение программы одновременно изменяют несколько элементов в этой программе.
ссылка на функцию (function reference)	Именованная функция, преобразованная в значение, которое может использоваться как функция-литерал.
стирание типов (type erasure)	Потеря информации о типе для обобщений во время выполнения программы.
стрелка, оператор (arrow operator)	Оператор, используемый в лямбда-выражениях для разделения параметров в теле функции, в выражении <code>when</code> — для разделения условий и результатов и в определениях функциональных типов — для разделения типов параметров от типов результатов: <code>(->)</code> .
строитель сопрограммы (coroutine builder)	Функция, создающая сопрограмму. См. <i>сопрограмма</i> .
строка (string)	Последовательность символов.
строка, необработанная (string, raw)	Синтаксис, позволяющий интерпретировать все символы и пробельные символы в строковом литерале в том виде, в котором они введены.
строковый шаблон (string template)	Синтаксис, позволяющий подставлять значения переменных внутри кавычек, ограничивающих строки.
суперкласс (superclass)	Класс, от которого наследуют подклассы.
тело класса (class body)	Часть определения класса в фигурных скобках, содержащая реализацию поведения и объявления данных.

тело функции (function body)	Часть объявления функции в фигурных скобках, определяющая поведение функции и возвращаемый тип.
тип (type)	Категория данных; тип переменной определяет значения, которые может содержать переменная.
тип, автоматическое определение (type inference)	Способность компилятора узнавать тип переменной в зависимости от присвоенного значения.
тип, возвращаемый (return type)	Тип выходных данных, которые возвращает функция после завершения работы.
тип данных, алгебраический (algebraic data type)	Тип, выражающий замкнутое множество возможных подтипов, например перечисляемый класс. См. <i>класс перечисления</i> ; <i>класс изолированный</i> .
тип, динамическая система (dynamic type system)	Система типов, в которой типы проверяются во время выполнения (а не во время компиляции).
тип, коллекции (type, collection)	Тип данных, представляющий группу элементов данных (например, список).
тип, обобщенный (generic type)	Тип, используемый классом или функцией, который задается в точке вызова.
тип-получатель (receiver type)	Тип, функциональность которого дополняется расширением.
только для чтения (read-only)	Возможность чтения без возможности записи. См. <i>изменяемость</i> .
точечный синтаксис (dot syntax)	Синтаксис, соединяющий два элемента через точку (.); используется при вызове функции, определяемой для типа, и при обращении к свойству класса.
точка входа в приложение (application entry point)	Место начала программы. В Kotlin это функция <code>main</code> .
умное приведение типа (smart casting)	Отслеживание компилятором информации, которая была проверена в условном выражении, например, имеет ли переменная значение <code>null</code> . См. <i>приведение типа</i> .

условное выражение (conditional expression)	Выражение с условием, присвоенное значению, которое может быть использовано позже.
форматная строка (format string)	Строка с заполнителями для подстановки данных. Заполнители могут включать правила форматирования для преобразования входных данных в текст.
функции стандартной библиотеки Kotlin (Kotlin standard library functions)	Набор функций, встроенных в язык программирования Kotlin.
функциональное программирование (functional programming)	Стиль программирования, который полагается на функции высшего порядка, спроектированные для работы с коллекциями и для вызова в цепочке с целью создания сложного поведения.
функциональный тип (function type)	Тип анонимной функции, объявленный своими входными/выходными данными и параметрами.
функция (function)	Фрагмент кода, который можно использовать многократно для решения определенной задачи.
функция класса (class function)	Функция, определенная внутри класса.
функция, абстрактная (function, abstract)	Функция без реализации, объявленная в абстрактном классе. См. класс, абстрактный.
функция, анонимная (function, anonymous)	Функция, объявленная без ключевого слова <code>fun</code> ; часто используется как аргумент для другой функции. См. <i>функция, именованная</i> ; <i>лямбда-функция</i> .
функция, вызов (function call)	Строка кода, которая запускает функцию и передает ей необходимые аргументы.
функция, вызов цепочкой (function call, chainable)	Вызов функции, возвращающий объект-получатель или иной объект, для которого можно вызвать следующую функцию.
функция, высшего порядка (function, higher-order)	Функция, которая получает другую функцию в аргументе или возвращает функцию как результат своей работы.

функция, заголовок (function header)	Часть объявления функции, которая содержит модификатор доступа, ключевое слово объявления функции, имя, параметры и возвращаемый тип.
функция, именованная (function, named)	Функция, объявленная с именем и ключевым словом <code>fun</code> . См. <i>функция, анонимная</i> .
функция, инфиксная (function, infix)	Функция, которая может вызываться без точки или без заключения аргумента в круглые скобки.
функция, итератор (function, iterator)	Функция, которая позволяет перебрать серию данных в определенном порядке.
функция, комбинатор (function, combining)	Функция, которая берет несколько коллекций и объединяет их в одну новую коллекцию.
функция, компокуемая (function, composable)	Функция, которую можно скомбинировать с другими функциями.
функция, литерал (function literal)	См. <i>лямбда-выражение</i> .
функция, мутатор (function, mutator)	Функция, которая изменяет содержимое изменяемой коллекции.
функция, предусловие (function, precondition)	Функция стандартной библиотеки Kotlin, определяющая условия, которые должны проверяться перед выполнением следующего кода.
функция, преобразователь (function, transform)	В функциональном программировании — функция, которая работает с содержимым коллекции, изменяя каждый элемент с помощью функции преобразования; функция-преобразователь возвращает измененную копию исходной коллекции. См. <i>функциональное программирование</i> .
функция, расширение (function, extension)	Функция, добавляющая новую функциональность в указанный тип.
функция, с одним выражением (function, single-expression)	Функция, тело которой состоит из одного выражения. См. <i>выражение</i> .

функция, фильтр
(function, filter)

Функция, которая работает с содержимым коллекции, применяя предикат для проверки каждого элемента на соответствие условию: элементы, удовлетворяющие условиям предиката, добавляются в новую коллекцию, возвращаемую фильтром.

число с плавающей точкой (floating point number)

Число, в котором разделитель дробной части может находиться в произвольной позиции в зависимости от количества значащих цифр.

числовой тип со знаком
(signed numeric type)

Числовой тип, который может содержать как положительные, так и отрицательные значения.

экземпляр (instance)

Конкретный представитель класса.

Джош Скин, Дэвид Гринхол, Эндрю Бэйли
Kotlin. Программирование для профессионалов.
2-е изд.

Перевел с английского Е. Матвеев

Заведующая редакцией	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>Е. Строганова</i>
Литературный редактор	<i>Ю. Леонова</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>С. Беляева, Н. Викторова</i>
Верстка	<i>Л. Соловьева</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сапсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 11.2022.

Наименование: книжная продукция.

Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск,
ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 29.09.22. Формат 70×100/16. Бумага офсетная.
Усл. п. л. 45,150. Тираж 700. Заказ 0000.



КУПИТЬ

Марк Лой, Патрик Нимайер, Дэн Лук

Программируем на Java. 5-е межд. изд.

Неважно, кто вы — разработчик ПО или пользователь, в любом случае вы слышали о языке Java. В этой книге вы на конкретных примерах изучите основы Java, API, библиотеки классов, приемы и идиомы программирования. Особое внимание авторы уделяют построению реальных приложений.

Вы освоите средства управления ресурсами и исключениями, а также познакомитесь с новыми возможностями языка, появившимися в последних версиях Java.

- Программируйте на Java с использованием компилятора, интерпретатора и других инструментов.
- Исследуйте средства управления потоками и параллельной обработки.
- Изучайте обработку текста и мощные API.
- Создавайте приложения и службы на базе современных сетевых коммуникаций или веб-технологий.



КУПИТЬ

Джейми Чан

Java: быстрый старт

Всегда хотели научиться программировать на Java, но не знаете, с чего начать? Или хотите быстро перейти с другого языка на Java?

Уже перепробовали множество книг и курсов, но ничего не подходит?

Серия «Быстрый старт» — отличное решение, и вот почему: сложные понятия разбиты на простые шаги — вы сможете освоить язык Java, даже если никогда раньше не занимались программированием; все фундаментальные концепции подкреплены реальными примерами; вы получите полное представление о Java: концепции объектно-ориентированного программирования, средства обработки ошибок, работа с файлами, лямбда-выражения и т. д.; в конце книги вас ждет интересный проект, который поможет усвоить полученные знания.

Ну что, готовы? Погнали!