O'REILLY®

Head First Kotlin

Руководство для начинающих программистов

Как избежать идиотских ошибок в лямбда-выражениях

Пишем функции высшего порядка не-от-мира-сего

Все, что вы хотели знать о дженериках

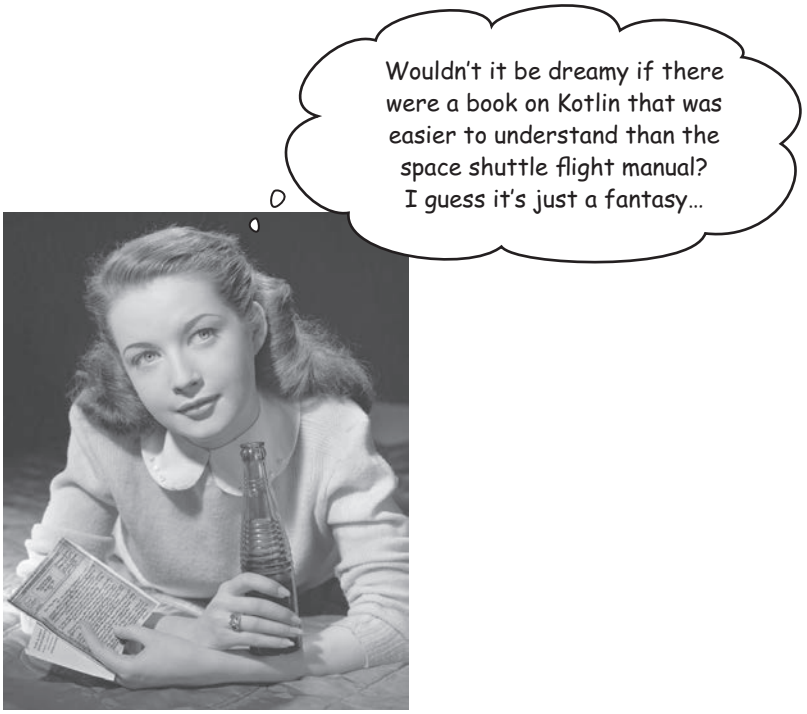
Как Элвис может изменить вашу жизнь

Коллекции под микроскопом

Развлечения с Kotlin Standard Library

Дон Гриффитс и Дэвид Гриффитс

Head First
Kotlin



Dawn Griffiths
David Griffiths

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Head First

Kotlin

Дон Гриффитс
Дэвид Гриффитс

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск
2020

Дон Гриффитс, Дэвид Гриффитс

Head First. Kotlin

Перевел с английского *Е. Матвеев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*А. Руденко*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Б. Файзуллин*
Верстка	*Н. Лукьянова*

ББК 32.973.2-018.1
УДК 004.43

Гриффитс Дон, Гриффитс Дэвид

Г85 Head First. Kotlin. — СПб.: Питер, 2020. — 464 с.: ил. — (Серия «Head First O'Reilly»).

ISBN 978-5-4461-1335-4

Вот и настало время изучить Kotlin. В этом вам поможет уникальная методика Head First, выходящая за рамки синтаксиса и инструкций по решению конкретных задач. Хотите мыслить, как выдающиеся разработчики Kotlin? Эта книга даст вам все необходимое — от азов языка до продвинутых методов. А еще вы сможете попрактиковаться в объектно-ориентированном и функциональном программировании. Если вы действительно хотите понять, как устроен Kotlin, то эта книга для вас!

Почему эта книга не похожа на другие? Подход Head First основан на новейших исследованиях в области когнитивистики и теории обучения. Визуальный формат позволяет вовлечь в обучение мозг читателя лучше, чем длинный текст, который вгоняет в сон. Зачем тратить время на борьбу с новыми концепциями? Head First задействует разные каналы получения информации и разрабатывался с учетом особенностей работы вашего мозга.

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

ISBN 978-1491996690 англ.

ISBN 978-5-4461-1335-4

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 09.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 — Книги печатные профессиональные, технические и научные.

Подписано в печать 19.09.19. Формат 84×108/16. Бумага писчая. Усл. п. л. 48,720. Тираж 1500. Заказ 0000.

Содержание (сводка)

	Введение	21
1	Первые шаги. *Не теряя времени*	31
2	Базовые типы и переменные. *Из жизни переменных*	61
3	Функции. *За пределами main*	89
4	Классы и объекты. *Высокий класс*	121
5	Подклассы и суперклассы. *Наследование*	121
6	Абстрактные классы и интерфейсы. *Серьезно о полиморфизме*	185
7	Классы данных. *Работа с данными*	221
8	Null и исключения. *В целости и сохранности*	249
9	Коллекции. *Порядок превыше всего*	281
10	Обобщения. *На каждый вход знай свой выход*	319
11	Лямбда-выражения и функции высшего порядка. *Обработка кода как данных*	325
12	Встроенные функции высшего порядка. *Расширенные возможности*	393
Приложение I. Сопрограммы. *Параллельный запуск*		427
Приложение II. Тестирование. *Код под контролем*		439
Приложение III. Остатки. *Топ-10 тем, которые мы не рассмотрели*		445

Содержание (настоящее)

Введение

Ваш мозг и Kotlin. Вы сидите за книгой и пытаетесь что-нибудь выучить, но ваш мозг считает, что вся эта писанина не нужна. Мозг говорит: «Выгляни в окно! На свете есть более важные вещи, например сноуборд». Как заставить мозг изучить программирование на Kotlin?

Для кого написана эта книга?	22
Мы знаем, о чем вы думаете	23
И мы знаем, о чем думает ваш *мозг*	23
Метапознание: наука о мышлении	25
Вот что сделали МЫ	26
Примите к сведению	28
Научные редакторы	29
Благодарности	30

1 Первые шаги

Не теряя времени

Kotlin впечатляет. С выхода самой первой версии Kotlin впечатляет программистов своим ***удобным синтаксисом, компактностью, гибкостью и мощью***. В этой книге мы научим вас **строить собственные приложения Kotlin**, а для начала покажем, как построить простейшее приложение и запустить его. Попутно вы познакомитесь с базовыми элементами синтаксиса Kotlin: *командами*, *циклами* и *условными конструкциями*. Приготовьтесь, путешествие начинается!

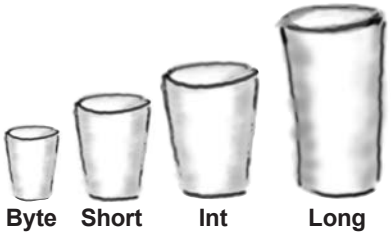
Раз вы можете выбирать платформу, для которой должен компилироваться код, это означает, что ваш код Kotlin может выполняться на серверах, в браузерах, на мобильных устройствах и т. д.

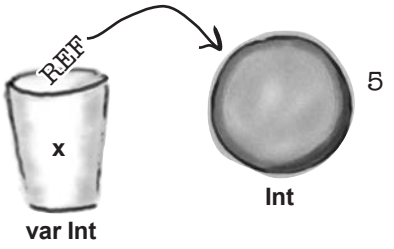
Добро пожаловать в Котлинвиль	32
Kotlin может использоваться практически везде	33
Чем мы займемся в этой главе	34
Установка IntelliJ IDEA (Community Edition)	37
Построение простейшего приложения	38
Вы только что создали свой первый проект Kotlin	41
Включение файла Kotlin в проект	42
Анатомия функции main	43
Добавление функции main в файл App.kt	44
Тест-драйв	45
Что можно сделать в функции main?	46
Цикл, цикл, цикл...	47
Пример цикла	48
Условные конструкции	49
Использование if для возвращения значения	50
Обновление функции main	51
Использование интерактивной оболочки Kotlin	53
В REPL можно вводить многострочные фрагменты	54
Путаница с сообщениями	57
Ваш инструментарий Kotlin	60

2 Базовые типы и переменные

Из жизни переменных

От переменных зависит весь ваш код. В этой главе мы заглянем «под капот» и покажем, ***как на самом деле работают переменные Kotlin***. Вы познакомитесь с **базовыми типами**, такими как *Int*, *Float* и *Boolean*, и узнаете, что компилятор Kotlin способен **вычислить тип переменной по присвоенному ей значению.** Вы научитесь пользоваться **строковыми шаблонами** для построения сложных строк с минимумом кода, и узнаете, как создать **массивы** для хранения нескольких значений. Напоследок мы расскажем, *почему объекты играют такую важную роль в программировании Kotlin*.





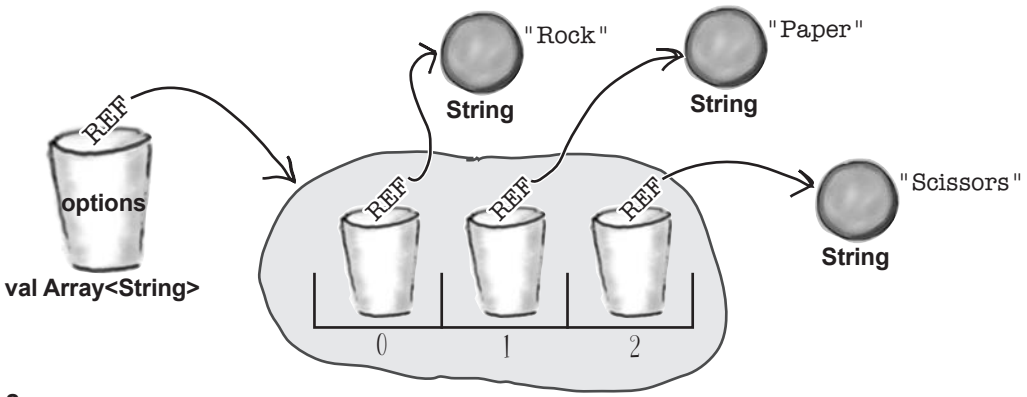
Без переменных не обойтись 62
Что происходит при объявлении переменной 63
В переменной хранится ссылка на объект 64
Базовые типы Kotlin 65
Как явно объявить тип переменной 67
Используйте значение, соответствующее типу переменной 68
Присваивание значения другой переменной 69
Значение необходимо преобразовать 70
Что происходит при преобразовании значений 71
Осторожнее с преобразованием 72
Сохранение значений в массивах 75
Построение приложения Phrase-O-Matic 76
Добавление кода в файл PhraseOMatic.kt 77
Компилятор определяет тип массива по значениям элементов 79
var означает, что переменная может указывать на другой массив 80
val означает, что переменная всегда будет указывать на один и тот же массив... 81
Путаница со ссылками 84
Ваш инструментарий Kotlin 88

3 Функции

За пределами main

А теперь пришло время поближе познакомиться с функциями. До сих пор весь написанный нами код размещался в функции *main* приложения. Но если вы хотите, чтобы код был **лучше структурирован** и **проще в сопровождении**, необходимо знать, ***как разбить его на отдельные функции***. В этой главе на примере игры вы научитесь *писать функции* и *взаимодействовать* с ними из приложения. Вы узнаете, как писать компактные ***функции единичных выражений***. Попутно вы научитесь ***перебирать диапазоны и коллекции*** в мощных циклах *for*.

Построение игры: камень-ножницы-бумага	90
Высокоуровневая структура игры	91
Выбор варианта игры	93
Как создаются функции	94
Функции можно передавать несколько значений	95
Получение значений из функции	96
Функции из единственного выражения	97
Добавление функции getGameChoice в файл Game.kt	98
Функция getUserChoice	105
Как работают циклы for	106
Запрос выбора пользователя	108
Проверка пользовательского ввода	111
Добавление функции getUserChoice в файл Game.kt	113
Добавление функции printResult в файл Game.kt	117
Ваш инструментарий Kotlin	119

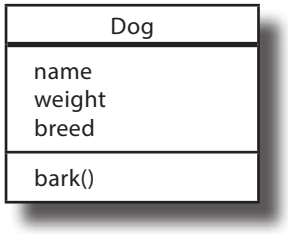


4 Классы и объекты

Высокий класс

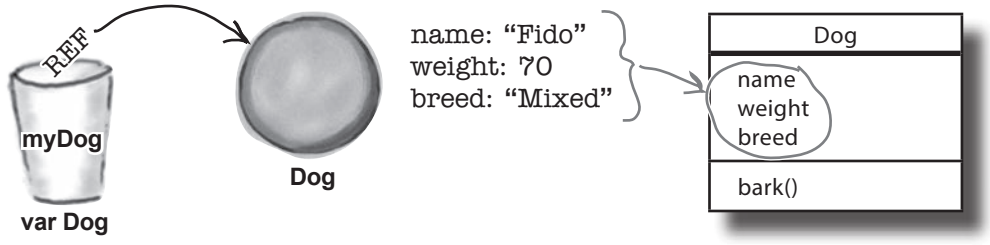
Пришло время выйти за границы базовых типов Kotlin. Рано или поздно базовых типов Kotlin вам станет *недостаточно*. И здесь на помощь приходят ***классы***. Классы представляют собой *шаблоны* для **создания ваших собственных типов объектов** и определения их свойств и функций. В этой главе вы научитесь **проектировать и определять классы,** а также использовать их для **создания новых типов объектов**. Вы познакомитесь с ***конструкторами***, ***блоками инициализации***, ***get-*** и ***set-***методами и научитесь использовать их для защиты свойств. В завершающей части вы узнаете о **средствах защиты данных, встроенных в *весь* код Kotlin**. Это сэкономит ваше время, силы и множество нажатий клавиш.

Один класс



Много объектов

Классы используются для определения типов объектов	122
Как спроектировать собственный класс	123
Определение класса Dog	124
Как создать объект Dog	125
Как обращаться к свойствам и функциям	126
Создание приложения Songs	127
Чудо создания объекта	128
Как создаются объекты	129
Под капотом: вызов конструктора Dog	130
Подробнее о свойствах	135
Гибкая инициализация свойств	136
Как использовать блоки инициализации	137
Свойства ДОЛЖНЫ инициализироваться	138
Как проверить значения свойств?	141
Как написать пользовательский get-метод	142
Как написать пользовательский set-метод	143
Полный код проекта Dogs	145
Ваш инструментарий Kotlin	150

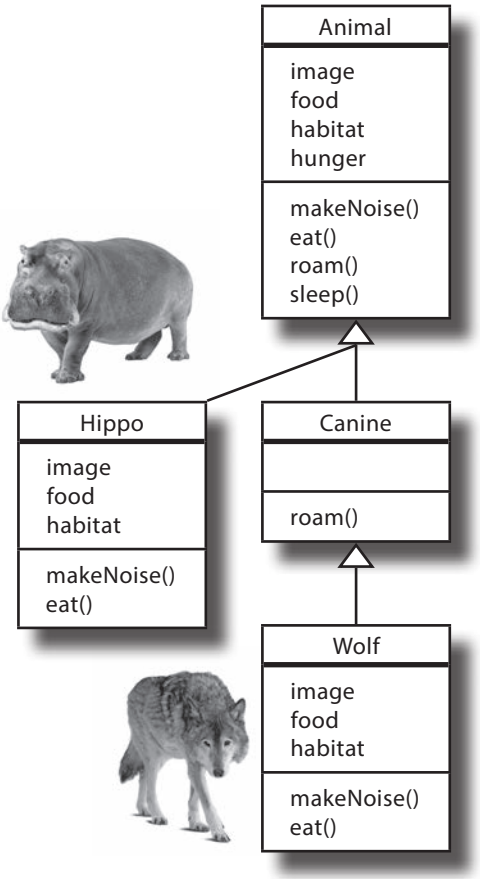


Подклассы и суперклассы

5 Наследование

Вам когда-нибудь казалось, что если немного изменить тип объекта, то он идеально подойдет для ваших целей?

Что ж, это одно из преимуществ **наследования**. В этой главе вы научитесь создавать **подклассы** и наследовать свойства и функции **суперклассов**. Вы узнаете, ***как переопределять функции и свойства,*** чтобы классы работали так, как нужно *вам,* и когда это стоит (или не стоит) делать. Наконец, вы увидите, как наследование помогает **избежать дублирования кода**, и узнаете, как сделать код более гибким при помощи **полиморфизма**.

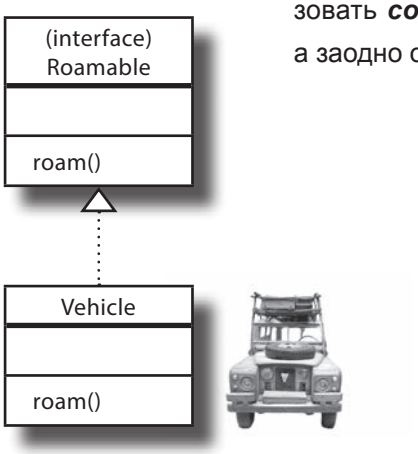


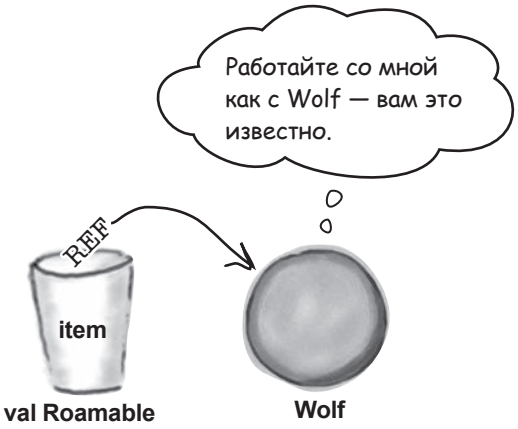
Наследование предотвращает дублирование кода	152
Что мы собираемся сделать	153
Проектирование структуры наследования классов	154
Используйте наследование для предотвращения дублирования кода в подклассах	155
Что должны переопределять подклассы?	156
Некоторых животных можно сгруппировать	157
Добавление классов Canine и Feline	158
Используйте «правило ЯВЛЯЕТСЯ» для проверки иерархии классов	159
Создание объектов животных в Kotlin	163
Объявление суперкласса и его свойств/функций с ключевым словом open	164
Как подкласс наследует от суперкласса	165
Как (и когда) переопределяются свойства	166
Возможности переопределения свойств не сводятся к присваиванию значений по умолчанию	167
Как переопределять функции	168
Переопределенная функция или свойство остаются открытыми...	169
Добавление класса Hippo в проект Animals	170
Добавление классов Canine и Wolf	173
Какая функция вызывается?	174
При вызове функции для переменной реагирует версия объекта	176
Супертип может использоваться для параметров и возвращаемого типа функции	177
Обновленный код Animals	178
Ваш инструментарий Kotlin	183

Абстрактные классы и интерфейсы

6 Серьезно о полиморфизме

Иерархия наследования суперклассов — только первый шаг. Чтобы ***в полной мере использовать возможности полиморфизма***, следует проектировать иерархии с **абстрактными классами** и **интерфейсами**. В этой главе вы узнаете, как при помощи абстрактных классов управлять тем, какие классы ***могут или не могут создаваться в вашей иерархии***. Вы увидите, как с их помощью заставить конкретные подклассы ***предоставлять собственные реализации***. В этой главе мы покажем, как при помощи интерфейсов организовать ***совместное использование поведения в независимых классах,*** а заодно опишем нюансы операторов ***is***, ***as*** и ***when***.





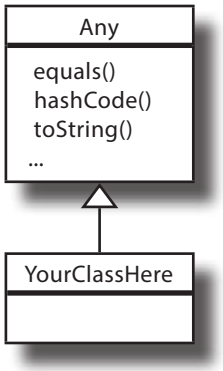
Снова об иерархии классов Animal	186
Некоторые классы не подходят для создания экземпляров	187
Абстрактный или конкретный?	188
Абстрактный класс может содержать абстрактные свойства и функции	189
Класс Animal содержит две абстрактные функции	190
Как реализовать абстрактный класс	192
Вы должны реализовать все абстрактные свойства и функции	193
Внесем изменения в код проекта Animals	194
Независимые классы могут обладать общим поведением	199
Интерфейс позволяет определить общее поведение за пределами иерархии суперкласса	200
Определение интерфейса Roamable	201
Как определяются свойства интерфейсов	202
Объявите о том, что класс реализует интерфейс...	203
Реализация нескольких интерфейсов	204
Класс, подкласс, абстрактный класс или интерфейс?	205
Обновление проекта Animals	206
Интерфейсы позволяют использовать полиморфизм	211
Когда используется оператор is	212
Оператор when проверяет переменную по нескольким вариантам	213
Оператор is выполняет умное приведение типа	214
Используйте as для выполнения явного приведения типа	215
Обновление проекта Animals	216
Ваш инструментарий Kotlin	219

7

Классы данных

Работа с данными

Никому не хочется тратить время и заново делать то, что уже было сделано. В большинстве приложений используются классы, предназначенные для *хранения данных*. Чтобы упростить работу, создатели Kotlin предложили концепцию **класса данных**. В этой главе вы узнаете, как классы данных помогают писать более ***элегантный и лаконичный*** код, о котором раньше можно было только мечтать. Мы рассмотрим ***вспомогательные функции*** классов данных и узнаем, как ***разложить объект данных на компоненты***. Заодно расскажем, как ***значения параметров по умолчанию*** делают код более гибким, а также познакомим вас с **Any** — предком всех суперклассов.



По умолчанию функция equals проверяет, являются ли два объекта одним фактическим объектом.

Оператор == вызывает функцию с именем equals	222
equals наследуется от суперкласса Any	223
Общее поведение, наследуемое от Any	224
Простая проверка эквивалентности двух объектов	225
Класс данных позволяет создавать объекты данных	226
Объекты классов переопределяют свое унаследованное поведение	227
Копирование объектов данных функцией copy	228
Классы данных определяют функции componentN...	229
Создание проекта Recipes	231
Путаница с сообщениями	233
Сгенерированные функции используют только свойства, определенные в конструкторе	235
Инициализация многих свойств делает код громоздким	236
Как использовать значения по умолчанию из конструкторов	237
Функции тоже могут использовать значения по умолчанию	240
Перегрузка функций	241
Обновление проекта Recipes	242
Ваш инструментарий Kotlin	247

8 Null и исключения

В целости и сохранности

Все мечтают о безопасности кода, и, к счастью, она была заложена в основу языка Kotlin. В этой главе мы сначала покажем, что при использовании **null-совместимых типов** Kotlin вы *вряд ли когда-либо столкнетесь с исключениями NullPointerException за все время программирования на Kotlin*. Вы научитесь использовать *безопасные вызовы* и узнаете, как Элвис-оператор спасает от *всевозможных бед*. А когда мы разберемся с null, то вы сможете **выдавать и перехватывать исключения** как настоящий профессионал.

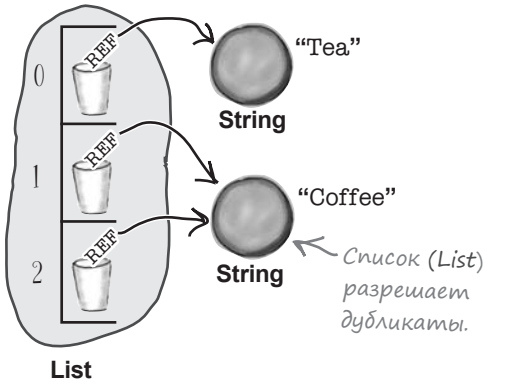
Как удалить ссылку на объект из переменной?	250
Удаление ссылки на объект с использованием null	251
null-совместимые типы могут использоваться везде, где могут использоваться не-null-совместимые	252
Как создать массив null-совместимых типов	253
Как обращаться к функциям и свойствам null-совместимых типов	254
Безопасные вызовы	255
Безопасные вызовы можно сцеплять	256
История продолжается...	257
Безопасные вызовы могут использоваться для присваивания...	258
Использование let для выполнения кода	261
Использование let с элементами массива	262
Вместо выражений if...	263
Оператор !! намеренно выдает исключение NullPointerException	264
Создание проекта Null Values	265
Исключения выдаются в исключительных обстоятельствах	269
Перехват исключений с использованием try/catch	270
finally и выполнение операций, которые должны выполняться всегда	271
Исключение — объект типа Exception	272
Намеренная выдача исключений	274
try и throw являются выражениями	275
Ваш инструментарий Kotlin	280

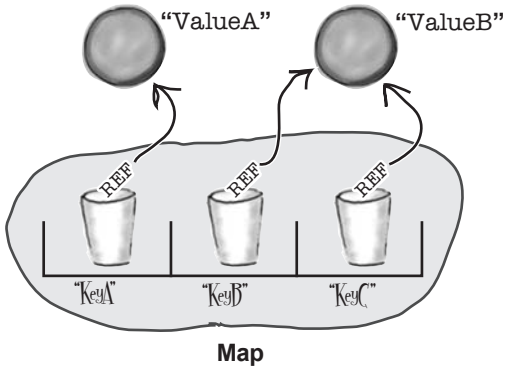
9

Коллекции

Порядок превыше всего

Хотели бы вы иметь структуру данных более гибкую, чем массив? Kotlin содержит подборку удобных **коллекций**, гибких и предоставляющих больше возможностей для управления ***хранением и управлением группами объектов***. Хотите *список с автоматически изменяемым размером, к которому можно добавлять новые элементы снова и снова?* С возможностью *сортировки, перетасовки или перестановки содержимого в обратном порядке*? Или хотите структуру данных, которая автоматически *уничтожает дубликаты* без малейших усилий с вашей стороны? Если вас заинтересовало все это (а также многое другое) — продолжайте читать.





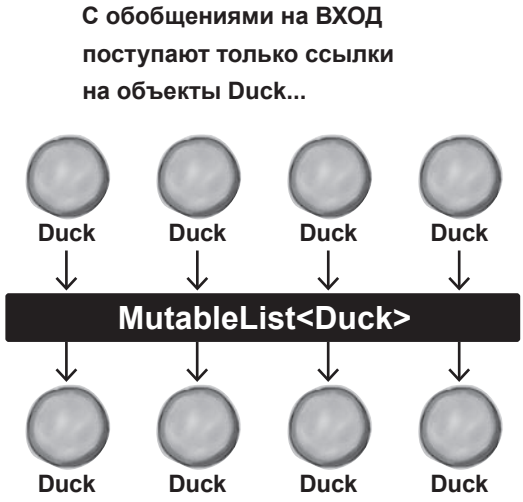
Ассоциативный массив (Map) разрешает дубликаты значений, но не дубликаты ключей.

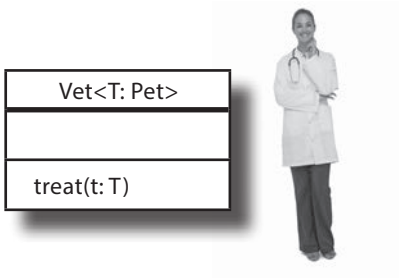
Массивы полезны...	282
...но с некоторыми задачами не справляются	283
Не уверены — обращайтесь в библиотеку	284
List, Set и Map	285
Эти невероятные списки...	286
Создайте объект MutableList...	287
Значения можно удалять...	288
Можно изменять порядок и вносить массовые изменения...	289
Создание проекта Collections	290
List позволяет дублировать значения	293
Как создать множество Set	294
Как Set проверяет наличие дубликатов	295
Хеш-коды и равенство	296
Правила переопределения hashCode и equals	297
Как использовать MutableSet	298
Копирование MutableSet	299
Обновление проекта Collections	300
Ассоциативные массивы Map	306
Как использовать Map	307
Создание MutableMap	308
Удаление элементов из MutableMap	309
Копирование Map и MutableMap	310
Полный код проекта Collections	311
Путаница с сообщениями	315
Ваш инструментарий Kotlin	317

10 Обобщения

На каждый вход знай свой выход

Всем нравится понятный и предсказуемый код. А один из способов написания универсального кода, в котором реже возникают проблемы, заключается в использовании ***обобщений***. В этой главе мы покажем, как **классы коллекций Kotlin используют обобщения**, чтобы вы не смешивали салат и машинное мало. Вы узнаете, как и в каких случаях ***писать собственные обобщенные классы, интерфейсы и функции*** и как ограничить **обобщенный тип** конкретным супертипом. Наконец, научитесь пользоваться ***ковариантностью и контрвариантностью***, чтобы ВЫ сами управляли поведением своего обобщенного типа.



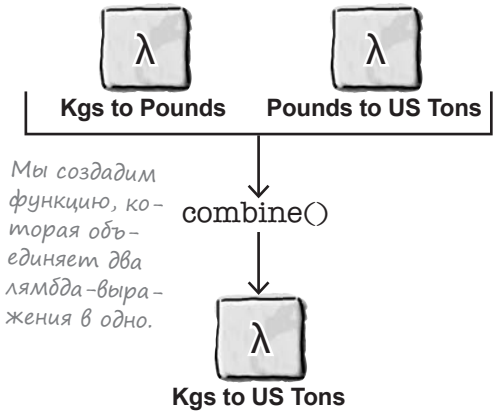


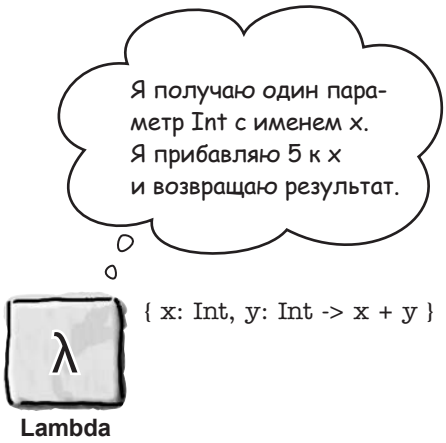
В коллекциях используются обобщения	320
Как определяется MutableList	321
Использование параметров типа с MutableList	322
Что можно делать с обобщенным классом или интерфейсом	323
Что мы собираемся сделать	324
Создание иерархии классов Pet	325
Определение класса Contest	326
Добавление свойства scores	327
Создание функции getWinners	328
Создание объектов Contest	329
Создание проекта Generics	331
Иерархия Retailer	335
Определение интерфейса Retailer	336
Мы можем создать объекты CatRetailer, DogRetailer и FishRetailer...	337
out и ковариантность обобщенного типа	338
Обновление проекта Generics	339
Класс Vet	343
Создание объектов Vet	344
in и контрвариантность обобщенных типов	345
Обобщенный тип может обладать локальной контрвариантностью	346
Обновление проекта Generics	347
Ваш инструментарий Kotlin	354

11 Лямбда-выражения и функции высшего порядка

Обработка кода как данных

Хотите писать еще более гибкий и мощный код? Тогда вам понадобятся **лямбда-выражения**. *Лямбда-выражение,* или просто *лямбда,* представляет собой блок кода, который можно передавать как объект. В этой главе вы узнаете, ***как определить лямбда-выражение***, ***присвоить его переменной***, а затем ***выполнить его код***. Вы узнаете о **функциональных типах** и о том, как они используются для написания **функций высшего порядка**, использующих лямбда-выражения для параметров или возвращаемых значений. А попутно вы узнаете, как ***синтаксический сахар подсластит вашу программистскую жизнь.***



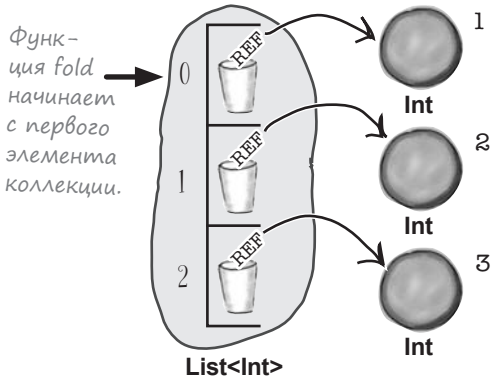


Знакомство с лямбда-выражениями	356
Как выглядит код лямбда-выражения	357
Присваивание лямбд переменной	358
Что происходит при выполнении лямбда-выражений	359
История продолжается...	360
У лямбда-выражений есть тип	361
Компилятор может автоматически определять типы параметров лямбда-выражений	362
Используйте лямбда-выражение, соответствующее типу переменной	363
Создание проекта Lambdas	364
Путаница с сообщениями	365
Лямбда-выражение может передаваться функции	369
Выполнение лямбда-выражения в теле функции	370
Что происходит при вызове функции	371
Лямбда-выражение можно вынести ЗА СКОБКИ...	373
Обновление проекта Lambdas	374
Функция может возвращать лямбда-выражение	377
Написание функции, которая получает и возвращает лямбда-выражения	378
Как использовать функцию combine	379
typealias и назначение альтернативного имени для существующего типа	383
Обновление проекта Lambdas	384
Ваш инструментарий Kotlin	391

12 Встроенные функции высшего порядка

Расширенные возможности

Kotlin содержит подборку встроенных функций высшего порядка. В этой главе представлены некоторые полезные функции этой категории. Вы познакомитесь с гибкими ***фильтрами*** и узнаете, как они используются для сокращения размера коллекции. Научитесь ***преобразовывать коллекции функцией map***, ***перебирать их элементы в forEach***, а также ***группировать элементы коллекций функцией groupBy***. Мы покажем, как использовать ***fold*** для выполнения сложных вычислений *всего в одной строке кода*. К концу этой главы вы научитесь писать **мощный** код, о котором и не мечтали.



Kotlin содержит подборку встроенных функций высшего порядка	394
Функции min и max работают с базовыми типами	395
Лямбда-параметр minBy и maxBy	396
Функции sumBy и sumByDouble	397
Создание проекта Groceries	398
Функция filter	401
Функция map и преобразования коллекций	402
Что происходит при выполнении кода	403
forEach работает как цикл for	405
У forEach нет возвращаемого значения	406
Обновление проекта Groceries	407
Функция groupBy используется для разбиения коллекции на группы	411
Функция groupBy может использоваться в цепочках вызовов	412
Как использовать функцию fold	413
За кулисами: функция fold	414
Примеры использования fold	416
Обновление проекта Groceries	417
Путаница с сообщениями	421
Ваш инструментарий Kotlin	424
Пара слов на прощанье...	425

I Сопрограммы

Параллельный запуск

Некоторые задачи лучше выполнять в фоновом режиме. Если вы загружаете данные с медленного внешнего сервера, то вряд ли захотите, чтобы остальной код простаивал и дожидался завершения загрузки. В подобных ситуациях **на помощь приходят сопрограммы**. Сопрограммы позволяют писать код, предназначенный для ***асинхронного выполнения***. А это означает *сокращение времени простоя, более удобное взаимодействие с пользователем и улучшенная масштабируемость приложений*. Продолжайте читать, и вы узнаете, как говорить с Бобом, одновременно слушая Сьюзи.

Бам! Бам! Бам! Бам! Бам! Бам! Дынц! Дынц!

Код воспроизводит звуковой файл toms шесть раз.

После этого звук тарелок воспроизводится дважды.

II Тестирование

Код под контролем

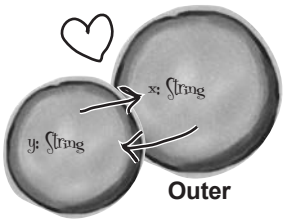
Хороший код должен работать, это все знают. Но при любых изменениях кода появляется опасность внесения новых ошибок, из-за которых код не будет работать так, как положено. Вот почему так важно провести *тщательное тестирование*: вы узнаете о любых проблемах в коде *до того, как он будет развернут в среде реальной эксплуатации*. В этом приложении рассматриваются ***JUnit*** и ***KotlinTest*** — библиотеки **модульного тестирования**, которые дадут вам *дополнительные гарантии безопасности*.

III

Остатки

Топ-10 тем, которые мы не рассмотрели

Но и это еще не все. Осталось еще несколько тем, о которых, как нам кажется, вам следует знать. Делать вид, что их не существует, было бы неправильно — как, впрочем, и выпускать книгу, которую поднимет разве что культурист. Прежде чем откладывать книгу, ознакомьтесь с этими **лакомыми кусочками**, которые мы оставили напоследок.

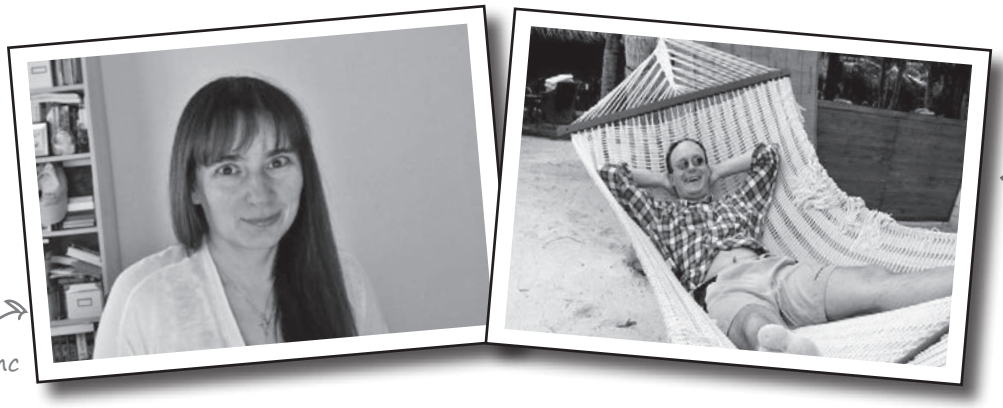


Объекты Inner и Outer объединены связью особого рода. Inner может использовать переменные Outer, и наоборот.

1. Пакеты и импортирование 446
2. Модификаторы видимости 448
3. Классы перечислений 450
4. Изолированные классы 452
5. Вложенные и внутренние классы 454
6. Объявления объектов и выражения 456
7. Расширения 459
8. Return, break и continue 460
9. Дополнительные возможности функций 462
10. Совместимость 464

Посвящается светлым умам разработчиков Kotlin, которые создали такой замечательный язык программирования.

Об авторах



Дон Гриффитс уже более 20 лет работает в IT-отрасли на должностях старшего разработчика и старшего архитектора ПО.

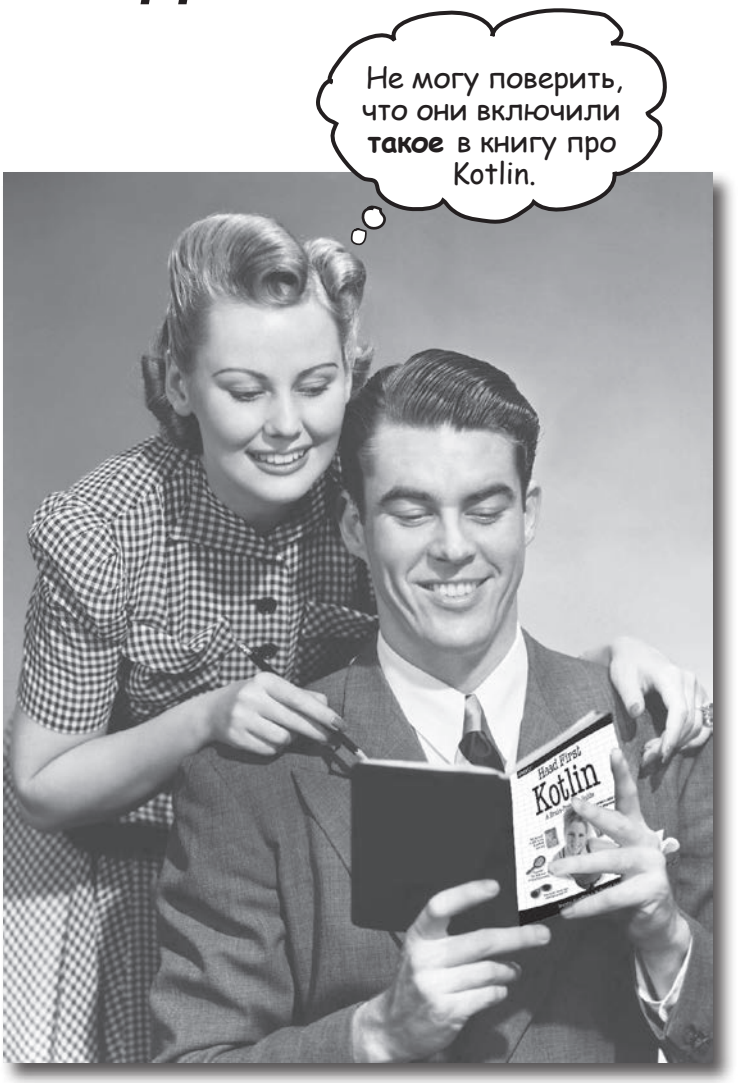
Она написала несколько книг из серии *Head First*, включая *Head First. Программирование для Android*. Кроме того, со своим мужем Дэвидом разработала анимационный видеокурс *The Agile Sketchpad* — метод представления ключевых концепций и приемов, который бы обеспечивал максимальную активность мозга и его непосредственное участие в процессе обучения.

Когда Дон не пишет книги и не снимает видео, она обычно совершенствует свое мастерство тайцзи, увлекается чтением, бегом, плетением кружев и кулинарией. Но больше всего ей нравится проводить время с ее замечательным мужем Дэвидом.

Дэвид Гриффитс работал Agile-коучем, разработчиком и рабочим на автозаправке (хотя и немного в другом порядке). Увлекся программированием в 12 лет, после просмотра документального фильма о работе Сеймура Пейперта. В 15 лет он написал реализацию языка программирования LOGO, созданного Пейпертом. Прежде чем браться за книгу *Head First Kotlin*, Дэвид написал еще несколько книг из серии *Head First*, включая *Head First. Программирование для Android*, и создал видеокурс *The Agile Sketchpad* при участии Дон. Когда он не занят программированием, литературной работой или преподаванием, проводит время в путешествиях со своей очаровательной женой и соавтором — Дон.

Наши микроблоги Twitter доступны по адресу https://twitter.com/*HeadFirstKotlin*.

Введение


Не могу поверить, что они включили **такое** в книгу про Kotlin.
Head First
Kotlin

Для кого написана эта книга?

Если вы ответите «да» на все следующие вопросы:

1. У вас уже есть опыт программирования?
2. Вы хотите изучить Kotlin?
3. Вы предпочитаете заниматься практической работой и применять полученные знания вместо того, чтобы выслушивать нудные многочасовые лекции?

Это НЕ справочник. Head First Kotlin — книга для тех, кто хочет ***изучить*** *язык, а не энциклопедия всевозможных фактов о Kotlin.*

...тогда эта книга для вас.

Кому эта книга не подойдет?

Если вы ответите «да» хотя бы на один из следующих вопросов:

1. Ваш опыт программирования ограничивается одним HTML, а с языками сценариев вы дела не имели?

   (Если вы работали с циклами и логикой «if/then», все будет нормально, но одних тегов HTML будет недостаточно.)

2. Вы — крутой программист Kotlin, которому нужен *справочник*?

3. Вы скорее пойдете к зубному врачу, чем опробуете что-нибудь новое? Вы считаете, что в книге о Kotlin должно быть рассказано *все* — особенно экзотические возможности, которыми вы никогда не будете пользоваться, а если читатель будет помирать со скуки — еще лучше?

...эта книга *не* для вас.

[Примечание от отдела продаж: вообще-то эта книга для всех, у кого есть кредитка... и если что — PayPal тоже подойдет.]

Мы знаем, о чем вы думаете

«Разве серьезные книги о Kotlin *такие?*»

«И почему здесь столько рисунков?»

«Можно ли так чему-нибудь *научиться?*»

И мы знаем, о чем думает ваш мозг

Мозг жаждет новых впечатлений. Он постоянно ищет, анализирует, *ожидает* чего-то необычного. Он так устроен, и это помогает нам выжить.

Как мозг поступает со всеми обычными, повседневными вещами? Он всеми силами пытается оградиться от них, чтобы они не мешали его *настоящей* работе — сохранению того, что действительно *важно*. Мозг не считает нужным сохранять скучную информацию. Она не проходит фильтр, отсекающий «очевидно несущественное».

Но как же мозг *узнает*, что важно? Представьте, что вы выехали на прогулку и вдруг прямо перед вами появляется тигр. Что происходит в вашей голове и теле?

Активизируются нейроны. Вспыхивают эмоции. Происходят химические реакции. И тогда ваш мозг понимает...

Конечно, это важно! Не забывать!

А теперь представьте, что вы находитесь дома или в библи в теплом, уютном месте, где тигры не водятся. Вы учитесь — тесь к экзамену. Или пытаетесь освоить сложную техническ на которую вам выделили неделю... максимум десять дней.

И тут возникает проблема: ваш мозг пытается оказать вам услугу. Он старается сделать так, чтобы на эту *очевидно* несущественную информацию не тратились драгоценные ресурсы. Их лучше потратить на что-нибудь важное. На тигров например. Или на то, что к огню лучше не прикасаться. Или что те фотки с вечеринки не стоило выкладывать на Faceboo

Нет простого способа сказать своему мозгу: «Послушай, моз я тебе, конечно, благодарен, но какой бы скучной ни была эта книга и пусть мой датчик эмоций сейчас на нуле, я *хочу* запомнить то, что здесь написано».

Эта книга для тех, кто хочет учиться.

Как мы что-то узнаем? Сначала нужно это «что-то» *понять*, а потом *не забыть*. Затолкать в голову побольше фактов недостаточно. Согласно новейшим исследованиям в области когнитивистики, нейробиологии и психологии обучения, для *усвоения материала* требуется что-то большее, чем простой текст на странице. Мы знаем, как заставить ваш мозг работать.

Основные принципы серии «Head First»

Наглядность. Графика запоминается лучше, чем обычный текст, и значительно повышает эффективность восприятия информации (до 89 % по данным исследований). Кроме того, материал становится более понятным. **Текст размещается на рисунках**, к которым он относится, а не под ними или на соседней странице — и вероятность успешного решения задач, относящихся к материалу, повышается вдвое.

Разговорный стиль изложения. Недавние исследования показали, что при разговорном стиле изложения материала (вместо формальных лекций) улучшение результатов на итоговом тестировании достигает 40 %. Рассказывайте историю, вместо того чтобы читать лекцию. Не относитесь к себе слишком серьезно. Что привлечет ваше внимание: занимательная беседа за столом или лекция?

Активное участие читателя. Пока вы не начнете напрягать извилины, в вашей голове ничего не произойдет. Читатель должен быть заинтересован в результате; он должен решать задачи, формулировать выводы и овладевать новыми знаниями. А для этого необходимы упражнения и каверзные вопросы, в решении которых задействованы оба полушария мозга и разные чувства.

Привлечение (и сохранение) внимания читателя. Ситуация, знакомая каждому: «Я очень хочу изучить это, но засыпаю на первой странице». Мозг обращает внимание на интересное, странное, притягательное, неожиданное. Изучение сложной технической темы не обязано быть скучным. Интересное узнается намного быстрее.

Обращение к эмоциям. Известно, что наша способность запоминать в значительной мере зависит от эмоционального сопереживания. Мы запоминаем то, что нам небезразлично. Мы запоминаем, когда что-то чувствуем. Нет, сантименты здесь ни при чем: речь идет о таких эмоциях, как удивление, любопытство, интерес и чувство «Да я крут!» при решении задачи, которую окружающие считают сложной, или когда вы понимаете, что разбираетесь в теме лучше, чем всезнайка Боб из технического отдела.

Метапознание: наука о мышлении

Если вы действительно хотите быстрее и глубже усваивать новые знания — задумайтесь над тем, как вы думаете. Учитесь учиться.

Мало кто из нас изучает теорию метапознания во время учебы. Нам *положено* учиться, но нас редко этому *учат*.

Но раз вы читаете эту книгу, то, вероятно, хотите освоить программирование на Kotlin, и по возможности быстрее. Вы хотите *запомнить* прочитанное, а для этого абсолютно необходимо сначала *понять* прочитанное.

Чтобы извлечь максимум пользы из учебного процесса, нужно заставить ваш мозг воспринимать новый материал как Нечто Важное. Критичное для вашего существования. Такое же важное, как тигр. Иначе вам предстоит бесконечная борьба с вашим мозгом, который всеми силами уклоняется от запоминания новой информации.

Как же УБЕДИТЬ мозг, что язык Kotlin не менее важен, чем голодный тигр?

Есть способ медленный и скучный, а есть быстрый и эффективный. Первый основан на тупом повторении. Всем известно, что даже самую скучную информацию можно запомнить, если повторять ее снова и снова. При достаточном количестве повторений ваш мозг прикидывает: «*Вроде бы* несущественно, но раз одно и то же повторяется *столько* раз... Ладно, уговорил».

Быстрый способ основан на **повышении активности мозга** и особенно на сочетании разных ее *видов*. Доказано, что все факторы, перечисленные на предыдущей странице, помогают вашему мозгу работать на вас. Например, исследования показали, что размещение слов *внутри* рисунков (а не в подписях, в основном тексте и т. д.) заставляет мозг анализировать связи между текстом и графикой, а это приводит к активизации большего количества нейронов. Больше нейронов — выше вероятность того, что информация будет сочтена важной и достойной запоминания.

Разговорный стиль тоже важен: обычно люди проявляют больше внимания, когда они участвуют в разговоре, так как им приходится следить за ходом беседы и высказывать свое мнение. Причем мозг совершенно не интересует, что вы «разговариваете» с книгой! С другой стороны, если текст сух и формален, то мозг чувствует то же, что чувствуете вы на скучной лекции в роли пассивного участника. Его клонит в сон.

Но рисунки и разговорный стиль — это только начало.

Вот что сделали МЫ

Мы использовали ***рисунки***, потому что мозг лучше приспособлен для восприятия графики, чем текста. С точки зрения мозга рисунок стоит тысячи слов. А когда текст комбинируется с графикой, мы внедряем текст прямо в рисунки, потому что мозг при этом работает эффективнее.

Мы используем ***избыточность***: повторяем одно и то же несколько раз, применяя разные средства передачи информации, обращаемся к разным чувствам — и все для повышения вероятности того, что материал будет закодирован в нескольких областях вашего мозга.

Мы используем концепции и рисунки несколько ***неожиданным*** образом, потому что мозг лучше воспринимает новую информацию. Кроме того, рисунки и идеи обычно имеют ***эмоциональное содержание***, потому что мозг обращает внимание на биохимию эмоций. То, что заставляет нас *чувствовать*, лучше запоминается — будь то ***шутка***, ***удивление*** или ***интерес***.

Мы используем ***разговорный стиль***, потому что мозг лучше воспринимает информацию, когда вы участвуете в разговоре, а не пассивно слушаете лекцию. Это происходит и при *чтении*.

В книгу включены многочисленные ***упражнения***, потому что мозг лучше запоминает, когда вы что-то делаете. Мы постарались сделать их непростыми, но интересными — то, что предпочитает большинство читателей.

Мы совместили ***несколько стилей обучения***, потому что одни читатели предпочитают пошаговые описания, другие стремятся сначала представить «общую картину», а третьим хватает фрагмента кода. Независимо от ваших личных предпочтений полезно видеть несколько вариантов представления одного материала.

Мы постарались задействовать ***оба полушария вашего мозга***; это повышает вероятность усвоения материала. Пока одна сторона мозга работает, другая часто имеет возможность отдохнуть; это повышает эффективность обучения в течение продолжительного времени.

А еще в книгу включены ***истории*** и упражнения, отражающие другие точки зрения. Мозг глубже усваивает информацию, когда ему приходится оценивать и выносить суждения.

В книге часто встречаются ***вопросы***, на которые не всегда можно дать простой ответ, потому что мозг быстрее учится и запоминает, когда ему приходится что-то делать. Невозможно накачать ***мышцы***, наблюдая за тем, как занимаются ***другие***. Однако мы позаботились о том, чтобы усилия читателей были приложены в ***верном*** направлении. Вам не придется ломать голову над невразумительными примерами или разбираться в сложном, перенасыщенном техническим жаргоном или слишком лаконичном тексте.

В историях, примерах, на картинках используются ***люди*** — потому что вы тоже ***человек***. И ваш мозг обращает на людей больше внимания, чем на неодушевленные ***предметы***.

Вырежьте и прикрепите на холодильник.

Что можете сделать ВЫ, чтобы заставить свой мозг повиноваться

Мы свое дело сделали. Остальное за вами. Эти советы станут отправной точкой; прислушайтесь к своему мозгу и определите, что вам подходит, а что не подходит. Пробуйте новое.

1 Не торопитесь. Чем больше вы поймете, тем меньше придется запоминать.

Просто читать недостаточно. Когда книга задает вам вопрос, не переходите к ответу. Представьте, что кто-то *действительно* задает вам вопрос. Чем глубже ваш мозг будет мыслить, тем скорее вы поймете и запомните материал.

2 Выполняйте упражнения, делайте заметки.

Мы включили упражнения в книгу, но выполнять их за вас не собираемся. И не *разглядывайте* упражнения. **Берите карандаш и пишите**. Физические действия *во время* учения повышают его эффективность.

3 Читайте врезки.

Это значит: читайте всё. ***Врезки – часть основного материала!*** Не пропускайте их.

4 Не читайте другие книги после этой перед сном.

Часть обучения (особенно перенос информации в долгосрочную память) происходит *после* того, как вы откладываете книгу. Ваш мозг не сразу усваивает информацию. Если во время обработки поступит новая информация, часть того, что вы узнали ранее, может быть потеряна.

5 Говорите вслух.

Речь активизирует другие участки мозга. Если вы пытаетесь что-то понять или получше запомнить, произнесите вслух. А еще лучше, попробуйте объяснить кому-нибудь другому. Вы будете быстрее усваивать материал и, возможно, откроете для себя что-то новое.

6 Пейте воду. И побольше.

Мозг лучше всего работает в условиях высокой влажности. Дегидратация (которая может наступить еще до того, как вы почувствуете жажду) снижает когнитивные функции.

7 Прислушивайтесь к своему мозгу.

Следите за тем, когда ваш мозг начинает уставать. Если вы начинаете поверхностно воспринимать материал или забываете только что прочитанное, пора сделать перерыв.

8 Чувствуйте!

Ваш мозг должен знать, что материал книги действительно *важен*. Переживайте за героев наших историй. Придумывайте собственные подписи к фотографиям. Поморщиться над неудачной шуткой все равно *лучше*, чем не почувствовать ничего.

9 Пишите побольше кода!

Освоить программирование Kotlin можно только одним способом: **писать побольше кода**. Именно этим мы и будем заниматься в книге. Программирование — искусство, и добиться мастерства в нем можно только практикой. Для этого у вас будут все возможности: в каждой главе приведены упражнения, в которых вам придется решать задачи. Не пропускайте их — работа над упражнениями является важной частью процесса обучения. К каждому упражнению приводится решение — не бойтесь **заглянуть** в него, если окажетесь в тупике! (Споткнуться можно даже о маленький камешек.) По крайней мере постарайтесь решить задачу, прежде чем заглядывать в решение. Обязательно добейтесь того, чтобы ваше решение заработало, прежде чем переходить к следующей части книги.

Примите к сведению

Это учебник, а не справочник. Мы намеренно убрали из книги все, что могло бы помешать изучению материала, над которым вы работаете. И при первом чтении начинать следует с самого начала, потому что книга предполагает наличие у читателя определенных знаний и опыта.

Предполагается, что у вас уже имеется некоторый опыт программирования.

Предполагается, что вы уже занимались программированием. Может, ваш опыт и не велик, но мы считаем, что вам уже знакомы такие конструкции, как циклы и переменные. И в отличие от многих книг о Kotlin, мы не предполагаем, что вы уже знаете Java.

Сначала мы излагаем основные концепции Kotlin, но затем мы начинаем сходу применять Kotlin на практике.

Основы программирования Kotlin рассматриваются в главе 1. К тому времени, когда вы перейдете к главе 2, вы уже будете создавать программы, которые делают что-то полезное. В оставшейся части книги ваши навыки Kotlin будут только расти, и из *новичка Kotlin* вы за минимальное время станете *мастером*.

Повторение применяется намеренно.

У книг этой серии есть одна принципиальная особенность: мы хотим, чтобы вы *действительно хорошо* усвоили материал. И чтобы вы запомнили все, что узнали. Большинство справочников не ставит своей целью успешное запоминание, но это не справочник, а *учебник*, поэтому некоторые концепции излагаются в книге по нескольку раз. Повторение применяется намеренно.

Мы постарались сделать примеры по возможности компактными.

Мы знаем, как неприятно продираться через 200 строк кода в поисках двух строк, которые нужно понять. Большинство примеров в этой книге представлено в минимально возможном контексте, чтобы та часть, которую вы изучаете, была простой и понятной. Не ждите, что этот код будет особенно надежным или хотя бы полным. Вы *сами* займетесь этим после прочтения книги, и все это является частью процесса обучения.

Упражнения ОБЯЗАТЕЛЬНЫ.

Упражнения являются частью основного материала книги. Одни упражнения способствуют запоминанию материала, другие помогают лучше понять его, третьи ориентированы на его практическое применение. ***Не пропускайте упражнения.***

Упражнения «Мозговой штурм» не имеют ответов.

В некоторых из них правильного ответа вообще нет, в других вы должны сами решить, насколько правильны ваши ответы (это является частью процесса обучения). В некоторых упражнениях «Мозговой штурм» даются подсказки, которые помогут вам найти нужное направление.

Научные редакторы

Научные редакторы

Инго Кротцки — квалифицированный специалист по обработке медицинской информации, работающий программистом баз данных/разработчиком программных продуктов для контрактных исследовательских организаций.

Кен Каусен — автор книг *Modern Java Recipes* (O'Reilly), *Gradle Recipes for Android* (O'Reilly) и *Making Java Groovy* (Manning), а также видеокурсов O'Reilly по изучению Android, Groovy, Gradle, нетривиальных возможностей Java и Spring. Регулярно выступает с докладами на конференциях по всему миру, в том числе: No Fluff, Just Stuff, JavaOne Rock Star в 2013 и 2016 годах. В своей компании Kousen I.T., Inc. проводил учебные курсы по разработке программного обеспечения для тысяч участников.

Благодарности

Нашему редактору:

Сердечное спасибо нашему редактору **Джеффу Блейлю** за его работу. Его доверие, поддержка и содействие были чрезвычайно полезными. Вы не представляете, сколько раз он указывал, что материал недостаточно понятен или нуждается в переосмыслении, — это помогло нам сделать книгу лучше.

Команде O'Reilly

Огромное спасибо **Брайану Фостеру** за его создействие на ранних стадиях; **Сьюзен Конант**, **Рэйчел Румелиотис** и **Нэнси Дэвис**, направлявших нас на правильный путь; **Рэнди Камер** за дизайн обложки; **команде предварительного выпуска** за подготовку ранних версий этой книги. Также мы хотим поблагодарить **Кристер Браун, Джасмин Квитин, Люси Хаскинс и остальных участников команды производства**, столь умело руководивших процессом и выполнявших огромную, хотя и незаметную на первый взгляд работу.

Семье, друзьям и коллегам:

Написание книги из серии Head First — затея непростая, и этот раз не стал исключением. Вероятно, эта книга не вышла бы в свет, если бы не доброта и поддержка наших друзей и родственников. Отдельное спасибо **Жаки**, **Иэну**, **Ванессе**, **Дон**, **Мэтту**, **Энди**, **Саймону**, **маме**, **папе**, **Робу** и **Лоррейн**.

Без кого эта книга не появилась бы:

Команда научных редакторов прекрасно справилась со своей задачей: поддерживать нас на правильном пути и обеспечить актуальность материала. Мы также благодарны всем, кто поделился своим мнением по поводу ранних версий книги. Как нам кажется, книга от этого стала намного, намного лучше.

И наконец, спасибо **Кэти Сьерра** и **Берту Бэйтсу** за создание этой замечательной серии книг.

Не теряя времени

Kotlin впечатляет. С выхода самой первой версии Kotlin впечатляет программистов своим ***удобным синтаксисом, компактностью, гибкостью и мощью***. В этой книге мы научим вас **строить собственные приложения Kotlin**, а для начала покажем, как построить простейшее приложение и запустить его. Попутно вы познакомитесь с базовыми элементами синтаксиса Kotlin: *командами*, *циклами* и *условными конструкциями*. Приготовьтесь, путешествие начинается!

Добро пожаловать в Котлинвиль

Kotlin произвел настоящий фурор в мире программирования. Хотя это один из самых молодых языков, многие разработчики остановили свой выбор на нем. Чем же Kotlin так выделяется на общем фоне?

Kotlin обладает многими особенностями современных языков, которые делают его привлекательным для разработчиков. Эти особенности более подробно рассматриваются позднее в книге, а пока ограничимся краткой сводкой.

Компактность, лаконичность и удобочитаемость

В отличие от некоторых языков, Kotlin чрезвычайно компактен — вы можете выполнять сложные операции всего в одной строке кода. Kotlin предоставляет сокращенную запись для основных операций, чтобы не приходилось писать повторяющийся шаблонный код, а также содержит богатую библиотеку функций, которыми вы можете пользоваться в своих программах. А чем меньше кода приходится просматривать, тем быстрее читаются и пишутся программы, тем быстрее вы понимаете их логику, и у вас остается больше времени для другой работы.

Объектно-ориентированное и функциональное программирование

Не можете решить, какое программирование изучать — объектно-ориентированное или функциональное? А почему бы не оба сразу? Kotlin позволяет создавать объектно-ориентированный код, в котором используются классы, наследование и полиморфизм, как и в языке Java. Но Kotlin также поддерживает функциональное программирование, и вы сможете пользоваться лучшими возможностями обеих парадигм.

Компилятор обеспечивает безопасность

Никому не нравится ненадежный код, кишащий ошибками. Компилятор Kotlin по максимуму старается очистить ваш код от ошибок, и предотвращает многие ошибки, встречающиеся в других языках. В частности, Kotlin является языком со статической типизацией, что не позволяет выполнять действия с переменной неподходящего типа, которые бы вызвали сбой в коде. И в большинстве случаев вам даже не обязательно явно задавать тип, потому что компилятор сможет определить его за вас.

Kotlin практически исключает возможность некоторых ошибок, постоянно встречающихся в других языках. Код получается более надежным и безопасным, а вам придется тратить меньше времени, отлавливая ошибки.

Итак, Kotlin — современный, мощный и гибкий язык программирования, обладающий многими преимуществами. Тем не менее это еще не все.

Kotlin может использоваться практически везде

Kotlin — язык настолько мощный и гибкий, что может использоваться как язык общего назначения во многих контекстах. Дело в том, что вы ***можете выбрать платформу, для которой должен компилироваться код Kotlin.***

Виртуальные машины Java (JVM)

Код Kotlin компилируется в байт-код JVM (Java Virtual Machine), поэтому Kotlin может использоваться практически везде, где может использоваться Java. Код Kotlin на 100% совместим с Java, а это значит, что в нем можно использовать существующие библиотеки Java. Если вы работаете над приложением, содержащим большой объем старого кода Java, вам не придется отказываться от всего старого кода — новый код Kotlin будет работать параллельно с ним. А если вы захотите использовать код Kotlin, написанный из кода Java, — это тоже возможно без малейших проблем.

Раз вы можете выбирать платформу, для которой должен компилироваться код, это означает, что код Kotlin может выполняться на серверах, в браузерах, на мобильных устройствах и т. д.

Android

Наряду с другими языками (например, Java) в Kotlin реализована полноценная поддержка Android. Kotlin полностью поддерживается средой Android Studio, поэтому вы сможете пользоваться многочисленными преимуществами Kotlin при разработке приложений на Android.

JavaScript на стороне клиента и на стороне сервера

Код Kotlin также может транспилироваться (то есть транслироваться на другой язык) на JavaScript, и вы можете выполнять его в браузере. В частности, это позволит вам использовать различные технологии на стороне клиента и на стороне сервера, такие как WebGL или Node.js.

Платформенные приложения

Если потребуется написать код, который будет быстро работать на менее мощных устройствах, код Kotlin можно откомпилировать непосредственно в машинный код конкретной платформы. В частности, это позволит вам писать код, выполняемый в iOS или Linux.

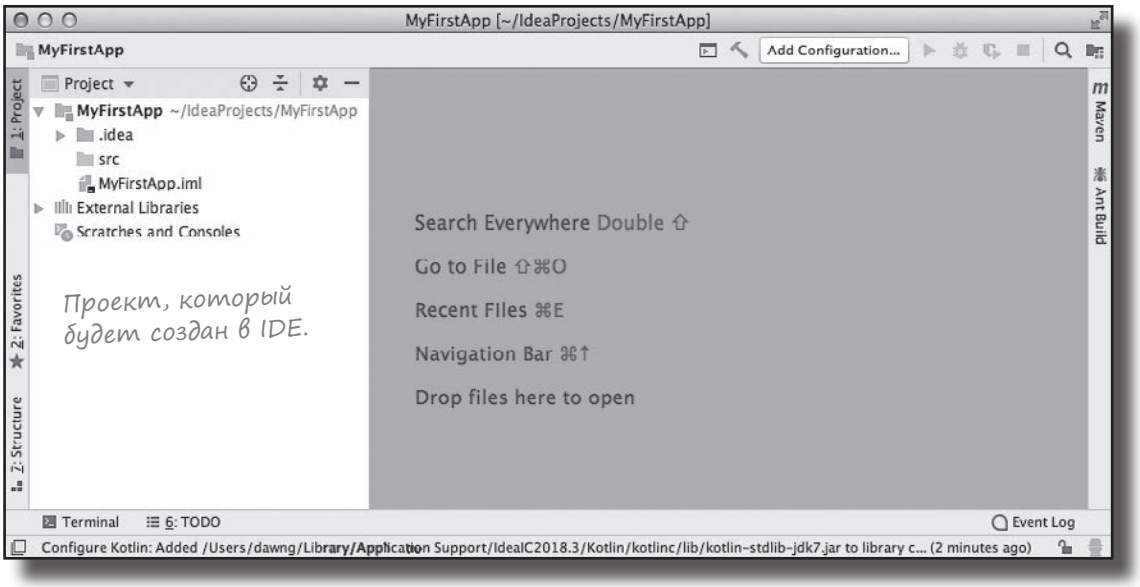
Хотя мы будем строить приложения для виртуальных машин Java, для чтения этой книги не обязательно знать Java. Предполагается, что у читателя есть некоторый опыт практического программирования и ничего более.

В этой книге мы сосредоточимся на создании приложений Kotlin для JVM, так как это позволит вам быстрее всего освоить язык. Позже вы сможете применить полученные знания на других платформах. Итак, за дело!

Чем мы займемся в этой главе

В этой главе мы покажем, как построить простейшее приложение Kotlin. Для этого необходимо выполнить ряд подготовительных действий.

1. **Создание нового проекта Kotlin.**
   Начнем с установки IntelliJ IDEA (Community Edition) — бесплатной интегрированной среды (IDE), поддерживающей разработку приложений на Kotlin. Затем воспользуемся этой интегрированной средой для построения нового проекта Kotlin:



2. **Создание функции для вывода текста.**
   Добавим в проект новый файл Kotlin, а затем напишем простую функцию `main`, которая будет выводить текст «Pow!»

3. **Обновление функции для расширения ее возможностей.**
   Kotlin поддерживает все основные языковые структуры: команды, циклы и условные конструкции. Воспользуемся ими и изменим функцию так, чтобы расширить ее функциональность.

4. **Проверка кода в интерактивной оболочке Kotlin.**
   Наконец, мы покажем, как протестировать фрагменты кода в интерактивной оболочке Kotlin (или REPL).

Вскоре мы займемся установкой IDEA, но для начала попробуйте выполнить следующее упражнение.

Возьми в руку карандаш

Да, мы знаем, что вы еще ничего не знаете о коде Kotlin, но попробуйте догадаться, что делает каждая строка кода. Мы уже выполнили упражнение для первой строки, чтобы вам было проще начать.

```
val name = "Misty"  Объявить переменную с именем 'name' и присвоить ей значение "Misty".
val height = 9

println("Hello")
println("My cat is called $name")
println("My cat is $height inches tall")

val a = 6
val b = 7
val c = a + b + 10
val str = c.toString()

val numList = arrayOf(1, 2, 3)
var x = 0
while (x < 3) {
    println("Item $x is ${numList[x]}")
    x = x + 1
}

val myCat = Cat(name, height)
val y = height - 3
if (y < 5) myCat.miaow(4)

while (y < 8) {
    myCat.play()
    y = y + 1
}
```

Возьми в руку карандаш
Решение

Да, мы знаем, что вы еще ничего не знаете о коде Kotlin, но попробуйте догадаться, что делает каждая строка кода. Мы уже выполнили упражнение для первой строки, чтобы вам было проще начать.

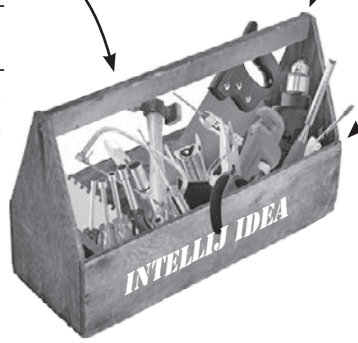
```
val name = "Misty"  Объявить переменную с именем 'name' и присвоить ей значение "Misty".
val height = 9  Объявить переменную с именем 'height' и присвоить ей значение 9.

println("Hello")  Вывести "Hello" в стандартный вывод.
println("My cat is called $name")  Вывести строку "My cat is called Misty".
println("My cat is $height inches tall")  Вывести строку "My cat is 9 inches tall".

val a = 6  Объявить переменную с именем 'a' и присвоить ей значение 6.
val b = 7  Объявить переменную с именем 'b' и присвоить ей значение 7.
val c = a + b + 10  Объявить переменную с именем 'c' и присвоить ей значение 23.
val str = c.toString()  Объявить переменную с именем 'str' и присвоить ей текстовое
                        значение "23".

val numList = arrayOf(1, 2, 3)  Создать массив со значениями 1, 2 и 3.
var x = 0  Объявить переменную с именем 'x' и присвоить ей значение 0.
while (x < 3) {  Продолжать цикл, пока значение x меньше 3.
    println("Item $x is ${numList[x]}")  Вывести индекс и значение каждого элемента
                                         в массиве.
    x = x + 1  Увеличить x на 1.
}  Конец цикла.

val myCat = Cat(name, height)  Объявить переменную с именем 'myCat' и создать объект Cat.
val y = height - 3  Объявить переменную с именем 'y' и присвоить ей значение 6.
if (y < 5) myCat.miaow(4)  Если y меньше 5, метод miaow объекта Cat должен быть
                           вызван 4 раза.

while (y < 8) {  Продолжать цикл, пока y остается меньше 8.
    myCat.play()  Вызвать метод play для объекта Cat.
    y = y + 1  Увеличить y на 1.
}  Конец цикла.
```

Установка IntelliJ IDEA (Community Edition)

Если вы хотите написать и выполнить код Kotlin, проще всего воспользоваться IntelliJ IDEA (Community Edition) — бесплатной интегрированной средой от JetBrains (людей, придумавших язык Kotlin), которая содержит все необходимое для разработки приложений на Kotlin, включая:

Редактор кода
Редактор кода поддерживает автозавершение, упрощающее написание кода Kotlin, форматирование и цветовое выделение кода, упрощающее его чтение. Также редактор выдает подсказки для улучшения кода.

Инструменты построения
Вы можете откомпилировать и запустить свой код при помощи удобных комбинаций клавиш.



...и много других возможностей, упрощающих работу.

Kotlin REPL
Простой доступ к оболочке Kotlin REPL, в которой можно опробовать тот или иной фрагмент вне основного кода.

Контроль версий
IntelliJ IDEA взаимодействует со всеми основными системами контроля версий — такими, как Git, SVN, CVS и другими.

Чтобы воспроизвести примеры, приведенные в книге, установите интегрированную среду IntelliJ IDEA (Community Edition). IDE можно бесплатно загрузить по адресу

https://www.jetbrains.com/idea/download/index.html

Не забудьте установить флажок загрузки бесплатной версии Community Edition среды IntelliJ IDEA.

После того как среда будет установлена, откройте ее. На экране должна появиться заставка IntelliJ IDEA. Теперь все готово для создания вашего первого приложения на Kotlin.

Экран приветствия IntelliJ IDEA.

- [x] **Построение приложения**
- [ ] Добавление функции
- [ ] Обновление функции
- [ ] Использование REPL

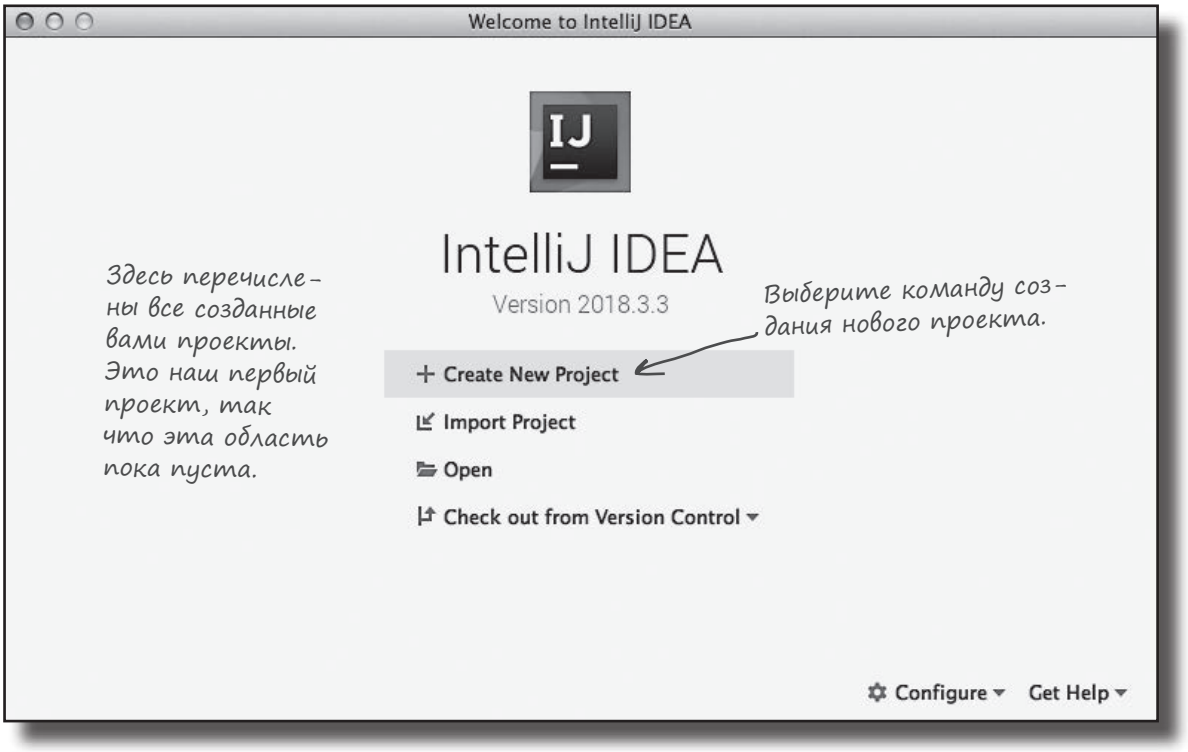
Построение простейшего приложения

Итак, среда разработки готова к использованию, и можно переходить к созданию первого приложения. Мы создадим очень простое приложение, которое будет выводить в IDE текст «Pow!»

Каждый раз, когда в IntelliJ IDEA создается новое приложение, для него необходимо создать новый проект. Убедитесь в том, что IDE открыта, и повторяйте за нами.

1. Создание нового проекта

Заставка IntelliJ IDEA содержит несколько вариантов возможных действий. Для создания нового проекта выберите команду «Create New Project».



Построение простейшего приложения (продолжение)

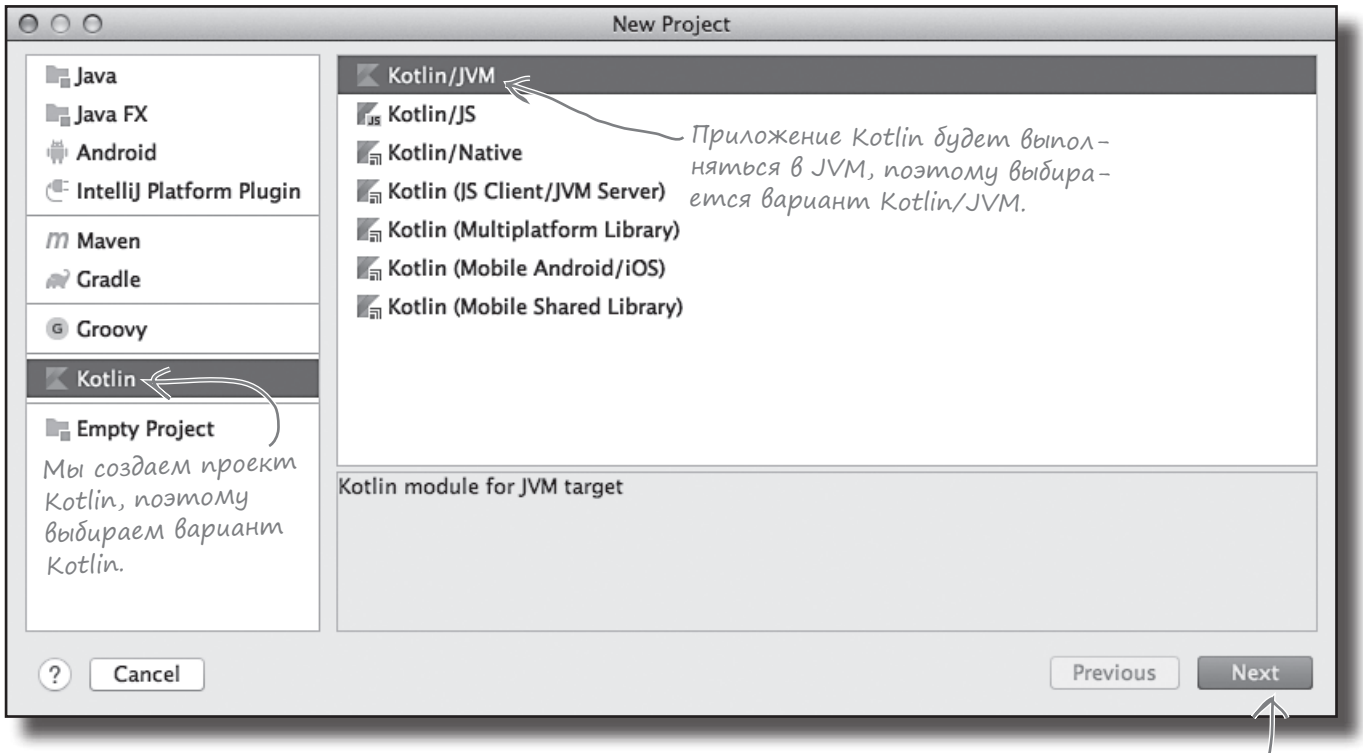
2. Выбор типа проекта

Теперь необходимо сообщить IntelliJ IDEA тип проекта, который вы хотите создать.

IntelliJ IDEA позволяет создавать проекты для разных языков и платформ (например, Java и Android). Мы собираемся создать проект Kotlin, поэтому выберите вариант «Kotlin».

Также необходимо выбрать платформу, для которой будет предназначен проект. Мы собираемся создать приложение Kotlin для виртуальной машины Java (JVM), поэтому выберите вариант Kotlin/JVM. Щелкните на кнопке Next.

Возможны и другие варианты, но мы сосредоточимся на создании приложений, выполняемых в JVM.



Приложение Kotlin будет выполняться в JVM, поэтому выбирается вариант Kotlin/JVM.

Мы создаем проект Kotlin, поэтому выбираем вариант Kotlin.

Щелкните на кнопке Next, чтобы перейти к следующему шагу.

- [x] **Построение приложения**
- [ ] Добавление функции
- [ ] Обновление функции
- [ ] Использование REPL

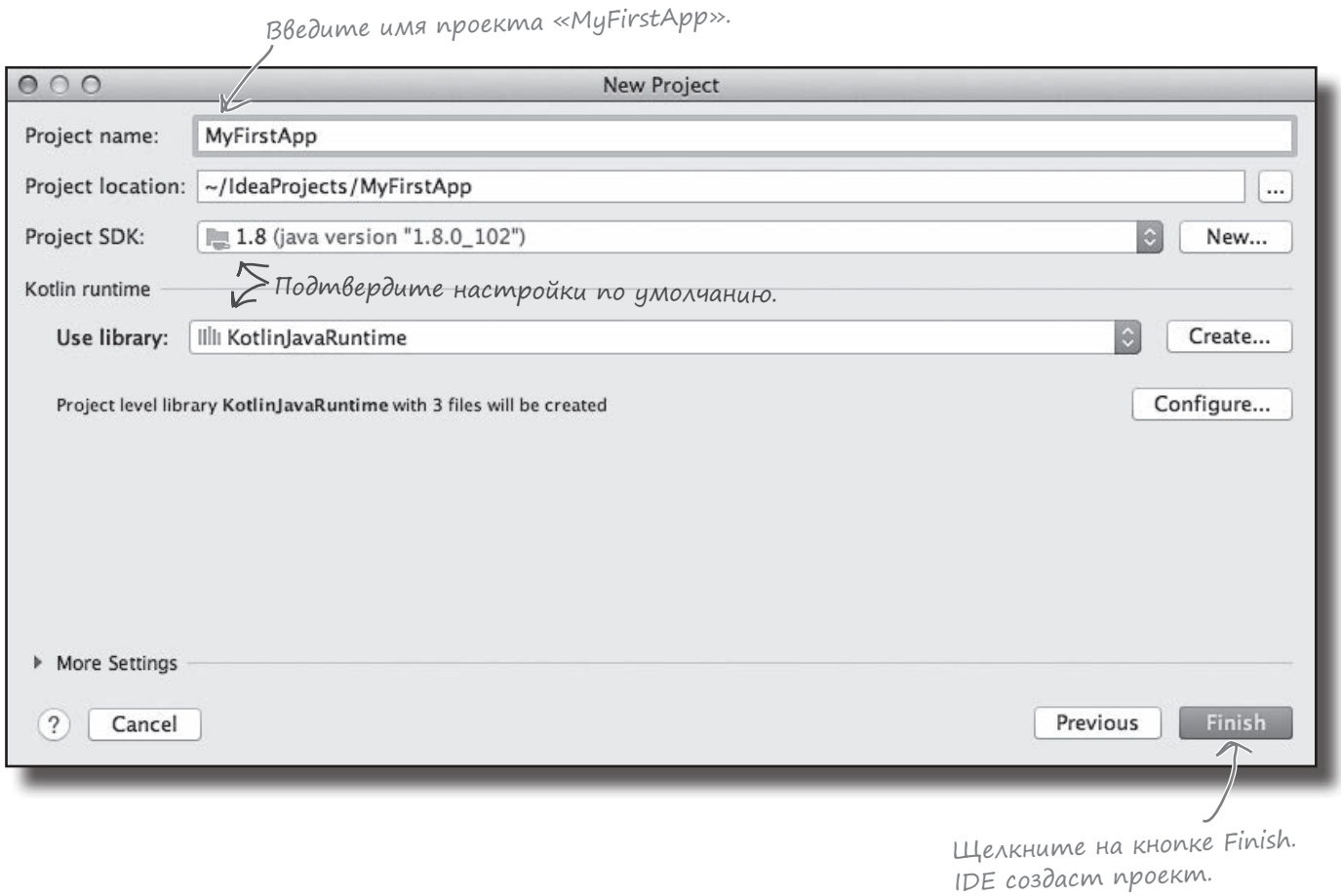
Построение простейшего приложения (продолжение)

3. Настройка конфигурации проекта

Теперь необходимо настроить конфигурацию проекта: как он будет называться, где должны храниться его файлы и какие файлы должны использоваться в проекте. В частности, следует указать версию Java, используемую JVM, и библиотеку для исполнительной среды Kotlin.

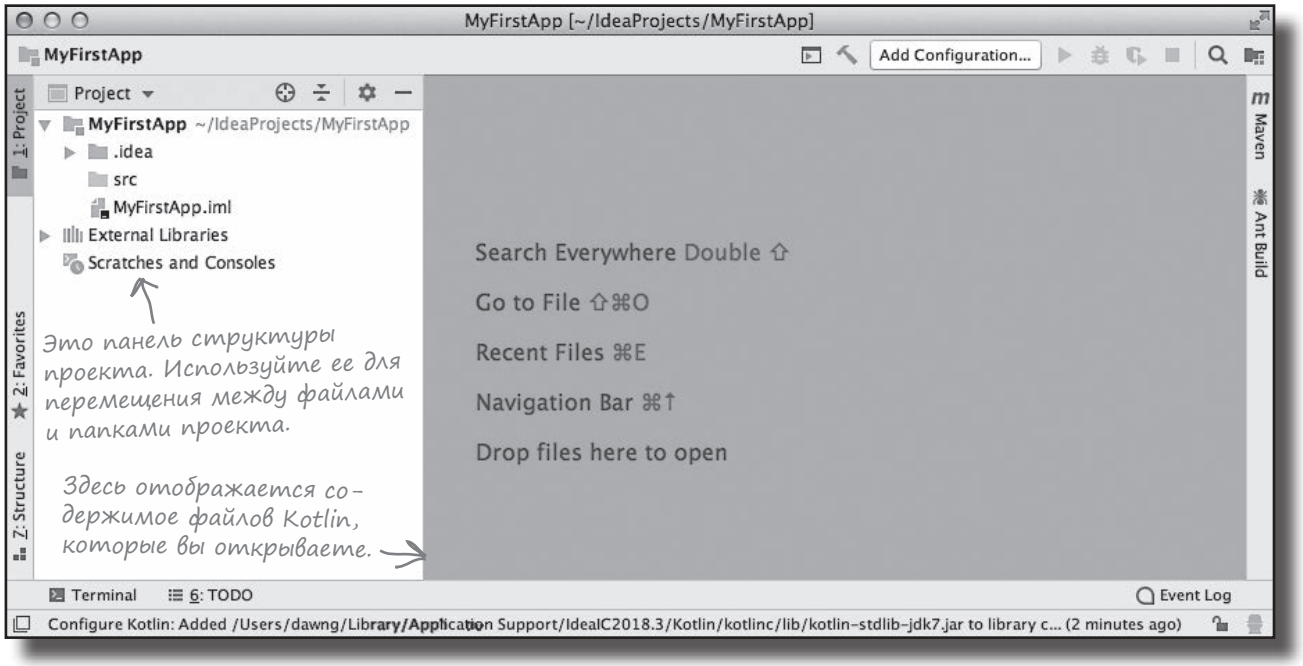
Введите имя проекта «MyFirstApp» и подтвердите остальные настройки по умолчанию.

Когда вы щелкнете на кнопке Finish, IntelliJ IDEA создаст проект.



Вы только что создали свой первый проект Kotlin

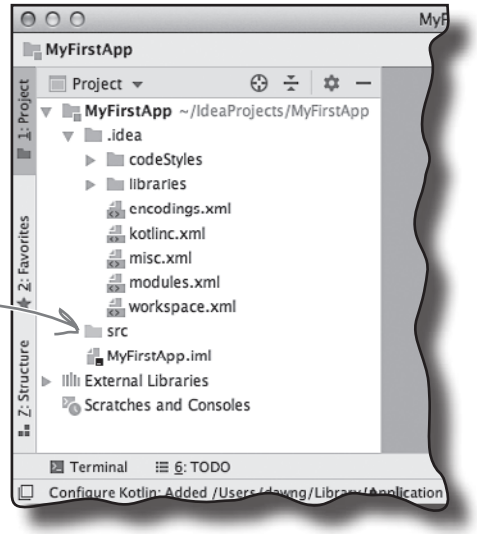
Когда все необходимые действия для создания нового проекта будут выполнены, IntelliJ IDEA создаст проект и отобразит его в интегрированной среде. Вот как выглядит проект, который был сгенерирован интегрированной средой:



Как видите, в проекте имеется панель, предназначенная для перехода между файлами и папками, образующими ваш проект. IntelliJ IDEA автоматически генерирует эту структуру папок при создании проекта.

Структура папок состоит из конфигурационных файлов, используемых IDE, и внешних библиотек, которые будут использоваться в приложении. Также в нее включена папка *src*, предназначенная для файлов с исходным кодом. Большую часть времени вы будете работать с папкой *src*.

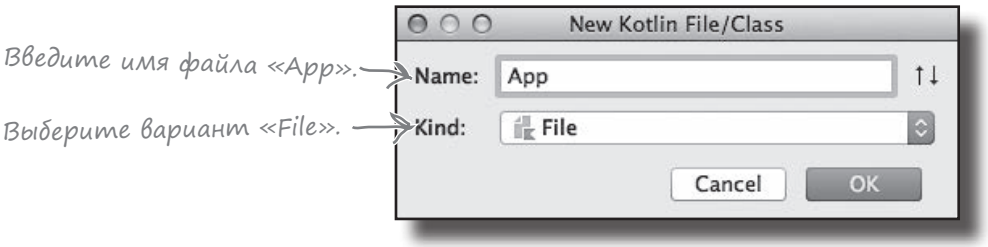
Папка *src* в настоящее время пуста, так как мы еще не добавили в нее ни одного файла Kotlin. Мы сделаем это на следующем шаге.



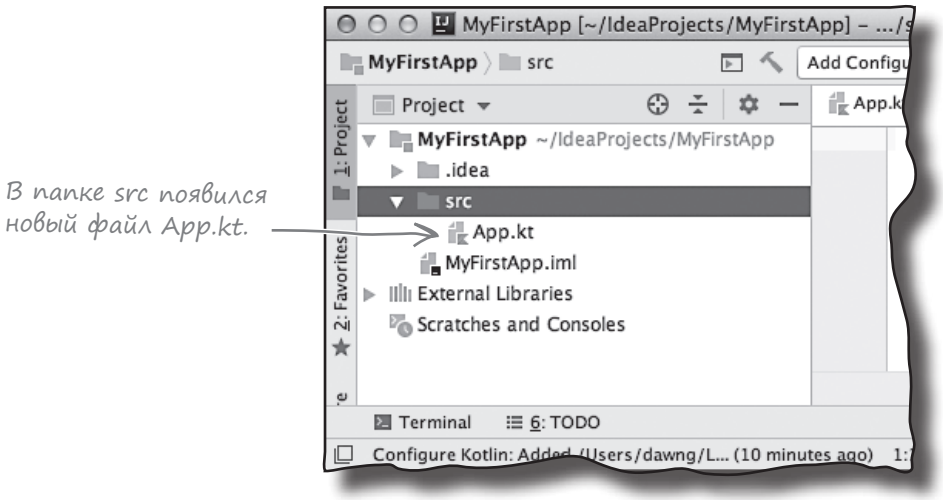
Включение файла Kotlin в проект

Прежде чем написать первый код на Kotlin, необходимо создать файл, в котором этот код будет храниться.

Чтобы добавить новый файл Kotlin в проект, выделите папку *src* на панели структуры IntelliJ IDEA, откройте меню File и выберите команду New → Kotlin File/Class. Вам будет предложено ввести имя и тип создаваемого файла Kotlin. Введите имя приложения «App» и выберите в списке Kind вариант File:



После нажатия кнопки OK IntelliJ IDEA создаст новый файл Kotlin с именем *App.kt* и добавит его в папку *src*:



А теперь рассмотрим код, который необходимо добавить в *App.kt*, чтобы приложение делало что-то полезное.

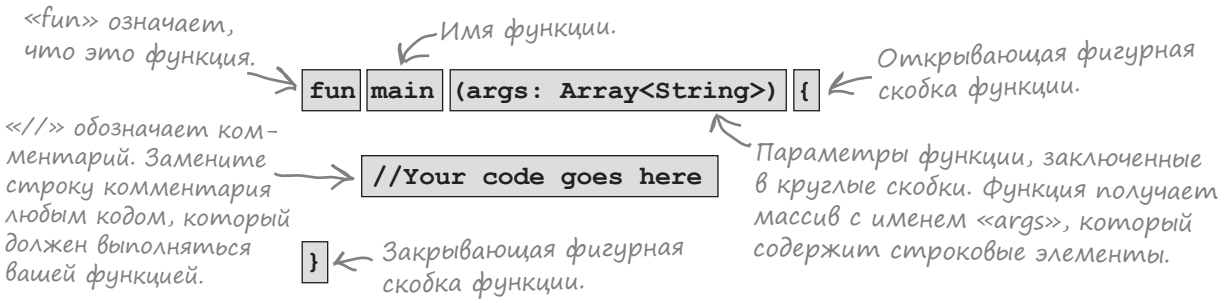
- [x] Построение приложения
- [ ] Добавление функции
- [ ] Обновление функции
- [ ] Использование REPL

Анатомия функции main

Наш код Kotlin должен выводить сообщение «Pow!» в окне вывода IDE. Для этого мы добавим функцию в файл *App.kt*.

В каждое созданное вами приложение Kotlin *обязательно* должна быть включена функция с именем `main`, которая запускает приложение. Когда вы запускаете свой код, JVM находит эту функцию и выполняет ее.

Функция `main` выглядит так:



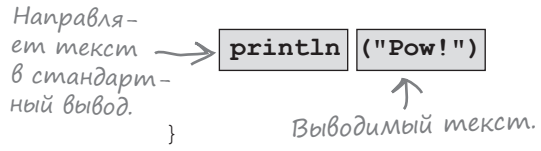
Функция начинается с ключевого слова **fun**, которое сообщает компилятору, что это объявление функции. Ключевое слово `fun` должно использоваться всегда, когда вы создаете новую функцию.

За ключевым словом `fun` следует имя функции — в данном случае **main**. Функция с именем `main` будет автоматически выполняться при запуске приложения.

Код в круглых скобках `()` за именем функции сообщает компилятору, какие аргументы получает функция (и получает ли вообще). В данном случае код `args: Array<String>` означает, что функция получает массив строк, и этому массиву присвоено имя `args`.

Весь код, который вы хотите выполнить, заключается в фигурные скобки `{}` функции `main`. Мы хотим, чтобы функция выводила в IDE сообщение «Pow!», и для этого используется код следующего вида:

```
fun main(args: Array<String>) {
```



Вызов `println("Pow!")` направляет строку символов (`String`) в стандартный вывод. Так как мы будем запускать код в IDE, сообщение «Pow!» появится на панели вывода IDE.

Итак, вы знаете, как должна выглядеть функция **main**. Теперь добавим ее в проект.

Функции main без параметров

Если вы используете Kotlin 1.2 или более раннюю версию, функция `main` для запуска приложения *должна* выглядеть так:

```
fun main(args: Array<String>) {
    //Здесь идет ваш код
}
```

Однако начиная с Kotlin1.3, параметры `main` можно опустить, чтобы сократить объявление функции:

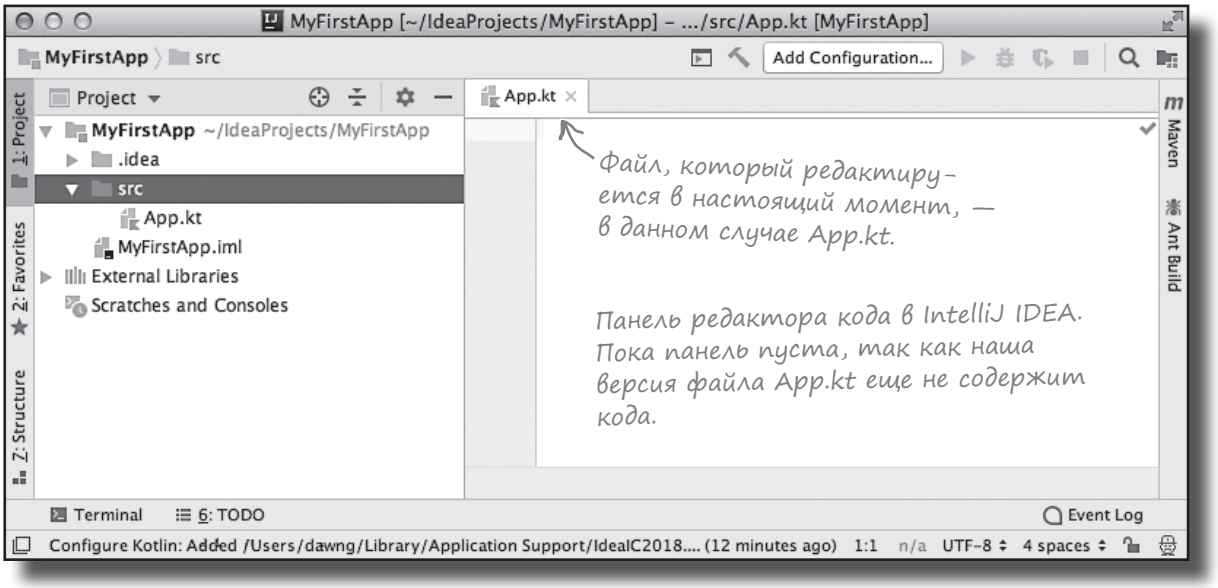
```
fun main() {
    //Здесь идет ваш код
}
```

В этой книге в основном используется более длинная форма функции `main`, потому что она работает во всех версиях Kotlin.

- [x] Построение приложения
- [ ] **Добавление функции**
- [ ] Обновление функции
- [ ] Использование REPL

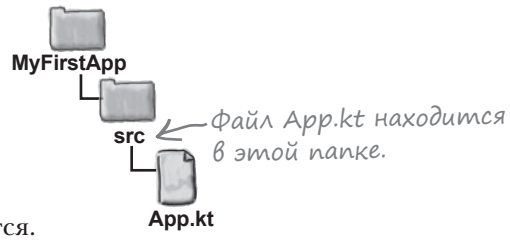
Добавление функции main в файл App.kt

Чтобы добавить функцию `main` в ваш проект, откройте файл *App.kt* — дважды щелкните на панели структуры проекта в IntelliJ IDEA. В среде открывается редактор кода, предназначенный для просмотра и редактирования файлов:



Обновите свою версию *App.kt*, чтобы она соответствовала нашей версии, приведенной ниже:

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



Давайте выполним код и посмотрим, что из этого получится.

Часто Задаваемые Вопросы

В: Функцию `main` нужно добавлять в каждый файл Kotlin, который я создаю?

О: Нет. Приложение Kotlin может состоять из десятков (и даже сотен) файлов, но только *один* из этих файлов может содержать функцию `main` — тот, который запускает приложение.

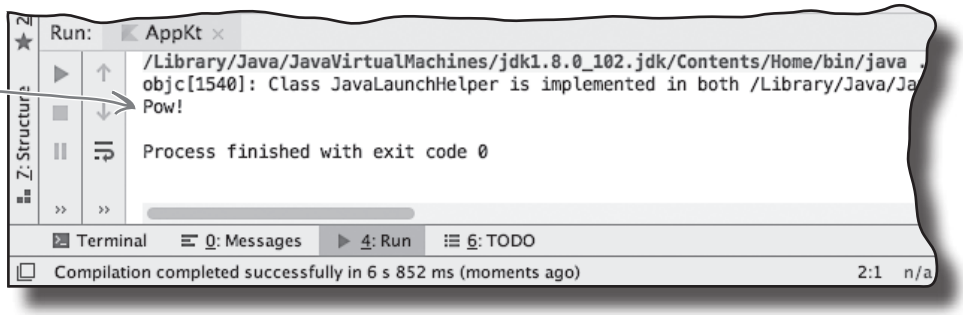
Тест-драйв

- [x] Построение приложения
- [x] Добавление функции
- [ ] Обновление функции
- [ ] Использование REPL

Чтобы запустить код в IntelliJ IDEA, откройте меню Run и выберите команду Run. По запросу выберите вариант AppKt. Команда строит проект и запускает код на выполнение.

После небольшой паузы в окне вывода в нижней части IDE должно появиться сообщение «Pow!»:

Выводимый текст в IDE.

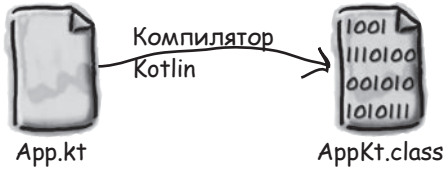


Что делает команда Run

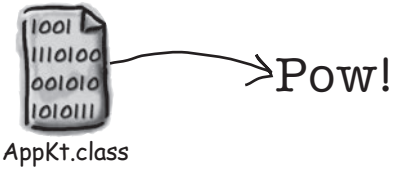
При выполнении команды Run IntelliJ IDEA выполняет пару операций перед тем, как показать вывод вашего кода:

1. **IDE компилирует исходный код Kotlin в байт-код JVM.**
Если код не содержит ошибок, при компиляции создается один или несколько файлов классов, которые могут выполняться JVM. В таком случае компиляция *App.kt* создает файл класса с именем *AppKt.class*.

Исходный код компилируется в байт-код JVM, так как при создании проекта был выбран вариант JVM. Если бы был выбран вариант запуска в другой среде, то компилятор преобразовал бы его в код для соответствующей платформы.



2. **IDE запускает JVM и выполняет класс AppKt.class.**
JVM преобразует байт-код *AppKt.class* в формат, понятный для текущей платформы, после чего выполняет его. На панели вывода IDE появляется строка «Pow!».



Теперь, когда вы знаете, как работает наша функция, посмотрим, что в ней можно изменить, чтобы расширить ее возможности.

- [x] Построение приложения
- [x] Добавление функции
- [ ] Обновление функции
- [ ] Использование REPL

Что можно сделать в функции main?

Внутри функции `main` (да и любой другой функции, если уж на то пошло) происходит самое интересное. В ней можно использовать те же конструкции, которые используются в большинстве языков программирования.

В своем коде вы можете:

Выполнять действия (команды)

```
var x = 3
val name = "Cormoran"
x = x * 10
print("x is $x.")
//This is a comment
```

Делать что-либо снова и снова (циклы)

```
while (x > 20) {
    x = x - 1
    print(" x is now $x.")
}
for (i in 1..10) {
    x = x + 1
    print(" x is now $x.")
}
```

Сделать что-либо по условию

```
if (x == 20) {
    println(" x must be 20.")
} else {
    println(" x isn't 20.")
}
if (name.equals("Cormoran")) {
    println("$name Strike")
}
```

Эти конструкции будут подробнее рассмотрены дальше.

Синтаксис под увеличительным стеклом

Несколько полезных советов по поводу синтаксиса, которые пригодятся вам в процессе программирования Kotlin:

★ Однострочный комментарий начинается с двух символов /:

```
//Это комментарий
```

★ Лишние пробелы игнорируются:

```
x          =          3
```

★ Для определения переменных используется ключевое слово `var` или `val`, за которым следует имя переменной. Ключевое слово `var` используется для переменных, значение которых должно изменяться, а `val` — для переменных, значение которых должно оставаться неизменным. Переменные подробно рассматриваются в главе 2:

```
var x = 100
val serialNo = "AS498HG"
```

- [x] Построение приложения
- [x] Добавление функции
- [ ] Обновление функции
- [ ] Использование REPL

Цикл, цикл, цикл...

В Kotlin поддерживаются три стандартные циклические конструкции: `while`, `do-while` и `for`. Пока мы ограничимся циклом `while`.

Синтаксис циклов `while` относительно несложен. Пока условие остается истинным, выполняются команды в *блоке* цикла. Блок цикла заключен в фигурные скобки; таким образом, все команды, которые должны повторяться, следует поместить в этот блок.

Если блок цикла состоит всего из одной строки, фигурные скобки не обязательны.

Правильность поведения цикла `while` очень сильно зависит от его *условия*. Условие — это выражение, результатом вычисления которого является логическое значение *true* или *false*. Например, если вы используете конструкцию вида «Пока *остаетсяМороженое* равно *true*, положить мороженого», в программе используется четкое и однозначное логическое условие. Значение *остаетсяМороженое* либо истинно, либо ложно — других вариантов нет. Но если вы используете конструкцию вида «Пока *Фред*, положить мороженого», здесь нет нормального условия, которое можно было бы проверить. Чтобы выражение приобрело смысл, нужно заменить его чем-то вроде «Пока *Фред* голоден, положить мороженого».

Простые логические проверки

Простая логическая проверка может осуществляться сравнением переменной с заданным значением при помощи одного из операторов сравнения:

< (меньше)

> (больше)

== (равно) ← Для проверки равенства используются два знака =, а не один.

<= (меньше либо равно)

>= (больше либо равно)

Обратите внимание на различия между оператором присваивания (один знак =) и оператором проверки равенства (два знака =).

Пример кода с использованием логических условий:

```
var x = 4 //Присвоить x значение 4
while (x > 3) {
    //Код цикла будет выполняться, пока x остается больше 4
    println(x)
    x = x - 1
}
var z = 27
while (z == 10) {
    //Код цикла не выполняется, так как переменная z равна 27
    println(z)
    z = z + 6
}
```

- [x] Построение приложения
- [x] Добавление функции
- [ ] **Обновление функции**
- [ ] Использование REPL

Пример цикла

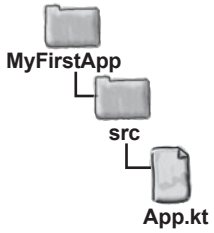
Обновим код в файле *App.kt* новой версией функции `main`. Мы изменим функцию `main` так, чтобы она выводила сообщение перед началом цикла, при каждом выполнении цикла и после его завершения.

Измените свою версию *App.kt* так, чтобы она соответствовала нашей (изменения выделены жирным шрифтом):

```
fun main(args: Array<String>) {
    println("Row!")
    var x = 1
    println("Before the loop. x = $x.")
    while (x < 4) {
        println("In the loop. x = $x.")
        x = x + 1
    }
    println("After the loop. x = $x.")
}
```

Удалите эту строку, она больше не нужна.

Выводит значение x.



Попробуем выполнить этот код.

Тест-драйв

Выполните код: откройте меню Run и выберите команду Run 'AppKt'. В окне вывода в нижней части IDE должны появиться следующие сообщения:

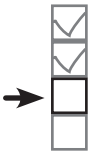
```
Before the loop. x = 1.
In the loop. x = 1.
In the loop. x = 2.
In the loop. x = 3.
After the loop. x = 4.
```

Итак, теперь вы знаете, как работают циклы `while` и логические проверки. Перейдем к рассмотрению условных конструкций `if`.

print и println

Вероятно, вы заметили, что в одних местах используется **`print`**, а в других **`println`**. Чем они отличаются?

`println` вставляет новую строку после вывода, а **`print`** продолжает вывод в *той же* строке. Если вы хотите, чтобы каждое сообщение выводилось с новой строки, используйте `println`. Если вы хотите, чтобы все сообщения выводились подряд в одной строке, используйте `print`.



Условные конструкции

Условная конструкция `if` похожа на логическое условие цикла `while`, только вместо «***пока*** остается мороженое...» вы говорите: «***если*** осталось мороженое...»

Чтобы вы лучше поняли, как работают условные конструкции, рассмотрим пример. Этот код выводит строковое сообщение, если одно число больше другого:

```
fun main(args: Array<String>) {
    val x = 3
    val y = 1
    if (x > y) {
        println("x is greater than y")
    }
    println("This line runs no matter what")
}
```

Если блок if содержит всего одну строку кода, фигурные скобки не обязательны.

Эта строка выполняется только в том случае, если x больше y.

Приведенный выше код выполняет строку, которая выводит сообщение «*x* is greater than *y*» только в том случае, если условие (`x` больше `y`) истинно. При этом строка, которая выводит сообщение «This line runs no matter what», выполняется независимо от того, истинно условие или ложно. Итак, в зависимости от значений `x` и `y` будет выведено либо одно сообщение, либо два.

Также можно добавить в условную конструкцию секцию `else`, чтобы в программе можно было использовать конструкции вида «*если* мороженое еще осталось, положить еще, *иначе* съесть мороженое, а потом купить еще».

Ниже приведена обновленная версия приведенного выше кода с секцией `else`:

```
fun main(args: Array<String>) {
    val x = 3
    val y = 1
    if (x > y) {
        println("x is greater than y")
    } else {
        println("x is not greater than y")
    }
    println("This line runs no matter what")
}
```

Эта строка будет только в том случае, если условие x > y не выполняется.

Во многих языках программирования действие `if` на этом заканчивается; условная конструкция выполняет код, *если* условие истинно. Однако Kotlin на этом не останавливается.

- [x] Построение приложения
- [x] Добавление функции
- [ ] **Обновление функции**
- [ ] Использование REPL

Использование if для возвращения значения

В Kotlin `if` может использоваться как **выражение**, возвращающее значение. Его можно сравнить с конструкцией вида: «*если* мороженое осталось, вернуть одно значение, иначе вернуть другое значение». Эта форма `if` может использоваться для написания более компактного кода.

Давайте посмотрим, как работает эта возможность, на примере кода с предыдущей страницы. Ранее для вывода строкового сообщения использовался следующий фрагмент:

```
if (x > y) {
    println("x is greater than y")
} else {
    println("x is not greater than y")
}
```

Когда вы используете if как выражение, ОБЯЗАТЕЛЬНО включайте секцию else.

Этот фрагмент можно переписать с использованием выражения `if`:

```
println(if (x > y) "x is greater than y" else "x is not greater than y")
```

Код:

```
if (x > y) "x is greater than y" else "x is not greater than y"
```

является выражением `if`. Сначала он проверяет условие `if`: `x > y`. Если условие истинно (*true*), то выражение возвращает строку «x is greater than y». В противном случае (`else`) условие ложно (*false*), и выражение вместо этого вернет строку «x is not greater than y».

Для вывода результата выражения `if` используется вызов `println`:

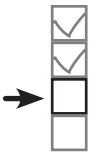
```
println(if (x > y) "x is greater than y" else "x is not greater than y")
```

Если x больше y, код выводит сообщение «x is greater than y». Если же x не больше y, то выводится сообщение «x is not greater than y».

Итак, если `x` больше `y`, выводится сообщение «x is greater than y», а если нет, будет выведено сообщение «x is not greater than y».

Как видите, такое использование выражения `if` приводит к тому же результату, что и на предыдущей странице, но оно более компактно.

Полный код всей функции приведен на следующей странице.



Обновление функции main

А теперь обновим код *App.kt* новой версией функции `main`, в которой используется выражение `if`. Внесите изменения в код вашей версии *App.kt*, чтобы он соответствовал нашей версии:

```
fun main(args: Array<String>) {
    var x = 1
    println("Before the loop. x = $x.")
    while (x < 4) {
        println("In the loop. x = $x.")
        x = x + 1
    }
    println("After the loop. x = $x.")
    val x = 3
    val y = 1
    println(if (x > y) "x is greater than y" else "x is not greater than y")
    println("This line runs no matter what")
}
```

Удалите эти строки.

А теперь опробуем новую версию кода в деле.

Тест-драйв

Запустите код: откройте меню Run и выберите команду Run 'AppKt'. В окне вывода в нижней части IDE должен появиться следующий текст:

```
x is greater than y
This line runs no matter what
```

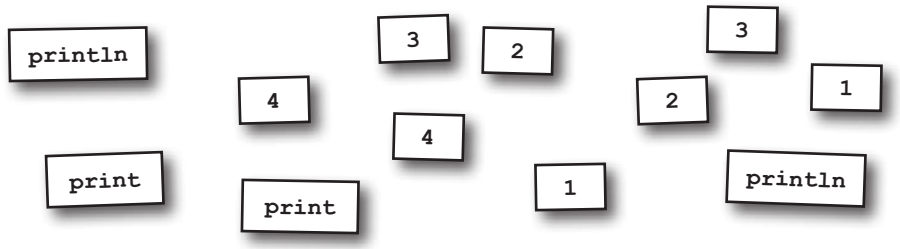
Теперь, когда вы узнали, как использовать `if` для условного выбора и в выражениях, попробуйте выполнить следующее упражнение.

Развлечения с магнитами

Кто-то выложил магнитами на холодильнике код новой функции **main**, которая выводит сообщение «YabbaDabbaDo». К сожалению, от сквозняка магниты упали на пол. Сможете ли вы снова собрать код функции?

Некоторые магниты могут остаться неиспользованными.

```
fun main(args: Array<String>) {
    var x = 1

    while (x <          ) {

        ..........(if (x ==          ) "Yab" else "Dab")

        ..........("ba")

        x = x + 1
    }
    if (x ==          ) println("Do")
}
```



Ответы на с. 59.

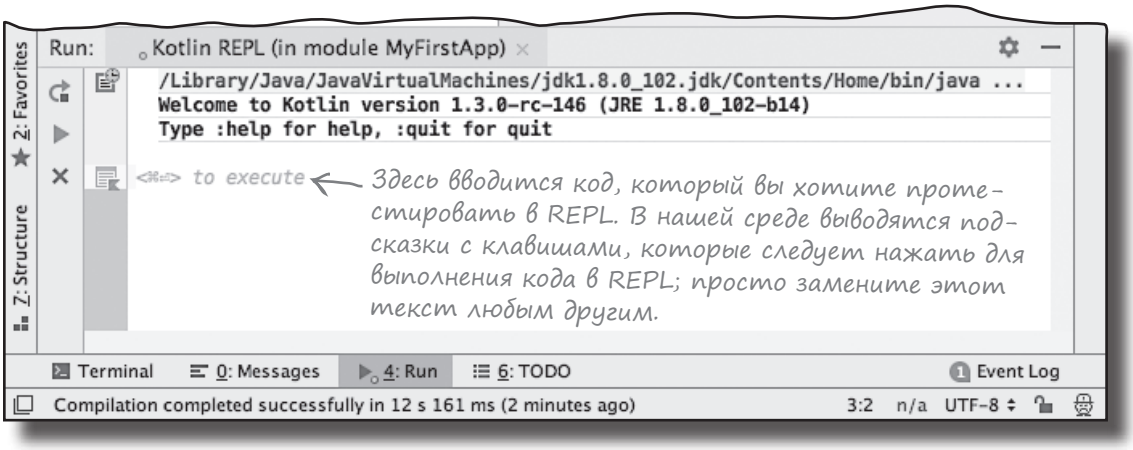
- [x] Построение приложения
- [x] Добавление функции
- [x] Обновление функции
- [ ] Использование REPL

Использование интерактивной оболочки Kotlin

Глава подходит к концу, и прежде чем переходить к следующей, мы хотим познакомить вас с еще одной полезной возможностью: интерактивной оболочкой Kotlin, или REPL. REPL позволяет быстро протестировать фрагменты кода за пределами основного кода приложения.

REPL означает «Read-Eval-Print Loop» («цикл чтение-вычисление-вывод»), но никто не использует полное название.

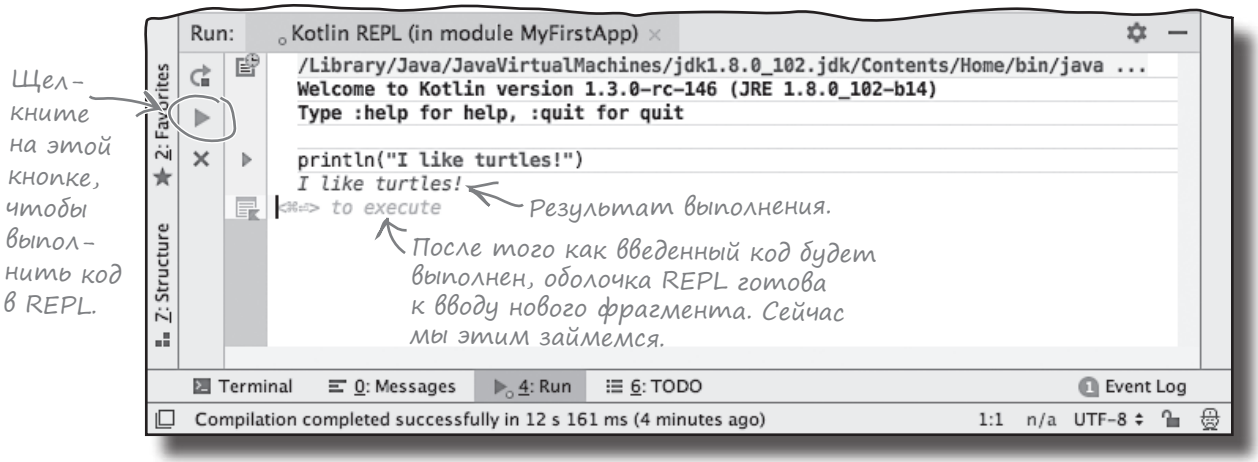
Чтобы запустить REPL, откройте меню Tools в IntelliJ IDEA и выберите команду Kotlin → Kotlin REPL. В нижней части экрана откроется новая панель:



Просто введите тот код, который хотите опробовать, в окне REPL. Например, попробуйте ввести следующий фрагмент:

```
println("I like turtles!")
```

После того как код будет введен, выполните его при помощи большой зеленой кнопки Run в левой части окна REPL. После небольшой паузы в окне REPL появляется сообщение «I like turtles!»:



- [x] Построение приложения
- [x] Добавление функции
- [x] Обновление функции
- [x] Использование REPL

Мы выполнили все четыре пункта этой главы.

В REPL можно вводить многострочные фрагменты

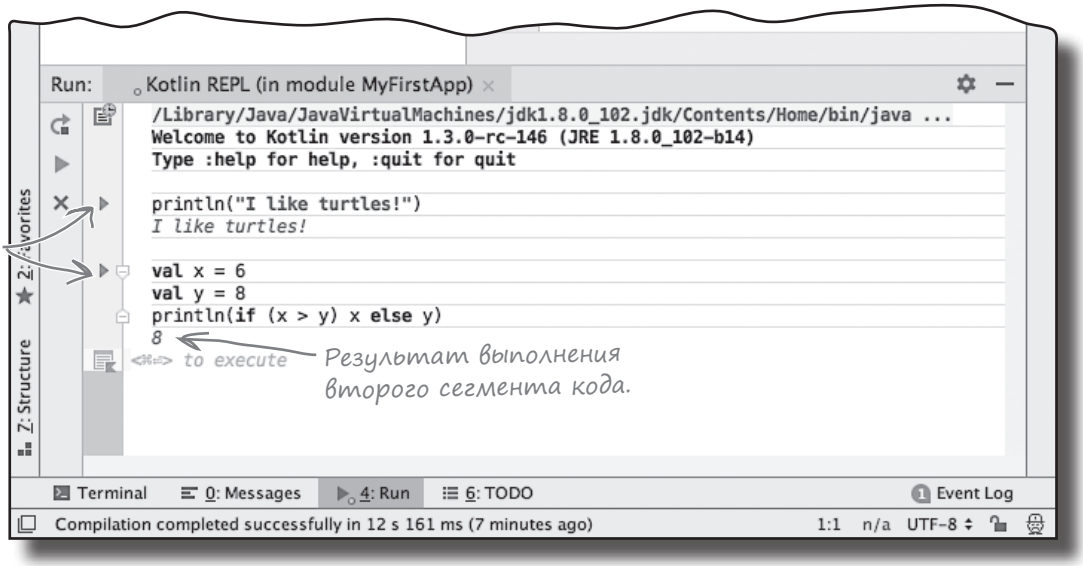
Кроме добавления однострочных фрагментов кода в REPL, как на предыдущей странице, вы также можете опробовать многострочные фрагменты кода. Например, попробуйте ввести следующие строки в окне REPL:

```
val x = 6
val y = 8
println(if (x > y) x else y)
```

Выводит большее из двух чисел, x и y.

При выполнении кода в REPL должен появиться результат 8:

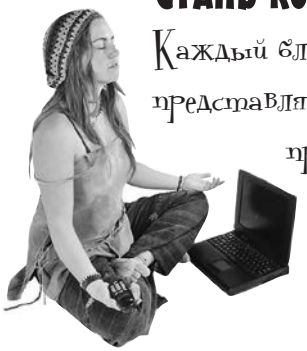
Похоже на уменьшенные версии кнопки выполнения, но в действительности это обозначение выполненных блоков кода.



Пришло время для упражнений

Итак, вы узнали, как писать код Kotlin, и ознакомились с базовыми элементами синтаксиса. Пора проверить новые знания на нескольких упражнениях. Помните: если вы сомневаетесь, фрагменты кода всегда можно опробовать в REPL.

СТАНЬ компилятором



Каждый блок кода Kotlin на этой странице представляет полный исходный файл. Попробуйте представить себя на месте компилятора и определить, будет ли компилироваться каждый из этих файлов. Если какие-то файлы не компилируются, то как бы вы их исправили?

A

```
fun main(args: Array<String>) {
    var x = 1
    while (x < 10) {
        if (x > 3) {
            println("big x")
        }
    }
}
```

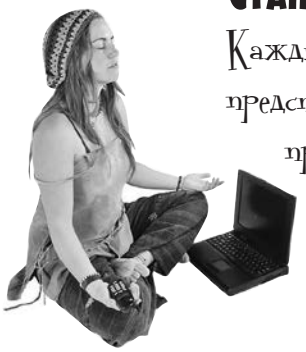
B

```
fun main(args: Array<String>) {
    val x = 10
    while (x > 1) {
        x = x - 1
        if (x < 3) println("small x")
    }
}
```

C

```
fun main(args: Array<String>) {
    var x = 10
    while (x > 1) {
        x = x - 1
        print(if (x < 3) "small x")
    }
}
```

СТАНЬ компилятором. Решение



Каждый блок кода Kotlin на этой странице представляет полный исходный файл. Попробуйте представить себя на месте компилятора и определить, будет ли компилироваться каждый из этих файлов. Если какие-то файлы не компилируются, то как бы вы их исправили?

A

```
fun main(args: Array<String>) {
    var x = 1
    while (x < 10) {
        x = x + 1
        if (x > 3) {
            println("big x")
        }
    }
}
```

Компилируется и выполняется без вывода результата, но без добавленной строки, цикл «while» будет выполняться бесконечно.

B

```
fun main(args: Array<String>) {
    ~~val~~ var x = 10
    while (x > 1) {
        x = x - 1
        if (x < 3) println("small x")
    }
}
```

Не компилируется. x определяется с ключевым словом val, поэтому значение x не может изменяться. Следовательно, код не сможет обновить значение x внутри цикла «while». Чтобы исправить ошибку, замените val на var.

C

```
fun main(args: Array<String>) {
    var x = 10
    while (x > 1) {
        x = x - 1
        print(if (x < 3) "small x" else "big x")
    }
}
```

Не компилируется, используется выражение if без секции else. Чтобы исправить ошибку, добавьте секцию else.

Путаница с сообщениями

Ниже приведена короткая программа Kotlin. Один блок в программе пропущен. Ваша задача — сопоставить блоки-кандидаты (слева) с выводом, который вы увидите при подстановке этого блока. Не все варианты вывода будут использоваться, а некоторые варианты могут использоваться более одного раза. Соедините каждый блок с подходящим вариантом вывода.

```
fun main(args: Array<String>) {
    var x = 0
    var y = 0
    while (x < 5) {
        [ Код блока подставляется сюда. ]
        print("$x$y ")
        x = x + 1
    }
}
```

Блоки:

```
y = x - y
```

```
y = y + x
```

```
y = y + 3
if (y > 4) y = y - 1
```

```
x = x + 2
y = y + x
```

```
if (y < 5) {
    x = x + 1
    if (y < 3) x = x - 1
}
y = y + 3
```

Соедините каждый блок с одним из возможных вариантов вывода.

Варианты вывода:

```
00 11 23 36 410
```

```
00 11 22 33 44
```

```
00 11 21 32 42
```

```
03 15 27 39 411
```

```
22 57
```

```
02 14 25 36 47
```

```
03 26 39 412
```

Путаница с сообщениями. Решение

Ниже приведена короткая программа Kotlin. Один блок в программе пропущен. Ваша задача — сопоставить блоки-кандидаты (слева) с выводом, который вы увидите при подстановке этого блока. Не все варианты вывода будут использоваться, а некоторые варианты могут использоваться более одного раза. Соедините каждый блок с подходящим вариантом вывода.

```
fun main(args: Array<String>) {
    var x = 0
    var y = 0
    while (x < 5) {
        [ ]
        print("$x$y ")
        x = x + 1
    }
}
```

Блоки:

```
y = x - y
```

```
y = y + x
```

```
y = y + 3
if (y > 4) y = y - 1
```

```
x = x + 2
y = y + x
```

```
if (y < 5) {
    x = x + 1
    if (y < 3) x = x - 1
}
y = y + 3
```

Варианты вывода:

```
00 11 23 36 410
```

```
00 11 22 33 44
```

```
00 11 21 32 42
```

```
03 15 27 39 411
```

```
22 57
```

```
02 14 25 36 47
```

```
03 26 39 412
```

Развлечения с магнитами. Решение

Кто-то выложил магнитами на холодильнике код новой функции **main**, которая выводит сообщение «YabbaDabbaDo». К сожалению, от сквозняка магниты упали на пол. Сможете ли вы снова собрать код функции?

Некоторые магниты могут остаться неиспользованными.

```
fun main(args: Array<String>) {
    var x = 1

    while (x < 3 ) {

        print (if (x == 1 ) "Yab" else "Dab")

        print ("ba")

        x = x + 1
    }
    if (x == 3 ) println("Do")
}
```

println

4

2

4

2

1

println

Эти магниты остались неиспользованными.

Ваш инструментарий Kotlin

Глава 1 осталась позади, а ваш инструментарий пополнился основными элементами синтаксиса Kotlin.

Весь код для этой главы можно загрузить по адресу https://tinyurl.com/HFKotlin.

ГЛАВА 1

КЛЮЧЕВЫЕ МОМЕНТЫ

- Ключевое слово `fun` используется для определения функций.
- В каждом приложении должна присутствовать функция с именем `main`.
- Символы `//` обозначают однострочный комментарий.
- Строка (`String`) представляет собой цепочку символов. Строковые значения заключаются в двойные кавычки.
- Программные блоки определяются в паре фигурных скобок `{ }`.
- Оператор присваивания состоит из *одного* знака присваивания `=`.
- Оператор проверки равенства состоит из *двух* знаков равенства `==`.
- Ключевое слово `var` определяет переменную, значение которой может изменяться.
- Ключевое слово `val` определяет переменную, значение которой остается неизменным.
- Цикл `while` выполняет все команды в блоке, пока условие остается истинным (*true*).
- Если условие ложно (*false*), блок цикла `while` выполняться не будет, а управление будет передано коду, следующему непосредственно за блоком цикла.
- Условие заключается в круглые скобки `( )`.
- Условные проверки в коде определяются ключевыми словами `if` и `else`. Секция `else` не обязательна.
- `if` может использоваться как выражение, возвращающее значение. В этом случае секция `else` обязательна.

Из жизни переменных

От переменных зависит весь ваш код. В этой главе мы заглянем «под капот» и покажем, ***как на самом деле работают переменные Kotlin***. Вы познакомитесь с **базовыми типами**, такими как *Int*, *Float* и *Boolean*, и узнаете, что компилятор Kotlin способен **вычислить тип переменной по присвоенному ей значению.** Вы научитесь пользоваться **строковыми шаблонами** для построения сложных строк с минимумом кода, и узнаете, как создать **массивы** для хранения нескольких значений. Напоследок мы расскажем, *почему объекты играют такую важную роль в программировании Kotlin*.

Без переменных не обойтись

Вы уже научились писать основные команды, выражения, циклы `while` и условия `if`. Однако существует одна штука, просто необходимая для написания качественного кода: переменные.

Вы уже видели, как объявлять переменные в коде. Это делается примерно так:

```
var x = 5
```

Объявление выглядит просто, но что же происходит за кулисами?

Переменная похожа на стакан

Представляя себе переменную в Kotlin, представьте себе стакан. Стаканы бывают самых разных форм и размеров: большие, маленькие, огромные для попкорна в кинотеатрах — но у всех них есть нечто общее: в них что-то находится.

Переменная похожа на стакан: в ней тоже что-то находится.

Объявление переменной напоминает заказ кофе в Starbucks. При оформлении заказа вы говорите бариста, какой напиток вам нужен, ваше имя, чтобы позвать вас, когда он будет готов, и даже нужно ли использовать модный стаканчик, пригодный для повторного использования, вместо обычного, который просто выбрасывается. Когда вы объявляете переменную в коде следующего вида:

```
var x = 5
```

вы сообщаете компилятору Kotlin, какое значение должно храниться в переменной, какое имя ей следует присвоить и может ли переменная повторно использоваться для других значений.

Чтобы создать переменную, компилятор должен знать три вещи:

- **Имя переменной.**
  Имя необходимо для обращения к значению переменной в коде.
- **Возможно ли повторное использование переменной.**
  Например, если изначально переменной присваивается значение 2, можно ли ей позднее присвоить значение 3? Или переменная навсегда остается равной 2?
- **Тип переменной.**
  Что это — целое число? Строка? Что-то более сложное?

Вы уже видели, как присвоить имя переменной и как при помощи ключевых слов `val` и `var` определить, может ли переменная использоваться для других значений. Но как насчет типа переменной?

Что происходит при объявлении переменной

Тип переменной очень важен для компилятора — информация о типе позволит предотвратить необычные или рискованные операции, которые могут привести к ошибкам. Скажем, компилятор не позволит присвоить строку «Fish» целочисленной переменной, потому что он знает, что математические операции с типом `String` не имеют смысла.

Чтобы механизм безопасности типов работал, компилятор должен знать тип переменной. А тип переменной может быть определен компилятором **по значению, присвоенному этой переменной**.

Давайте посмотрим, как это происходит.

Чтобы создать переменную, компилятор должен знать ее имя, тип и возможность ее изменения.

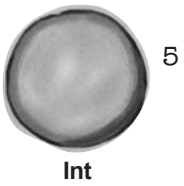
Значение преобразуется в объект...

При объявлении переменной в коде следующего вида:

```
var x = 5
```

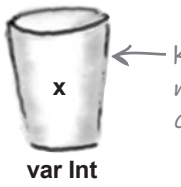
значение, присвоенное переменной, используется для создания нового объекта. В данном примере число 5 присваивается новой переменной с именем `x`. Компилятор знает, что 5 является целым числом, поэтому он создает новый объект `Int` со значением 5:

Вскоре мы более подробно рассмотрим другие типы переменных.



...и компилятор определяет тип переменной по этому объекту

Затем компилятор использует тип объекта в качестве типа переменной. В приведенном примере объект имеет тип `Int`, поэтому переменная тоже будет относиться к типу `Int`. Этот тип навсегда закрепляется за переменной.



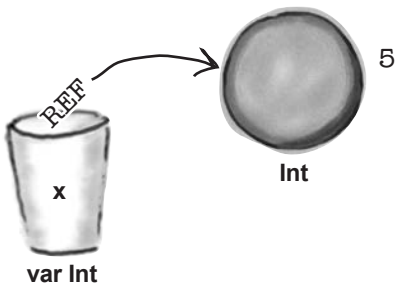
Компилятор знает, что вам нужна переменная с типом Int, соответствующим типу объекта.

Теперь объект присваивается переменной. Как это происходит?

В переменной хранится ссылка на объект

Когда объект присваивается переменной, **сам объект в переменной не сохраняется**. Вместо этого в переменной сохраняется *ссылка* на объект:

```
var x = 5
```



По ссылке, хранящейся в переменной, можно получить доступ к объекту.

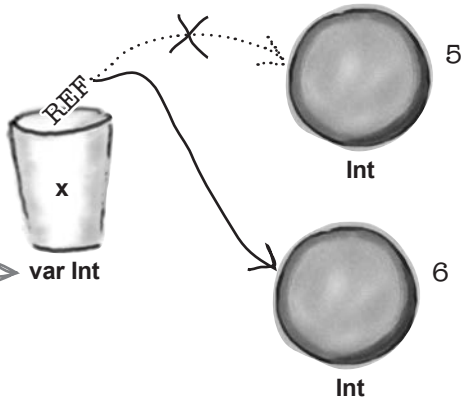
Снова о различиях между val и var

Если переменная объявляется с ключевым словом `val`, ссылка на объект сохраняется в переменной раз и навсегда и не может быть заменена. Но если вместо `val` используется ключевое слово `var`, переменной может быть присвоено другое значение. Например, следующая команда:

```
x = 6
```

присваивающая значение 6 переменной `x`, создает новый объект `Int` со значением 6 и сохраняет ссылку на него в `x`. При этом исходная ссылка заменяется новой:

```
x = 6
```



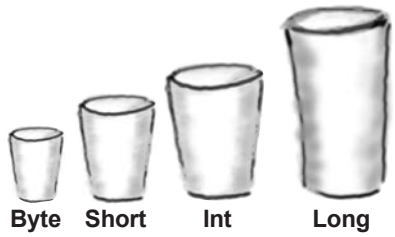
Ссылка, хранящаяся в переменной, заменяется, так как она была объявлена с ключевым словом var. Это было бы невозможно, если бы переменная была объявлена с ключевым словом val.

Итак, теперь вы знаете, что происходит при объявлении переменных. И мы можем рассмотреть некоторые базовые типы Kotlin для представления целых чисел, вещественных чисел, логических значений, символов и строк.

Базовые типы Kotlin

Целые числа

В Kotlin поддерживаются четыре целочисленных типа: **`Byte`**, **`Short`**, **`Int`** и **`Long`**. Каждый тип вмещает фиксированное количество битов. Например, тип `Byte` может содержать 8 битов, поэтому `Byte` может представлять целочисленные значения от –128 до 127. С другой стороны, тип `Int` вмещает до 32 битов, поэтому `Int` может представлять целочисленные значения в диапазоне от –2 147 483 648 до 2 147 483 647.



По умолчанию при объявлении переменной с присваиванием целого числа командой следующего вида:

```
var x = 1
```

будет создан объект и переменная типа `Int`. Если присваиваемое число слишком велико, чтобы поместиться в `Int`, то будет создан объект и переменная типа `Long`. Чтобы создать объект и переменную типа `Long`, можно также добавить суффикс «L» в конец целого числа:

```
var hugeNumber = 6L
```

В следующей таблице перечислены целочисленные типы, их размеры в битах и диапазоны значений:

Тип	Биты	Диапазон значений
Byte	8 битов	–128 до 127
Short	16 битов	–32768 до 32767
Int	32 бита	–2147483648 до 2147483647
Long	64 бита	–huge до (huge - 1)

Шестнадцатеричные и двоичные числа

★ Чтобы присвоить переменной двоичное число, поставьте перед числом префикс 0b.

```
x = 0b10
```

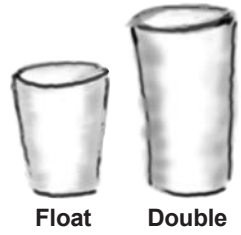
★ Чтобы присвоить шестнадцатеричное число, поставьте перед ним префикс 0x.

```
y = 0xAB
```

★ Восьмеричные числа не поддерживаются.

Значения с плавающей точкой

Существуют два основных типа с плавающей точкой: **`Float`** и **`Double`**. Тип `Float` вмещает до 32 битов, тогда как `Double` — до 64 битов.



По умолчанию при объявлении переменной с присваиванием числа с плавающей точкой:

```
var x = 123.5
```

будет создан объект и переменная типа `Double`. Если добавить в конец числа суффикс «F» или «f», то создается тип `Float`:

```
var x = 123.5F
```

Логические значения

Переменные **Boolean** используются для значений из набора `true` или `false`. Для создания объекта и переменных `Boolean` переменные объявляются в коде следующего вида:

```
var isBarking = true
var isTrained = false
```

Символы и строки

Остается сказать еще о двух базовых типах: **Char** и **String**.

Переменные `Char` используются для представления отдельных символов. При создании переменной `Char` присваивается символ, заключенный в одинарные кавычки:

```
var letter = 'D'
```

Переменные `String` предназначены для хранения наборов символов, объединенных в цепочку. Чтобы создать переменную типа `String`, присвойте ей последовательность символов, заключенных в двойные кавычки:

```
var name = "Fido"
```

Переменные Char используются для хранения отдельных символов. Переменные String предназначены для хранения наборов символов, объединенных в цепочку.

Вы сказали, что компилятор определяет тип переменной по типу присваиваемого значения. Но как мне создать переменную типа Byte или Short, если компилятор считает, что малые целые числа относятся к типу Int? И как мне определить переменную, значение которой мне еще неизвестно?

В таких ситуациях необходимо явно указать тип переменной.

Сейчас мы посмотрим, как это делается.

Как явно объявить тип переменной

Ранее мы показали, как создать переменную с присваиванием значения и поручить компилятору определить ее тип по значению. Однако в некоторых случаях требуется *явно сообщить компилятору тип создаваемой переменной*. Допустим, вы хотите использовать `Byte` или `Short` вместо `Int`, так как эти типы более эффективны. А может, хотите объявить переменную в начале кода и присвоить ей значение позднее.

Объявление переменной с явным указанием типа выглядит так:

```
var smallNum: Short
```

Чтобы не поручать компилятору определение типа переменной по ее значению, поставьте двоеточие (`:`) после имени переменной, а потом укажите нужный тип. Итак, приведенный выше код означает «создать изменяемую переменную с именем *smallNum* и типом *Short*».

Аналогичным образом для объявления переменной с типом `Byte` используется код следующего вида:

```
var tinyNum: Byte
```

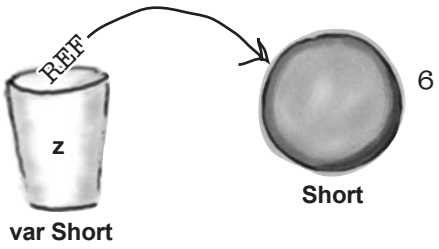
Явно объявляя тип переменной, вы передаете компилятору всю необходимую информацию для создания переменной: имя, тип и возможность изменения.

Объявление типа ВМЕСТЕ С присваиванием значения

В приведенных выше примерах переменные создавались без присваивания значения. Если вы хотите явно объявить тип переменной *и* присвоить ей значение — это тоже возможно. Например, следующая команда создает переменную типа `Short` с именем `z` и присваивает ей значение 6:

```
var z: Short = 6
```

В этом примере создается переменная с именем `z` и типом `Short`. Значение переменной 6 достаточно мало, чтобы поместиться в типе `Short`, поэтому будет создан объект `Short` со значением 6. Затем ссылка на объект `Short` сохраняется в переменной.



Когда вы присваиваете значение переменной, следите за тем, чтобы присваиваемое значение было совместимо с переменной. Проблема более подробно рассматривается на следующей странице.

Присваивание исходного значения переменной называется инициализацией. Переменная ДОЛЖНА быть инициализирована перед использованием, иначе компилятор сообщит об ошибке. Например, следующий код не будет компилироваться, потому что x не было присвоено исходное значение:

```
var x: Int
var y = x + 6
```

x не присвоено значение, и это огорчает компилятор.

Используйте значение, соответствующее типу переменной

Как говорилось ранее в этой главе, компилятор отслеживает тип переменной для предотвращения неуместных операций, которые могут привести к ошибкам в коде. Например, если вы попытаетесь присвоить значение с плавающей точкой — допустим, 3.12 — целочисленной переменной, компилятор откажется компилировать ваш код. Так, следующий код работать не будет:

```
var x: Int = 3.12
```

Компилятор понимает, что значение 3.12 невозможно разместить в `Int` без потери точности (то есть потери всей дробной части), поэтому отказывается компилировать код.

То же самое происходит при попытке разместить большое число в переменной, которая для этого числа слишком мала — компилятор огорчается. Например, если вы попробуете присвоить значение 500 переменной типа `Byte`, компилятор выдаст сообщение об ошибке:

```
//Не получится
var tinyNum: Byte = 500
```

Итак, чтобы присвоить литеральное значение переменной, необходимо убедиться в том, что это значение совместимо с типом переменной. Это особенно важно, когда значение одной переменной присваивается другой переменной. Сейчас мы рассмотрим эту ситуацию.

Компилятор Kotlin разрешает присвоить значение переменной только в том случае, если значение совместимо с переменной. Если значение слишком велико или относится к неподходящему типу, код не будет компилироваться.

Часто Задаваемые Вопросы

В: В Java числа являются примитивными типами, так что в переменной хранится само число, а не ссылка. В Kotlin не так?

О: Нет, не так. В Kotlin числа являются объектами, а в переменной хранится не сам объект, а ссылка на него.

В: Почему в Kotlin так важен тип переменной?

О: Потому что он делает код более безопасным и более устойчивым для ошибок. Возможно, кому-то это покажется мелочью, но поверьте нам — это полезно!

В: В Java примитивы `char` могут интерпретироваться как числа. А с типом `Char` в Kotlin это возможно?

О: Нет, тип `Char` в Kotlin содержит символы, а не числа. Запомните раз и навсегда: Kotlin — не Java.

В: Переменной можно присвоить любое имя?

О: Нет. Правила достаточно гибкие, но вы не сможете, допустим, присвоить переменной имя, совпадающее с зарезервированным словом. Называя переменную *while*, вы сами напрашиваетесь на неприятности. Впрочем, если вы попытаетесь присвоить переменной недействительное имя, IntelliJ IDEA немедленно обнаружит проблему.

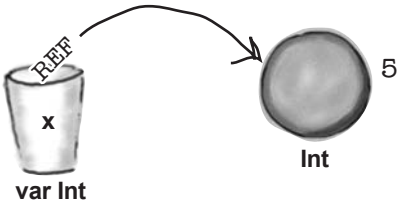
Присваивание значения другой переменной

Если значение одной переменной присваивается другой переменной, необходимо проследить за совместимостью их типов. Чтобы понять, почему это так, рассмотрим следующий пример:

```
var x = 5
var y = x
var z: Long = x
```

1 **var x = 5**

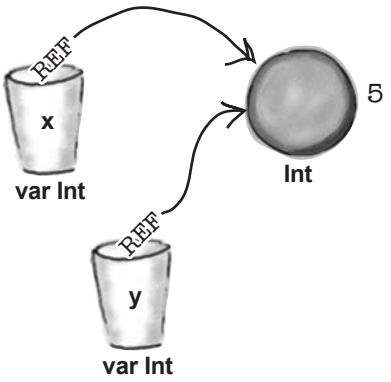
Команда создает переменную `Int` с именем `x` и объект `Int` со значением 5. Переменная `x` содержит ссылку на этот объект.



---

2 **var y = x**

Компилятор видит, что `x` является объектом `Int`, а следовательно, переменная `y` также должна иметь тип `Int`. Вместо создания второго объекта `Int` значение переменной `x` присваивается переменной `y`. Но что это означает на практике? По сути это означает «Взять биты из `x`, скопировать их и сохранить копию в `y`». **Следовательно, в `x` и `y` будут храниться ссылки на один и тот же объект**.



---

3 **var z: Long = x**

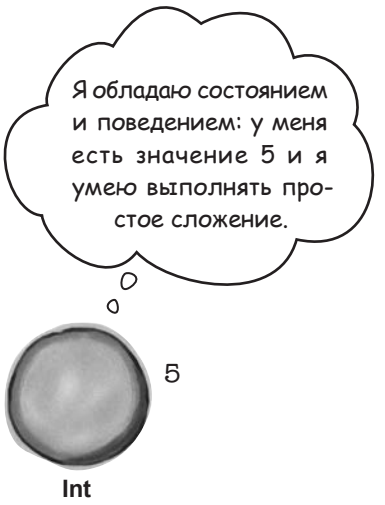
Эта строка сообщает компилятору, что вы хотите создать новую переменную `Long` с именем `z` и присвоить ей значение `x`. Но тут возникает проблема: переменная `x` содержит ссылку на объект `Int` со значением 5, а не объект `Long`. Мы знаем, что объект содержит значение 5, и что значение 5 помещается в объекте `Long`. Но поскольку переменная `z` отличается по типу от объекта `Int`, компилятор расстраивается и отказывается компилировать код.

Как же присвоить значение одной переменной другой переменной, если эти переменные относятся к разным типам?

Значение необходимо преобразовать

Предположим, вы хотите присвоить значение переменной `Int` другой переменной типа `Long`. Компилятор не позволит присвоить значение напрямую, так как две переменные относятся к разным типам: переменная `Long` может хранить только ссылку на объект `Long`, так что при попытке присвоить ей `Int` код не будет компилироваться.

Чтобы код компилировался, сначала необходимо преобразовать значение к правильному типу. А значит, если вы хотите присвоить значение переменной `Int` другой переменной, `Long`, сначала необходимо преобразовать ее значение к типу `Long`. Для этой цели используются *функции* объекта `Int`.



Объект обладает состоянием и поведением

Объект по определению обладает **состоянием** и **поведением**.

Под *состоянием* объекта понимаются данные, связанные с этим объектом: его свойства и значения. Например, числовой объект обладает числовым значением: 5, 42, 3.12 и т. д. (в зависимости от типа объекта). Значение объекта `Char` состоит из одного символа. Значение `Boolean` равно либо `true`, либо `false`.

Поведение объекта описывает то, что объект может делать, или то, что можно делать с ним. Например, строку `String` можно преобразовать к верхнему регистру. Числовые объекты умеют выполнять основные математические операции и преобразовывать свое значение в объект другого числового типа. Поведение объекта доступно через его функции.

Как преобразовать числовое значение к другому типу

В нашем примере значение переменной `Int` должно быть преобразовано в `Long`. У каждого числового объекта имеется функция с именем `toLong()`, которая берет значение объекта и использует его для создания нового объекта `Long`. Итак, если вы хотите присвоить значение переменной `Int` другой переменной, `Long`, для этого используется код следующего вида:

```
var x = 5
var z: Long = x.toLong()
```

Оператор «точка».

Оператор «точка» (`.`) используется для вызова функций объекта. Таким образом, запись `x.toLong()` означает «Перейти к объекту, ссылка на который хранится в переменной `x`, и вызвать его функцию `toLong()`».

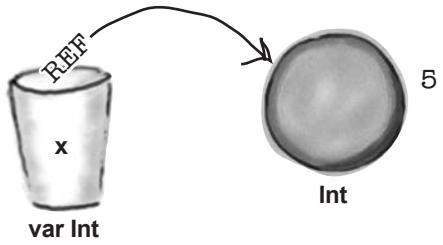
На следующей странице мы разберемся, что делает этот код.

Каждый числовой тип содержит следующие функции преобразования: toByte( ), toShort( ), toInt( ), toLong( ), toFloat( ) и toDouble( ).

Что происходит при преобразовании значений

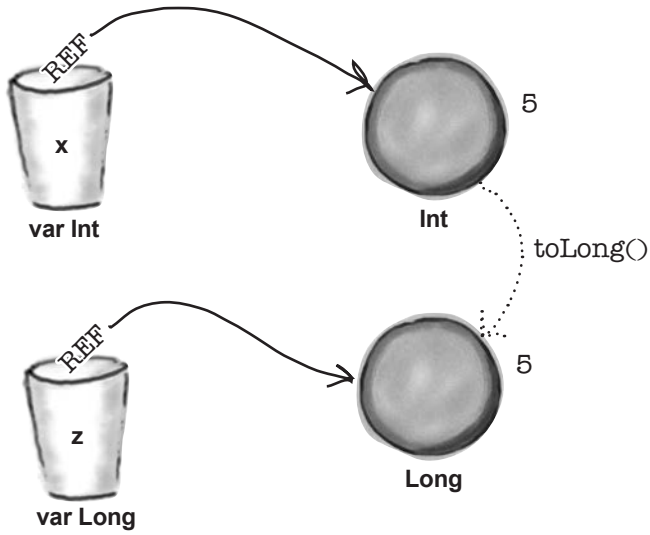
1 **`var x = 5`**

Создается переменная `Int` с именем `x` и объект `Int` со значением 5. В `x` сохраняется ссылка на этот объект.



2 **`var z: Long = x.toLong()`**

Создается новая переменная `Long` с именем `z`. Вызывается функция `toLong()` для объекта `x`, которая создает новый объект `Long` со значением 5. Ссылка на объект `Long` сохраняется в переменной `z`.



Функция toLong() создает новый объект Long с таким же значением, как у объекта Int.

Такое решение хорошо работает, если вы преобразуете значение к типу большей разрядности. Но что делать, если новый объект слишком мал для хранения значения?

Осторожнее с преобразованием

Пытаться поместить большое значение в маленькую переменную — все равно что пытаться вылить кувшин кофе в маленькую чашку. Часть кофе поместится в чашку, но большая часть прольется.

Допустим, вы хотите поместить значение `Long` в `Int`. Как было показано ранее, тип `Long` способен хранить числа бóльшие, чем тип `Int`.

Если значение `Long` лежит в диапазоне допустимых значений `Int`, преобразование из `Long` в `Int` не создаст проблем. Например, преобразование значения `Long` 42 в `Int` даст `Int` со значением 42:

```
var x = 42L
var y: Int = x.toInt()  //Значение 42
```

Но если значение `Long` слишком велико для `Int`, компилятор его усекает, и у вас остается нечто странное (хотя и пригодное для вычислений). Например, при попытке преобразовать значение `Long` 1234567890123 в `Int` полученное число `Int` будет иметь значение 1912276171:

В вычислениях используются знаки, биты, двоичные представления и другие нетривиальные штуки, на которые мы не будем отвлекаться. Впрочем, если вам будет интересно, поищите информацию по темам «дополнительный код» или «дополнение до 2».

```
var x = 1234567890123
var y: Int = x.toInt()  //Значение 1912276171!
```

Компилятор считает, что вы действуете сознательно, так что код откомпилируется. Теперь предположим, что у вас есть число с плавающей точкой и вы просто хотите отделить его целую часть. Если преобразовать число в `Int`, то компилятор отсечет всю дробную часть:

```
var x = 123.456
var y: Int = x.toInt()  //Значение 123
```

Важный урок: при преобразовании числовых значений из одного типа в другой следует позаботиться о том, чтобы новый тип был достаточно большим для преобразуемого значения, иначе вы рискуете получить неожиданные результаты в своем коде.

Итак, вы узнали, как работают переменные, и приобрели некоторый опыт использования базовых типов Kotlin. Попробуйте выполнить упражнение.

Возьми в руку карандаш

Приведенная ниже функция `main` не компилируется. Обведите строки, содержащие недопустимый код, и объясните, что именно препятствует компиляции кода.

```
fun main(args: Array<String>) {

    var x: Int = 65.2

    var isPunk = true

    var message = 'Hello'

    var y = 7

    var z: Int = y

    y = y + 50

    var s: Short

    var bigNum: Long = y.toLong()

    var b: Byte = 2

    var smallNum = b.toShort()

    b = smallNum

    isPunk = "false"

    var k = y.toDouble()

    b = k.toByte()

    s = 0b10001

}
```

Возьми в руку карандаш Решение

Приведенная ниже функция `main` не компилируется. Обведите строки, содержащие недопустимый код, и объясните, что именно препятствует компиляции кода.

```
fun main(args: Array<String>) {

    var x: Int = 65.2

    var isPunk = true

    var message = 'Hello'

    var y = 7

    var z: Int = y

    y = y + 50

    var s: Short

    var bigNum: Long = y.toLong()

    var b: Byte = 2

    var smallNum = b.toShort()

    b = smallNum

    isPunk = "false"

    var k = y.toDouble()

    b = k.toByte()

    s = 0b10001

}
```

Значение 65.2 недопустимо для Int.

Одинарные кавычки используются для определения типа Char, содержащего отдельные символы.

Переменная smallNum имеет тип Short, поэтому ее значение не может быть присвоено переменной Byte.

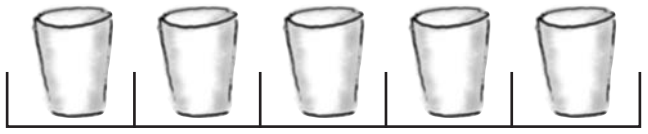
Переменная isPunk имеет тип Boolean, поэтому значение false не должно заключаться в двойные кавычки.

Сохранение значений в массивах

Есть еще одна разновидность объектов, о которой мы хотим рассказать вам, — массив. Допустим, вы хотите сохранить названия пятидесяти сортов мороженого или коды всех книг в библиотеке. Хранить всю эту информацию в переменных было бы неудобно. В такой ситуации следует использовать массивы.

Массивы идеально подходят для простого хранения однородных групп. Они легко создаются, а вы можете быстро обратиться к любому элементу массива.

Продолжая предыдущую аналогию, массив можно представить себе в виде подноса со стаканами, где каждый элемент массива соответствует одной переменной:

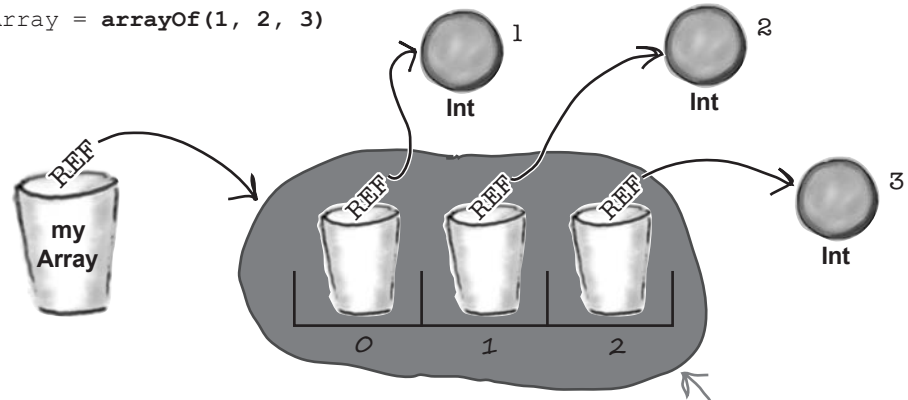


Массив можно представить в виде подноса со стаканами.

Как создать массив

Массивы создаются функцией `arrayOf()`. В следующем примере эта функция используется для создания массива с тремя элементами (переменные `Int` 1, 2 и 3), который присваивается переменной с именем `myArray`:

```
var myArray = arrayOf(1, 2, 3)
```



Массив является объектом, а в переменной хранится ссылка на него.

Чтобы получить значение элемента массива, обратитесь к переменной массива с указанием индекса. Например, вот как выводится значение первого элемента:

```
println(myArray[0])
```

Для получения размера массива используется функция:

```
myArray.size
```

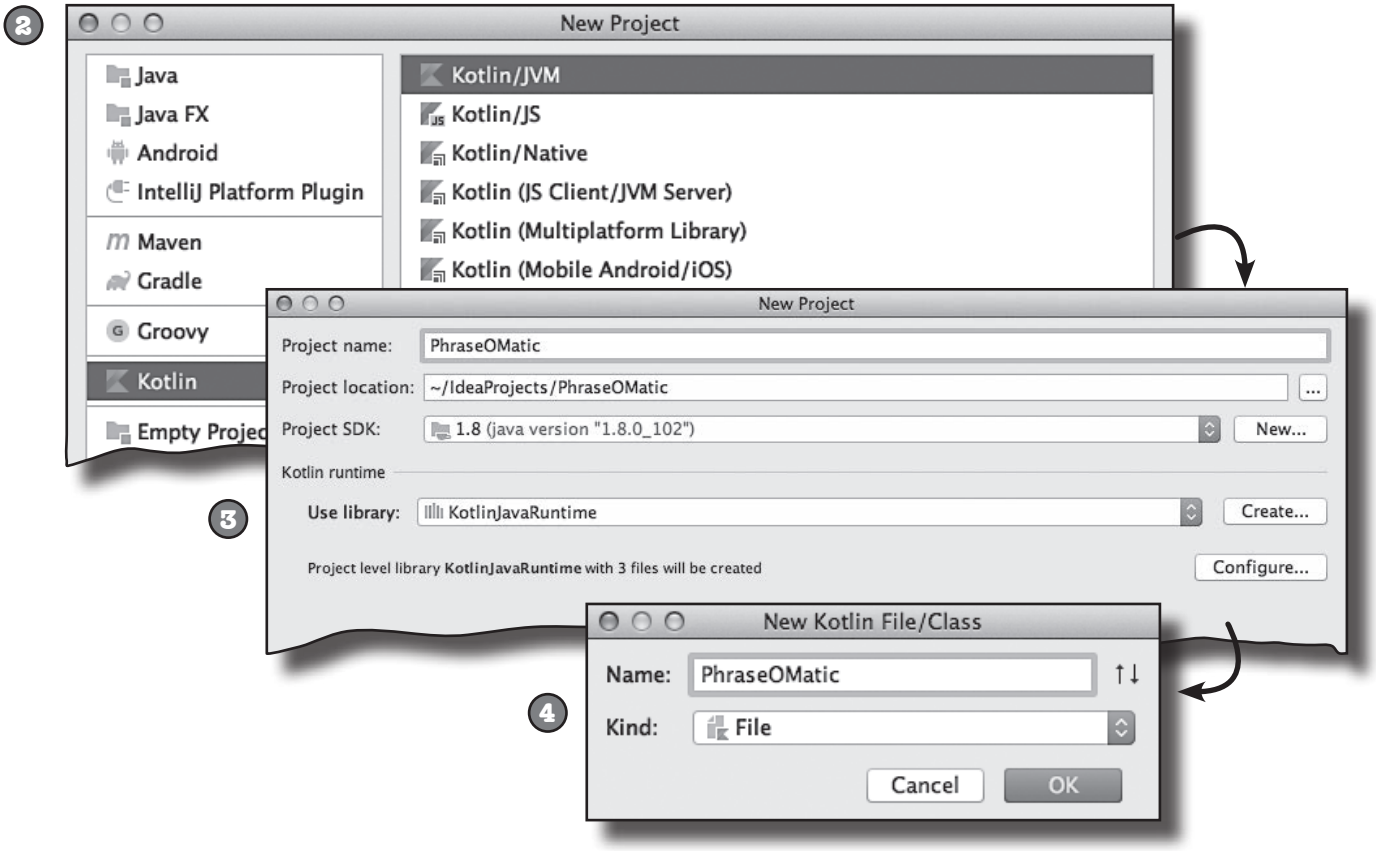
На следующей странице вся новая информация будет задействована при создании серьезного коммерческого приложения — Phrase-O-Matic.

Построение приложения Phrase-O-Matic

Мы создадим новое приложение, генерирующее эффектные рекламные слоганы.

Начните с создания нового проекта в IntelliJ IDEA:

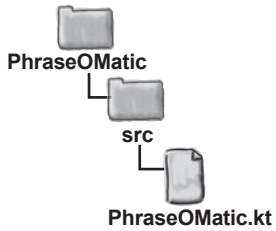
1. Откройте IntelliJ IDEA и выберите команду «Create New Project» на заставке. При этом запускается программа-мастер, описанная в главе 1.
2. Когда мастер выдаст соответствующий запрос, выберите настройки создания проекта Kotlin, предназначенного для JVM.
3. Присвойте проекту имя «PhraseOMatic», подтвердите остальные значения по умолчанию и щелкните на кнопке Finish.
4. Когда проект появится в IDE, создайте новый файл Kotlin с именем *PhraseOMatic.kt* — выделите папку *src*, откройте меню File и выберите команду New → Kotlin File/Class. Когда будет предложено, введите имя файла «PhraseOMatic» и выберите вариант File в группе Kind.



Добавление кода в файл PhraseOMatic.kt

Код Phrase-O-Matic состоит из функции `main`, которая создает три массива слов, случайным образом выбирает одно слово из каждого массива и объединяет их в одну строку. Добавьте приведенный ниже код в файл *PhraseOMatic.kt*:

```
fun main(args: Array<String>){
    val wordArray1 = arrayOf("24/7", "multi-tier", "B-to-B", "dynamic", "pervasive")
    val wordArray2 = arrayOf("empowered", "leveraged", "aligned", "targeted")
    val wordArray3 = arrayOf("process", "paradigm", "solution", "portal", "vision")

    val arraySize1 = wordArray1.size
    val arraySize2 = wordArray2.size
    val arraySize3 = wordArray3.size

    val rand1 = (Math.random() * arraySize1).toInt()
    val rand2 = (Math.random() * arraySize2).toInt()
    val rand3 = (Math.random() * arraySize3).toInt()

    val phrase = "${wordArray1[rand1]} ${wordArray2[rand2]} ${wordArray3[rand3]}"
    println(phrase)
}
```



Вы уже знаете, как работает этот код, но в нем есть пара строк, на которые следует обратить внимание.

Прежде всего, строка

```
val rand1 = (Math.random() * arraySize1).toInt()
```

генерирует случайное число. `Math.random()` возвращает случайное число в диапазоне от 0 до (почти) 1, которое для получения индекса необходимо умножить на количество элементов в массиве. После этого результат преобразуется в целое число вызовом `toInt()`.

Возможные результаты

- multi-tier leveraged solution
- dynamic targeted vision
- 24/7 aligned paradigm
- B-to-B empowered portal

Наконец, строка

```
val phrase = "${wordArray1[rand1]} ${wordArray2[rand2]} ${wordArray3[rand3]}"
```

использует **строковый шаблон** для того, чтобы выбрать три слова и объединить их в строку. Строковые шаблоны рассматриваются на следующей странице. Затем мы расскажем, что еще можно сделать с массивами.

Строковые шаблоны под увеличительным стеклом

Строковые шаблоны предоставляют простой и быстрый механизм включения переменных в строковые значения.

Чтобы включить значение переменной в строку, поставьте перед именем переменной префикс `$`. Например, для включения в строку значения переменной `Int` с именем `x` используется команда:

```
var x = 42
var value = "Value of x is $x"
```

Также строковые шаблоны могут использоваться для обращения к свойствам объектов и вызова их функций. В этом случае выражение заключается в фигурные скобки. Например, следующая команда включает в строку размер массива и значение его первого элемента:

```
var myArray = arrayOf(1, 2, 3)
var arraySize = "myArray has ${myArray.size} items"
var firstItem = "The first item is ${myArray[0]}"
```

Обратите внимание: выражение, вычисляемое внутри строки, заключается в фигурные скобки {}.

Строковые шаблоны даже могут использоваться для вычисления более сложных выражений внутри строк. Скажем, в следующем примере выражение `if` используется для включения разного текста в зависимости от размера массива `myArray`:

```
var result = "myArray is ${if (myArray.size > 10) "large" else "small"}"
```

Строковые шаблоны позволяют строить сложные строки с минимальным объемом кода.

Часто Задаваемые Вопросы

В: `Math.random()` — стандартный способ получения случайных чисел в Kotlin?

О: Это зависит от версии Kotlin, которую вы используете. До выхода версии 1.3 в Kotlin не существовало встроенного способа генерирования случайных чисел. Впрочем, в приложениях, выполняемых на JVM, можно было использовать метод `random()` из библиотеки Java `Math` library, как это сделано у нас. В версии 1.3 и выше вы можете использовать встроенные в Kotlin функции `Random`. Например, в следующем коде функция `Random` `nextInt()` используется для генерирования случайного числа `Int`:

```
kotlin.random.Random.nextInt()
```

В этой книге мы решили использовать `Math.random()` для генерирования случайных чисел, так как это решение будет работать со всеми версиями Kotlin, работающими на JVM.

Компилятор определяет тип массива по значениям элементов

Вы уже знаете, как создать массив и обращаться к его элементам. Посмотрим, как обновить значения его элементов.

Допустим, имеется массив с элементами `Int` с именем `myArray`:

```
var myArray = arrayOf(1, 2, 3)
```

Чтобы обновить второй элемент и присвоить ему значение 15, используйте код следующего вида:

```
myArray[1] = 15
```

Но здесь есть один нюанс: **значение должно иметь правильный тип**.

Компилятор проверяет тип каждого элемента в массиве и раз и навсегда определяет тип элементов, которые должны храниться в массиве. В предыдущем примере массив объявляется со значениями `Int`, поэтому компилятор делает вывод, что массив будет содержать только значения типа `Int`. При попытке сохранить в массиве что-либо кроме `Int` код не будет компилироваться:

```
myArray[1] = "Fido"  //Не компилируется
```

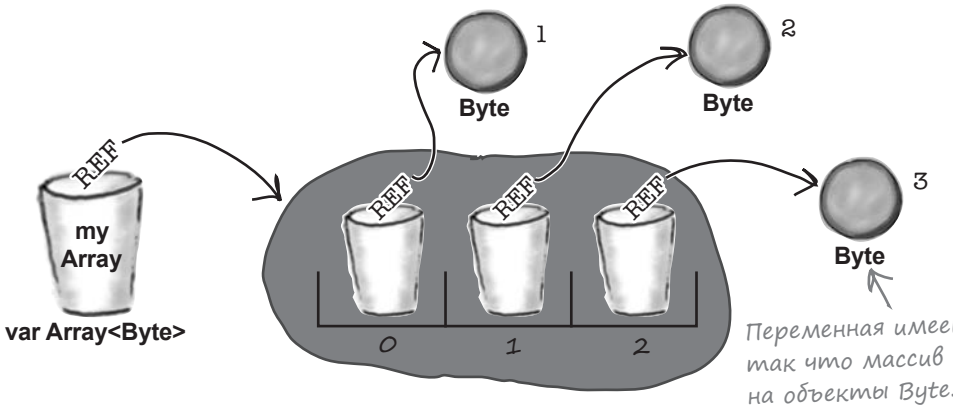
В массивах хранятся элементы определенного типа. Вы можете поручить определение типа компилятору на основании его значений или же явно определить тип при помощи Array<Type>.

Явное определение типа массива

Как и в случае с другими переменными, вы можете явно определить тип элементов, которые могут храниться в массиве. Допустим, вы хотите объявить массив для хранения значений `Byte`. Код будет выглядеть примерно так:

```
var myArray: Array<Byte> = arrayOf(1, 2, 3)
```

Код `Array<Byte>` сообщает компилятору, что вы хотите создать массив для хранения переменных `Byte`. В общем случае тип создаваемого массива просто указывается в угловых скобках (<>).



var означает, что переменная может указывать на другой массив

Остался еще один вопрос: чем же отличаются ключевые слова `val` и `var` при объявлении массива?

Как вам уже известно, в переменной хранится ссылка на объект. При объявлении переменной с ключевым словом `var` переменную можно изменить, чтобы в ней хранилась ссылка на другой объект. Если переменная содержит ссылку на массив, это означает, что переменную можно обновить, чтобы в ней хранилась ссылка на другой массив того же типа. Например, следующий код абсолютно правилен и откомпилируется без ошибок:

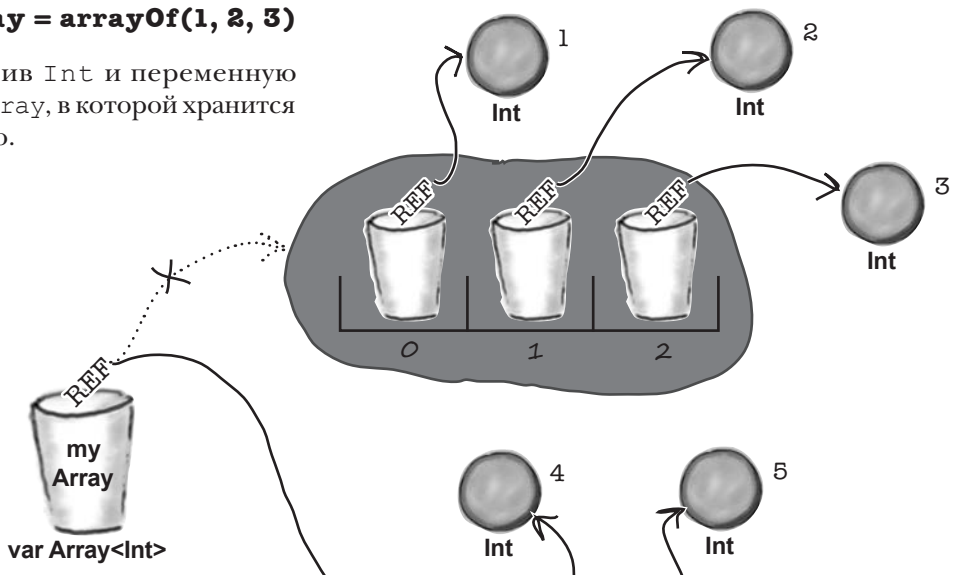
```
var myArray = arrayOf(1, 2, 3)
myArray = arrayOf(4, 5)
```

Новый массив.

Последовательно разберем, что же здесь происходит.

1 **`var myArray = arrayOf(1, 2, 3)`**

Создает массив `Int` и переменную с именем `myArray`, в которой хранится ссылка на него.



2 **`myArray = arrayOf(4, 5)`**

Создает новый масссив с элементами `Int`. Ссылка на новый массив сохраняется в переменной `myArray` и заменяет предыдущую ссылку.

А что произойдет, если переменная будет объявлена с ключевым словом `val`?

val означает, что переменная всегда будет указывать на один и тот же массив...

Если массив объявляется с ключевым словом `val`, вы не сможете обновить переменную так, чтобы в ней хранилась ссылка на другой массив. Например, следующий код не будет компилироваться:

```
val myArray = arrayOf(1, 2, 3)
myArray = arrayOf(4, 5, 6)
```

Если переменная массива объявляется с ключевым словом val, ее нельзя изменить и заставить указывать на другой массив.

После того как переменной будет присвоен массив, ссылка на этот массив так и будет храниться в переменной до конца работы приложения. Но хотя в переменной всегда хранится ссылка на один и тот же массив, **сам массив при этом может изменяться**.

...но переменные в массиве при этом можно обновлять

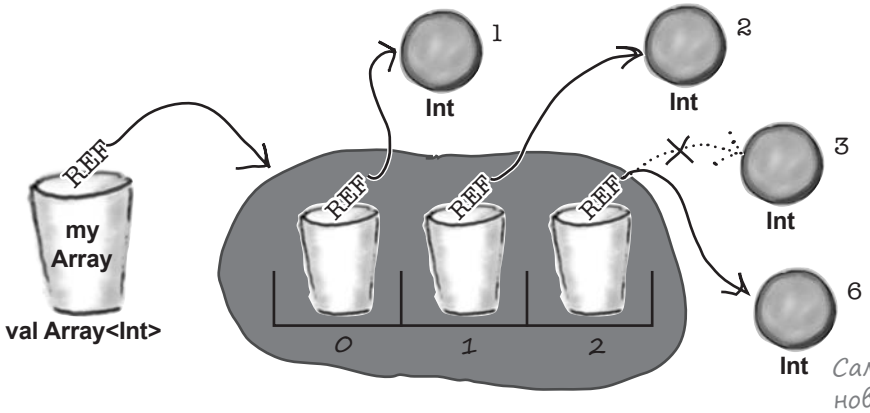
Когда вы объявляете переменную с ключевым словом `val`, вы тем самым сообщаете компилятору, что создается переменная, которая не может повторно использоваться для хранения других значений. Но эта инструкция относится только к самой переменной. Если переменная содержит ссылку на массив, элементы в массиве все равно могут изменяться.

Для примера возьмем следующий код:

```
val myArray = arrayOf(1, 2, 3)
myArray[2] = 6
```

Изменяет третий элемент в массиве.

Создает переменную с именем `myArray`, в которой хранится ссылка на массив `Int`. Переменная объявлена с ключевым словом `val`, так что переменная должна содержать ссылку на один и тот же массив на протяжении всего времени работы программы. После этого третий элемент массива успешно обновляется значением 6, так как сам массив может обновляться:



Сам массив может обновляться, хотя переменная и была объявлена с ключевым словом val.

Итак, теперь вы знаете, как работают массивы в Kotlin; можно перейти к упражнениям.

Объявление переменной с ключевым словом val означает, что переменная не может повторно использоваться для другого объекта. При этом сам объект можно изменять.

СТАНЬ компилятором

Каждый блок кода на этой странице представляет полный исходный файл. Попробуйте представить себя на месте компилятора и определить, будет ли компилироваться каждый из этих файлов. Если какие-то файлы не компилируются, то как бы вы их исправили?

A

```
fun main(args: Array<String>) {

    val hobbits = arrayOf("Frodo", "Sam", "Merry", "Pippin")
    var x = 0;

    while (x < 5) {
        println("${hobbits[x]} is a good Hobbit name")
        x = x + 1
    }

}
```

Требуется вывести строку с сообщением для каждого имени из массива hobbits.

B

```
fun main(args: Array<String>) {

    val firemen = arrayOf("Pugh", "Pugh", "Barney McGrew", "Cuthbert", "Dibble", "Grub")
    var firemanNo = 0;

    while (firemanNo < 6) {
        println("Fireman number $firemanNo is $firemen[firemanNo]")
        firemanNo = firemanNo + 1
    }

}
```

Требуется вывести строку с сообщением для каждого имени в массиве firemen.

Ответы на с. 85.

Развлечения с магнитами

На холодильнике была выложена работоспособная программа Kotlin. Сможете ли вы заполнить пропуски, чтобы функция выдавала следующий результат:

```
Fruit = Banana
Fruit = Blueberry
Fruit = Pomegranate
Fruit = Cherry
```

```
fun main(args: Array<String>) {
```

Магниты выставляются здесь.

```
}
```

```
x = x + 1
```

```
y = index[x]
```

```
var x = 0
```

```
while (x < 4) {
```

```
var y: Int
```

```
val index = arrayOf(1, 3, 4, 2)
```

```
}
```

```
println("Fruit = ${fruit[y]}")
```

```
val fruit = arrayOf("Apple", "Banana", "Cherry", "Blueberry", "Pomegranate")
```

Ответы на с. 86.

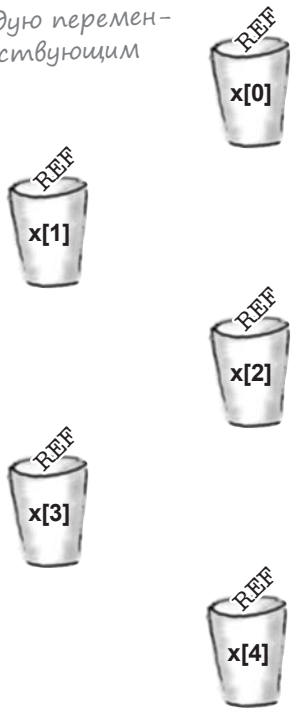
Путаница со ссылками

Ниже приведена короткая программа Kotlin. При достижении строки `//За работу` будут созданы некоторые объекты и переменные. Ваша задача — определить, какие из переменных относятся к тем или иным объектам при достижении строки `//За работу`. На некоторые объекты могут указывать сразу несколько ссылок. Проведите линии, соединяющие переменные с соответствующими объектами.

```
fun main(args: Array<String>) {
    val x = arrayOf(0, 1, 2, 3, 4)
    x[3] = x[2]
    x[4] = x[0]
    x[2] = x[1]
    x[1] = x[0]
    x[0] = x[1]
    x[4] = x[3]
    x[3] = x[2]
    x[2] = x[4]
    //За работу
}
```

Переменные:

Соедините каждую переменную с соответствующим объектом.



Объекты:

Ответы на с. 87.

СТАНЬ компилятором.—Решение

Каждый блок кода на этой странице представляет полный исходный файл. Попробуйте представить себя на месте компилятора и определить, будет ли компилироваться каждый из этих файлов. Если какие-то файлы не компилируются, то как бы вы их исправили?

A

```
fun main(args: Array<String>) {

    val hobbits = arrayOf("Frodo", "Sam", "Merry", "Pippin")
    var x = 0;

    while (x < ~~5~~ 4) {
        println("${hobbits[x]} is a good Hobbit name")
        x = x + 1
    }

}
```

Код компилируется, но при запуске выдает ошибку. Помните, что массивы начинаются с элемента с индексом 0, а индекс последнего элемента равен (size - 1).

B

```
fun main(args: Array<String>) {

    val firemen = arrayOf("Pugh", "Pugh", "Barney McGrew", "Cuthbert", "Dibble", "Grub")
    var firemanNo = 0;

    while (firemanNo < 6) {
        println("Fireman number $firemanNo is ${firemen[firemanNo]}")
        firemanNo = firemanNo + 1
    }

}
```

Для включения имен в строку необходимо заключить выражение firemen[firemanNo] в фигурные скобки.

Развлечения с магнитами. Решение

На холодильнике была выложена работоспособная программа Kotlin. Сможете ли вы заполнить пропуски, чтобы функция выдавала следующий результат:

```
Fruit = Banana
Fruit = Blueberry
Fruit = Pomegranate
Fruit = Cherry
```

```
fun main(args: Array<String>) {

    val index = arrayOf(1, 3, 4, 2)
    val fruit = arrayOf("Apple", "Banana", "Cherry", "Blueberry", "Pomegranate")
    var x = 0
    var y: Int
    while (x < 4) {
        y = index[x]
        println("Fruit = ${fruit[y]}")
        x = x + 1
    }

}
```

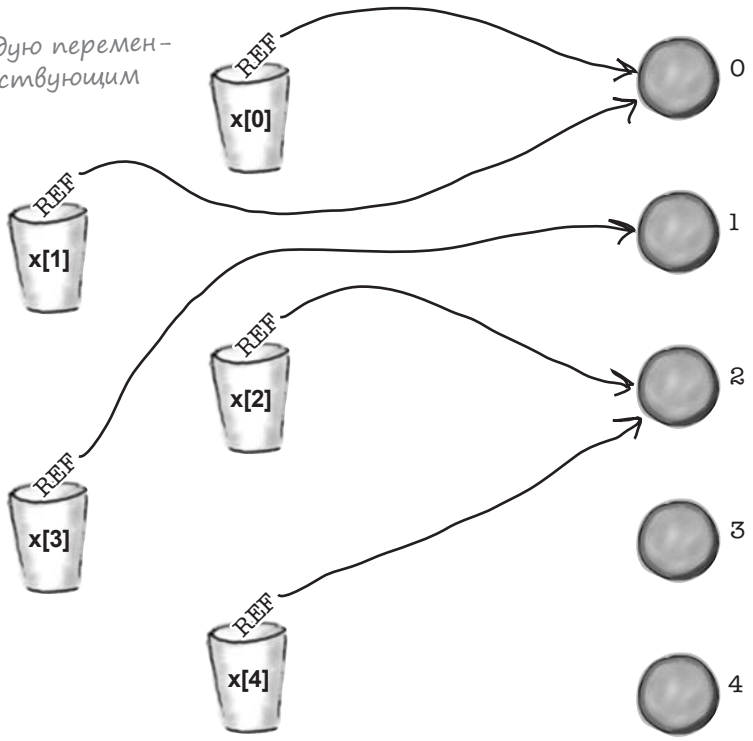
Путаница со ссылками. Решение

Ниже приведена короткая программа Kotlin. При достижении строки `//За работу` будут созданы некоторые объекты и переменные. Ваша задача — определить, какие из переменных относятся к тем или иным объектам при достижении строки `//За работу`. На некоторые объекты могут указывать сразу несколько ссылок. Проведите линии, соединяющие переменные с соответствующими объектами.

```
fun main(args: Array<String>) {
    val x = arrayOf(0, 1, 2, 3, 4)
    x[3] = x[2]
    x[4] = x[0]
    x[2] = x[1]
    x[1] = x[0]
    x[0] = x[1]
    x[4] = x[3]
    x[3] = x[2]
    x[2] = x[4]
    //За работу
}
```

Переменные: **Объекты:**

Соедините каждую переменную с соответствующим объектом.



Ваш инструментарий Kotlin

Глава 2 осталась позади, а ваш инструментарий пополнился базовыми типами и переменными.

Весь код для этой главы можно загрузить по адресу https://tinyurl.com/HFKotlin.

ГЛАВА 2

КЛЮЧЕВЫЕ МОМЕНТЫ

- Чтобы создать переменную, компилятор должен знать ее имя, тип и возможность ее повторного использования.
- Если тип переменной не задан явно, то компилятор определяет его по значению.
- В переменной хранится ссылка на объект.
- Объект обладает состоянием и поведением. Доступ к его поведению предоставляется через функции.
- Определение переменной с ключевым словом `var` означает, что ссылка на объект, хранящаяся в переменной, может быть изменена. Определение переменной с ключевым словом `val` означает, что переменная содержит одну и ту же ссылку до конца работы программы.
- В Kotlin поддерживаются следующие базовые типы: `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `Boolean`, `Char` и `String`.
- Чтобы явно определить тип переменной, поставьте двоеточие после имени переменной, а затем укажите ее тип:
  ```
  var tinyNum: Byte
  ```
- Значение может быть присвоено только переменной, обладающей совместимым типом.
- Один числовой тип можно преобразовать к другому числовому типу. Если значение не помещается в новый тип, происходит потеря точности.
- Для создания массивов используется функция `arrayOf`:
  ```
  var myArray = arrayOf(1, 2, 3)
  ```
- При обращении к элементу массива указывается его индекс: например `myArray[0]`. Первый элемент в массиве имеет индекс 0.
- Для получения размера массива используется конструкция `myArray.size`.
- Компилятор определяет тип массива по значениям его элементов. Вы также можете явно определить тип массива:
  ```
  var myArray: Array<Byte>
  ```
- Если массив определяется ключевым словом `val`, вы все равно можете определять элементы в массиве.
- Строковые шаблоны предоставляют простой и быстрый механизм включения переменных или результатов вычисления выражений в строку.

За пределами main

А теперь пришло время поближе познакомиться с функциями. До сих пор весь написанный нами код размещался в функции *main* приложения. Но если вы хотите, чтобы код был **лучше структурирован** и **проще в сопровождении**, необходимо знать, ***как разбить его на отдельные функции***. В этой главе на примере игры вы научитесь *писать функции* и *взаимодействовать* с ними из приложения. Вы узнаете, как писать компактные ***функции единичных выражений***. Попутно вы научитесь ***перебирать диапазоны и коллекции*** в мощных циклах *for*.

Построение игры: камень-ножницы-бумага

В примерах кода, рассматривавшихся до настоящего момента, код добавлялся в функцию `main` приложения. Как вам уже известно, эта функция запускает приложение, то есть становится его точкой входа.

Такой подход хорошо работал, пока мы изучали базовый синтаксис Kotlin, но в большинстве реальных приложений *код разделяется на функции*. Это делается по нескольким причинам:

- **Улучшение структуры кода.**
  Вы не сваливаете весь код в одну огромную функцию `main`, а разделяете его на фрагменты, с которыми удобно работать. Разделение сильно упрощает чтение и понимание кода.
- **Расширение возможности повторного использования кода.**
  Разделение кода на функции позволяет использовать его в других местах.

Существуют и другие причины, но эти две — самые важные.

Каждая функция представляет собой именованный блок кода, выполняющий конкретную задачу. Например, можно написать функцию `max` для определения большего из двух значений, а потом вызывать эту функцию в разных точках вашего приложения.

В этой главе мы поближе познакомимся с тем, как работают функции. Для этого мы построим классическую игру «Камень-ножницы-бумага».

Как работает игра

Цель: выбрать вариант, который побеждает вариант, выбранный компьютером.

Подготовка: при запуске приложения игра случайным образом выбирает один из трех вариантов: «Камень», «Ножницы» или «Бумага». Затем она предлагает *вам* выбрать один из этих вариантов.

Правила: игра сравнивает два варианта. Если они совпадают, игра заканчивается вничью. Если же варианты различны, победитель определяется по следующим правилам:

Варианты	Результат
Ножницы, бумага	«Ножницы» побеждают («ножницы режут бумагу»).
Камень, ножницы	«Камень» побеждает («камень разбивает ножницы»).
Бумага, камень	«Бумага» побеждает («бумага накрывает камень»).

Игра будет работать в окне вывода IDE.

Высокоуровневая структура игры

Прежде чем переходить к написанию кода, необходимо составить план работы приложения.

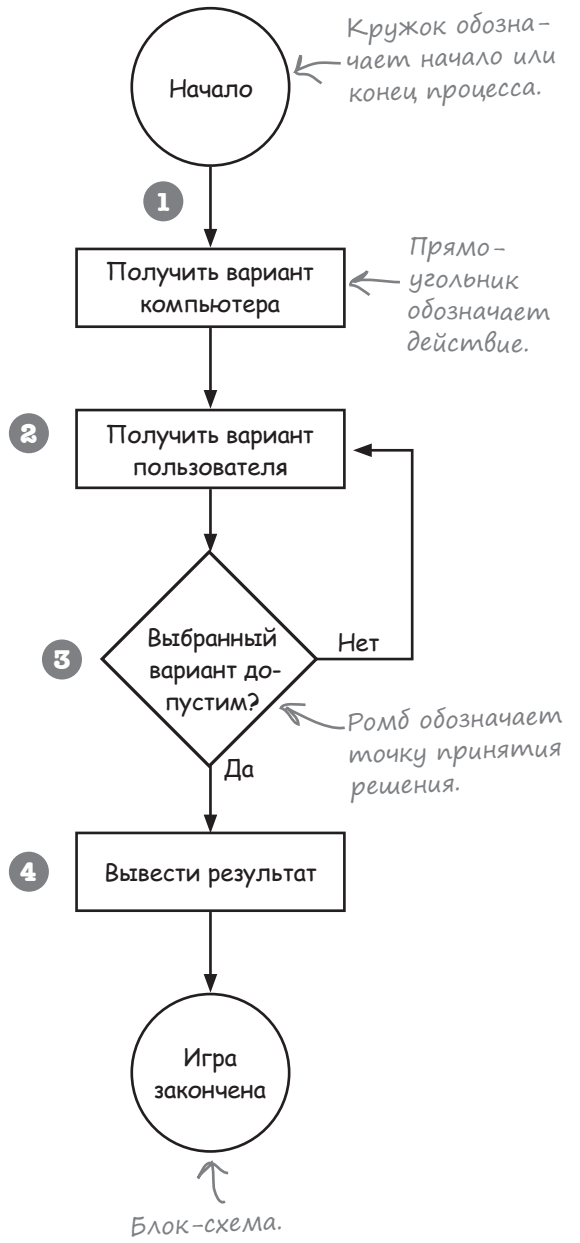
Сначала необходимо определить общий ход игры. Основная последовательность действий выглядит так:

1. **Вы запускаете игру.**
   Приложение случайным образом выбирает один из вариантов: «камень», «ножницы» или «бумага».

2. **Приложение запрашивает ваш вариант.**
   Вы вводите свое решение в окне вывода IDE.

3. **Приложение проверяет ваш выбор.**
   Если выбран недопустимый вариант, приложение возвращается к шагу 2 и предлагает ввести другой вариант. Это происходит до тех пор, пока не будет введен допустимый вариант.

4. **Игра выводит результат.**
   Она сообщает, какие варианты были выбраны вами и приложением, а также результат: выиграли вы, проиграли или же партия завершилась вничью.



Теперь, когда вы более четко представляете, как работает приложение, можно перейти к программированию.

Вот что мы собираемся сделать

При построении игры необходимо реализовать несколько высокоуровневых задач:

1. **Заставить игру выбрать вариант.**
   Мы создадим новую функцию с именем `getGameChoice`, которая будет случайным образом выбирать один из вариантов: «Камень», «Ножницы» или «Бумага».

2. **Запросить у пользователя выбранный им вариант.**
   Для этого мы напишем еще одну функцию с именем `getUserChoice`, которая будет запрашивать у пользователя выбранный им вариант. Функция проверяет введенный вариант, и если пользователь ввел недопустимое значение — запрашивает данные заново, пока не будет введен правильный вариант.

```
Please enter one of the following: Rock Paper Scissors.
Errr... dunno
You must enter a valid choice.
Please enter one of the following: Rock Paper Scissors.
Paper
```

3. **Вывести результат.**
   Мы напишем функцию с именем `printResult`, которая будет определять результат: выигрыш/проигрыш пользователя или ничья. Затем функция выводит результат.

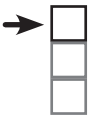
```
You chose Paper. I chose Rock. You win!
```

Все по порядку: создание проекта

Начнем с создания проекта приложения. Это делается точно так же, как в предыдущих главах.

Создайте новый проект Kotlin для JVM, присвойте проекту имя «Rock Paper Scissors». Создайте новый файл Kotlin с именем *Game.kt*: выделите папку *src*, откройте меню File и выберите команду New → Kotlin File/Class. Введите имя файла «Game» и выберите вариант File в группе Kind.

Проект создан, теперь можно переходить к написанию кода.



Выбор варианта игры

Первое, что нужно сделать, — заставить игру выбрать один из вариантов («Камень», «Ножницы» или «Бумага») случайным образом. Вот как это делается:

1. **Создайте массив, содержащий строки «Камень», «Ножницы» и «Бумага».**
   Мы добавим этот код в функцию `main` приложения.

2. **Создайте новую функцию `getGameChoice`, которая случайным образом выбирает один из вариантов.**

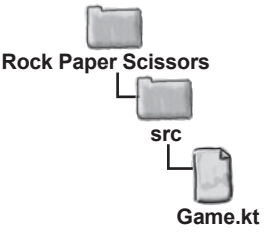
3. **Вызовите функцию `getGameChoice` из функции `main`.**

Начнем с создания массива.

Создание массива вариантов

Для создания массива будет использоваться функция `arrayOf`, как это делалось в предыдущей главе. Этот код будет добавлен в функцию `main` приложения, чтобы массив создавался при запуске приложения. Кроме того, мы сможем использовать его во всем коде, который будет написан в этой главе.

Чтобы создать функцию `main` и добавить массив, обновите свою версию *Game.kt* и приведите ее к следующему виду:




```
fun main(args: Array<String>) {
    val options = arrayOf("Rock", "Paper", "Scissors")
}
```

После создания массива необходимо определить новую функцию `getGameChoice`. Но перед этим необходимо разобраться в том, как создаются функции.

- Выбор игры
- Выбор пользователя
- Результат

Как создаются функции

Как вы узнали в главе 1, новые функции определяются ключевым словом `fun`, за которым следует имя функции. Например, если вы хотите создать новую функцию с именем `foo`, это делается примерно так:

'fun' сообщает Kotlin, что это функция.

```
fun foo() {
    //Здесь размещается ваш код
}
```

После того как вы определите функцию, ее можно будет вызывать в любой точке вашего приложения:

```
fun main(args: Array<String>) {
    foo()
}
```

Выполняет функцию с именем 'foo'.

Функции можно передавать данные

Иногда функции для выполнения ее операций требуется дополнительная информация. Скажем, если вы пишете функцию для определения большего из двух значений, эта функция должна знать эти два значения.

Чтобы сообщить компилятору, какие значения получает функция, вы указываете один или несколько **параметров**. Каждый параметр должен обладать именем и типом.

Например, следующее объявление сообщает, что функция `foo` получает один параметр `Int` с именем `param`:

Параметры объявляются в круглых скобках в начале функции.

```
fun foo(param: Int) {
    println("Parameter is $param")
}
```

После этого можно вызвать функцию и передать ей значение `Int`:

```
foo(6)
```

Функции foo передается значение '6'.

Если у функции есть **параметр**, **ей необходимо что-то передать**. И это «что-то» непременно должно быть значением подходящего типа. Следующий вызов функции работать не будет, потому что функция `foo` должна получать значение типа `Int`, а не `String`:

```
foo("Freddie")
```

Функции foo нельзя передать String, так как она принимает только Int.

Параметры и аргументы

В зависимости от вашего опыта программирования и личных предпочтений значения, передаваемые функции, могут называться *аргументами* или *параметрами*. Хотя в информатике существуют формальные различия между этими терминами, и умные парни в свитерах вам об этом охотно расскажут, у нас сейчас есть более серьезные дела. *Вы* можете называть их, как хотите (аргументами, параметрами, пончиками...), но мы будем использовать следующие термины:

Функция использует параметры. В точке вызова ей передаются аргументы.

Аргументы — то, что вы передаете функции. *Аргумент* (значение — например, 2 или «Pizza») попадает в *параметр*. А параметр представляет собой не что иное, как **локальную переменную**: переменную с именем и типом, которая используется внутри тела функции.

Функции можно передавать несколько значений

Если ваша функция должна получать несколько параметров, разделите их запятыми при объявлении, а также разделите запятыми аргументы при передаче их функции. Но что еще важнее, если функция имеет несколько параметров, ей необходимо передать аргументы правильных типов в правильном порядке.

Вызов функции с двумя параметрами и передача двух аргументов

```
fun main(args: Array<String>) {
    printSum(5, 6)
}

fun printSum(int1: Int, int2: Int) {
    val result = int1 + int2
    println(result)
}
```

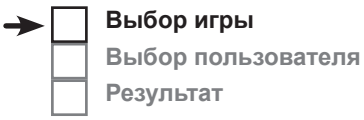
Аргументы передаются функции в том порядке, в котором вы их указали. Первый аргумент попадает в первый параметр, второй аргумент — во второй параметр и т. д.

Функции могут передаваться переменные при условии, что тип переменной соответствует типу параметра.

```
fun main(args: Array<String>) {
    val x: Int = 7
    val y: Int = 8
    printSum(x, y)
}

fun printSum(int1: Int, int2: Int) {
    val result = int1 + int2
    println(result)
}
```

Каждый передаваемый аргумент должен относиться к тому же типу, что и параметр, в котором он окажется.

Кроме передачи значений функции, также можно получать значения от нее. Посмотрим, как это делается.



Получение значений из функции

Если вы хотите получить значение из функции, его необходимо объявить. Например, вот как объявляется функция с именем `max`, которая возвращает значение `Int`:

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

: Int сообщает компилятору, что функция возвращает значение Int.

Чтобы вернуть значение, используйте ключевое слово 'return', за которым следует возвращаемое значение.

Если функция объявлена с возвращаемым значением, вы *должны* вернуть значение объявленного типа. Например, следующий код недопустим, потому что он возвращает `String` вместо `Int`:

```
fun max(a: Int, b: Int): Int {
    val maxValue = if (a > b) a else b
    return "Fish"
}
```

В объявлении функции указано, что она возвращает значение Int, поэтому компилятор расстроится при попытке вернуть что-то другое, например String.

Функции без возвращаемого значения

Если вы не хотите, чтобы функция возвращала значение, исключите возвращаемый тип из объявления функции или укажите возвращаемый тип `Unit`. Объявление возвращаемого типа `Unit` означает, что функция не возвращает значения. Например, следующие два объявления допустимы, и делают они одно и то же:

```
fun printSum(int1: Int, int2: Int) {
    val result = int1 + int2
    println(result)
}

fun printSum(int1: Int, int2: Int): Unit {
    val result = int1 + int2
    println(result)
}
```

: Unit означает, что функция не возвращает значения. Эта часть не обязательна, и ее можно опустить.

Если вы указали, что функция не имеет возвращаемого значения, следите за тем, чтобы она не делала этого. При попытке вернуть значение из функции без объявленного возвращаемого типа или с возвращаемым типом `Unit` код не будет компилироваться.

- [ ] Выбор игры
- [ ] Выбор пользователя
- [ ] Результат

Функции из единственного выражения

Если у вас имеется функция, тело которой состоит из одного выражения, код можно упростить, удалив фигурные скобки и команду `return` из объявления функции. Например, на предыдущей странице была представлена следующая функция для определения большего из двух значений:

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

Тело функции max состоит из единственного выражения, результат которого возвращается функцией.

Функция возвращает результат одного выражения `if`, а следовательно, ее можно переписать в следующем виде:

Используйте = для определения значения, возвращаемого функцией, и уберите фигурные скобки {}.

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

А поскольку компилятор может определить возвращаемый тип функции по выражению `if`, код можно дополнительно сократить, убрав из него `: Int`:

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

Компилятор знает, что a и b имеют тип Int, и поэтому может определить возвращаемый тип функции по выражению.

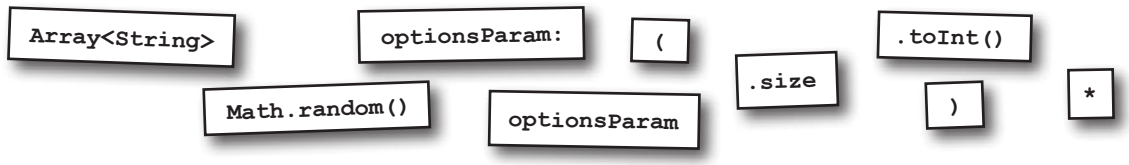
Создание функции getGameChoice

Теперь, когда вы научились создавать функции, посмотрим, удастся ли вам написать функцию `getGameChoice` для игры «Камень, ножницы, бумага». Попробуйте выполнить следующее упражнение.

Развлечения с магнитами

Функция `getGameChoice` получает один параметр (массив с элементами `String`) и возвращает один из элементов массива. Попробуйте составить код функции из магнитов.

```
fun getGameChoice(                                    ) =

    optionsParam[                                                    ]
```



Развлечения с магнитами. Решение

Функция `getGameChoice` получает один параметр (массив с элементами `String`) и возвращает один из элементов массива. Попробуйте составить код функции из магнитов.

- [ ] **Выбор игры**
- [ ] **Выбор пользователя**
- [ ] **Результат**

Функция получает один параметр: массив String.

```
fun getGameChoice( optionsParam: Array<String> ) =
    optionsParam[ ( Math.random() * optionsParam .size ) .toInt() ]
```

Выбрать один из элементов массива случайным образом.

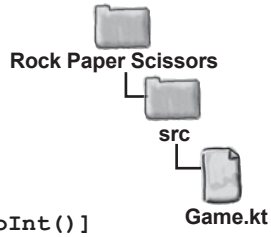
Добавление функции getGameChoice в файл Game.kt

Теперь вы знаете, как выглядит функция `getGameChoice`. Добавим ее в приложение и обновим функцию `main` так, чтобы она вызывала новую функцию. Обновите свою версию файла *Game.kt*, чтобы она соответствовала приведенной ниже (изменения выделены жирным шрифтом):

```
fun main(args: Array<String>) {
    val options = arrayOf("Rock", "Paper", "Scissors")
    val gameChoice = getGameChoice(options)
}

fun getGameChoice(optionsParam: Array<String>) =
        optionsParam[(Math.random() * optionsParam.size).toInt()]
```

Вызвать функцию getGameChoice и передать ей массив вариантов.

Необходимо добавить эту функцию.



Добавив функцию `getGameChoice` в приложение, посмотрим, что же происходит за кулисами при выполнении этого кода.

Часто Задаваемые Вопросы

В: Можно ли вернуть несколько значений из функции?

О: Функция может быть объявлена только с одним возвращаемым значением. Но если вы хотите вернуть, например, три значения `Int`, то объявленным типом может быть массив с элементами `Int` (`Array<Int>`). Поместите эти значения `Int` в массив и верните его из функции.

В: Обязательно ли что-то сделать с возвращаемым значением функции? Я могу его просто проигнорировать?

О: Kotlin не требует, чтобы возвращаемое значение использовалось в программе. Возможно, вы захотите вызвать функцию с возвращаемым типом даже в том случае, если возвращаемое значение вас не интересует. В этом случае функция вызывается для выполнения работы, которая совершается внутри функции, а не ради ее возвращаемого значения. Вы не обязаны присваивать или иным образом использовать возвращаемое значение.

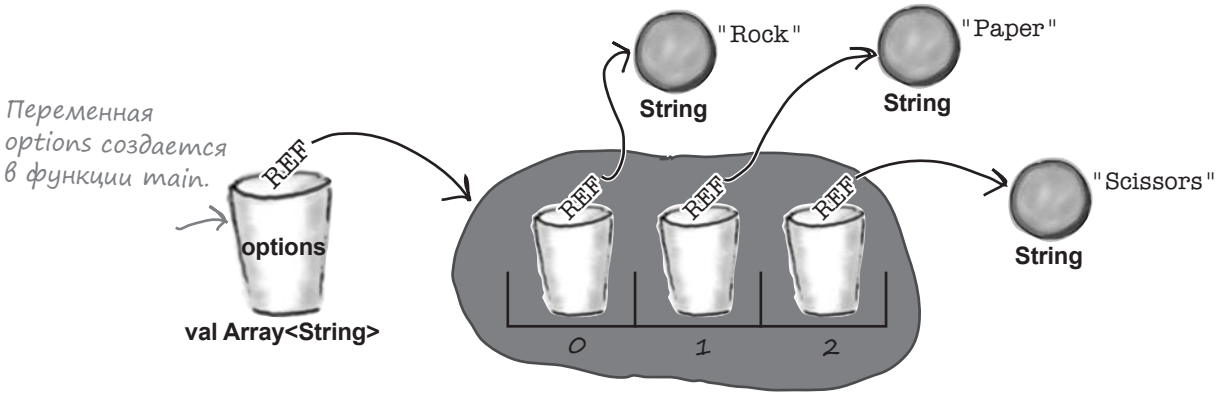
- [x] Выбор игры
- [ ] Выбор пользователя
- [ ] Результат

Что происходит под капотом

При выполнении кода происходит следующее:

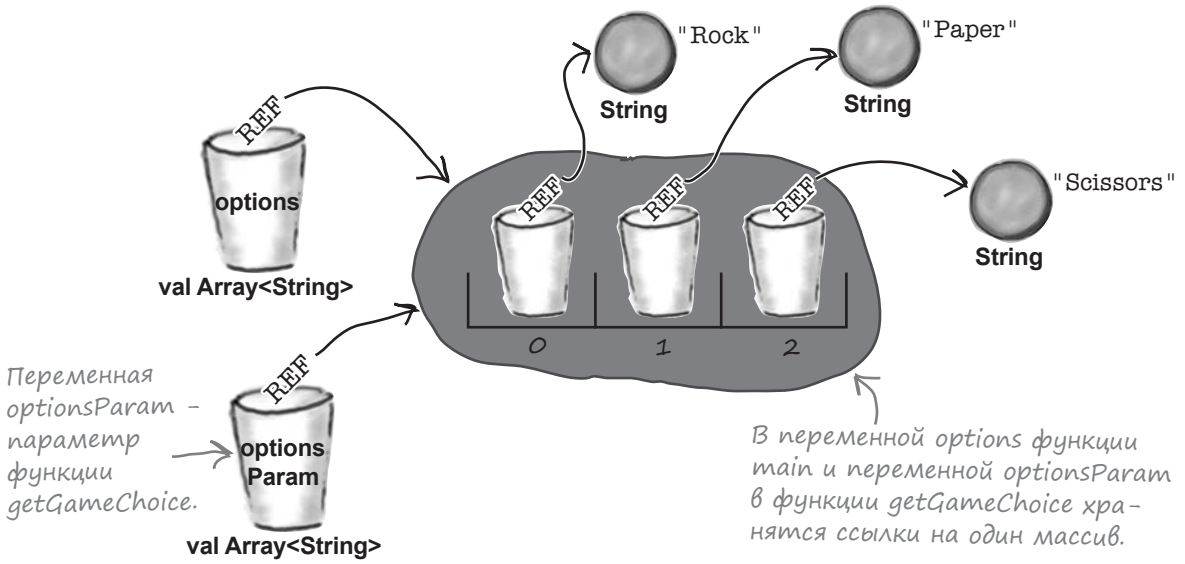
1 **val options = arrayOf("Rock", "Paper", "Scissors")**

Команда создает массив с элементами `String` и переменную с именем `options`, в которой хранится ссылка на него.



2 **val gameChoice = getGameChoice(options)**

Содержимое переменной `options` передается функции `getGameChoice`. В переменной `options` хранится ссылка на массив с элементами `String`; копия ссылки передается функции `getGameChoice` и попадает в ее параметр `optionsParam`. Это означает, что в переменных `options` и `optionsParam` **хранятся ссылки на один и тот же массив**.

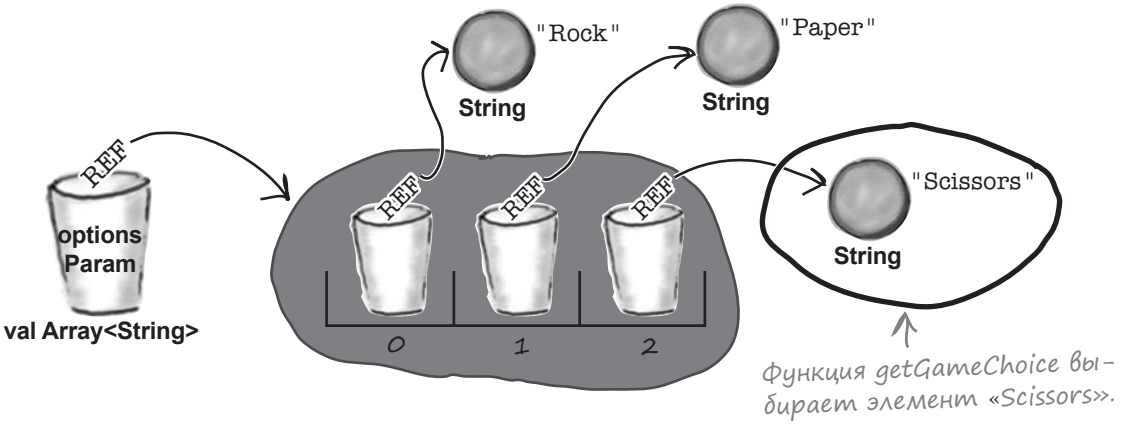


- [x] Выбор игры
- [ ] Выбор пользователя
- [ ] Результат

История продолжается

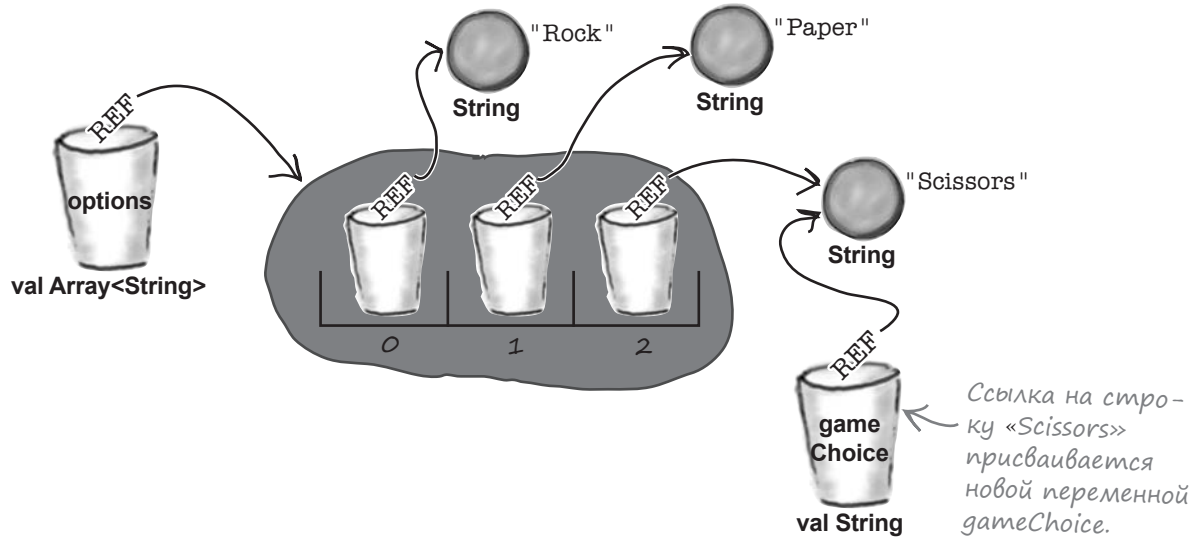
3 **fun getGameChoice(optionsParam: Array<String>) =**
optionsParam[(Math.random() * optionsParam.size).toInt()]

Функция `getGameChoice` случайным образом выбирает один из элементов `optionsParam` (например, элемент «Scissors»). Функция возвращает ссылку на этот элемент.



4 **val gameChoice = getGameChoice(options)**

Ссылка, возвращаемая функцией `getGameChoice`, помещается в новую переменную с именем `gameChoice`. Если, например, функция `getGameChoice` вернет ссылку на элемент «Scissors» из массива, это означает, что ссылка на объект «Scissors» помещается в переменной `gameChoice`.



- [x] **Выбор игры**
- [ ] Выбор пользователя
- [ ] Результат

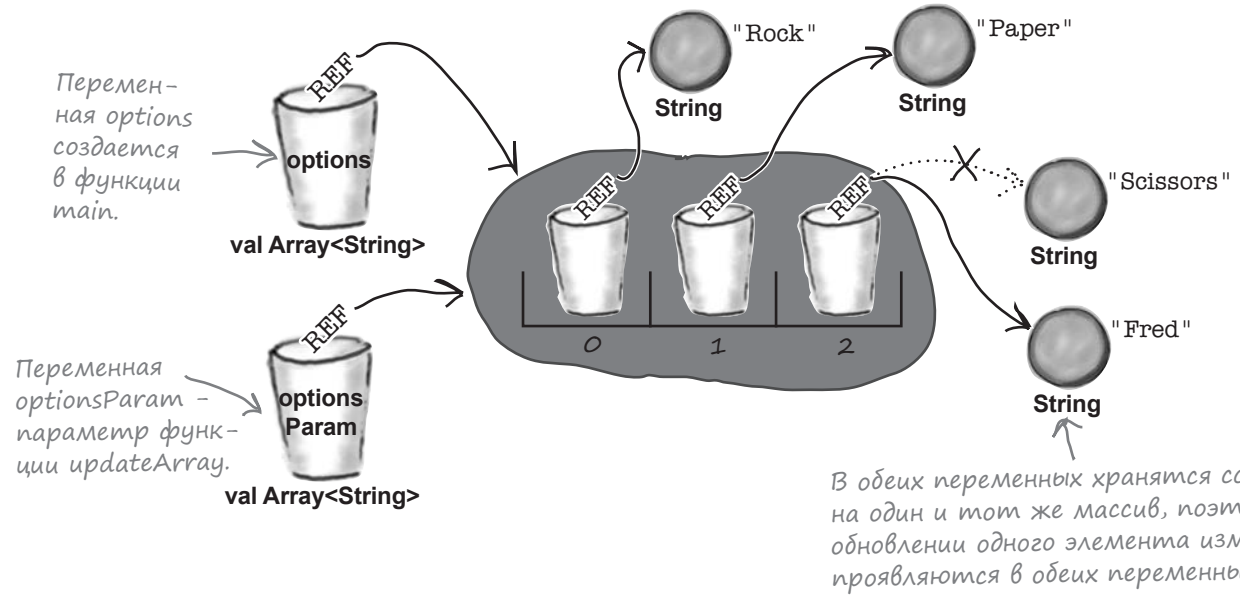
Таким образом, при передаче значения функции в действительности передается ссылка на объект. Означает ли это, что в объект можно вносить изменения?

Да, можно.

Допустим, имеется следующий код:

```
fun main(args: Array<String>) {
    val options = arrayOf("Rock", "Paper", "Scissors")
    updateArray(options)
    println(options[2])
}

fun updateArray(optionsParam: Array<String>) {
    optionsParam[2] = "Fred"
}
```

Функция `main` создает массив со строками «Rock», «Paper» и «Scissors». Ссылка на этот массив передается функции `updateArray`, которая заменяет третий элемент массива строкой «Fred». Наконец, функция `main` выводит значение третьего элемента массива, поэтому выводит текст «Fred».



Локальные переменные под увеличительным стеклом

Как уже упоминалось в этой главе, локальные переменные используются внутри тела функции. Они объявляются внутри функции и видны только внутри этой функции. При попытке использовать переменную, определенную внутри другой функции, компилятор выдает сообщение об ошибке:

```
fun main(args: Array<String>) {
    var x = 6
}

fun myFunction() {
    var y = x + 3
}
```

Код не компилируется, потому что myFunction не видит переменную x, объявленную в main.

Любые локальные переменные должны инициализироваться перед использованием. Например, если вы используете переменную для возвращения значения из функции, эту переменную необходимо инициализировать; в противном случае компилятор выдаст сообщение об ошибке:

```
fun myFunction(): String {
    var message: String
    return message
}
```

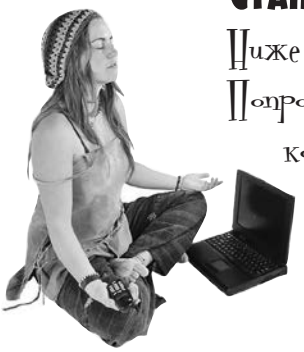
Необходимо инициализировать переменную, если вы хотите использовать ее для возвращения значения функции, иначе этот код компилироваться не будет.

Параметры функций практически не отличаются от локальных переменных; они тоже существуют только в контексте функции. При этом они всегда инициализируются, поэтому вы никогда не получите ошибку компилятора с указанием на то, что переменная параметра не была инициализирована. Дело в том, что компилятор выдаст сообщение об ошибке, если вы попытаетесь вызвать функцию без необходимых аргументов: компилятор гарантирует, что функции всегда будут вызываться с аргументами, соответствующими параметрам функции, и эти аргументы будут автоматически присвоены параметрам. Помните, что переменным параметров нельзя присвоить новые значения. За кулисами переменные параметров создаются как локальные переменные `val`, которые не могут повторно использоваться для других значений. Например, следующий код не будет компилироваться, потому что он пытается присвоить новое значение переменной параметра функции:

```
fun myFunction(message: String){
    message = "Hi!"
}
```

Переменные параметров рассматриваются как локальные переменные, созданные с ключевым словом val, поэтому они не могут повторно использоваться для других значений.

СТАНЬ компилятором



Ниже приведены три полные функции Kotlin. Попробуйте представить себя на месте компилятора и определить, будет ли компилироваться каждый из этих файлов. Если какие-то файлы не компилируются, то как бы вы их исправили?

A

```
fun doSomething(msg: String, i: Int): Unit {
    if (i > 0) {
        var x = 0
        while (x < i) {
            println(msg)
            x = x + 1
        }
    }
}
```

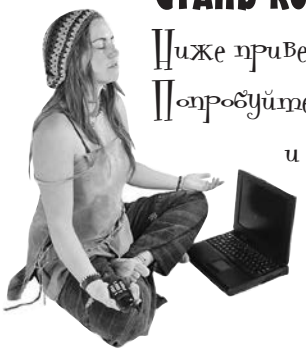
B

```
fun timesThree(x: Int): Int {
    x = x * 3
    return x
}
```

C

```
fun maxValue(args: Array<Int>) {
    var max = args[0]
    var x = 1
    while (x < args.size) {
        var item = args[x]
        max = if (max >= item) max else item
        x = x + 1
    }
    return max
}
```

СТАНЬ компилятором. Решение



Ниже приведены три полные функции Kotlin. Попробуйте представить себя на месте компилятора и определить, будет ли компилироваться каждый из этих файлов. Если какие-то файлы не компилируются, то как бы вы их исправили?

A

```
fun doSomething(msg: String, i: Int): Unit {
    if (i > 0) {
        var x = 0
        while (x < i) {
            println(msg)
            x = x + 1
        }
    }
}
```

Успешно компилируется и выполняется. Функция объявлена с возвращаемым типом Unit, а это означает, что у нее нет возвращаемого значения.

B

```
fun timesThree(x: Int): Int {
    val y = x * 3
    return y
}
```

Не компилируется из-за присваивания нового значения параметру функции. Также необходимо учитывать возвращаемый тип функции, так как в результате умножения Int на три может быть получено значение, слишком большое для типа Int.

C

```
fun maxValue(args: Array<Int>): Int {
    var max = args[0]
    var x = 1
    while (x < args.size) {
        var item = args[x]
        max = if (max >= item) max else item
        x = x + 1
    }
    return max
}
```

Не компилируется: в объявлении функции должен быть указан возвращаемый тип Int.

Функция getUserChoice

Мы написали код для выбора варианта игры, и теперь можно переходить к следующему шагу: получению варианта, выбранного пользователем. Для этого мы напишем новую функцию с именем `getUserChoice`, которая будет вызываться из функции `main`. Массив `options` будет передаваться в параметре функции `getUserChoice`, которая возвращает вариант, выбранный пользователем (в виде `String`):

```
fun getUserChoice(optionsParam: Array<String>): String {
    //Здесь размещается код
}
```

Ниже перечислены основные действия, которые должны выполняться функцией `getUserChoice`:

1. **Запросить вариант, выбранный пользователем.**
   Функция перебирает элементы массива `options` и предлагает пользователю ввести свой выбор в окне вывода.
2. **Прочитать вариант пользователя из окна вывода.**
   После того как пользователь введет свой вариант, его значение присваивается новой переменной.
3. **Проверить выбор пользователя.**
   Функция должна проверить, что пользователь выбрал вариант и выбранное значение присутствует в массиве. Если пользователь ввел действительный вариант, то функция вернет его. В противном случае функция будет запрашивать вариант снова и снова, пока не будет введено правильное значение.

Начнем с кода, запрашивающего вариант пользователя.

Получение варианта пользователя

Чтобы запросить вариант, выбранный пользователем, функция `getUserChoice` выводит следующее сообщение: «Please enter one of the following: Rock Paper Scissors».

В одном из возможных решений сообщение жестко программируется в функции `println`:

```
println("Please enter one of the following: Rock Paper Scissors.")
```

Тем не менее возможно более гибкое решение — перебрать все элементы массива `options` и вывести каждый элемент. Например, это может быть полезно, если вы решите изменить какие-либо из вариантов.

При желании можете добавить еще пару вариантов для усложнения игры.

Чтобы не перебирать элементы в цикле `while`, мы воспользуемся новой разновидностью циклов — так называемым циклом `for`. Посмотрим, как работает цикл `for`.

Как работают циклы for

- [x] Выбор игры
- [ ] Выбор пользователя
- [ ] Результат

Циклы **for** хорошо подходят для перебора чисел в фиксированном диапазоне или всех элементов в массиве (или другой разновидности коллекций — они рассматриваются в главе 9). А теперь посмотрим, как это делается.

Перебор чисел из диапазона

Допустим, вы хотите перебрать числа из заданного диапазона — от 1 до 10. Вы уже видели, как такие задачи решаются при помощи циклов `while`:

```
var x = 1
while (x < 11) {
    //Здесь размещается код
    x = x + 1
}
```

Однако решение с циклом `for` получается гораздо более элегантным и занимает меньше строк кода. Эквивалентный код выглядит так:

```
for (x in 1..10) {
    //Здесь размещается код
}
```

По сути это означает «для каждого числа от 1 до 10 *присвоить значение* переменной *x* и выполнить тело цикла».

Чтобы перебрать числа в диапазоне, сначала укажите имя переменной, которое должно использоваться в цикле. В данном примере переменной присвоено имя `x`, но вы можете использовать любое допустимое имя переменной. Переменная будет автоматически создана в начале цикла.

Диапазон значений задается оператором `..`. В представленном примере используется диапазон `1..10`, так что код будет перебирать числа от 1 до 10. В начале каждой итерации цикла текущее число присваивается переменной (в данном случае `x`).

Как и в случае с циклом `while`, если тело цикла состоит из одной команды, фигурные скобки не обязательны. Например, вот как выглядит цикл `for` для вывода чисел от 1 до 100:

```
for (x in 1..100) println(x)
```

Оператор `..` включает завершающее число в диапазоне. Если вы хотите исключить его из перебора, замените оператор `..` оператором `until`. Например, следующий код выводит числа от 1 до 99 и исключает значение 100:

```
for (x in 1 until 100) println(x)
```

Сокращения

Оператор инкремента `++` увеличивает переменную на 1. Так, выражение:

```
x++
```

является сокращением для команды:

```
x = x + 1
```

Оператор декремента `--` уменьшает переменную на 1. Выражение:

```
x--
```

является сокращением для:

```
x = x - 1
```

Если вы захотите увеличить переменную на другую величину вместо 1, используйте оператор `+=`. Таким образом, команда:

```
x += 2
```

делает то же самое, что и команда:

```
x = x + 2
```

Аналогичные сокращения `-=`, `*=` и `/=` поддерживаются для вычитания, умножения и деления.

Циклы while выполняются, пока заданное условие остается истинным. Циклы for работают в диапазоне значений или элементов.

Как работают циклы for (продолжение)

Использование downTo для перебора в обратном направлении

Если вы хотите перебрать диапазон чисел в обратном порядке, используйте **downTo** вместо `..` или `until`. Например, следующий код выводит числа от 15 до 1 в обратном порядке:

```
for (x in 15 downTo 1) println(x)
```

При использовании downTo вместо .. числа перебираются в обратном порядке.

Использование step для пропуска чисел в диапазоне

По умолчанию операторы `..`, `until` и `downTo` перебирают диапазон с единичным шагом. При желании можно увеличить размер шага при помощи ключевого слова `step`. Например, следующий код выводит нечетные числа в диапазоне от 1 до 100:

```
for (x in 1..100 step 2) println(x)
```

Перебор элементов в массиве

Цикл `for` также может использоваться для перебора элементов массива. В нашем примере перебираются элементы из массива с именем `options`. Для этого можно использовать цикл `for` в следующем формате:

```
for (item in optionsParam) {
    println("$item is an item in the array")
}
```

Перебирает все элементы массива с именем optionsParam.

Также можно выполнить перебор по индексам массива кодом следующего вида:

```
for (index in optionsParam.indices) {
    println("Index $index has item ${optionsParam[index]}")
}
```

Приведенный выше цикл можно даже немного упростить, возвращая индекс массива *и* значение как часть цикла:

```
for ((index, item) in optionsParam.withIndex()) {
    println("Index $index has item $item")
}
```

Перебирает все элементы массива. Индекс элемента присваивается переменной index, а сам элемент — переменной item.

Теперь, когда вы знаете, как работают циклы `for`, напишем код, который будет запрашивать у пользователя одно из трех значений: «Rock», «Paper» или «Scissors».

Запрос выбора пользователя

Мы будем выводить текст «Please enter one of the following: Rock Paper Scissors.» в цикле `for`. Ниже приведен код этой функции; мы обновим файл *Game.kt* позднее в этой главе, когда завершим работу над функцией `getUserChoice`:

```
fun getUserChoice(optionsParam: Array<String>): String {
    //Запросить у пользователя его выбор
    print("Please enter one of the following:")
    for (item in optionsParam) print(" $item")
    println(".")
}
```

Выводит значение каждого элемента в массиве.

Использование функции readLine для получения данных от пользователя

После того как мы запросим у пользователя выбранный вариант, необходимо прочитать его ответ. Для этого следует вызвать функцию **`readLine()`**:

```
val userInput = readLine()
```

Функция `readLine()` читает строку данных из стандартного входного потока (в нашем случае это окно ввода в IDE). Она возвращает значение `String` - текст, введенный пользователем.

Если входной поток вашего приложения был перенаправлен в файл, то функция `readLine()` возвращает `null` при достижении конца файла. `null` означает неопределенное значение или отсутствие данных.

Значения null подробно рассматриваются в главе 8, а пока это все, что вам нужно знать о них.

Обновленная версия функции `getUserChoice` (мы добавим ее в приложение, когда закончим работать над ней):

Функция getUserChoice будет обновлена через несколько страниц.

```
fun getUserChoice(optionsParam: Array<String>): String {
    //Запросить у пользователя его выбор
    print("Please enter one of the following:")
    for (item in optionsParam) print(" $item")
    println(".")
    //Прочитать пользовательский ввод
    val userInput = readLine()
}
```

Читает пользовательский ввод из стандартного входного потока. В нашем случае это окно вывода в IDE.

Затем необходимо проверить данные, введенные пользователем, и убедиться в том, что он ввел допустимое значение. Мы займемся этим после небольшого упражнения.

Путаница с сообщениями

Ниже приведена короткая программа на Kotlin. Один блок в программе пропущен. Ваша задача — сопоставить блоки-кандидаты (слева) с выводом, который вы увидите при подстановке этого блока. Не все варианты вывода будут использоваться, а некоторые варианты могут использоваться более одного раза. Проведите линию от каждого блока к подходящему варианту вывода.

```
fun main(args: Array<String>) {
    var x = 0
    var y = 20
    for(outer in 1..3) {
        for (inner in 4 downTo 2) {
            [ Блок с кодом подставляется сюда. ]
            y++
            x += 3
        }
        y -= 2
    }
    println("$x $y")
}
```

Блоки:

```
x += 6
```

```
x--
```

```
y = x + y
```

```
y = 7
```

```
x = x + y
y = x - 7
```

```
x = y
y++
```

Соедините каждый блок с одним из возможных вариантов вывода.

Варианты вывода:

```
4286 4275
```

```
27 23
```

```
27 6
```

```
81 23
```

```
27 131
```

```
18 23
```

```
35 32
```

```
3728 3826
```

Путаница с магнитами. Решение

Ниже приведена короткая программа Kotlin. Один блок в программе пропущен. Ваша задача — сопоставить блоки-кандидаты (слева) с выводом, который вы увидите при подстановке этого блока. Не все варианты вывода будут использоваться, а некоторые варианты могут использоваться более одного раза. Проведите линию от каждого блока к подходящему варианту вывода.

```
fun main(args: Array<String>) {
    var x = 0
    var y = 20
    for(outer in 1..3) {
        for (inner in 4 downTo 2) {
            [                    ]   <- Блок с кодом подставляется сюда.
            y++
            x += 3
        }
        y -= 2
    }
    println("$x $y")
}
```

Блоки:

- `x += 6`
- `x--`
- `y = x + y`
- `y = 7`
- `x = x + y`
`y = x - 7`
- `x = y`
`y++`

Варианты вывода:

- `3728 3826`
- `18 23`
- `27 6`
- `81 23`
- `27 131`
- `27 23`
- `35 32`
- `4286 4275`

Проверка пользовательского ввода

Последний код, который необходимо написать для функции `getUserChoice`, проверяет ввод пользователя и удостоверяется в том, что был введен допустимый вариант. Код должен решать следующие задачи:

1. **Проверить, что ввод пользователя отличен от null.** Как говорилось ранее, функция `readLine()` возвращает значение null, если при чтении строки из файла был достигнут конец файла. И хотя в нашем случае данные не читаются из файла, мы все равно должны проверить, что ввод пользователя отличен от `null`, чтобы не огорчать компилятор.
2. **Проверить, что вариант пользователя присутствует в массиве options.** Это можно сделать при помощи оператора `in`, который был представлен при обсуждении циклов `for`.
3. **Выполнять в цикле, пока пользователь не введет допустимый вариант.** Проверка должна работать в цикле, пока не будет выполнено условие (пользователь введет допустимый вариант). Для этой цели будет использоваться цикл `while`.

В принципе, вам уже знакома большая часть кода, который потребуется для реализации. Но чтобы написать более компактный код, мы используем логические выражения более мощные, чем представленные ранее. Сначала мы рассмотрим их, а потом приведем полный код функции `getUserChoice`.

Операторы 'И' и 'Или' (&& и ||)

Допустим, вы пишете код для выбора нового телефона, и выбор должен определяться множеством правил. Например, вас интересуют только телефоны из диапазона от 200 до 300 долларов. Для этого используется код следующего вида:

```
if (price >= 200 && price <= 300) {
    //Код выбора телефона
}
```

`&&` означает «И». Результат равен `true`, если истинны **обе** стороны `&&`. При выполнении этого кода Kotlin сначала вычисляет левую часть выражения. Если она ложна, то Kotlin даже не пытается вычислять правую часть. Так как одна сторона выражения ложна, это означает, что все выражение должно быть ложным.

Иногда это называется «ускоренным вычислением».

Если же вы хотите использовать выражение «ИЛИ», используйте оператор `||`:

```
if (price <= 10 || price >= 1000) {
    //Телефон слишком дешевый или слишком дорогой
}
```

Результат этого выражения равен `true`, если **хотя бы одна** сторона `||` истинна. На этот раз Kotlin не вычисляет правую часть выражения, если левая часть истинна.

Расширенные логические выражения (продолжение)

Не равно (!= и !)

Предположим, код должен выполняться для всех телефонов, кроме одной модели. Для этого используется код следующего вида:

```
if (model != 2000) {
    //Код выполняется, если значение model не равно 2000
}
```

Оператор != означает «не равно».

Также оператор ! может использоваться как обозначение «не». Например, следующий цикл выполняется в том случае, если переменная `isBroken` не является истинной:

```
while (!isBroken) {
    //Код выполняется, если телефон не сломан
}
```

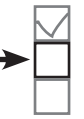
Круглые скобки делают код более понятным

Логические выражения могут быть очень большими и сложными:

```
if ((price <= 500 && memory >= 16) ||
    (price <= 750 && memory >= 32) ||
    (price <= 1000 && memory >= 64)) {
    //Сделать то, что нужно
}
```

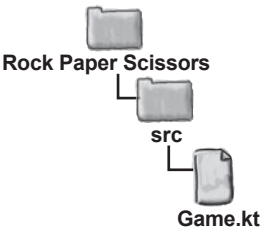
Если же вас интересуют подробности, стоит поинтересоваться приоритетом этих операторов. Мы не будем объяснять все малопонятные тонкости работы приоритетов, а рекомендуем использовать круглые скобки, чтобы сделать код более понятным.

Итак, мы рассмотрели расширенные возможности логических выражений. Теперь можно рассмотреть остальной код функции `getUserChoice` и добавить его в приложение.



Добавление функции getUserChoice в файл Game.kt

Ниже приведен обновленный код приложения вместе с полной функцией `getUserChoice`. Обновите свою версию *Game.kt* так, чтобы она соответствовала нашей (изменения выделены жирным шрифтом):




```
fun main(args: Array<String>) {
    val options = arrayOf("Rock", "Paper", "Scissors")
    val gameChoice = getGameChoice(options)
    val userChoice = getUserChoice(options)
}
```

Вызов функции getUserChoice.

```
fun getGameChoice(optionsParam: Array<String>) =
        optionsParam[(Math.random() * optionsParam.size).toInt()]

fun getUserChoice(optionsParam: Array<String>): String {
    var isValidChoice = false
    var userChoice = ""
    //Выполнять цикл, пока пользователь не введет допустимый вариант
    while (!isValidChoice) {

        //Запросить у пользователя его выбор
        print("Please enter one of the following:")
        for (item in optionsParam) print(" $item")
        println(".")
        //Прочитать пользовательский ввод
        val userInput = readLine()
        //Проверить пользовательский ввод
        if (userInput != null && userInput in optionsParam) {
            isValidChoice = true
            userChoice = userInput
        }
        //Если выбран недопустимый вариант, сообщить пользователю
        if (!isValidChoice) println("You must enter a valid choice.")
    }
    return userChoice
}
```

Переменная isValidChoice будет показывать, ввел ли пользователь допустимый вариант.

Цикл продолжается до тех пор, пока переменная isValidChoice не будет равна true.

Проверяем, что ввод пользователя отличен от null и что он присутствует в массиве options.

Если ввод пользователя прошел проверку, цикл прерывается.

Если ввод пользователя недопустим, цикл продолжается.

Опробуем этот код в деле и посмотрим, что происходит при его выполнении.

Тест-драйв

- [x] Выбор игры
- [x] **Выбор пользователя**
- [ ] Результат

Выполните свой код: откройте меню Run и выберите команду Run 'GameKt'. Когда откроется окно вывода IDE, вам будет предложено выбрать свой вариант из набора «Rock», «Paper» или «Scissors»:

```
Please enter one of the following: Rock Paper Scissors.
```

Когда вводится недействительный вариант, программа просит ввести действительный. Это повторяется до ввода корректного значения: «Камень»», «Ножницы», «Бумага». После этого программа завершится.

Мы ввели несколько недопустимых вариантов...

```
Fred
You must enter a valid choice.
Please enter one of the following: Rock Paper Scissors.
George
You must enter a valid choice.
Please enter one of the following: Rock Paper Scissors.
Ginny
You must enter a valid choice.
Please enter one of the following: Rock Paper Scissors.
```

...а затем ввели «Rock».

```
Rock
```

Остается вывести результаты

- [x] Выбор игры
- [x] Выбор пользователя
- [ ] **Результат**

Последнее, что осталось сделать в приложении — вывести результаты. Напомним: если игра и пользователь выбрали одинаковые варианты, игра завершается вничью. Но если варианты различны, игра определяет победителя по следующим правилам:

Варианты	**Результат**
Ножницы, бумага (Scissors, Paper)	«Ножницы» побеждают («ножницы режут бумагу»).
Камень, ножницы (Rock, Scissors)	«Камень» побеждает («камень разбивает ножницы»).
Бумага, камень (Paper, Rock)	«Бумага» побеждает («бумага накрывает камень»).

Результаты будут выводиться новой функцией с именем `printResult`. Эта функция будет вызываться из `main`, и ей будут передаваться два параметра: вариант, выбранный пользователем, и вариант, выбранный игрой.

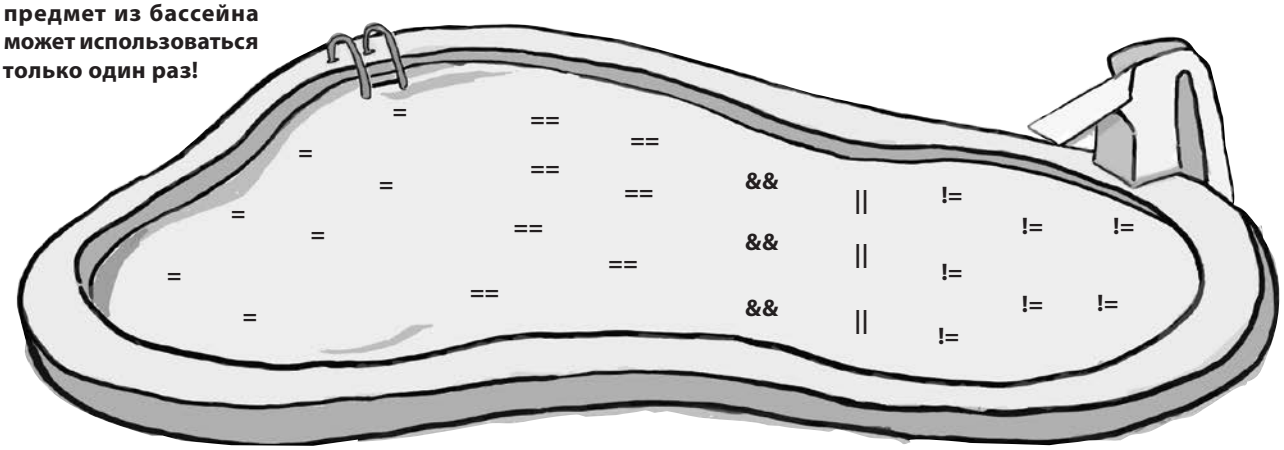
Прежде чем рассматривать код этой функции, давайте проверим, удастся ли вам справиться со следующим упражнением.

У бассейна

Выловите из бассейна фрагменты кода и разместите их в пустых строках функции `printResult`. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: вывести варианты, выбранные пользователем и игрой, и сообщить, кто же выиграл.

```
fun printResult(userChoice: String, gameChoice: String) {
    val result: String
    //Определить результат
    if (userChoice_____gameChoice) result = "Tie!"
    else if ((userChoice_____"Rock"_____gameChoice_____"Scissors")_____
            (userChoice_____"Paper"_____gameChoice_____"Rock")_____
            (userChoice_____"Scissors"_____gameChoice_____"Paper")) result = "You win!"
    else result = "You lose!"
    //Вывести результат
    println("You chose $userChoice. I chose $gameChoice. $result")
}
```

Примечание: каждый предмет из бассейна может использоваться только один раз!



У бассейна. Решение

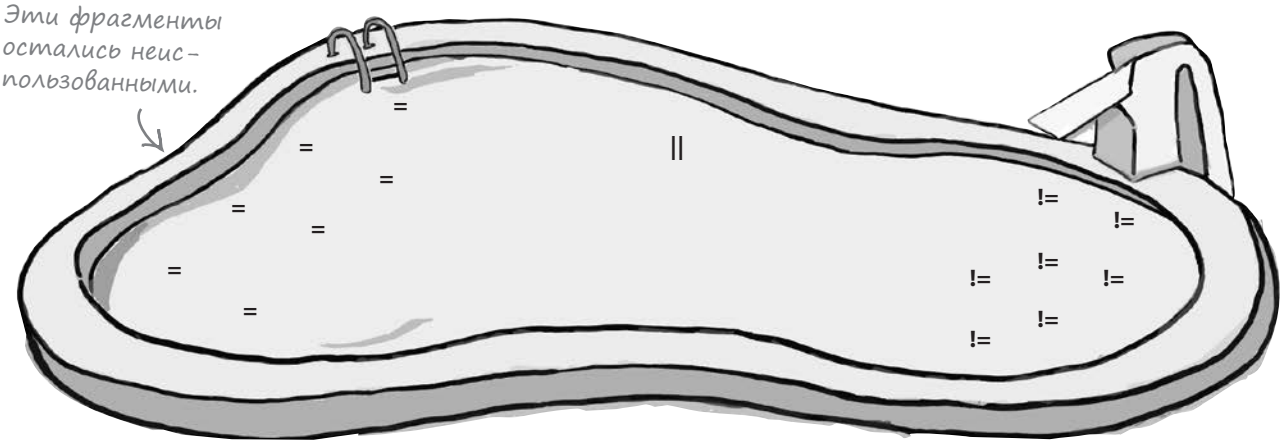
Выловите из бассейна фрагменты кода и разместите их в пустых строках функции `printResult`. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: вывести варианты, выбранные пользователем и игрой, и сообщить, кто же выиграл.

```
fun printResult(userChoice: String, gameChoice: String) {
    val result: String
    //Определить результат
    if (userChoice == gameChoice) result = "Tie!"
    else if ((userChoice == "Rock" && gameChoice == "Scissors") ||
            (userChoice == "Paper" && gameChoice == "Rock") ||
            (userChoice == "Scissors" && gameChoice == "Paper")) result = "You win!"
    else result = "You lose!"
    //Вывести результат
    println("You chose $userChoice. I chose $gameChoice. $result")
}
```

Если пользователь и игра выбрали один и тот же вариант, игра завершается вничью.

Если хотя бы одно из этих условий истинно, побеждает пользователь.

Эти фрагменты остались неиспользованными.



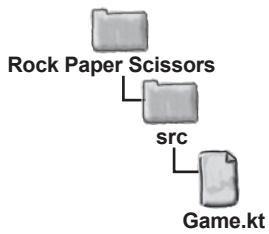
- [x] Выбор игры
- [x] Выбор пользователя
- [ ] Результат

Добавление функции printResult в файл Game.kt

Функция `printResult` добавляется в файл *Game.kt* и вызывается из функции `main`. Ниже приведен код. Приведите свою версию в соответствие с нашей (изменения выделены жирным шрифтом):

```
fun main(args: Array<String>) {
    val options = arrayOf("Rock", "Paper", "Scissors")
    val gameChoice = getGameChoice(options)
    val userChoice = getUserChoice(options)
    printResult(userChoice, gameChoice)
}

fun getGameChoice(optionsParam: Array<String>) =
        optionsParam[(Math.random() * optionsParam.size).toInt()]

fun getUserChoice(optionsParam: Array<String>): String {
    var isValidChoice = false
    var userChoice = ""
    //Выполнять цикл, пока пользователь не введет допустимый вариант
    while (!isValidChoice) {
        //Запросить у пользователя его выбор
        print("Please enter one of the following:")
        for (item in optionsParam) print(" $item")
        println(".")
        //Прочитать пользовательский ввод
        val userInput = readLine()
        //Проверить пользовательский ввод
        if (userInput != null && userInput in optionsParam) {
            isValidChoice = true
            userChoice = userInput
        }
        //Если выбран недопустимый вариант, сообщить пользователю
        if (!isValidChoice) println("You must enter a valid choice.")
    }
    return userChoice
}
```

Функция printResult вызывается из main.



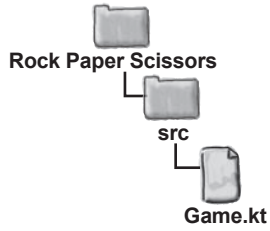
Продолжение на следующей странице

Код Game.kt (продолжение)

- [x] Выбор игры
- [x] Выбор пользователя
- [ ] Результат

```
fun printResult(userChoice: String, gameChoice: String) {
    val result: String
    //Определить результат
    if (userChoice == gameChoice) result = "Tie!"
    else if ((userChoice == "Rock" && gameChoice == "Scissors") ||
            (userChoice == "Paper" && gameChoice == "Rock") ||
            (userChoice == "Scissors" && gameChoice == "Paper")) result = "You win!"
    else result = "You lose!"
    //Вывести результат
    println("You chose $userChoice. I chose $gameChoice. $result")
}
```

Эту функцию следует добавить в файл.



Вот и весь код, необходимый для нашего приложения. Посмотрим, что произойдет, когда мы его запустим.

Тест-драйв

При запуске кода открывается окно вывода IDE. Введите один из вариантов: «Rock», «Paper» или «Scissors» (мы выбрали «Paper»):

```
Please enter one of the following: Rock Paper
Scissors.
Paper
You chose Paper. I chose Rock. You win!
```

Приложение выводит наш вариант, вариант, выбранный игрой, и результат.

Часто Задаваемые Вопросы

В: Я ввел вариант «paper», но приложение сообщает, что мой вариант недопустим. Почему?

О: Потому что вы ввели строку в нижнем регистре, а она должна начинаться с символа верхнего регистра. Игра требует, чтобы вы точно ввели одну из строк, «Rock», «Paper» или «Scissors», и не распознает «paper» как один из возможных вариантов.

В: Можно ли заставить Kotlin игнорировать регистр символов? Можно ли изменить регистр символов введенного значения, прежде чем проверять его на присутствие в массиве?

О: В Kotlin существуют функции `toLowerCase`, `toUpperCase` и `capitalize` для создания строки, преобразованной к нижнему регистру, верхнему регистру или к написанию с начальной прописной буквы. Например, следующая команда использует функцию `capitalize` для преобразования первой буквы строки с именем `userInput`:

```
userInput = userInput.capitalize()
```

Таким образом, пользовательский ввод можно преобразовать к нужному формату, перед тем как проверять, совпадает ли он с каким-либо из значений в массиве.

Ваш инструментарий Kotlin

Глава 3 осталась позади, а ваш инструментарий пополнился функциями.

Весь код для этой главы можно загрузить по адресу https://tinyurl.com/HFKotlin.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Функции улучшают структуру кода и упрощают его повторное использование.
- Функции могут получать параметры — одно или несколько значений.
- Количество и типы значений, передаваемых функции, должны соответствовать порядку и типу параметров, объявленных функцией.
- Функция может возвращать значение. Тип этого значения (если оно есть) должен быть объявлен заранее.
- Возвращаемый тип `Unit` означает, что функция не возвращает никакое значение.
- Используйте циклы `for` вместо циклов `while`, если вам заранее известно, сколько раз должен выполняться код цикла.
- Функция `readLine()` читает строку ввода из стандартного входного потока. Она возвращает значение `String` — текст, введенный пользователем.
- Если входной поток был перенаправлен в файл, то при достижении конца файла функция `readLine()` вернет `null` — признак того, что значение отсутствует или не определено.
- `&&` означает «И». `||` означает «ИЛИ». `!` означает «НЕ».

Высокий класс

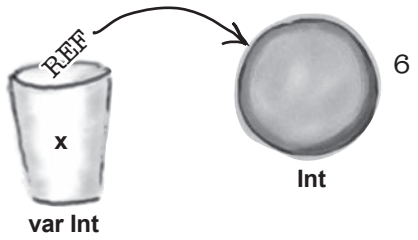
Пришло время выйти за границы базовых типов Kotlin. Рано или поздно базовых типов Kotlin вам станет *недостаточно*. И здесь на помощь приходят ***классы***. Классы представляют собой *шаблоны* для **создания ваших собственных типов объектов** и определения их свойств и функций. В этой главе вы научитесь **проектировать и определять классы,** а также использовать их для **создания новых типов объектов**. Вы познакомитесь с ***конструкторами***, ***блоками инициализации***, ***get-*** и ***set-***методами и научитесь использовать их для защиты свойств. В завершающей части вы узнаете о **средствах защиты данных, встроенных в *весь* код Kotlin**. Это сэкономит ваше время, силы и множество нажатий клавиш.

Классы используются для определения типов объектов

К настоящему моменту вы научились создавать и использовать переменные базовых типов Kotlin: числа, строки и массивы. Например, следующая команда:

```
var x = 6
```

создает объект `Int` со значением 6, и ссылка на этот объект присваивается новой переменной с именем `x`:

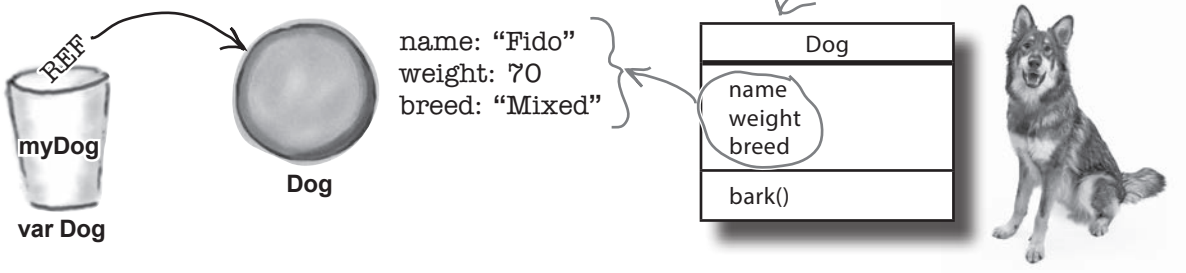


Эти типы определяются при помощи **классов**. Класс представляет собой шаблон, определяющий свойства и функции, связанные с объектами этого типа. Например, при создании объекта `Int` компилятор проверяет класс `Int` и видит, что объект должен содержать целочисленное значение, а также такие функции, как `toLong` и `toString`.

Вы можете определять собственные классы

Если вы хотите, чтобы ваши приложения работали с типами объектов, отсутствующими в Kotlin, вы можете определять собственные типы; для этого следует написать новые классы. Например, если ваше приложение предназначено для хранения информации о собаках, вы можете определить класс `Dog` для создания ваших собственных объектов `Dog`. В таких объектах должно храниться имя (name), вес (weight) и порода (breed) каждой собаки:

Это класс Dog. Он сообщает компилятору, что у Dog есть имя (name), вес (weight) и порода (breed), а также функция bark.



Как же определяются классы?

Как спроектировать собственный класс

Когда вы хотите определить собственный класс, подумайте над тем, какие объекты будут создаваться на основе этого класса. При этом следует учитывать то,что:

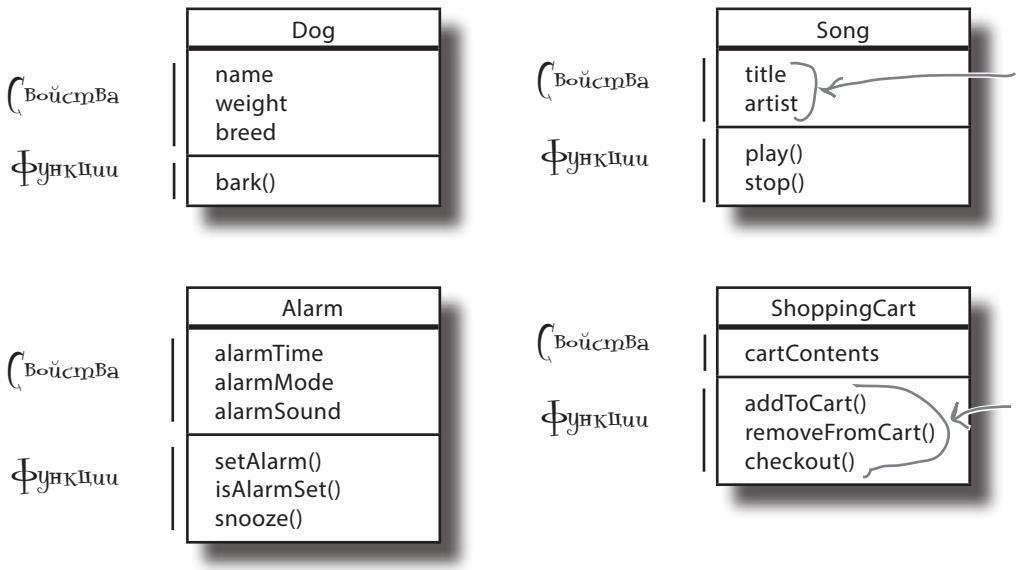
- **каждый объект должен знать о себе,**
- **может сделать каждый объект.**

То, что объект знает о себе, — свойства. То, что объект может сделать, — его функции.

То, что объект знает о себе — это его **свойства**. Они представляют состояние объекта (то есть его данные), при этом каждый объект этого типа может хранить собственный набор уникальных значений. Класс `Dog`, например, может содержать свойства `name`, `weight` и `breed`. Класс `Song` может содержать свойства `title` и `artist`.

То, что объект может сделать, — это его **функции**. Они определяют поведение объекта и могут использовать свойства объекта. Например, класс `Dog` может содержать функцию `bark`, а класс `Song` — функцию `play`.

Несколько примеров классов со свойствами и функциями:



Свойства — то, что объект знает о самом себе. В данном примере объект Song знает свое название (title) и исполнителя (artist).

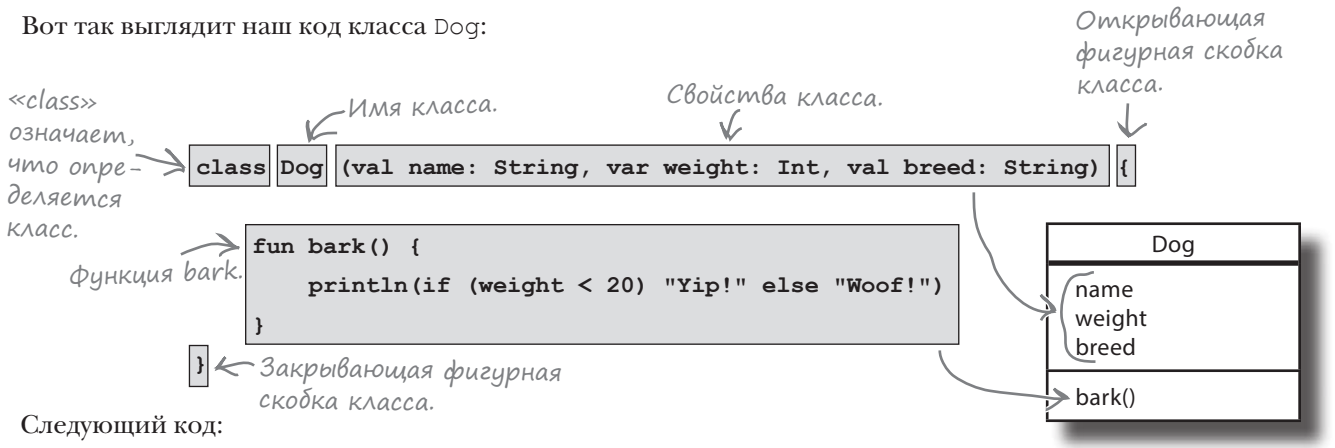
Функции — те операции, которые может выполнять объект. Например, объект ShoppingCart (покупательская корзина) умеет добавлять товары, удалять товары и оформлять заказ.

Теперь вы знаете, какими свойствами и функциями должен обладать ваш класс, и сможете написать код для его создания. Сейчас мы покажем, как это делается.

Определение класса Dog

Сейчас мы создадим класс `Dog`, который будет использоваться для создания объектов `Dog`. Каждый объект `Dog` содержит имя, вес и породу; мы будем хранить их в виде свойств этих объектов. Также будет определена функция `bark` (лаять), работа которой будет зависеть от веса собаки.

Вот так выглядит наш код класса `Dog`:



Следующий код:

```
class Dog(val name: String, var weight: Int, val breed: String) {
    ...
}
```

определяет имя класса (`Dog`) и свойства, которыми обладает класс `Dog`. Через несколько страниц мы более подробно покажем, что происходит под капотом, а пока все, что вам нужно знать, — этот код определяет свойства `name`, `weight` и `breed`, и при создании объекта `Dog` этим свойствам будут присвоены значения.

Функция, определенная внутри класса, называется функцией класса. Также иногда используется термин «метод».

Функции класса определяются в теле класса (в фигурных скобках `{}`). В нашем примере определяется функция `bark`, поэтому код выглядит так:

```
class Dog(val name: String, var weight: Int, val breed: String) {
    fun bark() {
        println(if (weight < 20) "Yip!" else "Woof!")
    }
}
```

Почти не отличается от функций, которые были представлены в предыдущей главе. Единственное различие в том, что эта функция определяется в теле класса Dog.

Мы рассмотрели код класса `Dog`, а теперь посмотрим, как использовать его для создания объектов `Dog`.

Как создать объект Dog

Класс можно представить как шаблон для создания объектов, так как он сообщает компилятору, как создавать объекты этого конкретного типа. В частности, класс сообщает компилятору, какими свойствами должен обладать каждый объект. При этом каждый объект, созданный на основе этого класса, может содержать собственные значения. Например, каждый объект `Dog` должен обладать свойствами `name`, `weight` и `breed`, причем каждый объект `Dog` содержит собственный набор значений.

Один класс

Много объектов

Мы используем класс `Dog` для создания объекта `Dog` и присваивания его новой переменной `myDog`. Код выглядит так:

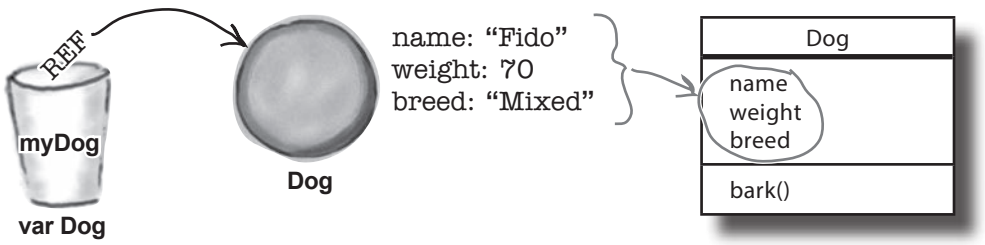
```
var myDog = Dog("Fido", 70, "Mixed")
```

При создании объекта Dog передаются аргументы для трех свойств.

Код передает объекту `Dog` три аргумента. Они соответствуют свойствам, которые мы определили в классе `Dog`: `name`, `weight` и `breed`:

```
class Dog(val name: String, var weight: Int, val breed: String) {
    ...
}
```

При выполнении этого кода создается новый объект `Dog`, а аргументы используются для присваивания значений свойствам `Dog`. В данном случае у создаваемого объекта `Dog` свойство `name` равно «Fido», свойство `weight` — 70, а свойство `breed` содержит строку «Mixed»:



Теперь вы знаете, как создаются новые объекты `Dog`. Посмотрим, как обращаться к его свойствам и функциям.

Как обращаться к свойствам и функциям

После того как объект будет создан, к его свойствам можно обращаться при помощи оператора «точка» (`.`). Например, чтобы вывести свойство `name` объекта `Dog`, используйте следующий код:

```
var myDog = Dog("Fido", 70, "Mixed")
println(myDog.name)
```

myDog.name означает «взять объект myDog и получить его свойство name».

Также можно изменять любые свойства объекта, определенные с ключевым словом `var`. Например, вот как задать свойству `weight` объекта `Dog` значение 75:

```
myDog.weight = 75
```

Взять объект myDog и задать его свойству weight значение 75.

Компилятор не позволит изменять свойства, определенные с ключевым словом `val`. Если вы попробуете это сделать, компилятор сообщит об ошибке.

Оператор «точка» также может использоваться для вызова функций объекта. Например, чтобы вызвать функцию `bark` объекта `Dog`, используйте следующий код:

```
myDog.bark()
```

Взять объект myDog и вызвать его функцию bark.

Массивы объектов

Создаваемые объекты также могут сохраняться в массиве. Скажем, если вы хотите создать массив с элементами `Dog`, это делается так:

```
var dogs = arrayOf(Dog("Fido", 70, "Mixed"), Dog("Ripper", 10, "Poodle"))
```

Код создает два объекта Dog и добавляет их в массив array<Dog> с именем dogs.

Этот фрагмент определяет переменную с именем `dogs`, а поскольку это массив для хранения объектов `Dog`, компилятор назначает ему тип `array<Dog>`. Затем два объекта `Dog` добавляются в массив.

Вы можете обращаться к свойствам и функциям любого объекта `Dog` в массиве. Предположим, что вы хотите обновить свойство `weight` второго объекта `Dog` и вызвать его функцию `bark`. Для этого следует получить ссылку на второй элемент массива `dogs` (`dogs[1]`), а затем использовать оператор «точка» для обращения к свойству `weight` и функции `bark` объекта `Dog`:

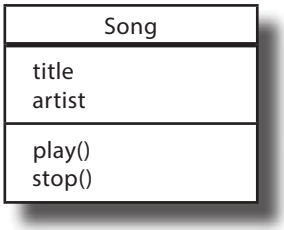
```
dogs[1].weight = 15
dogs[1].bark()
```

Компилятор знает, что dogs[1] является объектом Dog, что позволяет обращаться к свойствам объекта Dog и вызывать его функции.

По сути это означает «взять второй объект из массива dogs, изменить свойство weight на 15 и вызвать функцию bark».

Создание приложения Songs

Прежде чем продолжать изучение классов, немного потренируемся и создадим новый проект Songs. В проекте мы определим класс `Song`, а затем создадим и используем несколько объектов `Song`.



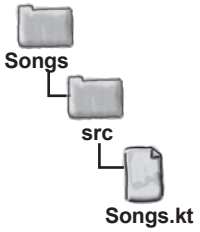
Создайте новый проект Kotlin для JVM и присвойте ему имя «Songs». Затем создайте новый файл Kotlin с именем *Songs.kt* — выделите папку *src*, откройте меню File и выберите команду New → Kotlin File/Class. Введите имя файла «Songs» и выберите вариант File в группе Kind.

Затем добавьте следующий код в *Songs.kt*:

```
class Song(val title: String, val artist: String) {
    fun play() {
        println("Playing the song $title by $artist")
    }

    fun stop() {
        println("Stopped playing $title")
    }
}

fun main(args: Array<String>) {
    val songOne = Song("The Mesopotamians", "They Might Be Giants")
    val songTwo = Song("Going Underground", "The Jam")
    val songThree = Song("Make Me Smile", "Steve Harley")
    songTwo.play()
    songTwo.stop()
    songThree.play()
}
```

Определяем свойства title и artist.

Добавляем функции play и stop.



Создать три объекта Song.

Воспроизвести (play) для объекта songTwo, затем остановить его (stop) и воспроизвести songThree.

Тест-драйв

При выполнении кода в окне вывода IDE будет следующий текст:

```
Playing the song Going Underground by The Jam
Stopped playing Going Underground
Playing the song Make Me Smile by Steve Harley
```

Итак, теперь вы знаете, как определить класс и использовать его для создания объектов. Теперь можно заняться более основательным изучением загадочного мира создания объектов.

Чудо создания объекта

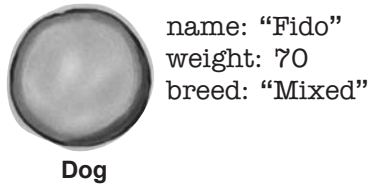
Объявление объекта с присваиванием состоит из трех основных этапов:

1. **Объявление переменной.**

```
var myDog = Dog("Fido", 70, "Mixed")
```

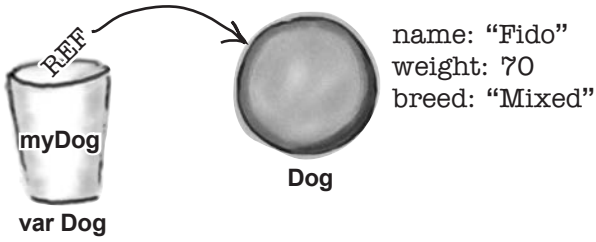
2. **Создание объекта.**

```
var myDog = Dog("Fido", 70, "Mixed")
```



3. **Связывание объекта с переменной посредством присваивания ссылки.**

```
var myDog = Dog("Fido", 70, "Mixed")
```



Главные чудеса происходят в фазе 2 — при создании объекта. Там происходят довольно важные события, поэтому к этой фазе стоит присмотреться повнимательнее.

Как создаются объекты

При определении объекта используется код следующего вида:

```
var myDog = Dog("Fido", 70, "Mixed")
```

Из-за круглых скобок все выглядит так, словно мы вызываем функцию с именем Dog.

Кажется, будто мы вызываем функцию с именем `Dog`. Но хотя внешне и по поведению эта конструкция похожа на функцию, это не так. Вместо этого здесь вызывается **конструктор** `Dog`.

Конструктор содержит код, необходимый для инициализации объекта. Он гарантированно выполняется до того, как ссылка на объект будет сохранена в переменной; это означает, что у вас будет возможность вмешаться в происходящее и подготовить объект к использованию. Многие разработчики используют конструкторы для определения свойств объекта и присваивания им значений.

Каждый раз, когда вы создаете новый объект, вызывается конструктор класса этого объекта. Таким образом, при выполнении следующего кода:

```
var myDog = Dog("Fido", 70, "Mixed")
```

вызывается конструктор класса `Dog`.

Конструктор выполняется при создании экземпляра объекта. Он используется для определения свойств и их инициализации.

Как выглядит конструктор Dog

При создании класса `Dog` мы включаем конструктор — круглые скобки и заключенный в них код — в заголовок класса:

```
class Dog(val name: String, var weight: Int, val breed: String) {
    ...
}
```

Этот код (вместе с круглыми скобками) является конструктором класса. Формально он называется первичным конструктором.

Конструктор `Dog` определяет три свойства — `name`, `weight` и `breed`. Каждый объект `Dog` содержит эти свойства, и при создании объекта `Dog` конструктор присваивает значение каждому свойству. Конструктор инициализирует состояние каждого объекта `Dog` и готовит его к использованию.

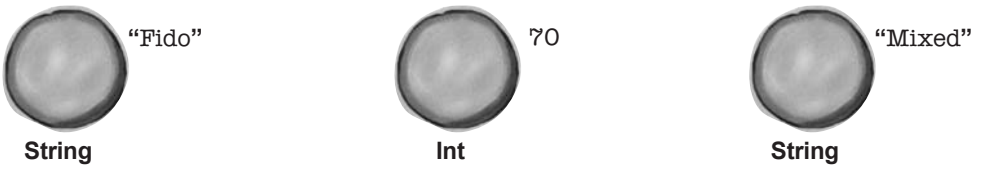
Посмотрим, что происходит при вызове конструктора `Dog`.

Под капотом: вызов конструктора Dog

Давайте разберемся, что происходит при выполнении кода:

```
var myDog = Dog("Fido", 70, "Mixed")
```

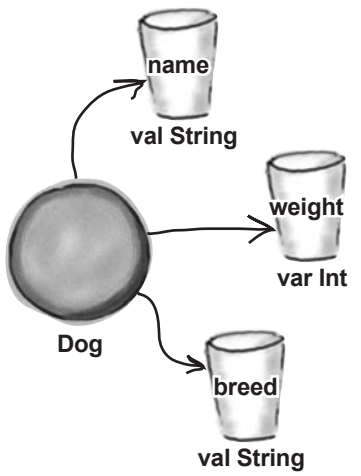
1. **Система создает объект для каждого аргумента, переданного конструктору Dog.**
Она создает объект `String` со значением «Fido», объект `Int` со значением 70 и объект `String` со значением «Mixed».



2. **Система выделяет память для нового объекта Dog и вызывает конструктор Dog.**

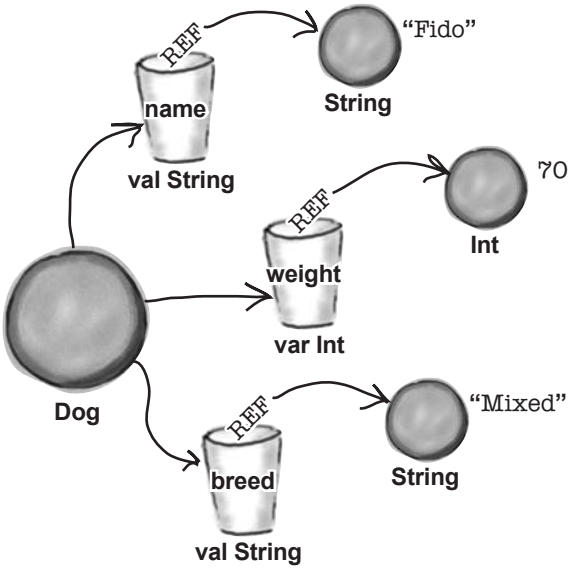
3. **Конструктор Dog определяет три свойства: name, weight и breed.**
Во внутреннем представлении **каждое свойство является переменной**. Для каждого свойства создается переменная соответствующего типа, определенного в конструкторе.

```
class Dog(val name: String,
                  var weight:
Int,
                  val breed:
String) {


}
```

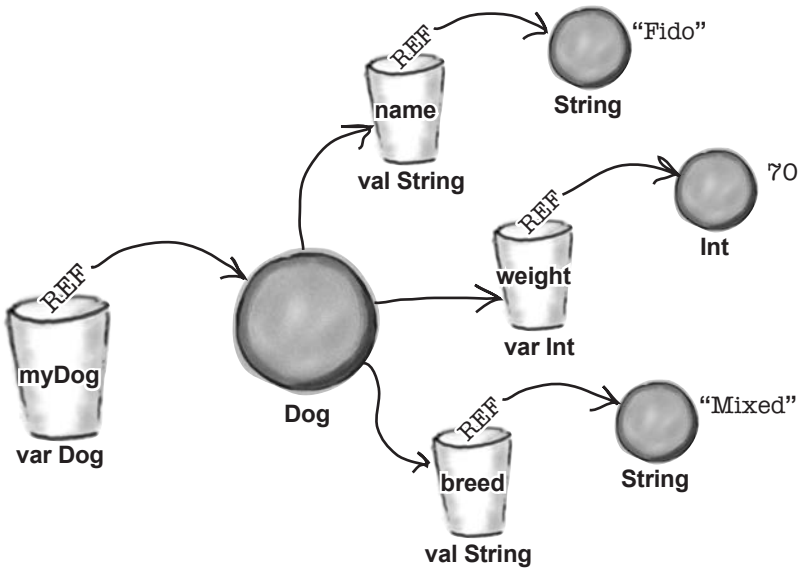


История продолжается...

4 **Каждой из переменных свойств объекта Dog присваивается ссылка на соответствующий объект значения.**
Например, свойству `name` присваивается ссылка на объект `String` со значением «Fido» и так далее.



5 **Наконец, ссылка на объект Dog присваивается новой переменной Dog с именем myDog.**



Понятно. Конструктор Dog определяет свойства, а каждое свойство на самом деле — это всего лишь переменная, локальная для объекта. И этой переменной присваивается значение.

Все верно! Свойство — это переменная, локальная для объекта.

Это означает, что все, что вы знаете о переменных, относится и к свойствам. Если свойство определяется с ключевым словом `val`, это означает, что ей нельзя присвоить новое значение. С другой стороны, вы можете изменять свойства, объявленные с ключевым словом `var`.

В нашем примере ключевое слово `val` используется для свойств `name` и `breed`, а `var` — для определения свойства `weight`:

```
class Dog(val name: String, var weight: Int, val breed: String) {
    ...
}
```

Это означает, что у объектов `Dog` изменяться может только свойство `weight`, но не свойства `name` и `breed`.

Часто Задаваемые Вопросы

В: Выделяет ли конструктор память для создаваемого объекта?

О: Нет, это делает система. Конструктор инициализирует объект и следит за тем, чтобы свойства объекта были созданы и им были присвоены исходные значения. Всем выделением памяти занимается система.

В: Могу ли я определить класс без определения конструктора?

О: Да, можете. В этой главе мы покажем, как это делается.

Объект иногда называется экземпляром класса, а его свойства — переменными экземпляров.

Развлечения с магнитами

Кто-то выложил на холодильнике код нового класса **`DrumKit`** (ударная установка) и функции `main`, которая выводит следующий результат:

```
ding ding ba-da-bing!
bang bang bang!
ding ding ba-da-bing!
```

К сожалению, все магниты перемешались. Удастся ли вам снова собрать код в исходном виде?

```
class DrumKit(var hasTopHat: Boolean, var hasSnare: Boolean) {

}
```

Разместите магниты в этих прямоугольниках.

```
fun main(args: Array<String>) {

}
```

```
println("ding ding ba-da-bing!")
{
d.hasSnare =
fun playSnare()
val d = DrumKit(true, true)
(hasSnare)
fun playTopHat()
}
(hasTopHat)
}
{
if
d.playTopHat()
d.playSnare()
d.playTopHat()
d.playSnare()
if
false
println("bang bang bang!")
```

Развлечения с магнитами. Решение

Кто-то выложил на холодильнике код нового класса **`DrumKit`** (ударная установка) и функции `main`, которая выводит следующий результат:

```
ding ding ba-da-bing!
bang bang bang!
ding ding ba-da-bing!
```

К сожалению, все магниты перемешались. Удастся ли вам снова собрать код в исходном виде?

```
class DrumKit(var hasTopHat: Boolean, var hasSnare: Boolean) {

    fun playTopHat() {
        if (hasTopHat) println("ding ding ba-da-bing!")
    }

    fun playSnare() {
        if (hasSnare) println("bang bang bang!")
    }

}

fun main(args: Array<String>) {

    val d = DrumKit(true, true)
    d.playTopHat()
    d.playSnare()
    d.hasSnare = false
    d.playTopHat()
    d.playSnare()

}
```

Функция playTopHat выводит текст, если свойство hasTopHat равно true.

Функция playSnare выводит текст, если свойство hasSnare равно true.

Создать переменную DrumKit.

Оба свойства, hasTopHat и hasSnare, равны true, поэтому обе функции, playTopHat и playSnare, выводят текст.

Так как свойству hasSnare задается значение false, выводит текст только функция playTopHat.

Подробнее о свойствах

Пока что мы вам показали, как определить свойство в конструкторе класса и как присвоить значение свойству при вызове конструктора. А если нужно сделать что-то другое? Что, если вы хотите проверить значение перед тем, как присваивать его свойству? Или если свойство должно инициализироваться значением по умолчанию, чтобы его не нужно было включать в конструктор класса?

Чтобы узнать, как это делается, необходимо внимательнее присмотреться к коду конструктора.

Под капотом конструктора Dog

Как вы уже знаете, текущий конструктор `Dog` определяет три свойства, `name`, `weight` и `breed`, для каждого объекта `Dog` и присваивает значение каждому свойству при вызове конструктора `Dog`:

```
class Dog(val name: String, var weight: Int, val breed: String) {
    ...
}
```

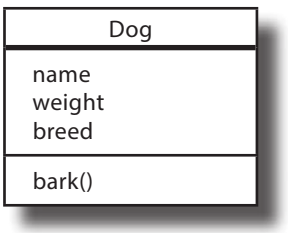
Такое компактное решение работает потому, что в коде конструктора используется сокращенная запись для подобных операций. Во время проектирования языка Kotlin его разработчики решили, что определение и инициализация свойств — настолько распространенное действие, что синтаксис для него должен быть предельно компактным и простым.

Если бы то же действие выполнялось без использования сокращенной записи, код выглядел бы примерно так:

Параметры конструктора не имеют префиксов val и var, поэтому конструктор не создает для них свойства.

```
class Dog(name_param: String, weight_param: Int, breed_param: String) {
    val name = name_param
    var weight = weight_param
    val breed = breed_param

    ...
}
```

Здесь свойства определяются в теле класса.

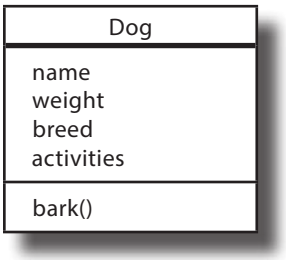


Здесь три параметра конструктора — `name_param`, `weight_param` и `breed_param` — не имеют префиксов `val` и `var`, а это означает, что они не определяют свойства. Это самые обычные параметры, ничем не отличающиеся от тех, которые встречались вам в определениях функций. Свойства `name`, `weight` и `breed` определяются в теле класса, и каждому из них присваивается значение соответствующего параметра конструктора.

Как это помогает нам в работе со свойствами?

Гибкая инициализация свойств

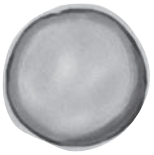
Определение свойств в теле класса обеспечивает намного большую гибкость, чем простое добавление их в конструктор, так как в этом случае не приходится инициализировать каждое свойство значением параметра конструктора.



Предположим, вы хотите определить для каждого свойства значение по умолчанию, не включая его в конструктор. Допустим, вы хотите добавить в класс `Dog` свойство `activities` и инициализировать его массивом, который по умолчанию содержит значение «Walks». Код, который это делает, выглядит так:

```
class Dog(val name: String, var weight: Int, val breed: String) {
    var activities = arrayOf("Walks")
    ...
}
```

Каждый созданный объект Dog содержит свойство activities. Его исходным значением является массив, содержащий значение «Walks».



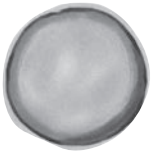
name: “Fido”
weight: 70
breed: “Mixed”
activities: “Walks”

Dog

А может оказаться, что вы хотите изменить значение параметра конструктора перед тем, как присваивать его свойству. Допустим, свойству `breed` вместо значения, переданного конструктору, должна присваиваться версия строки, преобразованная к верхнему регистру. Для этого функция `toUpperCase` создает версию строки, преобразованную к верхнему регистру, которая затем присваивается свойству `breed`:

```
class Dog(val name: String, var weight: Int, breed_param: String) {
    var activities = arrayOf("Walks")
    val breed = breed_param.toUpperCase()
    ...
}
```

Берет значение breed_param, преобразует его к верхнему регистру и присваивает свойству breed.



name: “Fido”
weight: 70
breed: “MIXED”
activities: “Walks”

Dog

Этот способ инициализации свойств хорошо работает, если вы хотите присвоить простое значение или выражение. А если понадобится сделать что-то более сложное?

Как использовать блоки инициализации

Если свойство должно инициализироваться чем-то более сложным, чем простое выражение, или если при создании каждого объекта должен выполняться дополнительный код, можно использовать один или несколько **блоков инициализации**. Блоки инициализации выполняются при инициализации объекта сразу же после вызова конструктора и снабжаются префиксом **`init`**. Следующий блок инициализации выводит сообщение каждый раз, когда инициализируется объект `Dog`:

```
class Dog(val name: String, var weight: Int, breed_param: String) {
    var activities = arrayOf("Walks")
    val breed = breed_param.toUpperCase()

    init {
        println("Dog $name has been created.")
    }

    ...
}
```

Это блок инициализации. Он содержит код, который должен выполняться при инициализации объекта Dog.

Ваш класс может содержать несколько блоков инициализации. Они выполняются в том порядке, в котором определяются в теле класса, чередуясь с инициализаторами свойств. Пример кода с несколькими блоками инициализации:

```
class Dog(val name: String, var weight: Int, breed_param: String) {

    init {
        println("Dog $name has been created.")
    }

    var activities = arrayOf("Walks")
    val breed = breed_param.toUpperCase()

    init {
        println("The breed is $breed.")
    }

    ...
}
```

Сначала создаются свойства, определенные в конструкторе.

Затем выполняется блок инициализации.

Эти свойства создаются после завершения первого блока инициализации.

Второй блок инициализации выполняется после создания свойств.

Итак, существуют разные способы инициализации переменных.
Но есть ли в них необходимость?

Свойства ДОЛЖНЫ инициализироваться

В главе 2 вы узнали, что каждая переменная, объявленная в функции, должна быть инициализирована перед использованием. Это относится ко всем свойствам, определяемым в классе, — **свойства должны инициализироваться перед их использованием в коде**. Это настолько важно, что при объявлении свойства без его инициализации в объявлении свойства или в блоке инициализации компилятор откажется компилировать ваш код. Например, следующий код не будет компилироваться из-за добавления нового свойства `temperament`, которое не было инициализировано:

```
class Dog(val name: String, var weight: Int, breed_param: String) {
    var activities = arrayOf("Walks")
    val breed = breed_param.toUpperCase()
    var temperament: String
    ...
}
```

Свойство temperament не было инициализировано, поэтому код не компилируется.

Практически во всех случаях вы сможете назначать свойствам значения по умолчанию. Так, в приведенном выше примере код будет компилироваться, если инициализировать свойство `temperament` значением `""`:

```
var temperament = ""
```

Инициализирует свойство temperament пустой строкой.

Часто Задаваемые Вопросы

В: В Java переменные, объявленные внутри класса, инициализировать не обязательно. Можно ли избежать инициализации свойств класса в Kotlin?

О: Если вы абсолютно уверены в том, что исходное значение свойства не может быть задано при вызове конструктора класса, это свойство можно снабдить префиксом **`lateinit`**. Тем самым вы сообщаете компилятору, что свойство намеренно еще не было инициализировано, и вы решите эту проблему позднее. Например, если вы хотите отметить свойство *temperament* для отложенной инициализации, используется следующая конструкция:

```
lateinit var temperament: String
```

Тогда компилятор сможет скомпилировать ваш код. Однако в общем случае мы настоятельно рекомендуем инициализировать свойства.

В: Что произойдет, если я попытаюсь использовать значение свойства до того, как оно было инициализировано?

О: Если свойство не было инициализировано при попытке его использования, при выполнении кода произойдет ошибка времени выполнения.

В: `lateinit` может использоваться с любыми типами свойств?

О: Префикс `lateinit` может использоваться только со свойствами, определенными с ключевым словом `var`; кроме того, он не может использоваться со следующими типами: `Byte`, `Short`, `Int`, `Long`, `Double`, `Float`, `Char` или `Boolean`, что связано с особенностями обработки этих типов при выполнении кода в JVM. Это означает, что свойства этих типов должны инициализироваться при определении свойства или в блоке инициализации.

Пустые конструкторы под увеличительным стеклом

Чтобы быстро создавать объекты без передачи значений каких-либо свойств, определите класс без конструктора.

Допустим, вы хотите быстро создавать объекты `Duck`. Для этого определите класс `Duck` без конструктора:

```
class Duck {
    fun quack() {
        println("Quack! Quack! Quack!")
    }
}
```

После имени класса нет круглых скобок (), а следовательно, конструктор класса не определяется.

Когда вы определяете класс без конструктора, компилятор незаметно генерирует его за вас. Он добавляет *пустой конструктор* (конструктор без параметров) в откомпилированный код. Таким образом, при компилировании приведенного выше класса `Duck` компилятор обрабатывает его так, как если бы вы написали следующий код:

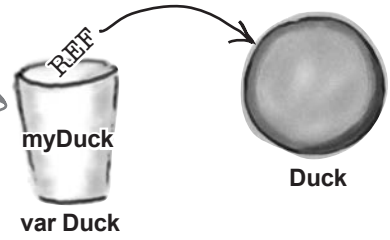
```
class Duck() {
    fun quack() {
        println("Quack! Quack! Quack!")
    }
}
```

Пустой конструктор: конструктор без параметров. При определении класса без конструктора компилятор автоматически включает пустой конструктор в откомпилированный код.

Таким образом, для создания объекта `Duck` используется следующий код:

```
var myDuck = Duck()
```

Создает переменную Duck и присваивает ей ссылку на объект Duck.



Но не такой код:

```
var myDuck = Duck
```

Этот код не компилируется.

Компилятор создал за вас пустой конструктор для класса `Duck`. Это означает, что вы *должны* вызвать пустой конструктор для создания экземпляра `Duck`.

СТАНЬ компилятором

Каждый блок кода Kotlin на этой странице представляет полный исходный файл. Попробуйте представить себя на месте компилятора и определить, будет ли компилироваться каждый из этих файлов. Если какие-то файлы не компилируются, то как бы вы их исправили?

A

```
class TapeDeck {
    var hasRecorder = false

    fun playTape() {
        println("Tape playing")
    }

    fun recordTape() {
        if (hasRecorder) {
            println ("Tape recording")
        }
    }
}

fun main(args: Array<String>) {
    t.hasRecorder = true
    t.playTape()
    t.recordTape()
}
```

B

```
class DVDPlayer(var hasRecorder: Boolean) {

    fun recordDVD() {
        if (hasRecorder) {
            println ("DVD recording")
        }
    }
}

fun main(args: Array<String>) {
    val d = DVDPlayer(true)
    d.playDVD()
    d.recordDVD()
}
```

Ответы на с. 149.

Как проверить значения свойств?

Ранее в этой главе вы узнали, как напрямую прочитать или задать значение свойства оператором «точка». Например, для вывода свойства `name` объекта `Dog` используется следующая команда:

```
println(myDog.name)
```

А команда присваивания `weight` значения 75 выглядит так:

```
myDog.weight = 75
```

Но в руках постороннего прямой доступ ко всем свойствам может стать опасным оружием. Ведь ничто не помешает злоумышленнику написать следующий код:

```
myDog.weight = -1
```

Все плохо.

Объект `Dog` с отрицательным весом (`weight`) неизбежно вызовет проблемы.

Чтобы избежать подобных ситуаций, необходимо как-то проверить значение перед тем, как присваивать его свойству.

Решение: пользовательские get- и set-методы

Если вы хотите внести изменения в прочитанное значение свойства или проверить значение перед тем, как оно будет записано в свойство, напишите собственные **get- и set-методы**.

Get- и set-методы предназначены для чтения и записи значений свойств. Единственное предназначение get-метода — вернуть значение, запрошенное для данного свойства. А set-методы получают значение аргумента и используют его для записи значения в свойство.

Если вы предпочитаете формальную терминологию, называйте их «методами чтения» и «методами записи».

Специализированные get- и set-методы позволяют защитить значения свойств и управлять тем, какие значения читаются или записываются в свойства. Чтобы показать, как они работают, мы добавим две новые характеристики в класс `Dog`:

- **Get-метод для получения веса Dog в килограммах.**
- **Set-метод для проверки предполагаемого значения свойства weight перед его присваиванием.**

Начнем с создания пользовательского get-метода, который будет пересчитывать вес `Dog` в килограммах (вместо фунтов).

Как написать пользовательский get-метод

Чтобы добавить пользовательский get-метод, который будет пересчитывать значение `weight` объектов `Dog` в килограммах, необходимо сделать две вещи: добавить в класс `Dog` новое свойство `weightInKgs` и написать для него пользовательский get-метод, который будет возвращать соответствующее значение. Следующий фрагмент решает обе задачи:

```
class Dog(val name: String, var weight: Int, breed_param: String) {
    var activities = arrayOf("Walks")
    val breed = breed_param.toUpperCase()
    val weightInKgs: Double
        get() = weight / 2.2
    ...
}
```

Код добавляет новое свойство weightInKgs с пользовательским get-методом. Get-метод получает значение параметра weight и делит его на 2,2, чтобы вычислить значение в килограммах.

Строка:

```
get() = weight / 2.2
```

определяет get-метод. Он представляет собой функцию без параметров с именем **`get`**, которая добавляется в свойство. Чтобы включить его в свойство, следует записать его сразу же под объявлением свойства. Тип возвращаемого значения **должен** совпадать с типом свойства, значение которого должен возвращать get-метод, в противном случае код не будет компилироваться. В приведенном примере свойство `weightInKgs` имеет тип `Double`, так что get-метод свойства тоже должен возвращать тип `Double`.

Формально get- и set-методы являются необязательными частями объявления свойства.

Каждый раз, когда вы запрашиваете значение свойства в синтаксисе вида:

```
myDog.weightInKgs
```

вызывается get-метод свойства. Например, этот код вызывает get-метод для свойства `weightInKgs`. Get-метод использует свойство `weight` объекта `Dog` в килограммах и возвращает результат.

Заметим, что в данном случае инициализировать свойство `weightInKgs` не нужно, потому что его значение будет вычислено get-методом. Каждый раз, когда запрашивается значение свойства, вызывается get-метод, который вычисляет возвращаемое значение.

Теперь, когда вы знаете, как добавить пользовательский get-метод, посмотрим, как добавить пользовательский set-метод, увеличивающий свойство `weight` на 1.

Часто Задаваемые Вопросы

В: Разве нельзя написать обычную функцию для вычисления веса в килограммах?

О: Можно, но иногда бывает удобнее создать новое свойство с get-методом. Например, многие библиотеки позволяют связать компонент GUI со свойством, так что создание нового свойства в таких случаях может существенно упростить вашу работу.

Как написать пользовательский set-метод

Дополним свойство `weight` пользовательским set-методом, чтобы `weight` могло обновляться только значениями больше 0. Для этого необходимо переместить определение свойства `weight` из конструктора в тело класса, а затем добавить set-метод к свойству. Это делает следующий код:

```
class Dog(val name: String, weight_param: Int, breed_param: String) {
    var activities = arrayOf("Walks")
    val breed = breed_param.toUpperCase()
    var weight = weight_param
        set(value) {
            if (value > 0) field = value
        }
    ...
}
```

Этот код добавляет пользовательский set-метод для свойства weight. Благодаря наличию set-метода свойство weight будет обновляться только значениями, которые больше нуля.

Set-метод определяется следующим кодом:

```
set(value) {
    if (value > 0) field = value
}
```

Set-метод представляет собой функцию с именем **`set`**, которая записывается под объявлением свойства. Set-метод имеет один параметр (обычно с именем `value`), который содержит новое значение свойства.

В приведенном примере значение свойства `weight` обновляется только в том случае, если значение параметра `value` больше 0. Если попытаться обновить свойство `weight` значением, меньшим либо равным 0, set-метод запретит обновление свойства.

Для обновления свойства `weight` set-метод использует идентификатор **`field`**, обозначающий поле данных для свойства. Его можно рассматривать как ссылку на нижележащее значение свойства. Очень важно использовать `field` в get- и set-методах вместо имени свойства, потому что так вы предотвращаете зацикливание. Например, при выполнении следующего кода set-метода система попытается обновить свойство `weight`, из-за чего set-метод будет вызван снова... и снова... и снова:

```
var weight = weight_param
    set(value) {
        if (value > 0) weight = value
    }
```

Не делайте так, иначе программа зациклится! Используйте ключевое слово field.

Set-метод свойства выполняется при каждой попытке присваивания значения свойства. Например, следующий код вызывает set-метод свойства weight и передает ему значение 75:

myDog.weight = 75

Сокрытие данных под увеличительным стеклом

К а к было показано на нескольких последних страницах, пользовательские get- и set-методы позволяют защитить эти свойства от некорректного использования. Пользовательский get-метод дает возможность управлять тем, какое значение будет возвращаться при запросе значения свойства, а пользовательский set-метод — проверить значение перед тем, как оно будет задано свойству.

За кулисами компилятор незаметно генерирует get- и set-методы для всех свойств, у которых этого свойства нет. Если свойство определяется с ключевым словом `val`, то компилятор добавляет get-метод, а если свойство определяется с ключевым словом `var`, то компилятор добавляет и get-, и set-метод. Таким образом, когда вы пишете следующий код:

Свойству val не нужен set-метод, потому что после инициализации его значение невозможно изменить.

```
var myProperty: String
```

компилятор незаметно добавляет следующие get- и set-методы во время компиляции:

```
var myProperty: String
    get() = field
    set(value) {
        field = value
    }
```

Это означает, что каждый раз, когда вы используете оператор «точка» для чтения или записи значения свойства, за кулисами **всегда вызывается get- или set-метод этого свойства**.

Почему компилятор действует именно так?

Добавление get- и set-методов для каждого свойства означает, что у объектов существует стандартный механизм обращения к значениям этих свойств. Get-метод обрабатывает любые запросы на получение значения, а set-метод обрабатывает любые запросы на их запись. Если позднее вы решите изменить реализацию таких запросов, это можно будет сделать без нарушения работоспособности кода.

Блокировка прямого доступа к значению свойства, «обернутого» в get- и set-методы, называется сокрытием данных.

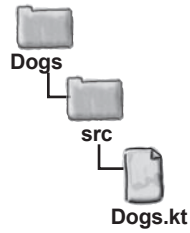
Полный код проекта Dogs

Глава подходит к концу, но прежде чем двигаться дальше, мы решили показать полный код проекта Dogs.

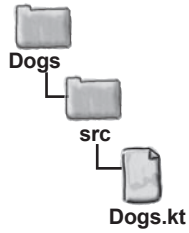
Создайте новый проект Kotlin для JVM и присвойте ему имя «Dogs». Затем создайте новый файл Kotlin с именем *Dogs.kt*: выделите папку *src*, откройте меню File и выберите команду New → Kotlin File/Class. По запросу введите имя файла «Dogs» и выберите вариант File в группе Kind.

Затем добавьте следующий код в файл *Dogs.kt*:


```
class Dog(val name: String,
          weight_param: Int,
          breed_param: String) {

    init {
        print("Dog $name has been created. ")
    }

    var activities = arrayOf("Walks")
    val breed = breed_param.toUpperCase()

    init {
        println("The breed is $breed.")
    }

    var weight = weight_param
        set(value) {
            if (value > 0) field = value
        }

    val weightInKgs: Double
        get() = weight / 2.2

    fun bark() {
        println(if (weight < 20) "Yip!" else "Woof!")
    }
}
```



Код Dogs (продолжение)...




```
fun main(args: Array<String>) {
    val myDog = Dog("Fido", 70, "Mixed")
    myDog.bark()
    myDog.weight = 75
    println("Weight in Kgs is ${myDog.weightInKgs}")
    myDog.weight = -2
    println("Weight is ${myDog.weight}")
    myDog.activities = arrayOf("Walks", "Fetching balls", "Frisbee")
    for (item in myDog.activities) {
        println("My dog enjoys $item")
    }

    val dogs = arrayOf(Dog("Kelpie", 20, "Westie"), Dog("Ripper", 10, "Poodle"))
    dogs[1].bark()
    dogs[1].weight = 15
    println("Weight for ${dogs[1].name} is ${dogs[1].weight}")
}
```

Тест-драйв

При выполнении кода в окне вывода IDE появляется следующий результат :

```
Dog Fido has been created. The breed is MIXED.
Woof!
Weight in Kgs is 34.090909090909086
Weight is 75
My dog enjoys Walks
My dog enjoys Fetching balls
My dog enjoys Frisbee
Dog Kelpie has been created. The breed is WESTIE.
Dog Ripper has been created. The breed is POODLE.
Yip!
Weight for Ripper is 15
```

У бассейна

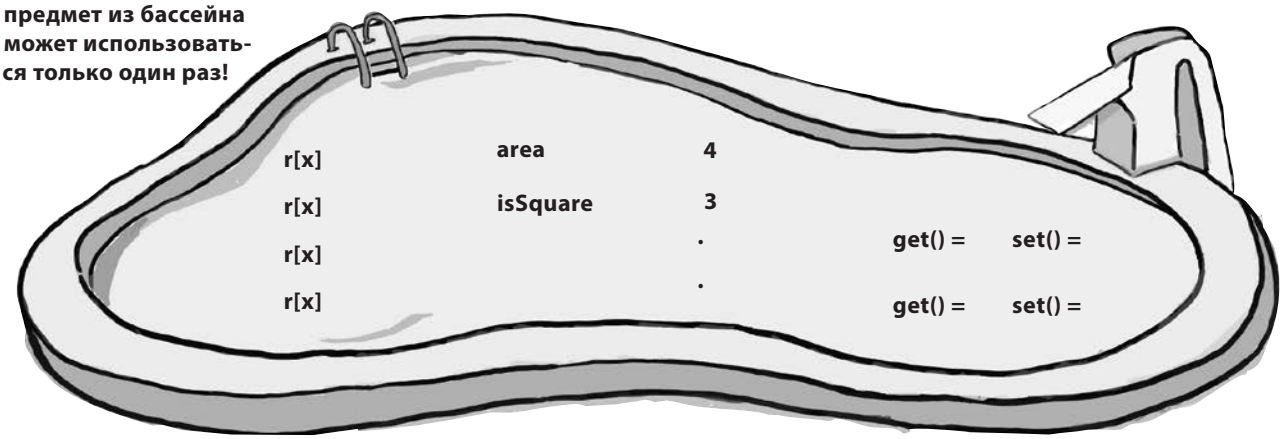
Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: выложить код, который выводит указанный результат.

Программа должна выводить этот результат.

```
Rectangle 0 has area 15. It is not a square.
Rectangle 1 has area 36. It is a square.
Rectangle 2 has area 63. It is not a square.
Rectangle 3 has area 96. It is not a square.
```

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

    val area: Int
        ..............(width * height)
}

fun main(args: Array<String>) {
    val r = arrayOf(Rectangle(1, 1), Rectangle(1, 1),
                    Rectangle(1, 1), Rectangle(1, 1))
    for (x in 0........) {
        ........width = (x + 1) * 3
        ........height = x + 5
        print("Rectangle $x has area ${..........................}. ")
        println("It is ${if (..........................) "" else "not "}a square.")
    }
}
```

Примечание: каждый предмет из бассейна может использоваться только один раз!



У бассейна. Решение

Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: выложить код, который выводит указанный результат.

```
Rectangle 0 has area 15. It is not a square.
Rectangle 1 has area 36. It is a square.
Rectangle 2 has area 63. It is not a square.
Rectangle 3 has area 96. It is not a square.
```

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

    val area: Int
        get() = (width * height)
}

fun main(args: Array<String>) {
    val r = arrayOf(Rectangle(1, 1), Rectangle(1, 1),
                    Rectangle(1, 1), Rectangle(1, 1))
    for (x in 0..3) {
        r[x].width = (x + 1) * 3
        r[x].height = x + 5
        print("Rectangle $x has area ${r[x].area}. ")
        println("It is ${if (r[x].isSquare) "" else "not "}a square.")
    }
}
```

Этот get-метод проверяет, является ли прямоугольник квадратом.

Get-метод для вычисления площади прямоугольника.

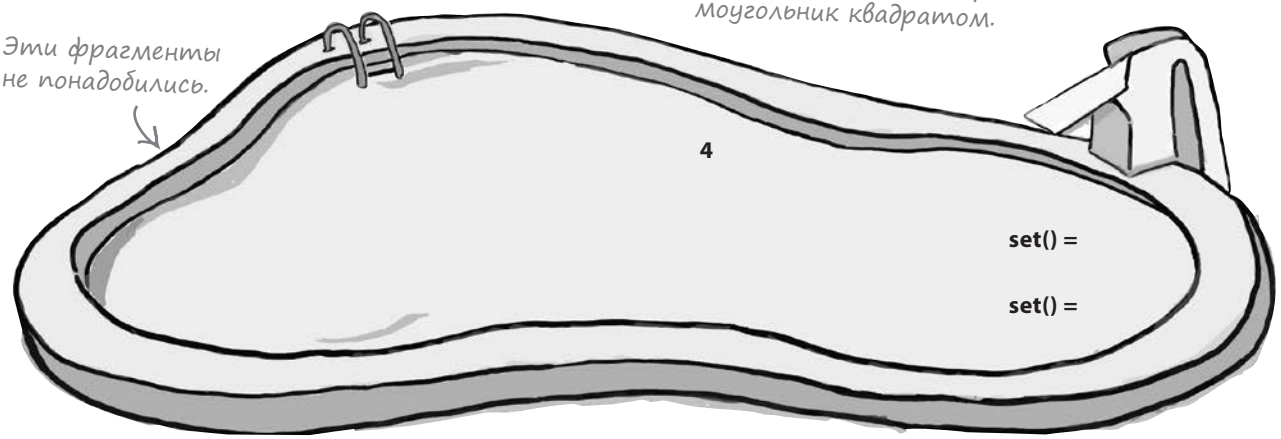
Массив r состоит из 4 элементов, поэтому в цикле индекс меняется с 0 до 3.

Назначение ширины и высоты прямоугольника.

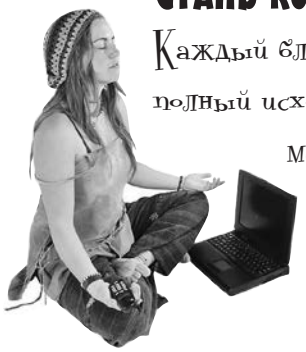
Выводит площадь прямоугольника.

Выводит сообщение о том, является ли прямоугольник квадратом.

Эти фрагменты не понадобились.



СТАНЬ компилятором



Каждый блок кода Kotlin на этой странице представляет полный исходный файл. Попробуйте представить себя на месте компилятора и определить, будет ли компилироваться каждый из этих файлов. Если какие-то файлы не компилируются, то как бы вы их исправили?

A

```
class TapeDeck {
    var hasRecorder = false

    fun playTape() {
        println("Tape playing")
    }

    fun recordTape() {
        if (hasRecorder) {
            println ("Tape recording")
        }
    }
}

fun main(args: Array<String>) {
    val t = TapeDeck()
    t.hasRecorder = true
    t.playTape()
    t.recordTape()
}
```

Не компилируется, потому что объект TapeDeck должен быть создан перед использованием.

B

```
class DVDPlayer(var hasRecorder: Boolean) {

    fun playDVD() {
      println("DVD playing")
    }

    fun recordDVD() {
        if (hasRecorder) {
            println ("DVD recording")
        }
    }
}

fun main(args: Array<String>) {
    val d = DVDPlayer(true)
    d.playDVD()
    d.recordDVD()
}
```

Не компилируется, потому что класс DVDPlayer должен содержать функцию playDVD.

Ваш инструментарий Kotlin

Глава 4 осталась позади, а ваш инструментарий пополнился классами и объектами.

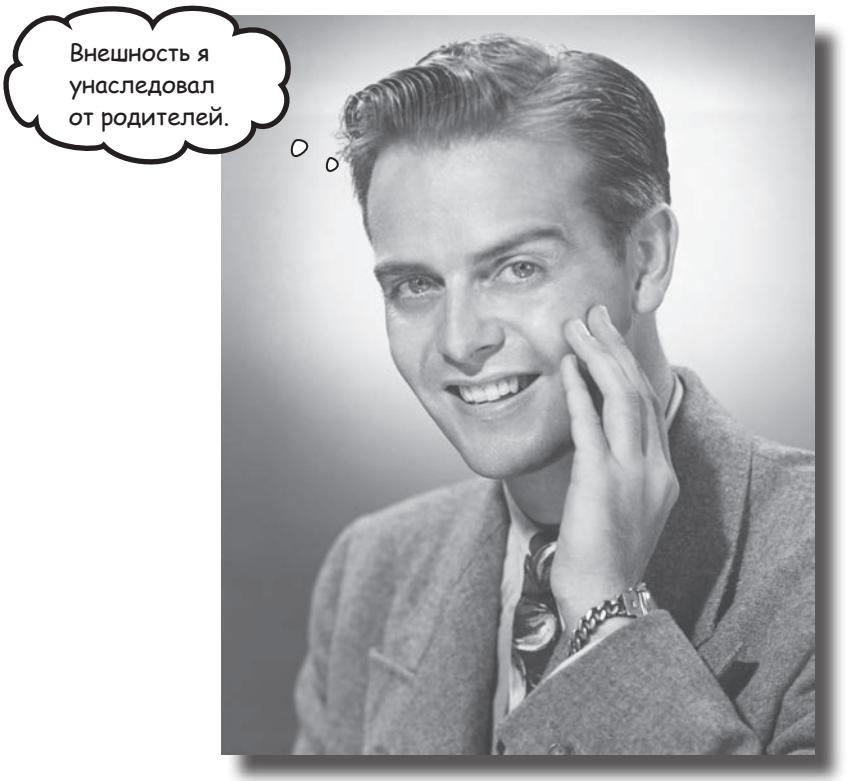
Весь код для этой главы можно загрузить по адресу https://tinyurl.com/HFKotlin.

ГЛАВА 4

КЛЮЧЕВЫЕ МОМЕНТЫ

- С помощью классов вы можете определять собственные типы.
- Класс представляет собой шаблон для создания объектов. На основе одного класса можно создать много объектов.
- То, что объект знает о себе, — свойства. То, что объект может сделать, — функции.
- Свойство представляет собой переменную, локальную по отношению к классу.
- Класс определяется ключевым словом `class`.
- Оператор «точка» используется для обращения к свойствам и функциям объекта.
- Конструктор выполняется при инициализации объекта.
- Свойство можно определить в первичном конструкторе с префиксом `val` или `var`. Свойство также можно определить вне конструктора: для этого оно добавляется в тело класса.
- Блоки инициализации выполняются при инициализации объекта.
- Каждое свойство должно быть инициализировано перед использованием его значения.
- Get- и set-методы предназначены для чтения и записи значений свойств.
- За кулисами компилятор генерирует для каждого свойства get- и set-метод по умолчанию.

Наследование



Вам когда-нибудь казалось, что если немного изменить тип объекта, то он идеально подойдет для ваших целей? Что ж, это одно из преимуществ **наследования**. В этой главе вы научитесь создавать **подклассы** и наследовать свойства и функции **суперклассов**. Вы узнаете, ***как переопределять функции и свойства,*** чтобы классы работали так, как нужно *вам,* и когда это стоит (или не стоит) делать. Наконец, вы увидите, как наследование помогает **избежать дублирования кода**, и узнаете, как сделать код более гибким при помощи **полиморфизма**.

Наследование предотвращает дублирование кода

При разработке больших приложений с множеством классов приходится задумываться о **наследовании**. Когда вы проектируете архитектуру на базе наследования, общий код размещается в одном классе, а затем более конкретные специализированные классы наследуют этот код. Если же код потребуется обновить, достаточно внести изменения в одном месте. Эти изменения отразятся во всех классах, наследующих это поведение.

Класс, содержащий общий код, называется **суперклассом**, а классы, наследующие от него, называются **подклассами**.

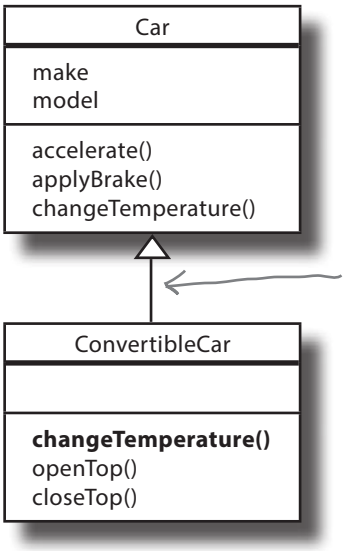
Суперклассы также называются «базовыми классами», а подклассы — «производными классами». В этой книге мы будем использовать термины «суперкласс» и «подкласс».

Пример наследования

Допустим, имеются два класса: `Car` и `ConvertibleCar`.

Класс `Car` включает свойства и функции для создания обобщенного автомобиля: свойства `make` и `model`, а также функции `accelerate`, `applyBrake` и `changeTemperature`.

Класс `ConvertibleCar` является подклассом по отношению к классу `Car`, поэтому он автоматически наследует все свойства и функции `Car`. Но класс `ConvertibleCar` также может добавлять новые функции и свойства, а также переопределять аспекты, наследованные от суперкласса `Car`:



Отношения наследования обозначаются вот такими стрелками.

Класс `ConvertibleCar` добавляет две дополнительные функции с именами `openTop` и `closeTop`. Он также переопределяет функцию `changeTemperature`, которая описывает автоматическое закрытие крыши при низкой температуре.

Суперкласс содержит общие свойства и функции, унаследованные одним или несколькими подклассами.

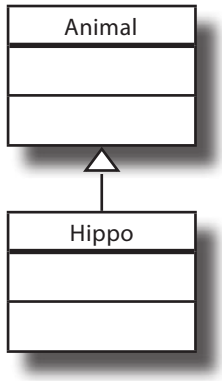
Подкласс может включать дополнительные свойства и функции, а также переопределять аспекты, унаследованные от суперкласса.

Что мы собираемся сделать

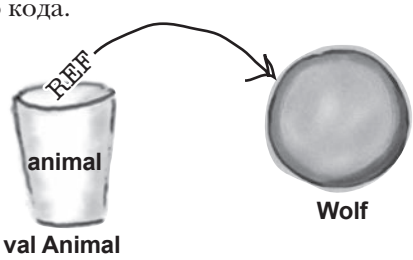
В этой главе мы покажем вам, как проектировать и запрограммировать иерархию классов наследования. Это будет сделано в три этапа:

1. **Проектирование иерархии классов животных.**
Мы возьмем несколько видов животных и спроектируем для них структуру наследования. При этом мы опишем общие шаги проектирования наследования, которые вы затем сможете применить в собственных проектах.

2. **Написание (части) кода для иерархии классов животных.**
После того как система наследования будет спроектирована, мы напишем код для некоторых классов.



3. **Написание кода, использующего иерархию классов животных.**
Мы покажем, как использовать структуру наследования для написания более гибкого кода.



Начнем с проектирования структуры наследования.

- Проектирование классов
- Построение классов
- Использование классов

Проектирование структуры наследования классов

Представьте, что вам поручили разработать структуру классов для программы моделирования, в которой пользователь помещает в условную среду различных животных и наблюдает за ними.

Мы не будем разрабатывать все приложение, сейчас нас интересует только структура классов.

Мы знаем *некоторые,* но не все типы животных, которые будут включены в приложение. Каждое животное будет представлено объектом и будет делать то, на что запрограммирован этот конкретный тип животного.

Возможно, в будущем в приложение будут добавляться новые типы животных. Важно, чтобы структура классов была достаточно гибкой и предоставляла возможность расширения.

Но прежде чем думать о конкретных животных, для начала необходимо определить характеристики, общие для всех животных. Эти характеристики можно будет оформить в суперкласс, от которого могут наследоваться все подклассы животных.

Мы собираемся представить общую схему проектирования иерархии наследования классов. Это первый этап.

1. **Поиск атрибутов и аспектов поведения, общих для всех классов.**
   Рассмотрим все эти типы животных. Что у них всех общего?

   Этот шаг помогает абстрагировать те атрибуты и поведение, которые могут быть включены в суперкласс.

Используйте наследование для предотвращения дублирования кода в подклассах

Мы добавим некоторые общие свойства и функции в суперкласс **Animal**, чтобы они наследовались каждым из подклассов животных. Список не претендует на полноту, но и этого достаточно, чтобы понять суть происходящего.

В суперкласс добавляются четыре **свойства**:

image: имя файла с изображением животного.

food: пища, которой питается данное животное (например, мясо или трава).

habitat: среда обитания животного: лес, саванна, вода и т. д.

hunger : значение Int, представляющее уровень голода животного; изменяется в зависимости от того, когда животное питалось (и сколько съело).

А также четыре **функции**:

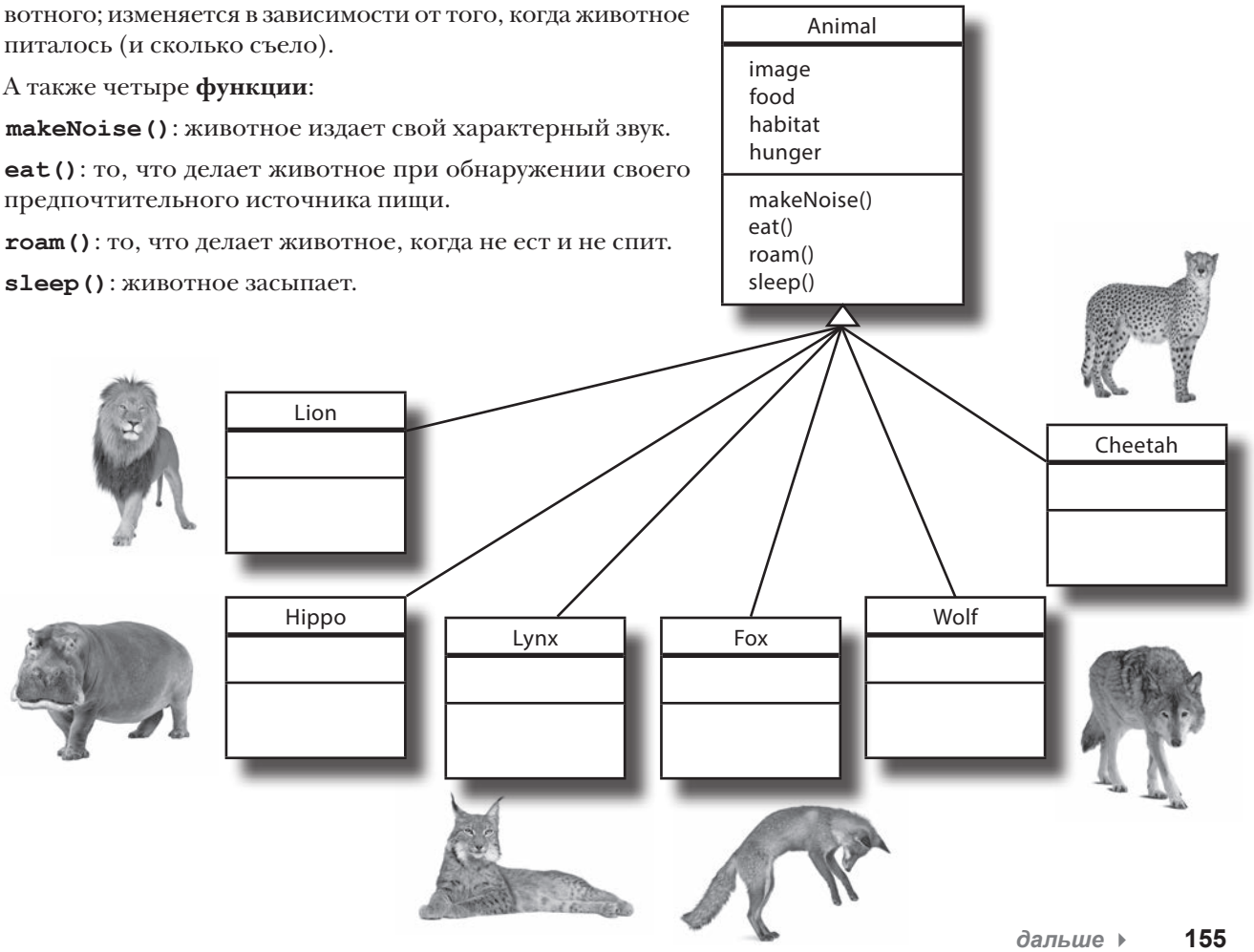
makeNoise(): животное издает свой характерный звук.

eat(): то, что делает животное при обнаружении своего предпочтительного источника пищи.

roam(): то, что делает животное, когда не ест и не спит.

sleep(): животное засыпает.

2 **Проектирование суперкласса, представляющего общее состояние и поведение.**
Свойства и функции, общие для всех животных, выделяются в новый суперкласс `Animal`. Эти свойства и функции будут наследоваться всеми подклассами животных.



Что должны переопределять подклассы?

Необходимо решить, как свойства и функции должны переопределять подклассы животных. Начнем со свойств.

Животные обладают разными значениями свойств...

Суперкласс `Animal` содержит свойства `image`, `food`, `habitat` и `hunger`; все эти свойства наследуются подклассами животных.

Все животные выглядят по-разному, живут в разных местах и имеют разные гастрономические предпочтения. Это означает, что мы можем переопределить свойства `image`, `food` и `habitat`, чтобы они по-разному инициализировались для каждого типа животного. Например, свойство `habitat` у `Hippo` будет инициализироваться значением «water», а свойство `food` у `Lion` — значением «meat».

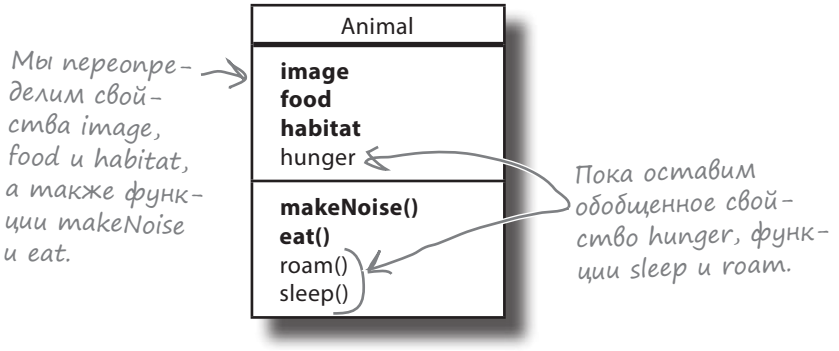
3 **Определение того, нужны ли классу значения свойств по умолчанию или реализации функций, специфические для этого подкласса.**
В своем примере мы переопределим свойств `image`, `food` и `habitat`, а также функции `makeNoise` и `eat`.

...и разными реализациями функций

Каждый подкласс животного наследует функции `makeNoise`, `eat`, `roam` и `sleep` от подкласса `Animal`. Какие же из этих функций следует переопределять?

Львы рычат, волки воют, бегемоты фыркают. Все животные издают разные звуки, это означает, что функция `makeNoise` должна переопределяться в каждом подклассе животного. Каждый подкласс по-прежнему содержит функцию function, но реализация этой функции будет меняться от животного к животному.

Кроме того, все животные едят, но *что именно* они едят — зависит от животного. Бегемот ест траву, а гепард охотится за добычей. Чтобы отразить такие разные гастрономические предпочтения, мы будем переопределять функцию `eat` в каждом подклассе животного.

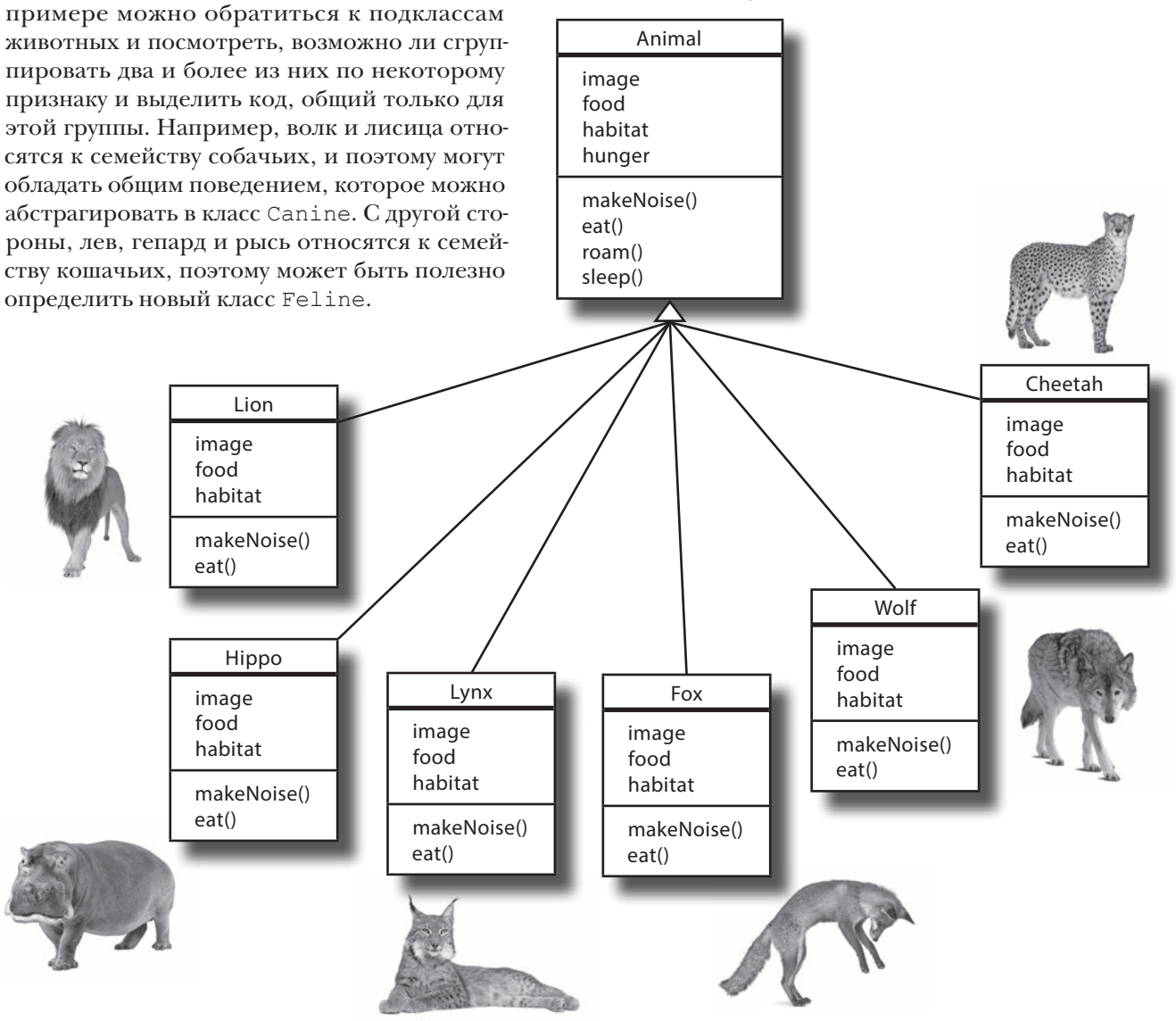


Некоторых животных можно сгруппировать

Иерархия классов постепенно начинает формироваться. Каждый подкласс переопределяет набор свойств и функций, так что никто не спутает вой волка с фырканьем бегемота.

Но это еще не все. При проектировании наследования можно построить целую **иерархию классов**, наследующих друг от друга от верхнего суперкласса и до самого низа. В нашем примере можно обратиться к подклассам животных и посмотреть, возможно ли сгруппировать два и более из них по некоторому признаку и выделить код, общий только для этой группы. Например, волк и лисица относятся к семейству собачьих, и поэтому могут обладать общим поведением, которое можно абстрагировать в класс `Canine`. С другой стороны, лев, гепард и рысь относятся к семейству кошачьих, поэтому может быть полезно определить новый класс `Feline`.

4 **Выявление дополнительных возможностей для абстрагирования свойств и функций с поиском двух и более подклассов с общим поведением.** Обратившись к подклассам, мы видим, что среди них присутствуют два представителя семейства собачьих, три представителя семейства кошачьих и бегемот (который ни к одному из этих семейств не относится).



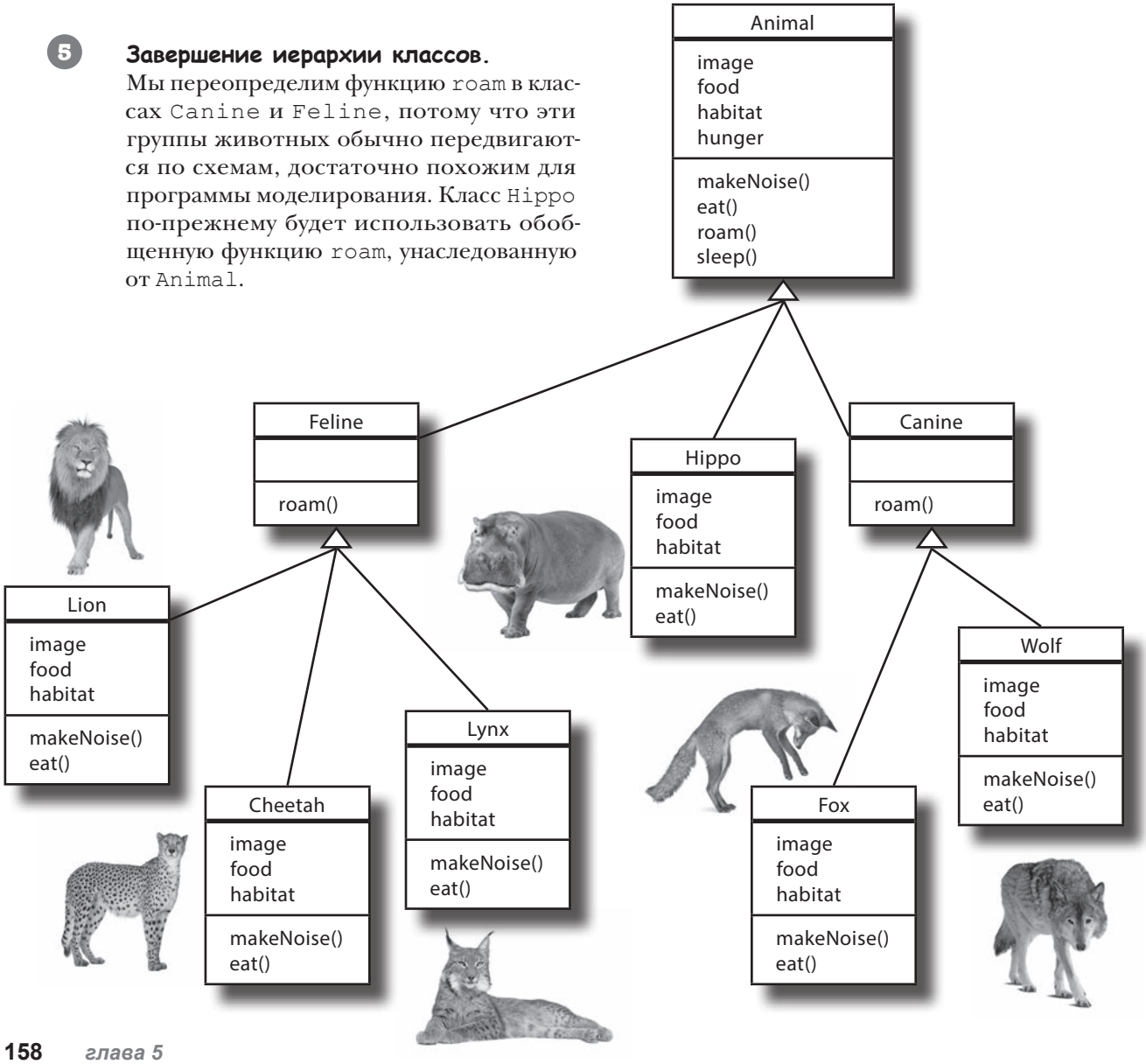
- [x] **Проектирование классов**
- [ ] Построение классов
- [ ] Использование классов

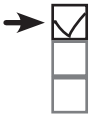
Добавление классов Canine и Feline

У животных уже имеется система классификации, и мы можем отразить ее в своей иерархии классов на наиболее логичном уровне. Мы воспользуемся биологической системой классификации и добавим в свою иерархию классы `Canine` и `Feline`. Класс `Canine` будет содержать свойства и функции, общие для представителей семейства собачьих (волки и лисы), а класс `Feline` — свойства и функции семейства кошачьих (львы, гепарды и рыси).

Каждый подкласс также может определять собственные свойства и функции, но сейчас нас интересуют исключительно общие аспекты всех животных.

5 **Завершение иерархии классов.** Мы переопределим функцию `roam` в классах `Canine` и `Feline`, потому что эти группы животных обычно передвигаются по схемам, достаточно похожим для программы моделирования. Класс `Hippo` по-прежнему будет использовать обобщенную функцию `roam`, унаследованную от `Animal`.





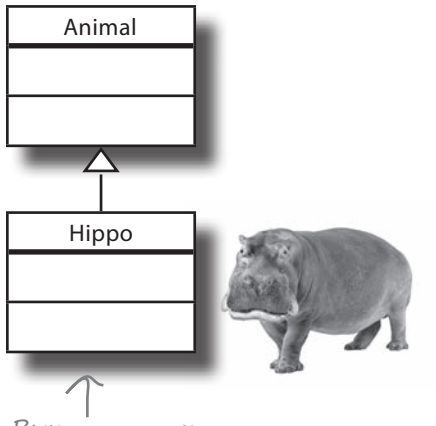
Используйте «правило ЯВЛЯЕТСЯ» для проверки иерархии классов

Чтобы при проектировании иерархии классов определить, должен ли один класс быть подклассом другого, можно воспользоваться «правилом ЯВЛЯЕТСЯ». Просто спросите себя: «Можно ли сказать, что тип X ЯВЛЯЕТСЯ (частным случаем) типа Y?» Если вы ответите утвердительно, оба класса должны принадлежать одной ветви иерархии наследования, так как, скорее всего, они обладают одинаковым или перекрывающимся поведением. Если же ответ будет *отрицательным*, тогда иерархию следует пересмотреть.

На самом деле все не так просто, но пока нам подойдет это правило. Проблемы проектирования классов более подробно рассматриваются в следующей главе.

Например, фраза «Бегемот ЯВЛЯЕТСЯ животным» выглядит вполне разумно. Бегемот — частный случай животного, так что класс `Hippo` разумно объявить подклассом `Animal`.

Учтите, что отношения «ЯВЛЯЕТСЯ» подразумевают, что если «`X` ЯВЛЯЕТСЯ `Y`», то `X` может сделать все, что может `Y` (а возможно, и больше), так что «правило ЯВЛЯЕТСЯ» работает только в одном направлении. Было бы неправильно утверждать, что «животное ЯВЛЯЕТСЯ бегемотом», потому что животное не является разновидностью бегемота.



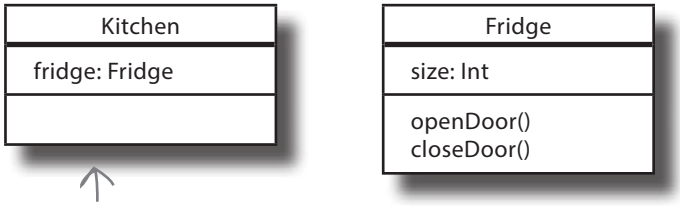
Выражение «Бегемот ЯВЛЯЕТСЯ животным» выглядит вполне осмысленно, так что класс Hippo вполне может стать подклассом Animal.

Для других отношений используется «правило СОДЕРЖИТ»

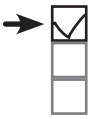
Даже если «правило ЯВЛЯЕТСЯ» не выполняется для двух классов, они все равно могут быть связаны определенным образом.

Предположим, у вас есть два класса `Fridge` (холодильник) и `Kitchen` (кухня). Выражение «Холодильник ЯВЛЯЕТСЯ кухней» бессмысленно, как и выражение «Кухня ЯВЛЯЕТСЯ холодильником». Но два класса все равно связаны друг с другом, хотя и не через наследование.

Классы `Kitchen` и `Fridge` связаны отношением «**СОДЕРЖИТ**». Выражение «Кухня СОДЕРЖИТ холодильник» разумно? Если разумно, то это означает, что класс `Kitchen` содержит свойство `Fridge`. Другими словами, `Kitchen` включает ссылку на `Fridge`, но `Kitchen` не является подклассом `Fridge`, и наоборот.



Кухня (Kitchen) СОДЕРЖИТ холодильник (Fridge), так что связь существует. Тем не менее ни один класс не является подклассом другого.



«Правило ЯВЛЯЕТСЯ» работает в любых позициях дерева наследования

В хорошо спроектированном дереве наследования «правило ЯВЛЯЕТСЯ» должно выполняться для вопроса, ЯВЛЯЕТСЯ ли *любой* подкласс любым из его супертипов.

Если класс `B` является подклассом класса `A`, класс `B` ЯВЛЯЕТСЯ классом A. **Это утверждение истинно для любых классов в дереве наследования.** Если класс `C` является подклассом `B`, то **класс `C` проходит «правило ЯВЛЯЕТСЯ» и для `B`, и для `A`.**

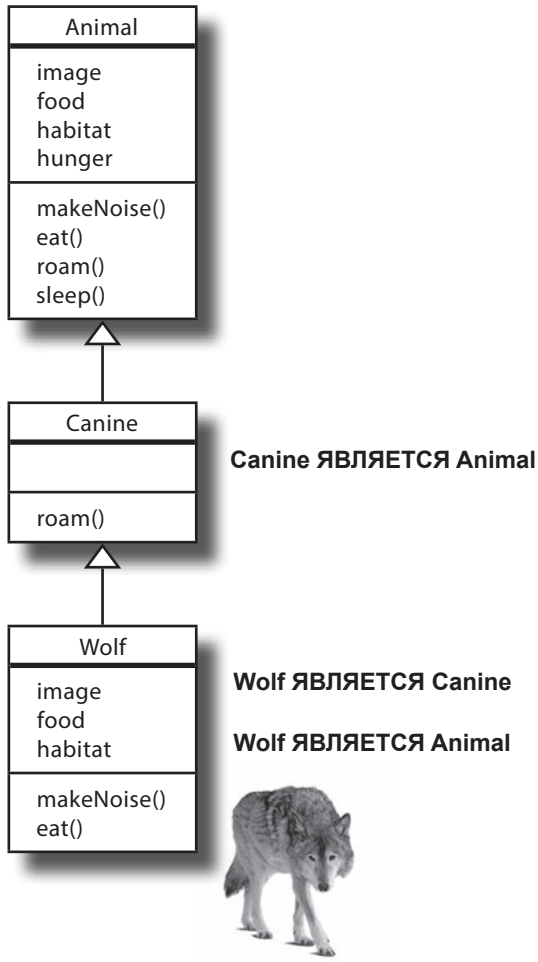
С таким деревом наследования вы всегда можете сказать: «Wolf является подклассом Animal» или «Wolf ЯВЛЯЕТСЯ Animal». При этом неважно, что `Animal` является суперклассом суперкласса `Wolf`. Если **`Animal` находится в иерархии наследования где-то выше `Wolf`, значит, `Wolf` ЯВЛЯЕТСЯ `Animal`.**

Структура дерева наследования `Animal` сообщает миру следующее:

«Wolf ЯВЛЯЕТСЯ Canine, так что Wolf может делать все, что может Canine. И Wolf ЯВЛЯЕТСЯ Animal, так что Wolf может сделать все, что может сделать Animal».

Неважно, определяет ли `Wolf` какие-либо функции `Animal` или `Canine`. В том, что касается кода, `Wolf` может выполнять эти функции. Как именно это делает `Wolf` и в каком классе они переопределяются — несущественно. `Wolf` может выполнять операции `makeNoise`, `eat`, `roam` и `sleep`, потому что `Wolf` является подклассом `Animal`.

Теперь, когда вы узнали, как проектируются иерархии классов, попробуйте выполнить следующее упражнение. После этого мы перейдем к программированию иерархии классов `Animal`.



Будьте осторожны!

Если «правило ЯВЛЯЕТСЯ» не проходит, не используйте наследование только для того, чтобы повторно использовать код из другого класса.

Представьте, что вы добавили специальный код голосовой активации в класс Alarm (сигнал), который собираетесь повторно использовать в классе Kettle (чайник). Kettle не является частным случаем Alarm, поэтому класс Kettle не должен быть подклассом Alarm. Вместо этого стоит подумать о создании отдельного класса голосовой активации VoiceActivation, которым могут пользоваться все объекты голосовой активации для использования отношений СОДЕРЖИТ. (Другие возможности проектирования будут представлены в следующей главе.)

Возьми в руку карандаш

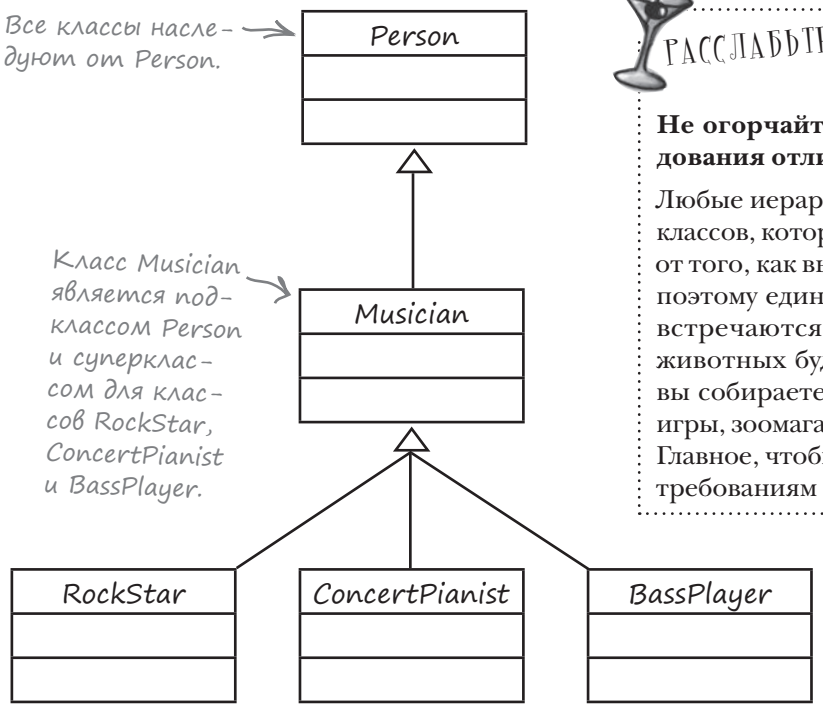
Ниже приведена таблица с именами классов: `Person` (человек), `Musician` (музыкант), `RockStar` (рок-звезда), `BassPlayer` (басист) и `ConcertPianist` (концертный пианист). Определите, какие отношения имеют смысл, и выберите суперклассы и подклассы для каждого класса. Нарисуйте дерево наследования для классов.

Класс	Суперклассы	Подклассы
Person		
Musician		
RockStar		
BassPlayer		
ConcertPianist		

Возьми в руку карандаш
Решение

Ниже приведена таблица с именами классов: `Person` (человек), `Musician` (музыкант), `RockStar` (рок-звезда), `BassPlayer` (басист) и `ConcertPianist` (концертный пианист). Определите, какие отношения имеют смысл, и выберите суперклассы и подклассы для каждого класса. Нарисуйте дерево наследования для классов.

Класс	Суперклассы	Подклассы
Person		*Musician, RockStar, BassPlayer, ConcertPianist*
Musician	*Person*	*RockStar, BassPlayer, ConcertPianist*
RockStar	*Musician, Person*	
BassPlayer	*Musician, Person*	
ConcertPianist	*Musician, Person*	



РАССЛАБЬТЕСЬ

Не огорчайтесь, если ваше дерево наследования отличается от нашего.

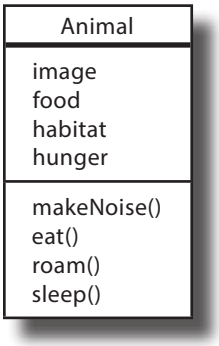
Любые иерархии наследования и структуры классов, которые вы разрабатываете, зависят от того, как вы собираетесь их использовать, поэтому единственно правильные решения встречаются редко. Например, иерархия животных будет зависеть от того, для чего вы собираетесь ее использовать: для видеоигры, зоомагазина или зоологического музея. Главное, чтобы архитектура соответствовала требованиям вашего приложения.

Создание объектов животных в Kotlin

Мы разработали иерархию классов животных, а теперь давайте напишем для нее код.

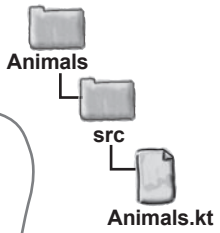
Создайте новый проект Kotlin для JVM с именем «Animals». Создайте новый файл Kotlin с именем *Animals.kt*: выделите папку *src*, откройте меню File и выберите команду New → Kotlin File/Class. Введите имя файла «Animals» и выберите вариант File в группе Kind.

Добавим в проект новый класс с именем `Animal`, который содержит код по умолчанию для создания обобщенного животного. Ниже приведен код файла, обновите свою версию *Animals.kt* и приведите ее в соответствие с нашей:




```
class Animal {
    val image = ""
    val food = ""
    val habitat = ""
    var hunger = 10

    fun makeNoise() {
        println("The Animal is making a noise")
    }

    fun eat() {
        println("The Animal is eating")
    }

    fun roam() {
        println("The Animal is roaming")
    }

    fun sleep() {
        println("The Animal is sleeping")
    }
}
```

Класс Animal содержит свойства image, food, habitat и hunger.



Мы определили реализации по умолчанию для функций makeNoise, eat, roam и sleep.

Класс `Animal` готов. Теперь необходимо сообщить компилятору, что `Animal` будет использоваться в качестве суперкласса.

Проектирование классов
Построение классов
Использование классов

Объявление суперкласса и его свойств/функций с ключевым словом open

Чтобы класс можно было использовать в качестве суперкласса, необходимо явно сообщить об этом компилятору. Для этого перед именем класса — и любым свойством и функцией, которые вы собираетесь переопределять, — ставится ключевое слово **open**. Тем самым вы сообщаете компилятору, что класс проектировался как суперкласс, и согласны с тем, что его свойства и функции, объявленные как открытые, будут переопределяться.

В нашей иерархии классов класс `Animal` должен использоваться в качестве суперкласса, а большинство их свойств и функций будет переопределяться. Ниже приведен код, который делает это. Обновите свою версию *Animals.kt* и приведите ее в соответствие с нашей (изменения выделены жирным шрифтом):

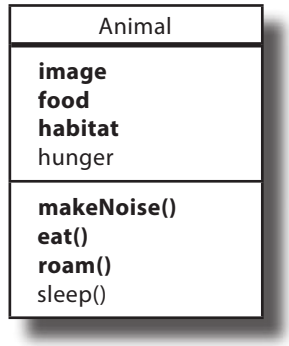
Чтобы класс мог использоваться в качестве суперкласса, он должен быть объявлен открытым (open). Все аспекты класса, которые вы хотите переопределить, тоже должны быть открытыми.

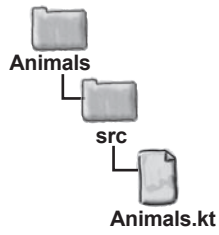
Класс будет использоваться в качестве суперкласса, поэтому он должен быть объявлен открытым.

```
open class Animal {
    open val image = ""
    open val food = ""
    open val habitat = ""
    var hunger = 10

    open fun makeNoise() {
        println("The Animal is making a noise")
    }

    open fun eat() {
        println("The Animal is eating")
    }

    open fun roam() {
        println("The Animal is roaming")
    }

    fun sleep() {
        println("The Animal is sleeping")
    }
}
```

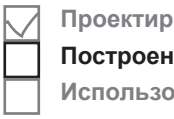
Свойства image, food и habitat должны переопределяться, поэтому каждый из них должен быть объявлен с префиксом open.

Функции makeNoise, eat и roam объявлены открытыми, потому что они будут определяться в подклассах.





Суперкласс `Animal` объявлен открытым вместе со всеми свойствами и функциями, которые вы хотите переопределить. Теперь можно переходить к созданию подклассов животных. Для этого мы напишем код класса `Hippo`.



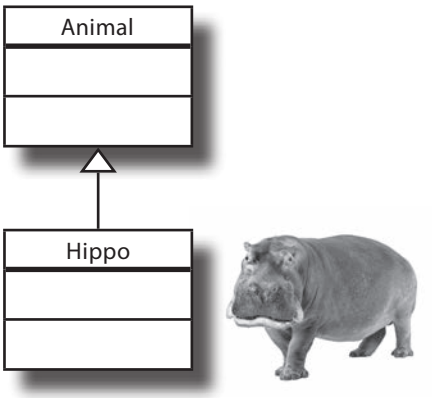
Как подкласс наследует от суперкласса

Чтобы класс наследовал от другого класса, добавьте в заголовок класса двоеточие (`:`), за которым следует имя суперкласса. Класс становится подклассом и получает все свойства и функции того класса, от которого наследуется.

В нашем случае класс `Hippo` должен наследовать от суперкласса `Animal`, поэтому код выглядит так:

```
class Hippo : Animal() {
    //Здесь размещается код Hippo
}
```

Это означает «класс Hippo является подтипом Animal». Мы добавим класс Hippo в код через несколько страниц.



`Animal()` после двоеточия (`:`) — вызов конструктора `Animal`. Он обеспечивает выполнение всего кода инициализации `Animal` — например, присваивание значений свойствам. Вызов конструктора суперкласса обязателен: **если у суперкласса есть первичный конструктор, вы *должны* вызвать его в заголовке подкласса, иначе код не будет компилироваться. И даже если вы явно не добавили конструктор в свой суперкласс, помните, что компилятор автоматически создает пустой конструктор при компиляции кода**.

Мы не добавили конструктор в свой класс Animal, поэтому компилятор добавляет пустой конструктор при компиляции кода. Animal() — вызов этого конструктора.

Если конструктор суперкласса получает параметры, значения этих параметров должны передаваться при вызове конструктора. Допустим, у вас имеется класс `Car`, конструктор которого получает два параметра с именами `make` и `model`:

```
open class Car(val make: String, val model: String) {
    //Код класса Car
}
```

Конструктор Car определяет два свойства: make и model.

Чтобы определить подкласс `Car` с именем `ConvertibleCar`, вызовите конструктор `Car` в заголовке класса `ConvertibleCar` и передайте ему значения параметров `make` и `model`. В подобных ситуациях конструктор обычно добавляется в подкласс, которому необходимы эти значения, а затем передает их конструктуру суперкласса, как в следующем примере:

Конструктор ConvertibleCar получает два параметра: make_param и model_param. Он передает значения этих параметров конструктору Car, который инициализирует свойства make и model.

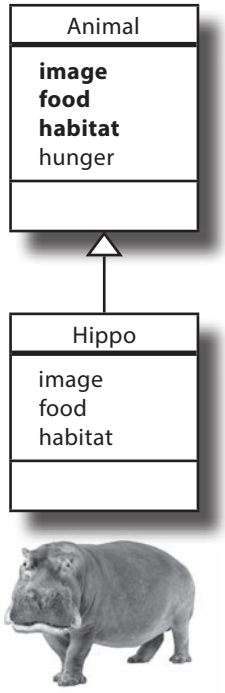
```
class ConvertibleCar(make_param: String,
                     model_param: String) : Car(make_param, model_param) {
    //Код класса ConvertibleCar
}
```

Итак, вы знаете, как объявляются суперклассы, и теперь мы можем перейти к переопределению его свойств и функций. Начнем со свойств.

Как (и когда) переопределяются свойства

Чтобы переопределить свойство, унаследованное от суперкласса, добавьте свойство в подкласс и поставьте перед ним ключевое слово **`override`**.

В нашем примере нужно переопределить свойства `image`, `food` и `habitat`, унаследованные классом `Hippo` от суперкласса `Animal`, чтобы они инициализировались значениями, специфическими для `Hippo`. Код выглядит так:



Переопределяет свойства image, food и habitat из класса Animal.

Мы добавим класс Hippo в проект через несколько страниц.

```
class Hippo : Animal() {
    override val image = "hippo.jpg"
    override val food = "grass"
    override val habitat = "water"
}
```

В своем примере мы переопределим три свойства для того, чтобы они инициализировались значениями, отличными от значения из суперкласса. Это связано с тем, что каждое свойство определяется в суперклассе `Animal` с ключевым словом `val`.

Как вы узнали на предыдущей странице, если класс наследуется от суперкласса, вы обязаны вызвать конструктор суперкласса: это нужно для того, чтобы суперкласс мог выполнить свой код инициализации, включая создание свойств и их инициализацию. Это означает, что **при определении свойства в суперклассе с ключевым словом `val` вы *обязаны* переопределить его в подклассе, если хотите присвоить ему другое значение**.

Если свойство суперкласса определяется с ключевым словом `var`, то переопределять его для присваивания нового значения не обязательно, так как переменные `var` могут повторно использоваться для других значений. Можно присвоить ему новое значение в блоке инициализации подкласса, как в следующем примере:

```
open class Animal {
    var image = ""
    ...
}
```

Здесь свойство image определяется с ключевым словом var и инициализируется значением "".

```
class Hippo : Animal() {
    init {
        image = "hippo.jpg"
    }
    ...
}
```

Для присваивания нового значения свойству image используется блок инициализации Hippo. В данном случае переопределять свойство не обязательно.

Возможности переопределения свойств не сводятся к присваиванию значений по умолчанию

До настоящего момента рассматривалось лишь переопределение свойств с целью инициализации их значением, отличным от значения из суперкласса, но это не единственный пример того, как переопределение свойств может упростить структуру классов:

- **Переопределение get- и set-методов свойств.** В предыдущей главе вы научились добавлять пользовательские get- и set-методы для свойств. Если вы хотите, чтобы свойство использовало другой get- или set-метод вместо унаследованного от суперкласса, переопределите свойство и добавьте соответствующий get- или set-метод в подклассе.

- **Переопределение свойства val в суперклассе свойством var в подклассе.** Если свойство в суперклассе было определено с ключевым словом `val`, в подклассе оно может быть переопределено как свойство `var`. Для этого просто переопределите свойство и объявите его с ключевым словом `var`. Учтите, что замена работает только в одном направлении: при попытке переопределить свойство `var` с ключевым словом `val` компилятор откажется компилировать ваш код.

- **Переопределение типа свойства одним из подтипов версии суперкласса.** Когда вы переопределяете свойство, его тип должен соответствовать типу версии суперкласса свойства или одному из его подтипов.

Мы разобрались с тем, как переопределяются свойства и когда это следует делать. Теперь можно заняться переопределением функций.

Часто Задаваемые Вопросы

В: Можно ли переопределить свойство, определенное в конструкторе суперкласса?

О: Да. Любые свойства, определяемые в конструкторе класса, могут иметь префикс `open` или `override`, и вы можете переопределить свойства, определенные в конструкторе суперкласса.

В: Почему я должен использовать префикс `open` для тех классов, свойств и функций, которые я собираюсь переопределить? В Java это не нужно.

О: В Kotlin можно наследовать от суперклассов и переопределять их свойства и функции только в том случае, если они снабжены префиксом `open`. В Java выбран противоположный подход.

В Java классы открыты по умолчанию, а для того, чтобы запретить наследование от классов или переопределение их переменных и методов экземпляров, используется ключевое слово `final`.

В: Почему в Kotlin выбран подход, противоположный подходу Java?

О: Потому что префикс `open` намного более явно выражает, какие классы проектировались для использования в качестве суперклассов и какие свойства и функции могут переопределяться. Такой подход соответствует одному из принципов Джошуа Блоха (Joshua Bloch) из книги *Effective Java*: «Проектируйте и документируйте классы для наследования либо запрещайте его».

Как переопределять функции

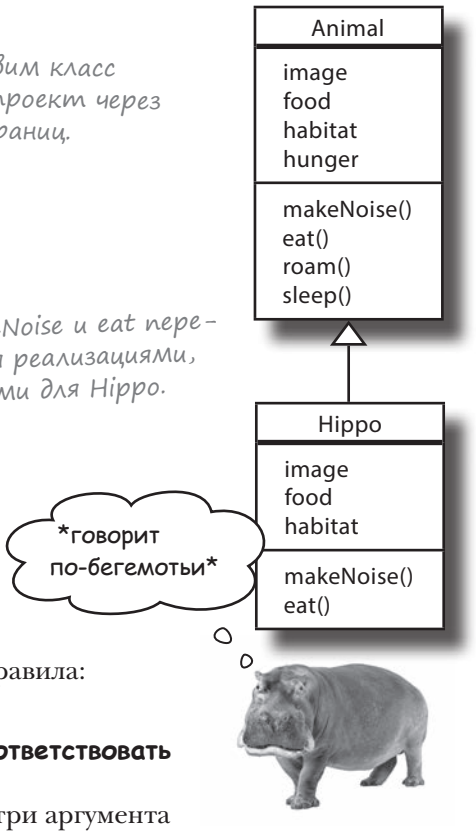
Функции переопределяются по аналогии со свойствами: функция добавляется в подкласс с префиксом `override`.

В нашем примере в классе `Hippo` должны переопределяться функции `makeNoise` и `eat`, чтобы выполняемые ими операции были специфическими для `Hippo`. Код выглядит так:

```
class Hippo : Animal() {
    override val image = "hippo.jpg"
    override val food = "grass"
    override val habitat = "water"

    override fun makeNoise() {
        println("Grunt! Grunt!")
    }

    override fun eat() {
        println("The Hippo is eating $food")
    }
}
```

Мы добавим класс Hippo в проект через пару страниц.

Функции makeNoise и eat переопределяются реализациями, специфическими для Hippo.



Правила переопределения функций

При переопределении функций необходимо соблюдать два правила:

- **Параметры функции в подклассе должны соответствовать параметрам функции в суперклассе.**
  Например, если функция в суперклассе получает три аргумента `Int`, то переопределенная функция в подклассе тоже должна получать три аргумента `Int`; в противном случае код не будет компилироваться.

- **Возвращаемые типы функций должны быть совместимыми.**
  Какой бы возвращаемый тип ни был объявлен функцией суперкласса, переопределяющая функция должна возвращать либо тот же тип, либо тип подкласса. Тип подкласса гарантированно делает все, что объявлено в его суперклассе, поэтому подкласс можно безопасно вернуть в любой ситуации, где ожидается суперкласс.

В приведенном выше коде `Hippo` переопределяемые функции не имеют параметров и возвращаемого типа. Они соответствуют определениям функций в суперклассе и поэтому выполняют правила переопределения функций.

Об использовании подклассов вместо суперкласса будет рассказано позднее в этой главе.

Переопределенная функция или свойство остаются открытыми...

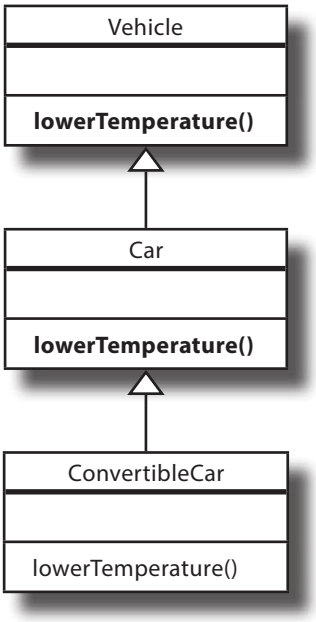
Как вы узнали ранее в этой главе, чтобы переопределить функцию или свойство, необходимо объявить их открытыми в суперклассе. Но при этом мы *не* сказали, что функция или свойство *остаются* открытыми в каждом из подклассов, даже если они были переопределены, так что вам не придется объявлять их открытыми ниже по дереву. Например, следующий код правилен:

```
open class Vehicle {
    open fun lowerTemperature() {
        println("Turn down temperature")
    }
}

open class Car : Vehicle() {
    override fun lowerTemperature() {
        println("Turn on air conditioning")
    }
}

class ConvertibleCar : Car() {
    override fun lowerTemperature() {
        println("Open roof")
    }
}
```

Класс Vehicle определяет открытую функцию lowerTemperature().

Функция lowerTemperature() остается открытой в подклассе Car, хотя мы ее переопределяем...

...это означает, что ее можно снова переопределить в классе ConvertibleCar.



...пока не будут объявлены с префиксом final

Если вы хотите запретить возможность переопределения функции или свойства ниже в иерархии классов, снабдите их префиксом **`final`**. Например, следующий код запрещает подклассу класса `Car` переопределение функции `lowerTemperature`:

Объявление функции с ключевым словом final в классе Car означает, что она более не может переопределяться в подклассах Car.

```
open class Car : Vehicle() {
    final override fun lowerTemperature() {
        println("Turn on air conditioning")
    }
}
```

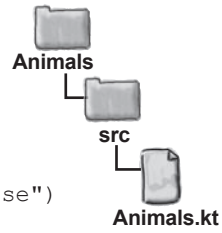
Теперь, когда вы знаете, как наследовать свойства и функции от суперкласса и переопределять их, добавим код `Hippo` в свой проект.

Добавление класса Hippo в проект Animals

Добавим код класса `Hippo` в проект Animals. Обновите свою версию кода *Animals.kt*, чтобы она соответствовала нашей (изменения выделены жирным шрифтом):

Класс Animal не изменился.

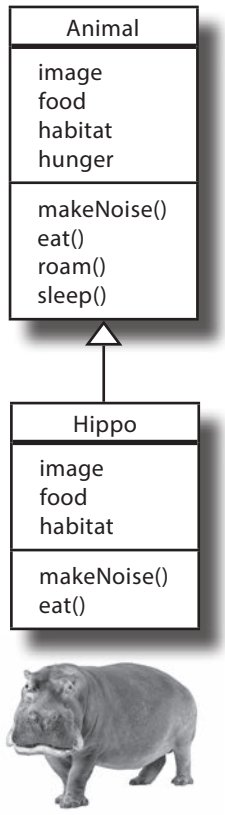
```
open class Animal {
    open val image = ""
    open val food = ""
    open val habitat = ""
    var hunger = 10

    open fun makeNoise() {
        println("The Animal is making a noise")
    }

    open fun eat() {
        println("The Animal is eating")
    }

    open fun roam() {
        println("The Animal is roaming")
    }

    fun sleep() {
        println("The Animal is sleeping")
    }
}
```



Класс Hippo является подклассом Animal.

Подкласс Hippo переопределяет эти свойства и функции.

```
class Hippo : Animal() {
    override val image = "hippo.jpg"
    override val food = "grass"
    override val habitat = "water"

    override fun makeNoise() {
        println("Grunt! Grunt!")
    }

    override fun eat() {
        println("The Hippo is eating $food")
    }
}
```



Итак, вы увидели, как создать класс `Hippo`. Посмотрим, удастся ли вам создать классы `Canine` и `Wolf` в следующем упражнении.

Развлечения с магнитами

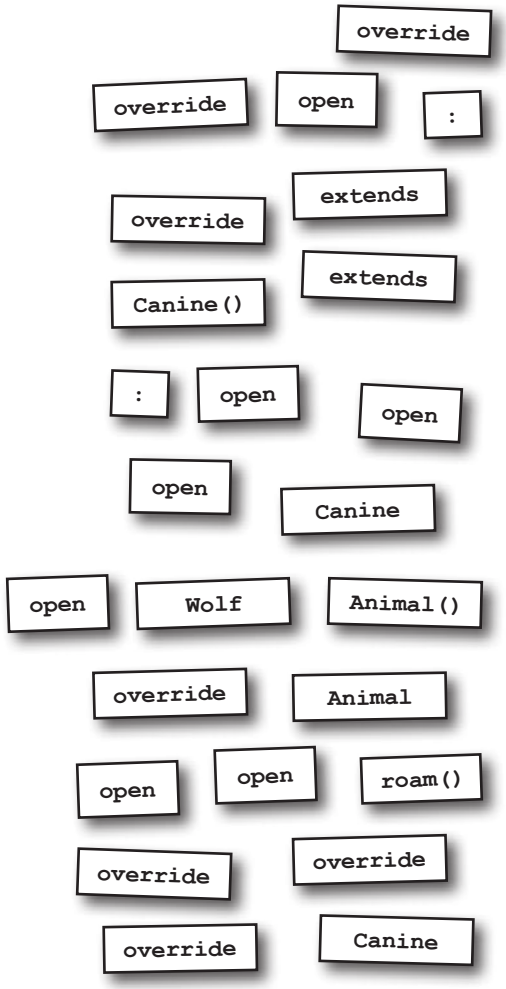
Попробуйте расставить магниты в правильных местах, чтобы создать классы `Canine` и `Wolf`.

Класс `Canine` является подклассом `Animal`: он переопределяет функцию `roam`.

Класс `Wolf` является подклассом `Canine`: он переопределяет свойства `image`, `food` и `habitat`, а также функции `makeNoise` и `eat` из класса `Animal`.

Некоторые магниты могут остаться неиспользованными.

```
_____ class Canine _____{

        _____fun _____ {

         println("The _____ is roaming")
    }
}

class Wolf _____{

    _____val image = "wolf.jpg"

    _____val food = "meat"

    _____val habitat = "forests"


    _____fun makeNoise() {

         println("Hooooowl!")
    }


    _____fun eat() {

         println("The Wolf is eating $food")
    }
}
```



Развлечения с магнитами. Решение

Попробуйте расставить магниты в правильных местах, чтобы создать классы `Canine` и `Wolf`.

Класс `Canine` является подклассом `Animal`: он переопределяет функцию `roam`.

Класс `Wolf` является подклассом `Canine`: он переопределяет свойства `image`, `food` и `habitat`, а также функции `makeNoise` и `eat` из класса `Animal`.

Некоторые магниты могут остаться неиспользованными.

```
open class Canine : Animal() {
    override fun roam() {
        println("The Canine is roaming")
    }
}

class Wolf : Canine() {
    override val image = "wolf.jpg"
    override val food = "meat"
    override val habitat = "forests"

    override fun makeNoise() {
        println("Hooooowl!")
    }

    override fun eat() {
        println("The Wolf is eating $food")
    }
}
```

Класс Canine является подклассом Animal. Он объявлен открытым, чтобы его можно было использовать в качестве суперкласса для класса Wolf.

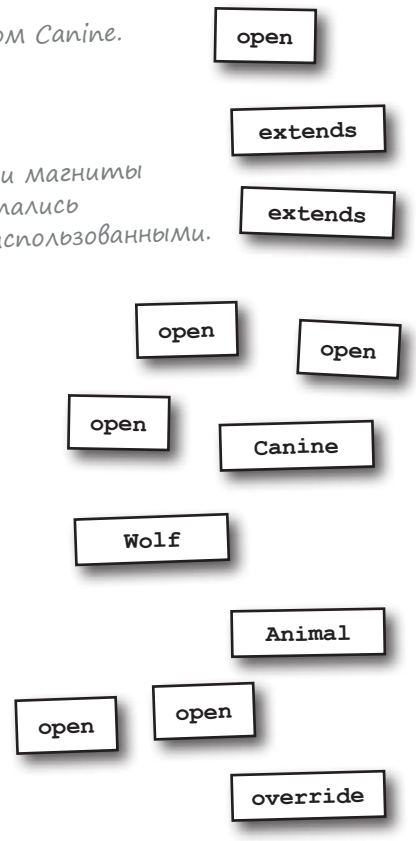
Переопределить функцию roam().

Wolf является подклассом Canine.

Эти свойства переопределяются.

Эти две функции переопределяются.

Эти магниты остались неиспользованными.



Добавление классов Canine и Wolf

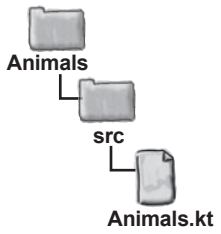
Классы `Canine` и `Wolf` готовы, добавим их в проект Animals. Обновите код *Animals.kt* и добавьте эти два класса (изменения выделены жирным шрифтом):

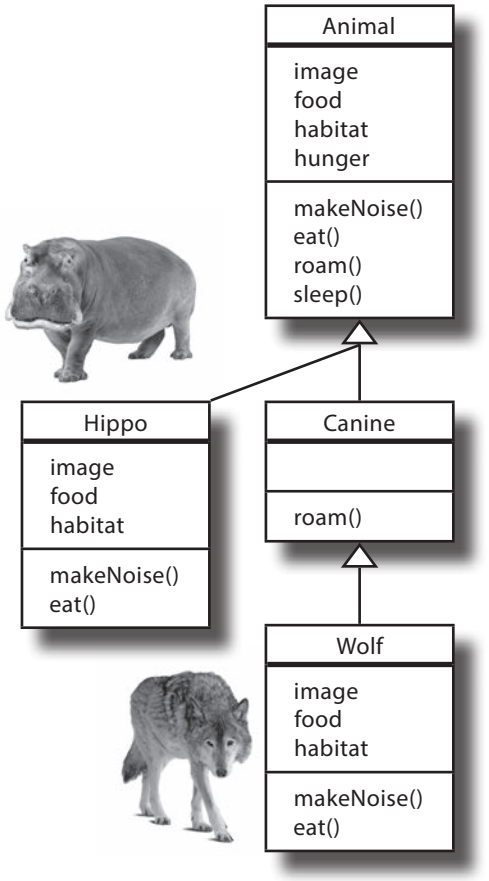
```
open class Animal {
    ...
}

class Hippo : Animal() {
    ...
}

open class Canine : Animal() {
    override fun roam() {
        println("The Canine is roaming")
    }
}

class Wolf : Canine() {
    override val image = "wolf.jpg"
    override val food = "meat"
    override val habitat = "forests"

    override fun makeNoise() {
        println("Hoooooowl!")
    }

    override fun eat() {
        println("The Wolf is eating $food")
    }
}
```

Код классов Animal и Hippo остался без изменений.

Добавляем класс Canine...

...а также класс Wolf.





Посмотрим, что происходит при создании объекта `Wolf` и вызове некоторых из его функций.

- [x] Проектирование классов
- [x] Построение классов
- [ ] **Использование классов**

Какая функция вызывается?

Класс `Wolf` содержит четыре функции: одна наследуется от `Animal`, другая наследуется от `Canine` (и является переопределенной версией функции из класса `Animal`), а еще две переопределяются в классе `Wolf`. Если создать объект `Wolf` и присвоить его переменной, вы сможете использовать оператор «точка» для вызова любой из этих четырех функций. Но какая версия этих функций при этом вызывается?

При вызове функции по ссылке на объект будет вызвана **самая конкретная версия функции для этого типа объекта**: то есть та, которая находится ниже всего в дереве наследования.

Скажем, при вызове функции для объекта `Wolf` система сначала ищет функцию в классе `Wolf`. Если функция будет найдена в этом классе, то она выполняется. Но если функция *не* определена в классе `Wolf`, система идет вверх по дереву наследования до класса `Canine`. Если функция определена в этом классе, система выполняет ее, а если нет, то поиск вверх по дереву продолжается. Система продолжает идти вверх по иерархии классов, пока не найдет совпадение для функции.

Чтобы увидеть этот механизм в действии, представьте, что вы решили создать новый объект `Wolf` и вызвать его функцию `makeNoise`. Система ищет функцию в классе `Wolf`, а поскольку функция была переопределена в этом классе, система выполняет эту версию:

```
val w = Wolf()
w.makeNoise()
```

Вызывает функцию makeNoise(), определенную в классе Wolf.

А что если вы решите вызвать функцию `roam` класса `Wolf`? Эта функция не переопределяется в классе `Wolf`, поэтому система ищет ее в классе `Canine`. Так как функция была переопределена в этом классе, система указывает эту версию.

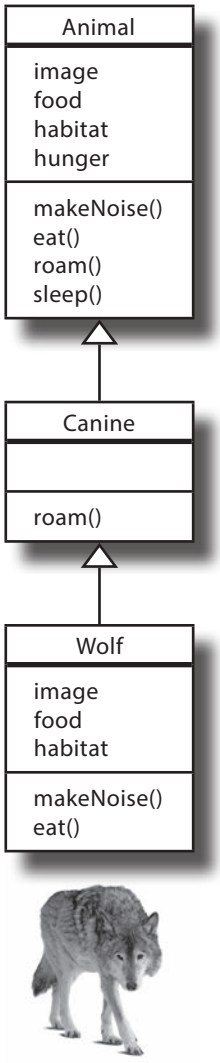
```
w.roam()
```

Вызывает функцию в классе Canine.

Наконец, допустим, что вы вызываете функцию `sleep` класса `Wolf`. Система ищет функцию в классе `Wolf`, а поскольку она здесь не переопределялась, система переходит вверх по дереву наследования к классу `Canine`. Функция не была переопределена и в этом классе, поэтому система использует версию из `Animal`.

```
w.sleep()
```

Вызывает функцию в классе Animal.



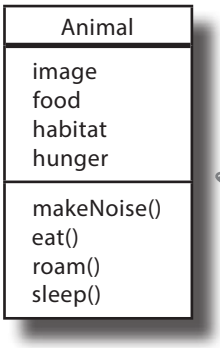
- [x] Проектирование классов
- [x] Построение классов
- [ ] Использование классов

Наследование обеспечивает наличие функций и свойств во всех подклассах, определенных в суперклассе

Если вы определяете набор свойств и функций в суперклассе, эти свойства и функции будут заведомо присутствовать во всех подклассах. Иначе говоря, вы определяете общий протокол, или контракт, для набора классов, связанных наследованием.

Класс `Animal`, например, устанавливает общий протокол для всех подтипов животных, который гласит: «любой объект *Animal* содержит свойства с именами *image*, *food*, *habitat* и *hunger*, а также функции *makeNoise*, *eat*, *roam* и *sleep*».

Говоря «любой объект Animal», мы имеем в виду «любой объект класса Animal и любого подкласса Animal».



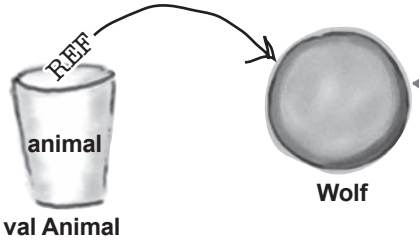
Вы сообщаете, что любой объект Animal содержит эти свойства и может выполнять эти операции.

В любом месте, где может использоваться суперкласс, может использоваться один из его подклассов

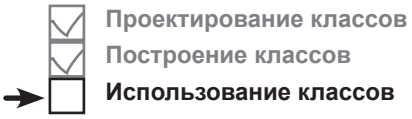
Когда вы определяете супертип для группы классов, **вы можете использовать любой подкласс вместо суперкласса, от которого он наследуется**. Это означает, что при объявлении переменной ей может быть присвоен объект любого подкласса типа переменной. Например, следующий код определяет переменную `Animal` и присваивает ей ссылку на объект `Wolf`. Компилятор знает, что `Wolf` является разновидностью `Animal`, поэтому следующий код компилируется:

```
val animal: Animal = Wolf()
```

Animal и Wolf — разные типы, но так как Wolf ЯВЛЯЕТСЯ Animal, код успешно компилируется.



Код создает объект Wolf и присваивает его переменной типа Animal.

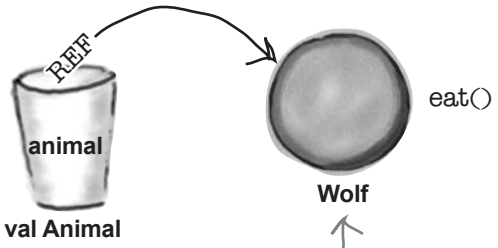


При вызове функции для переменной реагирует версия объекта

Как вам уже известно, при присваивании объекта переменной эта переменная может использоваться для вызова функций объекта. Это утверждение остается истинным и в том случае, если переменная относится к супертипу объекта.

Предположим, например, что объект `Wolf` присваивается переменной `Animal`, после чего вызывается его функция `eat`:

```
val animal: Animal = Wolf()
animal.eat()
```



При вызове функции `eat` будет вызвана версия класса `Wolf`, реагирующая на этот вызов. Система знает, что по ссылке хранится объект `Wolf`, поэтому реагирует на вызов так, как положено реагировать `Wolf`.

Если переменной Animal присваивается объект Wolf, то при использовании ее для вызова eat будет вызвана функция eat() класса Wolf.

Вы также можете создать массив разных типов животных и заставить каждого из них реагировать так, как должен реагировать этот конкретный объект. А поскольку каждое животное является подклассом `Animal`, то их всех можно добавить в массив и вызвать функции для каждого элемента в массиве:

```
val animals = arrayOf(Hippo(),
                      Wolf(),
                      Lion(),
                      Cheetah(),
                      Lynx(),
                      Fox())
```

Компилятор видит, что все перечисленные типы являются разновидностями Animal, поэтому создает массив типа Array<Animal>.

```
for (item in animals) {
    item.roam()
    item.eat()
}
```

Этот цикл перебирает животных и вызывает для каждого объекта roam() и eat(). Каждый объект реагирует способом, соответствующим его типу.

Таким образом, проектирование на основе наследования означает, что вы можете писать гибкий код с полной уверенностью в том, что каждый объект будет реагировать нужным образом при вызове его функций.

Впрочем, это еще не все.

Супертип может использоваться для параметров и возвращаемого типа функции

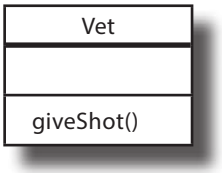
Как вы думаете, что произойдет, если объявить переменную с супертипом (допустим, `Animal`) и присвоить ей объект подкласса (допустим, `Wolf`) при использовании подкласса в качестве аргумента функции?

Предположим, вы создали класс `Vet` с функцией `giveShot`:

Функция giveShot класса Vet получает параметр Animal.

```
class Vet {
    fun giveShot(animal: Animal) {
        // Код медицинской процедуры, которая не понравится животному
        animal.makeNoise()
    }
}
```

giveShot вызывает функцию makeNoise для Animal.



Параметр `Animal` может получать из аргумента любую разновидность `Animal`. Таким образом, при вызове функции `giveShot` класса `Vet` будет выполнена функция `makeNoise` для `Animal`, и на этот вызов отреагирует конкретная разновидность `Animal`:

```
val vet = Vet()
val wolf = Wolf()
val hippo = Hippo()
vet.giveShot(wolf)
vet.giveShot(hippo)
```

Wolf и Hippo являются разновидностями Animal, поэтому мы можем передать объекты Wolf и Hippo в аргументах функции giveShot.

Таким образом, если вы хотите, чтобы другие типы животных работали с классом `Vet`, для этого достаточно проследить за тем, чтобы каждый из них был подклассом класса `Animal`. Функция `giveShot` класса `Vet` будет нормально работать, хотя при ее написании мы ничего не знали о новых подклассах Animal.

Возможность использования объектов одного типа в месте, в котором явно обозначен другой тип, называется **полиморфизмом**. По сути, речь идет о возможности предоставлять разные реализации функций, которые были унаследованы от другого класса.

Полный код проекта Animal приведен на следующей странице.

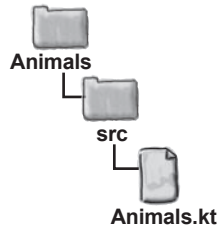
Полиморфизм означает «много форм». Этот механизм позволяет разным подклассам переопределять разные реализации одной функции.

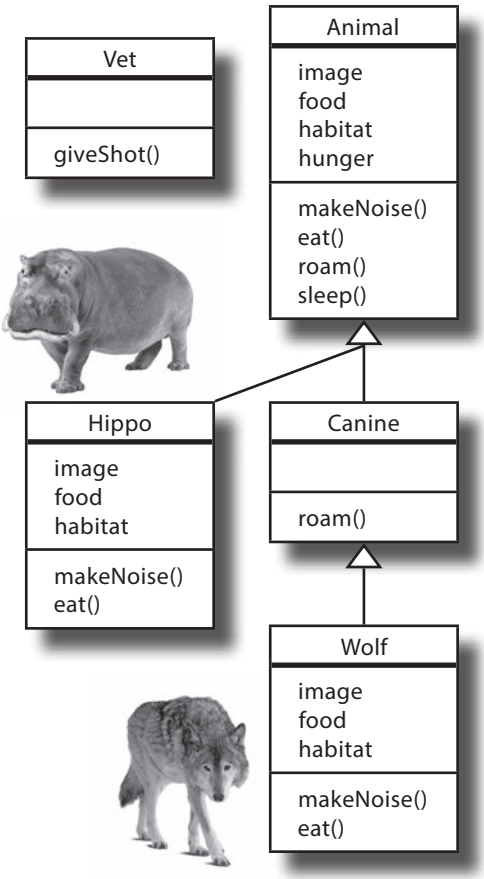
Обновленный код Animals

Обновленная версия *Animals.kt* включает класс `Vet` и функцию `main`. Обновите свою версию кода и приведите ее в соответствие с нашей (изменения выделены жирным шрифтом):

```
open class Animal {
    open val image = ""
    open val food = ""
    open val habitat = ""
    var hunger = 10

    open fun makeNoise() {
        println("The Animal is making a noise")
    }

    open fun eat() {
        println("The Animal is eating")
    }

    open fun roam() {
        println("The Animal is roaming")
    }

    fun sleep() {
        println("The Animal is sleeping")
    }
}

class Hippo: Animal() {
    override val image = "hippo.jpg"
    override val food = "grass"
    override val habitat = "water"

    override fun makeNoise() {
        println("Grunt! Grunt!")
    }

    override fun eat() {
        println("The Hippo is eating $food")
    }
}

open class Canine: Animal() {
    override fun roam() {
        println("The Canine is roaming")
    }
}
```

Код на этой странице остался без изменений.

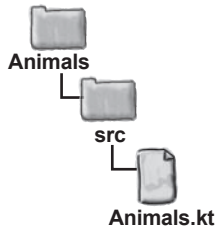




Продолжение кода на следующей странице.

Код (продолжение)...

```
class Wolf: Canine() {
    override val image = "wolf.jpg"
    override val food = "meat"
    override val habitat = "forests"

    override fun makeNoise() {
        println("Hooooowl!")
    }

    override fun eat() {
        println("The Wolf is eating $food")
    }
}
```



Добавлен класс Vet.

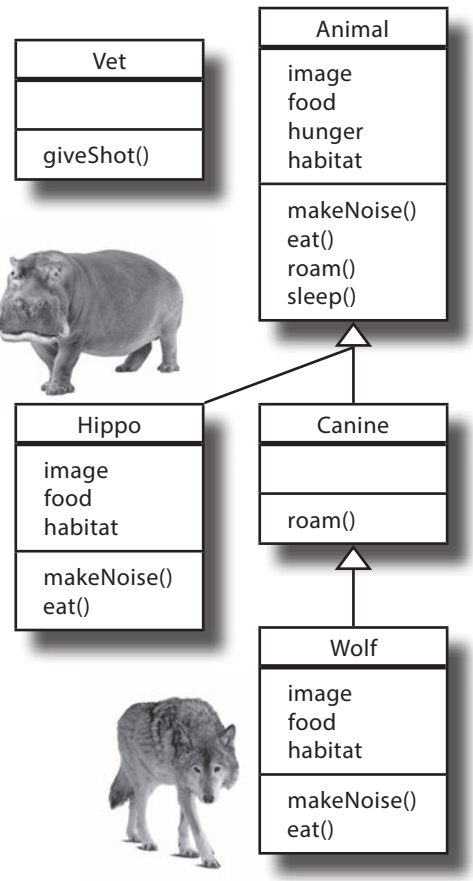
```
class Vet {
    fun giveShot(animal: Animal) {
        //Code to do something medical
        animal.makeNoise()
    }
}
```

Добавлена функция main.

```
fun main(args: Array<String>) {
    val animals = arrayOf(Hippo(), Wolf())
    for (item in animals) {
        item.roam()
        item.eat()
    }

    val vet = Vet()
    val wolf = Wolf()
    val hippo = Hippo()
    vet.giveShot(wolf)
    vet.giveShot(hippo)
}
```

В цикле перебирается массив с элементами Animal.

Вызывается функция giveShot класса Vet, которой передаются два подкласса Animal.



Тест-драйв

При выполнении кода в окне вывода IDE появляется следующий текст:

```
The Animal is roaming
The Hippo is eating grass
The Canine is roaming
The Wolf is eating meat
Hooooowl!
Grunt! Grunt!
```

Hippo наследует функцию roam от Animal.

Wolf наследует функцию roam от Canine.

Каждый объект Animal реагирует на вызов функции giveShot класса Vet по своим правилам.

Часто Задаваемые Вопросы

В: Почему Kotlin разрешает мне переопределить свойство `val` с `var`?

О: Как упоминалось в главе 4, при создании свойства `val` компилятор незаметно добавляет для него get-метод. А при создании свойства `var` компилятор добавляет как get-, так и set-метод.

При переопределении свойства `val` с `var` вы фактически приказываете компилятору добавить дополнительный set-метод для свойства в подклассе. Это не запрещено, поэтому код компилируется.

В: Могу ли я переопределить свойство `var` с `val`?

О: Нет. При попытке переопределить свойство `var` с ключевым словом `val` код не компилируется.

При определении иерархии класса подкласс гарантированно может делать все то, что может делать его суперкласс. А при попытке переопределения свойства `var` с ключевым словом `val` вы сообщаете компилятору, что значение переменной не должно изменяться. Тем самым нарушается общий протокол между суперклассом и его подклассами, поэтому такой код не компилируется.

В: Вы сказали, что при вызове функции для переменной система обходит иерархию наследования в поисках совпадения. Что произойдет, если система не найдет нужную функцию?

О: Не беспокойтесь, этого не случится. Компилятор гарантирует вызов указанной функции для конкретного типа переменной, где бы она ни была найдена во время выполнения. Например, если вы вызываете функцию `sleep` для класса `Wolf`, то компилятор проверяет, что функция `sleep` существует, но не обращает внимания на то, что эта функция определяется в классе `Animal` (и наследуется от него).

Запомните: если класс *наследует* функцию, то он *содержит* эту функцию. Для компилятора неважно, где определяется унаследованная функция. Во время выполнения система всегда выбирает правильную, самую конкретную версию функции для указанного объекта.

В: Возможно ли наследование от нескольких непосредственных суперклассов?

О: Нет. Множественное наследование запрещено в Kotlin, поэтому каждый подкласс может иметь только один непосредственный суперкласс. Эта тема более подробно рассматривается в главе 6.

В: При переопределении функции в подклассе типы параметров функции должны совпадать. Могу ли я определить функцию с таким же именем, как в суперклассе, но с другими типами параметров?

О: Да, можете. Допускается определение нескольких функций с одинаковыми именами, если они отличаются по типу параметров. Это называется *перегрузкой* (не переопределением!) и не имеет никакого отношения к наследованию.

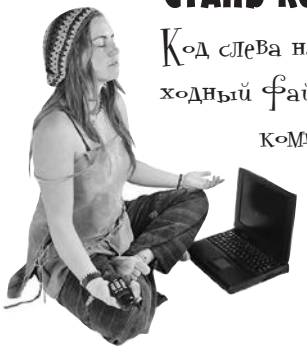
Перегрузка функций рассматривается в главе 7.

В: Можете еще раз объяснить, что такое полиморфизм?

О: Конечно. Полиморфизм — это возможность использования объекта любого подкласса вместо его суперкласса. Так как разные подклассы могут содержать разные реализации одной функции, это позволяет каждому объекту реагировать на вызов функции способом, наиболее подходящим для каждого объекта.

Возможности практического применения полиморфизма будут описаны в следующей главе.

СТАНЬ компилятором



Код слева на этой странице представляет полный исходный файл. Попробуйте представить себя на месте компилятора и определить, какая из пар функций A–B в правой части откомпилируется и выдаст нужный результат при подстановке в код слева. Функция A подставляется в класс Monster, а функция B в класс Vampire.

Вывод:

Код должен выводить следующий результат:

```
Fancy a bite?
Fire!
Aargh!
```

Основной код.

```
open class Monster {

    A

}

class Vampire : Monster() {

    B

}

class Dragon : Monster() {
    override fun frighten(): Boolean {
        println("Fire!")
        return true
    }
}

fun main(args: Array<String>) {
    val m = arrayOf(Vampire(),
                    Dragon(),
                    Monster())
    for (item in m) {
        item.frighten()
    }
}
```

Пары функций:

1A
```
open fun frighten(): Boolean {
    println("Aargh!")
    return true
}
```

1B
```
override fun frighten(): Boolean {
    println("Fancy a bite?")
    return false
}
```

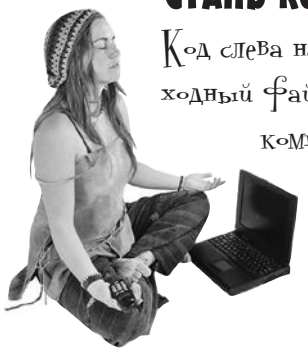
2A
```
fun frighten(): Boolean {
    println("Aargh!")
    return true
}
```

2B
```
override fun frighten(): Boolean {
    println("Fancy a bite?")
    return true
}
```

3A
```
open fun frighten(): Boolean {
    println("Aargh!")
    return false
}
```

3B
```
fun beScary(): Boolean {
    println("Fancy a bite?")
    return true
}
```

СТАНЬ компилятором. Решение



Код слева на этой странице представляет полный исходный файл. Попробуйте представить себя на месте компилятора и определить, какая из пар функций A–B в правой части откомпилируется и выдаст нужный результат при подстановке в код слева. Функция A подставляется в класс Monster, а функция B в класс Vampire.

Результат:

```
Fancy a bite?
Fire!
Aargh!
```

```
open class Monster {

    A

}

class Vampire : Monster() {

    B

}

class Dragon : Monster() {
    override fun frighten(): Boolean {
        println("Fire!")
        return true
    }
}

fun main(args: Array<String>) {
    val m = arrayOf(Vampire(),
                    Dragon(),
                    Monster())
    for (item in m) {
        item.frighten()
    }
}
```

1A
```
open fun frighten(): Boolean {
    println("Aargh!")
    return true
}
```
Этот код компилируется и выдает правильный результат.

1B
```
override fun frighten(): Boolean {
    println("Fancy a bite?")
    return false
}
```

2A
```
fun frighten(): Boolean {
    println("Aargh!")
    return true
}
```
Этот код не компилируется, потому что функция frighten() в классе Monster не объявлена открытой.

2B
```
override fun frighten(): Boolean {
    println("Fancy a bite?")
    return true
}
```

3A
```
open fun frighten(): Boolean {
    println("Aargh!")
    return false
}
```
Компилируется, но выдает неправильный результат, потому что Vampire не переопределяет frighten().

3B
```
fun beScary(): Boolean {
    println("Fancy a bite?")
    return true
}
```

Ваш инструментарий Kotlin

Глава 5 осталась позади, ваш инструментарий пополнился суперклассами и подклассами.

Весь код для этой главы можно загрузить по адресу https://tinyurl.com/HFKotlin.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Суперкласс содержит общие свойства и функции, наследуемые одним или несколькими подклассами.
- Подкласс может включать дополнительные свойства и функции, не входящие в суперкласс, а также переопределять унаследованные аспекты.
- «Правило ЯВЛЯЕТСЯ» проверяет правильность структуры наследования. Если `X` является *подклассом* `Y`, то утверждение «`X` *ЯВЛЯЕТСЯ* `Y`» должно иметь смысл.
- Отношения ЯВЛЯЕТСЯ работают в одном направлении. `Hippo` является `Animal`, но не все объекты `Animal` являются `Hippo`.
- Если класс `B` является подклассом класса `A`, а класс `C` является подклассом класса `B`, то класс `C` проходит «правило ЯВЛЯЕТСЯ» как для `B`, так и для `A`.
- Чтобы использовать класс в качестве суперкласса, необходимо объявить его открытым (`open`). Все свойства и функции, которые вы хотите переопределить, также должны быть объявлены открытыми.
- Используйте **:** для назначения суперкласса в объявлении подкласса.
- Если у суперкласса имеется первичный конструктор, он должен вызываться в заголовке подкласса.
- Чтобы переопределить свойства и функции в подклассе, снабдите их префиксом `override`. Когда вы переопределяете свойство, его тип должен быть совместим со свойством суперкласса, а тип возвращаемого значения должен быть совместим с возвращаемым типом суперкласса.
- Переопределенные функции и свойства остаются открытыми до тех пор, пока не будут объявлены `final`.
- Если вы переопределяете функцию в подклассе, а эта функция вызывается для экземпляра подкласса, будет вызвана переопределенная версия функции.
- Наследование гарантирует, что все подклассы содержат функции и свойства, определенные в суперклассе.
- Подкласс может использоваться в любом месте, в котором предполагается использование типа суперкласса.
- Полиморфизм означает «много форм». Благодаря этому механизму разные подклассы могут содержать разные реализации одной функции.

Серьезно о полиморфизме

Иерархия наследования суперклассов — только первый шаг.

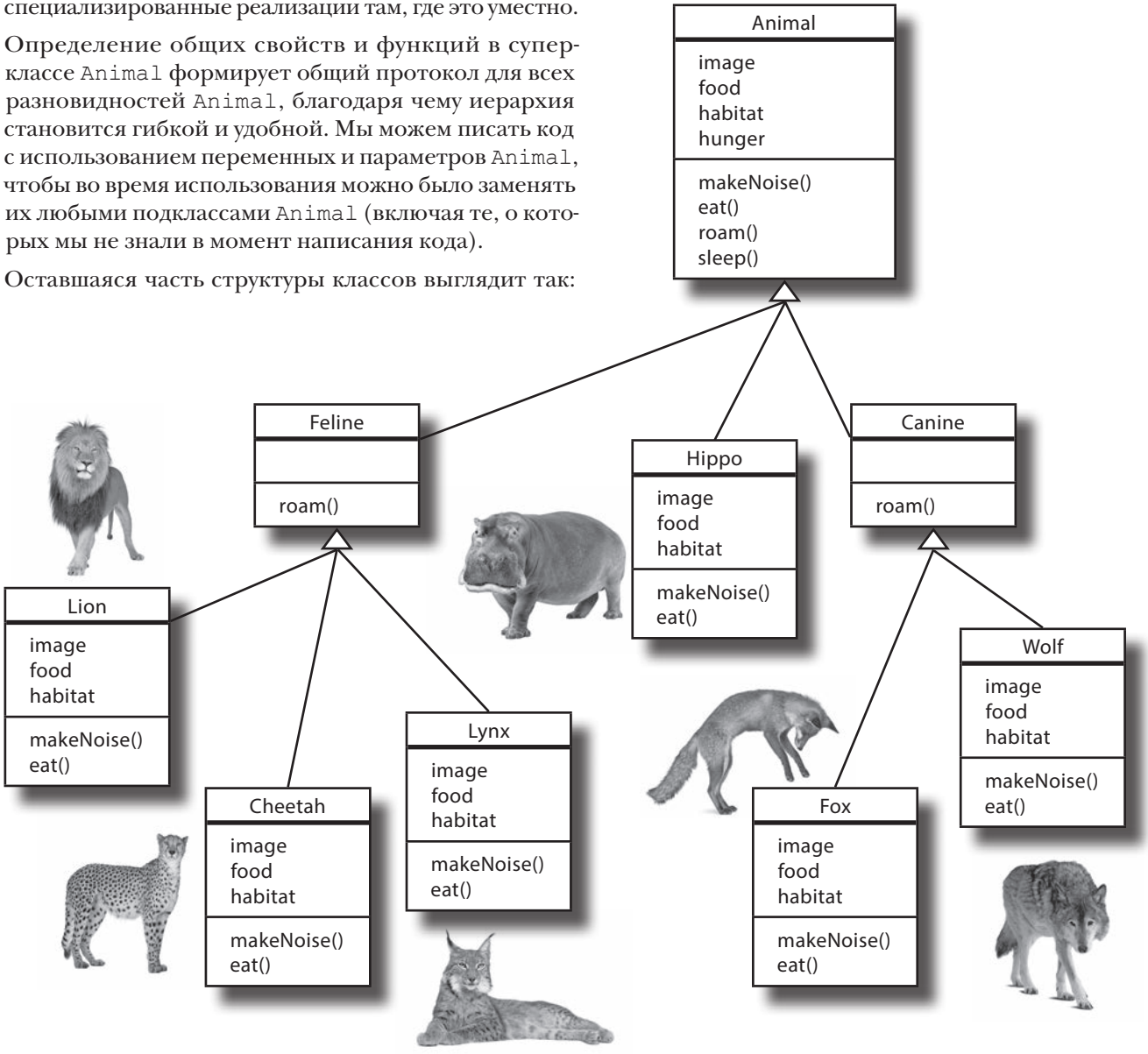
Чтобы ***в полной мере использовать возможности полиморфизма***, следует проектировать иерархии с **абстрактными классами** и **интерфейсами**. В этой главе вы узнаете, как при помощи абстрактных классов управлять тем, какие классы ***могут или не могут создаваться в вашей иерархии***. Вы увидите, как с их помощью заставить конкретные подклассы ***предоставлять собственные реализации***. В этой главе мы покажем, как при помощи интерфейсов организовать ***совместное использование поведения в независимых классах,*** а заодно опишем нюансы операторов ***is***, ***as и when***.

Снова об иерархии классов Animal

В предыдущей главе вы узнали, как проектируются иерархии классов, на примере структуры классов, представляющих разных животных. Общие свойства и функции были выделены в суперкласс `Animal`, а некоторые свойства и функции переопределялись в подклассах `Animal`, чтобы мы могли определить специализированные реализации там, где это уместно.

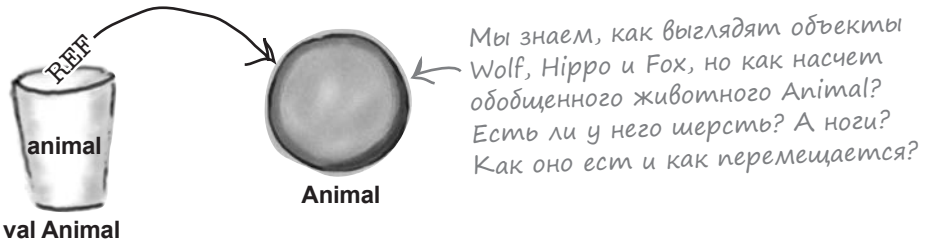
Определение общих свойств и функций в суперклассе `Animal` формирует общий протокол для всех разновидностей `Animal`, благодаря чему иерархия становится гибкой и удобной. Мы можем писать код с использованием переменных и параметров `Animal`, чтобы во время использования можно было заменять их любыми подклассами `Animal` (включая те, о которых мы не знали в момент написания кода).

Оставшаяся часть структуры классов выглядит так:



Некоторые классы не подходят для создания экземпляров

Тем не менее структура классов нуждается в усовершенствовании. Создание объектов `Wolf`, `Hippo` или `Fox` выглядит разумно, но иерархия наследования также позволяет создавать обобщенные объекты `Animal`, а это нежелательно, потому что мы не можем сказать, как выглядит такое обобщенное животное, что оно ест, какие звуки издает и т. д.



Что же делать? Класс `Animal` необходим для наследования и полиморфизма, однако создаваться должны только экземпляры менее абстрактных подклассов `Animal`, а не самого класса `Animal`. Наша иерархия должна допускать создание объектов `Hippo`, `Wolf` и `Fox`, но не объектов `Animal`.

Объявление абстрактного класса для предотвращения создания его экземпляров

Если вы хотите запретить создание экземпляров класса, пометьте класс как **абстрактный** при помощи префикса `abstract`. Например, вот как класс `Animal` преобразуется в абстрактный класс:

```
abstract class Animal {
    ...
}
```

Чтобы объявить класс абстрактным, снабдите его префиксом «abstract».

Если класс объявлен абстрактным, это означает, что никто не сможет создавать никакие объекты этого класса, даже если для него определен конструктор. Абстрактный класс может использоваться как тип при объявлении переменных, но вам не нужно беспокоиться о том, что кто-то создаст объект этого типа — компилятор проследит за тем, чтобы этого не произошло:

```
var animal: Animal
animal = Wolf()
animal = Animal()
```

Эта строка не компилируется, потому что создание объектов Animal запрещено.

Если суперкласс помечен как абстрактный, объявлять его открытым не обязательно.

Подумайте об иерархии классов `Animal`. Как вы думаете, какие классы должны объявляться абстрактными? Иначе говоря, у каких классов не должны создаваться экземпляры?

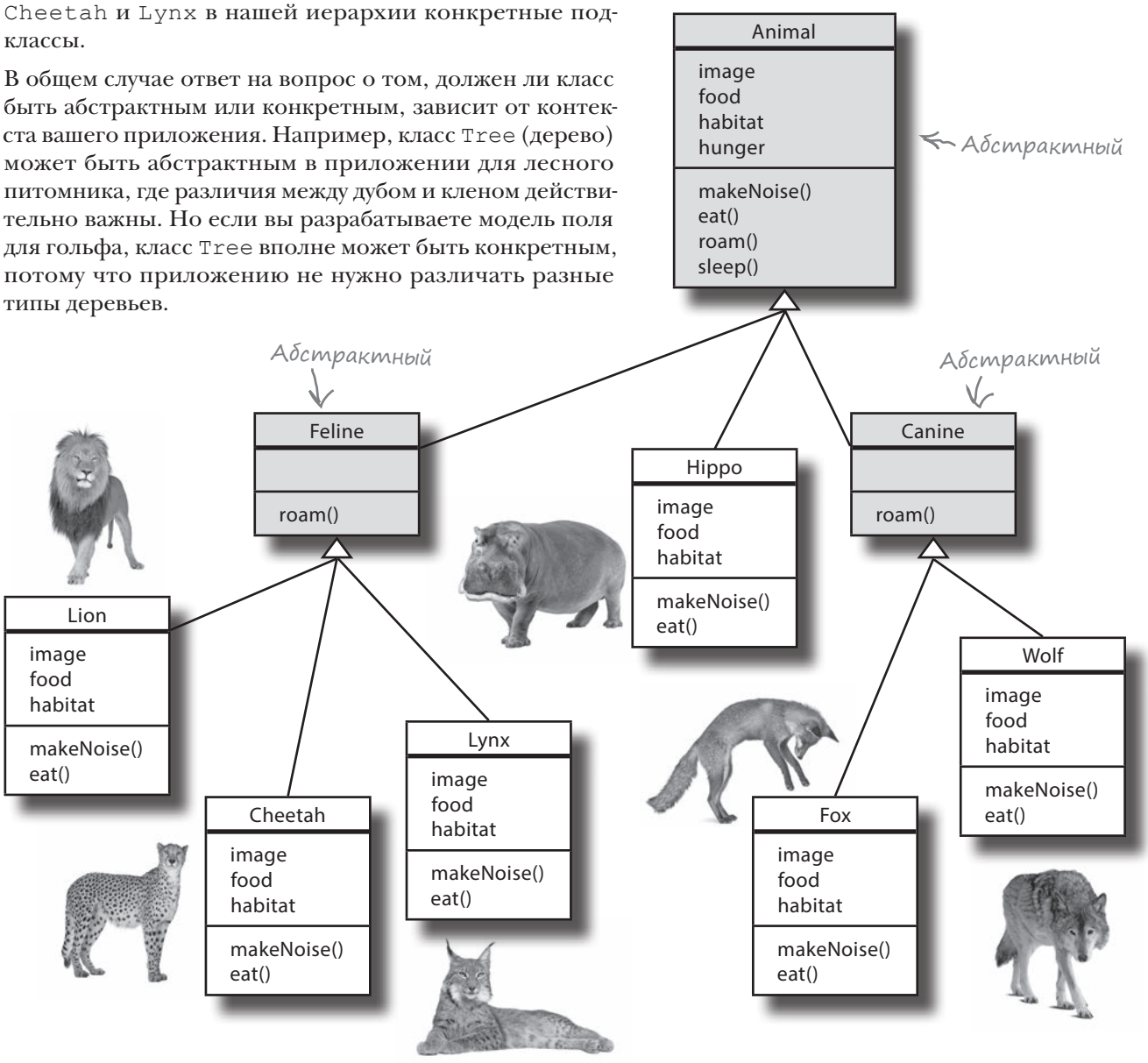
Абстрактный или конкретный?

В нашей иерархии классов `Animal` существуют три класса, которые должны быть объявлены абстрактными: `Animal`, `Canine` и `Feline`. Хотя эти классы нужны нам для наследования, никто не должен создавать объекты этих типов.

Класс, который не является абстрактным, называется **конкретным**; таким образом, `Hippo`, `Wolf`, `Fox`, `Lion`, `Cheetah` и `Lynx` в нашей иерархии конкретные подклассы.

В общем случае ответ на вопрос о том, должен ли класс быть абстрактным или конкретным, зависит от контекста вашего приложения. Например, класс `Tree` (дерево) может быть абстрактным в приложении для лесного питомника, где различия между дубом и кленом действительно важны. Но если вы разрабатываете модель поля для гольфа, класс `Tree` вполне может быть конкретным, потому что приложению не нужно различать разные типы деревьев.

Абстрактные классы Animal, Canine и Feline выделены серым фоном.



Абстрактный класс может содержать абстрактные свойства и функции

В абстрактном классе свойства и функции могут быть помечены как абстрактные. Это может быть полезно, если часть поведения класса не имеет смысла без реализации в более конкретном подклассе, и при этом не существует никакой обобщенной реализации, которая могла бы с пользой наследоваться подклассами.

Посмотрим, какие свойства и функции стоит объявить абстрактными в классе `Animal`.

Абстрактный класс может содержать абстрактные и неабстрактные свойства и функции. Абстрактный класс также может не содержать ни одного абстрактного компонента.

Три свойства можно пометить как абстрактные

При создании класса `Animal` мы решили заполнить свойства `image`, `food` и `habitat` обобщенными значениями и переопределить их в подклассах конкретных животных. Это было связано с тем, что этим свойствам невозможно было присвоить значение, которое бы могло быть полезным для подклассов.

Поскольку эти свойства содержат обобщенные значения, которые все равно должны переопределяться, мы можем пометить их как абстрактные, для этого перед свойством ставится префикс `abstract`. Соответствующий код выглядит так:

```
abstract class Animal {
    abstract val image: String
    abstract val food: String
    abstract val habitat: String
    var hunger = 10
    ...
}
```

Свойства image, food и habitat помечены как абстрактные.

Абстрактные свойства и функции не обязательно помечать как открытые.

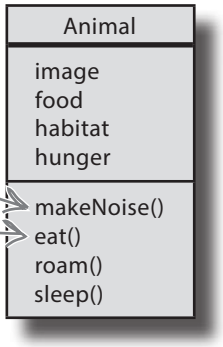
Обратите внимание, что в приведенном выше коде мы не инициализировали ни одно из абстрактных свойств. Если вы попытаетесь инициализировать абстрактное свойство или определите для него пользовательский get- или set-метод, компилятор откажется компилировать код. Пометив свойство как абстрактное, вы обозначили то, что у него нет полезного начального значения и полезной реализации для пользовательского get- или set-метода.

Теперь, когда мы знаем, какие свойства можем пометить как абстрактные, рассмотрим функции.

Класс Animal содержит две абстрактные функции

Класс `Animal` определяет две функции — `makeNoise` и `eat`, — которые переопределяются во всех конкретных подклассах. Так как эти две функции всегда переопределяются и не существует обобщенной реализации, которая могла бы быть полезной для подклассов, функции `makeNoise` и `eat` можно пометить как абстрактные при помощи префикса `abstract`. Код выглядит так:

```
abstract class Animal {
    ...
    abstract fun makeNoise()

    abstract fun eat()

    open fun roam() {
        println("The Animal is roaming")
    }

    fun sleep() {
        println("The Animal is sleeping")
    }
}
```



В этом коде ни одна абстрактная функция не имеет тела. Дело в том, что, помечая функцию как абстрактную, вы сообщаете компилятору, что не можете написать полезный код для тела функции.

Если вы попытаетесь добавить тело в абстрактную функцию, компилятор откажется компилировать ваш код. Например, следующий код не компилируется из-за фигурных скобок за определением функции:

```
abstract fun makeNoise() {}
```

Фигурные скобки образуют пустое тело функции, поэтому код не будет компилироваться.

Чтобы код компилировался, необходимо удалить фигурные скобки. Измененный код выглядит так:

```
abstract fun makeNoise()
```

Так как на этот раз абстрактная функция не содержит тела, код успешно компилируется.

Будьте осторожны!

Если свойство или функция помечены как абстрактные, класс тоже должен быть помечен как абстрактный.

Если класс содержит хотя бы одно абстрактное свойство или функцию, этот класс тоже должен быть помечен как абстрактный, в противном случае код не будет компилироваться.

Не понимаю. Если в абстрактную функцию нельзя добавить код, для чего она нужна? Я думала, что абстрактные классы создаются для хранения общего кода, который может наследоваться подклассами.

Абстрактные свойства и функции определяют общий протокол для применения полиморфизма.

Наследуемые реализации функций (функции с реальным телом) удобно размещать в суперклассах *тогда, когда это имеет смысл*. А в абстрактном классе это часто *не имеет смысла*, потому что вы не можете написать обобщенный код, который может быть полезен для подклассов.

Абстрактные функции полезны. Хотя они и не содержат реального кода функций, они определяют протокол для группы подклассов, которые могут использоваться в полиморфном коде. Как вы узнали в предыдущей главе, термин «полиморфизм» означает, что при определении супертипа для группы классов можно использовать любой подкласс вместо суперкласса, от которого он наследуется. Это позволяет использовать тип суперкласса как тип переменной, аргумент функции, возвращаемый тип или тип элемента массива, как в следующем примере:

```
val animals = arrayOf(Hippo(),
                      Wolf(),
                      Lion(),
                      Cheetah(),
                      Lynx(),
                      Fox())
```

Массив с разными объектами Animal.

```
for (item in animals) {
    item.roam()
    item.eat()
}
```

Каждый объект Animal в массиве реагирует на вызов по-своему.

А это означает, что вы можете добавлять новые подтипы (например, новые подклассы `Animal`) в свое приложение и вам не придется переписывать или добавлять новые функции для работы с этими новыми типами.

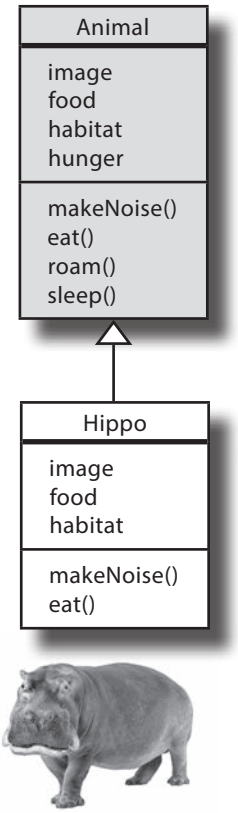
Итак, вы узнали, как (и когда) классы, свойства и функции помечаются как абстрактные. Давайте посмотрим, как они реализуются.

Как реализовать абстрактный класс

Объявление класса, наследующего от абстрактного суперкласса, практически не отличается от объявления класса, наследующего от обычного суперкласса: поставьте двоеточие в заголовке класса и укажите имя абстрактного класса. Например, следующее объявление означает, что класс `Hippo` наследуется от абстрактного класса `Animal`:

```
class Hippo : Animal() {
    ...
}
```

Как и при наследовании от обычного суперкласса, необходимо вызвать конструктор абстрактного класса в заголовке подкласса.



Чтобы реализовать абстрактные свойства и функции, переопределите их и предоставьте реализации. Это означает, что все абстрактные свойства необходимо инициализировать, а для любых абстрактных функций необходимо предоставить тело.

В нашем примере класс `Hippo` является конкретным подклассом `Animal`. Следующий код класса `Hippo` реализует свойства `image`, `food` и `habitat`, а также функции `makeNoise` и `eat`:

```
class Hippo : Animal() {
    override val image = "hippo.jpg"
    override val food = "grass"
    override val habitat = "water"

    override fun makeNoise() {
        println("Grunt! Grunt!")
    }

    override fun eat() {
        println("The Hippo is eating $food")
    }
}
```

Чтобы реализовать абстрактные свойства и функции, вы переопределяете их. Здесь все происходит так же, как и для конкретных суперклассов.

При реализации абстрактных свойств и функций необходимо следовать тем же правилам переопределения, которые используются при переопределении обычных свойств и функций:

- Реализация абстрактного *свойства* должна иметь то же имя, а его тип должен быть совместим с типом, определенным в абстрактном суперклассе. Иначе говоря, это должен быть тот же класс или один из его подклассов.

- Реализация абстрактной *функции* должна иметь такую же сигнатуру (имя и аргументы), как и функция, определенная в абстрактном суперклассе. Возвращаемый тип должен быть совместим с объявленным возвращаемым типом.

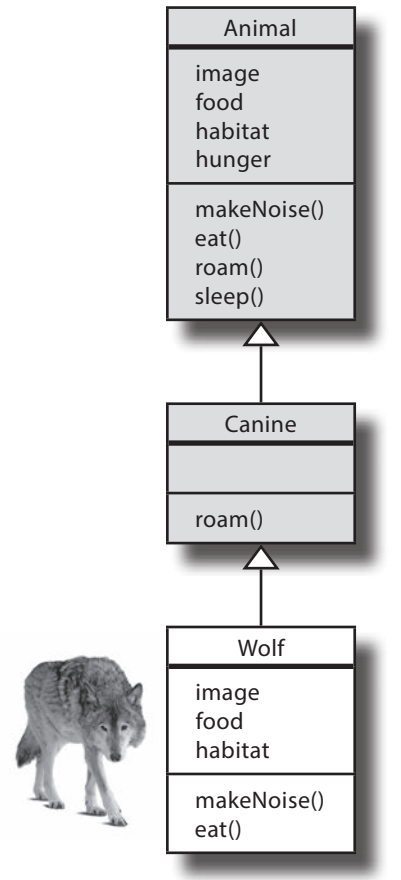
Вы ДОЛЖНЫ реализовать все абстрактные свойства и функции

Первый **конкретный** класс, расположенный в дереве наследования ниже абстрактного суперкласса, *должен* реализовать все абстрактные свойства и функции. Например, в нашей иерархии класс `Hippo` является непосредственным конкретным подклассом `Animal`, поэтому он должен реализовать все абстрактные свойства и функции, определенные в классе `Animal`, в противном случае код не откомпилируется.

В **абстрактных** подклассах у вас есть выбор: либо реализовать абстрактные свойства и функции, либо передать эстафету их подклассам. Если оба класса, `Animal` и `Canine`, являются абстрактными, класс `Canine` может либо реализовать абстрактные свойства и функции от `Animal`, либо ничего не сказать о них и оставить их реализацию своим подклассам.

Любые абстрактные свойства и функции, не реализованные в классе `Canine`, должны быть реализованы в его конкретных подклассах (таких, как `Wolf`). А если класс `Canine` определит новые абстрактные свойства и функции, то подклассы `Canine` должны будут реализовать их тоже.

После знакомства с абстрактными классами, свойствами и функциями обновим код нашей иерархии `Animal`.



Подкласс, наследующий от абстрактного суперкласса, тоже может определять новые функции и свойства.

Часто Задаваемые Вопросы

В: Почему первый конкретный класс должен реализовать все унаследованные абстрактные свойства и функции?

О: Каждое свойство и каждая функция в конкретном классе должны быть реализованы, чтобы компилятор знал, что делать при обращении к ним.

Только абстрактные классы могут содержать абстрактные свойства и функции. Если класс содержит какие-либо свойства и функции, помеченные как абстрактные, весь класс должен быть абстрактным.

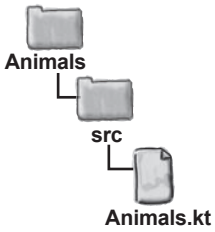
В: Я хочу определить пользовательские get- и set-методы для абстрактного свойства. Почему у меня не получается?

О: Помечая свойство как абстрактное, вы тем самым сообщаете компилятору, что свойство не имеет реализации, которая могла бы быть полезной для его подклассов. Если компилятор видит, что абстрактное свойство содержит некое подобие реализации (например, пользовательский get- или set-метод или исходное значение), он не понимает, что происходит, и отказывается компилировать код.

Внесем изменения в код проекта Animals

В предыдущей главе мы написали код классов `Animal`, `Canine`, `Hippo`, `Wolf` и `Vet` и добавили их в проект Animals. Теперь этот код необходимо обновить, чтобы классы `Animal` и `Canine` стали абстрактными. Свойства `image`, `food` и `habitat` в классе `Animal` тоже станут абстрактными, как и функции `makeNoise` и `eat`.

Откройте проект Animals, созданный в предыдущей главе, и обновите свою версию кода в файле *Animals.kt*, чтобы она соответствовала нашей (изменения выделены жирным шрифтом):

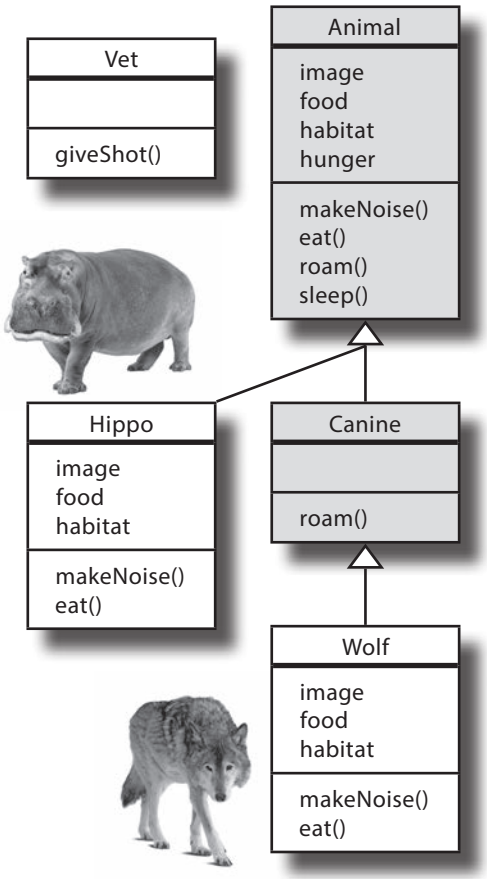


Класс Animal помечается как абстрактный, а не как открытый.

Эти свойства помечаются как абстрактные...

...и эти две функции тоже.

```
abstract open class Animal {
    abstract open val image: String
    abstract open val food: String
    abstract open val habitat: String
    var hunger = 10

    abstract open fun makeNoise() {
        println("The Animal is making a noise")
    }

    abstract open fun eat() {
        println("The Animal is eating")
    }

    open fun roam() {
        println("The Animal is roaming")
    }

    fun sleep() {
        println("The Animal is sleeping")
    }
}
```



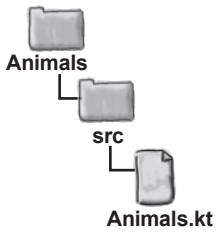
Продолжение на следующей странице.

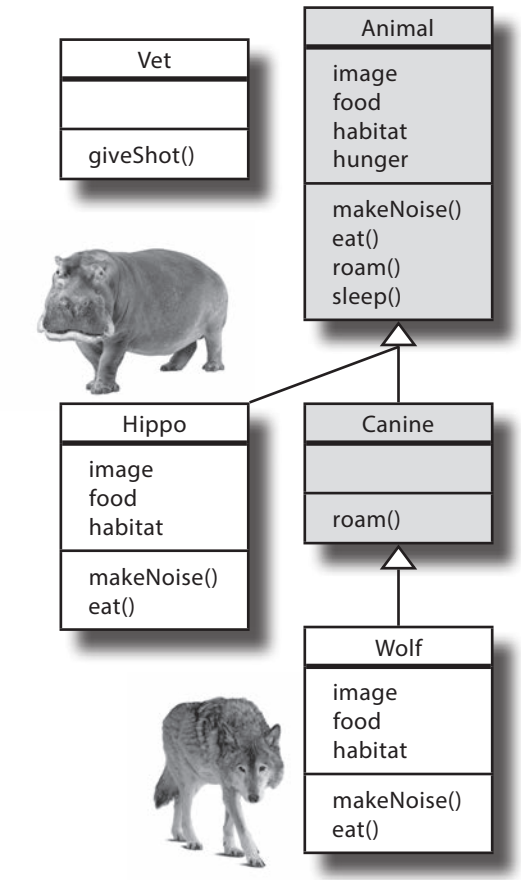
Продолжение кода...

```
class Hippo : Animal() {
    override val image = "hippo.jpg"
    override val food = "grass"
    override val habitat = "water"

    override fun makeNoise() {
        println("Grunt! Grunt!")
    }

    override fun eat() {
        println("The Hippo is eating $food")
    }
}
```

Класс Canine помечается как абстрактный.

```
abstract ~~open~~ class Canine : Animal() {
    override fun roam() {
        println("The Canine is roaming")
    }
}

class Wolf : Canine() {
    override val image = "wolf.jpg"
    override val food = "meat"
    override val habitat = "forests"

    override fun makeNoise() {
        println("Hooooowl!")
    }

    override fun eat() {
        println("The Wolf is eating $food")
    }
}
```



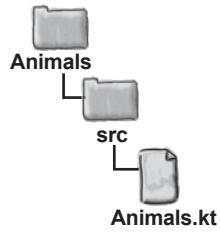


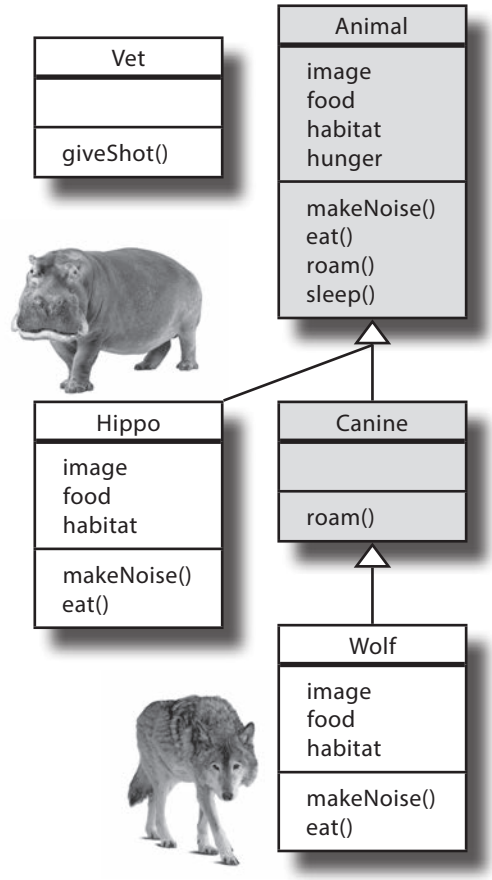
Продолжение на следующей странице.

Продолжение кода...

```
class Vet {
    fun giveShot(animal: Animal) {
        //Code to do something medical
        animal.makeNoise()
    }
}

fun main(args: Array<String>) {
    val animals = arrayOf(Hippo(), Wolf())
    for (item in animals) {
        item.roam()
        item.eat()
    }

    val vet = Vet()
    val wolf = Wolf()
    val hippo = Hippo()
    vet.giveShot(wolf)
    vet.giveShot(hippo)
}
```

Код на этой странице не изменился.





Давайте запустим наш код и посмотрим, что произойдет.

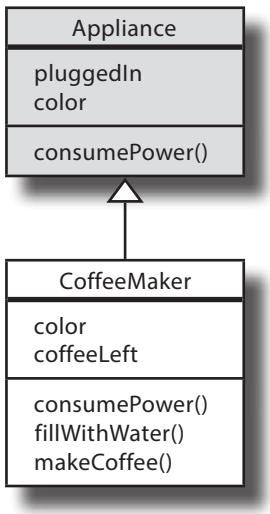
Тест-драйв

Запустите свой код. В окне вывода IDE выводится тот же текст, но в новой версии для управления возможностью создания экземпляров используются абстрактные классы.

```
The Animal is roaming
The Hippo is eating grass
The Canine is roaming
The Wolf is eating meat
Hooooowl!
Grunt! Grunt!
```

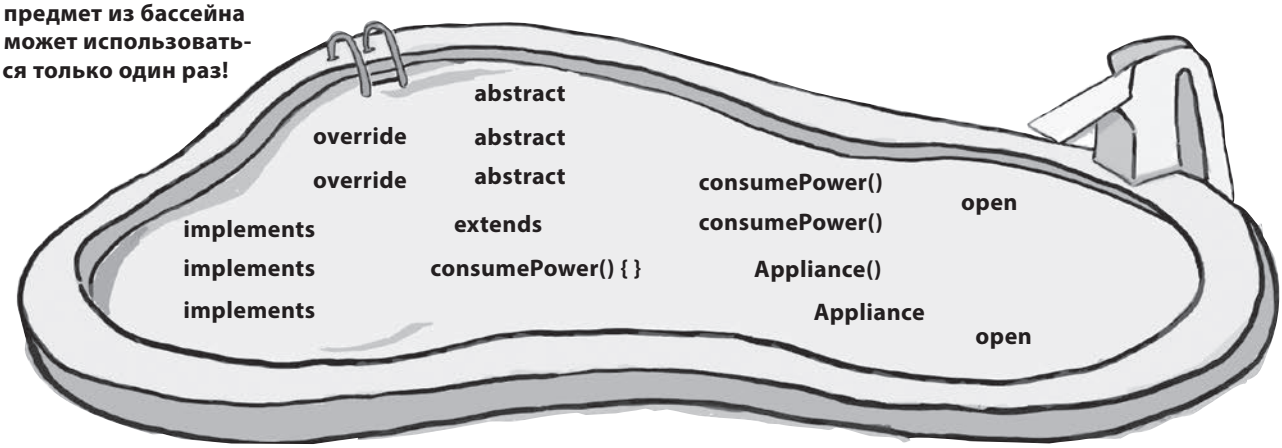
У бассейна

Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: выложить код, соответствующий приведенной ниже иерархии классов.



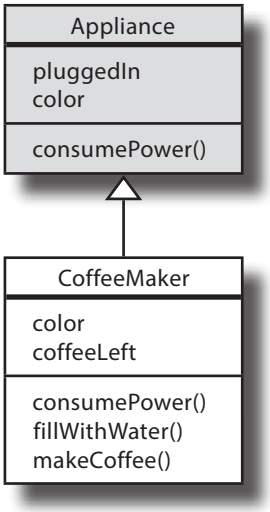

```
_____ class Appliance {
    var pluggedIn = true
    _____ val color: String

    _____ fun _____
}

class CoffeeMaker : _____ {
    _____val color = ""
    var coffeeLeft = false

    _____fun _____ {
        println("Consuming power")
    }

    fun fillWithWater() {
        println("Fill with water")
    }

    fun makeCoffee() {
        println("Make the coffee")
    }
}
```

Примечание: каждый предмет из бассейна может использоваться только один раз!



У бассейна. Решение

Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: выложить код, соответствующий приведенной ниже иерархии классов.



Класс Appliance помечается как абстрактный, как и свойство color и функция consumePower().

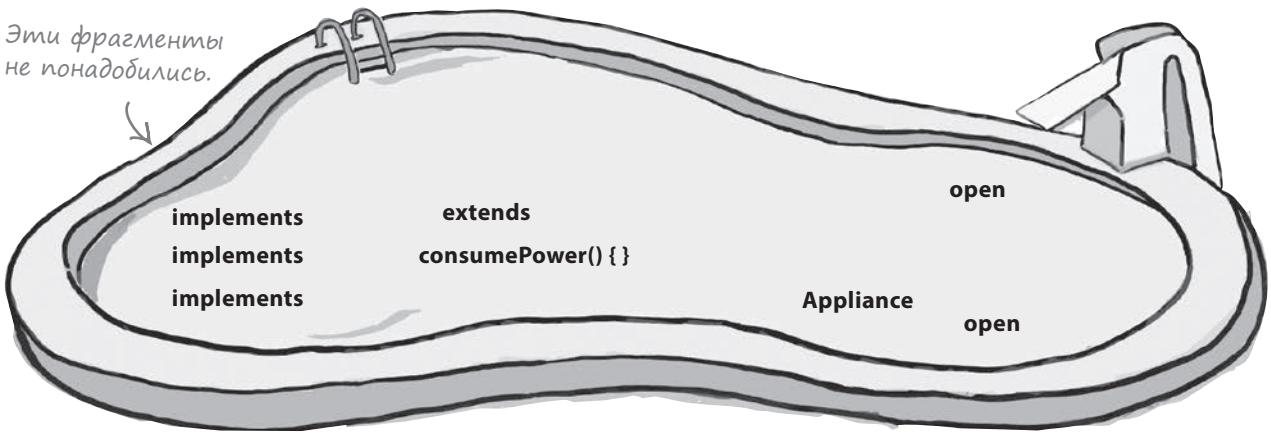
```
abstract class Appliance {
    var pluggedIn = true
    abstract val color: String

    abstract fun consumePower()
}
```

CoffeeMaker наследует от Appliance.

```
class CoffeeMaker : Appliance() {
    override val color = ""
    var coffeeLeft = false

    override fun consumePower() {
        println("Consuming power")
    }

    fun fillWithWater() {
        println("Fill with water")
    }

    fun makeCoffee() {
        println("Make the coffee")
    }
}
```

Свойство color переопределяется.

Функция consumePower() переопределяется.

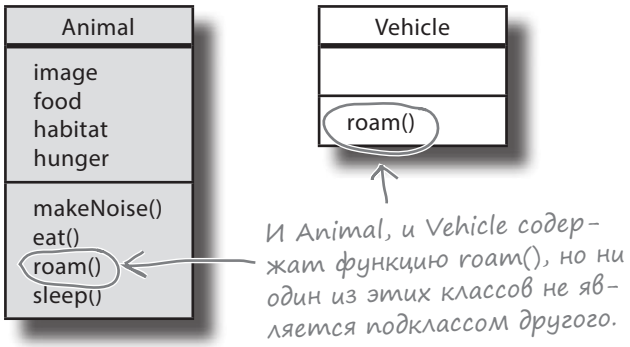


Независимые классы могут обладать общим поведением

Пока что вы узнали, как создавать иерархии наследования, комбинируя абстрактные и конкретные подклассы. Такой подход помогает избежать дублирования кода и позволяет писать гибкий код, в котором можно воспользоваться преимуществами полиморфизма. Но что если вы хотите включить в приложение классы, которые содержат лишь *часть* поведения, определяемого в иерархии наследования?

Предположим, что вы хотите добавить в приложение с животными класс `Vehicle`, который содержит только одну функцию `roam`. Это позволит нам создавать объекты `Vehicle` (транспортное средство), которые могут перемещаться в среде с животными.

Было бы полезно, если бы класс `Vehicle` мог каким-то образом реализовать функцию `roam` из класса `Animal`, так как это позволило бы создать полиморфный массив объектов, поддерживающих `roam`, и вызывать функции для каждого объекта. Однако класс `Vehicle` не принадлежит иерархии суперкласса `Animal`, так как он не проходит «правило ЯВЛЯЕТСЯ»: утверждение «Vehicle ЯВЛЯЕТСЯ Animal» выглядит бессмысленно, как и утверждение «Animal ЯВЛЯЕТСЯ Vehicle».



Если два класса не проходят «правило ЯВЛЯЕТСЯ», это означает, что, скорее всего, они не принадлежат одной иерархии суперкласса.

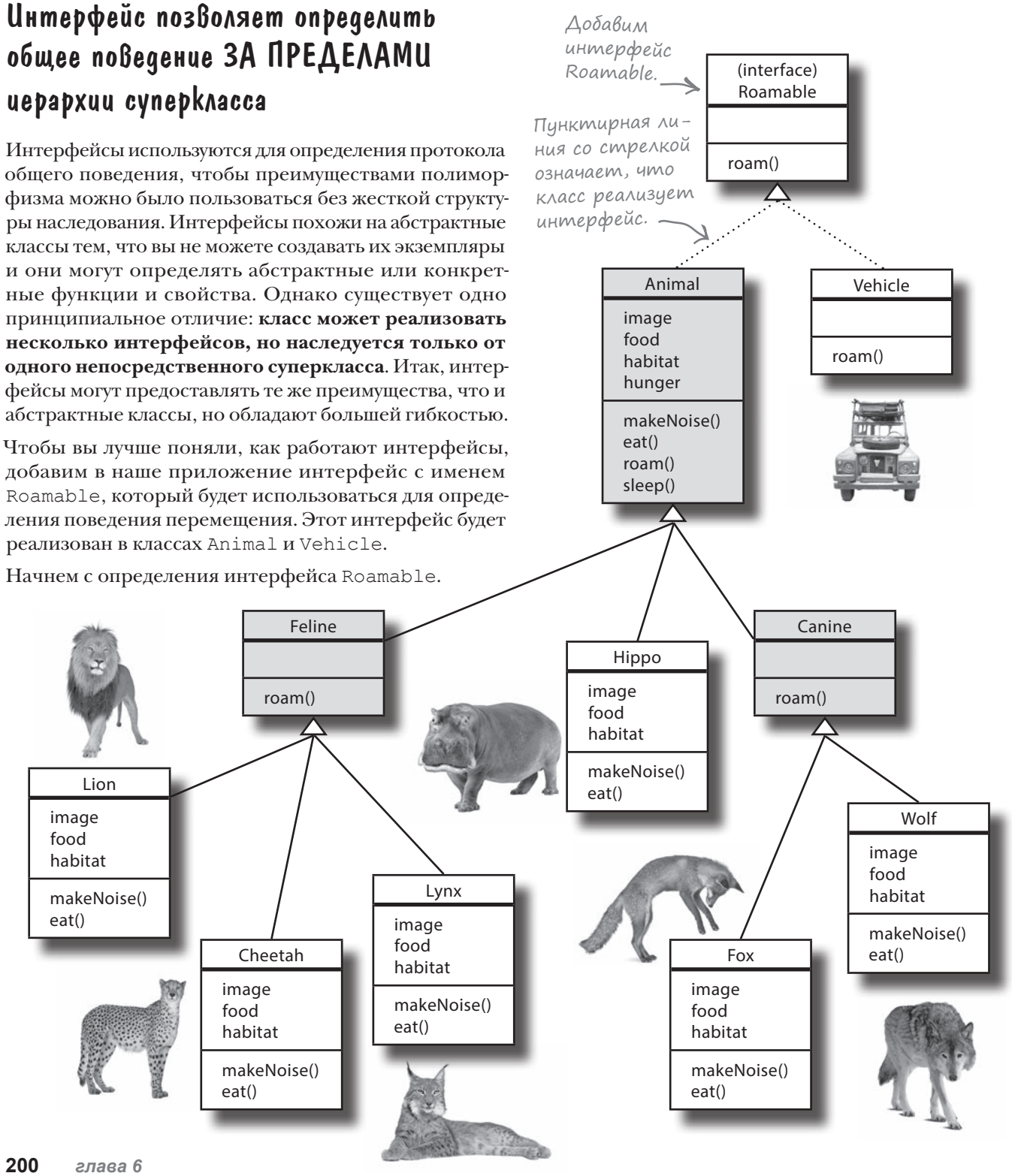
В ситуации с независимыми классами, обладающими общим поведением, такое поведение моделируется при помощи **интерфейса**. Что же такое интерфейс?

Интерфейс позволяет определить общее поведение ЗА ПРЕДЕЛАМИ иерархии суперкласса

Интерфейсы используются для определения протокола общего поведения, чтобы преимуществами полиморфизма можно было пользоваться без жесткой структуры наследования. Интерфейсы похожи на абстрактные классы тем, что вы не можете создавать их экземпляры и они могут определять абстрактные или конкретные функции и свойства. Однако существует одно принципиальное отличие: **класс может реализовать несколько интерфейсов, но наследуется только от одного непосредственного суперкласса**. Итак, интерфейсы могут предоставлять те же преимущества, что и абстрактные классы, но обладают большей гибкостью.

Чтобы вы лучше поняли, как работают интерфейсы, добавим в наше приложение интерфейс с именем `Roamable`, который будет использоваться для определения поведения перемещения. Этот интерфейс будет реализован в классах `Animal` и `Vehicle`.

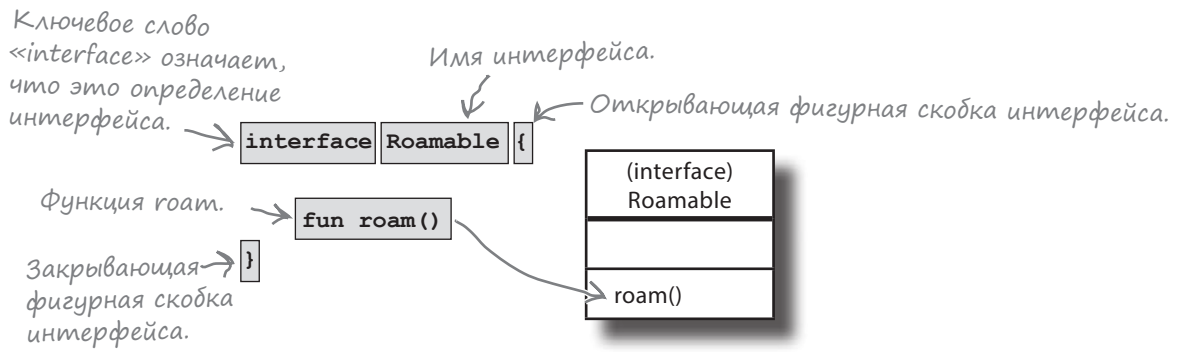
Начнем с определения интерфейса `Roamable`.



Определение интерфейса Roamable

Мы создадим интерфейс `Roamable`, который будет использоваться для определения общего протокола поведения перемещения. В нем будет определена абстрактная функция с именем `roam`, которая должна быть реализована классами `Animal` и `Vehicle` (код этих классов будет приведен позднее).

Вот как выглядит интерфейс `Roamable` (мы добавим его в проект Animals через несколько страниц):



Функции интерфейсов могут быть абстрактными или конкретными

Чтобы добавить функцию в интерфейс, включите ее в тело интерфейса (в фигурных скобках `{}`). В нашем примере определяется абстрактная функция с именем `roam`, поэтому код будет выглядеть так:

```
interface Roamable {
    fun roam()
}
```

Так определяется абстрактная функция в интерфейсе.

При добавлении абстрактной функции в интерфейс не нужно ставить перед именем функции префикс `abstract`, как это следовало бы сделать при добавлении абстрактной функции в абстрактный класс. В случае с интерфейсом компилятор автоматически определяет, что функция без тела должна быть абстрактной, поэтому ее не нужно определять как абстрактную.

Также в интерфейсы можно добавлять конкретные функции, для этого достаточно определить функцию с телом. Например, следующий код предоставляет конкретную реализацию для функции `roam`:

```
interface Roamable {
    fun roam() {
        println("The Roamable is roaming")
    }
}
```

Чтобы добавить конкретную функцию в интерфейс, просто определите тело функции.

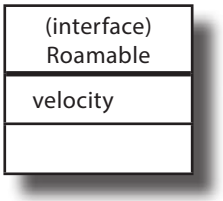
Как видите, функции определяются в интерфейсах почти так же, как в абстрактных классах. А как насчет свойств?

Как определяются свойства интерфейсов

Чтобы добавить свойство в интерфейс, включите его в тело интерфейса. Это *единственный* способ определения свойств интерфейсов, так как, в отличие от абстрактных классов, **интерфейсы не могут иметь конструкторов**. Например, в следующем примере в интерфейс `Roamable` добавляется абстрактное свойство Int с именем `velocity`:

```
interface Roamable {
    val velocity: Int
}
```

Как и в случае с абстрактными функциями, перед абстрактными свойствами не обязательно ставить ключевое слово abstract.



В отличие от свойств в абстрактных классах, свойства, определяемые в интерфейсах, не могут содержать состояния, а следовательно, не могут инициализироваться. Впрочем, вы можете вернуть значение свойства, определив пользовательский get-метод следующего вида:

```
interface Roamable {
    val velocity: Int
        get() = 20
}
```

Возвращает значение 20 при каждом обращении к свойству. Тем не менее это свойство может определяться в любых классах, реализующих интерфейс.

Другое ограничение заключается в том, что свойства интерфейсов **не базируются на полях данных**. В главе 4 вы узнали о том, что поле данных предоставляет ссылку на нижележащее значение свойства, так что вы не сможете определить пользовательский set-метод, обновляющий значение свойства:

```
interface Roamable {
    var velocity: Int
        get() = 20
        set(value) {
            field = value
        }
}
```

Если вы попытаетесь включить такой код в интерфейс, он не будет компилироваться. Дело в том, что в интерфейсах не может использоваться ключевое слово «field», и вы не сможете обновить базовое значение свойства.

Однако при этом вы можете определить set-метод, если он не пытается обращаться к базовому полю данных свойства. Например, следующий код вполне допустим:

```
interface Roamable {
    var velocity: Int
        get() = 20
        set(value) {
            println("Unable to update velocity")
        }
}
```

Этот код компилируется, потому что в нем не используется ключевое слово field. Тем не менее этот код не обновляет базовое значение свойства.

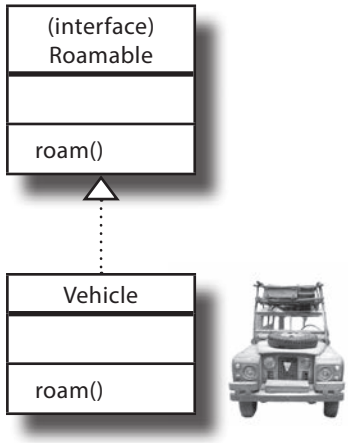
Теперь, когда вы знаете, как определяются интерфейсы, посмотрим, как они реализуются.

Объявите о том, что класс реализует интерфейс...

Объявление о реализации интерфейса классом почти не отличается от объявления о наследовании класса от суперкласса: поставьте в заголовок двоеточие, за которым следует имя интерфейса. Например, следующее объявление сообщает, что класс `Vehicle` реализует интерфейс `Roamable`:

```
class Vehicle : Roamable {
    ...
}
```

Это означает «Класс Vehicle реализует интерфейс Roamable».



В отличие от объявления о наследовании от суперкласса, после имени интерфейса не нужно ставить круглые скобки. Круглые скобки необходимы только для вызова конструктора суперкласса, а у интерфейсов нет конструкторов.

...а затем переопределите его свойства и функции

Объявление о реализации интерфейса классом наделяет класс всеми свойствами и функциями, входящими в интерфейс. Вы можете переопределять все свойства и функции — это делается точно так же, как при переопределении свойств и функций, унаследованных от суперкласса. Например, следующий код переопределяет функцию `roam` из интерфейса `Roamable`:

```
class Vehicle : Roamable {
    override fun roam() {
        println("The Vehicle is roaming")
    }
}
```

Этот код переопределяет функцию roam(), унаследованную классом Vehicle от интерфейса Roamable.

По аналогии с абстрактными суперклассами любые конкретные классы, реализующие интерфейс, должны содержать конкретную реализацию любых абстрактных свойств и функций. Например, класс `Vehicle` напрямую реализует интерфейс `Roamable`, поэтому он должен реализовать все абстрактные свойства и функции, определенные в этом интерфейсе, иначе код не будет компилироваться. Но если класс, реализующий интерфейс, объявлен абстрактным, этот класс может либо реализовать свойства и функции самостоятельно, либо передать эстафету своим подклассам.

Учтите, что класс, реализующий интерфейс, может определять свои свойства и функции. Например, класс `Vehicle` может определить собственное свойство `fuelType` и при этом реализовать интерфейс `Roamable`.

Ранее в этой главе было сказано, что класс может реализовать несколько интерфейсов. Посмотрим, как это делается.

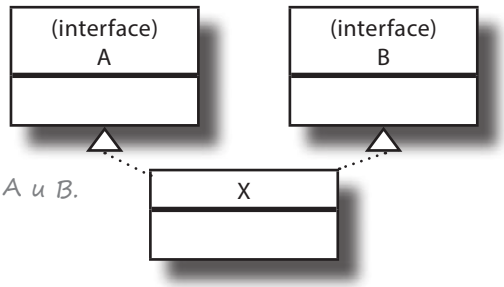
Конкретные классы не могут содержать абстрактные свойства и функции, поэтому они должны реализовать все унаследованные свойства и функции.

Реализация нескольких интерфейсов

Чтобы объявить, что класс (или интерфейс) реализует несколько интерфейсов, добавьте их в заголовок класса и разделите запятыми. Предположим, что у вас имеются два интерфейса с именами `A` и `B`. Следующий код объявляет, что класс с именем `X` реализует оба интерфейса:

```
class X : A, B {
    ...
}
```

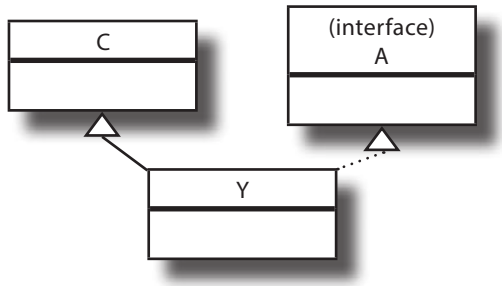
Класс X реализует интерфейсы A и B.



Кроме реализации одного или нескольких интерфейсов, класс также может наследоваться от суперкласса. В следующем фрагменте класс `Y` реализует интерфейс `A` и наследуется от класса `C`:

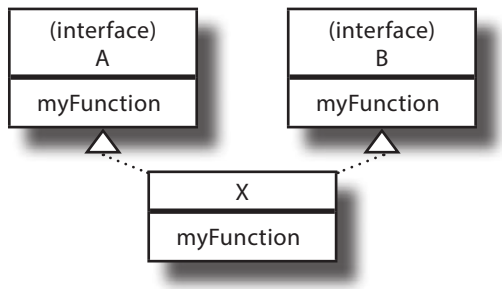
```
class Y : C(), A {
    ...
}
```

Класс Y наследуется от класса C и реализует интерфейс A.



Если класс наследует несколько реализаций одной функции или свойства, то должен предоставить собственную реализацию или указать, какую версию функции или свойства он будет использовать. Например, если интерфейсы `A` и `B` включают конкретную функцию с именем `myFunction`, а класс реализует оба интерфейса, класс `X` должен предоставить реализацию `myFunction`, чтобы компилятор знал, как обработать вызов функции:

```
interface A {
    fun myFunction() { println("from A") }
}

interface B {
    fun myFunction() { println("from B") }
}

class X : A, B {
    override fun myFunction() {
        super<A>.myFunction()
        super<B>.myFunction()
        //Специфический код класса X
    }
}
```



super<A> обозначает суперкласс (или интерфейс) с именем A. Таким образом, super<A>.myFunction() вызывает версию myFunction, определенную в A.

Этот код вызывает версию myFunction, определенную в A, а затем версию, определенную в B. После этого выполняется код, специфический для класса X.

Класс, подкласс, абстрактный класс или интерфейс?

Не уверены в том, что именно следует создать — класс, абстрактный класс или интерфейс? Вот несколько советов, которые помогут определиться:

- Создайте класс без суперкласса, если ваш новый класс не проходит «правило ЯВЛЯЕТСЯ» для любого другого типа.
- Создайте подкласс, наследующий от суперкласса, чтобы создать более конкретную версию класса и переопределить/добавить новые аспекты поведения.
- Создайте абстрактный класс, чтобы определить прототип для группы подклассов. Объявите класс абстрактным, чтобы никто гарантированно не смог создать объекты этого типа.
- Создайте интерфейс, чтобы определить общее поведение или роль, которую могут исполнять другие классы (независимо от того, где эти классы находятся в дереве наследования).

Наследуйте от одного, реализуйте два.

Класс в Kotlin может иметь только одного родителя (суперкласс); этот родительский класс определяет, что собой представляет подкласс. Но класс может реализовать несколько интерфейсов, которые определяют возможные роли класса.

Итак, теперь вы знаете, как определяются и реализуются интерфейсы, и мы можем обновить код нашего проекта Animals.

Часто Задаваемые Вопросы

В: Существуют ли специальные правила назначения имен интерфейсов?

О: Никаких жестких требований нет, но поскольку интерфейсы определяют поведение, часто используются слова с суффиксами *-ible* или *-able*; они ассоциируются с тем, что *делает* интерфейс, а не с тем, *что* он собой представляет.

В: Почему интерфейсы и абстрактные классы не нужно помечать как открытые?

О: Интерфейсы и абстрактные классы существуют для реализации или наследования. Компилятор это знает, поэтому каждый интерфейс и абстрактный класс неявно является открытым, хотя и не помечается как открытый.

В: Вы сказали, что любые свойства и функции, определенные в интерфейсе, могут переопределяться. Может, вы имели в виду любые *абстрактные* свойства и функции?

О: Нет. При наследовании можно переопределять любые свойства и функции. Таким образом, даже если функция в интерфейсе имеет конкретную реализацию, ее все равно можно переопределить.

В: Может ли интерфейс наследоваться от суперкласса?

О: Нет, но он *может* реализовать один или несколько интерфейсов.

В: Когда следует определять для функции конкретную реализацию, а когда ее следует оставить абстрактной?

О: Обычно конкретная реализация предоставляется в том случае, если вам удастся найти функциональность, полезную для всех наследующих ее классов.

Если найти полезную реализацию не удается, обычно такая функция оставляется абстрактной, так как тем самым вы заставляете любые конкретные подклассы предоставить собственную реализацию.

Обновление проекта Animals

Мы добавим в проект новый интерфейс `Roamable` и класс `Vehicle`. Класс `Vehicle` будет реализовывать интерфейс `Roamable`; то же самое делает абстрактный класс `Animal`.

Обновите свою версию кода в файле *Animals.kt*, чтобы она соответствовала нашей (изменения выделены жирным шрифтом):

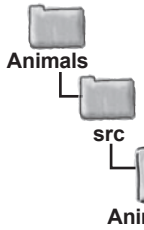
Добавление интерфейса Roamable с абстрактной функцией roam().

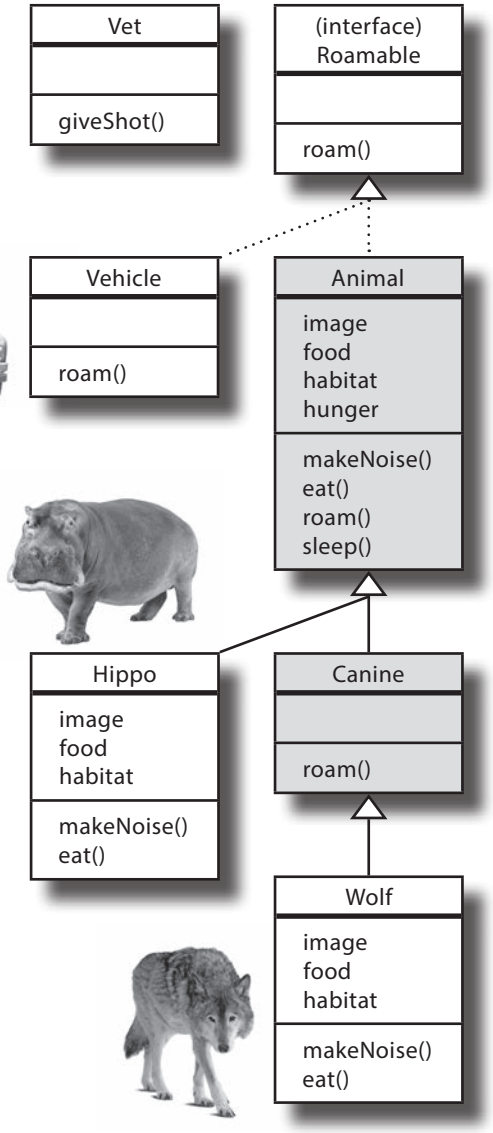
```
interface Roamable {
    fun roam()
}
```

Класс Animal должен реализовать интерфейс Roamable.

```
abstract class Animal : Roamable {
    abstract val image: String
    abstract val food: String
    abstract val habitat: String
    var hunger = 10

    abstract fun makeNoise()

    abstract fun eat()

    override fun roam() {
        println("The Animal is roaming")
    }

    fun sleep() {
        println("The Animal is sleeping")
    }
}
```

Переопределение функции roam() из интерфейса Roamable.



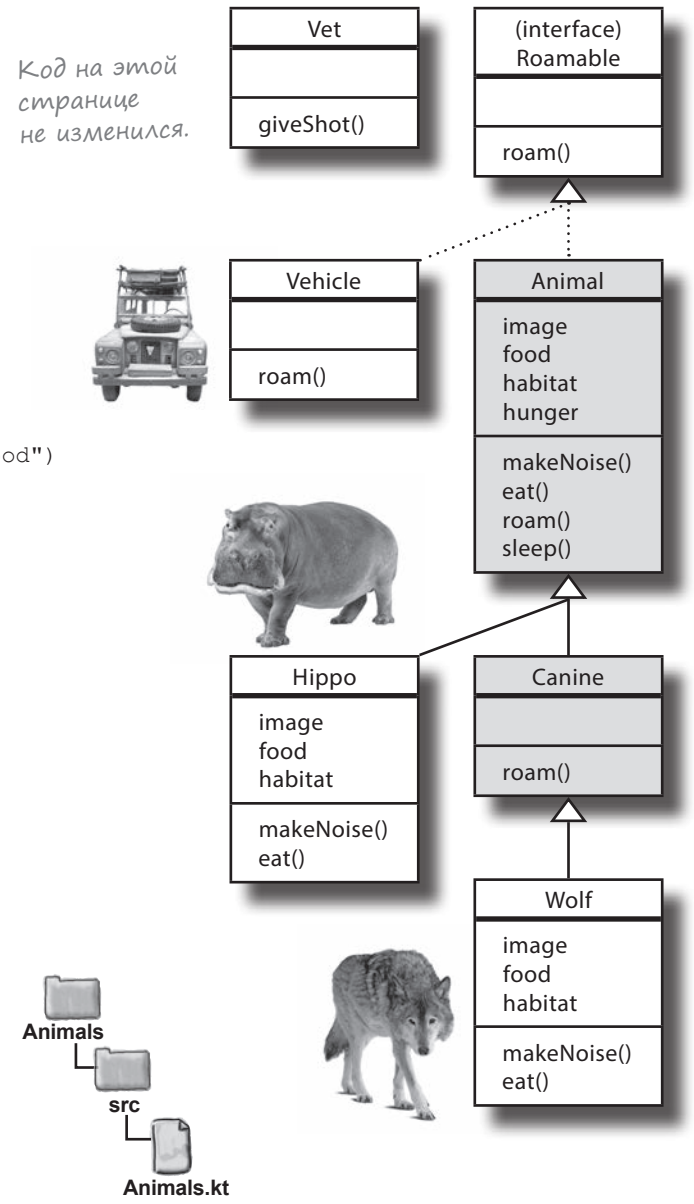


Продолжение на следующей странице.

Продолжение кода...

```
class Hippo : Animal() {
    override val image = "hippo.jpg"
    override val food = "grass"
    override val habitat = "water"

    override fun makeNoise() {
        println("Grunt! Grunt!")
    }

    override fun eat() {
        println("The Hippo is eating $food")
    }
}

abstract class Canine : Animal() {
    override fun roam() {
        println("The Canine is roaming")
    }
}

class Wolf : Canine() {
    override val image = "wolf.jpg"
    override val food = "meat"
    override val habitat = "forests"

    override fun makeNoise() {
        println("Hooooowl!")
    }

    override fun eat() {
        println("The Wolf is eating $food")
    }
}
```

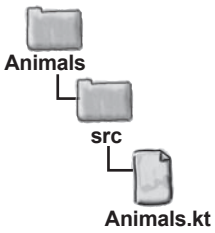
Код на этой странице не изменился.

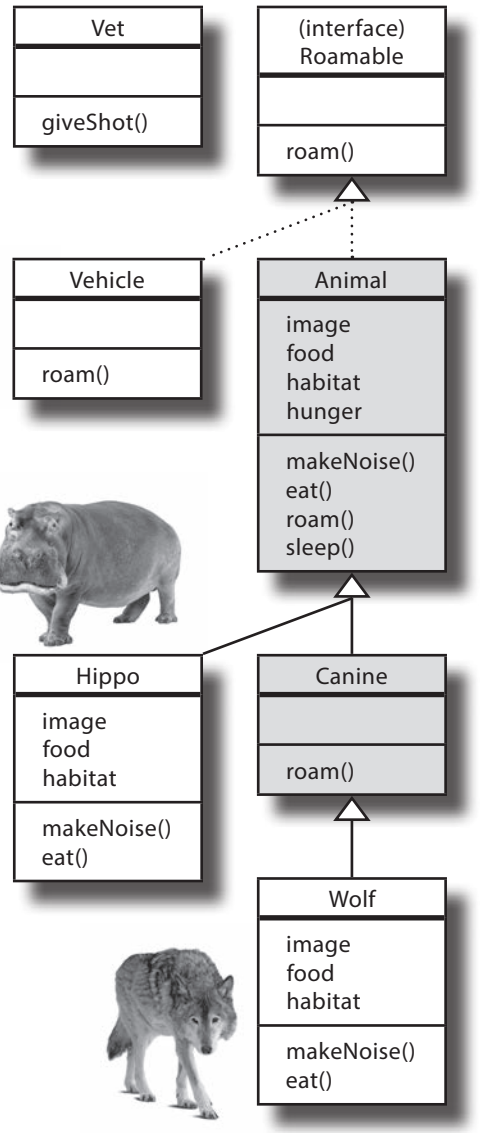


Продолжение на следующей странице.

Продолжение кода...

```
class Vehicle : Roamable {          ← Добавление класса Vehicle.
    override fun roam() {
        println("The Vehicle is roaming")
    }
}

class Vet {
    fun giveShot(animal: Animal) {
        //Код ветеринарной процедуры
        animal.makeNoise()
    }
}

fun main(args: Array<String>) {
    val animals = arrayOf(Hippo(), Wolf())
    for (item in animals) {
        item.roam()
        item.eat()
    }

    val vet = Vet()
    val wolf = Wolf()
    val hippo = Hippo()
    vet.giveShot(wolf)
    vet.giveShot(hippo)
}
```





Давайте проверим наш код в деле.

Тест-драйв

Выполните код. В окне вывода IDE выводится уже знакомый текст, но в новой версии класс `Animal` использует интерфейс `Roamable`.

В функции main нам по-прежнему нужно использовать объекты `Vehicle`. Попробуйте для начала выполнить следующее упражнение.

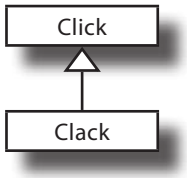
```
The Animal is roaming
The Hippo is eating grass
The Canine is roaming
The Wolf is eating meat
Hooooowl!
Grunt! Grunt!
```

Упражнение

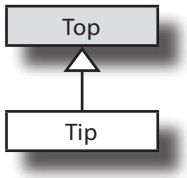
Слева изображены диаграммы классов. Напишите для них правильные объявления Kotlin. Код для первой диаграммы мы написали за вас.

Диаграмма:

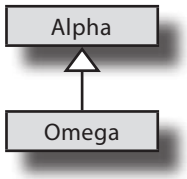
1



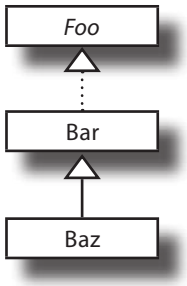
2



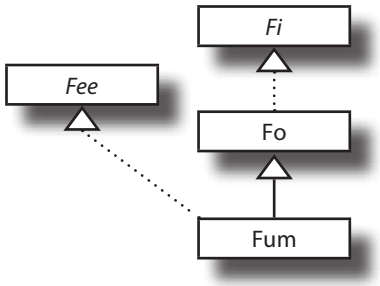
3



4



5



Объявление:

1
```
open class Click { }
class Clack : Click() { }
```

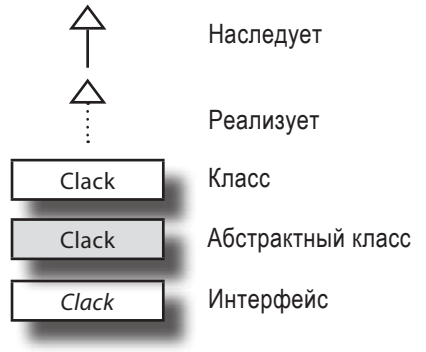
2

3

4

5

Условные обозначения:



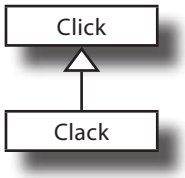
Упражнение
Решение

Слева изображены диаграммы классов. Напишите для них правильные объявления Kotlin. Код для первой диаграммы мы написали за вас.

Диаграмма:

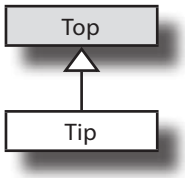
Объявление:

1




```
open class Click { }
class Clack : Click() { }
```

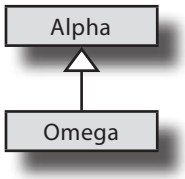
2



Tip реализует абстрактный класс Top.

```
abstract class Top { }
class Tip : Top() { }
```

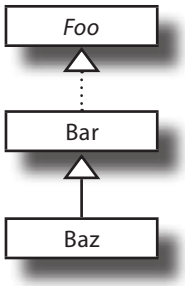
3



Omega наследуется от Alpha. Оба класса являются абстрактными.

```
abstract class Alpha { }
abstract class Omega : Alpha() { }
```

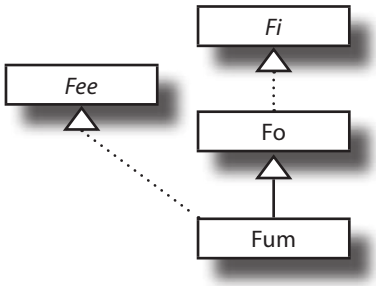
4



Bar необходимо пометить как открытый, чтобы класс Baz мог наследоваться от него.

```
interface Foo { }
open class Bar : Foo { }
class Baz : Bar() { }
```

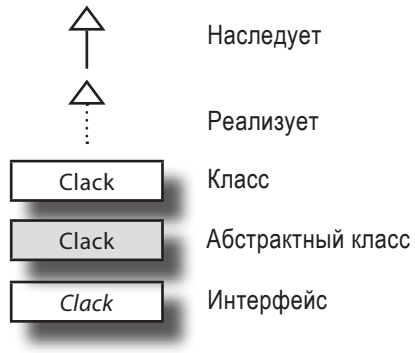
5




```
interface Fee { }
interface Fi { }
open class Fo : Fi { }
class Fum : Fo(), Fee { }
```

Fum наследуется от класса Fo() и реализует интерфейс Fee.

Условные обозначения:



Интерфейсы позволяют использовать полиморфизм

Вы уже знаете, что с интерфейсами приложения могут использовать полиморфизм. Например, благодаря полиморфизму можно создать массив объектов `Roamable` и вызвать функцию `roam` каждого объекта:

```
val roamables = arrayOf(Hippo(), Wolf(), Vehicle())
for (item in roamables) {
    item.roam()
}
```

Эта строка создает массив объектов Roamable.

Так как массив roamables содержит объекты Roamable, это означает, что переменная item относится к типу Roamable.

А если вам недостаточно обращения к функциям и свойствам, определенным в интерфейсе `Roamable`? Что, если вы хотите также вызвать для каждого объекта `Animal` функцию `makeNoise`? Использовать для этого следующий код не удастся:

```
item.makeNoise()
```

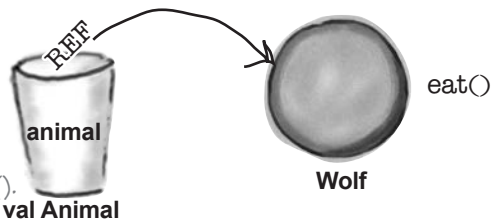
потому что `item` — переменная типа `Roamable`, которой о функции `makeNoise` ничего не известно.

Проверка типа объекта для обращения к специфическому поведению

Чтобы обратиться к поведению, которое не определяется типом переменной, сначала следует использовать оператор **`is`** для проверки типа используемого объекта. Если объект относится к подходящему типу, компилятор позволит обратиться к поведению, присущему этому типу. Например, следующий код проверяет, относится ли объект из переменной `Animal` к классу `Wolf`, и если относится, то вызывает функцию `eat`:

```
val animal: Animal = Wolf()
if (animal is Wolf) {
    animal.eat()
}
```

Компилятор знает, что объект относится к классу Wolf, и поэтому вызывает его функцию eat().



В этом коде компилятор знает, что объект относится к классу `Wolf`, поэтому для него может быть безопасно выполнен код, специфический для класса `Wolf`. Это означает, что при попытке вызвать функцию `eat` для каждого объекта `Animal` из массива `Roamable` можно использовать следующий код:

```
val roamables = arrayOf(Hippo(), Wolf(), Vehicle())
for (item in roamables) {
    item.roam()
    if (item is Animal) {
        item.eat()
    }
}
```

Если элемент относится к иерархии Animal, компилятор знает, что он может вызвать функцию eat() элемента.

Используйте оператор is для проверки того, относится ли объект к заданному классу (или одному из его подклассов).

Оператор `is` может пригодиться в разных ситуациях, о которых мы расскажем подробнее.

Когда используется оператор is

Вот несколько типичных ситуаций, в которых можно воспользоваться оператором `is`:

Условия if

Как вы уже видели, оператор `is` может использоваться как условие в конструкции `if`. Например, следующий код присваивает строку «Wolf» переменной `str`, если переменная `animal` содержит ссылку на объект `Wolf`, или строку «not Wolf» в противном случае:

```
val str = if (animal is Wolf) "Wolf" else "not Wolf"
```

Обратите внимание: должна существовать возможность того, что проверяемый объект относится к заданному типу, в противном случае код не будет компилироваться. Вы не сможете проверить, что переменная Animal содержит ссылку на Int, потому что типы Animal и Int несовместимы.

Условия с && и ||

Более сложные условия строятся при помощи операторов `&&` и `||`. Так, следующий код проверяет, содержит ли переменная `Roamable` ссылку на объект `Animal`, и если содержит, то проверяет, что свойство `hunger` объекта `Animal` меньше 5:

```
if (roamable is Animal && roamable.hunger < 5) {
    //Код обработки голодного животного
}
```

Правая часть условия if обрабатывается только в том случае, если переменная roamable относится к классу Animal, поэтому мы можем обратиться к ее свойству hunger.

Оператор `!is` проверяет, что объект *не* относится к заданному типу. Например, следующий код означает: «Если переменная roamable не содержит ссылку на Animal, или если свойство hunger объекта Animal больше либо равно 5»:

```
if (roamable !is Animal || x.hunger >= 5) {
    //Код для обработки "не животного" или сытого животного
}
```

Вспомните: правая часть условия || обрабатывается только в том случае, если левая часть ложна. Следовательно, правая часть будет проверяться только в том случае, если roamable относится к классу Animal.

В цикле while

Если вы хотите использовать оператор `is` в условии цикла `while`, код может выглядеть примерно так:

```
while (animal is Wolf) {
    //Код выполняется, если объект относится к классу Wolf
}
```

В этом примере цикл будет выполняться, пока переменная `animal` содержит ссылку на объект `Wolf`.

Оператор `is` также может использоваться в командах **`when`**. Давайте разберемся, что это за команды и как ими пользоваться.

Оператор when проверяет переменную по нескольким вариантам

Команды `when` удобны в ситуациях, когда значение переменной нужно проверить по набору вариантов. По сути, они работают как цепочки выражений `if/else`, но этот синтаксис получается более компактным и понятным.

Команда `when` выглядит примерно так:

Проверить значение переменной x.

Если значение x равно 0, выполняется этот код.

Этот код выполняется, если значение x равно 1 или 2.

Команды when могут содержать секцию else.

Этот блок кода выполняется в том случае, если x содержит другое значение.

```
when (x) {
    0 -> println("x is zero")
    1, 2 -> println("x is 1 or 2")
    else -> {
        println("x is neither 0, 1 nor 2")
        println("x is some other value")
    }
}
```

Приведенный выше код берет переменную `x` и проверяет ее значение по нескольким вариантам. Фактически это означает: «Если значение *x* равно 0, вывести "x is zero", если 1 или 2 — вывести "x is 1 or 2", в остальных случаях вывести другой текст».

Если вы хотите выполнять разный код в зависимости от типа проверяемого объекта, используйте оператор `is` внутри команды `when`. Например, следующий код использует оператор `is` для проверки типа объекта, на который ссылается переменная `roamable`. Для типа `Wolf` выполняется код, специфический для `Wolf`; для типа `Hippo` выполняется код, специфический для `Hippo`; а для любых других разновидностей `Animal` (не `Wolf` и не `Hippo`) будет выполняться другой код:

Проверяет значение roamable.

```
when (roamable) {
    is Wolf -> {
    //Код для Wolf
    }
    is Hippo -> {
        //Код для Hippo
    }
    is Animal -> {
        //Код для других видов Animal
    }
}
```

Этот код будет выполняться только в том случае, если roamable относится к другому классу Animal, кроме Wolf или Hippo.

Использование when как выражения

Оператор `when` также может использоваться как выражение; иначе говоря, вы можете использовать его для возвращения значения. Например, следующий код использует выражение `when` для присваивания значения переменной:

```
var y = when (x) {
    0 -> true
    else -> false
}
```

При таком использовании оператора `when` вы *должны* учесть каждое значение, которое может принимать переменная, — обычно для этого добавляется секция `else`.

Оператор is ...обычно выполняет умное приведение типа

В большинстве случаев оператор `is` выполняет **умное приведение типа**. *Приведение типа* означает, что компилятор работает с переменной так, словно ее тип отличается от того, с которым она объявлена, а *умное приведение типа* – что компилятор автоматически выполняет преобразование за вас. Например, следующий код использует оператор `is` для умного преобразования переменной с именем `item` к классу `Wolf`, чтобы внутри тела условия `if` компилятор мог работать с переменной `item` так, словно она относится к классу `Wolf`:


```
if (item is Wolf) {
    item.eat()
    item.makeNoise()
    //Other Wolf-specific code
}
```

Выполняется умное приведение item в Wolf на время выполнения этого блока.

Оператор `is` выполняет умное приведение каждый раз, когда компилятор может гарантировать, что переменная не изменится между проверкой типа объекта и его использованием. В приведенном выше коде компилятор знает, что переменной `item` не может быть присвоена ссылка на переменную другого типа между вызовом оператора `is` и вызовами функций, специфических для `Wolf`.

Но в некоторых ситуациях умное приведение типа не выполняется. В частности, оператор `is` не выполняет умного приведения свойств `var` в классах, потому что компилятор не может гарантировать, что другой код не вмешается и не обновит свойство. Следовательно, такой код компилироваться не будет, потому что компилятор не может преобразовать переменную `r` в `Wolf`:

```
class MyRoamable {
    var r: Roamable = Wolf()

    fun myFunction() {
        if (r is Wolf) {
            r.eat()
        }
    }
}
```

Компилятор не может выполнить умное приведение свойства r типа Roamable в Wolf. Дело в том, что компилятор не может гарантировать, что другой код не обновит свойство между проверкой типа и его использованием. Следовательно, этот код компилироваться не будет.

РАССЛАБЬТЕСЬ

Вам не нужно запоминать все ситуации, в которых не может использоваться умное преобразование типа.

Если вы попытаетесь использовать умное преобразование неподходящим образом, компилятор сообщит вам об этом.

Что же делать в подобных ситуациях?

Используйте as для выполнения явного приведения типа

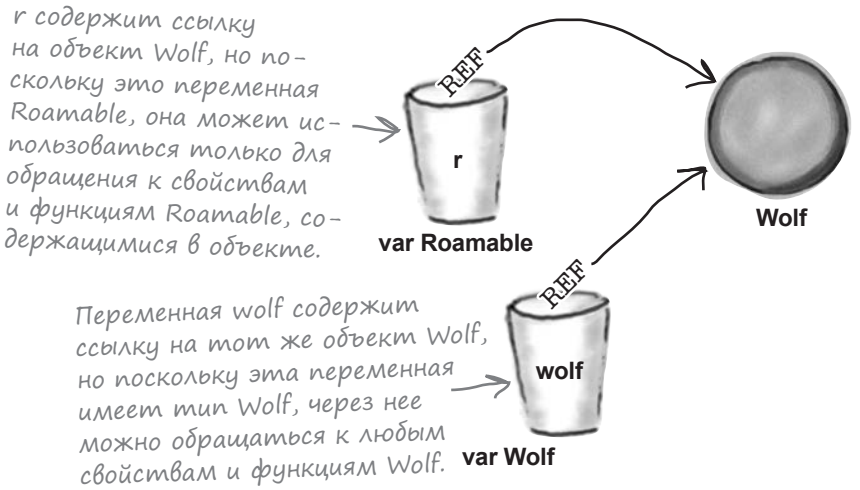
Если вы хотите получить доступ к поведению фактического объекта, но компилятор не может выполнить умное преобразование, можно выполнить явное преобразование объекта к нужному типу.

Допустим, вы уверены в том, что переменная `Roamable` с именем `r` содержит ссылку на объект `Wolf`, и вы хотите обратиться к поведению объекта, специфическому для `Wolf`. В такой ситуации можно воспользоваться оператором **`as`** для копирования ссылки, хранящейся в переменной `Roamable`, и ее преобразования к новой переменной `Wolf`. После этого переменная `Wolf` может использоваться для обращения к поведению `Wolf`. Код выглядит так:

```
var wolf = r as Wolf
wolf.eat()
```

Этот код явно приводит объект к классу Wolf, чтобы вы могли вызывать его функции Wolf.

При этом в переменных `wolf` и `r` **хранятся ссылки на один объект `Wolf`**. Но если переменная `r` знает лишь то, что объект реализует интерфейс `Roamable`, переменная `wolf` знает, что объект действительно относится к классу `Wolf`, и поэтому с ним можно работать как с объектом `Wolf`, которым он на самом деле и является:



Если вы не уверены в том, что фактический объект имеет класс `Wolf`, используйте оператор `is` для проверки перед тем, как выполнять преобразование:

```
if (r is Wolf) {
    val wolf = r as Wolf
    wolf.eat()
}
```

Если переменная r относится к классу Wolf, приведите ее в Wolf и вызовите функцию eat().

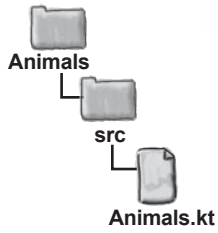
Теперь, когда вы знаете, как работает преобразование (и умное преобразование) типов, мы обновим код проекта Animals.

Обновление проекта Animals

Мы обновили код функции `main` и включили в него массив объектов `Roamable`. Обновите свою версию функции из файла *Animals.kt*, чтобы она соответствовала нашей (изменения выделены жирным шрифтом):

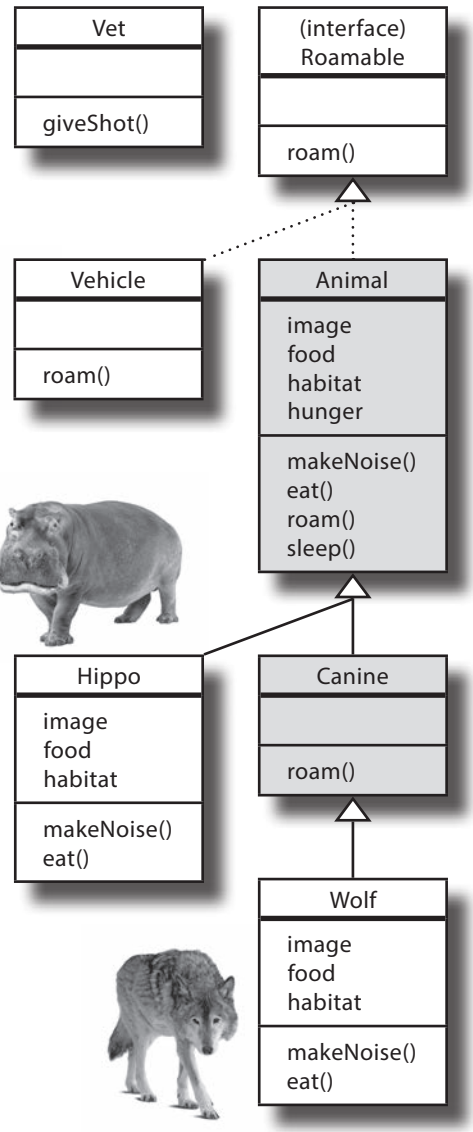
Изменяется только код функции main.

```
...
fun main(args: Array<String>) {
    val animals = arrayOf(Hippo(), Wolf())
    for (item in animals) {
        item.roam()
        item.eat()
    }

    val vet = Vet()
    val wolf = Wolf()
    val hippo = Hippo()
    vet.giveShot(wolf)
    vet.giveShot(hippo)

    val roamables = arrayOf(Hippo(), Wolf(), Vehicle())
    for (item in roamables) {
        item.roam()
        if (item is Animal) {
            item.eat()
        }
    }
}
```



Создание массива Roamable.

Для каждого объекта Animal в массиве вызывается функция eat().



После того как код будет обновлен, запустим его и посмотрим, что же получилось.

Тест-драйв

Запустите приложение. Когда код перебирает элементы массива `roamables`, для каждого элемента вызывается функция `roam`, но функция `eat` вызывается только для объектов `Animal`.

```
The Animal is roaming
The Hippo is eating grass
The Canine is roaming
The Wolf is eating meat
Hoooooowl!
Grunt! Grunt!
The Animal is roaming
The Hippo is eating grass
The Canine is roaming
The Wolf is eating meat
The Vehicle is roaming
```

СТАНЬ компилятором

Слева располагается код из исходного файла. Попробуйте представить себя на месте компилятора и определить, с какими фрагментами из правой части код будет компилироваться и выводить нужный результат.

Вывод:

Результат, который должна выводить программа.

```
Plane is flying
Superhero is flying
```

```
interface Flyable {
    val x: String

    fun fly() {
        println("$x is flying")
    }
}

class Bird : Flyable {
    override val x = "Bird"
}

class Plane : Flyable {
    override val x = "Plane"
}

class Superhero : Flyable {
    override val x = "Superhero"
}

fun main(args: Array<String>) {
    val f = arrayOf(Bird(), Plane(), Superhero())
    var x = 0
    while (x in 0..2) {
        [          ]
            x++
    }
}
```

Здесь размещаются фрагменты кода.

Это фрагменты кода.

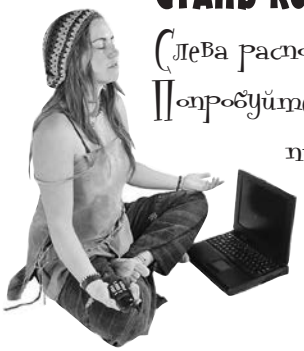
1
```
when (f[x]) {
    is Bird -> {
        x++
        f[x].fly()
    }
    is Plane, is Superhero ->
                        f[x].fly()
}
```

2
```
if (x is Plane || x is Superhero) {
    f[x].fly()
}
```

3
```
when (f[x]) {
    Plane, Superhero -> f[x].fly()
}
```

4
```
val y = when (f[x]) {
    is Bird -> false
    else -> true
}
if (y) {f[x].fly()}
```

СТАНЬ компилятором



Слева располагается код из исходного файла. Попробуйте представить себя на месте компилятора и определить, с какими фрагментами из правой части код будет компилироваться и выводить нужный результат.

Вывод:

```
Plane is flying
Superhero is flying
```

```
interface Flyable {
    val x: String

    fun fly() {
        println("$x is flying")
    }
}

class Bird : Flyable {
    override val x = "Bird"
}

class Plane : Flyable {
    override val x = "Plane"
}

class Superhero : Flyable {
    override val x = "Superhero"
}

fun main(args: Array<String>) {
    val f = arrayOf(Bird(), Plane(), Superhero())
    var x = 0
    while (x in 0..2) {
        ┌──────────────────────────────┐
        │                              │
        └──────────────────────────────┘
        x++
    }
}
```

1
```
when (f[x]) {
    is Bird -> {
        x++
        f[x].fly()
    }
    is Plane, is Superhero ->
                    f[x].fly()
}
```

Этот код компилируется и выводит правильный результат.

Не компилируется, так как переменная x относится к типу Int и не может быть преобразована в Plane или Superhero.

2
```
if (x is Plane || x is Superhero) {
    f[x].fly()
}
```

Не компилируется, потому что для проверки типа f[x] необходим оператор is.

3
```
when (f[x]) {
    Plane, Superhero -> f[x].fly()
}
```

4
```
val y = when (f[x]) {
    is Bird -> false
    else -> true
}
if (y) {f[x].fly()}
```

Код компилируется и выводит правильный результат.

Ваш инструментарий Kotlin

Глава 6 осталась позади, а ваш инструментарий пополнился абстрактными классами и интерфейсами.

Весь код для этой главы можно загрузить по адресу https://tinyurl.com/HFKotlin.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Для абстрактных классов запрещено создание экземпляров. Абстрактные классы могут содержать как абстрактные, так и неабстрактные свойства и функции.
- Любой класс, содержащий абстрактное свойство или функцию, должен быть объявлен абстрактным.
- Класс, который не является абстрактным, называется конкретным.
- Абстрактные свойства или функции реализуются переопределением.
- Все абстрактные свойства и функции должны переопределяться во всех конкретных подклассах.
- Интерфейс позволяет определить общее поведение вне иерархии суперкласса, чтобы независимые классы могли пользоваться преимуществами полиморфизма.
- Интерфейсы могут содержать как абстрактные, так и неабстрактные функции.
- Свойства интерфейсов могут быть абстрактными, а могут иметь get- и set-методы. Они не могут инициализироваться и не могут обращаться к полям данных.
- Класс может реализовать несколько интерфейсов.
- Если подкласс наследуется от суперкласса (или реализует интерфейс) с именем `A`, то код:

  ```
  super<A>.myFunction
  ```

  может использоваться для вызова реализации `myFunction`, определенной в `A`.
- Если переменная содержит ссылку на объект, оператор `is` может использоваться для проверки типа используемого объекта.
- Оператор `is` выполняет умное приведение, если компилятор может гарантировать, что используемый объект не изменяется между проверкой типа и его использованием.
- Оператор `as` выполняет явное приведение типа.
- Выражение `when` позволяет проверить значение переменной по набору вариантов.

Работа с данными

Никому не хочется тратить время и заново делать то, что уже было сделано. В большинстве приложений используются классы, предназначенные для *хранения данных*. Чтобы упростить работу, создатели Kotlin предложили концепцию **класса данных**. В этой главе вы узнаете, как классы данных помогают писать более ***элегантный и лаконичный*** код, о котором раньше можно было только мечтать. Мы рассмотрим ***вспомогательные функции*** классов данных и узнаем, как ***разложить объект данных на компоненты***. Заодно расскажем, как ***значения параметров по умолчанию*** делают код более гибким, а также познакомим вас с **Any** — предком всех суперклассов.

Оператор == вызывает функцию с именем equals

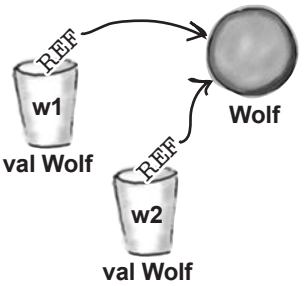
Как вы уже знаете, оператор == может использоваться для проверки равенства. Каждый раз, когда выполняется оператор ==, вызывается функция с именем `equals`. Каждый объект содержит функцию `equals`, а реализация этой функции определяет поведение оператора == .

По умолчанию функция `equals` для проверки равенства проверяет, содержат ли две переменные ссылки на один и тот же объект.

Чтобы понять принцип работы, представьте две переменные `Wolf` с именами `w1` и `w2`. Если `w1` и `w2` содержат ссылки на один объект `Wolf`, при сравнении их оператором == будет получен результат `true`:

```
val w1 = Wolf()
val w2 = w1
//w1 == w2 равно true
```

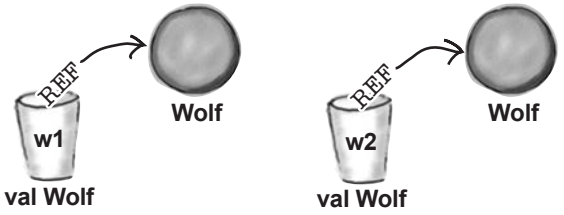
w1 и w2 ссылаются на один объект, поэтому значение w1 == w2 равно true.



Но если `w1` и `w2` содержат ссылки на разные объекты `Wolf`, сравнение их оператором == дает результат `false`, *даже если объекты содержат одинаковые значения свойств.*

```
val w1 = Wolf()
val w2 = Wolf()
//w1 == w2 is false
```

w1 и w2 ссылаются на разные объекты, поэтому значение w1 == w2 равно false.



Как говорилось ранее, в каждый объект, который вы создаете, автоматически включается функция `equals`. Но откуда берется эта функция?

equals наследуется от суперкласса Any

Каждый объект содержит функцию с именем `equals`, потому что его класс наследует функцию от класса с именем **`Any`**. Класс `Any` является предком всех классов: итоговым суперклассом *всего*. Каждый класс, который вы определяете, является подклассом `Any`, и вам не нужно указывать на это в программе. Таким образом, если вы напишете код класса с именем `myClass`, который выглядит так:

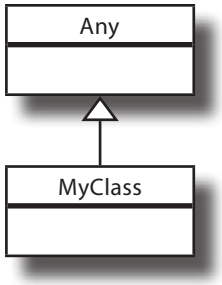
```
class MyClass {
    ...
}
```

Каждый класс является подклассом класса Any и наследует его поведение. Каждый класс ЯВЛЯЕТСЯ подклассом Any, и вам не придется сообщать об этом в программе.

Компилятор автоматически преобразует его к следующему виду:

```
class MyClass : Any() {
    ...
}
```

Компилятор незаметно делает каждый класс подклассом Any.



Важность наследования от Any

Включение `Any` как итогового суперкласса обладает двумя важными преимуществами:

- **Оно гарантирует, что каждый класс наследует общее поведение.** Класс `Any` определяет важное поведение, от которого зависит работа системы. А поскольку каждый класс является подклассом `Any`, это поведение наследуется всеми объектами, которые вы создаете. Так, класс `Any` определяет функцию с именем `equals`, а следовательно, эта функция автоматически наследуется всеми объектами.

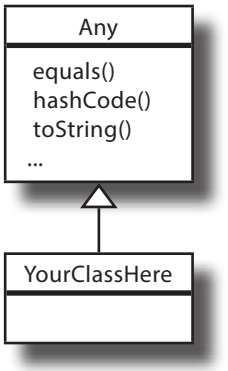
- **Оно означает, что полиморфизм может использоваться с любыми объектами.** Каждый класс является подклассом `Any`, поэтому у любого объекта, который вы создаете, класс `Any` является его итоговым супертипом. Это означает, что вы можете создать функцию с параметрами `Any` или возвращаемый тип `Any`, который будет работать с объектами любых типов. Также это означает, что вы можете создавать полиморфные массивы для хранения объектов любого типа, кодом следующего вида:

```
val myArray = arrayOf(Car(), Guitar(), Giraffe())
```

Компилятор замечает, что каждый объект в массиве имеет общий прототип Any, и поэтому создает массив типа Array<Any>.

Общее поведение, наследуемое классом `Any`, стоит рассмотреть поближе.

Общее поведение, наследуемое от Any



Класс `Any` определяет несколько функций, наследуемых каждым классом. Вот примеры основных функций и их поведения:

equals(any: Any): Boolean
Проверяет, считаются ли два объекта «равными». По умолчанию функция возвращает `true`, если используется для проверки одного объекта, или `false` — для разных объектов. За кулисами функция `equals` вызывается каждый раз, когда оператор == используется в программе.

equals возвращает false, потому что w1 и w2 содержат ссылки на разные объекты.

```
val w1 = Wolf()
val w2 = Wolf()
println(w1.equals(w2))
```

```
false
```

```
val w1 = Wolf()
val w2 = w1
println(w1.equals(w2))
```

```
true
```

equals возвращает true, потому что w1 и w2 содержат ссылки на один и тот же объект — то же самое, что проверка w1 == w2.

hashCode(): Int
Возвращает хеш-код для объекта. Хеш-коды часто используются некоторыми структурами данных для эффективного хранения и выборки значений.

```
val w = Wolf()
println(w.hashCode())
```

```
523429237
```

Значение хеш-кода w.

toString(): String
Возвращает сообщение `String`, представляющее объект. По умолчанию сообщение содержит имя класса и число, которое нас обычно не интересует.

```
val w = Wolf()
println(w.toString())
```

```
Wolf@1f32e575
```

Класс `Any` предоставляет реализацию по умолчанию для всех перечисленных функций, и эти реализации наследуются всеми классами. Однако вы можете переопределить эти реализации, чтобы изменить поведение по умолчанию всех перечисленных функций.

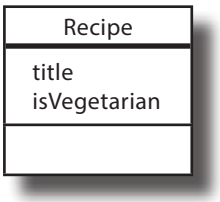
По умолчанию функция equals проверяет, являются ли два объекта одним фактическим объектом.

Функция equals определяет поведение оператора ==.

Простая проверка эквивалентности двух объектов

В некоторых ситуациях требуется изменить реализацию функции `equals`, чтобы изменить поведение оператора ==.

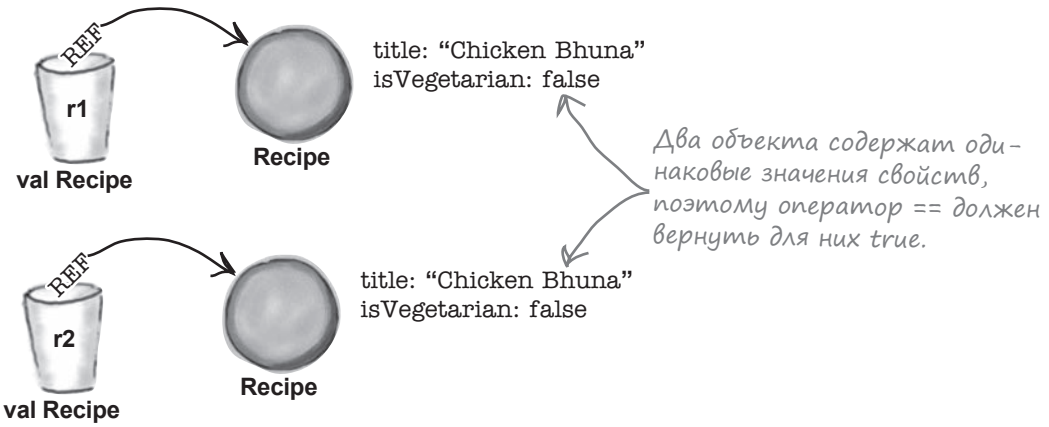
Допустим, что у вас имеется класс `Recipe`, который позволяет создавать объекты для хранения кулинарных рецептов. В такой ситуации вы, скорее всего, будете считать два объекта `Recipe` равными (или эквивалентными), если они содержат описание одного рецепта. Допустим, класс `Recipe` определяется с двумя свойствами — `title` и `isVegetarian`:




```
class Recipe(val title: String, val isVegetarian: Boolean) {
}
```

Оператор == будет возвращать результат `true`, если он используется для сравнения двух объектов `Recipe` с одинаковыми свойствами, `title` и `isVegetarian`:

```
val r1 = Recipe("Chicken Bhuna", false)
val r2 = Recipe("Chicken Bhuna", false)
```



Хотя вы *можете* изменить поведение оператора ==, написав дополнительный код для переопределения функции `equals`, разработчики Kotlin предусмотрели более удобное решение: они создали концепцию **класса данных**. Посмотрим, что собой представляют эти классы и как они создаются.

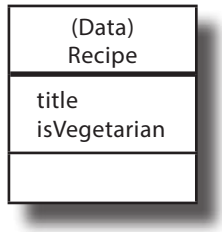
Класс данных позволяет создавать объекты данных

Классом *данных* называется класс для создания объектов, предназначенных для хранения данных. Он включает средства, полезные при работе с данными, — например, новую реализацию функции `equals`, которая проверяет, содержат ли два объекта данных одинаковые значения свойств. Если два объекта содержат одинаковые данные, то они могут считаться равными.

Чтобы определить класс данных, поставьте перед обычным определением класса ключевое слово **`data`**. Следующий код преобразует класс `Recipe`, созданный ранее, в класс данных:

Префикс data преобразует обычный класс в класс данных.

```
data class Recipe(val title: String, val isVegetarian: Boolean) {
}
```



Как создаются объекты на основе класса данных

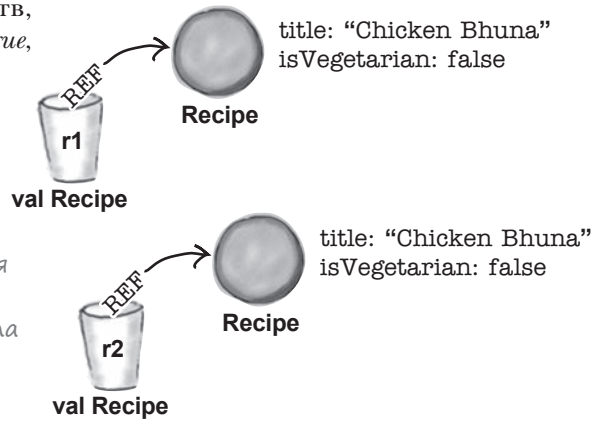
Объекты классов данных создаются так же, как и объекты обычных классов: вызовом конструктора этого класса. Например, следующий код создает новый объект данных `Recipe` и присваивает его новой переменной с именем `r1`:

```
val r1 = Recipe("Chicken Bhuna", false)
```

Классы данных автоматически переопределяют свои функции `equals` для изменения поведения оператора ==, чтобы равенство объектов проверялось **на основании значений свойств каждого объекта**. Если, например, вы создадите два объекта `Recipe` с одинаковыми значениями свойств, сравнение двух объектов оператором == даст результат *true*, потому что в них хранятся одинаковые данные:

```
val r1 = Recipe("Chicken Bhuna", false)
val r2 = Recipe("Chicken Bhuna", false)
//r1 == r2 равно true
```

r1 и r2 считаются «равными», потому что два объекта Recipe содержат одинаковые данные.



Кроме новой реализации функции `equals`, унаследованной от суперкласса `Any`, классы данных также переопределяют функции `hashCode` и `toString`. Посмотрим, как реализуются эти функции.

Объекты классов переопределяют свое унаследованное поведение

Для работы с данными классу данных необходимы объекты, поэтому он автоматически предоставляет следующие реализации для функций `equals`, `hashCode` и `toString`, унаследованных от суперкласса `Any`:

Объекты данных считаются равными, если их свойства содержат одинаковые значения.

Функция equals сравнивает значения свойств

При определении класса данных его функция `equals` (а следовательно, и оператор ==) по-прежнему возвращает `true`, если ссылки указывают на один объект. Но она также возвращает `true`, если объекты имеют одинаковые значения свойств, определенных в конструкторе:

```
val r1 = Recipe("Chicken Bhuna", false)
val r2 = Recipe("Chicken Bhuna", false)
println(r1.equals(r2))
```

```
true
```

Для равных объектов возвращаются одинаковые значения hashCode

Хеш-код — что-то вроде этикетки на корзине. Объекты считаются равными, если они лежат в одной корзине, а хеш-код сообщает системе, где их следует искать. Равные объекты ДОЛЖНЫ иметь одинаковые хеш-коды, так как от этого зависит работа системы. Подробнее об этом будет рассказано в главе 9.

Если два объекта данных считаются равными (другими словами, они имеют одинаковые значения свойств), функция `hashCode` возвращает для этих объектов одно и то же значение:

```
val r1 = Recipe("Chicken Bhuna", false)
val r2 = Recipe("Chicken Bhuna", false)
println(r1.hashCode())
println(r2.hashCode())
```

```
241131113
241131113
```

toString возвращает значения всех свойств

Наконец, функция `toString` уже не возвращает имя класса, за которым следует число, а возвращает полезную строку со значениями всех свойств, определенными в конструкторе класса данных:

```
val r1 = Recipe("Chicken Bhuna", false)
println(r1.toString())
```

```
Recipe(title=Chicken Bhuna, isVegetarian=false)
```

Кроме переопределения функций, унаследованных от суперкласса `Any`, класс данных также предоставляет дополнительные средства, которые обеспечивают более эффективную работу с данными, например, возможность копирования объектов данных. Посмотрим, как работают эти средства.

Копирование объектов данных функцией copy

Если вам потребуется создать копию объекта данных, изменяя некоторые из его свойств, но оставить другие свойства в исходном состоянии, воспользуйтесь функцией **copy**. Для этого функция вызывается для того объекта, который нужно скопировать, и ей передаются имена всех изменяемых свойств с новыми значениями.

Функция copy копирует объект данных с изменением некоторых из его свойств. Исходный объект при этом остается неизменным.

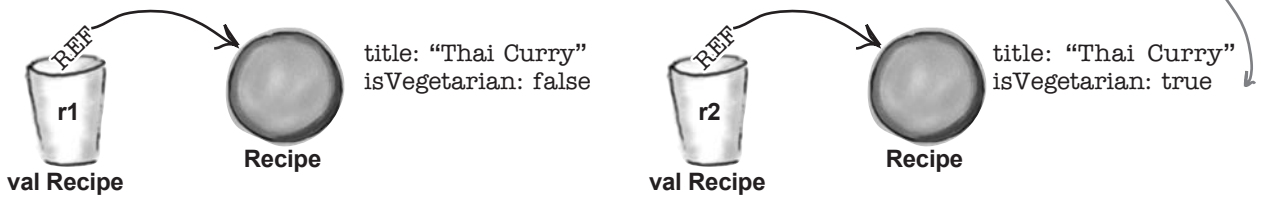
Предположим, имеется объект `Recipe` с именем `r1`, который определяется в коде примерно так:

```
val r1 = Recipe("Thai Curry", false)
```

Если вы хотите создать копию объекта `Recipe`, заменяя значение свойства `isVegetarian` на `true`, это делается так:

```
val r1 = Recipe("Thai Curry", false)
val r2 = r1.copy(isVegetarian = true)
```

Копирует объект r1 с изменением значения свойства isVegetarian на true.



По сути это означает «создать копию объекта *r1*, изменить значение его свойства *isVegetarian* на *true* и присвоить новый объект переменной с именем *r2*». При этом создается новая копия объекта, а исходный объект остается без изменений.

Кроме функции `copy`, классы данных также предоставляют набор функций для разбиения объекта данных на набор значений его свойств — этот процесс называется **деструктуризацией**. Посмотрим, как это делается.

Классы данных определяют функции componentN...

При определении класса данных компилятор автоматически добавляет в класс набор функций, которые могут использоваться как альтернативный механизм обращения к значениям свойств объекта. Эти функции известны под общим названием функций `componentN`, где N — количество извлекаемых свойств (в порядке объявления).

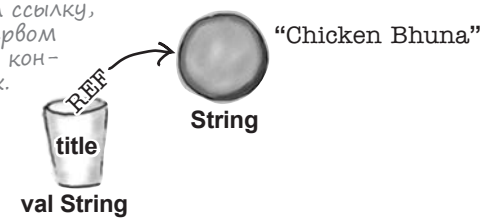
Чтобы увидеть, как работают функции `componentN`, предположим, что имеется следующий объект `Recipe`:

```
val r = Recipe("Chicken Bhuna", false)
```

Если вы хотите получить значение первого свойства объекта (свойство `title`), для этого можно вызвать функцию `component1()` объекта:

```
val title = r.component1()
```

component1() возвращает ссылку, которая содержится в первом свойстве, определенном в конструкторе класса данных.



Функция делает то же самое, что следующий код:

```
val title = r.title
```

Код с функцией получается более универсальным. Чем же именно функции `ComponentN` так полезны в классах данных?

...предназначенные для деструктуризации объектов данных

Деструктуризацией объекта данных называется разбиение его на компоненты.

Обобщенные функции `componentN` полезны прежде всего тем, что они предоставляют простой и удобный способ разбиения объекта данных на значения свойств, или его *деструктуризации*.

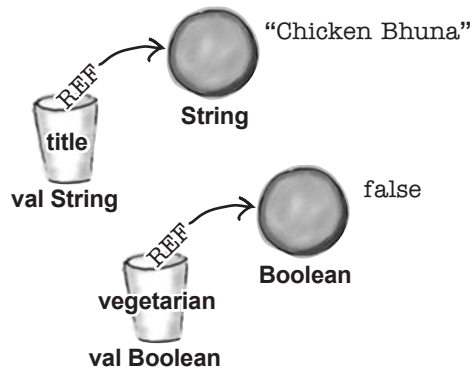
Предположим, что вы хотите взять значения свойств объекта `Recipe` и присвоить значение каждого его свойства отдельной переменной. Вместо кода

```
val title = r.title
val vegetarian = r.isVegetarian
```

с поочередной обработкой каждого свойства можно использовать следующий код:

```
val (title, vegetarian) = r
```

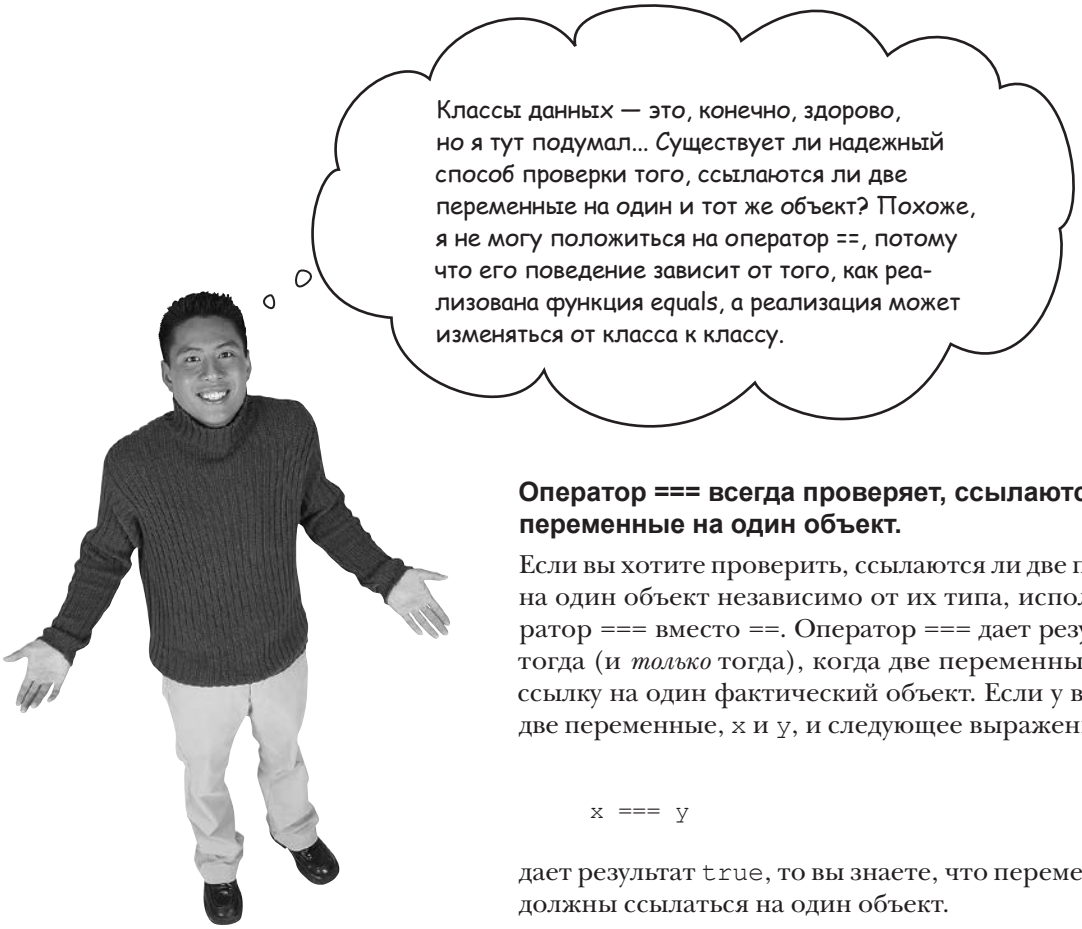
Присваивает первому свойству r значение title, а второму свойству значение vegetarian.



Этот код означает «создать две переменные, *title* и *vegetarian*, и присвоить значение одного из свойств *r* каждой переменной». Он делает то же самое, что и следующий фрагмент

```
val title = r.component1()
val vegetarian = r.component2()
```

но получается более компактным.

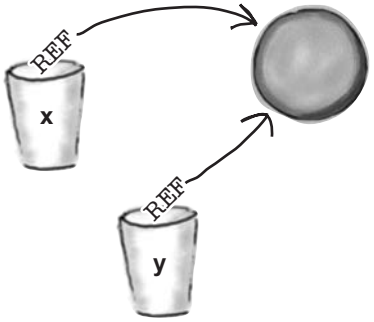


Оператор === всегда проверяет, ссылаются ли две переменные на один объект.

Если вы хотите проверить, ссылаются ли две переменные на один объект независимо от их типа, используйте оператор === вместо ==. Оператор === дает результат `true` тогда (и *только* тогда), когда две переменные содержат ссылку на один фактический объект. Если у вас имеются две переменные, `x` и `y`, и следующее выражение:

```
x === y
```

дает результат `true`, то вы знаете, что переменные `x` и `y` должны ссылаться на один объект.



== проверяет эквивалентность объектов.

=== проверяет тождественность объектов.

В отличие от оператора == , поведение оператора === не зависит от функции `equals`. Оператор === всегда ведет себя одинаково независимо от разновидности класса.

Теперь, когда вы узнали, как создавать и использовать классы данных, создадим проект для кода `Recipe`.

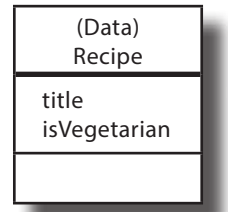
Создание проекта Recipes

Создайте новый проект Kotlin для JVM и присвойте ему имя «Recipes». Затем создайте новый файл Kotlin с именем *Recipes.kt*: выделите папку *src*, откройте меню File и выберите команду New → Kotlin File/Class. Введите имя файла «Recipes» и выберите вариант File в группе Kind.

Мы добавим в проект новый класс данных с именем `Recipe` и создадим объекты данных `Recipe`. Ниже приведен код. Обновите свою версию *Recipes.kt* и приведите ее в соответствие с нашей:

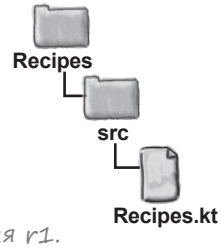
Фигурные скобки {} опущены, так как наш класс данных не имеет тела.

```
data class Recipe(val title: String, val isVegetarian: Boolean)

fun main(args: Array<String>) {
    val r1 = Recipe("Thai Curry", false)
    val r2 = Recipe("Thai Curry", false)
    val r3 = r1.copy(title = "Chicken Bhuna")
    println("r1 hash code: ${r1.hashCode()}")
    println("r2 hash code: ${r2.hashCode()}")
    println("r3 hash code: ${r3.hashCode()}")
    println("r1 toString: ${r1.toString()}")
    println("r1 == r2? ${r1 == r2}")
    println("r1 === r2? ${r1 === r2}")
    println("r1 == r3? ${r1 == r3}")
    val (title, vegetarian) = r1
    println("title is $title and vegetarian is $vegetarian")
}
```



Создание копии r1 с изменением свойства title.

Деструктуризация r1.



Тест-драйв

Когда вы запустите свой код, в окне вывода IDE отобразится следующий текст:

```
r1 hash code: -135497891
r2 hash code: -135497891
r3 hash code: 241131113
r1 toString: Recipe(title=Thai Curry, isVegetarian=false)
r1 == r2? true
r1 === r2? false
r1 == r3? false
title is Thai Curry and vegetarian is false
```

Условие r1 == r2 истинно, потому что их объекты имеют одинаковые значения. Но так как переменные содержат ссылки на разные объекты, условие r1 === r2 ложно.

Часто Задаваемые Вопросы

В: Вы сказали, что каждый класс является подклассом `Any`. Я думал, что у каждого подкласса есть только один прямой суперкласс.

О: Класс `Any` находится в корне любой иерархии суперклассов, поэтому каждый созданный вами класс является прямым или опосредованным подклассом `Any`. Это означает, что каждый класс ЯВЛЯЕТСЯ специализацией `Any` и наследует функции, определенные в этом классе: `equals`, `hashCode` и `toString`.

В: Понятно. И вы говорите, что классы данных автоматически переопределяют эти функции?

О: Да. Когда вы определяете класс данных, компилятор незаметно переопределяет унаследованные функции `equals`, `hashCode` и `toString`, чтобы они лучше подходили для объектов, предназначенных для хранения данных.

В: Можно ли переопределить эти функции без создания класса данных?

О: Да, это делается точно так же, как при переопределении функций любых других классов: вы предоставляете реализации функций в теле своего класса.

В: Существуют ли правила, которые нужно соблюдать при переопределении?

О: Главное правило: если вы переопределяете функцию `equals`, также необходимо переопределить функцию `hashCode`.

Если два объекта считаются равными, то они должны иметь одинаковые хеш-коды. Некоторые коллекции используют хеш-коды для эффективного хранения объектов, и система считает, что если два объекта равны, они также должны иметь одинаковые хеш-коды. Эта тема более подробно рассматривается в главе 9.

В: Как-то все сложно...

О: Конечно, проще создать класс данных, а при использовании этого класса код будет более элегантным и компактным. Впрочем, если вы хотите переопределить функции `equals`, `hashCode` и `toString` самостоятельно, IDE может сгенерировать большую часть кода за вас.

Чтобы среда разработки сгенерировала реализации функций equals, hashCode и toString, для начала напишите обычное определение класса вместе со всеми свойствами. Затем убедитесь в том, что курсор находится внутри класса, откройте меню Code и выберите команду Generate. Наконец, выберите функцию, для которой следует сгенерировать код.

В: Я заметил, что вы определили свойства класса данных в конструкторе с ключевым словом `val`. Можно ли определить их с ключевым словом `var`?

О: Можно, но мы настоятельно рекомендуем делать классы данных неизменяемыми, создавая только свойства `val`. В этом случае объект данных после его создания не может быть изменен, поэтому не нужно беспокоиться о том, что другой код изменит какие-либо из его свойств. Кроме того, для работы некоторых структур данных необходимо, чтобы в них входили только свойства `val`.

В: Почему классы данных включают функцию `copy`?

О: Классы данных обычно определяются со свойствами `val`, чтобы они были неизменяемыми. Наличие функции `copy` — хорошая альтернатива для изменяемых объектов данных, поскольку она позволяет легко создать другую версию объекта с измененными значениями свойств.

В: Можно ли объявить класс данных абстрактным? А открытым?

О: Нет. Классы данных не могут объявляться абстрактными или открытыми, так что класс данных не может использоваться в качестве суперкласса. Однако классы данных могут реализовать интерфейсы, а начиная с Kotlin версии 1.1, они также могут наследоваться от других классов.

Путаница с сообщениями

Ниже приведена короткая программа Kotlin. Один блок в программе пропущен. Ваша задача — сопоставить блоки-кандидаты (слева) с выводом, который вы увидите при подстановке этого блока. Используются все строки, некоторые могут задействоваться более одного раза. Проведите линию от каждого блока к подходящему варианту вывода.

```
data class Movie(val title: String, val year: String)

class Song(val title: String, val artist: String)

fun main(args: Array<String>) {
    var m1 = Movie("Black Panther", "2018")
    var m2 = Movie("Jurassic World", "2015")
    var m3 = Movie("Jurassic World", "2015")
    var s1 = Song("Love Cats", "The Cure")
    var s2 = Song("Wild Horses", "The Rolling Stones")
    var s3 = Song("Love Cats", "The Cure")

    [                                        ]

}
```

Код блока подставляется сюда.

Блоки:

```
println(m2 == m3)
```

```
println(s1 == s3)
```

```
var m4 = m1.copy()
println(m1 == m4)
```

```
var m5 = m1.copy()
println(m1 === m5)
```

```
var m6 = m2
m2 = m3
println(m3 == m6)
```

Соедините каждый блок с одним из вариантов вывода.

Варианты вывода:

```
true
```

```
false
```

Путаница с сообщениями
Решение

Ниже приведена короткая программа Kotlin. Один блок в программе пропущен. Ваша задача — сопоставить блоки-кандидаты (слева) с выводом, который вы увидите при подстановке этого блока. Используются все строки, некоторые могут задействоваться более одного раза. Проведите линию от каждого блока к подходящему варианту вывода.

```
data class Movie(val title: String, val year: String)

class Song(val title: String, val artist: String)

fun main(args: Array<String>) {
    var m1 = Movie("Black Panther", "2018")
    var m2 = Movie("Jurassic World", "2015")
    var m3 = Movie("Jurassic World", "2015")
    var s1 = Song("Love Cats", "The Cure")
    var s2 = Song("Wild Horses", "The Rolling Stones")
    var s3 = Song("Love Cats", "The Cure")

    [ Код блока подставляется сюда. ]

}
```

Условие m2 == m3 истинно, потому что m2 и m3 — объекты данных.

m4 и m1 имеют одинаковые значения свойств, поэтому условие m1 == m4 истинно.

m1 и m5 — разные объекты, поэтому условие m1 === m5 ложно.

Блоки:

```
println(m2 == m3)
```

```
println(s1 == s3)
```

```
var m4 = m1.copy()
println(m1 == m4)
```

```
var m5 = m1.copy()
println(m1 === m5)
```

```
var m6 = m2
m2 = m3
println(m3 == m6)
```

Варианты вывода:

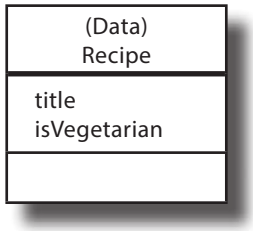
```
true
```

```
false
```

Сгенерированные функции используют только свойства, определенные в конструкторе

Пока мы показали, как определить класс данных и как добавить свойства в его конструктор. Например, следующий код определяет класс данных с именем `Recipe`, содержащий свойства `title` и `isVegetarian`:

```
data class Recipe(val title: String, val isVegetarian: Boolean) {
}
```

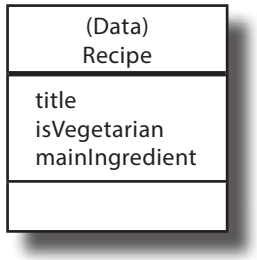


В классы данных, как и в любые другие, можно добавлять классы, включая их в тело класса. Однако здесь кроется Коварная Ловушка.

Когда компилятор генерирует реализации функций классов данных — например, при переопределении функции `equals` и создании функции `copy`, — **он включает только свойства, определенные в первичном конструкторе. Таким образом, если вы добавили в класс данных свойства, определив их в теле класса,** *эти свойства не будут включены ни в какие сгенерированные функции.*

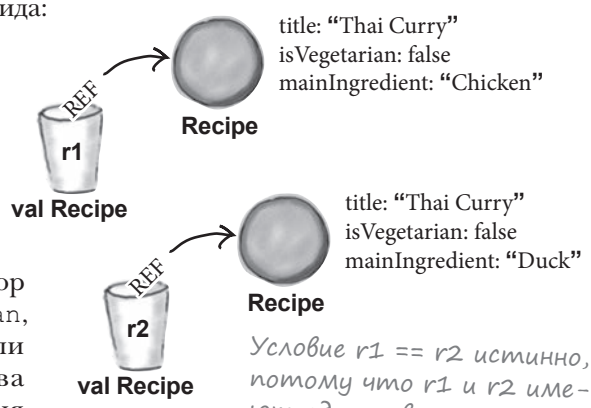
Допустим, в тело класса данных `Recipe` было добавлено новое свойство `mainIngredient`:

```
data class Recipe(val title: String, val isVegetarian: Boolean) {
    var mainIngredient = ""
}
```



Так как свойство `mainIngredient` было определено в теле класса, а не в конструкторе, оно игнорируется такими функциями, как `equals`. Это означает, что при создании двух объектов `Recipe` используется код следующего вида:

```
val r1 = Recipe("Thai curry", false)
r1.mainIngredient = "Chicken"
val r2 = Recipe("Thai curry", false)
r2.mainIngredient = "Duck"
println(r1 == r2)  // Дает результат true
```



Чтобы проверить два объекта на равенство, оператор == проверяет только свойства `title` и `isVegetarian`, определенные в конструкторе класса данных. Если два объекта имеют разные значения свойства `mainIngredient` (как в нашем примере), функция `equals` не будет проверять это свойство при проверке двух объектов на равенство.

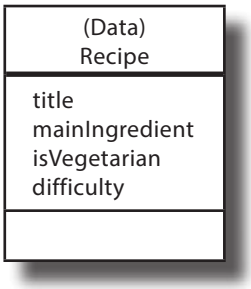
Условие r1 == r2 истинно, потому что r1 и r2 имеют одинаковые значения свойств title и isVegetarian. Оператор == игнорирует свойство mainIngredient, потому что оно не было определено в конструкторе.

А если ваш класс данных содержит много свойств, которые должны быть включены в функции, генерируемые классом данных?

Инициализация многих свойств делает код громоздким

Как вы уже узнали, свойства, которые должны быть включены в функции, генерируемые классом данных, должны определяться в его первичном конструкторе. Но если таких свойств *много*, код становится громоздким. Каждый раз, когда вы создаете новый объект, необходимо задать значения всех его свойств; таким образом, если у вас имеется класс данных `Recipe` следующего вида:

```
data class Recipe(val title: String,
                  val mainIngredient: String,
                  val isVegetarian: Boolean,
                  val difficulty: String) {
}
```



код создания объекта `Recipe` будет выглядеть так:

```
val r = Recipe("Thai curry", "Chicken",  false, "Easy")
```

Вроде бы неплохо, если ваш класс данных имеет небольшое количество свойств, но представьте, что вам приходится задавать значения 10, 20 и даже *50* свойств каждый раз, когда потребуется создать новый объект. Очень быстро код станет громоздким и неудобным.

Что же делать в подобных ситуациях?

Каждый класс данных должен иметь первичный конструктор, определяющий по крайней мере один параметр. Каждый параметр должен иметь префикс val или var.

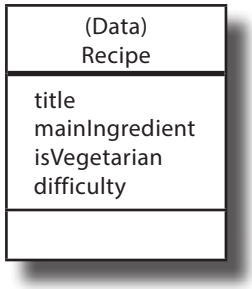
На помощь приходят значения параметров по умолчанию!

Если ваш конструктор определяет много свойств, для упрощения вызовов можно присвоить значение по умолчанию или выражение одному или нескольким определениям свойств в конструкторе. Например, вот как присваиваются значения по умолчанию свойствам `isVegetarian` и `difficulty` в конструкторе класса `Recipe`:

```
data class Recipe(val title: String,
                  val mainIngredient: String,
                  val isVegetarian: Boolean = false,
                  val difficulty: String = "Easy") {
}
```

isVegetarian имеет значение по умолчанию false.

difficulty имеет значение по умолчанию «Easy».



Посмотрим, на что влияет способ создания новых объектов `Recipe`.

Как использовать значения по умолчанию из конструкторов

Если в вашем конструкторе используются значения по умолчанию, его можно вызвать двумя способами: с передачей значений в порядке объявления и в именованных аргументах. Посмотрим, как работают эти способы.

1. Передача значений в порядке объявления

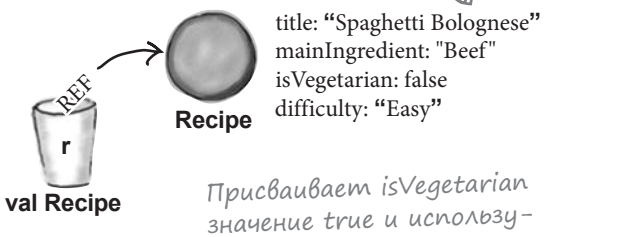
Этот способ не отличается от использовавшегося ранее, если не считать того, что вам не нужно предоставлять значения для аргументов, для которых указаны значения по умолчанию.

Допустим, вы хотите создать объект `Recipe` с названием «Spaghetti Bolognese», свойством `isVegetarian`, равным false, и свойством `difficulty`, равным «Easy». При создании этого объекта значения первых двух свойств задаются в конструкторе следующим кодом:

```
val r = Recipe("Spaghetti Bolognese", "Beef")
```

Мы не задали значения по умолчанию для свойств isVegetarian и difficulty, поэтому объект использует для этих свойств значения по умолчанию.

Этот код присваивает значения «Spaghetti Bolognese» и «Beef» свойствам `title` и `mainIngredient`. Затем для остальных свойств используются значения по умолчанию, указанные в конструкторе.



Этот способ может использоваться для переопределения значений свойств, если вы не хотите использовать значения по умолчанию. Например, если вы хотите создать объект `Recipe` для вегетарианской версии блюда (свойство `isVegetarian` равно `true`), код может выглядеть так:

```
val r = Recipe("Spaghetti Bolognese", "Tofu", true)
```

Присваивает isVegetarian значение true и использует значение по умолчанию для свойства difficulty.

Код присваивает значения «Spaghetti Bolognese», «Tofu» и *true* первым трем свойствам, определенным в конструкторе `Recipe`, а для последнего свойства `difficulty` используется значение по умолчанию «Easy».

Обратите внимание: необходимо передавать значения в порядке их объявления. Так, нельзя опустить значение свойства `isVegetarian`, если вы хотите переопределить значение следующего после него свойства `difficulty`. Например, следующий код недопустим:

```
val r = Recipe("Spaghetti Bolognese", "Beef", "Moderate")
```

Этот код не компилируется, так как компилятор ожидает, что третий аргумент относится к логическому типу.

Итак, вы увидели, как работает передача значений в порядке объявления; давайте посмотрим, как использовать именованные аргументы.

2. Именованные аргументы

Необходимо передать значение по умолчанию для каждого аргумента, которому не присвоено значение по умолчанию; в противном случае код не будет компилироваться.

Вызов конструктора с именованными аргументами позволяет явно указать, какому свойству должно быть присвоено то или иное значение, без соблюдения исходного порядка определения свойств.

Допустим, вы хотите создать объект `Recipe` с заданием значений свойств `title` и `mainIngredient`, как это делалось ранее. Чтобы создать объект с помощью именованных аргументов, используйте следующий код:

```
val r = Recipe(title = "Spaghetti Bolognese",
               mainIngredient = "Beef")
```

Задает имя каждого свойства и значение, которое ему должно быть присвоено.

Этот код присваивает значения «Spaghetti Bolognese» и «Beef» свойствам `title` и `mainIngredient`. Затем для остальных свойств используются значения по умолчанию, указанные в конструкторе.

При использовании именованных аргументов порядок их перечисления не важен. Например, следующий код делает то же, что приведенный выше, и правилен в той же степени:

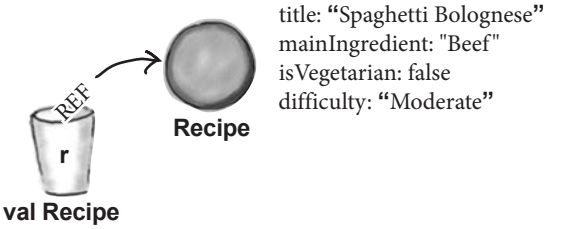
```
val r = Recipe(mainIngredient = "Beef",
               title = "Spaghetti Bolognese")
```

С именованными аргументами порядок перечисления значений свойств не важен.

У именованных аргументов есть большое преимущество: вы обязаны включать только те аргументы, которые не имеют значения по умолчанию, или имеют значения по умолчанию, которые вы хотите переопределить. Например, если вы хотите переопределить значение свойства `difficulty`, это можно сделать так:

```
val r = Recipe(title = "Spaghetti Bolognese",
               mainIngredient = "Beef",
               difficulty = "Moderate")
```

Значения параметров по умолчанию и именованные аргументы применимы не только с конструкторами классов данных; они также могут использоваться с конструкторами или функциями обычных классов. Сейчас мы ненадолго отвлечемся, а потом вернемся к использованию значений по умолчанию с функциями.



Вторичные конструкторы

Хотя вторичные конструкторы не так часто используются в Kotlin, мы решили привести небольшой обзор, чтобы вы знали, на что они похожи.

Как и в других языках (например, Java), классы в Kotlin позволяют определять один или несколько **вторичных конструкторов**. Это дополнительные конструкторы, позволяющие передавать разные комбинации параметров для создания объектов. В большинстве случаев использовать их не обязательно, так как механизм значений параметров по умолчанию обладает исключительной гибкостью.

Класс с именем `Mushroom` определяет два конструктора: первичный конструктор, определенный в заголовке класса, и вторичный конструктор, определенный в теле класса:

Первичный конструктор.

```
class Mushroom(val size: Int, val isMagic: Boolean) {
    constructor(isMagic_param: Boolean) : this(0, isMagic_param) {
        //Код, который выполняется при вызове вторичного конструктора
    }
}
```

Вторичный конструктор.

Каждый вторичный конструктор начинается с ключевого слова `constructor`, за которым следует набор параметров, используемых при вызове. Таким образом, для этого примера код:

```
constructor(isMagic_param: Boolean)
```

создает вторичный конструктор с параметром `Boolean`.

Если класс имеет первичный конструктор, то каждый вторичный конструктор должен делегировать ему управление. Например, следующий конструктор вызывает первичный конструктор класса `Mushroom` (при помощи ключевого слова `this`), передавая ему значение 0 для свойства `size`, и значение параметра `isMagic_param` для параметра `isMagic`:

Вызывает первичный конструктор текущего класса. Он передает первичному конструктору значение 0 для аргумента size и значение isMagic_param для параметра isMagic.

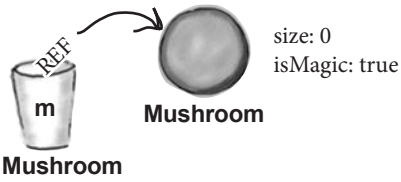
```
constructor(isMagic_param: Boolean) : this(0, isMagic_param)
```

Дополнительный код, который будет выполняться вторичным конструктором при вызове, определяется в теле вторичного конструктора:

```
constructor(isMagic_param: Boolean) : this(0, isMagic_param) {
    //Код, который выполняется при вызове вторичного конструктора
}
```

Наконец, после того как вторичный конструктор будет определен, он может использоваться для создания объектов:

```
val m = Mushroom(true)
```



Функции тоже могут использовать значения по умолчанию

Допустим, имеется функция с именем `findRecipes`, которая ищет рецепты по заданным критериям:

```
fun findRecipes(title: String,
                ingredient: String,
                isVegetarian: Boolean,
                difficulty: String) : Array<Recipe> {
    //Код поиска рецептов
}
```

Чтобы код функции успешно компилировался, при каждом вызове необходимо передавать ей значения всех четырех параметров:

```
val recipes = findRecipes("Thai curry", "", false, "")
```

Чтобы функция была более гибкой, можно присвоить каждому параметру значение по умолчанию. В этом случае код будет компилироваться даже в том случае, если вы не передаете функции значения всех четырех параметров, а только те, которые нужно переопределить:

```
fun findRecipes(title: String = "",
                ingredient: String = "",
                isVegetarian: Boolean = false,
                difficulty: String = "") : Array<Recipe> {
    //Код поиска рецептов
}
```

Та же функция, но на этот раз каждому параметру присвоено значение по умолчанию.

Если вы хотите передать функции значение «Thai curry» в параметре `title`, а в остальных случаях использовать значения по умолчанию, это можно сделать так:

```
val recipes = findRecipes("Thai curry")
```

В обоих случаях вызывается функция findRecipes с передачей значения «Thai curry» в аргументе title.

А чтобы передать значение параметра в именованном аргументе, используйте следующий вызов:

```
val recipes = findRecipes(title = "Thai curry")
```

Значения по умолчанию позволяют создавать намного более гибкие и удобные функции. Однако в некоторых случаях вместо этого нужно написать новую версию функции, для этого применяется механизм **перегрузки**.

Перегрузка функций

Перегрузка функций означает, что две и более функции имеют одинаковые имена, но разные списки аргументов.

Допустим, функция с именем `addNumbers` выглядит так:

```
fun addNumbers(a: Int, b: Int) : Int {
    return a + b
}
```

Функция получает два аргумента `Int`, поэтому ей могут передаваться только значения `Int`. Если вы хотите использовать ее для суммирования двух значений `Double`, эти значения нужно будет преобразовать в `Int` перед тем, как передать их функции.

Однако вы можете существенно упростить вызов и перегрузить функцию версией, которая получает аргументы `Double`:

```
fun addNumbers(a: Double, b: Double) : Double {
    return a + b
}
```

Это перегруженная версия тех самых функций, которые используют Double вместо Int.

Это означает, что при вызове функции `addNumbers` следующего вида:

```
addNumbers(2, 5)
```

система заметит, что параметры 2 и 5 относятся к типу `Int`, и вызовет `Int`-версию функции. Но если функция `addNumbers` вызывается в следующем формате:

```
addNumbers(1.6, 7.3)
```

то система вызовет `Double`-версию функции, поскольку оба параметра относятся к типу `Double`.

Перегруженная функция — это просто другая функция, которая имеет такое же имя, но другие аргументы. НЕ ПУТАЙТЕ перегрузку функций с переопределением — это не одно и то же.

Правила перегрузки функций:

- **Возвращаемые типы могут различаться.**
  Вы можете выбрать для перегруженной функции другой возвращаемый тип — важно, чтобы различались списки аргументов.

- **Нельзя изменить ТОЛЬКО возвращаемый тип.**
  Если две функции различаются только возвращаемыми типами, это не является допустимой перегрузкой — компилятор будет считать, что вы пытаетесь переопределить функцию. Вдобавок переопределение не будет допустимым, если только возвращаемый тип не является подтипом возвращаемого типа, объявленного в суперклассе. Чтобы перегрузить функцию, вы ОБЯЗАНЫ изменить список аргументов, хотя при этом возвращаемый тип можно заменить любым другим.

Обновление проекта Recipes

Итак, вы научились использовать значения параметров по умолчанию и перегружать функции. Давайте обновим код проекта Recipes.

Обновите свою версию кода в файле *Recipes.kt*, чтобы она соответствовала нашей (изменения выделены жирным шрифтом):

```
data class Recipe(val title: String,
                  val mainIngredient: String,
                  val isVegetarian: Boolean = false,
                  val difficulty: String = "Easy") {
}

class Mushroom(val size: Int, val isMagic: Boolean) {
    constructor(isMagic_param: Boolean) : this(0, isMagic_param) {
        //Код, выполняемый при вызове вторичного конструктора
    }
}

fun findRecipes(title: String = "",
                ingredient: String = "",
                isVegetarian: Boolean = false,
                difficulty: String = "") : Array<Recipe> {
    //Код поиска рецептов
    return arrayOf(Recipe(title, ingredient, isVegetarian, difficulty))
}

fun addNumbers(a: Int, b: Int) : Int {
    return a + b
}

fun addNumbers(a: Double, b: Double) : Double {
    return a + b
}
```

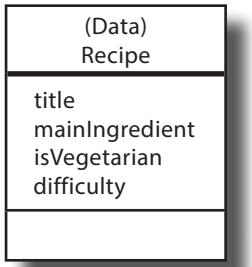
Добавляем новые свойства mainIngredient и difficulty.

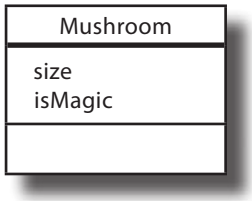
Для свойств isVegetarian и difficulty определяются значения по умолчанию.

Класс со вторичным конструктором — приведен для того, чтобы вы увидели вторичные конструкторы в действии.

Пример функции, использующей значения параметров по умолчанию.

Перегруженные функции.

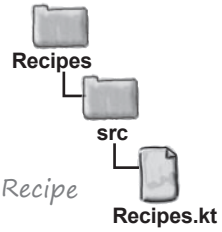




Продолжение...

Первичный конструктор Recipe изменился, поэтому необходимо изменить его вызов; в противном случае код не будет компилироваться.

```
fun main(args: Array<String>) {
    val r1 = Recipe("Thai Curry", "Chicken"~~, false~~)
    val r2 = Recipe(title = "Thai Curry", mainIngredient = "Chicken"~~, false~~)
    val r3 = r1.copy(title = "Chicken Bhuna")
    println("r1 hash code: ${r1.hashCode()}")
    println("r2 hash code: ${r2.hashCode()}")
    println("r3 hash code: ${r3.hashCode()}")
    println("r1 toString: ${r1.toString()}")
    println("r1 == r2? ${r1 == r2}")
    println("r1 === r2? ${r1 === r2}")
    println("r1 == r3? ${r1 == r3}")
    val (title, mainIngredient, vegetarian, difficulty) = r1
    println("title is $title and vegetarian is $vegetarian")

    val m1 = Mushroom(6, false)
    println("m1 size is ${m1.size} and isMagic is ${m1.isMagic}")
    val m2 = Mushroom(true)

    println("m2 size is ${m2.size} and isMagic is ${m2.isMagic}")

    println(addNumbers(2, 5))
    println(addNumbers(1.6, 7.3))
}
```



Разбираем новые свойства Recipe при деструктуризации r1.

Создание объекта Mushroom вызовом первичного конструктора.

Создание объекта Mushroom вызовом вторичного конструктора.

Вызов Int-версии addNumbers.

Вызов Double-версии addNumbers.

Тест-драйв

При выполнении этого кода в окне вывода IDE отображается следующий текст:

```
r1 hash code: 295805076
r2 hash code: 295805076
r3 hash code: 1459025056
r1 toString: Recipe(title=Thai Curry, mainIngredient=Chicken, isVegetarian=false, difficulty=Easy)
r1 == r2? true
r1 === r2? false
r1 == r3? false
title is Thai Curry and vegetarian is false
m1 size is 6 and isMagic is false
m2 size is 0 and isMagic is true
7
8.9
```

Часто Задаваемые Вопросы

В: Может ли класс данных содержать функции?

О: Да. Функции классов данных определяются точно так же, как и функции обычных классов: они добавляются в тело класса.

В: Значения параметров по умолчанию действительно настолько гибки?

О: Да! Их можно использовать в конструкторах классов (включая конструкторы классов данных) и функциях; вы даже можете определить значение параметра по умолчанию, которое представляет собой выражение. А это позволяет вам писать гибкий и при этом очень компактный код.

В: Вы сказали, что значения параметров по умолчанию в основном снимают необходимость в написании вторичных конструкторов. Существуют ли ситуации, в которых мне все равно могут понадобиться вторичные конструкторы?

О: Самая распространенная ситуация такого рода — необходимость расширения классов из фреймворков (например, Android), имеющих несколько конструкторов.

За информацией об использовании вторичных конструкторов обращайтесь к электронной документации Kotlin: *https://kotlinlang.org/docs/reference/classes.html*

В: Я хочу, чтобы программисты Java могли использовать мои классы Kotlin, но в Java нет концепции значений параметров по умолчанию. Смогу ли я использовать значения параметров по умолчанию в классах Kotlin?

О: Сможете. Просто проследите за тем, чтобы при вызове конструктора или функции Kotlin из Java в коде Java были заданы значения всех параметров, даже при наличии значения по умолчанию.

Если вы планируете часто вызывать свои конструкторы или функции Kotlin из кода Java, возможно альтернативное решение — пометить каждую функцию или конструктор, использующие значение параметра по умолчанию, аннотацией @JvmOverloads. Тем самым вы даете компилятору задачу автоматически создавать перегруженные версии, которые будет удобно вызывать из Java. Пример использования @JvmOverloads с функцией:

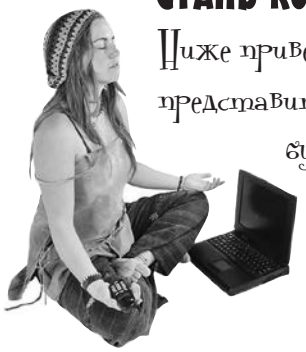
```
@JvmOverloads fun myFun(str: String = ""){
    //Здесь размещается код функции
}
```

Пример использования класса с первичным конструктором:

```
class Foo @JvmOverloads constructor(i: Int = 0){
    //Здесь размещается код функции
}
```

Обратите внимание: если вы помечаете первичный конструктор аннотацией `@JvmOverloads`, необходимо также поставить перед ним ключевое слово `constructor`. В большинстве случаев это ключевое слово не обязательно.

СТАНЬ компилятором

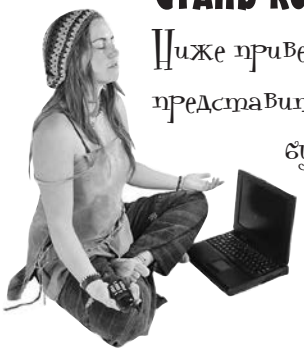


Ниже приведены два исходных файла Kotlin. Попробуйте представить себя на месте компилятора и определить, будет ли компилироваться каждый из этих файлов. Если какие-то файлы не компилируются, то как бы вы их исправили?

```
data class Student(val firstName: String, val lastName: String,
                   val house: String, val year: Int = 1)

fun main(args: Array<String>) {
    val s1 = Student("Ron", "Weasley", "Gryffindor")
    val s2 = Student("Draco", "Malfoy", house = "Slytherin")
    val s3 = s1.copy(firstName = "Fred", year = 3)
    val s4 = s3.copy(firstName = "George")

    val array = arrayOf(s1, s2, s3, s4)
    for ((firstName, lastName, house, year) in array) {
        println("$firstName $lastName is in $house year $year")
    }
}
```

```
data class Student(val firstName: String, val lastName: String,
                   val house: String, val year: Int = 1)

fun main(args: Array<String>) {
    val s1 = Student("Ron", "Weasley", "Gryffindor")
    val s2 = Student(lastName = "Malfoy", firstName = "Draco", year = 1)
    val s3 = s1.copy(firstName = "Fred")
    s3.year = 3
    val s4 = s3.copy(firstName = "George")

    val array = arrayOf(s1, s2, s3, s4)
    for (s in array) {
        println("${s.firstName} ${s.lastName} is in ${s.house} year ${s.year}")
    }
}
```

СТАНЬ компилятором. Решение



Ниже приведены два исходных файла Kotlin. Попробуйте представить себя на месте компилятора и определить, будет ли компилироваться каждый из этих файлов. Если какие-то файлы не компилируются, то как бы вы их исправили?

```
data class Student(val firstName: String, val lastName: String,
                   val house: String, val year: Int = 1)

fun main(args: Array<String>) {
    val s1 = Student("Ron", "Weasley", "Gryffindor")
    val s2 = Student("Draco", "Malfoy", house = "Slytherin")
    val s3 = s1.copy(firstName = "Fred", year = 3)
    val s4 = s3.copy(firstName = "George")

    val array = arrayOf(s1, s2, s3, s4)
    for ((firstName, lastName, house, year) in array) {
        println("$firstName $lastName is in $house year $year")
    }
}
```

Компилируется и успешно выполняется. Выводит значения свойств firstName, lastName, house и year для каждого объекта Student.

Эта строка деструктуризирует каждый объект Student в массиве.

```
data class Student(val firstName: String, val lastName: String,
                   val house: String, val year: Int = 1)

fun main(args: Array<String>) {
    val s1 = Student("Ron", "Weasley", "Gryffindor")
    val s2 = Student(lastName = "Malfoy", firstName = "Draco", year = 1, house = "Slytherin")
    val s3 = s1.copy(firstName = "Fred", year = 3)
    s3.year = 3
    val s4 = s3.copy(firstName = "George")

    val array = arrayOf(s1, s2, s3, s4)
    for (s in array) {
        println("${s.firstName} ${s.lastName} is in ${s.house} year ${s.year}")
    }
}
```

Не будет компилироваться, так как для свойства house объекта s2 должно быть задано значение, а year определяется с ключевым словом val, поэтому его значение может быть задано только при инициализации.

Ваш инструментарий Kotlin

Глава 7 осталась позади, а ваш инструментарий пополнился классами данных и значениями параметров по умолчанию.

Весь код для этой главы можно загрузить по адресу https://tinyurl.com/HFKotlin.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Поведение оператора == определяется реализацией функции `equals`.
- Каждый класс наследует функции `equals`, `hashCode` и `toString` от класса `Any`, потому что каждый класс является подклассом `Any`. Эти функции могут переопределяться.
- Функция `equals` проверяет, считаются ли два объекта «равными». По умолчанию она возвращает `true`, если используется для сравнения ссылок на один объект, или `false`, если сравниваются два разных объекта.
- Оператор === проверяет, ссылаются ли две переменные на один и тот же объект, независимо от типа объекта.
- Класс данных позволяет создавать объекты, предназначенные для хранения данных. Он автоматически переопределяет функции `equals`, `hashCode` и `toString` и включает функции `copy` и `componentN`.
- Функция `equals` классов данных проверяет равенство, сравнивая значения свойств объектов. Если два объекта данных содержат одни и те же данные, функция `equals` возвращает `true`.
- Функция `copy` создает новую копию объекта данных с изменением некоторых из свойств. Исходный объект остается неизменным.
- Функции `componentN` деструктуризируют объекты данных, разбивая их на значения отдельных свойств.
- Класс данных генерирует свои функции с учетом только тех свойств, которые были определены в его первичном конструкторе.
- Конструкторы и функции могут иметь значения параметров по умолчанию. При вызове конструктора или функции значения параметров могут передаваться по порядку объявления или с использованием именованных аргументов.
- Классы могут иметь вторичные конструкторы.
- Перегруженная функция имеет такое же имя, как и существующая. Перегруженная функция должна иметь другие аргументы; кроме того, она может иметь другой возвращаемый тип.

Правила для классов данных

* Первичный конструктор обязателен.
* Первичный конструктор должен определять один или несколько параметров.
* Каждый параметр должен быть помечен как val или var.
*Классы данных не должны быть открытыми или абстрактными.

В целости и сохранности

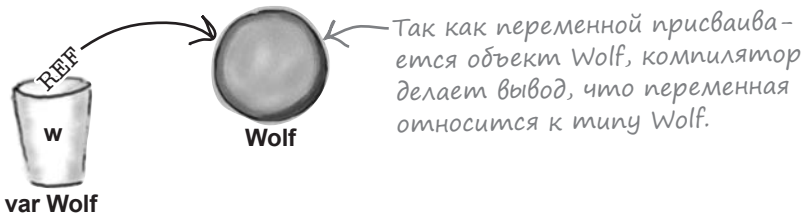
Все мечтают о безопасности кода, и, к счастью, она была заложена в основу языка Kotlin. В этой главе мы сначала покажем, что при использовании **null-совместимых типов** Kotlin вы *вряд ли когда-либо столкнетесь с исключениями NullPointerException за все время программирования на Kotlin*. Вы научитесь использовать *безопасные вызовы* и узнаете, как Элвис-оператор спасает от *всевозможных бед*. А когда мы разберемся с null, то вы сможете **выдавать и перехватывать исключения** как настоящий профессионал.

Как удалить ссылку на объект из переменной?

Вы уже знаете, что, если требуется определить новую переменную `Wolf` и присвоить ей ссылку на объект `Wolf`, это можно сделать следующей командой:

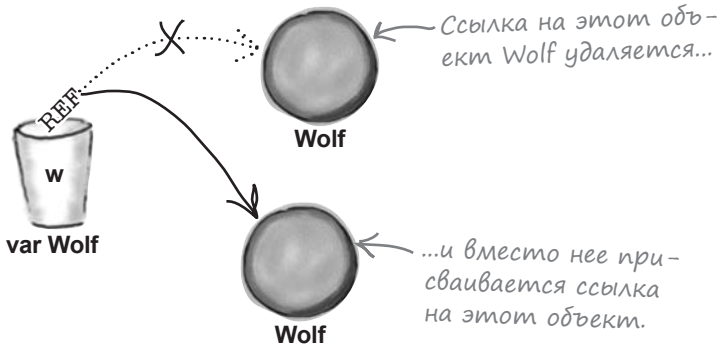
```
var w = Wolf()
```

Компилятор видит, что вы присваиваете объект `Wolf` переменной `w`, и поэтому делает вывод, что переменная должна иметь тип `Wolf`:



После того как компилятор определит тип переменной, он следит за тем, чтобы в ней хранились *только* ссылки на объекты `Wolf` и любые подклассы `Wolf`. Таким образом, если переменная определена с ключевым словом `var`, ее значение можно обновить так, чтобы в ней хранилась ссылка на другой объект `Wolf`, например:

```
w = Wolf()
```



А если вы хотите обновить переменную так, чтобы в ней не хранилась ссылка *ни на какой объект*? **Как удалить ссылку на объект из переменной после того, как вы выполнили присваивание?**

Удаление ссылки на объект с использованием null

Чтобы удалить ссылку на объект из переменной, присвойте переменной значение **`null`**:

```
w = null
```

Значение null означает, что переменная не содержит ссылку на какой-либо объект: переменная существует, но ни на что не указывает.

Но тут снова возникает Коварная Ловушка. По умолчанию *типы Kotlin не поддерживают значения null.* **Если вам нужна переменная, способная хранить null, вы должны явно указать, что ее тип является null-совместимым.**

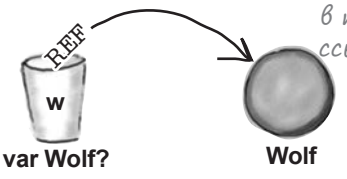
Для чего нужны null-совместимые типы?

null-совместимый тип способен хранить значения null. В отличие от других языков, Kotlin следит за значениями, которые могут быть равны null, чтобы вы не пытались выполнять с ними недопустимые операции. Выполнение недопустимых операций со значениями null — это самая распространенная причина ошибок времени выполнения в таких языках, как Java. Они могут вызвать сбой в вашем приложении. Однако в Kotlin такие проблемы встречаются редко благодаря умному использованию null-совместимых типов.

Чтобы объявить тип null-совместимым, поставьте после него вопросительный знак (`?`). Например, для создания null-совместимой переменной `Wolf` и присваивания ей нового объекта `Wolf` используется следующий код:

```
var w: Wolf? = Wolf()
```

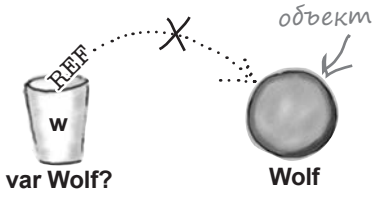
Запись Wolf? означает, что в переменной могут храниться ссылки на объекты Wolf или null.



А если вы хотите удалить ссылку на `Wolf` из переменной, используйте следующую команду:

```
w = null
```

Когда вы присваиваете w значение null, ссылка на объект Wolf удаляется.



Где же используются null-совместимые типы?

Мысл null

Когда вы присваиваете переменной `null`, происходит примерно то же, что при стирании программы на пульте дистанционного управления. У вас есть пульт (переменная), но он не связан с телевизором (объект).

Ссылка `null` содержит набор битов, представляющих «неопределенное значение», но мы не знаем и не хотим знать, что это за биты. Система автоматически делает это за нас.

Если вы попытаетесь выполнить недействительную операцию с null в Java, произойдет печально известное исключение NullPointerException. Исключение — предупреждение о том, что в программе произошло что-то особенно неприятное. Исключения будут более подробно рассмотрены позднее в этой главе.

«null-совместимым» называется тип, способный хранить значения null наряду со своим базовым типом. Например, переменная Duck? может хранить объекты Duck и null.

null-совместимые типы могут использоваться везде, где могут использоваться не-null-совместимые

Любой тип, который вы определяете, можно преобразовать в null-совместимую версию этого типа, просто добавив ? после имени. null-совместимые типы могут использоваться везде, где могут использоваться обычные (не-null-совместимые) типы:

- **При определении переменных и свойств.**
  Любая переменная или свойство могут быть null-совместимыми, но вы должны явно определить их таковыми простым объявлением их типа с ?. Компилятор не может сам определить, какой тип является null-совместимым, и по умолчанию всегда создает не-null-совместимый тип. Таким образом, если вы хотите создать null-совместимую переменную с именем `str` и присвоить ей значение «Pizza», вы должны объявить ее с типом `String?`:

```
var str: String? = "Pizza"
```

  Обратите внимание: переменные и свойства могут инициализироваться значением `null`. Например, следующий код компилирует и выводит текст "null":

```
var str: String? = null
println(str)
```

Не путайте с командой var str: String? = "". "" — объект String, не содержащий ни одного символа, тогда как null не является объектом String.

- **При определении параметров.**
  Любую функцию или параметр конструктора можно объявить с null-совместимым типом. Например, следующий код определяет функцию с именем `printInt`, которая получает параметр типа `Int?` (null-совместимый `Int`):

```
fun printInt(x: Int?) {
    println(x)
}
```

  При определении функции (или конструктора) с null-совместимым параметром при вызове функции вы все равно должны предоставить значение этого параметра, даже если это `null`. Как и в случае с не-null-совместимыми типами параметров, этот параметр нельзя пропустить при вызове, если только для него не определено значение по умолчанию.

- **При определении возвращаемых типов функций.**
  Функция может иметь null-совместимый возвращаемый тип. Например, следующая функция имеет возвращаемый тип `Long?`:

```
fun result() : Long? {
    //Код вычисляет и возвращает Long?
}
```

Функция должна возвращать значение типа Long или null.

Также можно создавать массивы null-совместимых типов. Посмотрим, как это делается.

Как создать массив null-совместимых типов

У массива null-совместимых типов элементы могут принимать значение null. Например, следующий код создает массив с именем `myArray`, в котором хранятся элементы `String?` (null-совместимые `String`):

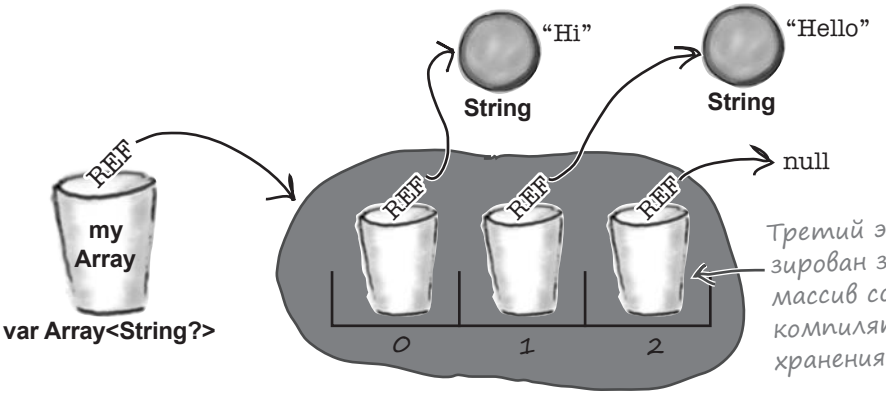
```
var myArray: Array<String?> = arrayOf("Hi", "Hello")
```

Array<String?> может хранить String и null.

Однако компилятор может прийти к выводу, что массив должен содержать null-совместимые типы, если инициализируется одним или несколькими значениями `null`. Таким образом, когда компилятор обрабатывает следующий код:

```
var myArray = arrayOf("Hi", "Hello", null)
```

он понимает, что массив может содержать сочетание `String` и `null`, и заключает, что массив должен иметь тип `Array<String?>`:



Третий элемент был инициализирован значением null. Так как массив содержит и String, и null, компилятор создает массив для хранения String?.

Вы научились определять null-совместимые типы. Теперь посмотрим, как обращаться к функциям и свойствам этих объектов.

Часто Задаваемые Вопросы

В: Что произойдет, если я инициализирую переменную значением null и предложу компилятору определить ее тип самостоятельно? Например:

```
var x = null
```

В: Компилятор видит, что переменная должна хранить значения null, но так как у него нет информации о других видах объектов, которые могут храниться в переменной, он создает переменную, способную хранить только `null`. Скорее всего, это не то, на что вы рассчитывали, поэтому если вы собираетесь инициализировать переменную значением null, обязательно укажите ее тип.

В: В предыдущей главе вы сказали, что любой объект является подклассом `Any`. Может ли переменная с типом `Any` хранить значения null?

О: Нет. Если вам нужна переменная, в которой могут храниться объекты любых типов и null, она должна иметь тип `Any?`. Пример:

```
var z: Any?
```

Как обращаться к функциям и свойствам null-совместимых типов

Допустим, имеется переменная с null-совместимым типом, и вы хотите обратиться к свойствам и функциям этого объекта. Вызывать функции или обращаться к свойствам значения null невозможно, потому что у null их нет. Чтобы избежать выполнения недействительных операций, компилятор *требует*, чтобы перед обращением к свойствам или функциям вы проверили, что значение переменной отлично от null.

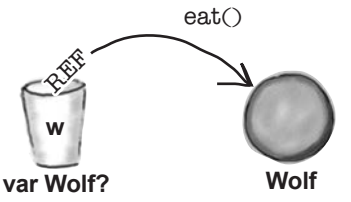
Предположим, имеется переменная `Wolf?`, которой была присвоена ссылка на новый объект `Wolf`:

```
var w: Wolf? = Wolf()
```

Чтобы получить доступ к функциям и свойствам объекта, необходимо сначала убедиться в том, что значение переменной отлично от `null`. Один из способов — проверка значения переменной внутри `if`. Например, следующий код проверяет, что значение `w` не равно `null`, после чего вызывает функцию `eat` объекта:

```
if (w != null) {
    w.eat()
}
```

Компилятор знает, что значение w отлично от null, поэтому вызов функции eat() возможен.



Этот подход может использоваться для построения более сложных выражений. Например, следующий код проверяет, что значение переменной `w` отлично от `null`, после чего вызывает для него функцию `eat`, если значение свойства `hunger` меньше 5:

```
if (w != null && w.hunger < 5) {
    w.eat()
}
```

Правая часть && выполняется только в том случае, если левая часть равна true; в данном случае компилятор знает, что переменная w не может быть равна null, и позволяет вызвать w.hunger.

Впрочем, в некоторых ситуациях и такой код не помогает. Например, если переменная `w` используется для определения `var`-свойства в классе, нельзя исключать, что между проверкой null и использованием ей было присвоено значение null, поэтому следующий код не компилируется:

```
class MyWolf {
    var w: Wolf? = Wolf()

    fun myFunction() {
        if (w != null){
            w.eat()
        }
    }
}
```

Не компилируется, потому что компилятор не может гарантировать, что другой код не обновил переменную w между проверкой на null и использованием.

К счастью, существует более безопасный подход, который помогает избегать подобных проблем.

Безопасные вызовы

?. — оператор безопасного вызова. Он позволяет безопасно обращаться к функциям и свойствам null-совместимых типов.

Для обращения к свойствам и функциям null-совместимых типов также можно воспользоваться **безопасными вызовами**. Безопасный вызов позволяет обращаться к свойствам и функциям за одну операцию без выполнения отдельной проверки null.

Приведем пример использования безопасных вызовов. Представьте, что у вас имеется свойство `Wolf?`, в котором (как и прежде) хранится ссылка на объект `Wolf`:

```
var w: Wolf? = Wolf()
```

Для выполнения безопасного вызова функции `eat` объекта `Wolf` используется следующий код:

```
w?.eat()
```

?. означает, что eat() вызывается только в том случае, если значение w отлично от null.

Функция `eat` объекта `Wolf` будет вызвана только в том случае, если значение `w` отлично от `null`. По сути такой вызов означает: «если значение *w* не равно null, вызвать *eat*».

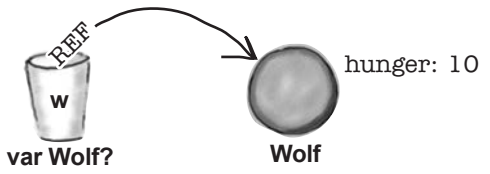
Аналогичным образом следующий код выполняет безопасное обращение к свойству `hunger` объекта `w`:

```
w?.hunger
```

Если значение `w` не равно `null`, выражение возвращает ссылку на значение свойства `hunger`. Но если значение `w` равно `null`, то вычисление всего выражения также дает результат `null`. Возможны две ситуации:

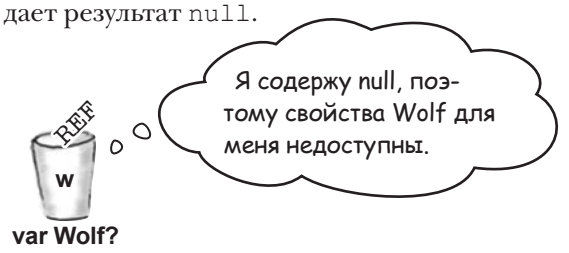
A **Ситуация A: w не содержит null.**
В переменной `w` хранится ссылка на объект `Wolf`, значение ее свойства `hunger` равно 10. Код `w?.hunger` дает значение 10.

```
w?.hunger
//Возвращает 10
```



B **Ситуация B: w содержит null.**
Переменная `w` содержит значение null вместо объекта `Wolf`, поэтому все выражение дает результат `null`.

```
w?.hunger
//Возвращает
null
```



Безопасные вызовы можно сцеплять

Другое преимущество безопасных вызовов — возможность их сцепления для построения мощных, но при этом компактных выражений.

Допустим, имеется класс с именем `MyWolf`, который содержит одно свойство `Wolf?` с именем `w`. Определение класса выглядит так:

```
class MyWolf {
    var w: Wolf? = Wolf()
}
```

Также имеется переменная `MyWolf?` с именем `myWolf`:

```
var myWolf: MyWolf? = MyWolf()
```

Если вы хотите получить значение свойства `hunger` для объекта `Wolf` из переменной `myWolf`, это можно сделать так:

```
myWolf?.w?.hunger
```

Если значение myWolf не равно null и значение w не равно null, получить hunger. В противном случае использовать null.

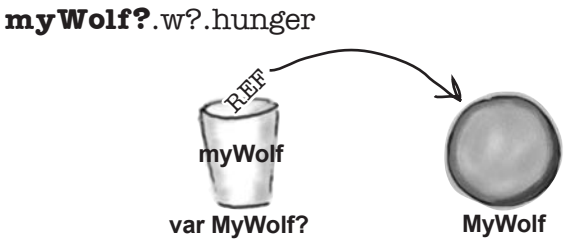
Эта запись означает: «Если *myWolf* или *w* содержит null, вернуть значение null. В противном случае вернуть значение свойства *hunger* переменной *w*». Выражение возвращает значение свойства `hunger` тогда и только тогда, когда оба значения, `myWolf` и `w`, отличны от null. Если хотя бы одно из значений, `myWolf` или `w`, равно `null`, все выражение дает результат `null`.

Как обрабатывается цепочка безопасных вызовов

Давайте посмотрим, что происходит при обработке цепочки безопасных вызовов системой:

```
myWolf?.w?.hunger
```

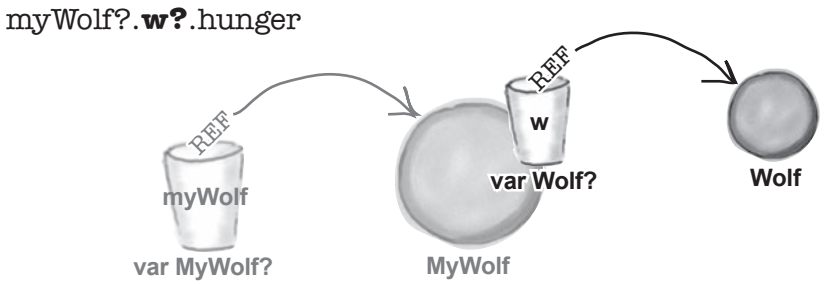
1. **Сначала система проверяет, что myWolf не содержит null.** Если `myWolf` содержит `null`, то все выражение дает результат `null`. Если значение `myWolf` отлично от `null` (как в этом примере), система переходит к следующей части выражения.



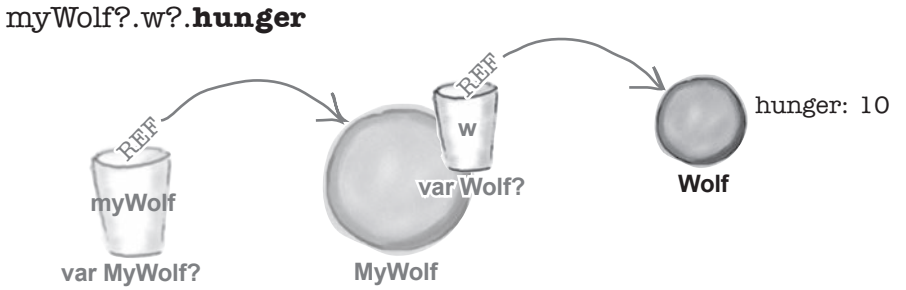
История продолжается...

2 **Затем система проверяет, что свойство w переменной myWolf не равно null.** Если значение `myWolf` не равно `null`, система переходит к следующей части выражения — `w?`.

Если значение `w` равно `null`, то все выражение дает результат `null`. Если значение `w` не равно `null`, как в нашем примере, система переходит к следующей части выражения.



3 **Если значение w не равно null, возвращается значение свойства hunger переменной w.**
Если ни переменная `myWolf`, ни ее свойство `w` не равны `null`, выражение возвращает значение свойства `hunger` переменной `w`. В данном примере выражение дает результат 10.



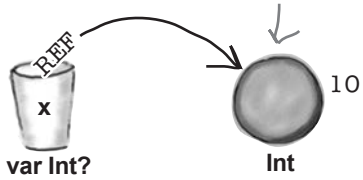
Итак, безопасные вызовы можно объединять в цепочки для получения компактных, но при этом очень мощных и безопасных выражений. Но и это еще не все!

Безопасные вызовы могут использоваться для присваивания...

Как и следовало ожидать, безопасные вызовы могут использоваться для присваивания значений переменным или свойствам. Если у вас имеется переменная `Wolf?` с именем `w`, вы можете присвоить значение ее свойства `hunger` новой переменной с именем `x`, для чего будет использоваться код следующего вида:

```
var x = w?.hunger
```

Если свойство hunger переменной w равно 10, var x = w?.hunger создает переменную Int? со значением 10.



Фактически это означает: «Если *w* содержит null, присвоить *x* значение null; в противном случае присвоить *x* значение свойства *hunger* переменной *w*». Так как выражение

```
w?.hunger
```

может давать как `Int`, так и `null`, компилятор делает вывод, что переменная `x` должна иметь тип `Int?`.

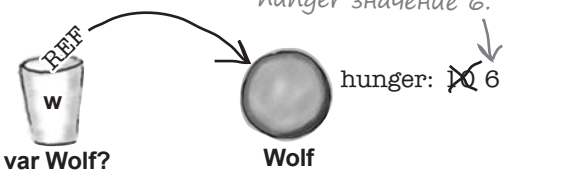
...в обе стороны

Безопасный вызов также может использоваться в левой части присваивания переменной или свойству.

Допустим, вы хотите присвоить значение 6 свойству `hunger` переменной `w` при условии, что `w` не содержит `null`. Для этого можно воспользоваться следующей командой:

```
w?.hunger = 6
```

Если переменная w не содержит null, w?.hunger = 6 присваивает ее свойству hunger значение 6.

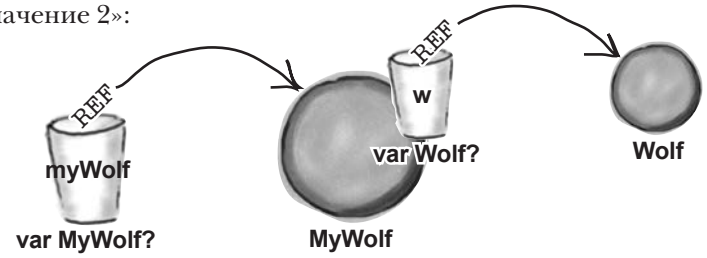


Код проверяет значение `w`, и если оно отлично от `null`, то присваивает значение 6 свойству `hunger`. Но если значение `w` равно `null`, то код ничего не делает.

Цепочки безопасных вызовов могут использоваться и в такой ситуации. Например, следующий код присваивает значение свойству `hunger` только в том случае, если оба значения, `myWolf` и `w`, отличны от `null`:

```
myWolf?.w?.hunger = 2
```

Это означает: «Если переменная *myWolf* не содержит null и свойство *hunger* переменной *w* отлично от null, присвоить свойству *hunger* переменной *w* значение 2»:



Значение 2 присваивается hunger только в том случае, если myWolf и w отличны от null.

Теперь, когда вы научились совершать безопасные вызовы с null-совместимыми типами, попробуйте выполнить следующее упражнение.

СТАНЬ компилятором

Каждый блок кода Kotlin на этой странице представляет полный исходный файл. Попробуйте представить себя на месте компилятора и определить, будет ли каждый из этих файлов компилироваться и выдавать результат, приведенный справа. Если нет, то почему?

Требуемый результат.

```
Misty: Meow!
Socks: Meow!
```

A

```
class Cat(var name: String? = "") {
    fun Meow() { println("Meow!") }
}

fun main(args: Array<String>) {
    var myCats = arrayOf(Cat("Misty"),
                         null,
                         Cat("Socks"))
    for (cat in myCats) {
        if (cat != null) {
            print("${cat.name}: ")
            cat.Meow()
        }
    }
}
```

B

```
class Cat(var name: String? = null) {
    fun Meow() { println("Meow!") }
}

fun main(args: Array<String>) {
    var myCats = arrayOf(Cat("Misty"),
                         Cat(null),
                         Cat("Socks"))
    for (cat in myCats) {
        print("${cat.name}: ")
        cat.Meow()
    }
}
```

C

```
class Cat(var name: String? = null) {
    fun Meow() { println("Meow!") }
}

fun main(args: Array<String>) {
    var myCats = arrayOf(Cat("Misty"),
                         null,
                         Cat("Socks"))
    for (cat in myCats) {
        print("${cat?.name}: ")
        cat?.Meow()
    }
}
```

D

```
class Cat(var name: String = "") {
    fun Meow() { println("Meow!") }
}

fun main(args: Array<String>) {
    var myCats = arrayOf(Cat("Misty"),
                         Cat(null),
                         Cat("Socks"))
    for (cat in myCats) {
        if (cat != null) {
            print("${cat?.name}: ")
            cat?.Meow()
        }
    }
}
```

СТАНЬ компилятором. Решение

Каждый блок кода Kotlin на этой странице представляет полный исходный файл. Попробуйте представить себя на месте компилятора и определить, будет ли каждый из этих файлов компилироваться и выдавать результат, приведенный справа. Если нет, то почему?

Требуемый результат.

```
Misty: Meow!
Socks: Meow!
```

A

```
class Cat(var name: String? = "") {
    fun Meow() { println("Meow!") }
}

fun main(args: Array<String>) {
    var myCats = arrayOf(Cat("Misty"),
                         null,
                         Cat("Socks"))
    for (cat in myCats) {
        if (cat != null) {
            print("${cat.name}: ")
            cat.Meow()
        }
    }
}
```

Компилируется и выдает правильный результат.

B

```
class Cat(var name: String? = null) {
    fun Meow() { println("Meow!") }
}

fun main(args: Array<String>) {
    var myCats = arrayOf(Cat("Misty"),
                         Cat(null),
                         Cat("Socks"))
    for (cat in myCats) {
        print("${cat.name}: ")
        cat.Meow()
    }
}
```

Компилируется, но с неправильным результатом (у второго объекта Cat свойство name содержит null).

C

```
class Cat(var name: String? = null) {
    fun Meow() { println("Meow!") }
}

fun main(args: Array<String>) {
    var myCats = arrayOf(Cat("Misty"),
                         null,
                         Cat("Socks"))
    for (cat in myCats) {
        print("${cat?.name}: ")
        cat?.Meow()
    }
}
```

Компилируется, но с неправильным результатом (для второго элемента массива myCats выводится null).

D

```
class Cat(var name: String = "") {
    fun Meow() { println("Meow!") }
}

fun main(args: Array<String>) {
    var myCats = arrayOf(Cat("Misty"),
                         Cat(null),
                         Cat("Socks"))
    for (cat in myCats) {
        if (cat != null) {
            print("${cat?.name}: ")
            cat?.Meow()
        }
    }
}
```

Не компилируется, потому что Cat не может содержать свойство name со значением null.

Использование let для выполнения кода

При использовании null-совместимых типов может оказаться, что код должен выполняться тогда и только тогда, когда некоторое значение отлично от `null`. Например, если имеется переменная `Wolf?` с именем `w`, значение свойства `hunger` переменной `w` может выводиться, если переменная `w` не равна `null`.

Один из способов выполнения подобных задач заключается в использовании такого кода:

```
if (w != null ) {
    println(w.hunger)
}
```

Например, это может случиться, если w определяет var-свойство в классе, и вы хотите использовать свойство hunger в отдельной функции. Эта ситуация уже была описана ранее в этой главе, когда мы объясняли, для чего нужны безопасные вызовы.

Но если компилятор не может гарантировать, что переменная `w` не изменится между проверкой null и ее использованием, такой код компилироваться не будет.

Альтернативное решение, которое будет работать во *всех* ситуациях, выглядит так:

```
w?.let {
    println(it.hunger)
}
```

Если w не содержит null, вывести ее свойство hunger.

Это означает: «Если переменная *w* не содержит *null*, вывести ее свойство *hunger*». Давайте разберемся в происходящем подробнее.

Ключевое слово **`let`**, используемое в сочетании с оператором безопасного вызова, сообщает компилятору, что некоторое действие должно выполняться только в том случае, если значение, к которому оно применено, не равно `null`. Таким образом, в следующей конструкции:

?.let позволяет выполнить код для значения, которое не равно null.

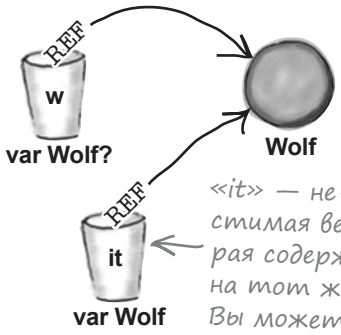
```
w?.let {
    // Код, который что-то делает
}
```

код в теле будет выполняться только в том случае, если значение `w` отлично от `null`.

После проверки того, что значение не равно `null`, вы сможете обращаться к нему в теле `let` с использованием **`it`**. Таким образом, в следующем примере `it` обозначает не-null-совместимую версию переменной `w`, к свойству `hunger` которой вы можете обратиться:

```
w?.let {
    println(it.hunger)
}
```

«it» может использоваться для прямых обращений к функциям и свойствам Wolf.



«it» — не-null-совместимая версия w, которая содержит ссылку на тот же объект Wolf. Вы можете обращаться к «it» в теле let.

Рассмотрим еще пару примеров ситуаций, в которых может пригодиться ключевое слово `let`.

Использование let с элементами массива

let также может использоваться для выполнения операций с элементами массива, отличными от null. Например, следующий код перебирает массив с элементами String? и выводит все элементы, которые не равны null:

```
var array = arrayOf("Hi", "Hello", null)
for (item in array) {
    item?.let {
        println(it)
    }
}
```

Эта строка выполняется только для элементов массива, отличных от null.

Применение let для упрощения выражений

Ключевое слово let особенно полезно в ситуациях с обработкой возвращаемых значений функций, которые могут быть равны null.

Допустим, имеется функция getAlphaWolf, которая возвращает тип Wolf?:

```
fun getAlphaWolf() : Wolf? {
    return Wolf()
}
```

Если вы хотите получить ссылку на возвращаемое значение функции и вызвать для него функцию eat, если оно отлично от null, это (в большинстве случаев) можно сделать так:

```
var alpha = getAlphaWolf()
if (alpha != null) {
    alpha.eat()
}
```

Но если переписать этот код с let, вам не придется создавать отдельную переменную для хранения возвращаемого значения функции:

```
getAlphaWolf()?.let {
    it.eat()
}
```

Вариант с ключевым словом let более компактен. Кроме того, он безопасен, поэтому его можно применять в любых ситуациях.

Это означает: «Получить объект Wolf?, и если он не равен null, вызвать для него eat».

Будьте осторожны!

Для обозначения тела let должны использоваться фигурные скобки.

Если опустить фигурные скобки { }, ваш код не будет компилироваться.

Вместо выражений if...

Другая задача, которая часто встречается с null-совместимыми типами, — выражения `if`, которые определяют альтернативное значение для чего-то, что содержит `null`.

Допустим, имеется переменная `Wolf?` с именем `w`, как и прежде, и вы хотите использовать выражение, которое возвращает значение свойства `hunger` переменной `w`, если `w` не содержит `null`, но по умолчанию использует –1, если `w` содержит `null`. В *большинстве* случаев следующее выражение будет работать:

```
if (w != null) w.hunger else -1
```

Но как и прежде, если компилятор думает, что существует вероятность обновления переменной `w` между проверкой null и ее использованием, этот код не будет компилироваться, потому что компилятор считает его небезопасным.

К счастью, существует альтернатива: «**Элвис-оператор**». ← *[Примечание от редактора: Элвис? Это что, шутка? Вернуть на доработку.]*

...можно использовать более безопасный «Элвис-оператор»

«Элвис-оператор» **`?:`** является безопасной альтернативой для выражений `if`. Почему он так называется? Потому что если положить его набок, он становится немного похож на Элвиса Пресли.

Пример выражения, использующего «Элвис-оператор»:

```
w?.hunger ?: -1
```

«Элвис-оператор» сначала проверяет значение в левой части, в данном примере:

```
w?.hunger
```

Если это значение отлично от `null`, «Элвис-оператор» возвращает его. Но если значение в левой части равно `null`, «Элвис-оператор» возвращает значение из правой части (в данном примере –1). Итак, команда

```
w?.hunger ?: -1
```

означает: «Если переменная *w* не равна null и ее свойство `hunger` не равно null, вернуть значение свойства *hunger*; в противном случае вернуть –1». Она делает то же самое, что и код:

```
if (w?.hunger != null) w.hunger else -1
```

Однако этот способ безопаснее, потому что его можно использовать где угодно.

На нескольких последних страницах мы показали, как обращаться к свойствам и функциям null-совместимых типов с использованием безопасных вызовов и как использовать `let` и «Элвис-оператор» вместо выражений и команд `if`. Однако существует еще один способ проверки значений `null`, о котором также стоит упомянуть: **оператор проверки на определенность**.

«Элвис-оператор» ?: — безопасная альтернатива для выражений if. Он возвращает значение из левой части, если оно отлично от null. В противном случае возвращается значение из правой части.

Оператор !! намеренно выдает исключение NullPointerException

Оператор проверки на определенность, или `!!`, отличается от других способов проверки null. Он не обеспечивает безопасность кода посредством проверки null, а намеренно выдает `NullPointerException`, если что-то оказывается равным `null`.

Допустим, как и прежде, имеется переменная `Wolf?` с именем `w`, и вы хотите присвоить значение ее свойства `hunger` новой переменной с именем `x`, если `w` и `hunger` не равны `null`. С оператором проверки на определенность это делается так:

```
var x = w!!.hunger
```

Здесь !! проверяет, что переменная w отлична от null.

Если `w` и `hunger` отличны от `null`, значение свойства `hunger` присваивается `x`. Но если значение `w` или `hunger` равно `null`, выдается исключение `NullPointerException`, на панели вывода IDE отображается текст, а выполнение приложения прерывается.

Сообщение, которое выводится в окне вывода, содержит информацию о `NullPointerException`, включая трассировку стека с позицией проверки на определенность, которая стала ее причиной. Например, следующий вывод сообщает, что исключение `NullPointerException` было выдано из функции `main` в строке 45 файла *App.kt*:

Исключение NullPointerException с трассировкой стека, содержащей информацию о месте возникновения ошибки.

```
Exception in thread "main" kotlin.KotlinNullPointerException
        at AppKt.main(App.kt:45)
```

Исключение произошло в строке 45.

С другой стороны, следующий вывод сообщает, что исключение `NullPointerException` было выдано из функции `myFunction` класса `MyWolf` в строке 98 файла *App.kt*. Эта функция вызывалась из функции `main` в строке 67 того же файла:

```
Exception in thread "main" kotlin.KotlinNullPointerException
        at MyWolf.myFunction(App.kt:98)
        at AppKt.main(App.kt:67)
```

Итак, проверки на определенность удобны тогда, когда вы хотите проверить некоторые условия для выявления проблем в вашем коде.

Как упоминалось выше, компилятор Kotlin предпринимает значительные усилия, чтобы обеспечить безошибочное выполнение вашего кода, однако в некоторых ситуациях бывает полезно знать, как выдавать исключения и обрабатывать их. Скоро мы рассмотрим исключения более подробно, но сначала приведем полный код нового проекта, который обрабатывает значения `null`.

Создание проекта Null Values

Создайте новый проект Kotlin для JVM и присвойте ему имя «Null Values». Создайте новый файл Kotlin с именем *App.kt*: выделите папку *src*, откройте меню File и выберите команду New → Kotlin File/Class. Введите имя файла «App» и выберите вариант File в группе Kind.

Мы добавим в проект различные классы и функции и функцию `main`, в которой они используются, чтобы вы лучше поняли, как работают значения null. Код приведен ниже — обновите свою версию *App.kt*, чтобы она соответствовала нашей:

Создание класса Wolf.

```
class Wolf {
    var hunger = 10
    val food = "meat"

    fun eat() {
        println("The Wolf is eating $food")
    }
}
```

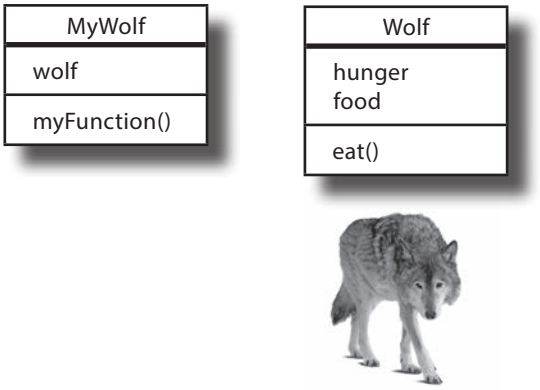
Создание класса MyWolf.

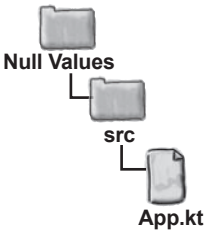
```
class MyWolf {
    var wolf: Wolf? = Wolf()

    fun myFunction() {
        wolf?.eat()
    }
}
```

Создание функции getAlphaWolf.

```
fun getAlphaWolf() : Wolf? {
    return Wolf()
}
```

Чтобы не усложнять код примера, мы используем сокращенную версию класса Wolf из предыдущих глав.

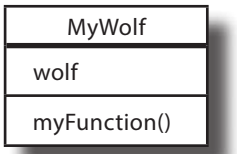


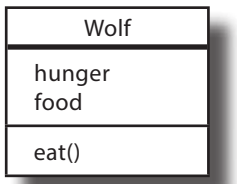


Продолжение на следующей странице.

Продолжение...

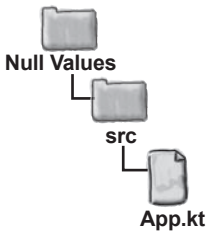
```
fun main(args: Array<String>) {
    var w: Wolf? = Wolf()

    if (w != null) {
        w.eat()
    }

    var x = w?.hunger
    println("The value of x is $x")

    var y = w?.hunger ?: -1
    println("The value of y is $y")

    var myWolf = MyWolf()
    myWolf?.wolf?.hunger = 8
    println("The value of myWolf?.wolf?.hunger is ${myWolf?.wolf?.hunger}")

    var myArray = arrayOf("Hi", "Hello", null)
    for (item in myArray) {
        item?.let { println(it) }
    }

    getAlphaWolf()?.let { it.eat() }

    w = null
    var z = w!!.hunger
}
```





Использует «Элвис-оператор» для присваивания y значения hunger, если переменная w не равна null. Если переменная w равна null, то y присваивается -1.

Выводит элементы массива, отличные от null.



Выдает исключение NullPointerException, так как значение w равно null.

Тест-драйв

При выполнении кода в окне вывода IDE отображается следующий текст:

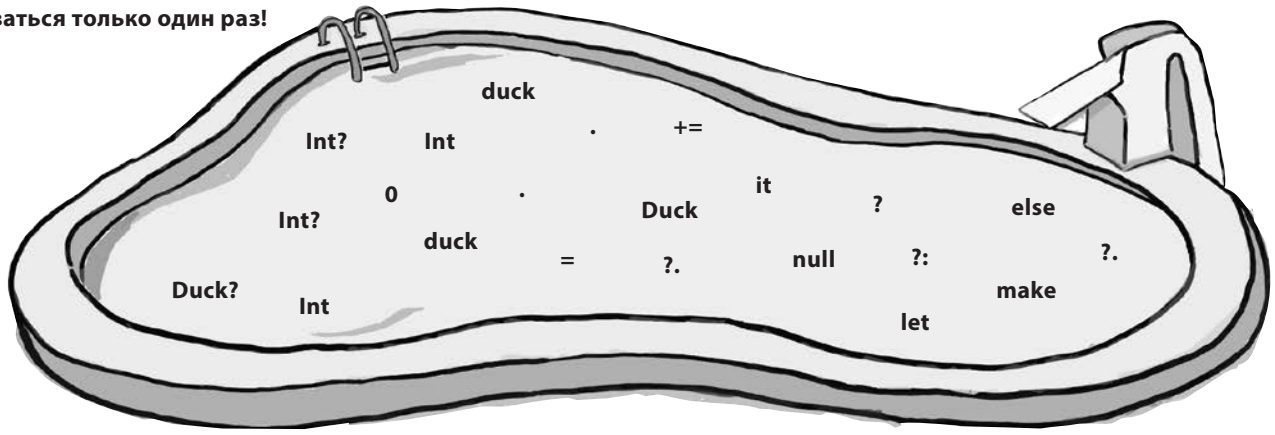
```
The Wolf is eating meat
The value of x is 10
The value of y is 10
The value of myWolf?.wolf?.hunger is 8
Hi
Hello
The Wolf is eating meat
Exception in thread "main" kotlin.KotlinNullPointerException
        at AppKt.main(App.kt:55)
```

У бассейна

Выловите из бассейна фрагменты кода и разместите их в пустых строках. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: создать два класса с именами `Duck` и `MyDucks`. Класс `MyDucks` должен содержать массив `Ducks` с null-совместимыми элементами, а также функцию `quack`, которая вызывает одноименную функцию для каждого элемента, и функцию `totalDuckHeight`, возвращающую общую высоту (`height`) всех объектов `Duck`.

```
class Duck(val height: _____ = null) {
    fun quack() {
        println("Quack! Quack!")
    }
}

class MyDucks(var myDucks: Array<_____>) {
    fun quack() {
        for (duck in myDucks) {
            _____{
                _____.quack()
            }
        }
    }

    fun totalDuckHeight(): Int {
        var h:_____ = _____
        for (duck in myDucks) {
            h _____duck_____height _____0
        }
        return h
    }
}
```

Примечание: каждый предмет из бассейна может использоваться только один раз!



У бассейна. Решение

Выловите из бассейна фрагменты кода и разместите их в пустых строках. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: создать два класса с именами `Duck` и `MyDucks`. Класс `MyDucks` должен содержать массив `Duck`s с null-совместимыми элементами, а также функцию `quack`, которая вызывает одноименную функцию для каждого элемента, и функцию `totalDuckHeight`, возвращающую общую высоту (`height`) всех объектов `Duck`.

Int?, а не Int, так как должно поддерживаться значение null.

```
class Duck(val height: Int? = null) {
    fun quack() {
        println("Quack! Quack!")
    }
}
```

myDucks — массив null-совместимых элементов Duck.

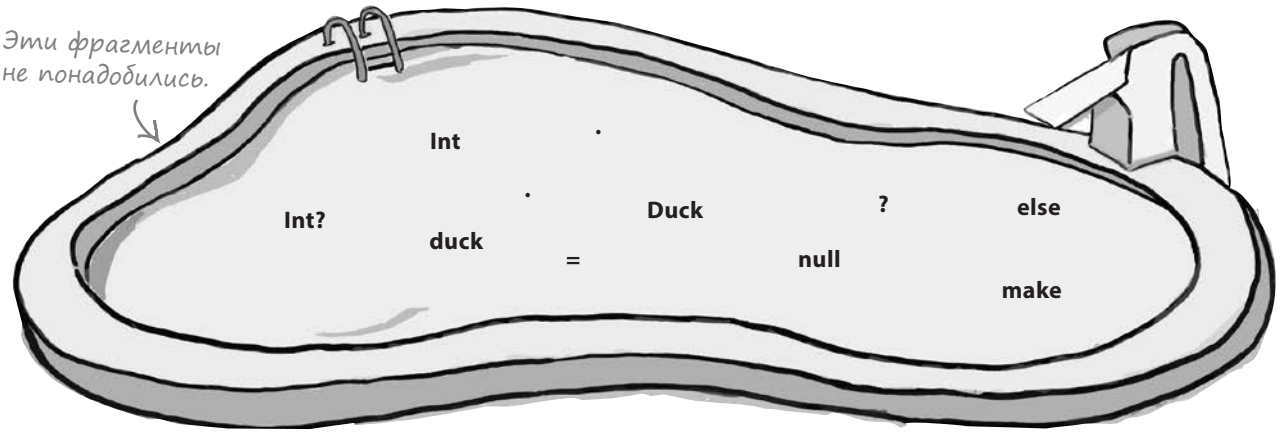
```
class MyDucks(var myDucks: Array<Duck?>) {
    fun quack() {
        for (duck in myDucks) {
            duck ?. let {
                it .quack()
            }
        }
    }

    fun totalDuckHeight(): Int {
        var h: Int = 0
        for (duck in myDucks) {
            h += duck ?. height ?: 0
        }
        return h
    }
}
```

Здесь мы используем let, чтобы вызвать quack для каждого элемента, но можно было использовать и duck?.quack().

totalDuckHeight() возвращает Int, поэтому переменная h должна иметь тип Int, а не Int?.

Если переменная duck и ее свойство height отличны от null, нужно прибавить значение height для duck к h. В противном случае h увеличивается на 0.

Эти фрагменты не понадобились.



Исключения выдаются в исключительных обстоятельствах

Как упоминалось ранее, исключение — это предупреждение об исключительных ситуациях, возникающих во время выполнения. При выдаче исключения программа говорит вам: «Произошло что-то плохое — я не знаю, что делать».

Допустим, имеется функция с именем `myFunction`, которая преобразует параметр `String` в `Int` и выводит его:

```
fun myFunction(str: String) {
    val x = str.toInt()
    println(x)
    println("myFunction has ended")
}
```

Если передать `myFunction` строку — например, «5», то программа успешно преобразует строку в `Int` и выводит значение 5 с текстом «myFunction has ended». Но если передать функции строку, которая не может быть преобразована в `Int`, — например, «I am a name, not a number», выполнение кода будет прервано, и программа выведет сообщение об исключении:

Ой!

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "I am a name, not a number"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:580)
        at java.lang.Integer.parseInt(Integer.java:615)
        at AppKt.myFunction(App.kt:119)
        at AppKt.main(App.kt:3)
```

В трассировке стека упоминается Java, потому что код выполняется в JVM.

Исключения можно перехватывать

Если в программе происходит исключение, с ним можно поступить двумя способами:

- **Оставить исключение без обработки.**
  В окне вывода отображается сообщение, а выполнение приложения прерывается (как выше).
- **Перехватить и обработать исключение.**
  Если вы знаете, что при выполнении конкретных строк кода может произойти исключение, вы можете подготовиться к этому и, возможно, устранить причины его возникновения.

Вы уже видели, что происходит с необработанными исключениями. Давайте посмотрим, как организуется их перехват.

Перехват исключений с использованием try/catch

Чтобы перехватить исключение, «упакуйте» рискованный код в блок **try/catch**. Блок `try/catch` сообщает компилятору, что вы знаете о возможности возникновения исключительной ситуации и готовы обработать ее. Компилятору неважно то, как вы ее обработаете; для него важно лишь то, что вы с ним как-то разберетесь.

Вот как выглядит блок `try/catch`:

```
fun myFunction(str: String) {

    try {
        val x = str.toInt()
        println(x)
    } catch (e: NumberFormatException) {
        println("Bummer")
    }

    println("myFunction has ended")
}
```

Это try... → **try {**

...а это catch. → **} catch (e: NumberFormatException) {**

Часть **try** блока `try/catch` содержит рискованный код, который может породить исключение. В предыдущем примере это код

```
try {
    val x = str.toInt()
    println(x)
}
```

Часть **catch** определяет перехватываемое исключение и включает код, который должен выполняться при его перехвате. Таким образом, если наш рискованный код выдает исключение `NumberFormatException`, для его перехвата и вывода содержательного сообщения может использоваться следующий код:

```
catch (e: NumberFormatException) {
    println("Bummer")
}
```

Эта строка выполняется только при перехвате исключения.

Затем выполняется код, следующий за блоком `catch`, в данном случае:

```
println("myFunction has ended")
```

finally и Выполнение операций, которые должны выполняться всегда

Если у вас есть важный завершающий код, который должен выполняться даже при возникновении исключения, поместите его в блок **`finally`**. Блок `finally` не обязателен, но он будет гарантированно выполняться при любых обстоятельствах.

Представьте, что вы собираетесь приготовить новое блюдо, и опасаетесь, что в ходе приготовления могут возникнуть проблемы.

Сначала вы включаете духовку.

Если приготовление блюда прошло успешно, ***духовку необходимо выключить.***

Если у вас ничего не вышло, ***духовку необходимо выключить.***

Духовку необходимо выключить в любом случае, поэтому код ее выключения следует разместить в блоке `finally`:

```
try {
    turnOvenOn()
    x.bake()
} catch (e: BakingException) {
    println("Baking experiment failed")
} finally {
    turnOvenOff()
}
```

Функция turnOvenOff() должна вызываться всегда, поэтому она размещается в блоке finally.

Без `finally` вызов `turnOvenOff` пришлось бы размещать *и* в `try`, *и* в `catch`, потому что ***духовка должна быть выключена в любом случае***. Блок `finally` позволяет разместить весь важный завершающий код в одном месте, а не повторять его в нескольких местах:

```
try {
    turnOvenOn()
    x.bake()
    turnOvenOff()
} catch (e: BakingException) {
    println("Baking experiment failed")
    turnOvenOff()
}
```

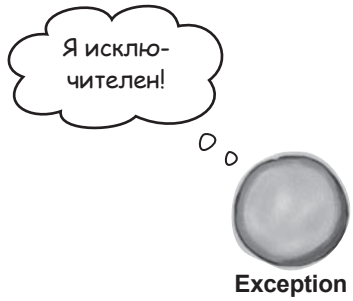
Последовательность выполнения try/catch/finally

★ Если в блоке try происходит исключение:
Управление немедленно передается в блок `catch`. Когда блок `catch` завершается, выполняется блок `finally`. Когда завершится блок `finally`, продолжается выполнение кода.

★ Если блок try завершится успешно (без исключения):
Блок `catch` пропускается, и управление передается блоку `finally`. Когда завершится блок `finally`, продолжается выполнение кода.

★ Если блок try или catch содержит команду return, блок finally все равно выполняется:
Управление передается в блок `finally`, а затем возвращается к команде `return`.

Исключение — объект типа Exception



Каждое исключение представляет собой объект типа `Exception` — суперкласса, от которого наследуют все типы исключений. Например, в JVM каждое исключение содержит функцию с именем `printStackTrace`, которая может использоваться для вывода трассировки стека в коде следующего вида:

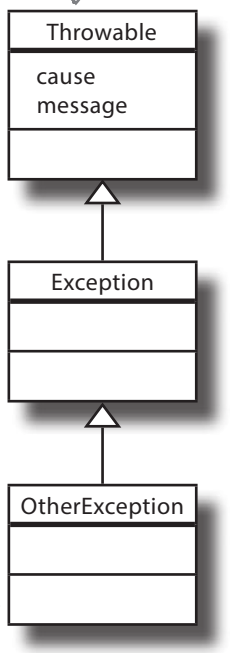
```
try {
    //Выполнить рискованную операцию
} catch (e: Exception) {
    e.printStackTrace()
    //Код, который выполняется при возникновении исключения
}
```

printStackTrace() — функция, доступная для всех исключений, выполняемых в JVM. Если вы не можете восстановить работу программы после исключения, функция printStackTrace() поможет выявить причину проблемы.

Существует много разных типов исключений, каждый из которых является подтипом `Exception`. Ниже перечислены некоторые распространенные (или известные) исключения:

- **NullPointerException**
  Выдается при попытке выполнения операций со значением null. Как говорилось ранее, исключения `NullPointerException` при работе с Kotlin почти не встречаются.
- **ClassCastException**
  Выдается при попытке преобразования объекта к неправильному типу, например, преобразования `Wolf` в `Tree`.
- **IllegalArgumentException**
  Выдается при передаче недопустимого аргумента.
- **IllegalStateException**
  Выдается в том случае, если объект обладает недопустимым состоянием.

Throwable является суперклассом Exception.



Любое исключение (и все перечисленные на этой странице) является подклассом Exception.

Вы также можете создавать собственные типы исключений, определяя новый класс с суперклассом `Exception`. Например, следующий код определяет новый тип исключения с именем `AnimalException`:

```
class AnimalException : Exception() { }
```

Определение собственных типов исключений иногда бывает полезным, если вы намеренно хотите выдавать исключения в своем коде. Мы покажем, как это делается, но сначала ненадолго отвлечемся на другую тему.

Безопасные преобразования типов под увеличительным стеклом

Как вы узнали в главе 6, каждый раз, когда используется оператор `is`, компилятор в большинстве случаев выполняет умное приведение. Например, в следующем коде компилятор проверяет, содержит ли переменная `r` объект `Wolf`, чтобы выполнить умное приведение переменной из `Roamable` в `Wolf`:

```
val r: Roamable = Wolf()
if (r is Wolf) {
    r.eat()
}
```

Здесь выполняется умное приведение r в Wolf.

В некоторых ситуациях компилятор не может выполнить умное преобразование, так как переменная может измениться между проверкой типа и использованием. Следующий код не компилируется, потому что компилятор не уверен в том, что свойство `r` все еще содержит `Wolf` после проверки:

```
class MyRoamable {
    var r: Roamable = Wolf()

    fun myFunction() {
        if (r is Wolf) {
            r.eat()
        }
    }
}
```

Не компилируется, так как компилятор не может гарантировать, что r все еще содержит ссылку на объект Wolf.

В главе 6 было показано, что эта проблема может решаться использованием ключевого слова `as` для явного преобразования `r` как объекта `Wolf`:

```
if (r is Wolf) {
    val wolf = r as Wolf
    wolf.eat()
}
```

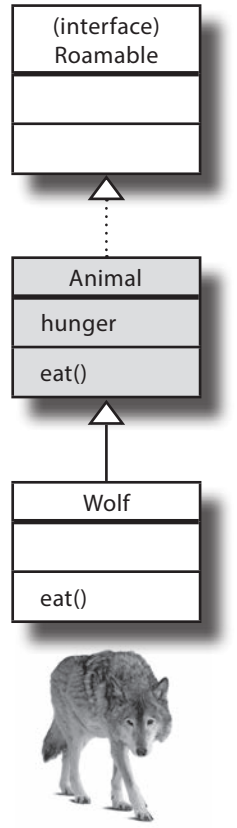
Компилируется, но если r не содержит ссылку на объект Wolf, во время выполнения произойдет исключение.

Но если между проверкой типа и приведением `r` будет присвоено значение другого типа, система выдаст исключение `ClassCastException`.

Безопасная альтернатива — выполнение **безопасного приведения** оператором **`as?`** в коде следующего вида:

```
val wolf = r as? Wolf
```

`r` преобразуется как объект `Wolf`, если переменная содержит объект этого типа; в противном случае возвращается `null`. Это избавляет вас от исключения `ClassCastException`, если ваши предположения о типе переменной окажутся ошибочными.



as? выполняет безопасное явное приведение. Если приведение завершается неудачей, возвращается null.

Намеренная выдача исключений

Иногда бывает полезно намеренно выдавать исключения в вашем коде. Скажем, если у вас имеется функция `setWorkRatePercentage`, можно выдать исключение `IllegalArgumentException`, если кто-то попытается задать процент меньше 0 или больше 100. Исключение побуждает вызывающую сторону решить проблему, а не полагаться на то, что функция как-нибудь сама разберется, что делать.

Исключения выдаются ключевым словом **`throw`**. В следующем примере функция `setWorkRatePercentage` выдает исключение `IllegalArgumentException`:

```
fun setWorkRatePercentage(x: Int) {
    if (x !in 0..100) {
        throw IllegalArgumentException("Percentage not in range 0..100: $x")
    }
    //Код, выполняемый для допустимого аргумента
}
```

Выдает исключение IllegalArgumentException, если x не принадлежит интервалу 0..100.

После этого исключение перехватывается следующим кодом:

```
try {
    setWorkRatePercentage(110)
} catch(e: IllegalArgumentException) {
    //Код обработки исключения
}
```

Функция setWorkRatePercentage() не может работать со значением 110%, поэтому вызывающая сторона должна решить проблему.

Правила для исключений

★ `catch` или `finally` не может существовать без `try`.

```
callRiskyCode()
catch (e: BadException) { }
```

Запрещено — нет try.

★ За `try` должен следовать либо блок `catch`, либо `finally`.

```
try { callRiskyCode() }
finally { }
```

Допустимо, потому что присутствует finally, даже несмотря на отсутствие catch.

★ Код не может размещаться между `try` и `catch`, или между `catch` и `finally`.

```
try { callRiskyCode() }
x = 7
catch (e: BadException) { }
```

Недопустимо — между try и catch не должно быть кода.

★ Блок `try` может иметь несколько блоков `catch`.

```
try { callRiskyCode() }
catch (e: BadException) { }
catch (e: ScaryException) { }
```

Допустимо, потому что try может иметь несколько блоков catch.

try и throw являются выражениями

В отличие от других языков (например, Java), `try` и `throw` являются *выражениями*, а следовательно, могут возвращать значения.

Использование try в качестве выражения

Возвращаемое значение `try` определяется либо последним выражением в `try`, либо последним выражением в `catch` (блок `finally`, если он есть, не влияет на возвращаемое значение). Рассмотрим следующий код:

```
val result = try { str.toInt() } catch (e: Exception) { null }
```

Означает «Попытаться присвоить result значение str.toInt(), а если не получится — присвоить result значение null».

Код создает переменную с именем `result` типа `Int?`. Блок `try` пытается преобразовать значение переменной `String` с именем `str` в `Int`. Если преобразование выполняется успешно, то значение `Int` присваивается `result`. Если же при выполнении блока `try` происходит ошибка, `result` вместо этого присваивается `null`.

Использование throw как выражения

`throw` также является выражением; например, это позволяет использовать его с «Элвис-оператором» в следующем коде:

```
val h = w?.hunger ?: throw AnimalException()
```

Если `w` и `hunger` отличны от `null`, то приведенный выше код присваивает значение свойства `hunger` переменной `w` новой переменной с именем `h`. Но если `w` или `hunger` содержит `null`, выдается исключение `AnimalException`.

Часто Задаваемые Вопросы

В: Вы сказали, что `throw` может использоваться как выражение. Означает ли это, что у `throw` есть тип? Что это за тип?

О: `throw` имеет возвращаемый тип **`Nothing`**. Это специальный тип, у которого нет значения, поэтому в переменной типа `Nothing?` может храниться только значение `null`. Например, следующий код создает переменную с именем `x` типа `Nothing?`, которая может содержать только `null`:

```
var x = null
```

В: Понятно. `Nothing` — тип, у которого нет значений. А для чего он может пригодиться?

О: `Nothing` также может использоваться для обозначения мест кода, которые должны оставаться недостижимыми. Например, `Nothing` может обозначать возвращаемый тип функции, которая не должна возвращать управление:

```
fun fail(): Nothing {
    throw BadException()
}
```

Компилятор знает, что выполнение этого кода прервется после вызова `fail()`.

В: А в Java нужно объявлять, что метод выдает исключение.

О: Все правильно, но в Kotlin это не нужно. Kotlin не различает проверяемые и непроверяемые исключения.

Возьми в руку карандаш

Взгляните на код слева. Как вы думаете, какой результат будет выведен при его выполнении? А что будет, если заменить код в строке 2 следующим кодом:

```
val test: String = "Yes"
```

Запишите свои ответы справа.

```
fun main(args: Array<String>) {
    val test: String = "No"

    try {
        println("Start try")
        riskyCode(test)
        println("End try")
    } catch (e: BadException) {
        println("Bad Exception")
    } finally {
        println("Finally")
    }

    println("End of main")
}

class BadException : Exception()

fun riskyCode(test: String) {
    println("Start risky code")

    if (test == "Yes") {
        throw BadException()
    }

    println("End risky code")
}
```

Результат для test = "No"

Результат для test = "Yes"

Ответы на с. 278.

Развлечения с магнитами

На холодильнике был выложен код Kotlin, но магниты перепутались. Удастся ли вам восстановить код, чтобы при передаче строки «Yes» функция `myFunction` выводила текст «thaws», а при передаче строки «No» выводился текст «throws».

Здесь размещаются магниты.

```
}

fun riskyCode(test:String) {

print("h")

} finally {

class BadException : Exception()

fun myFunction(test: String) {

if (test == "Yes") {

throw BadException()

print("w")

riskyCode(test)

print("t")

try {

print("a")

}

print("s")

print("o")

print("r")

}

} catch (e: BadException) {

}
```

Ответы на с. 279.

Возьми в руку карандаш
Решение

Взгляните на код слева. Как вы думаете, какой результат будет выведен при его выполнении? А что будет, если заменить код в строке 2 следующим кодом:

```
val test: String = "Yes"
```

Запишите свои ответы справа.

```
fun main(args: Array<String>) {
    val test: String = "No"

    try {
        println("Start try")
        riskyCode(test)
        println("End try")
    } catch (e: BadException) {
        println("Bad Exception")
    } finally {
        println("Finally")
    }

    println("End of main")
}

class BadException : Exception()

fun riskyCode(test: String) {
    println("Start risky code")

    if (test == "Yes") {
        throw BadException()
    }

    println("End risky code")
}
```

Результат для test = "No"

```
Start try
Start risky code
End risky code
End try
Finally
End of main
```

Результат для test = "Yes"

```
Start try
Start risky code
Bad Exception
Finally
End of main
```

Развлечения с магнитами. Решение

На холодильнике был выложен код Kotlin, но магниты перепутались. Удастся ли вам восстановить код, чтобы при передаче строки «Yes» функция `myFunction` выводила текст «thaws», а при передаче строки «No» выводился текст «throws».

```
class BadException : Exception()

fun myFunction(test: String) {
    try {
        print("t")
        riskyCode(test)
        print("o")
    } catch (e: BadException) {
        print("a")
    } finally {
        print("w")
    }
    print("s")
}

fun riskyCode(test:String) {
    print("h")
    if (test == "Yes") {
        throw BadException()
    }
    print("r")
}
```

Создание подкласса Exception.

Создание myFunction.

Пытаемся выполнить код.

Этот код выполняется при выдаче исключения BadException.

Этот код выполняется в любом случае.

Определение riskyCode.

Выдает BadException, если test == "Yes".

Ваш инструментарий Kotlin

Глава 8 осталась позади, а ваш инструментарий пополнился значениями null и исключениями.

Весь код для этой главы можно загрузить по адресу https://tinyurl.com/HFKotlin.

КЛЮЧЕВЫЕ МОМЕНТЫ

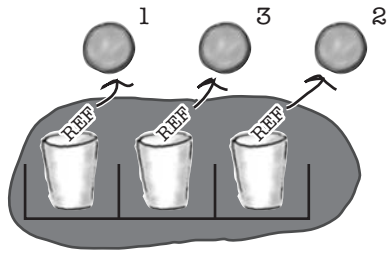
- `null` — это значение, показывающее, что переменная не содержит ссылку на объект. Переменная существует, но ни на что не ссылается.
- null-совместимый тип может хранить значения null в дополнение к базовому типу. Чтобы определить тип как null-совместимый, добавьте `?` после имени.
- Чтобы обратиться к свойству или функции null-совместимой переменной, сначала необходимо убедиться в том, что они отличны от `null`.
- Если компилятор не может гарантировать, что переменная не стала равной `null` между проверкой и использованием, к ее свойствам и функциям следует обращаться с использованием оператора безопасного вызова (`?.`).
- Безопасные вызовы могут объединяться в цепочку.
- Чтобы код выполнялся тогда и только тогда, когда значение отлично от `null`, используйте `?.let`.
- «Элвис-оператор» (`?:`) — безопасная альтернатива для выражений `if`.
- Оператор проверки на определенность (`!!`) выдает исключение `NullPointerException`, если проверяемое значение равно `null`.
- Исключение — предупреждение, которое выдается в исключительных ситуациях. Исключение представляет собой объект типа `Exception`.
- Исключения выдаются командой throw.
- Для перехвата исключений используется конструкция `try/catch/finally`.
- `try` и `throw` являются выражениями.
- Используйте безопасное приведение (`as?`), чтобы избежать исключения `ClassCastException`.

Порядок превыше всего

Хотели бы вы иметь структуру данных более гибкую, чем массив? Kotlin содержит подборку удобных **коллекций**, гибких и предоставляющих больше возможностей для управления ***хранением и управлением группами объектов***. Хотите *список с автоматически изменяемым размером, к которому можно добавлять новые элементы снова и снова?* С возможностью *сортировки, перетасовки или перестановки содержимого в обратном порядке*? Или хотите структуру данных, которая автоматически *уничтожает дубликаты* без малейших усилий с вашей стороны? Если вас заинтересовало все это (а также многое другое) — продолжайте читать.

Массивы полезны...

До настоящего момента каждый раз, когда требовалось хранить ссылки на набор объектов в одном месте, мы использовали массив. Массивы быстро создаются и обладают множеством полезных функций. Вот некоторые операции, которые можно выполнять с массивами (в зависимости от типа их элементов):



- **Создание массива:**

```
var array = arrayOf(1, 3, 2)
```

- **Создание массива, инициализированного null:**

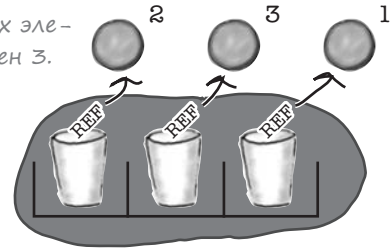
```
var nullArray: Array<String?> = arrayOfNulls(2)
```

Создает массив с размером 2, инициализированный значениями null. То же самое, что arrayOf(null, null)

- **Определение размера массива:**

```
val size = array.size
```

Массив содержит место для трех элементов, поэтому его размер равен 3.



- **Перестановка элементов в обратном порядке:**

```
array.reverse()
```

Переставляет элементы массива в обратном порядке.

- **Проверка присутствия заданного значения:**

```
val isIn = array.contains(1)
```

Массив содержит 1, поэтому функция возвращает true.

- **Вычисление суммы элементов (для числовых массивов):**

```
val sum = array.sum()
```

Возвращает 6, так как 2 + 3 + 1 = 6.

- **Вычисление среднего значения элементов (для числовых массивов):**

```
val average = array.average()
```

Возвращает Double — в данном случае (2 + 3 + 1)/3 = 2.0.

- **Поиск наименьшего или наибольшего элемента (работает для чисел, String, Char и Boolean):**

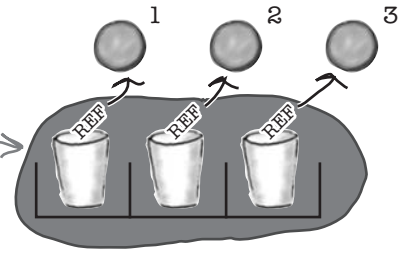
```
array.min()
array.max()
```

min() возвращает 1, так как это наименьшее значение в массиве. max() возвращает 3 – наибольшее значение.

- **Сортировка массива в естественном порядке (работает для чисел, String, Char и Boolean):**

```
array.sort()
```

Изменяет порядок элементов в массиве, чтобы они следовали от наименьшего значения к наибольшему, или от false к true.



И все же массивы не идеальны.

...но с некоторыми задачами не справляются

И хотя с массивами можно выполнять многие полезные операции, есть две важные области, для которых массивов оказывается недостаточно.

Размер массива не может изменяться динамически

При создании массива компилятор определяет его размер по количеству элементов, с которыми он инициализируется. Этот размер фиксируется раз и навсегда. Массив не увеличится, если вы захотите добавить в него новый элемент, и не уменьшится, если вы захотите элемент удалить.

Изменяемость массивов

Другое ограничение заключается в том, что после создания массива вы не можете предотвратить его модификацию. Если вы создаете массив кодом следующего вида:

```
val myArray = arrayOf(1, 2, 3)
```

ничто не помешает обновить его содержимое:

```
myArray[0] = 6
```

Если ваш код зависит от неизменяемости массива, это может стать источником ошибок в вашем приложении.

Что же делать в таких случаях?

Часто Задаваемые Вопросы

В: Можно ли удалить элемент из массива, присвоив ему `null`?

О: Если вы создали массив для хранения null-совместимых типов, вы можете присвоить одному или нескольким его элементам `null`:

```
val a: Array<Int?> = arrayOf(1, 2, 3)
a[2] = null
```

Однако размер массива при этом не изменится. В приведенном примере размер массива остается равным 3 при том, что одному из его элементов было присвоено значение `null`.

В: Можно ли создать копию массива с другим размером?

О: Можно. Массивы даже содержат функцию `plus`, которая упрощает эту операцию; `plus` копирует массив и добавляет в конец копии новый элемент. Тем не менее размер исходного массива при этом не изменится.

В: А это создает проблемы?

О: Да. Вам придется писать лишний код, а если другие переменные содержат ссылки на старую версию массива, в программе могут возникнуть ошибки.

Тем не менее у массивов существуют хорошие альтернативы, которые будут рассмотрены ниже.

Не уверены — обращайтесь в библиотеку

Kotlin поставляется с сотнями готовых классов и функций. Некоторые из них вам уже встречались — как, например, `String` и `Any`. И к счастью, **стандартная библиотека Kotlin** содержит классы, которые могут стать отличной альтернативой для массивов.

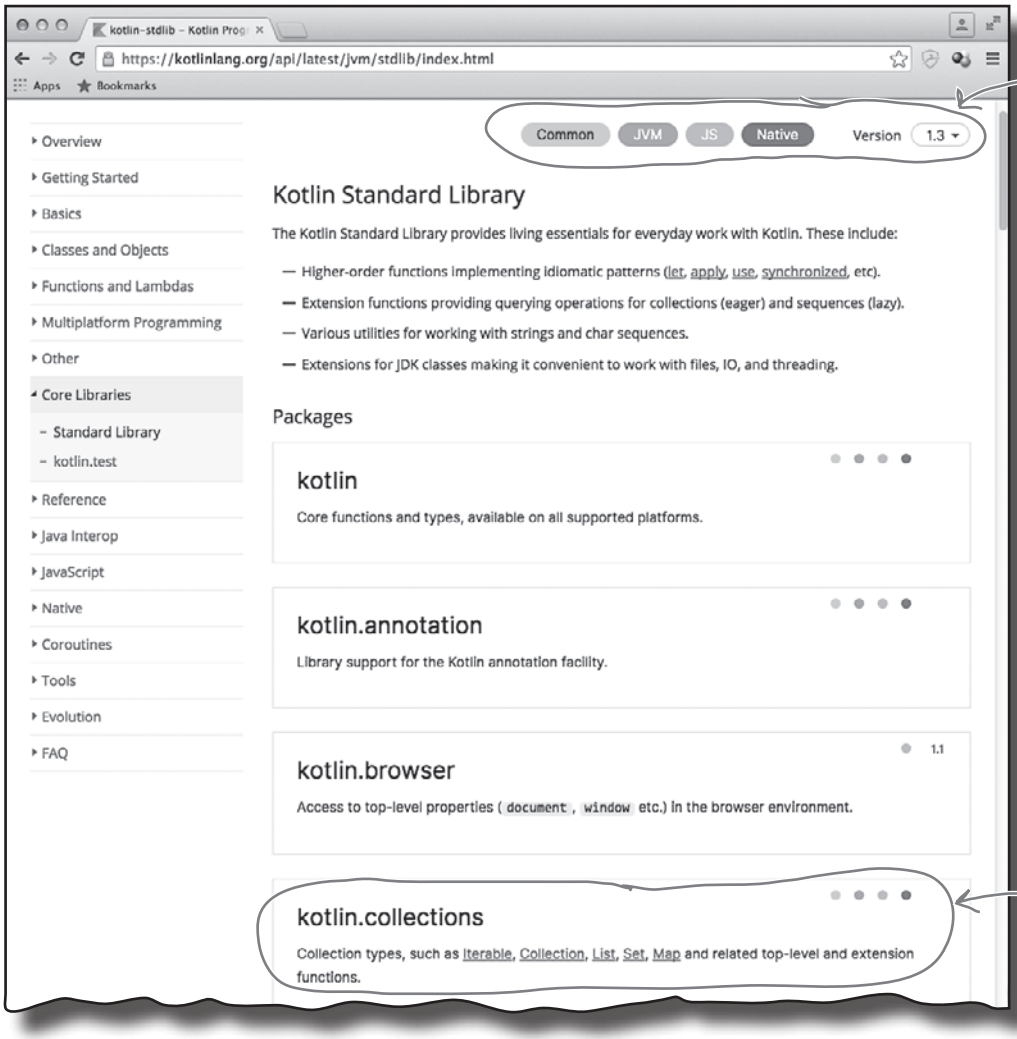
В стандартной библиотеке Kotlin классы и функции объединяются в **пакеты**. Каждый класс принадлежит определенному пакету, и у каждого пакета есть имя. Например, пакет *kotlin* содержит основные функции и типы, а пакет *kotlin.math* — математические функции и константы.

Пакет, который нас сейчас интересует, называется *kotlin.collections*. Этот пакет включает подборку классов, которая позволяет объединять объекты в **коллекции**. Давайте познакомимся с основными разновидностями коллекций.

Стандартная библиотека

За информацией о классах и функциях, входящих в стандартную библиотеку Kotlin, обращайтесь по адресу

https://kotlinlang.org/api/latest/jvm/stdlib/index.html



Вы можете использовать эти фильтры для отображения только тех коллекций, которые актуальны для конкретной платформы или версии Kotlin.

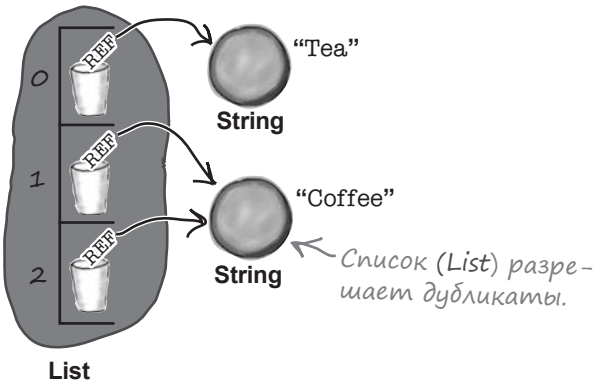
Пакет kotlin.collections из стандартной библиотеки Kotlin.

List, Set и Map

В Kotlin существуют три основных типа коллекций — **`List`**, **`Set`** и **`Map`**. Каждый тип имеет четко определенное предназначение:

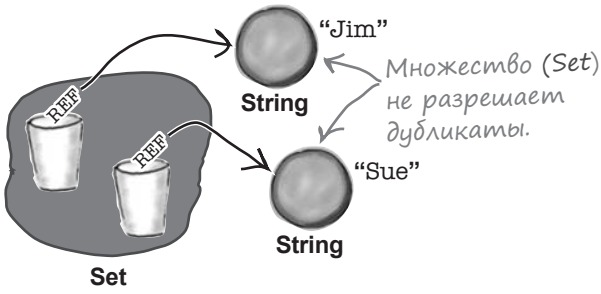
List — когда важен порядок

`List` хранит и отслеживает позицию элементов. Она знает, в какой позиции списка находится тот или иной элемент, и несколько элементов могут содержать ссылки на один объект.



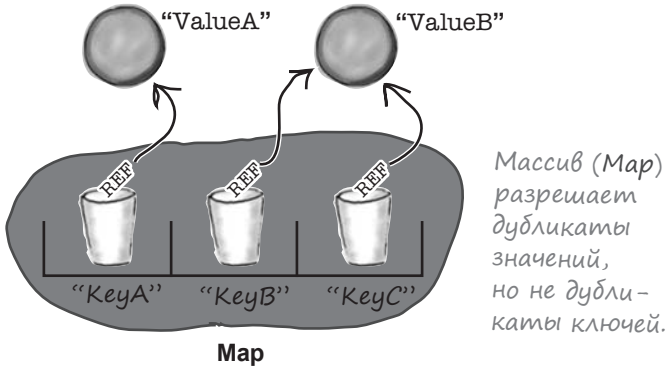
Set — когда важна уникальность

`Set` не разрешает дубликаты и не отслеживает порядок, в котором хранятся значения. Коллекция не может содержать несколько элементов, ссылающихся на один и тот же объект, или несколько элементов, ссылающихся на два объекта, которые считаются равными.



Map — когда важен поиск по ключу

`Map` использует пары «ключ-значение». Этот тип коллекции знает, какое значение связано с заданным ключом. Два ключа могут ссылаться на один объект, но дубликаты ключей невозможны. Хотя ключи обычно представляют собой строковые имена (например, для составления списков свойств в формате «имя-значение»), ключом также может быть произвольный объект.



Простые коллекции `List`, `Set` и `Map` *неизменяемы*; это означает, что после инициализации коллекции вы уже не сможете добавлять или удалять элементы. Если вам необходимо добавлять или удалять элементы, Kotlin предоставляет изменяемые версии подклассов: **`MutableList`**, **`MutableSet`** и **`MutableMap`**. Например, если вы хотите пользоваться всеми преимуществами `List` и при этом иметь возможность обновлять его содержимое, выберите `MutableList`.

Итак, теперь вы знаете три основных типа коллекций из стандартной библиотеки Kotlin. Посмотрим, как использовать каждый из них. Начнем с `List`.

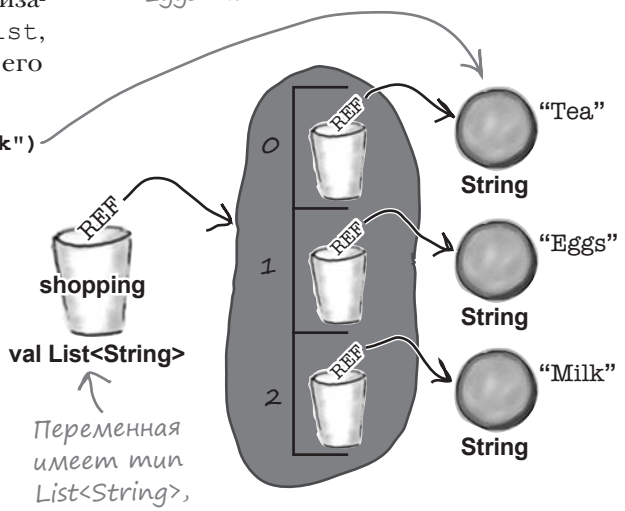
Эти невероятные списки...

Создание списка **List** имеет много общего с созданием массива: в программе вызывается функция с именем **listOf**, которой передаются значения для инициализации элементов. Например, следующий код создает List, инициализирует его тремя строками и присваивает его новой переменной с именем shopping:

```
val shopping = listOf("Tea", "Eggs", "Milk")
```

Компилятор определяет тип объекта, который должен содержаться в списке List, по типам всех значений, переданных при создании. Например, список в нашем примере инициализируется тремя строками, поэтому компилятор создает List с типом List<String>. Тип List также можно задать явно:

```
val shopping: List<String>
shopping = listOf("Tea", "Eggs", "Milk")
```



...и как ими пользоваться

После того как объект List будет создан, вы сможете обращаться к содержащимся в нем элементам функцией **get**. Например, следующий код проверяет, что размер List больше 0, после чего выводит элемент с индексом 0:

```
if (shopping.size > 0) {
    println(shopping.get(0))
    //Выводит "Tea"
}
```

Размер List желательно проверять заранее, так как при передаче недействительного индекса get() выдаст исключение ArrayIndexOutOfBoundsException.

Перебор всех элементов List выполняется следующим кодом:

```
for (item in shopping) println (item)
```

Вы также можете проверить, содержит ли List ссылку на конкретный объект, и получить индекс соответствующего элемента:

```
if (shopping.contains("Milk")) {
    println(shopping.indexOf("Milk"))
    //Выводит 2
}
```

Как видите, в программах списки List используются практически так же, как и массивы. Но между List и массивами также существует серьезное различие: списки List неизменяемы — хранящиеся в них ссылки невозможно обновить.

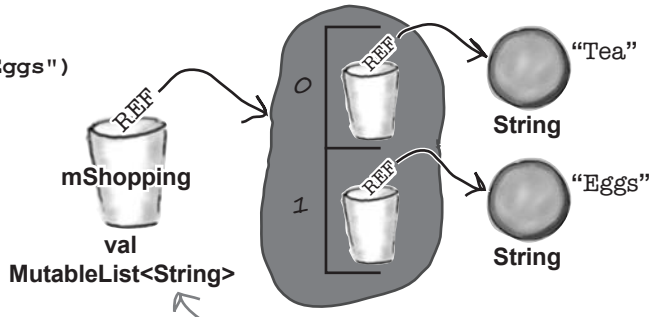
В List и других коллекциях могут храниться ссылки на объекты любых типов: String, Int, Duck, Pizza и т. д.

Создайте объект MutableList...

Если вам нужен список с возможностью обновления элементов, используйте **MutableList**. Объект MutableList определяется почти так же, как определяется List, но только в этом случае используется функция **mutableListOf**:

```
val mShopping = mutableListOf("Tea", "Eggs")
```

MutableList является подклассом List, поэтому для MutableList можно вызывать те же функции, что и для List. Однако у MutableList есть дополнительные функции, которые используются для добавления, удаления, обновления или перестановки существующих значений.



Если передать функции mutableListOf() строковые значения, компилятор определяет, что вам нужен объект типа MutableList<String> (MutableList для хранения String).

...и добавьте в него значения

Новые элементы добавляются в MutableList функцией **add**. Чтобы добавить новое значение в конец MutableList, передайте значение функции add в единственном параметре. Например, следующий код добавляет строку «Milk» в конец mShopping:

```
mShopping.add("Milk")
```

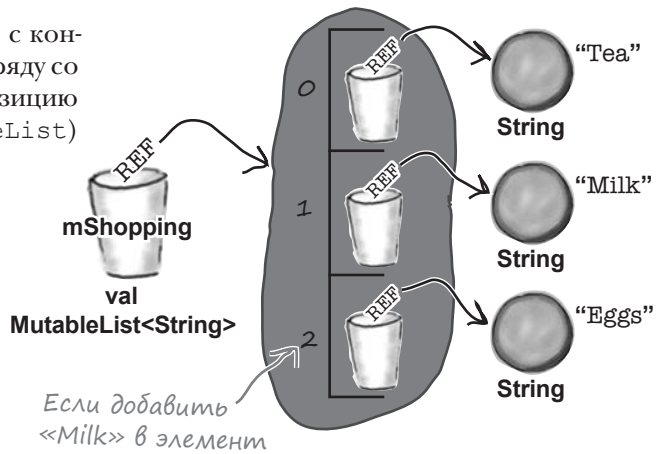
Размер MutableList увеличивается, чтобы в списке хранились три значения вместо двух.

Если же вы хотите вставить значение в позицию с конкретным индексом, передайте индекс функции наряду со значением. Например, вставка значения «Milk» в позицию с индексом 1 (вместо добавления в конец MutableList) выполняется так:

```
mShopping.add(1, "Milk")
```

При вставке значения с конкретным индексом другие значения сдвигаются и освобождают для него место. Так, в нашем примере значение «Eggs» перемещается из позиции с индексом 1 в позицию с индексом 2, чтобы значение «Milk» можно было вставить в позиции с индексом 1.

Кроме добавления значений в MutableList, элементы также можно удалять и заменять. Давайте посмотрим, как это делается.



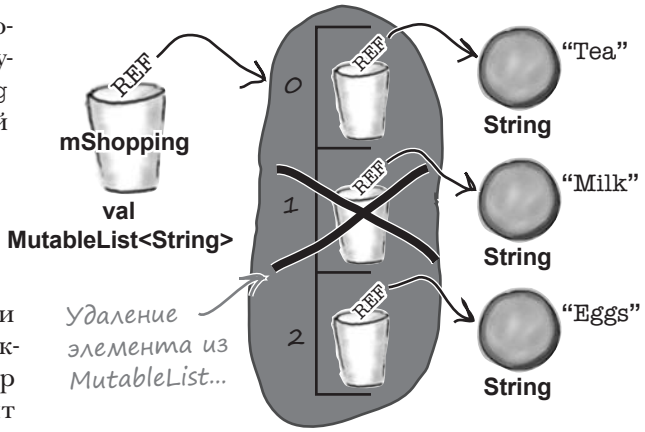
Если добавить «Milk» в элемент с индексом 1, то «Eggs» переходит на индекс 2, чтобы освободить место для нового значения.

Значения можно удалять...

Существуют два способа удаления значений из `MutableList`.

В первом способе вызывается функция **`remove`**, которой передается удаляемое значение. Например, следующий код проверяет, содержит ли список `mShopping` строку «Milk», после чего удаляет соответствующий элемент:

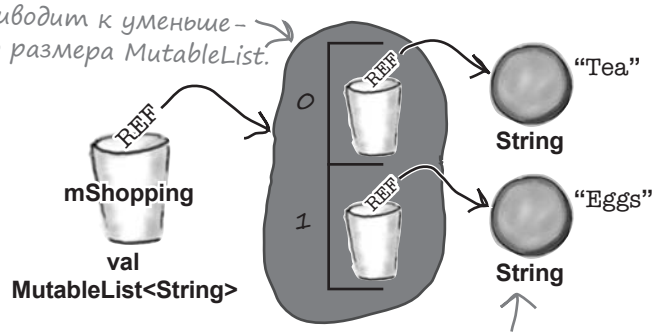
```
if (mShopping.contains("Milk")) {
    mShopping.remove("Milk")
}
```



Второй способ основан на использовании функции **`removeAt`** для удаления значения с заданным индексом. Например, следующий код проверяет, что размер `mShopping` больше 1, после чего удаляет элемент с индексом 1:

```
if (mShopping.size > 1) {
    mShopping.removeAt(1)
}
```

...приводит к уменьшению размера MutableList.



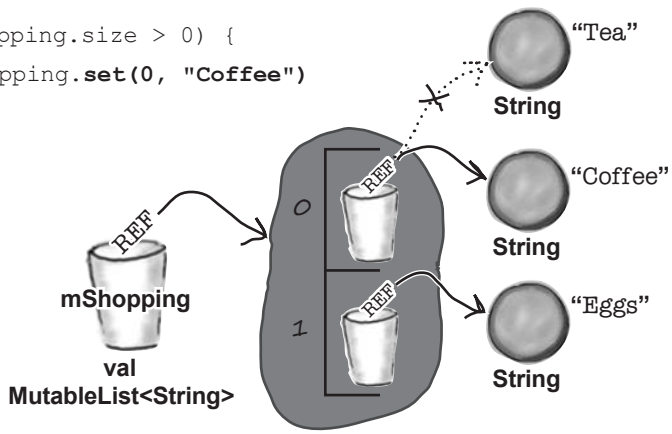
После удаления значения «Milk» элемент «Eggs» переходит из позиции с индексом 2 в позицию с индексом 1.

Какой бы способ вы ни выбрали, при удалении значения из `MutableList` размер списка уменьшается.

...и заменять их другими значениями

Если вы хотите обновить `MutableList` так, чтобы значение с конкретным индексом было заменено другим значением, это можно сделать при помощи функции **`set`**. Например, следующий код заменяет значение «Tea» с индексом 0 строкой «Coffee»:

```
if (mShopping.size > 0) {
    mShopping.set(0, "Coffee")
}
```



Функция set() присваивает элементу с заданным индексом ссылку на другой объект.

Можно изменять порядок и вносить массовые изменения...

MutableList также включает функции для изменения порядка хранения элементов. Например, можно отсортировать содержимое MutableList в естественном порядке при помощи функции **sort** или переставить их в обратном порядке функцией **reverse**:

```
mShopping.sort()
mShopping.reverse()
```

Этот фрагмент сортирует MutableList в обратном порядке.

Или сгенерировать случайную перестановку функцией **shuffle**:

```
mShopping.shuffle()
```

Также существуют полезные функции для внесения массовых изменений в MutableList. Например, функция **addAll** добавляет все элементы, хранящиеся в другой коллекции. Следующий код добавляет в mShopping значения «Cookies» и «Sugar»:

```
val toAdd = listOf("Cookies", "Sugar")
mShopping.addAll(toAdd)
```

Функция **removeAll** удаляет элементы, входящие в другую коллекцию:

```
val toRemove = listOf("Milk", "Sugar")
mShopping.removeAll(toRemove)
```

Функция **retainAll** оставляет все элементы, входящие в другую коллекцию, и удаляет все остальные:

```
val toRetain = listOf("Milk", "Sugar")
mShopping.retainAll(toRetain)
```

Также можно воспользоваться функцией **clear** для удаления всех элементов:

```
mShopping.clear()
```

Уничтожает содержимое mShopping, чтобы размер был равен 0.

...или копировать весь объект MutableList

Иногда требуется скопировать List или MutableList, чтобы сохранить «снимок» содержимого списка. Для этой цели используется функция **toList**. Например, следующий код копирует mShopping и присваивает копию новой переменной с именем shoppingSnapshot:

```
val shoppingCopy = mShopping.toList()
```

Функция toList возвращает List, а не MutableList, поэтому содержимое shoppingCopy обновить невозможно. Другие полезные функции, которые могут использоваться для копирования MutableList, — **sorted** (возвращает отсортированный список List), **reversed** (возвращает список List со значениями в обратном порядке) и **shuffled** (возвращает список List и генерирует случайную перестановку его значений).

MutableList также содержит функцию toMutableList(), которая возвращает копию в форме нового объекта MutableList.

Часто Задаваемые Вопросы

В: Что такое «пакет»?

О: Пакет представляет собой механизм группировки классов и функций. Пакеты полезны по двум причинам.

Во-первых, они используются для организации проектов или библиотек. Вы группируете их по пакетам в соответствии с функциональностью, а не сваливаете все классы в одну кучу.

Во-вторых, пакеты определяют видимость имен; это означает, что разные программисты могут создавать классы с одинаковыми именами — главное, чтобы эти классы размещались в разных пакетах.

Информацию о структурировании кода в пакетах см. в приложении III.

В: В Java мне приходится импортировать все пакеты, которые я хочу использовать, включая коллекции. А как это делается в Kotlin?

О: Kotlin автоматически импортирует множество пакетов из стандартной библиотеки Kotlin, включая пакет *kotlin.collections*. Тем не менее во многих ситуациях пакеты приходится импортировать вручную; за дополнительной информацией обращайтесь к приложению III.

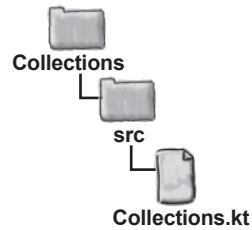
Создание проекта Collections

Итак, мы рассмотрели списки `List` и `MutableList`. Давайте создадим проект, в котором используются эти классы.

Создайте новый проект Kotlin для JVM с именем «Collections». Затем создайте новый файл Kotlin с именем *Collections.kt*: выделите папку *src*, щелкните на меню File и выберите команду New → Kotlin File/Class. Введите имя файла «Collections» и выберите вариант File в категории Kind.

Добавьте в *Collections.kt* следующий код:

```
fun main(args: Array<String>) {
    val mShoppingList = mutableListOf("Tea", "Eggs", "Milk")
    println("mShoppingList original: $mShoppingList")
    val extraShopping = listOf("Cookies", "Sugar", "Eggs")
    mShoppingList.addAll(extraShopping)
    println("mShoppingList items added: $mShoppingList")
    if (mShoppingList.contains("Tea")) {
        mShoppingList.set(mShoppingList.indexOf("Tea"), "Coffee")
    }
    mShoppingList.sort()
    println("mShoppingList sorted: $mShoppingList")
    mShoppingList.reverse()
    println("mShoppingList reversed: $mShoppingList")
}
```



Тест-драйв

При выполнении кода в окне вывода IDE отображается следующий текст:

```
mShoppingList original: [Tea, Eggs, Milk]
mShoppingList items added: [Tea, Eggs, Milk, Cookies, Sugar, Eggs]
mShoppingList sorted: [Coffee, Cookies, Eggs, Eggs, Milk, Sugar]
mShoppingList reversed: [Sugar, Milk, Eggs, Eggs, Cookies, Coffee]
```

При выводе List или MutableList элементы выводятся в порядке индексов в квадратных скобках.

А теперь проверьте свои силы в следующем упражнении.

Развлечения с магнитами

На холодильнике была выложена функция `main`, которая выводит результат, приведенный справа. К сожалению, магниты перепутались. Удастся ли вам восстановить функцию?

Функция должна выдавать этот результат:

```
[Zero, Two, Four, Six]
[Two, Four, Six, Eight]
[Two, Four, Six, Eight, Ten]
[Two, Four, Six, Eight, Ten]
```

Здесь размещается код.

```
a.add(2, "Four")
println(a)
println(a)
println(a)
}
a.add(0, "Zero")
a.add(1, "Two")
fun main(args: Array<String>) {
a.add(3, "Six")
println(a)
a.removeAt(0)
var a: MutableList<String> = mutableListOf()
if (a.indexOf("Four") != 4) a.add("Ten")
if (a.contains("Zero")) a.add("Eight")
if (a.contains("Zero")) a.add("Twelve")
```

Развлечения с магнитами. Решение

На холодильнике была выложена функция `main`, которая выводит результат, приведенный справа. К сожалению, магниты перепутались. Удастся ли вам восстановить функцию?

[Zero, Two, Four, Six]
[Two, Four, Six, Eight]
[Two, Four, Six, Eight, Ten]
[Two, Four, Six, Eight, Ten]

```
fun main(args: Array<String>) {
    var a: MutableList<String> = mutableListOf()
    a.add(0, "Zero")
    a.add(1, "Two")
    a.add(2, "Four")
    a.add(3, "Six")
    println(a)
    if (a.contains("Zero")) a.add("Eight")
    a.removeAt(0)
    println(a)
    if (a.indexOf("Four") != 4) a.add("Ten")
    println(a)
    if (a.contains("Zero")) a.add("Twelve")
    println(a)
}
```

List позволяет дублировать значения

Как вы уже знаете, списки `List` и `MutableList` обладают большей гибкостью, чем массивы. В отличие от массивов, вы можете явно указать, должна ли коллекция быть неизменяемой или вы собираетесь добавлять, удалять и обновлять ее элементы.

Однако в некоторых ситуациях список `List` (или `MutableList`) оставляет желать лучшего.

Представьте, что вы устраиваете обед с группой друзей, и чтобы забронировать столик, нужно знать, сколько людей собирается прийти. В такой ситуации можно воспользоваться списком `List`, но здесь возникает проблема: в **`List` могут присутствовать дубликаты.** Например, в созданном списке имена некоторых друзей могут встречаться дважды:

```
val friendList = listOf("Jim",
                        "Sue",
                        "Sue",
                        "Nick",
                        "Nick")
```

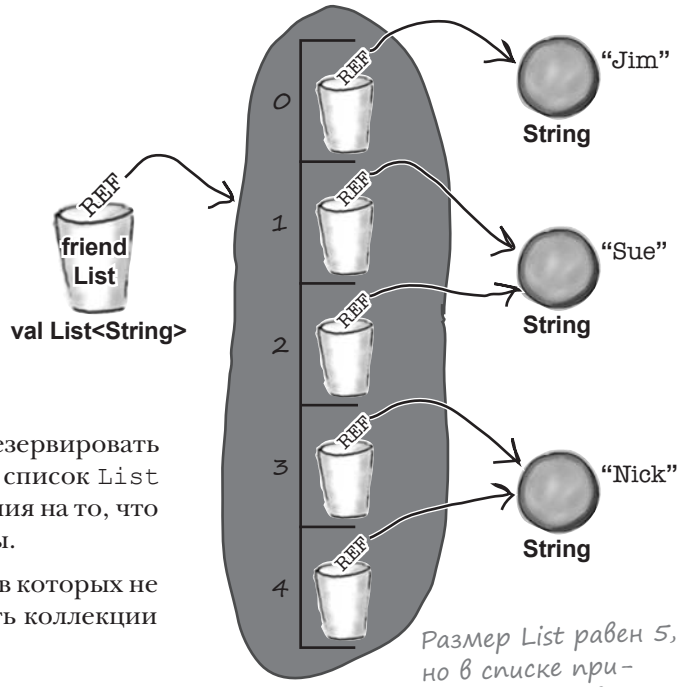
В списке трое друзей, но имена двоих повторяются..

Но если вы хотите знать, сколько *разных* друзей содержит список, вы не сможете воспользоваться таким кодом:

```
friendList.size
```

чтобы определить, на сколько людей следует зарезервировать столик. Свойство `size` показывает лишь то, что список `List` состоит из пяти элементов, и не обращает внимания на то, что в списке присутствуют повторяющиеся элементы.

В подобных ситуациях используются коллекции, в которых не может быть дубликатов. Какую же разновидность коллекции следует выбрать?



Размер List равен 5, но в списке присутствуют всего 3 разных значения.

МОЗГОВОЙ ШТУРМ

Ранее в этой главе упоминались разные типы коллекций, доступные в Kotlin. Как вы думаете, какие виды коллекций лучше всего подойдут в такой ситуации?

.....

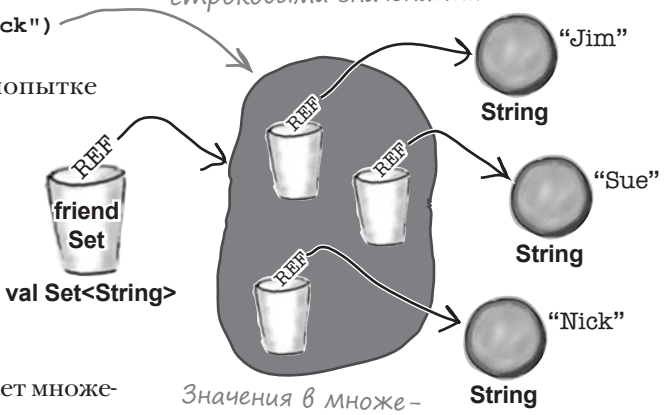
Как создать множество Set

Если вам нужна коллекция, которая не допускает дублирования, используйте **`Set`**: неупорядоченную коллекцию без повторяющихся значений.

Для создания объекта `Set` вызывается функция с именем **`setOf`**, которой передаются значения для инициализации элементов множества. Например, следующий фрагмент создает множество `Set`, инициализирует его тремя строками и присваивает новой переменной с именем `friendSet`:

```
val friendSet = setOf("Jim", "Sue", "Nick")
```

Создает Set с тремя строковыми значениями.



Значения в множестве Set не упорядочены, и дубликатов в них быть не может.

В `Set` не может быть дубликатов, поэтому при попытке создать множество следующей командой:

```
val friendSet = setOf("Jim",
                      "Sue",
                      "Sue",
                      "Nick",
                      "Nick")
```

`Set` игнорирует дубликаты «Sue» и «Nick». Код создает множество `Set`, которое содержит три разных строковых значения, как и в предыдущем случае.

Компилятор определяет тип элементов множества `Set` по значениям, передаваемым при создании. Например, следующий код инициализирует `Set` строковыми значениями, так что компилятор создает множество `Set` с типом `Set<String>`.

Как использовать значения Set

Значения `Set` не упорядочены, поэтому в отличие от `List`, у них нет функции `get` для получения значения с заданным индексом. Впрочем, функция `contains` позволяет проверить, содержит ли множество `Set` конкретное значение:

```
val isFredGoing = friendSet.contains("Fred")
```

Возвращает true, если friendSet содержит значение «Fred», и false в противном случае.

Перебор элементов множества `Set` в цикле выполняется так:

```
for (item in friendSet) println(item)
```

Множество `Set` неизменяемо — в него нельзя добавлять новые или удалять существующие значения. Для выполнения таких операций используется класс `MutableSet`. Но прежде чем мы покажем, как создавать и использовать такой класс, следует ответить на один важный вопрос: **как `Set` определяет, является ли значение дубликатом?**

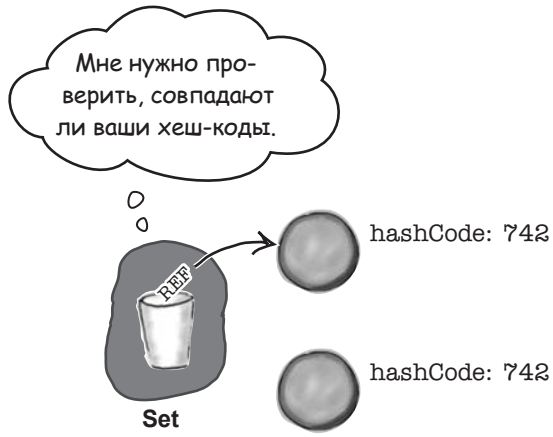
В отличие от List, множество Set не упорядочено и не может содержать повторяющихся значений.

Как Set проверяет наличие дубликатов

Чтобы ответить на этот вопрос, воспроизведем последовательность действий, которые выполняет `Set` для решения вопроса о том, является ли значение дубликатом.

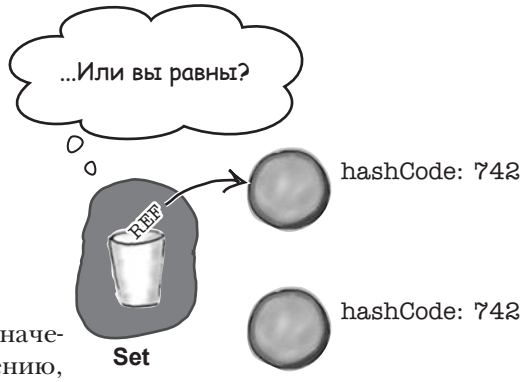
1 **Set получает хеш-код объекта и сравнивает его с хеш-кодами объектов, уже находящихся в множестве Set.**
`Set` использует хеш-коды для сохранения элементов способом, заметно ускоряющим обращение к ним. Хеш-код — это своего рода этикетка на «корзине», в которой хранятся элементы, так что все объекты с хеш-кодом 742 (например) хранятся в корзине с меткой 742.

Если совпадающих хеш-кодов не находится, `Set` считает, что это не дубликат, и добавляет новое значение. Но если совпадающие хеш-коды будут обнаружены, класс `Set` должен выполнить дополнительные проверки — происходит переход к шагу 2.



2 **Set использует оператор === для сравнения нового значения с любыми содержащимися в нем объектами с тем же хеш-кодом.**
Как было показано в главе 7, оператор === используется для проверки того, указывают ли две ссылки на один объект. Таким образом, если оператор === возвращает `true` для любого объекта с тем же хеш-кодом, `Set` знает, что новое значение является дубликатом, и отклоняет его. Но если оператор === возвращает `false`, то `Set` переходит к шагу 3.

3 **Set использует оператор == для сравнения нового значения со всеми объектами, содержащимися в коллекции, с совпадающими хеш-кодами.**
Оператор == вызывает функцию `equals` значения. Если функция возвращает `true`, то `Set` рассматривает новое значение как дубликат и отвергает его. Если же оператор == возвращает `false`, то `Set` считает, что новое значение не является дубликатом, и добавляет его.



Итак, есть две ситуации, в которых `Set` рассматривает новое значение как дубликат: если это *тот же* объект или оно *равно* значению, уже содержащемуся в коллекции. Рассмотрим происходящее более подробно.

Хеш-коды и равенство

Как вы узнали в главе 7, оператор === проверяет, указывают ли две ссылки на один объект, а оператор == проверяет, указывают ли ссылки на объекты, которые должны считаться равными. Однако класс `Set` *использует эти операторы только после того, как установит, что два объекта имеют одинаковые значения хеш-кодов*. Это означает, что для правильной работы `Set` **равные объекты должны иметь одинаковые хеш-коды**.

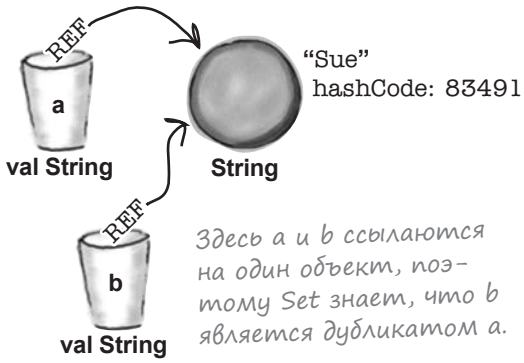
Посмотрим, как это относится к операторам === и ==.

Проверка оператором ===

Если у вас имеются две ссылки, указывающие на один объект, вы получите одинаковые результаты при вызове функции `hashCode` для каждой ссылки. Если функция `hashCode` не переопределена, то в поведении по умолчанию (унаследованном от суперкласса `Any`) каждый класс получит уникальный хеш-код.

При выполнении следующего кода `Set` замечает, что `a` и `b` имеют одинаковые хеш-коды и ссылаются на один объект, поэтому в `Set` добавляется только одно значение:

```
val a = "Sue"
val b = a
val set = setOf(a, b)
```



Здесь a и b ссылаются на один объект, поэтому Set знает, что b является дубликатом a.

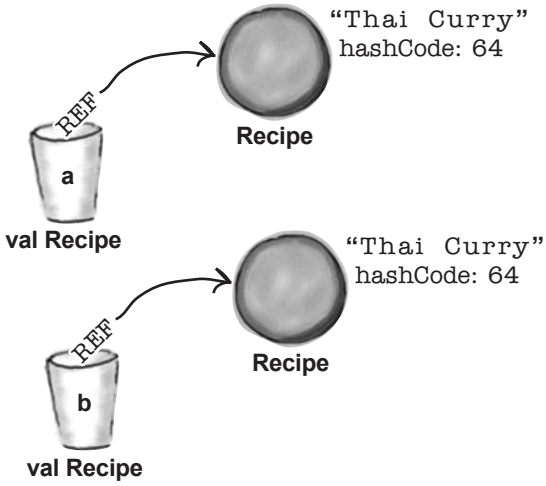
Проверка оператором ==

Если вы хотите, чтобы класс `Set` рассматривал два разных объекта `Recipe` как равные (или эквивалентные), есть два варианта: сделать `Recipe` классом данных или переопределить функции `hashCode` и `equals`, унаследованные от `Any`. Преобразование `Recipe` в класс данных — самый простой вариант, так как в этом случае обе функции переопределяются автоматически.

Как упоминалось выше, в поведении по умолчанию (из `Any`) каждому объекту предоставляется уникальное значение хеш-кода. Таким образом, вы *должны* переопределить `hashCode`, чтобы быть уверенными в том, что два эквивалентных объекта возвращают один хеш-код. Но вы также должны переопределить `equals`, чтобы оператор == возвращал `true` при сравнении объектов с совпадающими значениями свойств.

В следующем примере в `Set` будет добавлено одно значение, если `Recipe` является классом данных:

```
val a = Recipe("Thai Curry")
val b = Recipe("Thai Curry")
val set = setOf(a, b)
```



Здесь a и b указывают на разные объекты. Set считает b дубликатом только в том случае, если a и b имеют одинаковые хеш-коды и a == b. В частности, это условие будет выполняться в том случае, если Recipe является классом данных.

Правила переопределения hashCode и equals

Если вы решите вручную переопределить функции `hashCode` и `equals` в своем классе (вместо того, чтобы использовать класс данных), вам придется соблюдать ряд правил. Если эти правила будут нарушены, в мире Kotlin произойдет катастрофа — множества `Set` будут работать некорректно. Непременно соблюдайте эти правила!

Перечислим эти правила:

- Если два объекта равны, они должны иметь одинаковые хеш-коды.
- Если два объекта равны, вызов `equals` для любого из объектов должен возвращать `true`. Другими словами, если `(a.equals(b))`, то `(b.equals(a))`.
- Если два объекта имеют одинаковые хеш-коды, это не значит, что они равны. Но если они равны, то они должны иметь одинаковые хеш-коды.
- При переопределении `equals` вы должны переопределить `hashCode`.
- По умолчанию функция `hashCode` генерирует уникальное целое число для каждого объекта. Таким образом, если вы не переопределите `hashCode` в классе, не являющемся классом данных, два объекта этого типа ни при каких условиях не будут считаться равными.
- По умолчанию функция `equals` должна выполнять сравнение `===` , то есть проверять, что две ссылки относятся к одному объекту. Следовательно, если вы не переопределяете `equals` в классе, не являющемся классом данных, два объекта не будут считаться равными, потому что по ссылкам на два разных объекта всегда будут содержаться разные наборы битов.

a.equals(b) также должно означать, что a.hashCode() == b.hashCode()

Но a.hashCode() == b.hashCode() не означает, что a.equals(b)

Часто Задаваемые Вопросы

В: Как хеш-коды могут быть равными, если объекты не равны?

О: Как уже говорилось, `Set` использует хеш-коды для хранения элементов способом, ускоряющим обращение к элементам. Если вам потребуется найти объект в множестве `Set`, было бы неэффективно начинать поиск с начала и проверять каждый элемент, пока не будет найдено совпадение. Вместо этого хеш-код используется в качестве этикетки для «корзины», в которой был сохранен элемент. Если вы говорите «Я хочу найти объект в Set, который выглядит так...», класс `Set` получает значение хеш-кода по предоставленному вами объекту, а затем переходит прямо к корзине с этим хеш-кодом.

Это не все, но этого более чем достаточно, чтобы эффективно пользоваться `Set` и понимать, что при этом происходит.

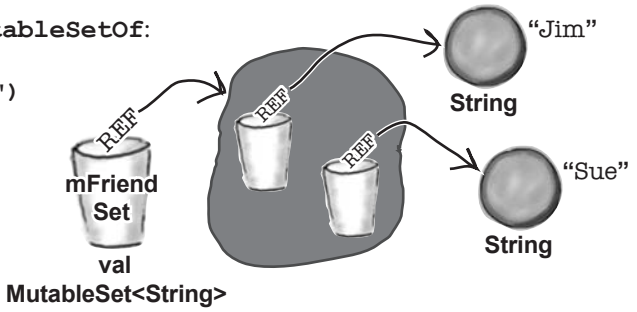
Суть в том, что хеш-коды могут быть одинаковыми, и это еще не гарантирует равенства объектов, потому что алгоритм хеширования, используемый в функции `hashCode`, может возвращать одинаковое значение для разных объектов. Да, это означает, что несколько объектов могут попасть в одну корзину в `Set` (потому что каждая корзина представляет отдельное значение хеш-кода), но в этом нет ничего ужасного. Это может означать, что `Set` работает чуть менее эффективно или что коллекция заполнена аномально большим количеством элементов, но если `Set` обнаружит более одного объекта в гнезде с одним хеш-кодом, `Set` просто использует операторы `===` и `==` для поиска идеального совпадения. Иначе говоря, значения хеш-кодов иногда используются для сужения поиска, но для нахождения точного совпадения `Set` придется перебрать все объекты в этой корзине (со всеми объектами, имеющими одинаковые хеш-коды) и проверить, присутствует ли совпадающий объект в этой корзине.

Как использовать MutableSet

Теперь, когда вы знаете о `Set`, перейдем к **`MutableSet`**. `MutableSet` является подклассом `Set`, но содержит дополнительные функции для добавления и удаления значений.

Объект `MutableSet` создается вызовом функции **`mutableSetOf`**:

```
val mFriendSet = mutableSetOf("Jim", "Sue")
```



Если передать функции mutableSetOf() строковые значения, компилятор определяет, что вам нужен объект типа MutableSet<String> (MutableSet для хранения String).

Здесь `MutableSet` инициализируется двумя строками, поэтому компилятор заключает, что вам нужен объект `MutableSet` с типом `MutableSet<String>`.

Новые значения добавляются в `MutableSet` функцией **`add`**. Например, следующий код добавляет значение "Nick" в `mFriendSet`:

```
mFriendSet.add("Nick")
```

Функция `add` проверяет, встречается ли переданный объект в `MutableSet`. Если дубликат будет найден, возвращается `false`. Но если значение не является дубликатом, оно добавляется в `MutableSet` (с увеличением размера на 1), а функция возвращает `true` — признак успешного выполнения операции.

Для удаления значений из `MutableSet` используется функция `remove`. Например, следующий код удаляет строку «Nick» из `mFriendSet`:

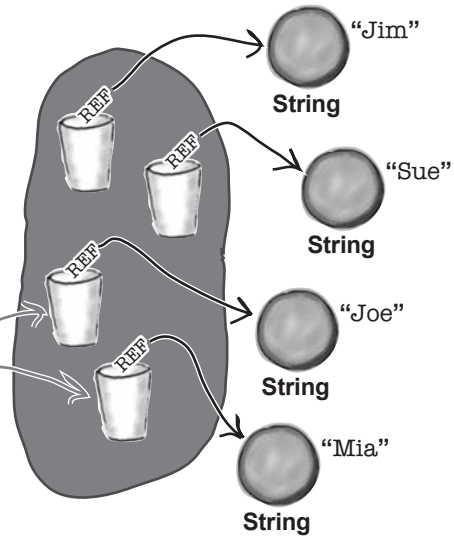
```
mFriendSet.remove("Nick")
```

Если строка «Nick» существует в `MutableSet`, функция удаляет ее и возвращает `true`. Но если подходящий объект найти не удается, функция просто возвращает `false`.

Функции **`addAll`**, **`removeAll`** и **`retainAll`** также могут использоваться для внесения массовых изменений в `MutableSet` (по аналогии с `MutableList`). Например, функция `addAll` добавляет в `MutableSet` все элементы, присутствующие в другой коллекции, так что для добавления «Joe» и «Mia» в `mFriendSet` можно использовать следующий код:

```
val toAdd = setOf("Joe", "Mia")
mFriendSet.addAll(toAdd)
```

addAll() добавляет значения, содержащиеся в другом объекте Set.



Как и в случае с `MutableList`, также можно воспользоваться функцией **`clear`** для удаления всех элементов из `MutableSet`:

```
mFriendSet.clear()
```

Копирование MutableSet

Если вы хотите сделать снимок содержимого `MutableSet`, вы можете сделать это по аналогии с `MutableList`. Например, при помощи функции **`toSet`** можно создать неизменяемую копию `mFriendSet` и присвоить копию новой переменной `friendSetCopy`:

```
val friendSetCopy = mFriendSet.toSet()
```

Также можно скопировать `Set` или `MutableSet` в новый объект `List` функцией **`toList`**:

```
val friendList = mFriendSet.toList()
```

А если у вас имеется объект `MutableList` или `List`, его можно скопировать в `Set` функцией **`toSet`**:

Объект MutableSet также содержит функции toMutableSet() (для копирования в новый объект MutableSet) и toMutableList() (для копирования в новый объект MutableList).

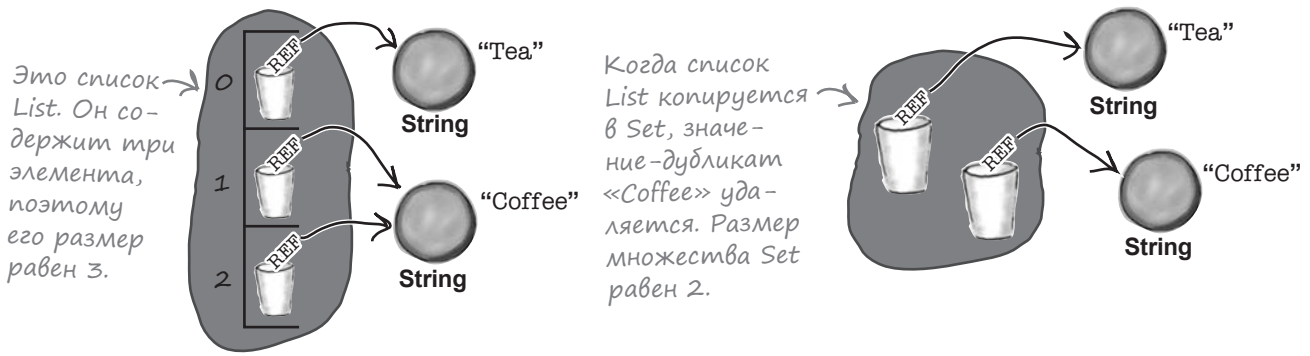
```
val shoppingSet = mShopping.toSet()
```

Копирование коллекции в другой тип может быть особенно полезно, если вы хотите выполнить другую операцию, которая без этого была бы неэффективной. Например, чтобы проверить, содержит ли список дубликаты, можно скопировать `List` в `Set` и проверить размер каждой коллекции. В следующем коде этим способом мы проверяем, содержит ли дубликаты список `mShopping` (`MutableList`):

Создает версию mShopping в формате Set и возвращает ее размер.

```
if (mShopping.size > mShopping.toSet().size) {
    //mShopping содержит дубликаты
}
```

Если список `mShopping` содержит дубликаты, его размер будет больше, чем после копирования в `Set`, потому что при преобразовании `MutableList` в `Set` дубликаты будут удалены.



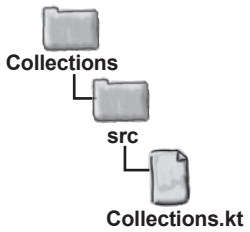
Обновление проекта Collections

После знакомства с `Set` и `MutableSet` обновим проект Collections, чтобы в них использовались новые коллекции.

Обновите свою версию *Collections.kt*, чтобы она соответствовала нашей (изменения выделены жирным шрифтом):

Переменная mShoppingList должна объявляться с ключевым словом var, чтобы позднее ее можно было обновить другим значением MutableList<String>.

```
fun main(args: Array<String>) {
    ~~val~~ var mShoppingList = mutableListOf("Tea", "Eggs", "Milk")
    println("mShoppingList original: $mShoppingList")
    val extraShopping = listOf("Cookies", "Sugar", "Eggs")
    mShoppingList.addAll(extraShopping)
    println("mShoppingList items added: $mShoppingList")
    if (mShoppingList.contains("Tea")) {
        mShoppingList.set(mShoppingList.indexOf("Tea"), "Coffee")
    }
    mShoppingList.sort()
    println("mShoppingList sorted: $mShoppingList")
    mShoppingList.reverse()
    println("mShoppingList reversed: $mShoppingList")

    val mShoppingSet = mShoppingList.toMutableSet()
    println("mShoppingSet: $mShoppingSet")
    val moreShopping = setOf("Chives", "Spinach", "Milk")
    mShoppingSet.addAll(moreShopping)
    println("mShoppingSet items added: $mShoppingSet")
    mShoppingList = mShoppingSet.toMutableList()
    println("mShoppingList new version: $mShoppingList")
}
```



Добавьте этот код.

А теперь проведем тест-драйв.

Тест-драйв

При выполнении кода в окне вывода IDE отображается следующий текст:

```
mShoppingList original: [Tea, Eggs, Milk]
mShoppingList items added: [Tea, Eggs, Milk, Cookies, Sugar, Eggs]
mShoppingList sorted: [Coffee, Cookies, Eggs, Eggs, Milk, Sugar]
mShoppingList reversed: [Sugar, Milk, Eggs, Eggs, Cookies, Coffee]
mShoppingSet: [Sugar, Milk, Eggs, Cookies, Coffee]
mShoppingSet items added: [Sugar, Milk, Eggs, Cookies, Coffee, Chives, Spinach]
mShoppingList new version: [Sugar, Milk, Eggs, Cookies, Coffee, Chives, Spinach]
```

При выводе содержимого Set или MutableSet каждый элемент заключается в квадратные скобки.

Часто Задаваемые Вопросы

В: Вы сказали, что я могу создать копию `List` в форме `Set` и создать копию `Set` в форме `List`. А с массивами это возможно?

О: Да, возможно. Массивы содержат набор функций для копирования массива в новую коллекцию: `toList()`, `toMutableList()`, `toSet()` и `toMutableSet()`. Таким образом, следующий код создает массив с элементами `Int`, после чего копирует его содержимое в `Set<Int>`:

```
val a = arrayOf(1, 2, 3)
val s = a.toSet()
```

Аналогичным образом в List и Set (а следовательно, в MutableList и MutableSet) имеется функция toTypedArray(), которая копирует коллекцию в новый массив подходящего размера. Код

```
val s = setOf(1, 2, 3)
val a = s.toTypedArray()
```

создает массив типа `Array<Int>`.

В: Можно ли отсортировать множество `Set`?

О: Нет, `Set` — неупорядоченная коллекция, которую невозможно отсортировать напрямую. Впрочем, вы можете воспользоваться ее функцией `toList()` для копирования `Set` в `List`, а затем отсортировать `List`.

В: Можно ли использовать оператор == для сравнения содержимого двух множеств `Set`?

О: Да, можно. Предположим, имеются два множества, `a` и `b`. Если `a` и `b` содержат одинаковые значения, `a == b` вернет `true`, как в следующем примере:

```
val a = setOf(1, 2, 3)
val b = setOf(3, 2, 1)
//a == b равно true
```

Но если два множества содержат разные значения, то результат будет равен `false`.

В: А если одним из множеств является `MutableSet`? Нужно ли сначала скопировать его в `Set`?

О: Оператор `==` можно использовать и без копирования `MutableSet` в `Set`. В следующем примере `a == b` возвращает `true`:

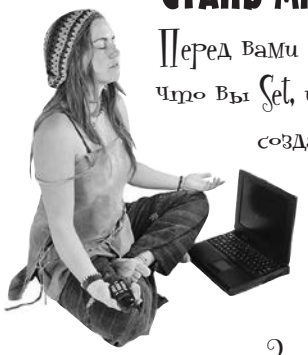
```
val a = setOf(1, 2, 3)
val b = mutableSetOf(3, 2, 1)
```

В: Понятно. Значит, оператор == работает и с `List`?

О: Да, вы можете использовать `==` для сравнения содержимого двух списков `List`. Оператор вернет `true`, если списки `List` содержат одинаковые значения в позициях с одинаковыми индексами, или `false`, если списки `List` содержат разные значения (или те же значения, но в другом порядке). Таким образом, в следующем примере `a == b` вернет `true`:

```
val a = listOf(1, 2, 3)
val b = listOf(1, 2, 3
```

СТАНЬ множеством Set



Перед вами четыре класса Duck. Представьте, что вы Set, и скажите, какие классы будут создавать множество Set, содержащее ровно один элемент, с приведенной справа функцией main. Нарушают ли какие-либо классы Duck правила hashCode() и equals()? Если да, то как?

Это функция main.

```
fun main(args: Array<String>) {
    val set = setOf(Duck(), Duck(17))
    println(set)
}
```

A

```
class Duck(val size: Int = 17) {
    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other is Duck && size == other.size) return true
        return false
    }

    override fun hashCode(): Int {
        return size
    }
}
```

B

```
class Duck(val size: Int = 17) {
    override fun equals(other: Any?): Boolean {
        return false
    }

    override fun hashCode(): Int {
        return 7
    }
}
```

C

```
data class Duck(val size: Int = 18)
```

D

```
class Duck(val size: Int = 17) {
    override fun equals(other: Any?): Boolean {
        return true
    }

    override fun hashCode(): Int {
        return (Math.random() * 100).toInt()
    }
}
```

Ответы на с. 304.

Возьми в руку карандаш

Четверо друзей составили списки `List` своих домашних питомцев. Один элемент представляет одно животное. Списки выглядят так:

```
val petsLiam = listOf("Cat", "Dog", "Fish", "Fish")
val petsSophia = listOf("Cat", "Owl")
val petsNoah = listOf("Dog", "Dove", "Dog", "Dove")
val petsEmily = listOf("Hedgehog")
```

Напишите код для создания новой коллекции `pets`, в которой присутствуют все животные из всех списков.

.....

.....

.....

.....

.....

Как бы вы использовали коллекцию `pets` для получения общего количества животных?

.....

Напишите код для вывода количества разных видов животных.

.....

.....

Как бы вы перечислили разные виды животных в алфавитном порядке?

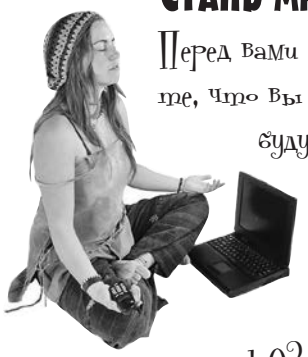
.....

.....

.....

Ответы на с. 305.

СТАНЬ множеством Set. Решение



Перед вами четыре класса Duck. Представьте, что вы Set, и скажите, какие классы будут создавать множество Set, содержащее ровно один элемент, с приведенной справа функцией main. Нарушают ли какие-либо классы Duck правила hashCode() и equals()? Если да, то как?

Это функция main.

```
fun main(args: Array<String>) {
    val set = setOf(Duck(), Duck(17))
    println(set)
}
```

A

```
class Duck(val size: Int = 17) {
    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other is Duck && size == other.size) return true
        return false
    }

    override fun hashCode(): Int {
        return size
    }
}
```

Соответствует правилам hashCode() и equals(). Set понимает, что второй объект Duck является дубликатом, поэтому функция main создает Set с одним элементом.

B

```
class Duck(val size: Int = 17) {
    override fun equals(other: Any?): Boolean {
        return false
    }

    override fun hashCode(): Int {
        return 7
    }
}
```

Создает множество Set с двумя элементами. Класс нарушает правила hashCode() и equals(), так как equals() всегда возвращает false, даже при сравнении объекта с самим собой.

C

```
data class Duck(val size: Int = 18)
```

Соответствует правилам, но создает множество Set с двумя элементами.

D

```
class Duck(val size: Int = 17) {
    override fun equals(other: Any?): Boolean {
        return true
    }

    override fun hashCode(): Int {
        return (Math.random() * 100).toInt()
    }
}
```

Создает множество Set с двумя элементами. Класс нарушает правила, так как hashCode() возвращает случайное число. Правила требуют, чтобы для равных объектов возвращались одинаковые хеш-коды.

Возьми в руку карандаш
Решение

Четверо друзей составили списки `List` своих домашних питомцев. Один элемент представляет одно животное. Списки выглядят так:

```
val petsLiam = listOf("Cat", "Dog", "Fish", "Fish")
val petsSophia = listOf("Cat", "Owl")
val petsNoah = listOf("Dog", "Dove", "Dog", "Dove")
val petsEmily = listOf("Hedgehog")
```

Напишите код для создания новой коллекции `pets`, в которой присутствуют все животные из всех списков.

Не беспокойтесь, если ваши ответы отличаются от наших. К одному результату можно прийти разными способами.

```
var pets: MutableList<String> = mutableListOf()
pets.addAll(petsLiam)
pets.addAll(petsSophia)
pets.addAll(petsNoah)
pets.addAll(petsEmily)
```

Как бы вы использовали коллекцию `pets` для получения общего количества животных?

```
pets.size
```

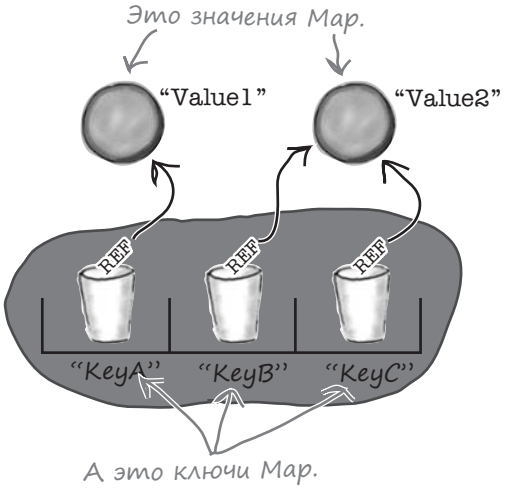
Напишите код для вывода количества разных видов животных.

```
val petSet = pets.toMutableSet()
println(petSet.size)
```

Как бы вы перечислили разные виды животных в алфавитном порядке?

```
val petList = petSet.toMutableList()
petList.sort()
println(petList)
```

Ассоциативные массивы Map



`List` и `Set` прекрасно работают, но есть еще один тип коллекций, с которым мы хотим вас познакомить: **`Map`**. Коллекция `Map` работает как список свойств. Вы передаете ей ключ, а `Map` возвращает значение, связанное с этим ключом. Хотя ключи обычно имеют тип `String`, они могут быть объектами любого типа.

Каждый элемент `Map` состоит из двух объектов — ***ключа*** и ***значения***. С каждым ключом связывается одно значение. В коллекции могут присутствовать повторяющиеся *значения*, но не повторяющиеся *ключи*.

Как создать Map

Чтобы создать `Map`, вызовите функцию с именем **`mapOf`** и передайте ей пары «ключ-значение» для инициализации `Map`. Например, следующий код создает `Map` с тремя элементами. Ключами являются строки («Recipe1», «Recipe2» и«Recipe3»), а значениями — объекты `Recipe`:

```
val r1 = Recipe("Chicken Soup")
val r2 = Recipe("Quinoa Salad")
val r3 = Recipe("Thai Curry")

val recipeMap = mapOf("Recipe1" to r1, "Recipe2" to r2, "Recipe3" to r3)
```

Каждый элемент определяется в форме «ключ-значение». Ключи обычно являются строками, как в данном примере.

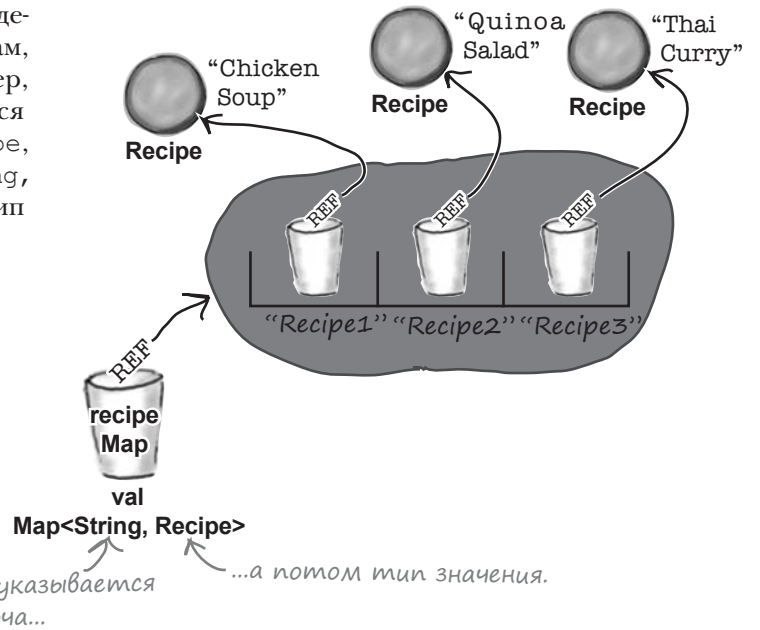
Как нетрудно догадаться, компилятор определяет тип пар «ключ-значение» по элементам, которыми он инициализируется. Например, приведенный выше `Map` инициализируется строковыми ключами и значениями `Recipe`, поэтому будет создан `Map` с типом `Map<String, Recipe>`. Также можно явно определить тип `Map` кодом следующего вида:

```
val recipeMap: Map<String, Recipe>
```

Обычно тип `Map` задается в форме:

```
Map<тип_ключа, тип_значения>
```

Итак, вы научились создавать ассоциативные массивы `Map`. Теперь посмотрим, как их использовать.



Как использовать Map

С Map чаще всего выполняются три операции: проверка наличия конкретного ключа или значения, выборка значения для заданного ключа и перебор всех элементов Map.

Для проверки наличия конкретного ключа или значения в Map используются его функции **containsKey** и **containsValue**. Например, следующий код проверяет, содержит ли Map с именем recipeMap ключ «Recipe1»:

```
recipeMap.containsKey("Recipe1")
```

Вы можете проверить, содержит ли recipeMap объект Recipe для «Chicken Soup» при помощи функции containsValue:

```
val recipeToCheck = Recipe("Chicken Soup")
if (recipeMap.containsValue(recipeToCheck)) {
    //Код, выполняемый при наличии значения в Map
}
```

Предполагается, что Recipe является классом данных, так что Map может определить, когда два объекта Recipe равны.

Для получения значения, связанного с конкретным ключом, используются функции **get** и **getValue**. Если заданный ключ не существует, get возвращает null, а getValue вызывает исключение. В следующем примере функция getValue получает объект Recipe, связанный с ключом «Recipe1»:

```
if (recipeMap.containsKey("Recipe1")) {
    val recipe = recipeMap.getValue("Recipe1")
    //Код использования объекта Recipe
}
```

Если в recipeMap нет ключа «Recipe1», эта строка вызовет исключение.

Также вы можете перебрать все элементы Map. Например, вот как цикл for используется для вывода всех пар «ключ-значение» в recipeMap:

```
for ((key, value) in recipeMap) {
    println("Key is $key, value is $value")
}
```

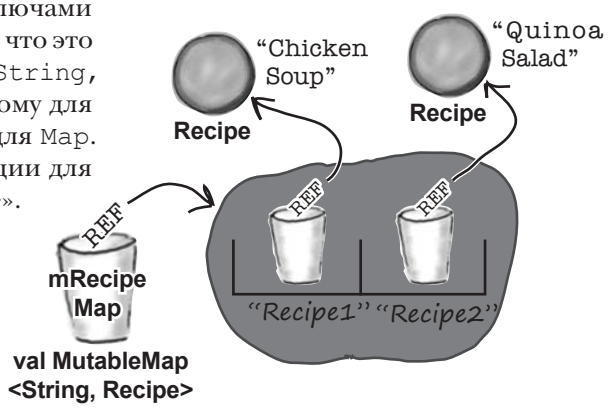
Объект Map неизменяем, поэтому вы не сможете добавлять или удалять пары «ключ-значение» или обновлять значение, хранящееся для заданного ключа. Для выполнения такой операции следует использовать класс MutableMap. Посмотрим, как он работает.

Создание MutableMap

Объекты **MutableMap** определяются практически так же, как Map, если не считать того, что вместо функции mapOf используется функция **mutableMapOf**. Например, следующий код создает массив MutableMap с тремя элементами, как и в предыдущем примере:

```
val r1 = Recipe("Chicken Soup")
val r2 = Recipe("Quinoa Salad")

val mRecipeMap = mutableMapOf("Recipe1" to r1, "Recipe2" to r2)
```

Объект MutableMap инициализируется строковыми ключами и значениями Recipe, поэтому компилятор делает вывод, что это должна быть коллекция MutableMap типа MutableMap<String, Recipe>. MutableMap является подклассом Map, поэтому для MutableMap могут вызываться те же функции, что и для Map. Однако MutableMap содержит дополнительные функции для добавления, удаления и обновления пар «ключ-значение».



Если функции mutableMapOf() передаются строковые ключи и значения Recipe, компилятор определяет, что нужно создать объект типа MutableMap<String, Recipe>.

Включение элементов в MutableMap

Для включения элементов в MutableMap используется функция **put**. Например, следующий код добавляет ключ «Recipe3» в mRecipeMap и связывает его с объектом Recipe для «Thai Curry»:

```
val r3 = Recipe("Thai Curry")
mRecipeMap.put("Recipe3", r3)
```

Сначала задается ключ, а потом значение.

Если MutableMap уже содержит заданный ключ, функция put заменяет значение для этого ключа и возвращает исходное значение.

В MutableMap можно добавить сразу несколько пар «ключ-значение» при помощи функции **putAll**. Функция получает один аргумент — Map с добавляемыми элементами. Например, следующий код добавляет объекты «Jambalaya» и «Sausage Rolls» в Map с именем recipesToAdd, после чего добавляет эти элементы в mRecipeMap:

```
val r4 = Recipe("Jambalaya")
val r5 = Recipe("Sausage Rolls")
val recipesToAdd = mapOf("Recipe4" to r4, "Recipe5" to r5)
mRecipeMap.putAll(recipesToAdd)
```

Теперь посмотрим, как происходит удаление значений.

Удаление элементов из MutableMap

Для удаления элементов из `MutableMap` используется функция **`remove`**. Функция перегружена, чтобы ее можно было вызывать двумя разными способами.

В первом способе функции `remove` передается ключ удаляемого элемента. Например, следующий код удаляет из `mRecipeMap` элемент с ключом «Recipe2»:

```
mRecipeMap.remove("Recipe2")
```

Удаление элемента с ключом «Recipe2».

Во втором варианте функции `remove` передается ключ и значение. Функция удаляет запись только в том случае, если будет найдено совпадение для ключа и для значения. Таким образом, следующий код удаляет элемент для «Recipe2» только тогда, когда он связан с объектом `Recipe` «Quinoa Salad»:

```
val recipeToRemove = Recipe("Quinoa Salad")
mRecipeMap.remove("Recipe2", recipeToRemove)
```

Удаление элемента с ключом «Recipe2», но только в том случае, если его значением является объект Recipe «Quinoa Salad».

Какой бы способ вы ни выбрали, при удалении элемента из коллекции `MutableMap` размер последней уменьшается.

Наконец, вы можете воспользоваться функцией **`clear`** для удаления всех элементов из `MutableMap` по аналогии с тем, как это делается с `MutableList` и `MutableSet`:

```
mRecipeMap.clear()
```

Функция clear() удаляет каждый элемент, но сам объект MutableMap все равно продолжает существовать.

Итак, вы научились обновлять `MutableMap`. Теперь давайте посмотрим, как их копировать.

Копирование Map и MutableMap

Как и в случае с другими типами коллекций, вы можете создать снимок MutableMap. Например, при помощи функции **toMap** можно создать копию mRecipeMap, доступную только для чтения, и присвоить ее новой переменной:

```
val recipeMapCopy = mRecipeMap.toMap()
```

Map или MutableMap можно скопировать в новый объект List, содержащий все пары «ключ-значение», при помощи функции **toList**:

MutableMap также содержит функции toMutableMap() и toMutableList().

```
val RecipeList = mRecipeMap.toList()
```

И вы также можете получить прямой доступ к парам «ключ-значение», обратившись к свойству **entries** объекта Map. Свойство entries возвращает Set при использовании с Map или MutableSet при использовании с MutableMap. Например, следующий код возвращает объект MutableSet с парами «ключ-значение» из mRecipeMap:

```
val recipeEntries = mRecipeMap.entries
```

Другие полезные свойства — **keys** (возвращает множество Set или MutableSet с ключами Map) и **values** (возвращает обобщенную коллекцию значений Map). Например, при помощи этих свойств можно проверить, встречаются ли в Map повторяющиеся значения:

Обратите внимание: свойства entries, keys и values фактически хранятся в Map (или MutableMap), это не копии. А если вы работаете с MutableMap, эти свойства могут обновляться.

```
if (mRecipeMap.size > mRecipeMap.values.toSet().size) {
    println("mRecipeMap contains duplicates values")
}
```

Это объясняется тем, что вызов

```
mRecipeMap.values.toSet()
```

копирует значения Map в Set с удалением всех дубликатов.

Теперь, когда узнали, как работать с массивами Map и MutableMap, немного доработаем наш проект Collections.

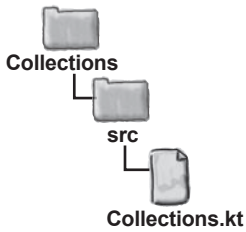
Полный код проекта Collections

Обновите свою версию *Collections.kt* и приведите ее в соответствие с нашей (изменения выделены жирным шрифтом):

```
data class Recipe(var name: String)
```

Добавьте класс данных Recipe.

```
fun main(args: Array<String>) {
    var mShoppingList = mutableListOf("Tea", "Eggs", "Milk")
    println("mShoppingList original: $mShoppingList")
    val extraShopping = listOf("Cookies", "Sugar", "Eggs")
    mShoppingList.addAll(extraShopping)
    println("mShoppingList items added: $mShoppingList")
    if (mShoppingList.contains("Tea")) {
        mShoppingList.set(mShoppingList.indexOf("Tea"), "Coffee")
    }
    mShoppingList.sort()
    println("mShoppingList sorted: $mShoppingList")
    mShoppingList.reverse()
    println("mShoppingList reversed: $mShoppingList")

    val mShoppingSet = mShoppingList.toMutableSet()
    println("mShoppingSet: $mShoppingSet")
    val moreShopping = setOf("Chives", "Spinach", "Milk")
    mShoppingSet.addAll(moreShopping)
    println("mShoppingSet items added: $mShoppingSet")
    mShoppingList = mShoppingSet.toMutableList()
    println("mShoppingList new version: $mShoppingList")
```



Добавьте этот фрагмент.

```
    val r1 = Recipe("Chicken Soup")
    val r2 = Recipe("Quinoa Salad")
    val r3 = Recipe("Thai Curry")
    val r4 = Recipe("Jambalaya")
    val r5 = Recipe("Sausage Rolls")
    val mRecipeMap = mutableMapOf("Recipe1" to r1, "Recipe2" to r2, "Recipe3" to r3)
    println("mRecipeMap original: $mRecipeMap")
    val recipesToAdd = mapOf("Recipe4" to r4, "Recipe5" to r5)
    mRecipeMap.putAll(recipesToAdd)
    println("mRecipeMap updated: $mRecipeMap")
    if (mRecipeMap.containsKey("Recipe1")) {
        println("Recipe1 is: ${mRecipeMap.getValue("Recipe1")}")
    }
}
```

А теперь посмотрим, как работает этот код.

Тест-драйв

При выполнении этого кода в окне вывода IDE отображается следующий текст:

```
mShoppingList original: [Tea, Eggs, Milk]
mShoppingList items added: [Tea, Eggs, Milk, Cookies, Sugar, Eggs]
mShoppingList sorted: [Coffee, Cookies, Eggs, Eggs, Milk, Sugar]
mShoppingList reversed: [Sugar, Milk, Eggs, Eggs, Cookies, Coffee]
mShoppingSet: [Sugar, Milk, Eggs, Cookies, Coffee]
mShoppingSet items added: [Sugar, Milk, Eggs, Cookies, Coffee, Chives, Spinach]
mShoppingList new version: [Sugar, Milk, Eggs, Cookies, Coffee, Chives, Spinach]
mRecipeMap original: {Recipe1=Recipe(name=Chicken Soup), Recipe2=Recipe(name=Quinoa Salad),
    Recipe3=Recipe(name=Thai Curry)}
mRecipeMap updated: {Recipe1=Recipe(name=Chicken Soup), Recipe2=Recipe(name=Quinoa Salad),
    Recipe3=Recipe(name=Thai Curry), Recipe4=Recipe(name=Jambalaya),
    Recipe5=Recipe(name=Sausage Rolls)}
Recipe1 is: Recipe(name=Chicken Soup)
```

При выводе Map или MutableMap каждая пара «ключ-значение» выводится в фигурных скобках.

Часто Задаваемые Вопросы

В: Почему в Kotlin реализованы две версии каждого вида коллекций: изменяемая и неизменяемая? Почему бы не ограничиться только изменяемыми версиями?

О: Потому что это заставляет вас явно указать, должна ли коллекция быть изменяемой или неизменяемой. А это означает, что вы можете запретить изменение коллекции, если считаете это нежелательным.

В: Разве это нельзя сделать при помощи `val` и `var`?

О: Нет. `val` и `var` указывают, можно ли заменить ссылку на переменную, хранящуюся в переменной, другой ссылкой после ее инициализации. Даже если переменная, определенная с ключевым словом `val`, содержит ссылку на изменяемую коллекцию, то эта коллекция все равно может обновляться. `val` всего лишь означает, что сама переменная может ссылаться только на этот объект коллекции.

В: Возможно ли создать необновляемое представление изменяемой коллекции?

О: Предположим, имеется множество `MutableSet` с элементами `Int`, присвоенное переменной с именем `x`:

```
val x = mutableSetOf(1, 2)
```

`x` можно присвоить переменной `Set` с именем `y` следующей командой

```
val y: Set<Int> = x
```

Так как `y` является переменной `Set`, объект не удастся обновить без предварительного преобразования в `MutableSet`.

В: И это отличается от использования `toSet`?

О: Да, `toSet` *копирует* коллекцию, поэтому при внесении изменений в исходную коллекцию они не будут отражены в копии.

В: Могу ли я явно создавать и использовать коллекции Java в Kotlin?

О: Да. Kotlin включает различные функции для создания коллекций Java. Например, функция `arrayListOf` создает коллекцию `ArrayList`, а функция `hashMapOf` — коллекцию `HashMap`. Однако следует учитывать, что эти функции создают изменяемые объекты.

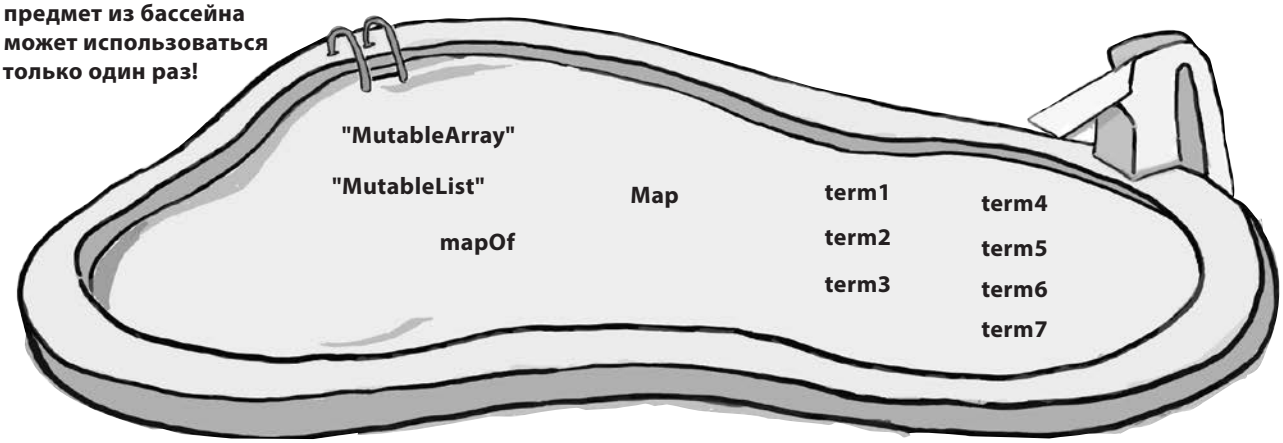
Мы рекомендуем ограничиваться коллекциями Kotlin, которые рассматриваются в этой главе, если только у вас нет веских причин для обратного.

У бассейна

Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: вывести элементы `Map` с именем `glossary`, содержащие определения всех типов коллекций, о которых вы узнали в этой главе.

```
fun main(args: Array<String>) {
    val term1 = "Array"
    val term2 = "List"
    val term3 = "Map"
    val term4 = ……
    val term5 = "MutableMap"
    val term6 = "MutableSet"
    val term7 = "Set"

    val def1 = "Holds values in no particular order."
    val def2 = "Holds key/value pairs."
    val def3 = "Holds values in a sequence."
    val def4 = "Can be updated."
    val def5 = "Can't be updated."
    val def6 = "Can be resized."
    val def7 = "Can't be resized."

    val glossary = ……(……to "$def3 $def4 $def6",
            ……to "$def1 $def5 $def7",
            ……to "$def3 $def4 $def7",
            ……to "$def2 $def4 $def6",
            ……to "$def3 $def5 $def7",
            ……to "$def1 $def4 $def6",
            ……to "$def2 $def5 $def7")
    for ((key, value) in glossary) println("$key: $value")
}
```

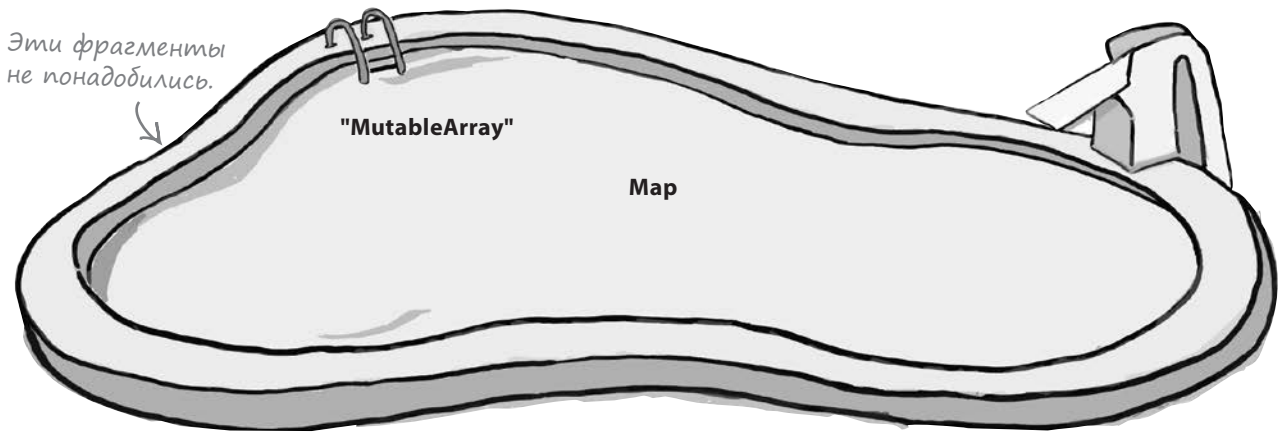
Примечание: каждый предмет из бассейна может использоваться только один раз!



У бассейна. Решение

Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: вывести элементы `Map` с именем `glossary`, содержащие определения всех типов коллекций, о которых вы узнали в этой главе.

```
fun main(args: Array<String>) {
    val term1 = "Array"
    val term2 = "List"
    val term3 = "Map"
    val term4 = "MutableList"
    val term5 = "MutableMap"
    val term6 = "MutableSet"
    val term7 = "Set"

    val def1 = "Holds values in no particular order."
    val def2 = "Holds key/value pairs."
    val def3 = "Holds values in a sequence."
    val def4 = "Can be updated."
    val def5 = "Can't be updated."
    val def6 = "Can be resized."
    val def7 = "Can't be resized."

    val glossary = mapOf( term4 to "$def3 $def4 $def6",
            term7 to "$def1 $def5 $def7",
            term1 to "$def3 $def4 $def7",
            term5 to "$def2 $def4 $def6",
            term2 to "$def3 $def5 $def7",
            term6 to "$def1 $def4 $def6",
            term3 to "$def2 $def5 $def7")
    for ((key, value) in glossary) println("$key: $value")
}
```



Путаница с сообщениями

Ниже приведена короткая программа Kotlin. Один блок в программе пропущен. Ваша задача — сопоставить блоки-кандидаты (слева) с выводом, который вы увидите при подстановке этого блока. Используются не все строки вывода, а некоторые могут использоваться более одного раза. Проведите линию от каждого блока к подходящему варианту вывода.

```
fun main(args: Array<String>) {
    val mList = mutableListOf("Football", "Baseball", "Basketball")

    [ Код блока подставляется сюда. ]

}
```

Соедините каждый блок с одним из вариантов вывода.

Блоки:

```
mList.sort()
println(mList)
```

```
val mMap = mutableMapOf("0" to "Netball")
var x = 0
for (item in mList) {
    mMap.put(x.toString(), item)
}
println(mMap.values)
```

```
mList.addAll(mList)
mList.reverse()
val set = mList.toSet()
println(set)
```

```
mList.sort()
mList.reverse()
println(mList)
```

Варианты вывода:

```
[Netball]
```

```
[Baseball, Basketball, Football]
```

```
[Basketball]
```

```
[Football, Basketball, Baseball]
```

```
{Basketball}
```

```
[Basketball, Baseball, Football]
```

```
{Netball}
```

```
[Football]
```

```
{Basketball, Baseball, Football}
```

```
[Football, Baseball, Basketball]
```

Путаница с сообщениями. Решение

Ниже приведена короткая программа Kotlin. Один блок в программе пропущен. Ваша задача — сопоставить блоки-кандидаты (слева) с выводом, который вы увидите при подстановке этого блока. Используются не все строки вывода, а некоторые могут использоваться более одного раза. Проведите линию от каждого блока к подходящему варианту вывода.

Код блока подставляется сюда.

```
fun main(args: Array<String>) {
    val mList = mutableListOf("Football", "Baseball", "Basketball")

}
```

Блоки:

```
mList.sort()
println(mList)
```

```
val mMap = mutableMapOf("0" to "Netball")
var x = 0
for (item in mList) {
    mMap.put(x.toString(), item)
}
println(mMap.values)
```

```
mList.addAll(mList)
mList.reverse()
val set = mList.toSet()
println(set)
```

```
mList.sort()
mList.reverse()
println(mList)
```

Варианты вывода:

```
[Netball]
```

```
[Baseball, Basketball, Football]
```

```
[Basketball]
```

```
[Football, Basketball, Baseball]
```

```
{Basketball}
```

```
[Basketball, Baseball, Football]
```

```
{Netball}
```

```
[Football]
```

```
{Basketball, Baseball, Football}
```

```
[Football, Baseball, Basketball]
```

Ваш инструментарий Kotlin

Глава 9 осталась позади, а ваш инструментарий пополнился несколькими видами коллекций.

Весь код для этой главы можно загрузить по адресу https://tinyurl.com/HFKotlin.

- Массив, инициализированный значениями `null`, создается функцией `arrayOfNulls`.
- Полезные функции массивов: `sort`, `reverse`, `contains`, `min`, `max`, `sum`, `average`.
- Стандартная библиотека Kotlin содержит готовые классы и функции, объединенные в пакеты.
- Список `List` — коллекция, отслеживающая позицию (индекс) элементов; может содержать повторяющиеся значения.
- Множество `Set` — неупорядоченная коллекция, в которой не может быть дубликатов.
- Ассоциативный массив `Map` — коллекция пар «ключ-значение»; может содержать дубликаты значений, но не дубликаты ключей.
- Коллекции `List`, `Set` и `Map` являются неизменяемыми. `MutableList`, `MutableSet` и `MutableMap` — изменяемые разновидности этих коллекций.
- Списки `List` создаются функцией `listOf`.
- Списки `MutableList` создаются функцией `mutableListOf`.
- Множества `Set` создаются функцией `setOf`.
- Множества `MutableSet` создаются функцией `mutableSetOf`.
- Чтобы проверить наличие дубликатов, множество `Set` сначала проверяет совпадающие значения хеш-кодов, а затем использует операторы `===` и `==` для проверки ссылочного и объектного равенства.
- `Map` создаются функцией `mapOf`, которой передаются пары «ключ-значение».
- `MutableMap` создаются функцией `mutableMapOf`.

На каждый вход знай свой выход

Всем нравится понятный и предсказуемый код. А один из способов написания универсального кода, в котором реже возникают проблемы, заключается в использовании ***обобщений***. В этой главе мы покажем, как **классы коллекций Kotlin используют обобщения**, чтобы вы не смешивали салат и машинное мало. Вы узнаете, как и в каких случаях ***писать собственные обобщенные классы, интерфейсы и функции*** и как ограничить **обобщенный тип** конкретным супертипом. Наконец, научитесь пользоваться ***ковариантностью и контравариантностью,*** чтобы ВЫ сами управляли поведением своего обобщенного типа.

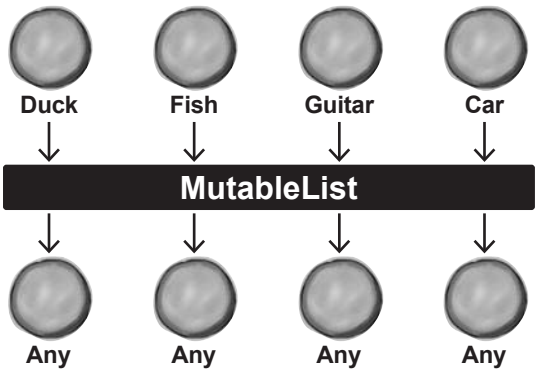
В коллекциях используются обобщения

Как вы узнали в предыдущих главах, каждый раз, когда вы явно объявляете тип коллекции, следует указывать как тип коллекции, так и тип содержащихся в ней элементов. Например, следующий код определяет переменную для хранения ссылки на список `MutableList` с элементами `String`:

```
val x: MutableList<String>
```

Тип элемента определяется в угловых скобках <> — это означает, что он использует **обобщения**. Обобщения позволяют писать код, безопасный по отношению к типам. Что это такое, спросите вы? То, что не позволяет поместить объект `Volkswagen` в список с элементами `Duck`. Компилятор знает, что в `MutableList<Duck>` можно добавлять только объекты `Duck`, а следовательно, на стадии компиляции будет выявляться большее количество проблем.

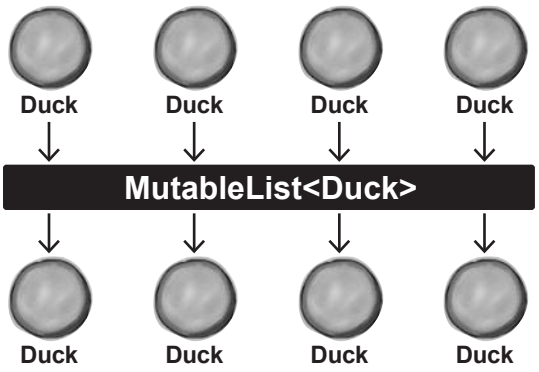
БЕЗ обобщений на ВХОД будут поступать ссылки на объекты Duck, Fish, Guitar и Car...



Без обобщений было бы невозможно объявить, какие типы объектов должны содержаться в MutableList.

...а на ВЫХОДЕ вы получите ссылку с типом Any.

С обобщениями на ВХОД поступают только ссылки на объекты Duck...



С обобщениями вы можете быть уверены в том, что ваша коллекция содержит только элементы правильного типа. Не нужно беспокоиться о том, что кто-то затолкает объект Pumpkin в MutableList<Duck> или что вы получите что-то, кроме Duck.

...и на ВЫХОДЕ они остаются ссылками с типом Duck.

Как определяется MutableList

Заглянем в электронную документацию, чтобы понять, как определяется класс `MutableList` и как он использует обобщения. Нас сейчас интересуют два ключевых момента: объявление интерфейса и определение функции `add`.

Документация коллекций (или в чем смысл «E»?)

Упрощенная версия определения `MutableList`:

«E» — условное обозначение РЕАЛЬНОГО типа, который будет использоваться при объявлении MutableList.

MutableList наследуется от интерфейсов List и MutableCollection. Тип, указанный вами для MutableList (значение «E»), автоматически используется в качестве типа List и MutableCollection.

```
interface MutableList<E> : List<E>, MutableCollection<E> {

    fun add(index: Int, element: E): Unit

    //...

}
```

Какой бы тип ни скрывался за «E», он определяет, какие объекты можно добавлять в MutableList.

`MutableList` использует «E» как условное обозначение типа элемента, который должен храниться и возвращаться коллекцией. Когда вы видите «E» в документации, мысленно подставьте на это место тот тип, который должен храниться в коллекции.

Например, `MutableList<String>` означает, что «E» превращается в «String» в любой функции или объявлении переменной, в которых используется «E». А `MutableList<Duck>` означает, что все вхождения «E» превращаются в «Duck».

Давайте разберемся с этим.

Часто Задаваемые Вопросы

В: Значит, `MutableList` — это не класс?

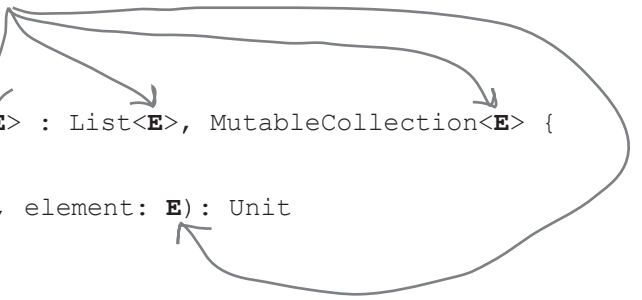
О: Нет, это интерфейс. Когда вы создаете объект `MutableList` функцией `mutableListOf`, система создает *реализацию* этого интерфейса. Впрочем, при использовании для вас важно лишь то, чтобы он обладал всеми свойствами и функциями, определенными в интерфейсе `MutableList`.

Использование параметров типа с MutableList

Когда вы пишете код вида

```
val x: MutableList<String>
```

это означает, что MutableList:



```
interface MutableList<E> : List<E>, MutableCollection<E> {

    fun add(index: Int, element: E): Unit

    //Ваш код
}
```

Он интерпретируется компилятором в виде

```
interface MutableList<String> : List<String>, MutableCollection<String> {

    fun add(index: Int, element: String): Unit

    //...
}
```

Иначе говоря, «E» заменяется *реальным* типом (также называемым *параметром*), который используется при определении MutableList. И это объясняет, почему функция add не позволит добавить в коллекцию объекты с типом, несовместимым с «E». Таким образом, при создании MutableList<String> функция add позволит добавлять объекты String. А если вы создадите MutableList с типом Duck, функция add позволит добавлять объекты Duck.

Что можно делать с обобщенным классом или интерфейсом

Ниже перечислены важнейшие операции, которые можно выполнять при использовании класса или интерфейса с обобщенными типами:

Создание экземпляра обобщенного класса.
При создании коллекции (например, `MutableList`) необходимо задать тип объектов, которые могут в ней храниться, или поручить компилятору определить этот тип по типам элементов:

```
val duckList: MutableList<Duck>
duckList = mutableListOf(Duck("Donald"), Duck("Daisy"), Duck("Daffy"))

val list = mutableListOf("Fee", "Fi", "Fum")
```

Создание функции, получающей обобщенный тип.
Вы можете создать функцию с обобщенным параметром — просто укажите его тип точно так же, как сделали бы это с обычным параметром:

```
fun quack(ducks: MutableList<Duck>) {
    //...
}
```

Создание функции, возвращающей обобщенный тип.
Функция также может возвращать обобщенный тип. Например, следующий код возвращает список `MutableList` с элементами `Duck`:

```
fun getDucks(breed: String): MutableList<Duck> {
    //Получение объектов Duck нужного вида
}
```

По поводу обобщений остается целый ряд важных вопросов — например, как определять собственные обобщенные классы и интерфейсы? И как *полиморфизм* работает для обобщенных типов? А если у вас имеется список `MutableList<Animal>`, то что произойдет при попытке присвоить ему `MutableList<Dog>`?

Чтобы получить ответы на эти и другие вопросы, мы создадим приложение, в котором используются обобщенные типы.

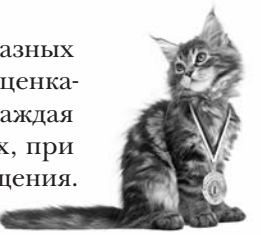
Что мы собираемся сделать

Наше приложение предназначено для работы с информацией о домашних животных. Мы создадим несколько видов животных, проведем для них выставки и создадим продавцов определенных типов животных. А поскольку мы будем использовать обобщения, приложение гарантирует, что каждая выставка и каждый продавец будет работать только с определенным видом животных.

Основные этапы того процесса, который нам предстоит реализовать:

1. **Создание иерархии Pet.**
   Создадим иерархию классов для создания животных трех типов: кошек, собак и рыбок.

2. **Создание класса Contest.**
   Класс `Contest` используется для проведения выставок разных типов животных. С его помощью мы будем управлять оценками участников для определения победителя. А чтобы каждая выставка ограничивалась конкретным видом животных, при определении класса `Contest` будут использоваться обобщения.



3. **Создание иерархии Retailer.**
   Мы создадим интерфейс `Retailer` и конкретные реализации этого интерфейса с именами `CatRetailer`, `DogRetailer` и `FishRetailer`. Обобщения гарантируют, что каждая разновидность продавца `Retailer` продает только животных конкретного типа, поэтому купить кошку `Cat` у продавца рыбок `FishRetailer` не выйдет.

4. **Создание класса Vet.**
   Наконец, мы создадим класс `Vet`, чтобы за каждой выставкой можно было закрепить ветеринара. Класс `Vet` будет определяться с использованием обобщений, чтобы каждый ветеринар `Vet` специализировался на конкретном виде животных `Pet`.

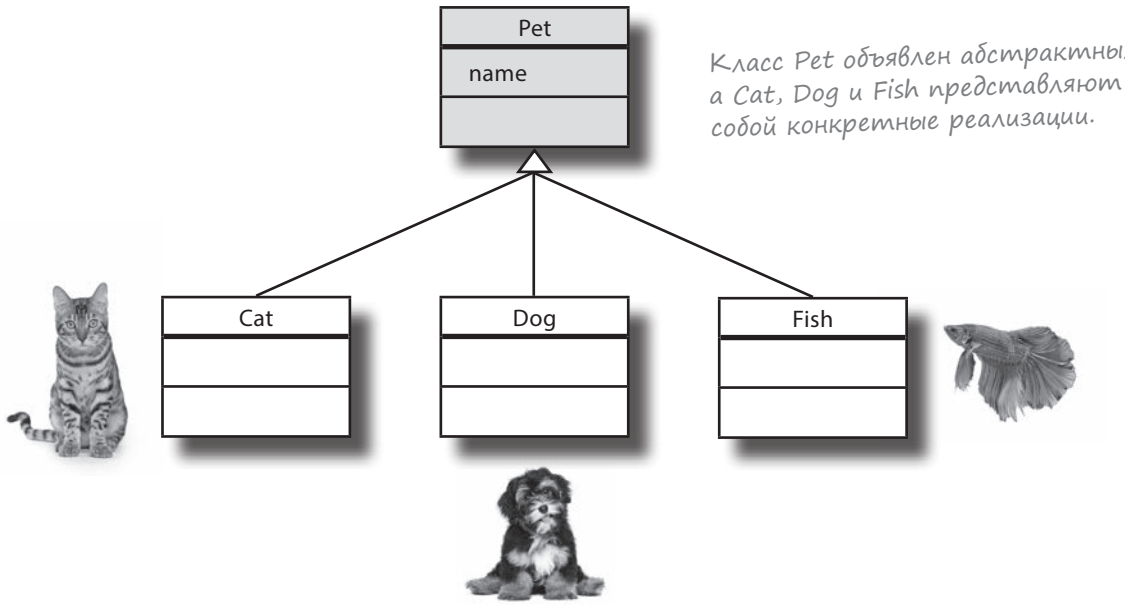
Начнем с создания иерархии классов животных.

Создание иерархии классов Pet

Наша иерархия будет состоять из четырех классов: класс `Pet`, который будет помечен как абстрактный, и конкретные подклассы с именами `Cat`, `Dog` и `Fish`. В класс `Pet` включается свойство `name`, которое будет наследоваться конкретными подклассами.

Класс `Pet` помечается как абстрактный, потому что в приложении должны создаваться объекты только подклассов `Pet` — например, объекты `Cat` или `Dog`, а как вы узнали в главе 6, помечая класс как абстрактный, вы запрещаете создание экземпляров этого класса.

Иерархия классов выглядит так:



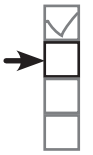
Класс Pet объявлен абстрактным, а Cat, Dog и Fish представляют собой конкретные реализации.

А это код иерархии классов:

```
abstract class Pet(var name: String)

class Cat(name: String) : Pet(name)

class Dog(name: String) : Pet(name)

class Fish(name: String) : Pet(name)
```

Каждый подкласс Pet содержит свойство name (унаследованное от Pet), которое задается в конструкторе класса.

Затем мы создадим класс `Contest` для хранения информации о выставках для разных видов животных.



Определение класса Contest

Класс `Contest` будет использоваться для управления оценками на выставке и определения победителя. Класс содержит одно свойство с именем `scores` и две функции — `addScore` и `getWinners`.

Каждая выставка ограничивается животными определенного типа. Например, в выставке кошек участвуют только кошки, а в конкурсе рыбок — только рыбки. Чтобы обеспечить соблюдение этого правила, мы воспользуемся обобщениями.

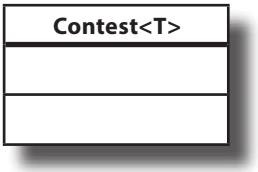
Объявление использования обобщенного типа в Contest

Чтобы указать, что класс использует обобщенный тип, укажите имя типа в угловых скобках после имени класса. В данном случае обобщенный тип обозначается «T». Считайте, что «T» заменяет *реальный* тип, который используется каждым отдельным объектом `Contest`.

Код выглядит так:

```
class Contest<T> {
    //...
}
```

<T> после имени класса сообщает компилятору, что T является обобщенным типом.



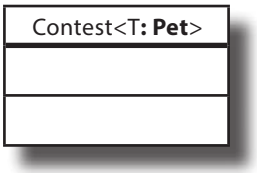
В качестве имени обобщенного типа может использоваться любой допустимый идентификатор, но по общепринятому соглашению (которое нужно соблюдать) стоит использовать «T». Впрочем, у этого правила есть исключение: если вы пишете класс коллекции или интерфейс, используйте обозначение «E» (сокращение от «Element»), а для ключей и значений массивов — «K» или «V» (сокращения от «Key» и «Value»).

Ограничение T определенным суперклассом

В приведенном примере `T` можно заменить любым реальным типом при создании экземпляра класса. Тем не менее вы можете установить ограничения для `T`, указав нужный вам *тип*. Например, следующий код сообщает компилятору, что тип `T` должен быть разновидностью `Pet`:

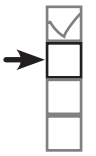
```
class Contest<T: Pet> {
    //...
}
```

T — обобщенный тип, которым должен быть Pet или один из его подклассов.



Этот код означает, что вы можете создавать объекты `Contest` для хранения `Cat`, `Fish` или `Pet`, но не для объектов `Bicycle` или `Begonia`.

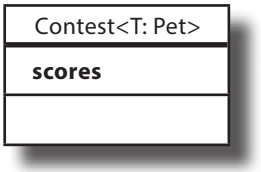
Теперь добавим в класс `Contest` свойство `scores`.



Добавление свойства scores

Свойство `scores` используется для хранения информации о том, какую оценку получил тот или иной участник. Мы будем использовать карту `MutableMap`, ключами которой будут участники, а значениями — полученные оценки. Так как каждый участник является объектом типа `T`, а оценка относится к типу `Int`, свойство `scores` имеет тип `MutableMap<T, Int>`. Если создать объект `Contest<Cat>` для участников `Cat`, тип свойства `scores` принимает вид `MutableMap<Cat, Int>`, но для объекта `Contest<Pet>` тип `scores` автоматически превращается в `MutableMap<Pet, Int>`.

Обновленный код класса `Contest`:




```
class Contest<T: Pet> {
    val scores: MutableMap<T, Int> = mutableMapOf()
    //...
}
```

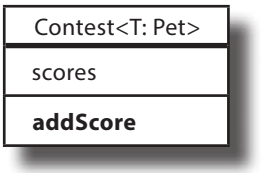
Определяет MutableMap с ключами T и значениями Int, где T — обобщенный тип Pet, к которому относится Contest.

После добавления свойства `scores` добавим функции `addScore` и `getWinners`.

Создание функции addScore

Функция `addScore` должна добавлять оценку участника в массив `MutableMap` с именем `scores`. Участник и оценка передаются функции в параметрах; если оценка равна 0 или выше, функция добавляет данные в `MutableMap` как пару «ключ-значение».

Код функции выглядит так:




```
class Contest<T: Pet> {
    val scores: MutableMap<T, Int> = mutableMapOf()

    fun addScore(t: T, score: Int = 0) {
        if (score >= 0) scores.put(t, score)
    }

    //...
}
```

Добавить участника и его оценку в MutableMap при условии, что оценка больше либо равна 0.

Наконец, добавим функцию `getWinners`.

Создание функции getWinners

Функция `getWinners` должна возвращать участников с наивысшей оценкой. Она получает значение максимальной оценки из свойства `scores`, а затем возвращает всех участников с этой оценкой в форме `MutableSet`. Так как каждый участник имеет обобщенный тип `T`, функция должна вернуть возвращаемый тип `MutableSet<T>`.

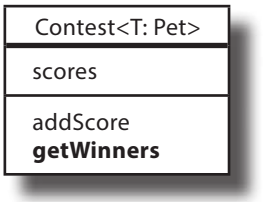
Код функции `getWinners`:

```
fun getWinners(): MutableSet<T> {
    val highScore = scores.values.max()
    val winners: MutableSet<T> = mutableSetOf()
    for ((t, score) in scores) {
        if (score == highScore) winners.add(t)
    }
    return winners
}
```

Получить максимальное значение из scores.

Включить участников с максимальной оценкой в MutableSet.

Вернуть множество MutableSet с данными победителей.



А теперь приведем полный код класса `Contest`:

```
class Contest<T: Pet> {
    val scores: MutableMap<T, Int> = mutableMapOf()

    fun addScore(t: T, score: Int = 0) {
        if (score >= 0) scores.put(t, score)
    }

    fun getWinners(): MutableSet<T> {
        val highScore = scores.values.max()
        val winners: MutableSet<T> = mutableSetOf()
        for ((t, score) in scores) {
            if (score == highScore) winners.add(t)
        }
        return winners
    }
}
```

Мы добавим этот класс в новое приложение через несколько страниц.

Класс `Contest` готов, используем его для создания нескольких объектов.

Создание объектов Contest

Чтобы создать объект `Contest`, следует указать тип объектов, с которыми он будет работать, и вызвать конструктор. Например, следующий код создает объект `Contest<Cat>` с именем `catContest` для работы с объектами `Cat`:

```
val catContest = Contest<Cat>()
```

Создает объект Contest для работы с объектами Cat.

Это означает, что вы можете добавить объекты `Cat` в его свойство `scores` и использовать функцию `getWinners` для возвращения множества `MutableSet` с элементами `Cat`:

```
catContest.addScore(Cat("Fuzz Lightyear"), 50)
catContest.addScore(Cat("Katsu"), 45)
val topCat = catContest.getWinners().first()
```

getWinners() возвращает множество MutableSet<Cat>, так как мы указали, что catContest работает с Cat.

А поскольку `Contest` использует обобщения, компилятор не позволяет передать ему ссылки на другие объекты, кроме `Cat`. Например, следующий код компилироваться не будет:

```
catContest.addScore(Dog("Fido"), 23)
```

Компилятор не позволяет добавить в Contest<Cat> другие объекты, кроме Cat, поэтому эта строка не компилируется.

Однако `Contest<Pet>` может получать любую разновидность `Pet`:

```
val petContest = Contest<Pet>()
petContest.addScore(Cat("Fuzz Lightyear"), 50)
petContest.addScore(Fish("Finny McGraw"), 56)
```

Так как Contest<Pet> работает с объектами Pet, участники могут относиться к любому из подклассов Pet.

Компилятор может определить обобщенный тип

В некоторых случаях компилятор может определить обобщенный тип по имеющейся информации. Если вы создадите переменную типа `Contest<Dog>`, компилятор автоматически определит, что любой передаваемый объект `Contest` является `Contest<Dog>` (если только вы не укажете обратное). Например, следующий код создает объект `Contest<Dog>` и присваивает его `dogContest`:

```
val dogContest: Contest<Dog>
dogContest = Contest()
```

Здесь можно использовать Contest() вместо Contest<Dog>(), так как компилятор может определить тип объекта по типу переменной.

Там, где это возможно, компилятор также может определить обобщенный тип по параметрам конструктора. Например, если бы мы использовали обобщенный параметр типа в первичном конструкторе класса `Contest`:

```
class Contest<T: Pet>(t: T) {...}
```

компилятор смог бы определить, что следующий код создает `Contest<Fish>`:

```
val contest = Contest(Fish("Finny McGraw"))
```

То же самое, что создание Contest вызовом Contest<Fish>(Fish("Finny McGraw")). Часть <Fish> можно опустить, так как компилятор определяет ее по аргументу конструктора.

Обобщенные функции под увеличительным стеклом

До настоящего момента мы показывали, как определить функцию, использующую обобщенный тип внутри определения класса. А что, если вы захотите определить функцию с обобщенным типом за пределами класса? Или если нужно, чтобы функция внутри класса использовала обобщенный тип, не включенный в определение класса?

Если вы хотите определить функцию с собственным обобщенным типом, это можно сделать, объявив обобщенный тип как часть определения функции. Например, следующий код определяет функцию с именем `listPet` с обобщенным типом `T`, который ограничивается разновидностями `Pet`. Функция получает параметр `T` и возвращает ссылку на объект `MutableList<T>`:

Для функций, которые объявляют собственный обобщенный тип, <T: Pet> ставится перед именем функции.

```
fun <T: Pet> listPet(t: T): MutableList<T> {
    println("Create and return MutableList")
    return mutableListOf(t)
}
```

Обратите внимание: при таком объявлении обобщенной функции тип должен объявляться в угловых скобках *перед* именем функции:

```
fun <T: Pet> listPet...
```

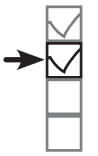
При вызове функции необходимо указать тип объекта, с которым должна работать функция. Например, следующий код вызывает функцию `listPet`, и при помощи угловых скобок указывает, что она должна использоваться с объектами `Cat`:

```
val catList = listPet<Cat>(Cat("Zazzles"))
```

Однако обобщенный тип можно опустить, если компилятор может определить его по аргументам функции. Например, следующий код допустим, потому что компилятор может определить, что функция `listPet` используется с `Cat`:

```
val catList = listPet(Cat("Zazzles"))
```

Эти два вызова функций делают одно и то же, так как компилятор может определить, что функция должна работать с объектами Cat.



Создание проекта Generics

Итак, теперь вы знаете, как создать класс, использующий обобщения. Давайте добавим его в новое приложение.

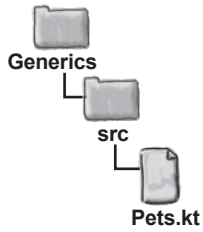
Создайте новый проект Kotlin для JVM с именем «Generics». Создайте новый файл Kotlin с именем *Pets.kt*: выделите папку *src*, откройте меню File и выберите команду New → Kotlin File/Class. Введите имя файла «Pets» и выберите вариант File из группы Kind.

Затем обновите свою версию *Pets.kt*, чтобы она соответствовала нашей:

```
abstract class Pet(var name: String)

class Cat(name: String) : Pet(name)

class Dog(name: String) : Pet(name)

class Fish(name: String) : Pet(name)

class Contest<T: Pet> {
    val scores: MutableMap<T, Int> = mutableMapOf()

    fun addScore(t: T, score: Int = 0) {
        if (score >= 0) scores.put(t, score)
    }

    fun getWinners(): MutableSet<T> {
        val winners: MutableSet<T> = mutableSetOf()
        val highScore = scores.values.max()
        for ((t, score) in scores) {
            if (score == highScore) winners.add(t)
        }
        return winners
    }
}
```

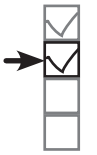
Добавление иерархии Pet.

Добавление класса Contest.



Продолжение на следующей странице

Продолжение...

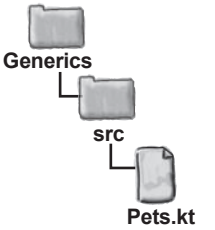



```
fun main(args: Array<String>) {
    val catFuzz = Cat("Fuzz Lightyear")
    val catKatsu = Cat("Katsu")
    val fishFinny = Fish("Finny McGraw")

    val catContest = Contest<Cat>()
    catContest.addScore(catFuzz, 50)
    catContest.addScore(catKatsu, 45)
    val topCat = catContest.getWinners().first()
    println("Cat contest winner is ${topCat.name}")

    val petContest = Contest<Pet>()
    petContest.addScore(catFuzz, 50)
    petContest.addScore(fishFinny, 56)
    val topPet = petContest.getWinners().first()
    println("Pet contest winner is ${topPet.name}")
}
```

Создание двух объектов Cat и объекта Fish.

Создание объекта Contest, предназначенного только для Cat.

Создание объекта Contest для Pet; такой объект будет принимать любые подклассы Pet.



Тест-драйв

При выполнении этого кода в окне вывода IDE отображается следующий результат:

```
Cat contest winner is Fuzz Lightyear
Pet contest winner is Finny McGraw
```

После того как вы справитесь со следующим упражнением, мы перейдем к иерархии `Retailer`.

Часто Задаваемые Вопросы

В: Может ли обобщенный тип быть null-совместимым?

О: Да. Если имеется функция, которая возвращает обобщенный тип и вы хотите, чтобы этот тип был null-совместимым, просто поставьте `?` после обобщенного возвращаемого типа:

```
class MyClass<T> {
    fun myFun(): T?
}
```

В: Может ли класс иметь более одного обобщенного типа?

О: Да. Чтобы определить несколько обобщенных типов, заключите их в угловые скобки и разделите запятыми. Например, класс `MyMap` с обобщенными типами `K` и `V` определяется так:

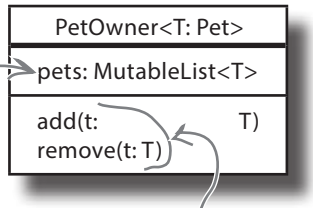
```
class MyMap<K, V> {
    //...
}
```

У бассейна

Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: создать класс с именем `PetOwner`, получающий обобщенные типы `Pet`, который затем будет использоваться для создания новой реализации `PetOwner<Cat>`, содержащей ссылки на два объекта `Cat`.

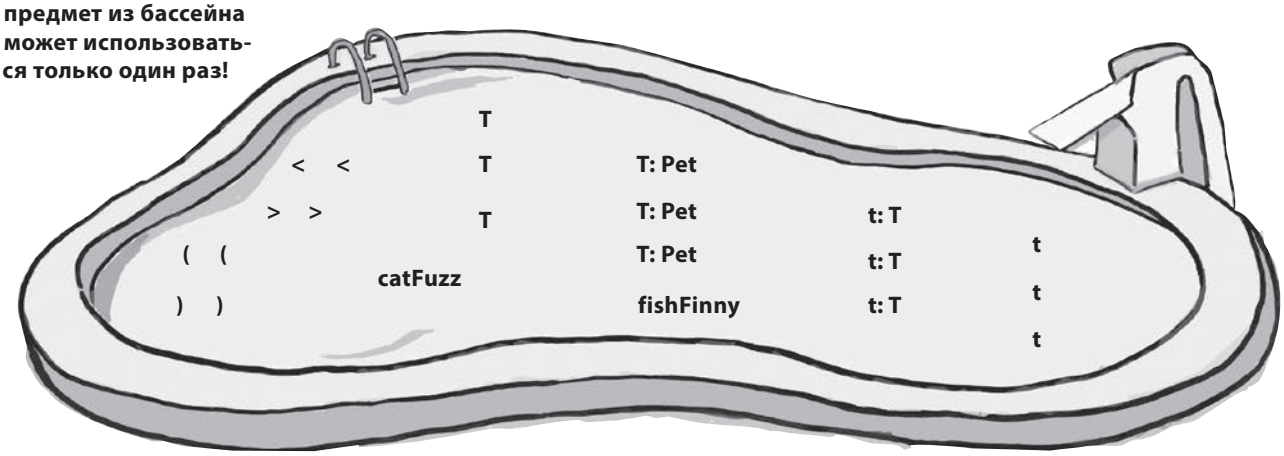
```
class PetOwner_____ {
    val pets = mutableListOf(___)

    fun add(_____) {
        pets.add(___)
    }

    fun remove(_____) {
        pets.remove(___)
    }
}

fun main(args: Array<String>) {
    val catFuzz = Cat("Fuzz Lightyear")
    val catKatsu = Cat("Katsu")
    val fishFinny = Fish("Finny McGraw")
    val catOwner = PetOwner_____
    catOwner.add(catKatsu)
}
```

pets содержит ссылку на каждое животное, принадлежащее владельцу. Список инициализируется значением, переданным конструктору PetOwner.



Функции add и remove используются для обновления свойства pets. Функция add добавляет ссылку, а функция remove удаляет ее.

Примечание: каждый предмет из бассейна может использоваться только один раз!



У бассейна. Решение

Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: создать класс с именем `PetOwner`, получающий обобщенные типы `Pet`, который затем будет использоваться для создания новой реализации `PetOwner<Cat>`, содержащей ссылки на два объекта `Cat`.

PetOwner<T: Pet>
pets: MutableList<T>
add(t: T) remove(t: T)

Задает обобщенный тип. *Конструктор.*

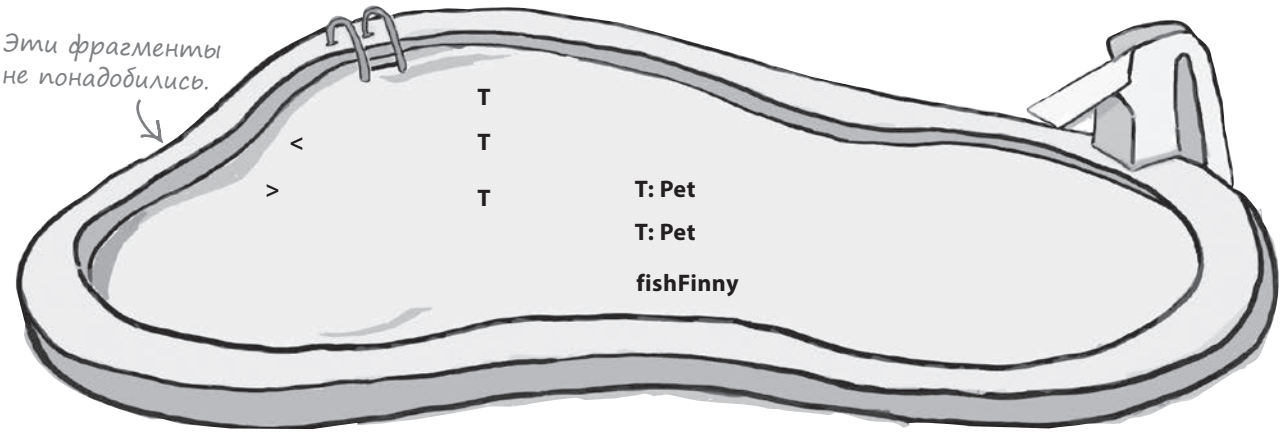
```
class PetOwner <T: Pet>(t: T) {
    val pets = mutableListOf( t )

    fun add( t: T ) {
        pets.add( t )
    }

    fun remove( t: T ) {
        pets.remove( t )
    }
}

fun main(args: Array<String>) {
    val catFuzz = Cat("Fuzz Lightyear")
    val catKatsu = Cat("Katsu")
    val fishFinny = Fish("Finny McGraw")
    val catOwner = PetOwner (catFuzz)
    catOwner.add(catKatsu)
}
```

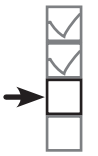
Создает MutableList<T>.

Добавление/удаление значений T.

Создает PetOwner<Cat> и инициализирует pets ссылкой на catFuzz.

Эти фрагменты не понадобились.



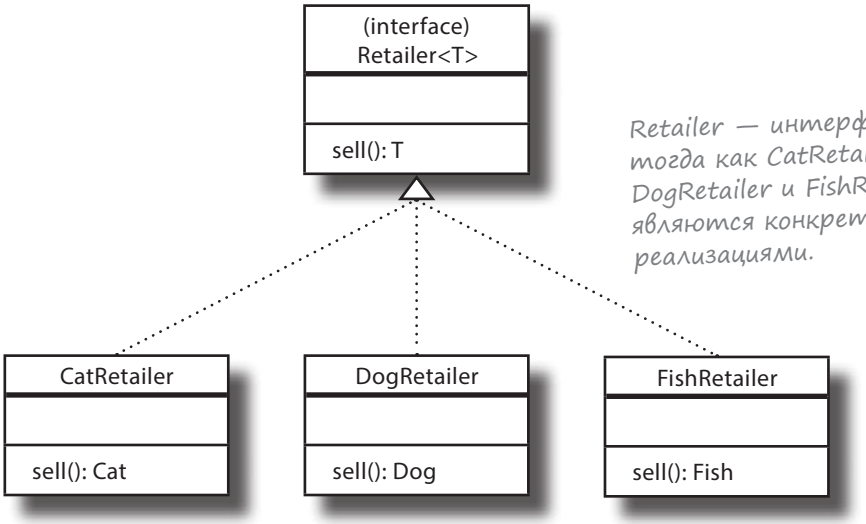


Иерархия Retailer

Мы используем созданные ранее классы `Pet` для определения иерархии продавцов, продающих разные виды животных. Для этого мы определим интерфейс `Retailer` с функцией `sell` и три конкретных класса с именами `CatRetailer`, `DogRetailer` и `FishRetailer`, реализующие этот интерфейс.

Каждый тип продавца должен продавать животных конкретного типа. Например, `CatRetailer` может продавать только объекты `Cat`, а `DogRetailer` — только объекты `Dog`. Чтобы обеспечить это требование, мы при помощи обобщений определим тип объектов, с которым работает каждый класс. Добавим в интерфейс `Retailer` обобщенный тип `T` и укажем, что функция `sell` должна возвращать объекты этого типа. Так как все классы `CatRetailer`, `DogRetailer` и `FishRetailer` реализуют этот интерфейс, каждый класс должен подставить на место обобщенного типа `T` «реальный» тип объектов.

Иерархия классов, которую мы будем использовать, выглядит так:



Retailer — интерфейс, тогда как CatRetailer, DogRetailer и FishRetailer являются конкретными реализациями.

Итак, мы рассмотрели иерархию классов. Теперь напишем для нее код, начиная с интерфейса `Retailer`.

Часто Задаваемые Вопросы

В: Почему мы не используем конкретный класс `PetRetailer`?

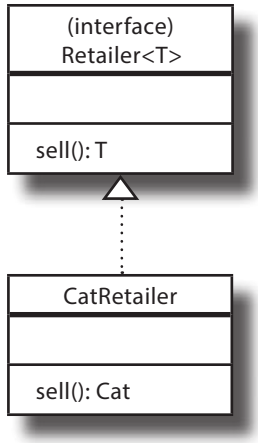
О: Вполне возможно, что в реальном приложении мы бы включили класс `PetRetailer` для продажи любых видов животных. Здесь же мы различаем разные виды `Retailer`, чтобы показать вам важные подробности, касающиеся обобщений.

Определение интерфейса Retailer

Интерфейс `Retailer` должен указать, что он использует обобщенный тип `T`, который используется как возвращаемый тип функции `sell`.

Код интерфейса:

```
interface Retailer<T> {
    fun sell(): T
}
```



Классы `CatRetailer`, `DogRetailer` и `FishRetailer` должны реализовать интерфейс `Retailer` с указанием типа объектов, с которыми должен работать каждый класс. Например, класс `CatRetailer` работает только с `Cat`, так что мы определим его в следующем виде:

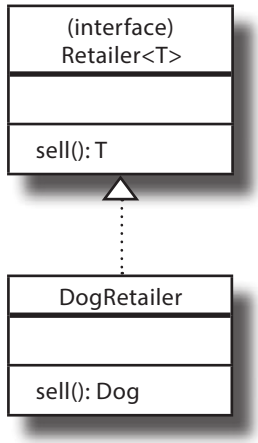
```
class CatRetailer : Retailer<Cat> {
    override fun sell(): Cat {
        println("Sell Cat")
        return Cat("")
    }
}
```

Класс CatRetailer реализует интерфейс Retailer для работы с Cat. Это означает, что функция sell() должна возвращать Cat.

Аналогичным образом класс `DogRetailer` работает с `Dog`, поэтому мы определяем его следующим образом:

```
class DogRetailer : Retailer<Dog> {
    override fun sell(): Dog {
        println("Sell Dog")
        return Dog("")
    }
}
```

DogRetailer заменяет обобщенный тип Retailer классом Dog, поэтому его функция sell() должна возвращать Dog.



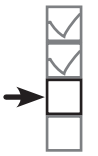
Каждая реализация интерфейса `Retailer` должна задавать тип объекта, с которым она работает; для этого определение «T» в интерфейсе заменяется реальным типом. Например, реализация `CatRetailer` заменяет «T» классом «Cat», поэтому ее функция `sell` должна возвращать `Cat`. Если вы попробуете использовать для возвращаемого типа `sell` что-либо помимо `Cat` (или подкласса `Cat`), код не будет компилироваться:

```
class CatRetailer : Retailer<Cat> {
    override fun sell(): Dog = Dog("")
}
```

Код не будет компилироваться, потому что функция sell() класса CatRetailer должна возвращать Cat, а Dog не является подклассом Cat.

Таким образом, обобщения фактически ограничивают возможности использования типов классами, в результате чего код становится более целостным и надежным.

Код классов продавцов готов, переходим к созданию объектов.



Мы можем создать объекты CatRetailer, DogRetailer и FishRetailer...

Как вы догадались, можно создать объект `CatRetailer`, `DogRetailer` и `FishRetailer` и присвоить его переменной, явно объявив тип переменной или поручив компилятору определить его по присвоенному значению. В следующем коде этот способ используется для создания двух переменных `CatRetailer` и присваивания каждой переменной объекта `CatRetailer`:

```
val catRetailer1 = CatRetailer()
val catRetailer2: CatRetailer = CatRetailer()
```

...но как насчет полиморфизма?

Так как `CatRetailer`, `DogRetailer` и `FishRetailer` реализуют интерфейс `Retailer`, у нас *должна* быть возможность создать переменную типа `Retailer` (с совместимым параметром типа) и присвоить ей один из подклассов. И все будет работать, если присвоить объект `CatRetailer` переменной `Retailer<Cat>`, или же `DogRetailer` — переменной `Retailer<Dog>`:

```
val dogRetailer: Retailer<Dog> = DogRetailer()
val catRetailer: Retailer<Cat> = CatRetailer()
```

Эти строки допустимы, потому что DogRetailer реализует Retailer<Dog>, а CatRetailer реализует Retailer<Cat>.

Но при попытке присвоить один из этих объектов `Retailer<Pet>` код не будет компилироваться:

```
val petRetailer: Retailer<Pet> = CatRetailer()
```

Не компилируется, хотя CatRetailer является реализацией Retailer<Cat>, а Cat является подклассом Pet.

И хотя `CatRetailer` является разновидностью `Retailer`, а `Cat` является разновидностью `Pet`, текущая версия нашего кода не позволит присвоить объект `Retailer<Cat>` переменной `Retailer<Pet>`. Переменная `Retailer<Pet>` принимает только объект `Retailer<Pet>`. Ни `Retailer<Cat>`, ни `Retailer<Dog>`, а только `Retailer<Pet>`.

Такое поведение вроде бы противоречит самой идее полиморфизма. Впрочем, не все так плохо: **вы можете откорректировать обобщенный тип в интерфейсе `Retailer`, чтобы управлять тем, какие типы объектов может принимать переменная `Retailer<Pet>`**.

out и ковариантность обобщенного типа

Если обобщенный тип является ковариантным, это означает, что вы можете использовать подтип вместо супертипа.

Если вы хотите, чтобы обобщенные подтипы можно было использовать вместо обобщенного супертипа, снабдите обобщенный тип префиксом **out**. В нашем примере `Retailer<Cat>` (подтип) должен присваиваться `Retailer<Pet>` (супертип), поэтому обобщенный тип `T` снабжается в интерфейсе `Retailer` префиксом `out`:

```
interface Retailer<out T> {
    fun sell(): T
}
```

Префикс out.

Если обобщенный тип помечается префиксом `out`, этот обобщенный тип называется **ковариантным**. Иначе говоря, это означает, что подтип может использоваться вместо супертипа.

С этим изменением переменной `Retailer<Pet>` можно будет присваивать объекты `Retailer`, работающие с подтипами `Pet`. Например, следующий код после этого будет компилироваться:

```
val petRetailer: Retailer<Pet> = CatRetailer()
```

Префикс out в интерфейсе Retailer означает, что вы теперь можете присвоить Retailer<Cat> переменной Retailer<Pet>.

В общем случае обобщенный тип класса или интерфейса может быть снабжен префиксом `out`, если класс содержит функции, у которых он является возвращаемым типом, или если класс содержит `val`-свойства этого типа. Тем не менее префикс `out` не может использоваться, если класс имеет параметры функций или `var`-свойства этого обобщенного типа.

Обобщенный тип с префиксом out может использоваться только в «выходных» позициях — например, в возвращаемом типе функции. Однако он не может использоваться во «входной» позиции, поэтому функция не может получать ковариантный тип как значение параметра.

Коллекции определяются с использованием ковариантных типов

Префикс `out` используется не только обобщенными классами и интерфейсами, которые вы определяете сами. Он также широко используется во встроенном коде Kotlin — например, в коллекциях.

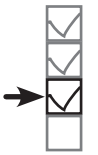
Так, коллекция `List` определяется кодом следующего вида:

```
public interface List<out E> ... { ... }
```

Это означает, что вы можете присвоить список `List` для `Cats` списку `List` для `Pet`, и код будет успешно компилироваться:

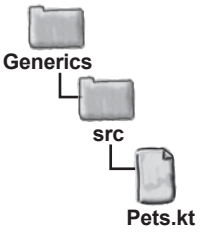
```
val catList: List<Cat> = listOf(Cat(""), Cat(""))
val petList: List<Pet> = catList
```

Вы узнали, как сделать обобщенный тип ковариантным при помощи ключевого слова `out`. Добавим вновь написанный код в проект.



Обновление проекта Generics

Обновите свою версию файла *Pets.kt* из проекта Generics, чтобы она соответствовала нашей (изменения выделены жирным шрифтом):




```
abstract class Pet(var name: String)
class Cat(name: String) : Pet(name)
class Dog(name: String) : Pet(name)
class Fish(name: String) : Pet(name)

class Contest<T: Pet> {
    val scores: MutableMap<T, Int> = mutableMapOf()

    fun addScore(t: T, score: Int = 0) {
        if (score >= 0) scores.put(t, score)
    }

    fun getWinners(): MutableSet<T> {
        val winners: MutableSet<T> = mutableSetOf()
        val highScore = scores.values.max()
        for ((t, score) in scores) {
            if (score == highScore) winners.add(t)
        }
        return winners
    }
}
```

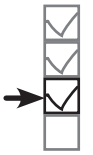
Добавление интерфейса Retailer.

```
interface Retailer<out T> {
    fun sell(): T
}

class CatRetailer : Retailer<Cat> {
    override fun sell(): Cat {
        println("Sell Cat")
        return Cat("")
    }
}
```

Добавление классов CatRetailer и DogRetailer.

```
class DogRetailer : Retailer<Dog> {
    override fun sell(): Dog {
        println("Sell Dog")
        return Dog("")
    }
}
```

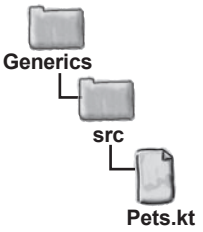
Продолжение на следующей странице.



Продолжение...

```
class FishRetailer : Retailer<Fish> {
    override fun sell(): Fish {
        println("Sell Fish")
        return Fish("")
    }
}
```

Добавление класса FishRetailer.




```
fun main(args: Array<String>) {
    val catFuzz = Cat("Fuzz Lightyear")
    val catKatsu = Cat("Katsu")
    val fishFinny = Fish("Finny McGraw")

    val catContest = Contest<Cat>()
    catContest.addScore(catFuzz, 50)
    catContest.addScore(catKatsu, 45)
    val topCat = catContest.getWinners().first()
    println("Cat contest winner is ${topCat.name}")

    val petContest = Contest<Pet>()
    petContest.addScore(catFuzz, 50)
    petContest.addScore(fishFinny, 56)
    val topPet = petContest.getWinners().first()
    println("Pet contest winner is ${topPet.name}")

    val dogRetailer: Retailer<Dog> = DogRetailer()
    val catRetailer: Retailer<Cat> = CatRetailer()
    val petRetailer: Retailer<Pet> = CatRetailer()
    petRetailer.sell()
}
```

Создание объектов Retailer.

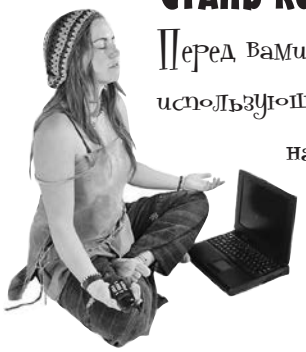
Тест-драйв

При выполнении кода в окне вывода IDE отображается следующий текст:

```
Cat contest winner is Fuzz Lightyear
Pet contest winner is Finny McGraw
Sell Cat
```

Итак, мы разобрались с тем, как объявляются ковариантные обобщенные типы с префиксом `out`. Проверьте свои силы на следующем упражнении.

СТАНЬ компилятором



Перед вами пять классов и интерфейсов, использующих обобщения. Представьте себя на месте компилятора и определите, какие из них будут нормально компилироваться. Если какой-то код не компилируется, то почему?

A

```
interface A<out T> {
    fun myFunction(t: T)
}
```

B

```
interface B<out T> {
    val x: T
    fun myFunction(): T
}
```

C

```
interface C<out T> {
    var y: T
    fun myFunction(): T
}
```

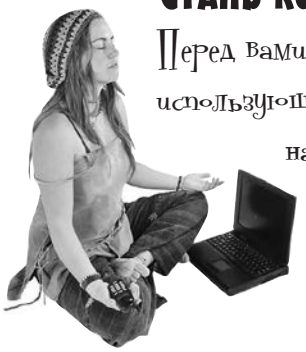
D

```
interface D<out T> {
    fun myFunction(str: String): T
}
```

E

```
abstract class E<out T>(t: T) {
    val x = t
}
```

СТАНЬ компилятором. Решение



Перед вами пять классов и интерфейсов, использующих обобщения. Представьте себя на месте компилятора и определите, какие из них будут нормально компилироваться. Если какой-то код не компилируется, то почему?

A

```
interface A<out T> {
    fun myFunction(t: T)
}
```

Этот код не компилируется, так как ковариантный тип T не может использоваться в качестве параметра функции.

B

```
interface B<out T> {
    val x: T
    fun myFunction(): T
}
```

Этот код успешно компилируется.

C

```
interface C<out T> {
    var y: T
    fun myFunction(): T
}
```

Этот код не компилируется, так как ковариантный тип T не может использоваться в типе var-свойства.

D

```
interface D<out T> {
    fun myFunction(str: String): T
}
```

Этот код успешно компилируется.

E

```
abstract class E<out T>(t: T) {
    val x = t
}
```

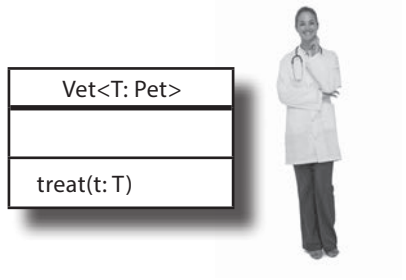
Этот код успешно компилируется.

Класс Vet

Как уже упоминалось, на каждую выставку должен быть назначен ветеринар, который при необходимости окажет медицинскую помощь участникам. Так как ветеринары могут специализироваться на конкретных видах животных, мы создадим класс `Vet` с обобщенным типом `T` и определим функцию `treat`, получающую аргумент указанного типа. Также нужно указать, что обобщенный тип `T` должен быть разновидностью `Pet`, чтобы вы не могли создать объект ветеринара `Vet` для лечения объектов `Planet` или `Broccoli`.

Класс `Vet` выглядит так:

```
class Vet<T: Pet> {
    fun treat(t: T) {
        println("Treat Pet ${t.name}")
    }
}
```



Внесем изменения в класс `Contest` и добавим поле для хранения объекта `Vet`.

Связывание объекта Vet с Contest

С каждым объектом выставки `Contest` должен быть связан объект ветеринара `Vet`, поэтому мы добавим свойство `Vet` в конструктор `Contest`. Обновленный код `Contest` выглядит так:

Vet<T> добавляется в конструктор Contest, чтобы объект Contest нельзя было создать без назначения объекта Vet.

```
class Contest<T: Pet>(var vet: Vet<T>) {
    val scores: MutableMap<T, Int> = mutableMapOf()

    fun addScore(t: T, score: Int = 0) {
        if (score >= 0) scores.put(t, score)
    }

    fun getWinners(): MutableSet<T> {
        val winners: MutableSet<T> = mutableSetOf()
        val highScore = scores.values.max()
        for ((t, score) in scores) {
            if (score == highScore) winners.add(t)
        }
        return winners
    }
}
```

А теперь создадим несколько объектов `Vet` и свяжем их с объектами `Contest`.

Создание объектов Vet

Объекты `Vet` можно создавать точно так же, как объекты `Contest`: вы указываете тип объектов, с которыми должен работать каждый объект `Vet`. Например, следующий код создает три объекта — по одному для типов `Vet<Cat>`, `Vet<Fish>` и `Vet<Pet>`:

```
val catVet = Vet<Cat>()
val fishVet = Vet<Fish>()
val petVet = Vet<Pet>()
```

Каждый объект `Vet` может работать с определенными видами `Pet`. Например, `Vet<Cat>` работает только с `Cat`, тогда как `Vet<Pet>` может работать с любыми разновидностями `Pet` — как с `Cat`, так и с `Fish`. Однако `Vet<Cat>` не может работать ни с чем, кроме `Cat`, так что объект `Fish` ему не подойдет:

```
catVet.treat(Cat("Fuzz Lightyear"))
petVet.treat(Cat("Katsu"))
petVet.treat(Fish("Finny McGraw"))
catVet.treat(Fish("Finny McGraw"))
```

Vet<Cat> и Vet<Pet> могут работать с Cat.

Vet<Pet> может работать с Fish.

Эта строка не компилируется, так как Vet<Cat> не может работать с Fish.

Посмотрим, что произойдет при попытке передать объекты `Vet` при создании `Contest`.

Передача Vet конструктору Contest

Класс `Contest` получает один параметр — объект `Vet`, работающий с типом `Pet`, для которого предназначен объект `Contest`. А это означает, что `Vet<Cat>` может передаваться `Contest<Cat>`, а `Vet<Pet>` — `Contest<Pet>`:

```
val catContest = Contest<Cat>(catVet)
val petContest = Contest<Pet>(petVet)
```

Но тут возникает проблема. `Vet<Pet>` может работать с любыми разновидностями `Pet`, включая `Cat`, однако **мы не сможем передать `Vet<Pet>` конструктору `Contest<Cat>`, так как код не будет компилироваться**:

```
val catContest = Contest<Cat>(petVet)
```

Несмотря на то что Vet<Pet> может работать с Cat, Contest<Cat> не примет Vet<Pet>, и эта строка не будет компилироваться.

Что же делать в такой ситуации?

in и контрвариантность обобщенных типов

Если обобщенный тип является контравариантным, это означает, что вы можете использовать супертип вместо подтипа. Контравариантность противоположна ковариантности.

Мы хотим, чтобы в нашем примере `Contest<Cat>` можно было передать `Pet<Vet>` вместо `Pet<Cat>`. Иначе говоря, мы хотим иметь возможность использовать обобщенный супертип вместо обобщенного подтипа.

В такой ситуации обобщенный тип, используемый классом `Vet`, снабжается префиксом **`in`**. Этот префикс является полной противоположностью `out`: если `out` позволяет использовать обобщенный подтип вместо супертипа (например, присвоить `Retailer<Cat>` переменной `Retailer<Pet>`), то `in` позволяет использовать обобщенный супертип вместо подтипа. Таким образом, добавление префикса `in` к обобщенному типу класса `Vet`

Префикс in.

```
class Vet<in T: Pet> {
    fun treat(t: T) {
        println("Treat Pet ${t.name}")
    }
}
```

означает, что мы можем использовать `Vet<Pet>` вместо `Vet<Cat>`. Следующий код теперь успешно компилируется:

```
val catContest = Contest<Cat>(Vet<Pet>())
```

Префикс in в классе Vet означает, что Vet<Pet> может использоваться вместо Vet<Cat>, поэтому код теперь успешно компилируется.

Когда перед обобщенным типом ставится префикс `in`, это означает, что обобщенный тип является **контравариантным**. Другими словами, супертип может использоваться вместо подтипа.

В частности, обобщенный тип класса или интерфейса может снабжаться префиксом `in`, если класс содержит функции, использующие его в качестве типа параметра. При этом `in` не может использоваться, если какие-либо функции класса используют его в качестве возвращаемого типа или если этот тип используется какими-либо свойствами (независимо от того, объявляются они с ключевым словом `val` или `var`).

Другими словами, обобщенный тип с префиксом in может использоваться только во «входных» позициях — скажем, в значении параметра функции. Он не может использоваться в «выходных» позициях.

Должен ли Vet<Cat> ВСЕГДА принимать Vet<Pet>?

Прежде чем ставить перед обобщенным типом класса или интерфейса префикс `in`, стоит подумать над тем, должен ли обобщенный подтип принимать обобщенный супертип в любой ситуации. Например, это позволяет присвоить объект `Vet<Pet>` переменной `Vet<Cat>` — возможно, в каких-то ситуациях это будет нежелательно:

```
val catVet: Vet<Cat> = Vet<Pet>()
```

Эта строка компилируется, так как класс Vet использует префикс in для T.

К счастью, в подобных ситуациях можно настроить обстоятельства, в которых обобщенный тип проявляет контравариантное поведение. Посмотрим, как это делается.

Обобщенный тип может обладать локальной контрвариантностью

Как было показано ранее, обобщенный тип с префиксом `in`, являющийся частью объявления класса или интерфейса, становится глобально контрвариантным. Тем не менее это поведение также может ограничиваться отдельными свойствами или функциями.

Допустим, вы хотите, чтобы ссылка `Vet<Pet>` могла использоваться вместо `Vet<Cat>`, но *только* в месте передачи ее `Contest<Cat>` в конструкторе. Чтобы добиться желаемого эффекта, удалите префикс `in` из обобщенного типа в классе `Vet` и включите его в свойство `vet` в конструкторе `Contest`.

Если обобщенный тип не имеет префикса in или out, он называется инвариантным. Инвариантный тип может принимать ссылки только этого конкретного типа.

Код выглядит так:

Префикс in удаляется из класса Vet...

```
class Vet<in T: Pet> {
    fun treat(t: T) {
        println("Treat Pet ${t.name}")
    }
}
class Contest<T: Pet>(var vet: Vet<in T>) {
    ...
}
```

...и добавляется в конструктор Contest. Это означает, что T является контрвариантным, но только в конструкторе Contest.

С этими изменениями `Vet<Pet>` можно будет передавать `Contest<Cat>`:

```
val catContest = Contest<Cat>(Vet<Pet>())
```

Эта строка компилируется, так как Vet<Pet> может использоваться вместо Vet<Cat> в конструкторе Contest<Cat>.

Однако компилятор не позволит присвоить объект `Vet<Pet>` переменной `Vet<Cat>`, потому что обобщенный тип класса `Vet` не обладает глобальной контрвариантностью:

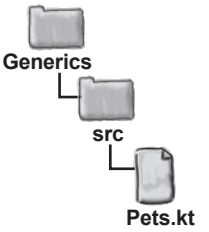
```
val catVet: Vet<Cat> = Vet<Pet>()
```

А эта строка компилироваться не будет, так как Vet<Pet> не может глобально использоваться вместо Vet<Cat>.

После того как вы научились пользоваться контрвариантностью, добавим код `Vet` в проект Generics.

Обновление проекта Generics

Обновите свою версию файла *Pets.kt* из проекта Generics, чтобы она соответствовала нашей (изменения выделены жирным шрифтом):




```
abstract class Pet(var name: String)
class Cat(name: String) : Pet(name)
class Dog(name: String) : Pet(name)
class Fish(name: String) : Pet(name)

class Vet<T: Pet> {
    fun treat(t: T) {
        println("Treat Pet ${t.name}")
    }
}

class Contest<T: Pet>(var vet: Vet<in T>) {
    val scores: MutableMap<T, Int> = mutableMapOf()

    fun addScore(t: T, score: Int = 0) {
        if (score >= 0) scores.put(t, score)
    }

    fun getWinners(): MutableSet<T> {
        val winners: MutableSet<T> = mutableSetOf()
        val highScore = scores.values.max()
        for ((t, score) in scores) {
            if (score == highScore) winners.add(t)
        }
        return winners
    }
}

interface Retailer<out T> {
    fun sell(): T
}

class CatRetailer : Retailer<Cat> {
    override fun sell(): Cat {
        println("Sell Cat")
        return Cat("")
    }
}
```

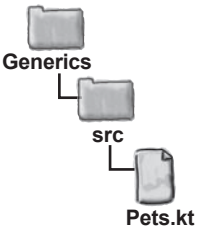
Добавляем класс Vet.

В класс Contest добавляется конструктор.

Продолжение на следующей странице.

Продолжение...


```
class DogRetailer : Retailer<Dog> {
    override fun sell(): Dog {
        println("Sell Dog")
        return Dog("")
    }
}

class FishRetailer : Retailer<Fish> {
    override fun sell(): Fish {
        println("Sell Fish")
        return Fish("")
    }
}
```




```
fun main(args: Array<String>) {
    val catFuzz = Cat("Fuzz Lightyear")
    val catKatsu = Cat("Katsu")
    val fishFinny = Fish("Finny McGraw")

    val catVet = Vet<Cat>()
    val fishVet = Vet<Fish>()
    val petVet = Vet<Pet>()

    catVet.treat(catFuzz)
    petVet.treat(catKatsu)
    petVet.treat(fishFinny)

    val catContest = Contest<Cat>(catVet)
    catContest.addScore(catFuzz, 50)
    catContest.addScore(catKatsu, 45)
    val topCat = catContest.getWinners().first()
    println("Cat contest winner is ${topCat.name}")
```

Создаем несколько объектов Vet.

Объекты Vet работают со своими разновидностями Pet.

Vet<Cat> связывается с Contest<Cat>.

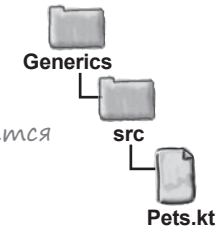
Продолжение на следующей странице.

Продолжение...

```
val petContest = Contest<Pet>(petVet)
petContest.addScore(catFuzz, 50)
petContest.addScore(fishFinny, 56)
val topPet = petContest.getWinners().first()
println("Pet contest winner is ${topPet.name}")

val fishContest = Contest<Fish>(petVet)

val dogRetailer: Retailer<Dog> = DogRetailer()
val catRetailer: Retailer<Cat> = CatRetailer()
val petRetailer: Retailer<Pet> = CatRetailer()
petRetailer.sell()
}
```

Vet<Pet> связывается с Contest<Pet>.

Vet<Pet> связывается с Contest<Fish>.



Тест-драйв

При выполнении кода в окне вывода IDE отображается следующий текст:

```
Treat Pet Fuzz Lightyear
Treat Pet Katsu
Treat Pet Finny McGraw
Cat contest winner is Fuzz Lightyear
Pet contest winner is Finny McGraw
Sell Cat
```

Часто Задаваемые Вопросы

В: Разве нельзя просто объявить свойство `vet` в `Contest` с типом `Vet<Pet>`?

О: Нет. В этом случае свойство `vet` будет принимать только `Vet<Pet>`. И хотя свойство `vet` *можно* объявить локально ковариантным

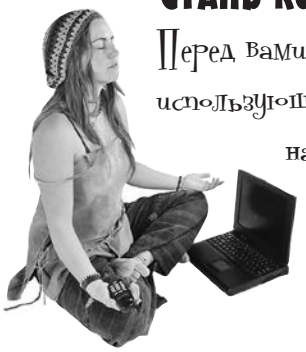
```
var vet: Vet<out Pet>
```

это означало бы, что на кошачью выставку `Contest<Cat>` будет назначен ветеринар для рыбок `Vet<Fish>`. Добром это не кончится.

В: Обобщения в Kotlin работают не так, как в Java. Правильно?

О: Да, правильно. В Java обобщенные типы всегда инвариантны, но при помощи шаблонов можно обойти некоторые возникающие проблемы. С другой стороны, Kotlin предоставляет гораздо больше возможностей для управления, так как вы можете сделать обобщенные типы ковариантными, контрвариантными или оставить инвариантными.

СТАНЬ компилятором



Перед вами четыре класса и интерфейса, использующие обобщения. Представьте себя на месте компилятора и определите, какие из них будут нормально компилироваться. Если какой-то код не компилируется, то почему?

A

```
class A<in T>(t: T) {
    fun myFunction(t: T) { }
}
```

B

```
class B<in T>(t: T) {
    val x = t
    fun myFunction(t: T) { }
}
```

C

```
abstract class C<in T> {
    fun myFunction(): T { }
}
```

D

```
class E<in T>(t: T) {
    var y = t
    fun myFunction(t: T) { }
}
```

→ Ответы на с. 352.

Возьми в руку карандаш

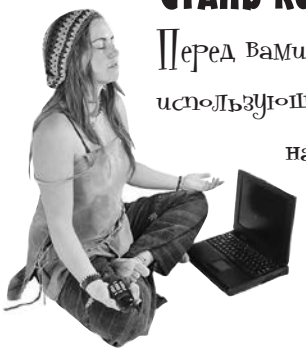
Ниже приведен полный исходный код из файла Kotlin. Этот код не компилируется. Какие строки не компилируются? Какие изменения вы бы внесли в определения классов и интерфейсов, чтобы они успешно компилировались?

Примечание: изменять функцию `main` нельзя.

```
//Типы продуктов
open class Food

class VeganFood: Food()

//Продавцы
interface Seller<T>

class FoodSeller: Seller<Food>

class VeganFoodSeller: Seller<VeganFood>

//Потребители
interface Consumer<T>

class Person: Consumer<Food>

class Vegan: Consumer<VeganFood>

fun main(args: Array<String>) {
    var foodSeller: Seller<Food>
    foodSeller = FoodSeller()
    foodSeller = VeganFoodSeller()

    var veganFoodConsumer: Consumer<VeganFood>
    veganFoodConsumer = Vegan()
    veganFoodConsumer = Person()
}
```

Ответы на с. 353.

СТАНЬ компилятором. Решение



Перед вами четыре класса и интерфейса, использующие обобщения. Представьте себя на месте компилятора и определите, какие из них будут нормально компилироваться. Если какой-то код не компилируется, то почему?

A

```
class A<in T>(t: T) {
    fun myFunction(t: T) { }
}
```

Этот код успешно компилируется, потому что контравариантный тип T может использоваться в качестве параметра типа конструктора или функции.

B

```
class B<in T>(t: T) {
    val x = t
    fun myFunction(t: T) { }
}
```

Код не компилируется, потому что T не может использоваться как тип свойства val.

C

```
abstract class C<in T> {
    fun myFunction(): T { }
}
```

Этот код не компилируется, потому что T не может использоваться в качестве возвращаемого типа функции.

D

```
class E<in T>(t: T) {
    var y = t
    fun myFunction(t: T) { }
}
```

Этот код не компилируется, потому что T не может использоваться в качестве типа var-свойства.

Возьми в руку карандаш
Решение

Ниже приведен полный исходный код из файла Kotlin. Этот код не компилируется. Какие строки не компилируются? Какие изменения вы бы внесли в определения классов и интерфейсов, чтобы они успешно компилировались?

Примечание: изменять функцию `main` нельзя.

```
//Типы продуктов
open class Food

class VeganFood: Food()

//Продавцы
interface Seller<out T>

class FoodSeller: Seller<Food>

class VeganFoodSeller: Seller<VeganFood>

//Потребители
interface Consumer<in T>

class Person: Consumer<Food>

class Vegan: Consumer<VeganFood>

fun main(args: Array<String>) {
    var foodSeller: Seller<Food>
    foodSeller = FoodSeller()
    foodSeller = VeganFoodSeller()

    var veganFoodConsumer: Consumer<VeganFood>
    veganFoodConsumer = Vegan()
    veganFoodConsumer = Person()
}
```

Эта строка не компилируется, потому что в ней Seller<VeganFood> присваивается Seller<Food>. Чтобы она компилировалась, необходимо поставить префикс out перед T в интерфейсе Seller.

Эта строка не компилируется, потому что в ней Consumer<Food> присваивается Consumer<VeganFood>. Чтобы она компилировалась, необходимо поставить префикс in перед T в интерфейсе Consumer.

Ваш инструментарий Kotlin

Глава 10 осталась позади, а ваш инструментарий пополнился обобщениями.

Весь код для этой главы можно загрузить по адресу https://tinyurl.com/HFKotlin.

ГЛАВА 10

- Обобщения позволяют писать универсальный безопасный по отношению к типам код. Коллекции (такие, как `MutableList`) используют обобщения.
- Обобщенный тип определяется в угловых скобках <>:

```
class Contest<T>
```

- Обобщенный тип ограничивается указанным супертипом:

```
class Contest<T: Pet>
```

- При создании экземпляра класса с обобщенным типом «реальный» тип указывается в угловых скобках:

```
Contest<Cat>
```

- Компилятор старается вычислить обобщенный тип, если это возможно.
- Вы можете определить функцию, использующую обобщенный тип, за пределами объявления класса или функцию с другим обобщенным типом:

```
fun <T> listPet(): List<T>{
    ...
}
```

- Обобщенный тип называется инвариантным, если он принимает ссылки только этого конкретного типа. Обобщенные типы инвариантны по умолчанию.
- Обобщенный тип называется ковариантным, если вы можете использовать подтип вместо супертипа. Чтобы указать, что тип является ковариантным, поставьте перед ним префикс `out`.
- Обобщенный тип называется контравариантным, если вы можете использовать супертип вместо подтипа. Чтобы указать, что тип является контравариантным, поставьте перед ним префикс `in`.

Обработка кода как данных

Хотите писать еще более гибкий и мощный код? Тогда вам понадобятся **лямбда-выражения**. *Лямбда-выражение,* или просто *лямбда,* представляет собой блок кода, который можно передавать как объект. В этой главе вы узнаете, ***как определить лямбда-выражение***, ***присвоить его переменной***, а затем ***выполнить его код***. Вы узнаете о **функциональных типах** и о том, как они используются для написания **функций высшего порядка**, использующих лямбда-выражения для параметров или возвращаемых значений. А попутно вы узнаете, как ***синтаксический сахар подсластит вашу программистскую жизнь.***

Знакомство с лямбда-выражениями

В этой книге вы научились пользоваться встроенными функциями Kotlin и создавать собственные функции. И хотя мы уже прошли долгий путь, это только начало. В Kotlin существуют функции, *еще более мощные*, чем те, которые вам встречались, но чтобы пользоваться ими, необходимо изучить **лямбда-выражения**.

Лямбда-выражение, или **лямбда**, представляет собой тип объекта, который содержит блок кода. Лямбда-выражение можно присвоить переменной точно так же, как любой другой объект, или передать его функции, которая затем выполнит содержащийся в нем код. А это означает, что **лямбды могут использоваться для передачи специализированного поведения более общей функции**.

Такое использование лямбда-выражений особенно полезно при работе с коллекциями. Например, пакет *collections* содержит встроенную функцию `sortBy`, которая предоставляет обобщенную реализацию сортировки `MutableList`: вы указываете, *как* функция должна сортировать коллекцию, передавая ей лямбда-выражение с описанием критерия.

Что мы собираемся сделать

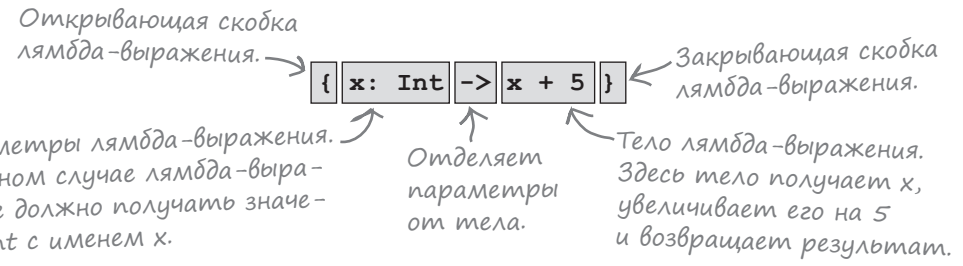
Прежде чем описывать использование лямбд со встроенными функциями, мы хотим подробнее объяснить, как работают лямбда-выражения, поэтому в этой главе будут рассмотрены следующие вопросы:

1. **Определение лямбда-выражения.**
   Вы узнаете, как выглядит лямбда-выражение, как присвоить его переменной, к какому типу оно относится и как выполнить содержащийся в нем код.
2. **Создание функции высшего порядка.**
   Вы научитесь создавать функции с лямбда-параметром и использовать лямбда-выражения с возвращаемым значением функции.

Для начала нужно разобраться, как выглядит лямбда-выражение.

Как выглядит код лямбда-выражения

Мы напишем простое лямбда-выражение, которое прибавляет 5 к значению параметра `Int`. Вот как выглядит лямбда-выражение для этой задачи:

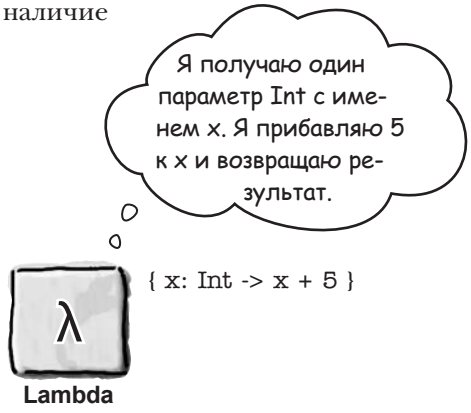


Лямбда-выражение начинается и завершается фигурными скобками `{}`. Все лямбда-выражения определяются в фигурных скобках, поэтому наличие скобок обязательно.

В фигурных скобках лямбда-выражение определяет один параметр `Int` с именем `x`, для чего используется обозначение `x: Int`. Лямбда-выражения могут иметь один параметр (как в нашем случае), несколько параметров или не иметь ни одного параметра.

За определением параметра следует **`->`** — эта часть отделяет параметры от тела лямбда-выражения. По сути вы говорите: «Параметры закончились, теперь к делу!»

Наконец, за `->` следует тело лямбда-выражения — в данном случае `x + 5`. Это тот код, который должен выполняться при выполнении лямбда-выражения. Тело может состоять из нескольких строк; последнее вычисленное выражение в теле используется как возвращаемое значение лямбда-выражения.



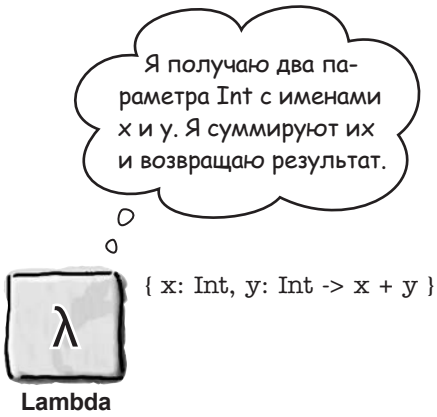
В приведенном примере лямбда-выражение получает значение `x` и возвращает `x + 5`. Происходит почти то же самое, что при написании функции:

```
fun addFive(x: Int) = x + 5
```

не считая того, что лямбда-выражения не обладают именем, то есть являются анонимными.

Как упоминалось ранее, лямбда-выражения могут получать несколько параметров. Например, следующее лямбда-выражение получает два параметра `Int`, `x` и `y`, и возвращает результат `x + y`:

```
{ x: Int, y: Int -> x + y }
```



Если лямбда-выражение не получает параметров, `->` можно опустить. Например, следующее лямбда-выражение не получает параметров и просто возвращает строку «Pow!»:

```
{ "Pow!" }
```

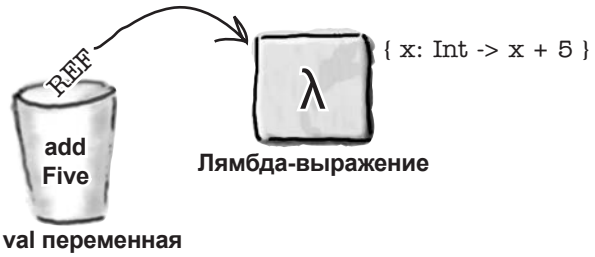
Эта лямбда не имеет параметров, поэтому -> можно опустить.

Теперь, когда вы знаете, как выглядят лямбда-выражения, посмотрим, как присвоить их переменной.

Присваивание лямбд переменной

Лямбда-выражение присваивается переменной точно так же, как любой другой объект: вы определяете переменную с ключевым словом `val` или `var`, а затем присваиваете ей лямбда-выражение. Например, следующий код присваивает лямбду новой переменной с именем `addFive`:

```
val addFive = { x: Int -> x + 5 }
```



Переменная `addFive` определена с ключевым словом `val`, поэтому ее не удастся обновить другим лямбда-выражением. Чтобы переменная обновлялась, она должна быть определена с ключевым словом `var`:

```
var addFive = { x: Int -> x + 5 }
addFive = { y: Int -> 5 + y }
```

Здесь addFive можно присвоить новое лямбда-выражение, потому что переменная определяется с ключевым словом var.

Присваивая переменной лямбда-выражение, вы присваиваете блок кода, а не результат выполнения этого кода. Чтобы выполнить код в лямбда-выражении, необходимо явно выполнить его.

Выполнение кода лямбда-выражения

Чтобы выполнить лямбда-выражение, вызовите его функцию `invoke` и передайте значения необходимых параметров. Например, следующий код определяет переменную с именем `addInts` и присваивает ей лямбда-выражение, которое суммирует два параметра `Int`. Затем код выполняет лямбда-выражение, передает значения параметров 6 и 7 и присваивает результат новой переменной с именем `result`:

```
val addInts = { x: Int, y: Int -> x + y }
val result = addInts.invoke(6, 7)
```

Лямбда-выражение также можно выполнить в сокращенной форме:

```
val result = addInts(6, 7)
```

Эта команда эквивалентна следующей:

```
val result = addInts.invoke(6, 7)
```

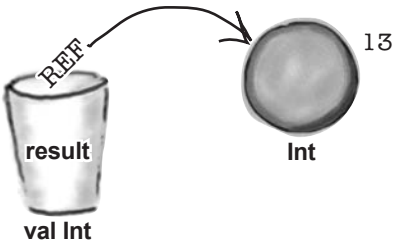
но с более компактным кодом. Это означает «Выполнить лямбда-выражение, хранящееся в переменной *addInts*, со значениями параметров 6 и 7».

А теперь посмотрим, что происходит при выполнении лямбда-выражений.

РАССЛАБЬТЕСЬ

Лямбда-выражения кажутся вам немного странными? Не беспокойтесь.

Не торопитесь, основательно проработайте материал главы — и у вас все получится.



Что происходит при выполнении лямбда-выражений

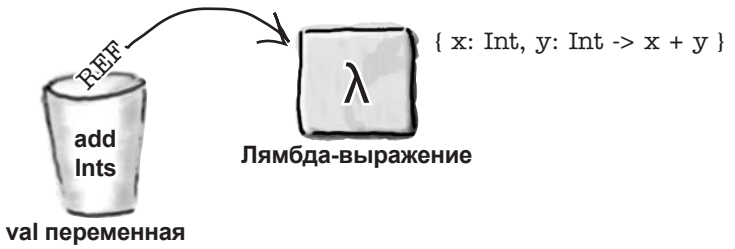
При выполнении этого кода

```
val addInts = { x: Int, y: Int -> x + y }
val result = addInts(6, 7)
```

происходит следующее:

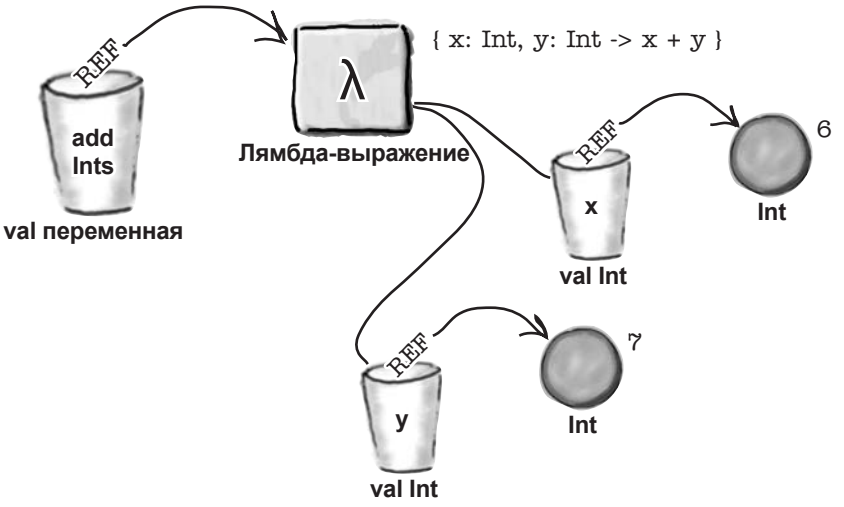
1 **val addInts = { x: Int, y: Int –> x + y }**

Команда создает лябмда-выражение со значением `{ x: Int, y: Int -> x + y }`. Ссылка на лямбда-выражение сохраняется в новой переменной с именем `addInts`.



2 val result = **addInts(6, 7)**

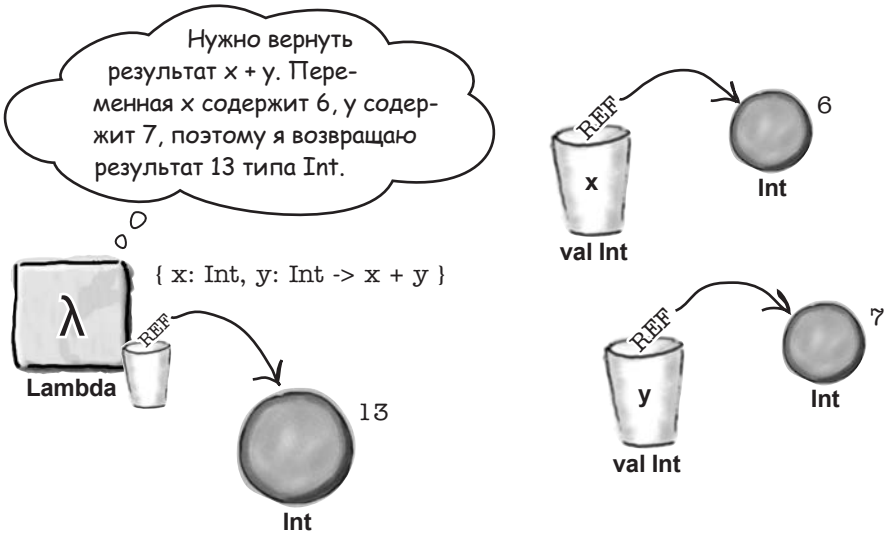
Команда выполняет лямбда-выражение, на которое ссылается переменная `addInts`, и передает ему значения 6 и 7. Значение 6 присваивается параметру `x` лямбда-выражения, а 7 — параметру `y` лямбда-выражения.



История продолжается...

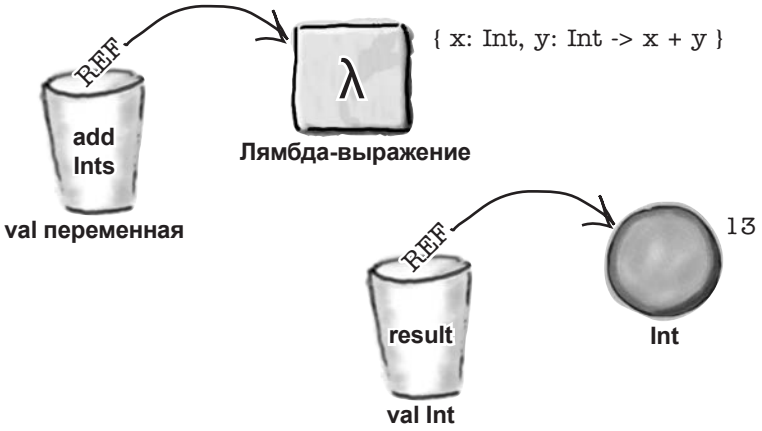
3 val addInts = { x: Int, y: Int -> **x + y** }

Выполняется тело лямбда-выражения, в котором вычисляется значение `x + y`. Лямбда-выражение создает объект `Int` со значением `13` и возвращает ссылку на него.



4 **val result =** addInts(6, 7)

Значение, возвращенное лямбда-выражением, присваивается новой переменной `Int` с именем `result`.



Теперь вы знаете, что происходит при выполнении лямбд.
Рассмотрим типы лямбда-выражений.

У лямбда-выражений есть тип

Лямбда-выражения, как и все остальные разновидности объектов, обладают типом. Однако тип лямбда-выражения отличается тем, что в нем не указывается имя класса, реализуемого лямбда-выражением. Вместо этого указываются типы параметров лямбда-выражения и возвращаемое значение.

Тип лямбда-выражения также называется функциональным типом.

Тип лямбда-выражения имеет следующую форму:

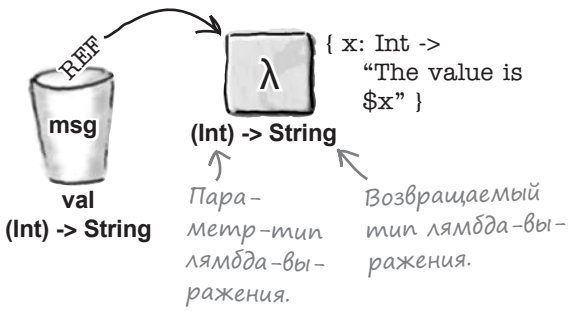
```
(parameters) -> return_type
```

Если лямбда-выражение получает один параметр `Int` и возвращает `String`

```
val msg = { x: Int -> "The value is $x" }
```

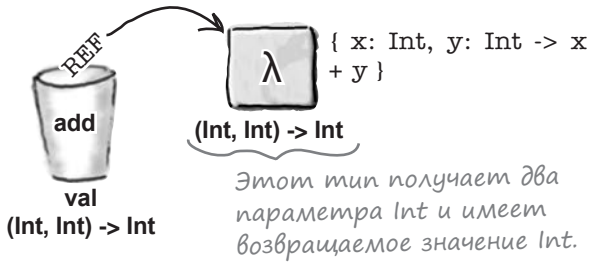
то его тип выглядит так:

```
(Int) -> String
```



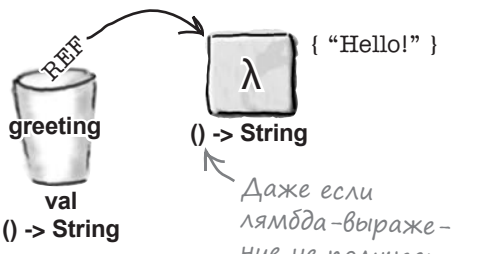
Когда вы присваиваете лямбда-выражение переменной, компилятор определяет тип переменной по присвоенному ей лямбда-выражению, как в приведенном примере. Однако как и для других типов объектов, тип переменной также может быть определен явно. В следующем коде определяется переменная с именем `add`, в которой может храниться ссылка на лямбда-выражение, которое получает два параметра `Int` и возвращает `Int`:

```
val add: (Int, Int) -> Int
add = { x: Int, y: Int -> x + y }
```



Аналогично следующий код определяет переменную с именем `greeting` для хранения ссылки на лямбда-выражение, которое не получает параметров и возвращает `String`:

```
val greeting: () -> String
greeting = { "Hello!" }
```



Как и при объявлении переменных других типов, вы можете явно объявить тип переменной и присвоить ей значение в одной строке кода. Это означает, что приведенный выше код можно переписать в следующем виде:

```
val greeting: () -> String = { "Hello!" }
```

Объявляем переменную. *Определяем ее тип.* *Присваиваем ей значение.*

Компилятор может автоматически определять типы параметров лямбда-выражений

Когда вы явно объявляете тип переменной, в лямбда-выражении можно опустить любые объявления типов, которые компилятор сможет определить самостоятельно.

Допустим, имеется следующий код, который присваивает лямбда-выражение переменной с именем `addFive`:

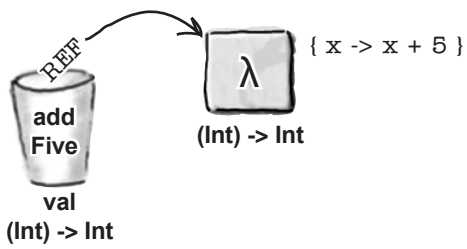
Лямбда-выражение прибавляет 5 к переменной Int с именем x.

```
val addFive: (Int) -> Int = { x: Int -> x + 5 }
```

Из определения типа `addFive` компилятору уже известно, что любое лямбда-выражение, присваиваемое этой переменной, должно иметь параметр `Int`. А это означает, что объявление типа `Int` из определения параметра лямбда-выражения можно опустить, потому что компилятор сможет автоматически вычислить его тип:

```
val addFive: (Int) -> Int = { x -> x + 5 }
```

Компилятор знает, что значение x должно иметь тип Int, поэтому указывать тип не обязательно.



Единственный параметр может заменяться на it

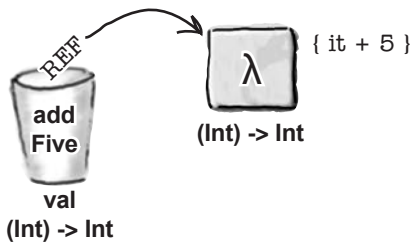
Если вы используете лямбда-выражение с одним параметром и компилятор может определить его тип, можно опустить параметр и ссылаться на него в теле лямбда-выражения по ключевому слову `it`.

Предположим, как и в предыдущем случае, у вас имеется лямбда-выражение, которое присваивается переменной:

```
val addFive: (Int) -> Int = { x -> x + 5 }
```

Так как лямбда-выражение имеет единственный параметр `x`, и компилятор может автоматически определить, что `x` имеет тип `Int`, можно убрать параметр `x` из тела лямбда-выражения и заменить его ключевым словом `it`:

```
val addFive: (Int) -> Int = { it + 5 }
```



В приведенном примере код `{ it + 5 }` эквивалентен `{ x -> x + 5 }`, но получается намного более компактным.

Обратите внимание: синтаксис `it` может использоваться только в том случае, когда компилятор может определить тип параметра. Например, следующий код компилироваться не будет, потому что компилятор не знает, к какому типу относится параметр:

```
val addFive = { it + 5 }
```

Не компилируется, потому что компилятор не может определить тип параметра.

Используйте лямбда-выражение, соответствующее типу переменной

Как вы уже знаете, компилятор внимательно следит за типом переменной. Это относится не только к типам обычных объектов, но и к типам лямбда-выражений. Следовательно, компилятор позволит присвоить переменной только такое лямбда-выражение, которое совместимо с типом этой переменной.

Допустим, имеется переменная с именем `calculation`, предназначенная для хранения ссылок на лямбда-выражения с двумя параметрами `Int` и возвращаемым значением `Int`:

```
val calculation: (Int, Int) -> Int
```

Если вы попытаетесь назначить лямбду для `calculation`, тип которой не соответствует типу переменной, компилятор расстроится.

Следующий код не будет компилироваться, так как лямбда-выражение явно использует `Double`:

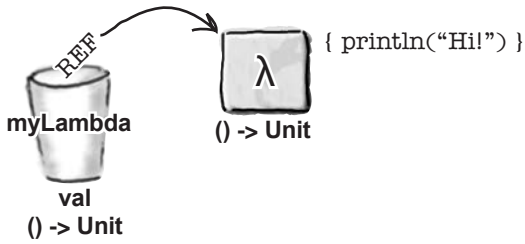
```
calculation = { x: Double, y: Double -> x + y }
```

Не компилируется, потому что переменная calculation допускает только лямбда-выражения с двумя параметрами Int и возвращаемым типом Int.

Unit и лямбда-выражения без возвращаемого значения

Если вы хотите указать, что лямбда-выражение не имеет возвращаемого значения, объявите его с возвращаемым типом `Unit`. Например, следующее лямбда-выражение не имеет возвращаемого значения и при вызове выводит текст «Hi!»:

```
val myLambda: () -> Unit = { println("Hi!") }
```



Запись **`Unit`** также позволяет явно указать, что вы не собираетесь обращаться к результату вычисления лямбда-выражения. Например, следующий код компилируется, но вы не сможете получить доступ к результату `x + y`:

```
val calculation: (Int, Int) -> Unit = { x, y -> x + y }
```

Часто Задаваемые Вопросы

В: Присвоит ли команда `val x = { "Pow!" }` текст «Pow!» переменной `x`?

О: Нет. Эта команда присваивает `x` лямбда-выражение, а не `String`. Тем не менее при выполнении это лямбда-выражение вернет строку «Pow!»

В: Можно ли присвоить лямбда-выражение переменной типа `Any`?

О: Да. Любой переменной `Any` можно присвоить ссылку на объект любого типа, включая лямбда-выражение.

В: Синтаксис выглядит знакомо. Я уже его где-то видел?

О: Да! В главе 8 он использовался с `let`. Тогда мы вам об этом не сказали, потому что хотели сосредоточиться на значениях null, но на самом деле `let` является функцией, которая получает лямбда-выражение в параметре.

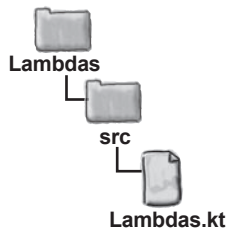
Создание проекта Lambdas

Теперь, когда вы научились создавать лямбда-выражения, добавим их в новое приложение.

Создайте новый проект Kotlin для JVM с именем «Lambdas». Создайте новый файл Kotlin с именем *Lambdas.kt*: выделите папку *src*, откройте меню File и выберите команду New → Kotlin File/Class. Введите имя файла «Lambdas» и выберите вариант File в группе Kind.

Затем обновите свою версию *Lambdas.kt* и приведите ее в соответствие с нашей:

```
fun main(args: Array<String>) {
    var addFive = { x: Int -> x + 5 }
    println("Pass 6 to addFive: ${addFive(6)}")

    val addInts = { x: Int, y: Int -> x + y }
    val result = addInts.invoke(6, 7)
    println("Pass 6, 7 to addInts: $result")

    val intLambda: (Int, Int) -> Int = { x, y -> x * y }
    println("Pass 10, 11 to intLambda: ${intLambda(10, 11)}")

    val addSeven: (Int) -> Int = { it + 7 }
    println("Pass 12 to addSeven: ${addSeven(12)}")

    val myLambda: () -> Unit = { println("Hi!") }
    myLambda()
}
```



Тест-драйв

При выполнении этого кода в окне вывода IDE отображается следующий текст:

```
Pass 6 to addFive: 11
Pass 6, 7 to addInts: 13
Pass 10, 11 to intLambda: 110
Pass 12 to addSeven: 19
Hi!
```

Путаница с сообщениями

Ниже приведена короткая программа Kotlin. Один блок в программе пропущен. Ваша задача — сопоставить блоки-кандидаты (слева) с выводом, который вы увидите при подстановке этого блока. Используются не все строки вывода, а некоторые могут использоваться более одного раза. Проведите линию от каждого блока к подходящему варианту вывода.

```
fun main(args: Array<String>) {
    val x = 20
    val y = 2.3
    
}
```

Код блока подставляется сюда.

Соедините каждый блок с одним из вариантов вывода.

Блоки:

```
val lam1 = { x: Int -> x }
println(lam1(x + 6))
```

```
val lam2: (Double) -> Double
lam2 = { (it * 2) + 5}
println(lam2(y))
```

```
val lam3: (Double, Double) -> Unit
lam3 = { x, y -> println(x + y) }
lam3.invoke(y, y)
```

```
var lam4 = { y: Int -> (y/2).toDouble() }
print(lam4(x))
lam4 = { it + 6.3 }
print(lam4(7))
```

Варианты вывода:

```
22.3
```

```
26
```

```
9.6
```

```
8.3
```

```
1.1513.3
```

```
9.3
```

```
10.013.3
```

```
4.6
```

Путаница с сообщениями. Решение

Ниже приведена короткая программа Kotlin. Один блок в программе пропущен. Ваша задача — сопоставить блоки-кандидаты (слева) с выводом, который вы увидите при подстановке этого блока. Используются не все строки вывода, а некоторые могут использоваться более одного раза. Проведите линию от каждого блока к подходящему варианту вывода.

```
fun main(args: Array<String>) {
    val x = 20
    val y = 2.3

}
```

Код блока подставляется сюда.

Блоки:

```
val lam1 = { x: Int -> x }
println(lam1(x + 6))
```

```
val lam2: (Double) -> Double
lam2 = { (it * 2) + 5}
println(lam2(y))
```

```
val lam3: (Double, Double) -> Unit
lam3 = { x, y -> println(x + y) }
lam3.invoke(y, y)
```

```
var lam4 = { y: Int -> (y/2).toDouble() }
print(lam4(x))
lam4 = { it + 6.3 }
print(lam4(7))
```

Варианты вывода:

```
22.3
26
9.6
8.3
1.1513.3
9.3
10.013.3
4.6
```

Ниже перечислены определения переменных и лямбда-выражения. Какие лямбда-выражения могут быть связаны с той или иной переменной? Соедините переменные с подходящими лямбда-выражениями.

Определения переменных:

```
var lambda1: (Double) -> Int
```

```
var lambda2: (Int) -> Double
```

```
var lambda3: (Int) -> Int
```

```
var lambda4: (Double) -> Unit
```

```
var lambda5
```

Лямбда-выражения:

```
{ it + 7.1 }
```

```
{ (it * 3) - 4 }
```

```
{ x: Int -> x + 56 }
```

```
{ println("Hello!") }
```

```
{ x: Double -> x + 75 }
```

Я и мой тип — РЕШЕНИЕ

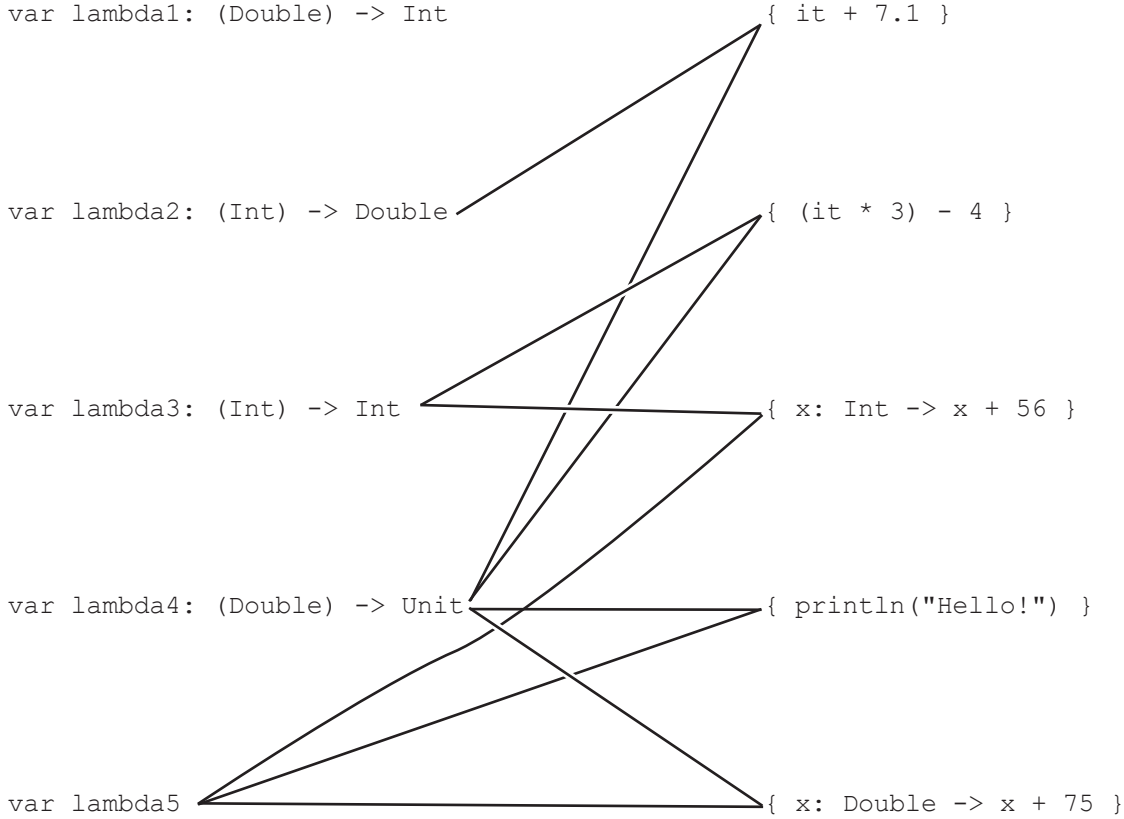
Ниже перечислены определения переменных и лямбда-выражения. Какие лямбда-выражения могут быть связаны с той или иной переменной? Соедините переменные с подходящими лямбда-выражениями.

Определения переменных:

```
var lambda1: (Double) -> Int
var lambda2: (Int) -> Double
var lambda3: (Int) -> Int
var lambda4: (Double) -> Unit
var lambda5
```

Лямбда-выражения:

```
{ it + 7.1 }
{ (it * 3) - 4 }
{ x: Int -> x + 56 }
{ println("Hello!") }
{ x: Double -> x + 75 }
```



Лямбда-выражение может передаваться функции

Функция, использующая лямбда-выражение для параметра или возвращаемого значения, называется функцией высшего порядка.

Кроме присваивания лямбда-выражения переменной, вы также можете использовать одно или несколько выражений как параметры функции. Это позволяет **передавать определенное поведение более общей функции**.

Чтобы понять, как это делается, мы напишем функцию `convert`, которая преобразует значение `Double` по формуле, передаваемой в лямбда-выражении, выводит результат и возвращает ее. Например, с помощью этой функции вы сможете преобразовать температуру по Цельсию в температуру по Фаренгейту или преобразовать вес из килограммов в фунты — все зависит от формулы, которая передается в лямбда-выражении (аргумента).

Начнем с определения параметров функции.

Добавление лямбда-параметра к функции

Чтобы функция могла преобразовать одно значение `Double` в другое, функции необходимо преобразуемое значение `Double` и лямбда-выражение, которое описывает процесс преобразования. Для этого в функцию `convert` будут добавлены два параметра: `Double` и лямбда-выражение.

Параметр лямбда-выражения определяется так же, как и любой другой параметр функции: вы указываете тип параметра и присваиваете ему имя. Назовем свое лямбда-выражение `converter`, а поскольку оно будет использоваться для преобразования `Double` в `Double`, оно должно иметь тип `(Double) -> Double` (лямбда-выражение, которое получает параметр `Double` и возвращает `Double`).

Определение функции (без тела функции) приведено ниже. Как видите, функция получает два параметра — `Double` с именем `x` и лямбда-выражение с именем `converter`, и возвращает `Double`:

Параметр x, тип Double.

Лямбда-параметр с именем converter. Имеет тип (Double) -> Double.

Функция возвращает Double.

```
fun convert(x: Double,
            converter: (Double) -> Double) : Double {
    //Код для преобразования в Int
}
```

Перейдем к написанию кода тела функции.

Выполнение лямбда-выражения в теле функции

Функция `convert` преобразует значение параметра `x` по формуле, переданной в параметре `converter` (лямбда-выражение). Мы выполним лямбда-выражение `converter` в теле функции и передадим значение `x`, после чего выведем результат.

Полный код функции `convert`:

```
fun convert(x: Double,
            converter: (Double) -> Double) : Double {
    val result = converter(x)
    println("$x is converted to $result")
    return result
}
```

Вызвать лямбда-выражение с именем converter и присвоить возвращенное значение result.

Вывести результат.

Вернуть результат.

Функция написана, попробуем вызвать ее.

Вызов функции с передачей параметров

Функция с лямбда-параметром вызывается точно так же, как и любая другая функция: с передачей значений всех аргументов. В данном случае `Double` и лямбда-выражений.

Давайте используем функцию `convert` для преобразования 20 градусов по Цельсию в градусы по Фаренгейту. Для этого функции нужно передать значения `20.0` и `{ c: Double -> c * 1.8 + 32 }`:

```
convert(20.0, { c: Double -> c * 1.8 + 32 })
```

Преобразуемое значение...

...и лямбда-выражение, которое будет использоваться для преобразования. Обратите внимание: мы можем использовать "it" вместо c, потому что лямбда-выражение имеет один параметр, тип которого компилятор может определить автоматически.

При выполнении этого кода будет возвращено значение 68.0 (результат преобразования 20 градусов по Цельсию в градусы по Фаренгейту).

Разберемся, что происходит при выполнении кода.

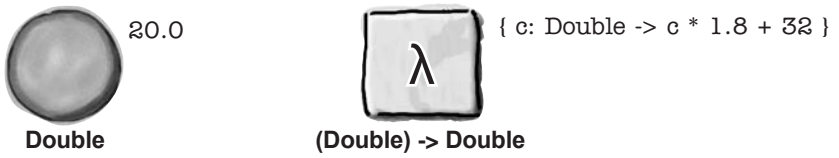
Что происходит при вызове функции

При вызове функции `convert` в виде, показанном ниже, происходит следующее:

```
val fahrenheit = convert(20.0, { c: Double -> c * 1.8 + 32 })
```

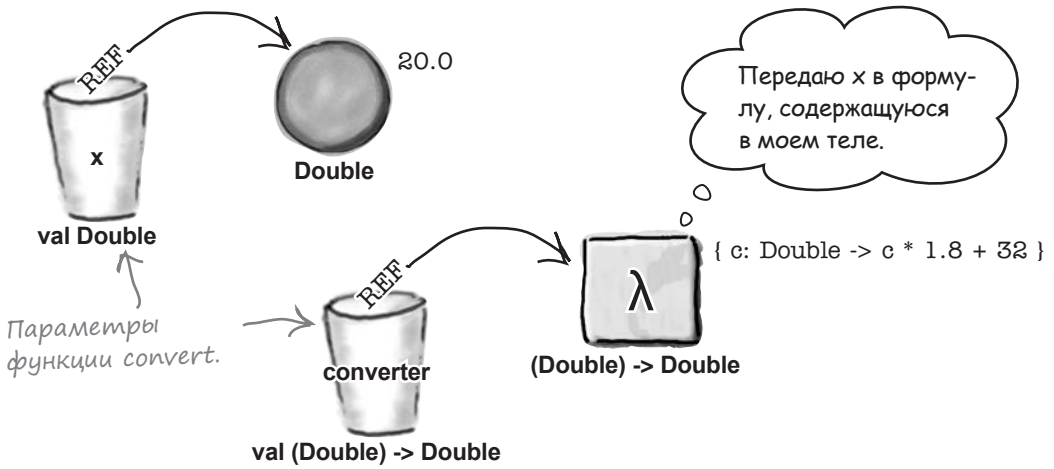
1 val fahrenheit = **convert(20.0, { c: Double –> c * 1.8 + 32 })**

Создает объект `Double` со значением `20.0` и лямбда-выражение со значением `{ c: Double -> c * 1.8 + 32 }`.



2
```
fun convert(x: Double, converter: (Double) –> Double) : Double {
    val result = converter(x)
    println("$x is converted to $result")
    return result
}
```

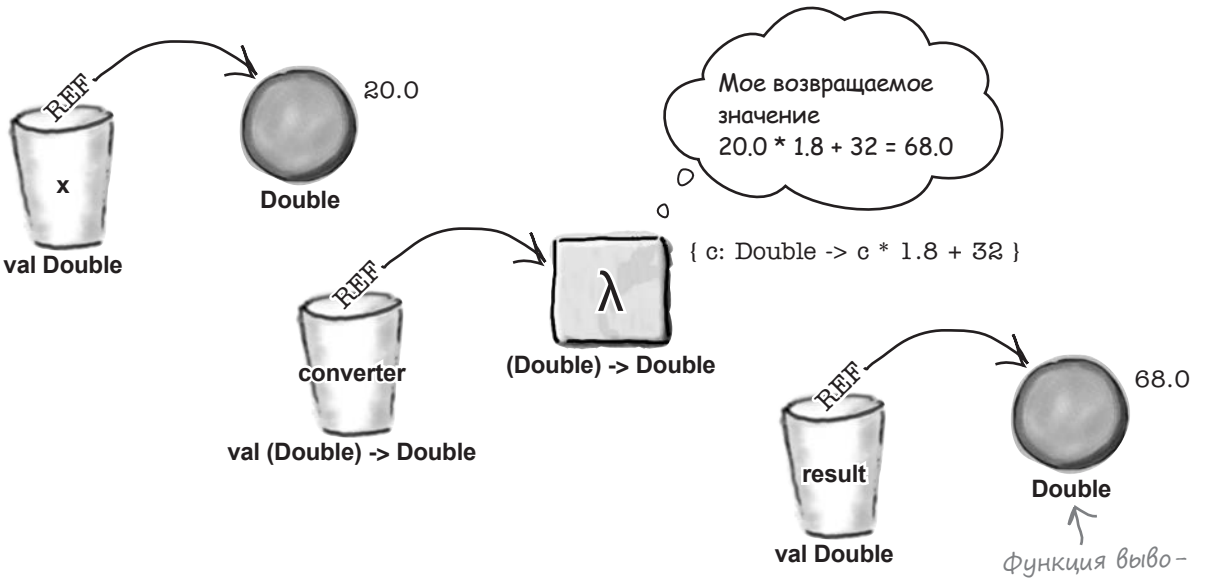
Код передает ссылку на созданные им объекты функции `convert`. Значение `Double` присваивается параметру `x` функции `convert`, а лямбда-выражение присваивается параметру `converter`. Затем выполняется лямбда-выражение `converter`, при этом `x` передается в параметре лямбда-выражения.



История продолжается...

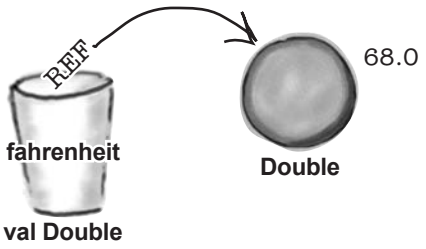
3 fun convert(x: Double, converter: (Double) -> Double) : Double {
val result = converter(x)
println("$x is converted to $result")
return result
}

Выполняется тело лямбда-выражения, а результат (`Double` со значением `68.0`) присваивается новой переменной с именем `result`. Функция выводит значения переменных `x` и `result` и возвращает ссылку на объект `result`.



4 **val fahrenheit =** convert(20.0, { c: Double -> c * 1.8 + 32 })

Создается новая переменная `fahrenheit`. Ей присваивается ссылка на объект, возвращенный функцией `convert`.



Итак, вы увидели, что происходит при вызове функции с лямбда-параметром. Рассмотрим некоторые приемы сокращенной записи, которые используются с функциями такого рода.

Лямбда-выражение можно вынести ЗА СКОБКИ...

До сих пор мы рассматривали вызов функций с лямбда-параметром и передачей аргументов функции в круглых скобках. Например, код вызова функции `convert` выглядел так:

```
convert(20.0, { c: Double -> c * 1.8 + 32 })
```

Если последний параметр вызываемой функции является лямбда-выражением, как в случае с функцией `convert`, лямбда-аргумент можно вынести *за* круглые скобки вызова функции. Например, следующий код делает то же самое, что и приведенный выше, но лямбда-выражение располагается вне круглых скобок:

```
convert(20.0) { c: Double -> c * 1.8 + 32 }
```

Закрывающая круглая скобка вызова функции.

Лямбда-выражение уже не ограничивается закрывающей круглой скобкой функции.

...или полностью удалить ()

Если функция имеет всего один параметр, и этот параметр представляет собой лямбда-выражение, круглые скобки при вызове функции можно полностью опустить.

Предположим, функция `convertFive` преобразует `Int` 5 в `Double` по формуле преобразования, которая передается в виде лямбда-выражения. Код функции выглядит так:

```
fun convertFive(converter: (Int) -> Double) : Double {
    val result = converter(5)
    println("5 is converted to $result")
    return result
}
```

Функция `convertFive` имеет всего один параметр — лямбда-выражение, и вызов функции может выглядеть так:

```
convertFive { it * 1.8 + 32 }
```

Обратите внимание: при вызове функции круглые скобки не используются. Это возможно благодаря тому, что единственный параметр функции представляет собой лямбда-выражение.

Этот вызов работает точно так же, как этот:

```
convertFive() { it * 1.8 + 32 }
```

но не содержит круглых скобок.

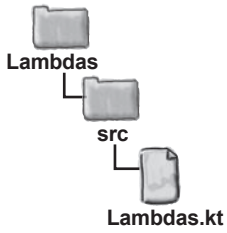
Итак, вы узнали, как пишутся функции с лямбда-параметрами, и теперь можно обновить код проекта.

Обновление проекта Lambdas

Добавим функции `convert` и `convertFive` в проект Lambdas. Обновите свою версию *Lambdas.kt*, чтобы она соответствовала нашей (изменения выделены жирным шрифтом):

```
fun convert(x: Double,
            converter: (Double) -> Double) : Double {
    val result = converter(x)
    println("$x is converted to $result")
    return result
}

fun convertFive(converter: (Int) -> Double) : Double {
    val result = converter(5)
    println("5 is converted to $result")
    return result
}

fun main(args: Array<String>) {
    var addFive = { x: Int -> x + 5 }
    println("Pass 6 to addFive: ${addFive(6)}")

    val addInts = { x: Int, y: Int -> x + y }
    val result = addInts.invoke(6, 7)
    println("Pass 6, 7 to addInts: $result")

    val intLambda: (Int, Int) -> Int = { x, y -> x * y }
    println("Pass 10, 11 to intLambda: ${intLambda(10, 11)}")

    val addSeven: (Int) -> Int = { it + 7 }
    println("Pass 12 to addSeven: ${addSeven(12)}")

    val myLambda: () -> Unit = { println("Hi!") }
    myLambda()

    convert(20.0) { it * 1.8 + 32 }
    convertFive { it * 1.8 + 32 }
}
```

Добавьте эти две функции.



Этот фрагмент не нужен, его можно удалить.

Добавьте эти строки. Здесь можно использовать «it», потому что каждое лямбда-выражение использует один параметр, тип которого может быть автоматически определен компилятором.

Попробуем применить этот код на практике.

Тест-драйв

При выполнении этого кода в окне вывода IDE отображается следующий текст:

```
20.0 is converted to 68.0
5 is converted to 41.0
```

Прежде чем разбираться в том, что можно сделать с лямбда-выражениями, рассмотрим следующее упражнение.

Формат лямбда-выражений под увеличительным стеклом

Как упоминалось ранее в этой главе, тело лямбда-выражения может состоять из нескольких строк кода. Например, следующее лямбда-выражение выводит значение своего параметра, а затем использует его в вычислениях:

```
{ c: Double -> println(c)
                c * 1.8 + 32 }
```

Если тело лямбда-выражения состоит из нескольких строк, последнее вычисленное выражение используется как возвращаемое значение лямбда-выражения. Так, в предыдущем примере возвращаемое значение определяется следующей строкой:

```
c * 1.8 + 32
```

Лямбда-выражение также можно отформатировать так, чтобы оно выглядело как блок кода, а фигурные скобки, в которые оно заключается, размещались отдельно от содержимого. В следующем коде этот прием используется для передачи лямбда-выражения `{ it * 1.8 + 32 }` функции `convertFive`:

```
convertFive {
    it * 1.8 + 32
}
```

Часто задаваемые вопросы

В: Похоже, есть немало полезных сокращений при работе с лямбда-выражениями. Мне действительно необходимо знать о них всех?

О: Об этих сокращениях полезно знать, потому что когда вы к ним привыкнете, они сделают ваш код более компактным и удобочитаемым. Альтернативный синтаксис, разработанный для удобства чтения вашего кода, иногда называют «синтаксическим сахаром». Но даже если вы не хотите использовать эти сокращения в своем коде, их все равно стоит знать, потому что вы можете столкнуться с ними в стороннем коде.

В: Почему лямбда-выражения так называются?

О: Потому что они происходят из области математики и информатики, называемой «лямбда-исчислением», где небольшие анонимные функции представляются греческой буквой λ (лямбда).

В: Почему лямбда-выражения не называются функциями?

О: Лямбда-выражение является разновидностью функций, но в большинстве языков у функций всегда есть имена. Как вы уже видели, у лямбда-выражения имени может не быть.

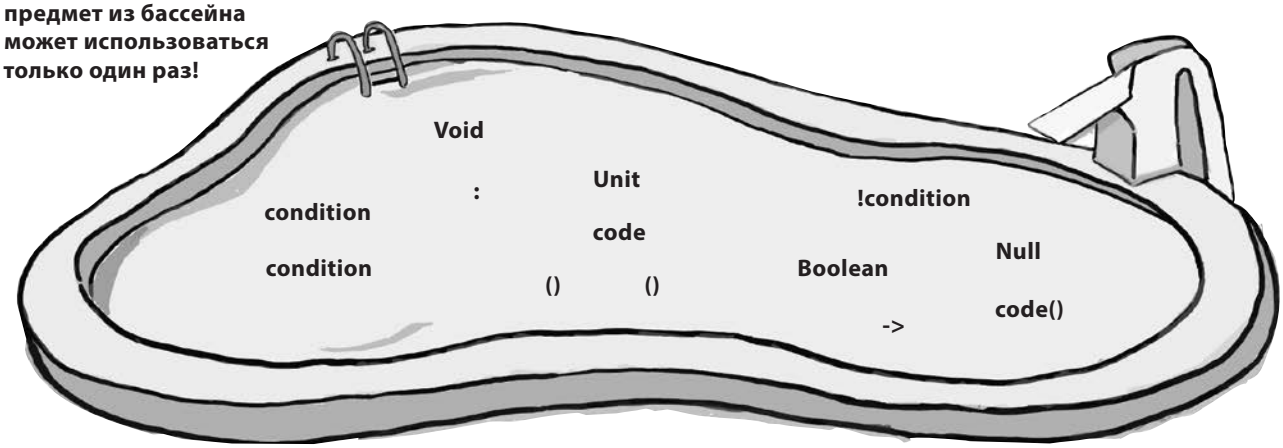
У бассейна

Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: создать функцию с именем `unless`, которая вызывается функцией `main`. Функция `unless` должна иметь два параметра, логический признак с именем `condition` и лямбда-выражение с именем `code`. Функция должна выполнять код лямбда-выражения в том случае, когда `condition` ложно.

```
fun unless(_____ , code:_____) {
    if (_____) {
        _____
    }
}

fun main(args: Array<String>) {
    val options = arrayOf("Red", "Amber", "Green")
    var crossWalk = options[(Math.random() * options.size).toInt()]
    if (crossWalk == "Green") {
        println("Walk!")
    }
    unless (crossWalk == "Green") {
        println("Stop!")
    }
}
```

Вывести «Stop!», если не выполняется условие crossWalk == «Green».

Примечание: каждый предмет из бассейна может использоваться только один раз!



Ответ на с. 390.

Функция может возвращать лямбда-выражение

Помимо использования лямбда-выражения в качестве параметра функция также может возвращать его, указывая тип лямбда-выражения в своем возвращаемом типе. Например, следующий код определяет функцию с именем `getConversionLambda`, которая возвращает лямбда-выражение типа `(Double) -> Double`. Точное лямбда-выражение, возвращаемое функцией, зависит от значения переданной строки.

Функция получает один параметр String.

Она возвращает лямбда-выражение с типом (Double) -> Double.

```
fun getConversionLambda(str: String): (Double) -> Double {
    if (str == "CentigradeToFahrenheit") {
        return { it * 1.8 + 32 }
    } else if (str == "KgsToPounds") {
        return { it * 2.204623 }
    } else if (str == "PoundsToUSTons") {
        return { it / 2000.0 }
    } else {
        return { it }
    }
}
```

(Double) -> Double

Функция возвращает одно из этих лямбда-выражений в зависимости от переданной строки.

Вы можете вызвать лямбда-выражение, возвращенное функцией, или использовать его в аргументе при вызове другой функции. Например, следующий код выполняет возвращаемое значение `getConversionLambda` для пересчета 2,5 кг в фунты и присваивает его переменной с именем pounds:

```
val pounds = getConversionLambda("KgsToPounds")(2.5)
```

Вызывает функцию getConversionLambda...

...и вызывает лямбда-выражение, возвращенное функцией.

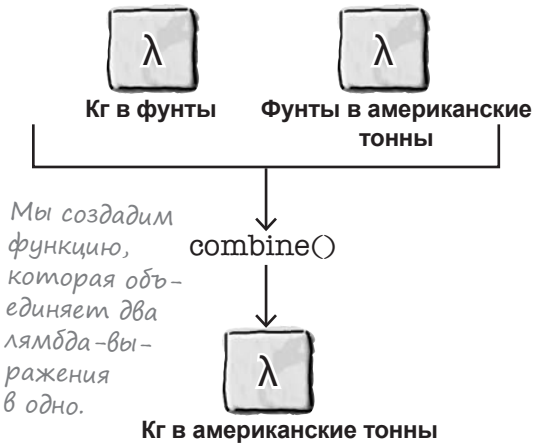
В следующем примере функция `getConversionLambda` используется для получения лямбда-выражения, преобразующего температуру из шкалы Цельсия в шкалу Фаренгейта, после чего передает ее функции `convert`:

Здесь возвращаемое значение getConversionLambda передается функции convert.

```
convert(20.0, getConversionLambda("CentigradeToFahrenheit"))
```

Также возможно определить функцию, которая получает и возвращает лямбда-выражение. Сейчас мы рассмотрим эту возможность.

Написание функции, которая получает и возвращает лямбда-выражения



Мы создадим функцию с именем `combine`, которая получает два лямбда-параметра, объединяет их и возвращает результат (другое лямбда-выражение). Если функция получает лямбда-выражения для преобразования значений из килограммов в фунты и из фунтов в американские тонны, то она вернет лямбда-выражение для преобразования значения из килограммов в американские тонны. После этого вы сможете использовать новое лямбда-выражение в своем коде. Начнем с определения параметров функции и возвращаемого типа.

Определение параметров и возвращаемого типа

Все лямбда-выражения, используемые функцией `combine`, преобразуют одно значение `Double` в другое значение `Double`, так что каждое имеет тип `(Double) -> Double`. Следовательно, наше определение функции должно выглядеть так:

```
fun combine(lambda1: (Double) -> Double,
            lambda2: (Double) -> Double): (Double) -> Double {
    //Код объединения двух лямбда-выражений
}
```

Функция combine имеет два лямбда-параметра с типом (Double) -> Double.

Функция тоже возвращает лямбда-выражение этого типа.

А теперь перейдем к телу функции.

Определение тела функции

Тело функции должно возвращать лямбда-выражение, обладающее следующими характеристиками:

- Функция должна получать один параметр `Double`. Мы присвоим этому параметру имя `x`.
- Тело лямбда-функции должно вызывать лямбда-выражение `lambda1`, передавая ему значение `x`. Результат этого вызова передается `lambda2`.

Для этого может использоваться следующий код:

```
fun combine(lambda1: (Double) -> Double,
            lambda2: (Double) -> Double): (Double) -> Double {
    return { x: Double -> lambda2(lambda1(x)) }
}
```

Лямбда-выражение, возвращаемое combine, получает параметр Double с именем x.

x передается выражению lambda1, которое получает и возвращает Double. Затем результат передается лямбда-выражению lambda2, которое также получает и возвращает Double.

Напишем код, в котором используется эта функция.

Как использовать функцию combine

Функция `combine`, которую мы только что написали, получает два лямбда-выражения и объединяет их в третье. Это означает, что если передать функции одно лямбда-выражение для преобразования значения из килограммов в фунты, а другое для преобразования значения из фунтов в американские тонны, функция вернет лямбда-выражение для преобразования из килограммов в американские тонны.

Код выглядит так:

```
//Определить два лямбда-выражения для преобразований
val kgsToPounds = { x: Double -> x * 2.204623 }
val poundsToUSTons = { x: Double -> x / 2000.0 }

//Объединить два выражения
val kgsToUSTons = combine(kgsToPounds, poundsToUSTons)

//Выполнить лямбда-выражение kgsToUSTons
val usTons = kgsToUSTons(1000.0)     //1.1023115
```

Эти лямбда-выражения преобразуют Double из килограммов в фунты, а затем из фунтов — в американские тонны.

Передать лямбда-выражения функции combine. При этом формируется лямбда-выражение для преобразования Double из килограммов в американские тонны.

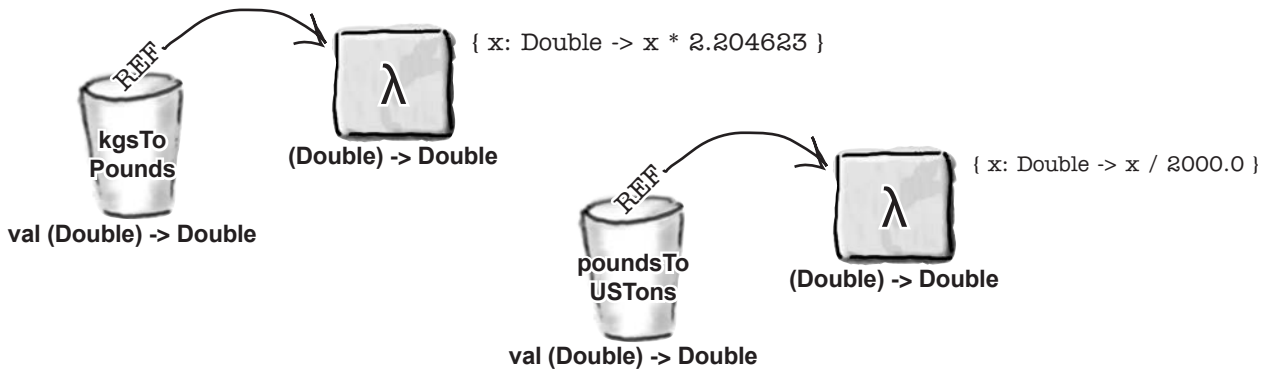
Выполнить полученное лямбда-выражение для значения 1000.0. Выражение возвращает 1.1023115.

А теперь заглянем за кулисы и посмотрим, что произойдет при выполнении этого кода.

Что происходит при выполнении кода

1. **val kgsToPounds = { x: Double –> x * 2.204623 }**
   val poundsToUSTons = { x: Double –> x / 2000.0 }
   val kgsToUSTons = **combine(kgsToPounds, poundsToUSTons)**

   Этот фрагмент создает две переменные и присваивает лямбда-выражение каждой из них. Затем ссылка на каждое лямбда-выражение передается функции `combine`.

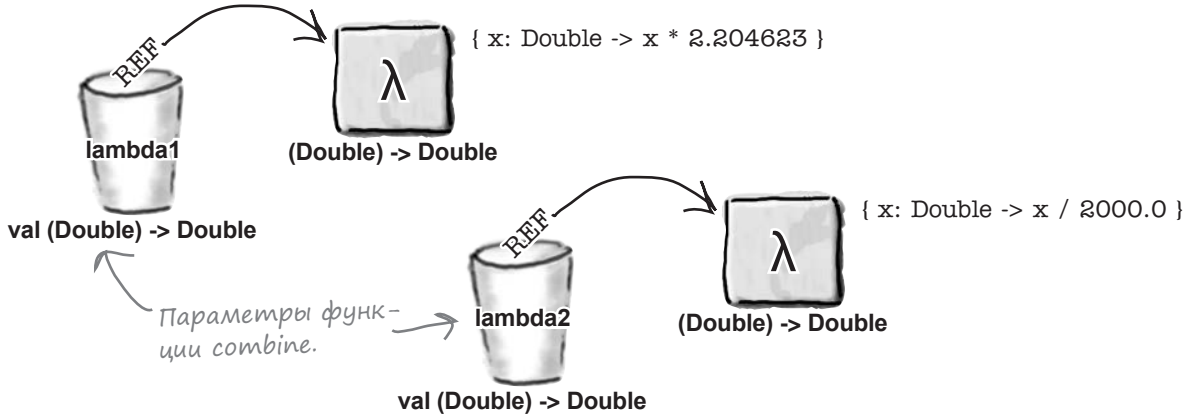


История продолжается...

2

```
fun combine(lambda1: (Double) –> Double,
            lambda2: (Double) –> Double): (Double) -> Double {
    return { x: Double -> lambda2(lambda1(x)) }
}
```

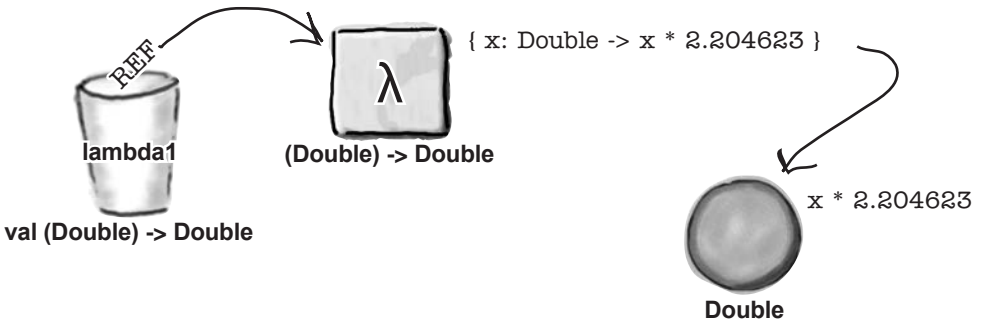
Лямбда-выражение `kgsToPounds` сохраняется в параметре `lambda1` функции `combine`, а лямбда-выражение `poundsToUSTons` — в параметре `lambda2`.



3

```
fun combine(lambda1: (Double) -> Double,
            lambda2: (Double) -> Double): (Double) -> Double {
    return { x: Double -> lambda2(lambda1(x)) }
}
```

Выполняется `lambda1(x)`. Так как тело `lambda1` содержит выражение `x * 2.204623`, где `x` относится к типу `Double`, создается объект `Double` со значением `x * 2.204623`.

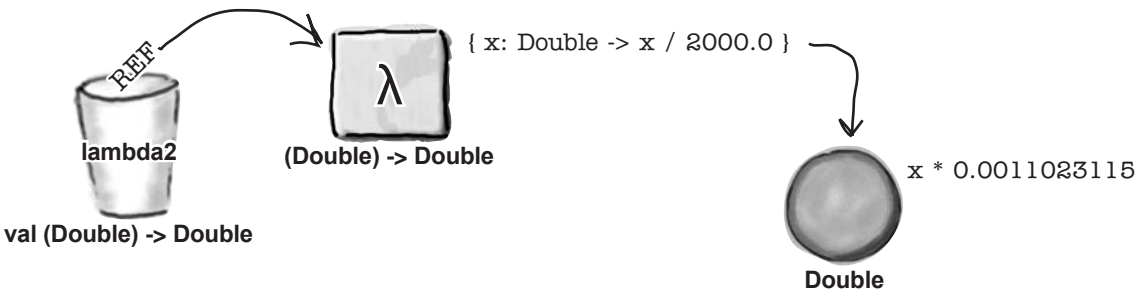


История продолжается...

4

```
fun combine(lambda1: (Double) -> Double,
            lambda2: (Double) -> Double): (Double) -> Double {
    return { x: Double -> lambda2(lambda1(x)) }
}
```

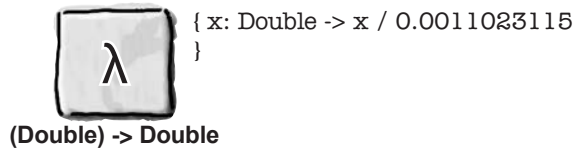
Затем объект `Double` со значением `x * 2.204623` передается `lambda2`. Так как тело `lambda2` содержит выражение `x / 2000.0`, это означает, что на место `x` подставляется `x * 2.204623`. В результате будет создан объект `Double` со значением `(x * 2.204623) / 2000.0`, или `x * 0.0011023115`.



5

```
fun combine(lambda1: (Double) -> Double,
            lambda2: (Double) -> Double): (Double) -> Double {
    return { x: Double -> lambda2(lambda1(x)) }
}
```

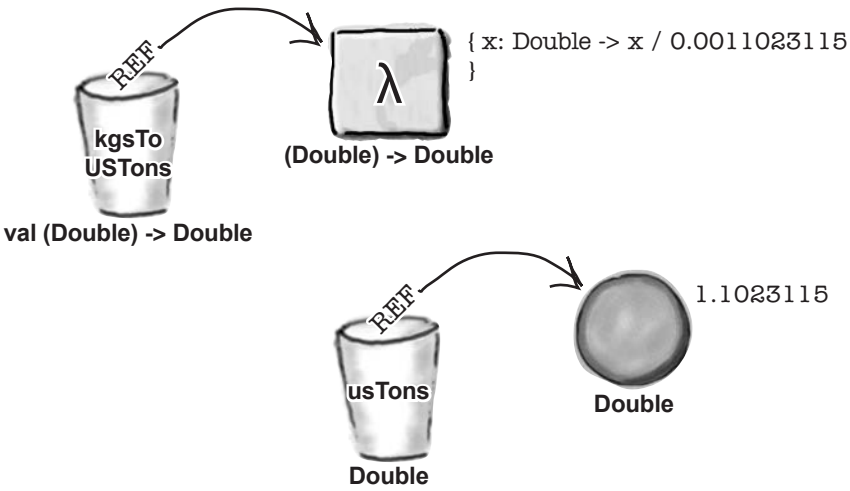
В результате создается лямбда-выражение `{ x: Double -> x * 0.0011023115 }`, и ссылка на него возвращается функцией.



История продолжается...

6 **val kgsToUSTons = combine(kgsToPounds, poundsToUSTons)**
val usTons = kgsToUSTons(1000.0)

Лямбда-выражение, возвращенное функцией `combine`, присваивается переменной с именем `kgsToUSTons`. Оно выполняется для аргумента 1000.0 и возвращает значение 1.1023115. Оно присваивается новой переменной с именем `usTons`.



Как сделать код с лямбда-выражениями более понятным

Глава понемногу подходит к концу, но прежде чем двигаться дальше, мы хотим рассмотреть еще одну тему: как сделать код с лямбда-выражениями более понятным.

При использовании функциональных типов (то есть типов, которые используются в определениях лямбда-выражений) ваш код становится более громоздким и непонятным. Например, в функции `combine` несколько раз встречается функциональный тип `(Double) -> Double`:

Функция combine использует три экземпляра функционального типа (Double) -> Double.

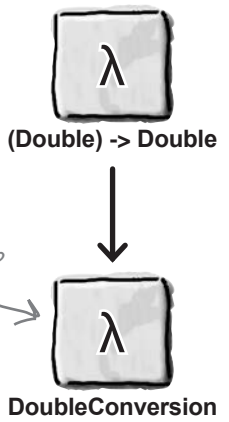
```
fun combine(lambda1: (Double) -> Double,
            lambda2: (Double) -> Double): (Double) -> Double {
    return { x: Double -> lambda2(lambda1(x)) }
}
```

Но ваш код станет более удобочитаемым, если вы замените функциональный тип **псевдонимом типа**. Посмотрим, что такое псевдонимы и как ими пользоваться.

typealias и назначение альтернативного имени для существующего типа

Псевдоним типа позволяет определить для существующего типа альтернативное имя, которое может использоваться в коде. Это означает, что если в вашем коде используется функциональный тип, например такой, как `(Double) -> Double`, — вы сможете определить псевдоним, который будет использоваться вместо этого типа, чтобы ваш код лучше читался.

Псевдонимы типов определяются ключевым словом **`typealias`**. В следующем примере это ключевое слово используется для определения псевдонима `DoubleConversion`, который будет использоваться вместо функционального типа `(Double) -> Double`:




```
typealias DoubleConversion = (Double) -> Double
```

Этот псевдоним означает, что мы можем использовать DoubleConversion вместо (Double) -> Double.

Функции `convert` и `combine` принимают следующий вид:

```
fun convert(x: Double,
            converter: DoubleConversion) : Double {
    val result = converter(x)
    println("$x is converted to $result")
    return result
}

fun combine(lambda1: DoubleConversion,
            lambda2: DoubleConversion): DoubleConversion {
    return { x: Double -> lambda2(lambda1(x)) }
}
```

Псевдоним типа DoubleConversion можно использовать в функциях convert и combine, чтобы код было проще читать.

Каждый раз, когда компилятор встречает тип `DoubleConversion`, он знает, что это всего лишь обозначение для типа `(Double) -> Double`. Функции `convert` и `combine` работают точно так же, как прежде, но код стал более понятным.

При помощи ключевого слова `typealias` можно определять альтернативные имена для любых типов, не только функциональных. Например, следующее определение

```
typealias DuckArray = Array<Duck>
```

позволяет обращаться к типу по имени `DuckArray` вместо `Array<Duck>`.

Пора обновить код нашего проекта.

Обновление проекта Lambdas

Мы добавим в проект Lambdas псевдоним `DoubleConversion`, функции `getConversionLambda` и `combine`, а также код, в котором они используются. Обновите свою версию *Lambdas.kt* в проекте, чтобы она соответствовала нашей (изменения выделены жирным шрифтом):

Добавляем typealias.

```
typealias DoubleConversion = (Double) -> Double
```

Функциональный тип заменяется псевдонимом.

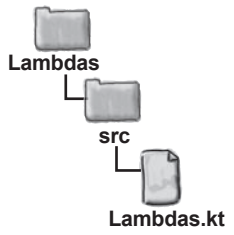
```
fun convert(x: Double,
            converter: (Double) -> Double DoubleConversion) : Double {
    val result = converter(x)
    println("$x is converted to $result")
    return result
}
```

Эту функцию можно удалить, она больше не нужна.

```
fun convertFive(converter: (Int) -> Double) : Double {
    val result = converter(5)
    println("5 is converted to $result")
    return result
}
```

Добавляем функцию getConversionLambda.

```
fun getConversionLambda(str: String): DoubleConversion {
    if (str == "CentigradeToFahrenheit") {
        return { it * 1.8 + 32 }
    } else if (str == "KgsToPounds") {
        return { it * 2.204623 }
    } else if (str == "PoundsToUSTons") {
        return { it / 2000.0 }
    } else {
        return { it }
    }
}
```

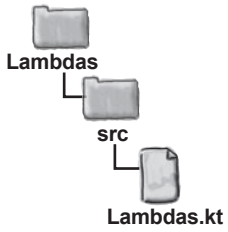


Добавляем функцию combine.

```
fun combine(lambda1: DoubleConversion,
            lambda2: DoubleConversion): DoubleConversion {
    return { x: Double -> lambda2(lambda1(x)) }
}
```

Продолжение на следующей странице.

Продолжение...




```
fun main(args: Array<String>) {
    convert(20.0) { it * 1.8 + 32 }
    convertFive { it * 1.8 + 32 }

    //Преобразовать 2.5 кг в фунты
    println("Convert 2.5kg to Pounds: ${getConversionLambda("KgsToPounds")(2.5)}")

    //Определить два лямбда-выражения для преобразований
    val kgsToPoundsLambda = getConversionLambda("KgsToPounds")
    val poundsToUSTonsLambda = getConversionLambda("PoundsToUSTons")

    //Два лямбда-выражения преобразуются в одно новое
    val kgsToUSTonsLambda = combine(kgsToPoundsLambda, poundsToUSTonsLambda)

    //Использовать новое лямбда-выражение для преобразования
    // 17,4 кг в американские тонны
    val value = 17.4
    println("$value kgs is ${convert(value, kgsToUSTonsLambda)} US tons")
}
```

Удалите эти строки.

getConversionLambda используется для получения двух лямбда-выражений.

Создаем лямбда-выражение для преобразования Double из килограммов в американские тонны.

Используем лямбда-выражение для преобразования 17,4 кг в американские тонны.

А теперь запустим код!

Тест-драйв

При выполнении этого кода в окне вывода отображается следующий текст:

```
Convert 2.5kg to Pounds: 5.5115575
17.4 is converted to 0.0191802201
17.4 kgs is 0.0191802201 US tons
```

Итак, теперь вы знаете, как использовать лямбда-выражения для создания функций высшего порядка. Проверьте свои силы на следующих упражнениях, а в следующей главе мы представим некоторые встроенные функции высшего порядка Kotlin и покажем, какими гибкими и мощными они бывают.

Часто Задаваемые Вопросы

В: Я слышал о функциональном программировании. Что это такое?

О: Лямбда-выражения являются важной частью функционального программирования. Если не-функциональная программа читает *входные данные* и генерирует *выходные данные*, то функциональные программы могут читать *функции* как входные данные и генерировать *функции* на выходе. Если ваш код включает функции высшего порядка, значит, вы занимаетесь функциональным программированием.

В: Сильно ли функциональное программирование отличается от объектно-ориентированного?

О: Это два способа организации кода. В объектно-ориентированном программировании данные объединяются с функциями, а в функциональном — функции объединяются с функциями. Эти два стиля программирования не являются противоположностями; это всего лишь две разные точки зрения на мир.

Развлечения с магнитами

На холодильнике была выложена функция для вывода имен элементов списка `List<Grocery>`, удовлетворяющих некоторому критерию. К сожалению, некоторые магниты упали на пол. Удастся ли вам восстановить функцию?

Здесь размещается функция.

```
data class Grocery(val name: String, val category: String,
                   val unit: String, val unitPrice: Double)
```

Класс данных Grocery.

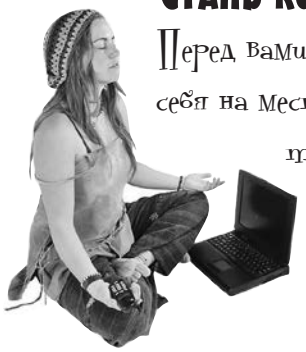
```
fun main(args: Array<String>) {
    val groceries = listOf(Grocery("Tomatoes", "Vegetable", "lb", 3.0),
            Grocery("Mushrooms", "Vegetable", "lb", 4.0),
            Grocery("Bagels", "Bakery", "Pack", 1.5),
            Grocery("Olive oil", "Pantry", "Bottle", 6.0),
            Grocery("Ice cream", "Frozen", "Pack", 3.0))
    println("Expensive ingredients:")
    search(groceries) {i: Grocery -> i.unitPrice > 5.0}
    println("All vegetables:")
    search(groceries) {i: Grocery -> i.category == "Vegetable"}
    println("All packs:")
    search(groceries) {i: Grocery -> i.unit == "Pack"}
}
```

Функция main использует функцию search.

println(l.name) | l in list | list: | , | for (| (g: Grocery) -> Boolean |) | criteria(l) | } | search | fun | } |) | { | { | (| List<Grocery> |) | if (| criteria: | } | {

Ответы на с. 388.

СТАНЬ компилятором



Перед вами пять функций. Представьте себя на месте компилятора и определите, будет ли компилироваться каждая из них. Если функция не компилируется, то почему?

A

```
fun myFun1(x: Int = 6, y: (Int) -> Int = 7): Int {
    return y(x)
}
```

B

```
fun myFun2(x: Int = 6, y: (Int) -> Int = { it }) {
    return y(x)
}
```

C

```
fun myFun3(x: Int = 6, y: (Int) -> Int = { x: Int -> x + 6 }): Int {
    return y(x)
}
```

D

```
fun myFun4(x: Int, y: Int,
           z: (Int, Int) -> Int = {
               x: Int, y: Int -> x + y
           }) {
    z(x, y)
}
```

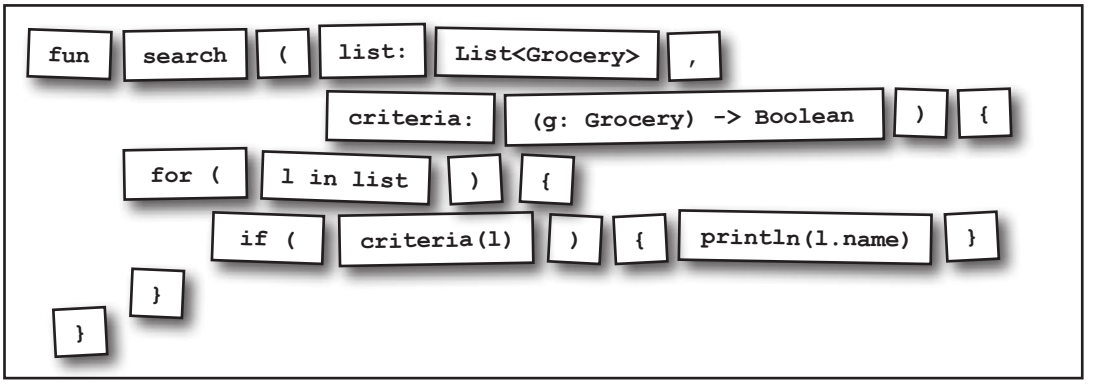
E

```
fun myFun5(x: (Int) -> Int = {
    println(it)
    it + 7
}) {
    x(4)
}
```

Ответы на с. 389.

Развлечения с магнитами. Решение

На холодильнике была выложена функция для вывода имен элементов списка `List<Grocery>`, удовлетворяющих некоторому критерию. К сожалению, некоторые магниты упали на пол. Удастся ли вам восстановить функцию?




```
data class Grocery(val name: String, val category: String,
                   val unit: String, val unitPrice: Double)

fun main(args: Array<String>) {
    val groceries = listOf(Grocery("Tomatoes", "Vegetable", "lb", 3.0),
            Grocery("Mushrooms", "Vegetable", "lb", 4.0),
            Grocery("Bagels", "Bakery", "Pack", 1.5),
            Grocery("Olive oil", "Pantry", "Bottle", 6.0),
            Grocery("Ice cream", "Frozen", "Pack", 3.0))
    println("Expensive ingredients:")
    search(groceries) {i: Grocery -> i.unitPrice > 5.0}
    println("All vegetables:")
    search(groceries) {i: Grocery -> i.category == "Vegetable"}
    println("All packs:")
    search(groceries) {i: Grocery -> i.unit == "Pack"}
}
```

СТАНЬ компилятором. Решение

Перед вами пять функций. Представьте себя на месте компилятора и определите, будет ли компилироваться каждая из них. Если функция не компилируется, то почему?

A

```
fun myFun1(x: Int = 6, y: (Int) -> Int = 7): Int {
    return y(x)
}
```

Не компилируется, так как лямбда-выражению присваивается значение по умолчанию 7 типа Int.

B

```
fun myFun2(x: Int = 6, y: (Int) -> Int = { it }) {
    return y(x)
}
```

Эта строка возвращает Int.

Не компилируется, потому что функция возвращает значение Int, которое не было объявлено.

C

```
fun myFun3(x: Int = 6, y: (Int) -> Int = { x: Int -> x + 6 }): Int {
    return y(x)
}
```

Компилируется. Параметрам назначены значения по умолчанию правильных типов, а ее возвращаемый тип правильно объявлен.

D

```
fun myFun4(x: Int, y: Int,
           z: (Int, Int) -> Int = {
               x: Int, y: Int -> x + y
           }) {
    z(x, y)
}
```

Компилируется. Переменной z в качестве значения по умолчанию присваивается допустимое лямбда-выражение.

E

```
fun myFun5(x: (Int) -> Int = {
    println(it)
    it + 7
}) {
    x(4)
}
```

Компилируется. Переменной x в качестве значения по умолчанию присваивается действительное лямбда-выражение, которое занимает несколько строк.

У бассейна. Решение

Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: создать функцию с именем `unless`, которая вызывается функцией `main`. Функция `unless` должна иметь два параметра, логический признак с именем `condition` и лямбда-выражение с именем `code`. Функция должна выполнять код лямбда-выражения в том случае, когда условие `condition` ложно.

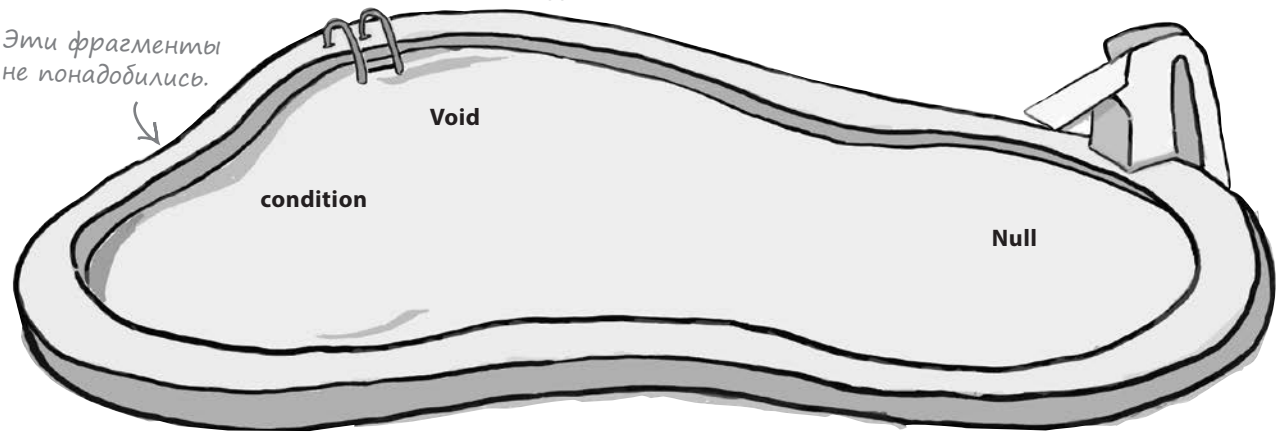
```
fun unless( condition: Boolean , code: () -> Unit ) {
    if ( !condition ) {
        code()
    }
}
```

Если condition ложно, выполнить код лямбда-выражения.

```
fun main(args: Array<String>) {
    val options = arrayOf("Red", "Amber", "Green")
    var crossWalk = options[(Math.random() * options.size).toInt()]
    if (crossWalk == "Green") {
        println("Walk!")
    }
    unless (crossWalk == "Green") {
        println("Stop!")
    }
}
```

Фрагмент отформатирован как программный блок, но в действительности это лямбда-выражение. Лямбда-выражение передается функции unless и выполняется, если значение crossWalk не равно «Green».

Эти фрагменты не понадобились.



Ваш инструментарий Kotlin

Глава 11 осталась позади, а ваш инструментарий пополнился лямбда-выражениями и функциями высшего порядка.

Весь код для этой главы можно загрузить по адресу https://tinyurl.com/HFKotlin.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Лямбда-выражение имеет вид:

```
{ x: Int -> x + 5 }
```

- Лямбда-выражения заключаются в фигурные скобки, могут включать параметры и содержат тело.
- Лямбда-выражение может состоять из нескольких строк. Последнее вычисленное выражение в теле используется как возвращаемое значение лямбда-выражения.
- Лямбда-выражение можно присвоить переменной. Тип переменной должен быть совместим с типом лямбда-выражения.
- Тип лямбда-выражения имеет формат:

```
(параметры) -> возвращаемое_значение
```

- Там, где это возможно, компилятор старается автоматически определить тип параметров лямбда-выражений.
- Если лямбда-выражение получает один параметр, его можно заменить обозначением `it`.
- Чтобы выполнить лямбда-выражение, вы либо передаете ему параметры в круглых скобках, либо вызываете его функцию `invoke`.
- Лямбда-выражение можно передать в параметре функции или использовать его в качестве возвращаемого выражения функции. Функция, использующая лямбда-выражения подобным образом, называется функцией высшего порядка.
- Если последний параметр функции является лямбда-выражением, то при вызове функции лямбда-выражение можно вынести за круглые скобки.
- Если функция получает один параметр, который является лямбда-выражением, то при вызове функции скобки можно опустить.
- Псевдоним типа позволяет задать альтернативное имя для существующего типа. Псевдоним типа определяется ключевым словом `typealias`.

Расширенные возможности

Kotlin содержит подборку встроенных функций высшего порядка. В этой главе представлены некоторые полезные функции этой категории. Вы познакомитесь с гибкими ***фильтрами*** и узнаете, как они используются для сокращения размера коллекции. Научитесь ***преобразовывать коллекции функцией map***, ***перебирать их элементы в forEach***, а также ***группировать элементы коллекций функцией groupBy***. Мы покажем, как использовать ***fold*** для выполнения сложных вычислений *всего в одной строке кода*. К концу этой главы вы научитесь писать **мощный** код, о котором и не мечтали.

Kotlin содержит подборку встроенных функций высшего порядка

Как упоминалось в начале главы 10, в Kotlin включена подборка встроенных функций высшего порядка, получающих лямбда-параметры. Многие из этих функций предназначены для работы с коллекциями. Например, с их помощью можно фильтровать коллекции по заданным критериям, или группировать элементы коллекции по значениям некоторого свойства.

Каждая функция высшего порядка имеет обобщенную реализацию, а ее специализированное поведение определяется переданным ей лямбда-выражением. Таким образом, чтобы отфильтровать коллекцию с помощью встроенной функции-фильтра, критерий фильтрации определяется в передаваемом функции лямбда-выражении.

Так как многие функции высшего порядка Kotlin предназначены для работы с коллекциями, мы познакомим вас с самыми полезными функциями высшего порядка из пакета *collections*. Для исследования этих функций мы определим класс данных `Grocery` и список `List` с элементами `Grocery`, который называется `groceries`. Код их определения выглядит так:

Класс данных Grocery.

```
data class Grocery(val name: String, val category: String,
                   val unit: String, val unitPrice: Double,
                   val quantity: Int)
```

Список groceries содержит пять элементов Grocery.

```
fun main(args: Array<String>) {
    val groceries = listOf(Grocery("Tomatoes", "Vegetable", "lb", 3.0, 3),
                           Grocery("Mushrooms", "Vegetable", "lb", 4.0, 1),
                           Grocery("Bagels", "Bakery", "Pack", 1.5, 2),
                           Grocery("Olive oil", "Pantry", "Bottle", 6.0, 1),
                           Grocery("Ice cream", "Frozen", "Pack", 3.0, 2))
}
```

Для начала посмотрим, как найти наименьшее или наибольшее значение в коллекции объектов.

Функции min и max работают с базовыми типами

Как вы уже знаете, для нахождения наименьшего или наибольшего значения в коллекциях с элементами базовых типов можно воспользоваться функциями `min` и `max`. Например, для нахождения наибольшего значения в `List<Int>` можно использовать следующий код:

```
val ints = listOf(1, 2, 3, 4)
val maxInt = ints.max()     //maxInt == 4
```

Функции `min` и `max` работают с базовыми типами Kotlin, потому что у этих типов существует естественный порядок. Например, `Int` можно расположить в числовом порядке, что позволит легко найти `Int` с наибольшим значением, а строки можно расположить в алфавитном порядке.

1, 2, 3, 4, 5...
«A», «B», «C»...

У чисел и строк существует естественный порядок. Это означает, что для определения наименьшего или наибольшего значения можно использовать функции min и max.

Однако функции `min` и `max` не могут использоваться с типами, не имеющими естественного порядка. Например, их невозможно использовать с `List<Grocery>` или `Set<Duck>`, потому что функции не знают, как именно следует упорядочивать элементы `Grocery` или `Duck`. Следовательно, для более сложных типов потребуется другой подход.

Функции minBy и maxBy работают со ВСЕМИ типами

Если вы хотите узнать наименьшее или наибольшее значение более сложного типа, используйте функции **`minBy`** и **`maxBy`**. Эти функции в целом похожи на `min` и `max`, но им можно передавать критерии. Например, с их помощью можно найти элемент `Grocery` с наименьшим значением `unitPrice` или элемент `Duck` с наибольшим значением `size`.

У этих предметов не существует естественного порядка. Чтобы найти наименьшее или наибольшее значение, необходимо задать некоторые критерии — например, unitPrice или quantity.

Функции `minBy` и `maxBy` получают один параметр: лямбда-выражение, которое сообщает функции, какое свойство должно использоваться для определения элемента с наибольшим или наименьшим значением. Если, например, вы захотите найти в `List<Grocery>` элемент с наибольшим значением `unitPrice` при помощи функции `maxBy`, это делается так:

```
val highestUnitPrice = groceries.maxBy { it.unitPrice }
```

Этот код означает «Найти в groceries элемент с наибольшим значением unitPrice».

А чтобы найти элемент с наименьшим значением `quantity`, используйте функцию `minBy`:

```
val lowestQuantity = groceries.minBy { it.quantity }
```

Эта строка возвращает ссылку на элемент groceries с наименьшим значением quantity.

Чтобы код правильно компилировался и работал, лямбда-выражение, передаваемое функции `minBy` или `maxBy`, должно определяться в строго определенной форме. Сейчас мы рассмотрим эту тему более подробно.

Лямбда-параметр minBy и maxBy

При вызове функции `minBy` или `maxBy` необходимо передать ей лямбда-выражение в следующей форме:

```
{ i: тип_элемента-> критерий }
```

Лямбда-выражение должно иметь один параметр, обозначенный выше `i: тип_элемента`. Тип параметра **должен соответствовать типу элементов, хранящихся в коллекции**; если вы хотите использовать одну из этих функций с `List<Grocery>`, параметр лямбда-выражения должен иметь тип `Grocery`:

```
{ i: Grocery -> criteria }
```

Так как каждое лямбда-выражение имеет один параметр известного типа, объявление параметра можно полностью опустить и обращаться к параметру в теле лямбда-выражения по обозначению `it`.

Тело лямбда-выражения задает критерий, который должен использоваться для определения наибольшего — или наименьшего — значения в коллекции. Обычно критерием является имя свойства — например, `{ it.unitPrice }`. Тип может быть любым, важно лишь то, чтобы функция могла использовать его для определения элемента, имеющего наибольшее или наименьшее значение свойства.

А как же возвращаемый тип minBy и maxBy?

При вызове функции `minBy` или `maxBy` ее возвращаемое значение соответствует типу элементов, хранящихся в коллекции. Например, если вы используете `minBy` с `List<Grocery>`, функция вернет `Grocery`. А если использовать `maxBy` с `Set<Duck>`, она вернет `Duck`.

После того как вы научились пользоваться функциями `minBy` и `maxBy`, перейдем к их ближайшим родственникам: `sumBy` и `sumByDouble`.

minBy и maxBy работают с коллекциями, содержащими объекты любого типа; они обладают большей гибкостью, чем min и max.

Если вызвать minBy или maxBy для коллекции, не содержащей ни одного элемента, функция вернет значение null.

Часто Задаваемые Вопросы

В: Функции `min` и `max` работают только с базовыми типами Kotlin, такими, как числа или `String`?

О: `min` и `max` работают с типами, для которых можно сравнить два значения и определить, что одно значение больше другого — это условие выполняется для базовых типов Kotlin. Эти объясняется тем, что каждый из них незаметно для вас реализует интерфейс `Comparable`, который определяет, как должны упорядочиваться и сравниваться экземпляры этого типа.

На самом деле функции `min` и `max` работают с любыми типами, реализующими `Comparable`. Но мы считаем, что вместо того, чтобы самостоятельно реализовать интерфейс Comparable в ваших классах, лучше использовать функции `minBy` и `maxBy`, потому что такое решение будет более гибким.

Функции sumBy и sumByDouble

Как и следовало ожидать, функции **sumBy** и **sumByDouble** возвращают сумму всех элементов коллекции, соответствующих критерию, переданному через лямбда-выражение. Например, с помощью этих функций можно просуммировать значения `quantity` для всех элементов `List<Grocery>` или вернуть сумму всех значений `unitPrice`, умноженных на `quantity`.

Функции `sumBy` и `sumByDouble` почти идентичны — они отличаются только тем, что `sumBy` работает с `Int`, а `sumByDouble` — с `Double`. Например, для вычисления суммы значений `quantity` в `Grocery` можно воспользоваться функцией `sumBy`, так как `quantity` имеет тип `Int`:

sumBy суммирует значения Int и возвращает Int.

sumByDouble суммирует Double и возвращает Double.

```
val sumQuantity = groceries.sumBy { it.quantity }
```

← *Возвращает сумму всех значений quantity в groceries.*

А чтобы вернуть сумму всех значений `unitPrice`, умноженных на `quantity`, следует использовать `sumByDouble`, так как результат `unitPrice * quantity` имеет тип `Double`:

```
val totalPrice = groceries.sumByDouble { it.quantity * it.unitPrice }
```

Лямбда-параметр sumBy и sumByDouble

Как и в случае с `minBy` и `maxBy`, вы должны передать `sumBy` и `sumByDouble` лямбда-выражение в следующей форме:

```
{ i: тип_элемента -> критерий }
```

Как и прежде, `тип_элемента` должен соответствовать типу элементов, хранящихся в коллекции. В предыдущих примерах функции используются с `List<Grocery>`, поэтому параметр лямбда-выражения должен иметь тип `Grocery`. Так как компилятор может определить этот факт самостоятельно, объявление параметра лямбда-выражения можно опустить и обращаться к параметру в теле лямбда-выражения по обозначению `it`.

Тело лямбда-выражения сообщает функции, что именно требуется просуммировать. Как говорилось ранее, это должно быть значение `Int` для функции `sumBy` или `Double` для функции `sumByDouble`. `sumBy` возвращает значение `Int`, а `sumByDouble` возвращает `Double`.

Итак, вы научились пользоваться функциями `minBy`, `maxBy`, `sumBy` и `sumByDouble`. Давайте создадим новый проект и добавим в него код, в котором используются эти функции.

Будьте осторожны!

Использовать sumBy или sumByDouble непосредственно с Map невозможно.

Тем не менее эти функции можно использовать со свойствами keys, values или entries карты Map. Например, следующая команда вычисляет сумму значений Map:

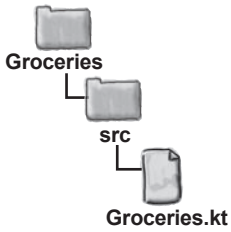
```
myMap.values.sumBy { it }
```

Создание проекта Groceries

Создайте новый проект Kotlin для JVM с именем «Groceries». Затем создайте новый файл Kotlin с именем *Groceries.kt*: выделите папку *src*, откройте меню File и выберите команду New → Kotlin File/Class. Введите имя файла «Groceries» и выберите вариант File в группе Kind.

Затем обновите свою версию *Groceries.kt*, чтобы она соответствовала нашей:

```
data class Grocery(val name: String, val category: String,
                   val unit: String, val unitPrice: Double,
                   val quantity: Int)

fun main(args: Array<String>) {
    val groceries = listOf(Grocery("Tomatoes", "Vegetable", "lb", 3.0, 3),
                           Grocery("Mushrooms", "Vegetable", "lb", 4.0, 1),
                           Grocery("Bagels", "Bakery", "Pack", 1.5, 2),
                           Grocery("Olive oil", "Pantry", "Bottle", 6.0, 1),
                           Grocery("Ice cream", "Frozen", "Pack", 3.0, 2))

    val highestUnitPrice = groceries.maxBy { it.unitPrice * 5 }
    println("highestUnitPrice: $highestUnitPrice")
    val lowestQuantity = groceries.minBy { it.quantity }
    println("lowestQuantity: $lowestQuantity")

    val sumQuantity = groceries.sumBy { it.quantity }
    println("sumQuantity: $sumQuantity")
    val totalPrice = groceries.sumByDouble { it.quantity * it.unitPrice }
    println("totalPrice: $totalPrice")
}
```

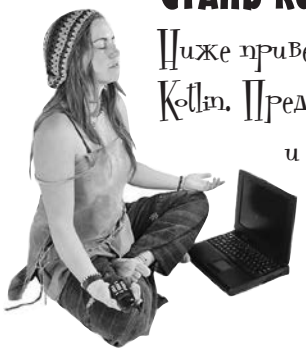


Тест-драйв

При выполнении этого кода в окне вывода IDE отображается следующий текст:

```
highestUnitPrice: Grocery(name=Olive oil, category=Pantry, unit=Bottle, unitPrice=6.0, quantity=1)
lowestQuantity: Grocery(name=Mushrooms, category=Vegetable, unit=lb, unitPrice=4.0, quantity=1)
sumQuantity: 9
totalPrice: 28.0
```

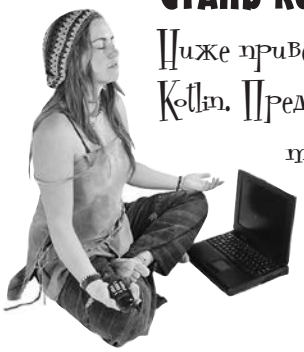
СТАНЬ компилятором



Ниже приведен полный файл с исходным кодом Kotlin. Представьте себя на месте компилятора и определите, будет ли файл нормально компилироваться. Если он не компилируется, то почему? Как бы вы исправили ошибку?

```
data class Pizza(val name: String, val pricePerSlice: Double, val quantity: Int)

fun main(args: Array<String>) {
    val ints = listOf(1, 2, 3, 4, 5)

    val pizzas = listOf(Pizza("Sunny Chicken", 4.5, 4),
            Pizza("Goat and Nut", 4.0, 1),
            Pizza("Tropical", 3.0, 2),
            Pizza("The Garden", 3.5, 3))

    val minInt = ints.minBy({ it.value })
    val minInt2 = ints.minBy({ int: Int -> int })
    val sumInts = ints.sum()
    val sumInts2 = ints.sumBy { it }
    val sumInts3 = ints.sumByDouble({ number: Double -> number })
    val sumInts4 = ints.sumByDouble { int: Int -> int.toDouble() }

    val lowPrice = pizzas.min()
    val lowPrice2 = pizzas.minBy({ it.pricePerSlice })
    val highQuantity = pizzas.maxBy { p: Pizza -> p.quantity }
    val highQuantity3 = pizzas.maxBy { it.quantity }
    val totalPrice = pizzas.sumBy { it.pricePerSlice * it.quantity }
    val totalPrice2 = pizzas.sumByDouble { it.pricePerSlice * it.quantity }
}
```

СТАНЬ компилятором. Решение



Ниже приведен полный файл с исходным кодом Kotlin. Представьте себя на месте компилятора и определите, будет ли файл нормально компилироваться. Если он не компилируется, то почему? Как бы вы исправили ошибку?

```
data class Pizza(val name: String, val pricePerSlice: Double, val quantity: Int)

fun main(args: Array<String>) {
    val ints = listOf(1, 2, 3, 4, 5)

    val pizzas = listOf(Pizza("Sunny Chicken", 4.5, 4),
            Pizza("Goat and Nut", 4.0, 1),
            Pizza("Tropical", 3.0, 2),
            Pizza("The Garden", 3.5, 3))

    val minInt = ints.minBy({ it.value })
    val minInt2 = ints.minBy({ int: Int -> int })
    val sumInts = ints.sum()
    val sumInts2 = ints.sumBy { it }
    val sumInts3 = ints.sumByDouble({ number: Double -> number it.toDouble() })
    val sumInts4 = ints.sumByDouble { int: Int -> int.toDouble() }

    val lowPrice = pizzas.min()
    val lowPrice2 = pizzas.minBy({ it.pricePerSlice })
    val highQuantity = pizzas.maxBy { p: Pizza -> p.quantity }
    val highQuantity3 = pizzas.maxBy { it.quantity }
    val totalPrice = pizzas.sumByDouble { it.pricePerSlice * it.quantity }
    val totalPrice2 = pizzas.sumByDouble { it.pricePerSlice * it.quantity }
}
```

Так как ints является списком List<Int>, 'it' соответствует Int и не имеет свойства value.

Эта строка не компилируется, так как параметр лямбда-выражения должен иметь тип Int. Лямбда-выражение можно заменить на { it.toDouble() }.

Функция min не работает с List<Pizza>.

*{ it.pricePerSlice * it.quantity } возвращает Double, так что функция sumBy не подходит. Вместо нее следует использовать sumByDouble.*

Функция filter

Следующая остановка на пути знакомства с Kotlin — функция **`filter`**. Эта функция позволяет отбирать (или *фильтровать*) коллекцию по критерию, переданному в лямбда-выражении.

Для большинства коллекций `filter` возвращает список `List`, включающий все элементы, соответствующие критерию. Полученный список можно использовать в других местах кода. Но при использовании с `Map` функция вернет `Map`. Например, следующий код использует функцию `filter` для получения списка `List` всех элементов `groceries`, у которых свойство `category` содержит «Vegetable»:

Возвращает List с элементами groceries, у которых свойство category содержит «Vegetable».

```
val vegetables = groceries.filter { it.category == "Vegetable" }
```

Как и другие функции из этой главы, лямбда-выражение, переданное функции `filter`, получает один параметр, тип которого должен соответствовать типу элементов коллекции. Так как параметр лямбда-выражения относится к известному типу, объявление параметра можно опустить, и обращаться к нему в теле лямбда-выражения по обозначению `it`.

Тело лямбда-выражения должно возвращать значение `Boolean`, используемое в качестве критерия `filter`. Функция возвращает ссылку на все элементы исходной коллекции, для которых тело лямбда-выражения дает `true`. Например, следующий код возвращает список `List` с элементами `Grocery`, у которых значение `unitPrice` больше 3.0:

```
val unitPriceOver3 = groceries.filter { it.unitPrice > 3.0 }
```

Существует целое СЕМЕЙСТВО функций filter

В Kotlin есть несколько разновидностей функции `filter`, которые могут вам пригодиться в работе. Например, функция `filterTo` работает так же, как и функция `filter`, если не считать того, что она присоединяет элементы, соответствующие заданному критерию, к другой коллекции. Функция `filterIsInstance` возвращает список `List` всех элементов, являющихся экземплярами заданного класса, а функция `filterNot` возвращает те элементы коллекции, которые *не соответствуют* переданному критерию. В следующем примере функция `filterNot` возвращает список всех элементов `Grocery`, у которых значение `category` отлично от «Frozen»:

За дополнительной информацией о фильтрах Kotlin обращайтесь к электронной документации:

https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index.html

```
val notFrozen = groceries.filterNot { it.category == "Frozen" }
```

filterNot возвращает элементы, для которых тело лямбда-выражения дает результат false.

Теперь вы знаете, как работает функция `filter`, и мы можем перейти к другой функции высшего порядка Kotlin — `map`.

Функция map и преобразования коллекций

Функция `map` берет элементы коллекции и преобразует каждый из них по заданной вами формуле. Результат преобразования возвращается в виде нового списка `List`.

Да! Функция map возвращает список List, а не Map.

Чтобы понять, как работает преобразование, представьте, что у вас имеется следующий список `List<Int>`:

```
val ints = listOf(1, 2, 3, 4)
```

Если с помощью функции `map` требуется создать новый список `List<Int>` с теми же элементами, умноженными на 2, это можно сделать так:

```
val doubleInts = ints.map { it * 2 }
```

Возвращает список List с элементами 2, 4, 6 и 8.

Также при помощи функции `map` можно создать новый список `List` со значениями name каждого элемента `Grocery` в `groceries`:

```
val groceryNames = groceries.map { it.name }
```

Создает новый список List и заполняет его значениями свойства name каждого элемента Grocery в groceries.

В данном случае функция `map` возвращает новый список `List`, а исходная коллекция остается без изменений. Если, например, вы воспользуетесь `map` для создания списка `List` каждого значения `unitPrice`, умноженного на 0,5, свойства `unitPrice` элементов `Grocery` в исходной коллекции останутся неизменными:

```
val halfUnitPrice = groceries.map { it.unitPrice * 0.5 }
```

Возвращает список List со всеми значениями unitPrice, умноженными на 0,5.

Как и в предыдущих случаях, лямбда-выражение, переданное функции `map`, получает один параметр, тип которого должен соответствовать типу элементов коллекции. Этот параметр (обычно в обозначении `it`) указывает, как именно преобразуется каждый элемент в коллекции.

Сцепленные вызовы функций

Так как функции `filter` и `map` возвращают коллекции, вызовы функций высшего порядка можно объединить в цепочку для компактного выполнения более сложных операций. Допустим, вы хотите создать список `List` всех значений `unitPrice`, умноженных на 2, если исходное значение `unitPrice` больше 3,0. Для этого можно сначала вызвать функцию `filter` для исходной коллекции, а затем вызвать `map` для преобразования результата:

```
val newPrices = groceries.filter { it.unitPrice > 3.0 }
                         .map { it.unitPrice * 2 }
```

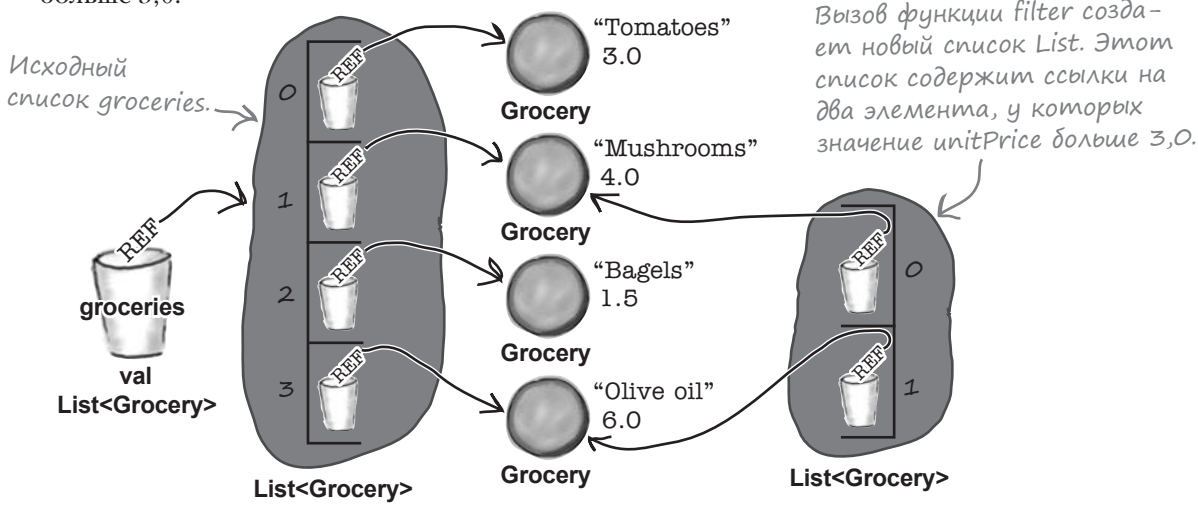
Вызывает функцию filter, а затем вызывает map для полученного списка List.

А теперь посмотрим, что же происходит при выполнении этого кода.

Что происходит при выполнении кода

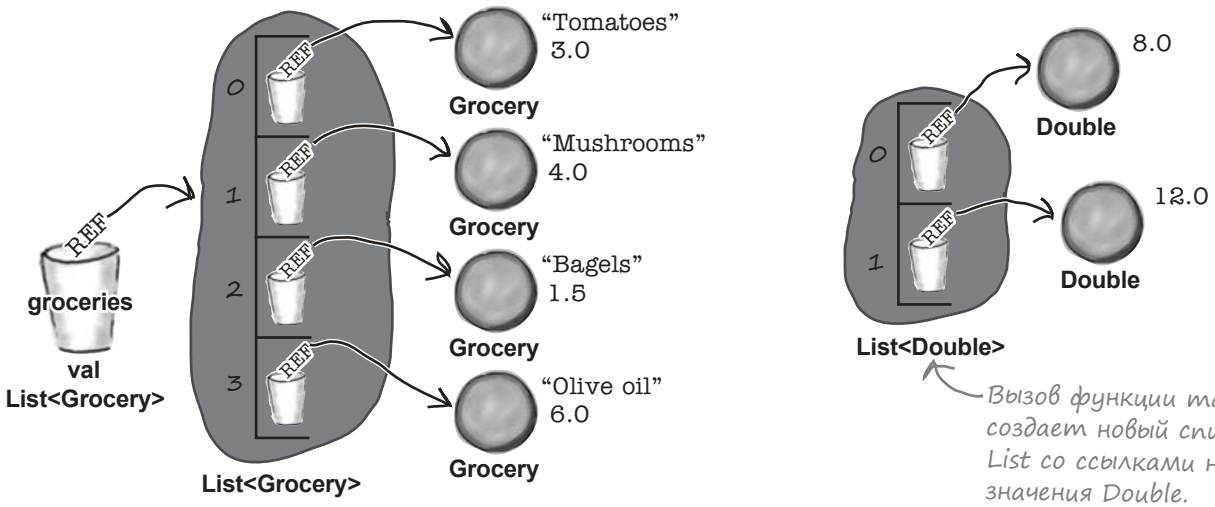
1 val newPrices = **groceries.filter { it.unitPrice > 3.0 }**
.map { it.unitPrice * 2 }

Функция `filter` вызывается для `groceries` (список `List<Grocery>`). Она создает новый список `List`, содержащий ссылки на элементы `Grocery`, у которых значение `unitPrice` больше 3,0.



2 val newPrices = groceries.filter { it.unitPrice > 3.0 }
.map { it.unitPrice * 2 }

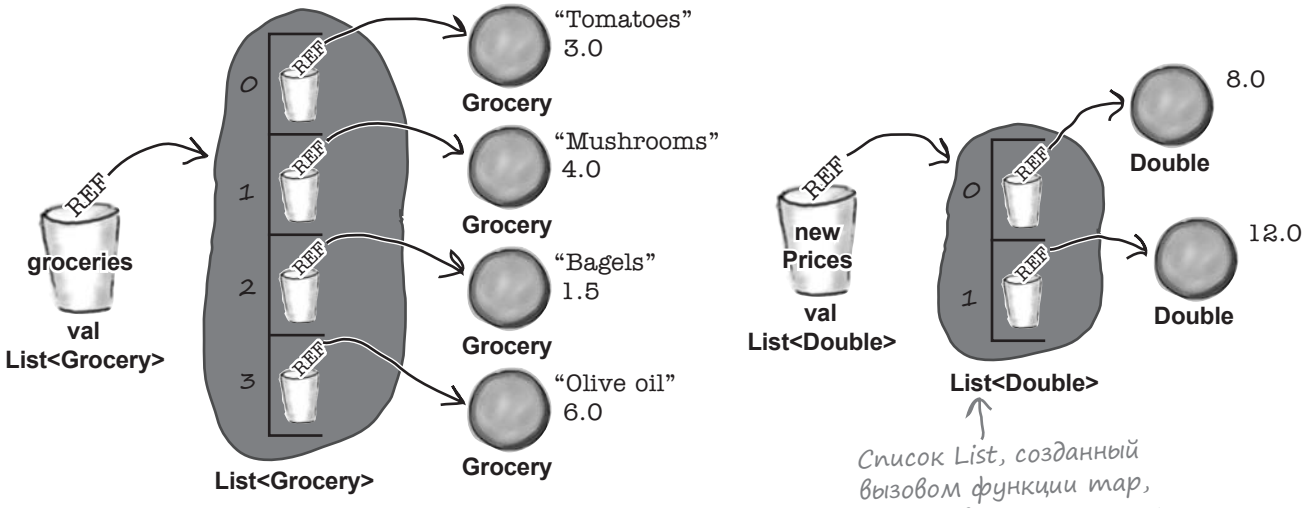
Функция `map` вызывается для нового объекта `List`. Так как лямбда-выражение `{ it.unitPrice * 2 }` возвращает `Double`, функция создает список `List<Double>` со ссылками на все значения `unitPrice`, умноженные на 2.



История продолжается...

3 `val newPrices = groceries.filter { it.unitPrice > 3.0 }`
`.map { it.unitPrice * 2 }`

Создается новая переменная `newPrices`, которой присваивается ссылка на список `List<Double>`, возвращенный функцией `map`.



Разобравшись со сцеплением вызовов функций высшего порядка, мы можем перейти к следующей функции `forEach`.

Часто Задаваемые Вопросы

В: Ранее вы сказали, что у функции `filter` есть несколько разновидностей, таких как `filterTo` и `filterNot`. Как насчет `map`? Эта функция тоже существует в нескольких вариантах?

О: Да! Среди них можно отметить функцию `mapTo` (присоединяет результаты преобразования к существующей коллекции), `mapNotNull` (убирает все значения null) и `mapValues` (работает с `Map` и возвращает `Map`). За подробностями обращайтесь по адресу

https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index.html

В: Вы говорили, что для функций высшего порядка, рассматривавшихся ранее, тип параметра лямбда-выражения должен соответствовать типу элементов коллекции. Как обеспечивается выполнение этого ограничения?

О: При помощи обобщений. Как вы, вероятно, помните из главы 10, обобщения позволяют писать код, который последовательно и целостно работает с типами. Именно они мешают включить ссылку на `Cabbage` в список `List<Duck>`. Встроенные функции высшего порядка Kotlin используют обобщения для того, чтобы они могли получать и возвращать значения только типа, подходящего для данной коллекции.

forEach работает как цикл for

Функция **forEach** работает примерно так же, как цикл `for`, — она тоже позволяет выполнить одну или несколько операций для каждого элемента коллекции. Эти операции определяются в форме лямбда-выражения.

Чтобы продемонстрировать работу `forEach`, предположим, что вы хотите перебрать все элементы списка `groceries` и вывести значение `name` каждого элемента. Вот как это делается в цикле `for`:

```
for (item in groceries) {
    println(item.name)
}
```

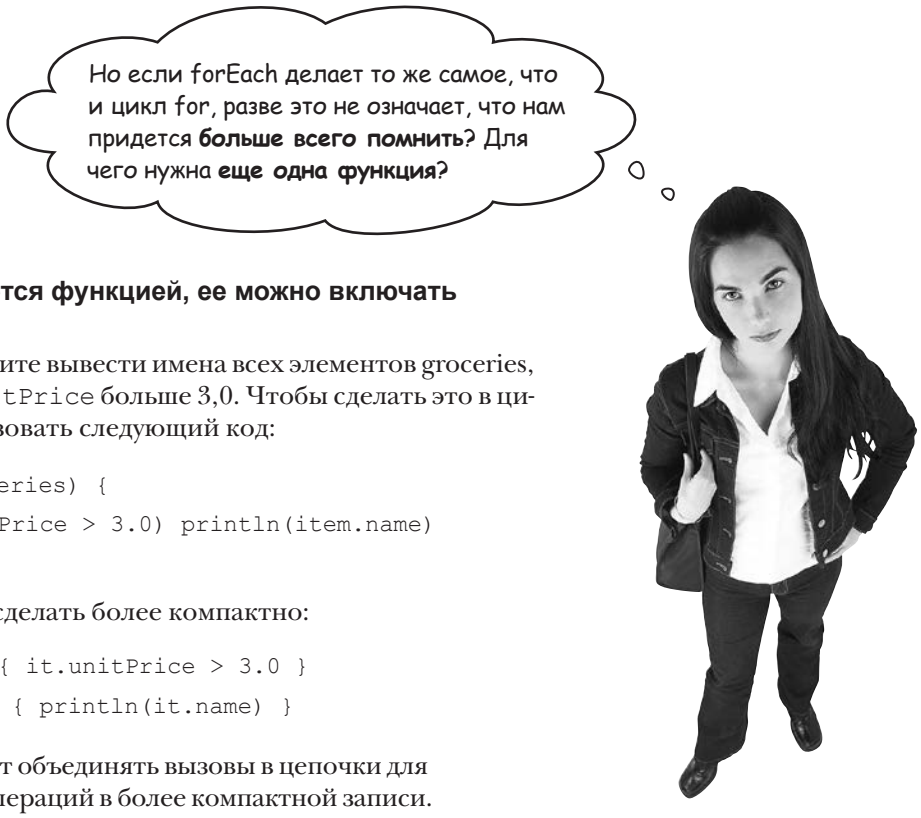
Можно использовать forEach со списками, множествами, элементами Map, а также свойствами keys и values.

А вот эквивалентный код с использованием функции `forEach`:

```
groceries.forEach { println(it.name) }
```

{ println(it.name) } — лямбда-выражение, передаваемое функции forEach. Тело лямбда-выражения может состоять из нескольких строк.

Оба примера делают одно и то же, но вариант с `forEach` получается более компактным.



Так как `forEach` является функцией, ее можно включать в цепочки вызовов.

Представьте, что вы хотите вывести имена всех элементов groceries, у которых значение `unitPrice` больше 3,0. Чтобы сделать это в цикле `for`, можно использовать следующий код:

```
for (item in groceries) {
    if (item.unitPrice > 3.0) println(item.name)
}
```

Но то же самое можно сделать более компактно:

```
groceries.filter { it.unitPrice > 3.0 }
         .forEach { println(it.name) }
```

Итак, `forEach` позволяет объединять вызовы в цепочки для выполнения сложных операций в более компактной записи.

Давайте поближе познакомимся с `forEach`.

У forEach нет возвращаемого значения

Как и у других функций, представленных в этой главе, лямбда-выражение, передаваемое функции `forEach`, получает один параметр, тип которого соответствует типу элементов коллекции. А поскольку параметр имеет известный тип, объявление параметра можно опустить и обращаться к нему в теле лямбда-выражения по обозначению `it`.

Однако в отличие от других функций, тело лямбда-выражения имеет возвращаемый тип `Unit`. Это означает, что вы не сможете использовать `forEach` для возвращения результата вычислений, потому что не сможете к этому результату обратиться. Тем не менее у проблемы существует обходное решение.

В лямбда-выражениях доступны переменные

Как вы уже знаете, в теле цикла `for` можно обращаться к переменным, определенным за пределами цикла. Например, следующий код определяет строковую переменную с именем `itemNames`, которая затем обновляется в теле цикла `for`:

```
var itemNames = ""
for (item in groceries) {
    itemNames += "${item.name} "
}
println("itemNames: $itemNames")
```

Переменную itemNames можно обновлять внутри тела цикла for.

Когда лямбда-выражение передается функции высшего порядка (такой, как `forEach`), в нем доступны те же переменные, *хотя они определены за пределами лямбда-выражения*. Вместо того чтобы передавать результат вычислений в возвращаемом значении, вы можете обновить переменную из тела лямбда-выражения. Например, следующий код вполне допустим:

Переменную itemNames также можно обновлять в теле лямбда-выражения, передаваемого forEach.

```
var itemNames = ""
groceries.forEach({ itemNames += "${it.name} " })
println("itemNames: $itemNames")
```

Переменные, определенные за пределами лямбда-выражения и доступные для него, иногда называются **замыканием** лямбда-выражения. Или если выражаться по-научному, *лямбда-выражение может обращаться к своему замыканию*. А если лямбда-выражение использует переменную `itemNames` в своем теле, мы говорим, что *переменная захвачена замыканием лямбда-выражения*.

Вы узнали, как пользоваться функцией `forEach`, и теперь можно обновить код проекта.

Замыкание означает, что лямбда-выражение может обращаться к любой захваченной им локальной переменной.

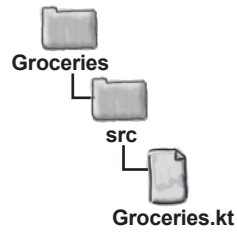
Обновление проекта Groceries

Добавим в проект Groceries код, в котором используются функции `filter`, `map` и `forEach`. Обновите свою версию *Groceries.kt* в проекте, чтобы она соответствовала нашей (изменения выделены жирным шрифтом):

```
data class Grocery(val name: String, val category: String,
                   val unit: String, val unitPrice: Double,
                   val quantity: Int)

fun main(args: Array<String>) {
    val groceries = listOf(Grocery("Tomatoes", "Vegetable", "lb", 3.0, 3),
                           Grocery("Mushrooms", "Vegetable", "lb", 4.0, 1),
                           Grocery("Bagels", "Bakery", "Pack", 1.5, 2),
                           Grocery("Olive oil", "Pantry", "Bottle", 6.0, 1),
                           Grocery("Ice cream", "Frozen", "Pack", 3.0, 2))
```

Удалите эти строки.

```
    val highestUnitPrice = groceries.maxBy { it.unitPrice * 5 }
    println("highestUnitPrice: $highestUnitPrice")
    val lowestQuantity = groceries.minBy { it.quantity }
    println("lowestQuantity: $lowestQuantity")

    val sumQuantity = groceries.sumBy { it.quantity }
    println("sumQuantity: $sumQuantity")
    val totalPrice = groceries.sumByDouble { it.quantity * it.unitPrice }
    println("totalPrice: $totalPrice")
```



Добавьте все эти строки.

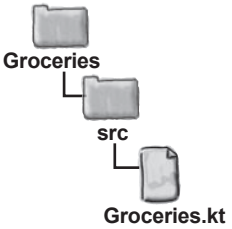
```
    val vegetables = groceries.filter { it.category == "Vegetable" }
    println("vegetables: $vegetables")
    val notFrozen = groceries.filterNot { it.category == "Frozen" }
    println("notFrozen: $notFrozen")

    val groceryNames = groceries.map { it.name }
    println("groceryNames: $groceryNames")
    val halfUnitPrice = groceries.map { it.unitPrice * 0.5 }
    println("halfUnitPrice: $halfUnitPrice")

    val newPrices = groceries.filter { it.unitPrice > 3.0 }
            .map { it.unitPrice * 2 }
    println("newPrices: $newPrices")
```

Продолжение на следующей странице.

Продолжение...

Добавьте эти строки в функцию main.

```
    println("Grocery names: ")
    groceries.forEach { println(it.name) }

    println("Groceries with unitPrice > 3.0: ")
    groceries.filter { it.unitPrice > 3.0 }
            .forEach { println(it.name) }

    var itemNames = ""
    groceries.forEach({ itemNames += "${it.name} " })
    println("itemNames: $itemNames")
}
```



Опробуем этот код на практике.

Тест-драйв

При выполнении этого кода в окне вывода IDE отображается следующий текст:

```
vegetables: [Grocery(name=Tomatoes, category=Vegetable, unit=lb, unitPrice=3.0, quantity=3),
Grocery(name=Mushrooms, category=Vegetable, unit=lb, unitPrice=4.0, quantity=1)]
notFrozen: [Grocery(name=Tomatoes, category=Vegetable, unit=lb, unitPrice=3.0, quantity=3),
Grocery(name=Mushrooms, category=Vegetable, unit=lb, unitPrice=4.0, quantity=1),
Grocery(name=Bagels, category=Bakery, unit=Pack, unitPrice=1.5, quantity=2),
Grocery(name=Olive oil, category=Pantry, unit=Bottle, unitPrice=6.0, quantity=1)]
groceryNames: [Tomatoes, Mushrooms, Bagels, Olive oil, Ice cream]
halfUnitPrice: [1.5, 2.0, 0.75, 3.0, 1.5]
newPrices: [8.0, 12.0]
Grocery names:
Tomatoes
Mushrooms
Bagels
Olive oil
Ice cream
Groceries with unitPrice > 3.0:
Mushrooms
Olive oil
itemNames: Tomatoes Mushrooms Bagels Olive oil Ice cream
```

После обновления кода проекта попробуйте выполнить упражнение, а потом мы перейдем к следующей функции высшего порядка.

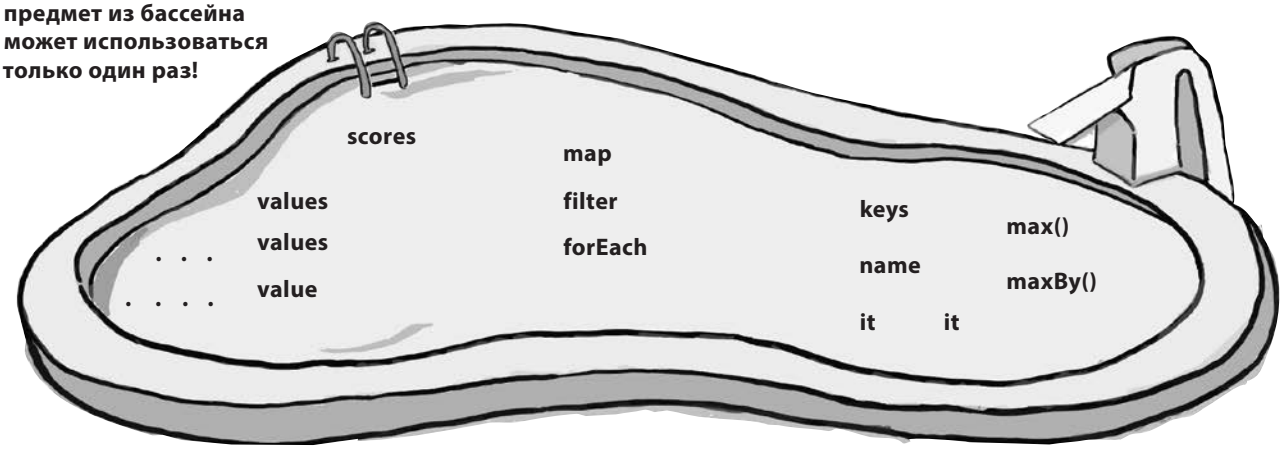
У бассейна

Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: завершить функцию `getWinners` в классе `Contest`, чтобы она возвращала множество `Set<T>` с данными участников, получившими наивысшую оценку (score), и выводила имена всех победителей.

```
abstract class Pet(var name: String)

class Cat(name: String) : Pet(name)

class Dog(name: String) : Pet(name)

class Fish(name: String) : Pet(name)

class Contest<T: Pet>() {
    var scores: MutableMap<T, Int> = mutableMapOf()

    fun addScore(t: T, score: Int = 0) {
        if (score >= 0) scores.put(t, score)
    }

    fun getWinners(): Set<T> {
        val highScore = .....
        val winners = scores.....{ ..... == highScore }.....
        winners..... { println("Winner: ${.....}") }
        return winners
    }
}
```

Этот код может показаться знакомым, так как мы написали другую его версию в главе 10.

Примечание: каждый предмет из бассейна может использоваться только один раз!



У бассейна. Решение

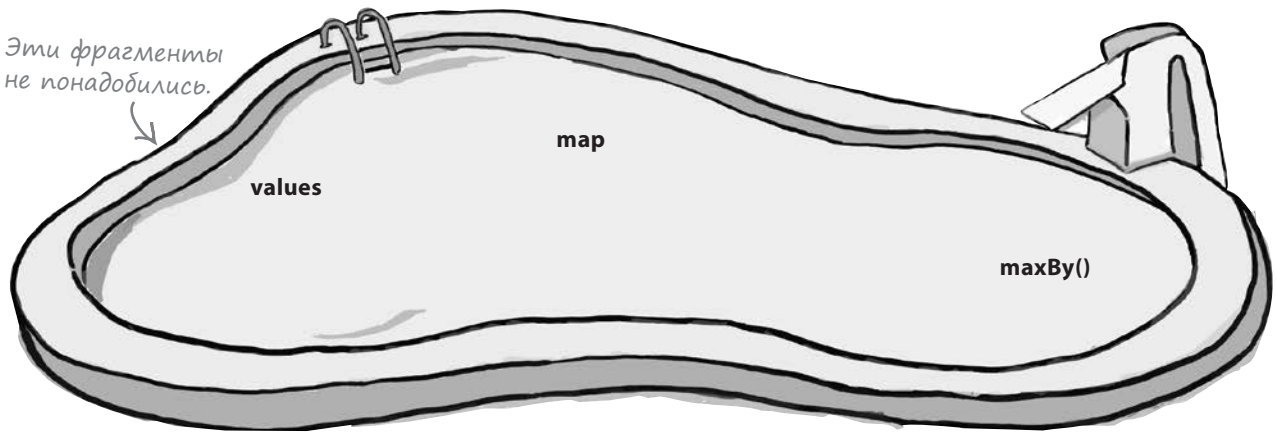
Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: завершить функцию `getWinners` в классе `Contest`, чтобы она возвращала множество `Set<T>` с данными участников, получившими наивысшую оценку (score), и выводила имена всех победителей.

```
abstract class Pet(var name: String)

class Cat(name: String) : Pet(name)

class Dog(name: String) : Pet(name)

class Fish(name: String) : Pet(name)

class Contest<T: Pet>() {
    var scores: MutableMap<T, Int> = mutableMapOf()

    fun addScore(t: T, score: Int = 0) {
        if (score >= 0) scores.put(t, score)
    }

    fun getWinners(): Set<T> {
        val highScore = scores.values.max()
        val winners = scores.filter { it.value == highScore }.keys
        winners.forEach { println("Winner: ${it.name}") }
        return winners
    }
}
```

Оценки хранятся в виде значений Int в MutableMap с именем scores, поэтому это выражение получает значение наивысшей оценки.

Содержимое scores фильтруется, чтобы в нем остались только элементы со значением highScore. Затем свойство keys результата используется для получения победителей.

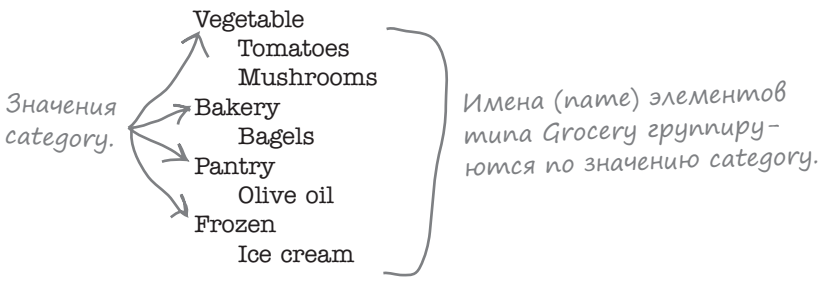
Функция forEach используется для вывода имен (name) всех победителей.



Функция groupBy используется для разбиения коллекции на группы

Следующая функция, которую мы рассмотрим, — **groupBy**. Эта функция позволяет группировать элементы коллекции по некоторому критерию, — например, значению одного из свойств. Например, его можно использовать (в сочетании с другими вызовами функций), чтобы вывести имя элементов `Grocery`, сгруппированных по значению `category`:

Учтите, что функцию groupBy нельзя применять к массиву Map напрямую, но ее можно вызывать для свойств keys, values или entries.

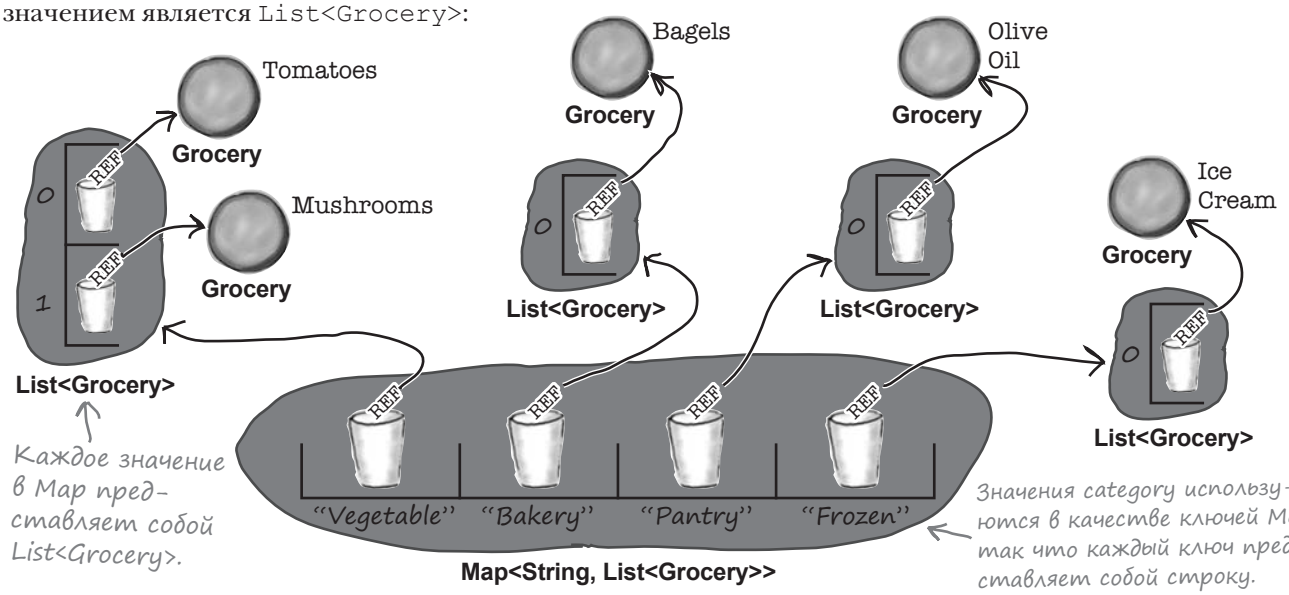


Функция `groupBy` получает один параметр: лямбда-выражение, которое используется для определения способа группировки элементов коллекции. Например, следующий код группирует элементы `groceries` (`List<Grocery>`) по значению `category`:

```
val groupByCategory = groceries.groupBy { it.category }
```

Это означает «сгруппировать элементы groceries по значению category».

Функция `groupBy` возвращает `Map`. Ключи определяются критерием, переданным в теле лямбда-выражения, а каждое ассоциированное значение представляет собой список `List` элементов из исходной коллекции. Например, приведенный выше код создает ассоциативный массив `Map`, ключами которого являются значения `category` элементов `Grocery`, а каждым значением является `List<Grocery>`:



Функция groupBy может использоваться в цепочках вызовов

Так как функция `groupBy` возвращает `Map` со значениями `List`, вы можете вызывать для возвращаемого значения другие функции высшего порядка, по аналогии с тем, как это делалось для функций `filter` и `map`.

Допустим, вы хотите вывести значение каждой категории для `List<Grocery>`, а также значение `name` для каждого элемента `Grocery`, свойство `category` которого содержит это значение. Для этого можно при помощи функции `groupBy` сгруппировать элементы `Grocery` по каждому значению `category`, а затем воспользоваться функцией `forEach` для перебора элементов полученного массива `Map`:

```
groceries.groupBy { it.category }.forEach {
    //...
}
```

groupBy возвращает Map; это означает, что для возвращаемого значения можно вызвать функцию forEach.

Так как функция `groupBy` использует значения category элементов `Grocery` в качестве ключей, мы можем вывести их, передавая код `println(it.key)` функции `forEach` в лямбда-выражении:

```
groceries.groupBy { it.category }.forEach {
    println(it.key)
    //...
}
```

Выводит ключи Map (значения category элементов Grocery).

А поскольку каждое значение `Map` представляет собой `List<Grocery>`, мы можем присоединить вызов `forEach` для вывода значения `name` каждого элемента:

```
groceries.groupBy { it.category }.forEach {
    println(it.key)
    it.value.forEach { println("    ${it.name}") }
}
```

Эта строка получает соответствующее значение для ключа Map. Так как значение представляет собой List<Grocery>, мы можем вызвать для него forEach, чтобы вывести значение свойства name элемента Grocery.

При выполнении приведенного выше кода будет получен следующий результат:

```
Vegetable
    Tomatoes
    Mushrooms
Bakery
    Bagels
Pantry
    Olive oil
Frozen
    Ice cream
```

Теперь вы знаете, как использовать функцию `groupBy`. Перейдем к последней функции, которая будет представлена в нашем обзоре, — функции `fold`.

Как использовать функцию fold

Функция fold может вызываться для свойств keys, values и entries массива Map, но не для самого объекта Map напрямую.

Пожалуй, функция **fold** является самой гибкой из функций высшего порядка Kotlin. С функцией `fold` вы можете задать исходное значение и выполнить некоторую операцию для каждого элемента коллекции. Например, с ее помощью можно перемножить все элементы `List<Int>` и вернуть результат или же выполнить конкатенацию значений `name` всех элементов из `List<Grocery>` — и все это в одной строке кода!

В отличие от других функций, встречавшихся в этой главе, `fold` получает два параметра: исходное значение и выполняемую операцию, заданную лямбда-выражением. Таким образом, если у вас имеется список `List<Int>`

```
val ints = listOf(1, 2, 3)
```

функция `fold` может использоваться для прибавления всех элементов к исходному значению 0:

```
val sumOfInts = ints.fold(0) { runningSum, item -> runningSum + item }
```

Исходное значение.

Сообщает функции, что вы хотите прибавить значение каждого элемента коллекции к исходному значению.

Первый параметр функции `fold` определяет исходное значение — в данном случае 0. Этим параметром может быть любой тип, но обычно это один из базовых типов Kotlin — число, строка и т. д.

Вторым параметром является лямбда-выражение, которое описывает операцию, выполняемую с исходным значением для каждого элемента в коллекции. В нашем примере требуется прибавить каждый элемент к исходному знаачению, поэтому используется следующее лямбда-выражение:

```
{ runningSum, item -> runningSum + item }
```

Здесь мы решили присвоить параметрам лямбда-выражения имена runningSum и item. Тем не менее вы можете присвоить параметрам любые допустимые имена переменных.

Лямбда-выражение, переданное `fold`, получает два параметра, которым в нашем примере присвоены имена `runningSum` и `item`.

Тип первого параметра лямбда-выражения `runningSum` определяется заданным исходным значением. Он инициализируется этим значением — в приведенном примере `runningSum` представляет собой значение `Int`, инициализированное значением 0.

Второй параметр лямбда-выражения `item` имеет тот же тип, что и элементы коллекции. В приведенном примере функция `fold` вызывается для `List<Int>`, поэтому `item` имеет тип `Int`.

Тело лямбда-выражения определяет операцию, которая должна быть выполнена для каждого элемента коллекции. После этого результат присваивается переменной первого параметра лямбда-выражения. В приведенном примере функция берет значение `runningSum`, прибавляет к нему значение текущего элемента, после чего присваивает полученное значение `runningSum`. Когда перебор всех элементов коллекции будет завершен, функция `fold` вернет итоговое значение этой переменной.

Разберемся в том, что происходит при вызове функции `fold`.

Под капотом: функция fold

Вот что происходит при выполнении кода

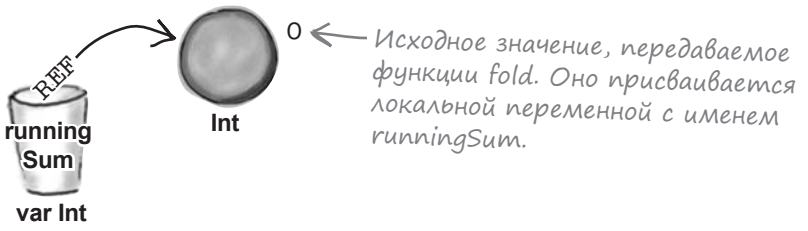
```
val sumOfInts = ints.fold(0) { runningSum, item -> runningSum + item }
```

где `ints` определяется следующим образом:

```
val ints = listOf(1, 2, 3)
```

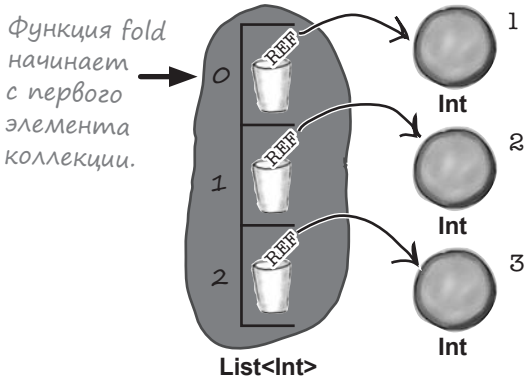
1 val sumOfInts = ints.fold(**0**) { **runningSum**, item -> runningSum + item }

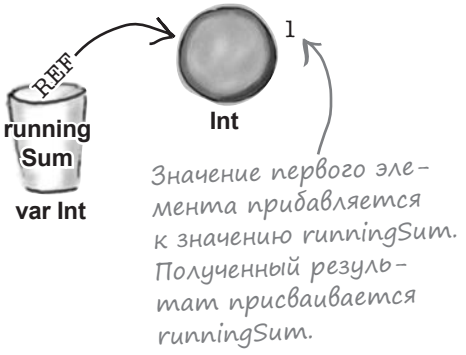
Эта строка создает переменную `Int` с именем `runningSum`, инициализированным значением 0. Эта переменная является локальной по отношению к функции `fold`.



2 val sumOfInts = ints.fold(0) { runningSum, item -> **runningSum + item** }

Функция берет значение первого элемента коллекции (`Int` со значением 1) и прибавляет его к значению `runningSum`. Полученное новое значение 1 присваивается `runningSum`.

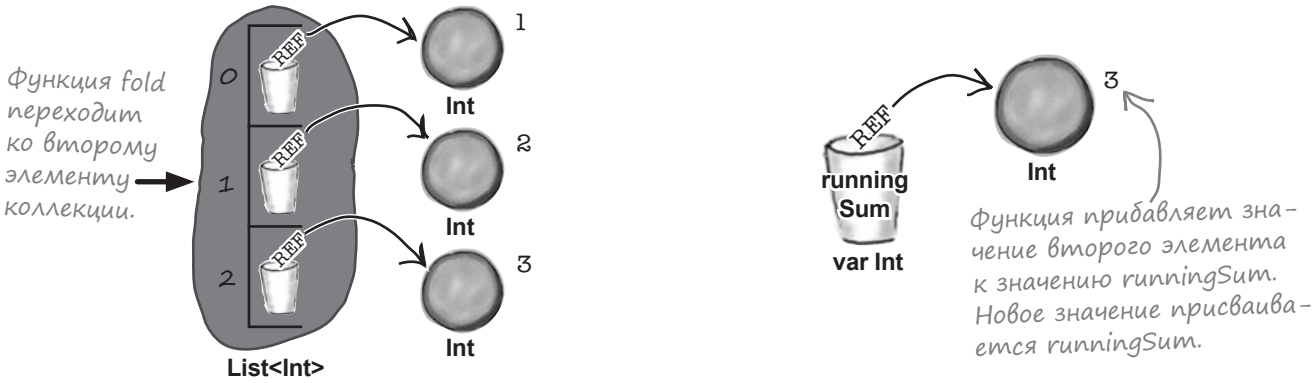




История продолжается...

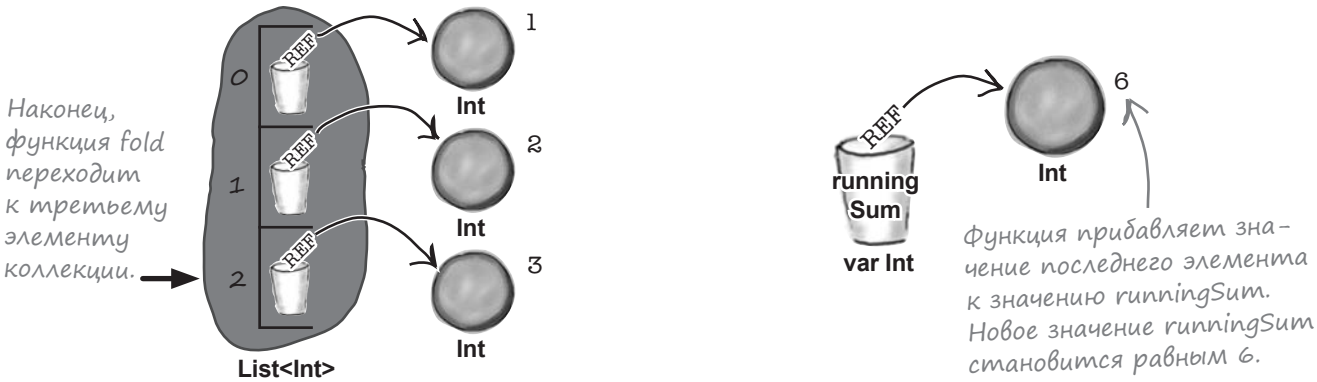
3 val sumOfInts = ints.fold(0) { runningSum, item -> **runningSum + item** }

Функция переходит ко второму элементу коллекции — `Int` со значением 2. Он прибавляется к переменной `runningSum`, значение которой увеличивается до 3.



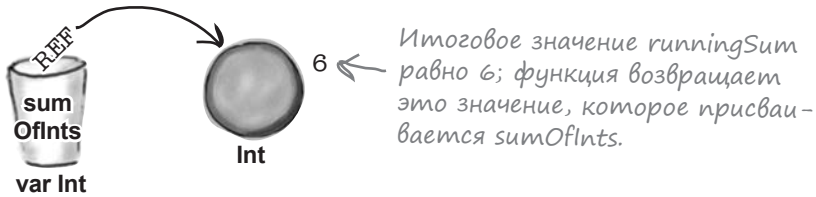
4 val sumOfInts = ints.fold(0) { runningSum, item -> **runningSum + item** }

Функция переходит к третьему и последнему элементу коллекции: `Int` со значением 3. Это значение прибавляется к `runningSum`, поэтому значение `runningSum` увеличивается до 6.



5 **val sumOfInts =** ints.fold(0) { runningSum, item -> runningSum + item }

Так как в коллекции не осталось других элементов, функция возвращает итоговое значение `runningSum`. Это значение присваивается новой переменной с именем `sumOfInts`.



Примеры использования fold

Мы показали, как использовать функцию `fold` для суммирования значений в `List<Int>`. Давайте рассмотрим еще несколько примеров.

Определение произведения в List<Int>

Если вы хотите перемножить все числа в `List<Int>` и вернуть результат, для этого можно передать функции `fold` исходное значение 1 и лямбда-выражение, тело которого выполняет умножение:

```
ints.fold(1) { runningProduct, item -> runningProduct * item }
```

Инициализировать runningProduct значением 1.

Умножить runningSum на значение каждого элемента.

Конкатенация имен из элементов списка List<Grocery>

Чтобы вернуть строку со значениями свойства name каждого элемента `Grocery` в `List<Grocery>`, передайте функции `fold` исходное значение `""` и лямбда-выражение, тело которого выполняет конкатенацию:

Также для выполнения подобных операций может использоваться функция joinToString.

```
groceries.fold("") { string, item -> string + " ${item.name}" }
```

Инициализировать строку значением "".

Фактически это означает: выполнить операцию string = string + " ${item.name}" для каждого элемента groceries.

Вычитание суммы из исходного значения

Также при помощи функции `fold` можно определить, сколько денег у вас останется после покупки всех товаров из списка `List<Grocery>`. Для этого в качестве исходного значения указывается доступная сумма денег, а тело лямбда-выражения используется для вычитания из нее значения `unitPrice` каждого элемента, умноженного на `quantity`:

```
groceries.fold(50.0) { change, item
    -> change - item.unitPrice * item.quantity }
```

Инициализировать переменную change значением 50.

*Вычитает общую стоимость (unitPrice * quantity) из change для каждого элемента в groceries.*

Теперь вы знаете, как использовать функции `groupBy` и `fold`, и мы можем перейти к обновлению кода проекта.

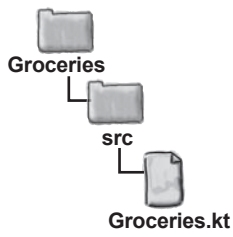
Обновление проекта Groceries

Мы добавим в проект Groceries код, в котором используются функции `groupBy` и `fold`. Обновите свою версию *Groceries.kt* в проекте, чтобы она соответствовала нашей (изменения выделены жирным шрифтом):

```
data class Grocery(val name: String, val category: String,
                   val unit: String, val unitPrice: Double,
                   val quantity: Int)

fun main(args: Array<String>) {
    val groceries = listOf(Grocery("Tomatoes", "Vegetable", "lb", 3.0, 3),
                           Grocery("Mushrooms", "Vegetable", "lb", 4.0, 1),
                           Grocery("Bagels", "Bakery", "Pack", 1.5, 2),
                           Grocery("Olive oil", "Pantry", "Bottle", 6.0, 1),
                           Grocery("Ice cream", "Frozen", "Pack", 3.0, 2))
```

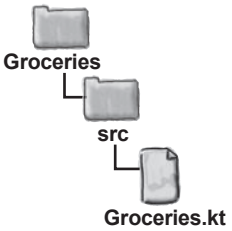
Эти строки больше не нужны, их можно удалить.

```
    val vegetables = groceries.filter { it.category == "Vegetable" }
    println("vegetables: $vegetables")
    val notFrozen = groceries.filterNot { it.category == "Frozen" }
    println("notFrozen: $notFrozen")

    val groceryNames = groceries.map { it.name }
    println("groceryNames: $groceryNames")
    val halfUnitPrice = groceries.map { it.unitPrice * 0.5 }
    println("halfUnitPrice: $halfUnitPrice")

    val newPrices = groceries.filter { it.unitPrice > 3.0 }
            .map { it.unitPrice * 2 }
    println("newPrices: $newPrices")

    println("Grocery names: ")
    groceries.forEach { println(it.name) }

    println("Groceries with unitPrice > 3.0: ")
    groceries.filter { it.unitPrice > 3.0 }
            .forEach { println(it.name) }

    var itemNames = ""
    groceries.forEach({ itemNames += "${it.name} " })
    println("itemNames: $itemNames")
```



Продолжение на следующей странице.

Продолжение...



Добавьте эти строки в функцию main.

```
    groceries.groupBy { it.category }.forEach {
        println(it.key)
        it.value.forEach { println("    ${it.name}") }
    }

    val ints = listOf(1, 2, 3)
    val sumOfInts = ints.fold(0) { runningSum, item -> runningSum + item }
    println("sumOfInts: $sumOfInts")

    val productOfInts = ints.fold(1) { runningProduct, item -> runningProduct * item }
    println("productOfInts: $productOfInts")

    val names = groceries.fold("") { string, item -> string + " ${item.name}" }
    println("names: $names")

    val changeFrom50 = groceries.fold(50.0) { change, item
                                -> change - item.unitPrice * item.quantity }
    println("changeFrom50: $changeFrom50")
}
```

Проверим, как работает этот код.

Тест-драйв

При выполнении этого кода в окне вывода IDE отображается следующий текст:

```
Vegetable
  Tomatoes
  Mushrooms
Bakery
  Bagels
Pantry
  Olive oil
Frozen
  Ice cream
sumOfInts: 6
productOfInts: 6
names:  Tomatoes Mushrooms Bagels Olive oil Ice cream
changeFrom50: 22.0
```

Часто Задаваемые Вопросы

В: Вы сказали, что некоторые функции высшего порядка не могут использоваться с `Map` напрямую. Почему?

О: Потому что определение `Map` несколько отличается от определений `List` и `Set`, и это влияет на то, какие функции могут работать с этими коллекциями. Во внутренней реализации `List` и `Set` наследуют поведение от интерфейса с именем `Collection`, который в свою очередь наследует поведение, определенное в интерфейсе `Iterable`. Однако `Map` не наследуется ни от одного из этих интерфейсов. Это означает, что `List` и `Set` являются типами `Iterable`, а `Map` — нет.

Это важное отличие, потому что такие функции, как `fold`, `forEach` и `groupBy`, предназначены для работы с `Iterable`. А поскольку `Map` не реализует `Iterable`, при попытке вызвать любую из этих функций непосредственно для `Map` вы получите ошибку компиляции.

К счастью, свойства `entries`, `keys` и `values` коллекции `Map` реализуют `Iterable`: `entries` и `keys` сами являются множествами `Set`, а `values` наследуется от интерфейса `Collection`. А следовательно, хотя вы не сможете вызвать `groupBy` и `fold` напрямую для `Map`, их можно использовать со свойствами `Map`.

В: Функции `fold` всегда нужно передавать исходное значение? Нельзя ли просто использовать первый элемент коллекции в качестве исходного значения?

О: При использовании функции `fold` *необходимо* задать исходное значение. Этот параметр обязателен, опустить его нельзя.

Но если вы хотите использовать первый элемент коллекции в качестве исходного значения, можно воспользоваться функцией **`reduce`**. Эта функция работает аналогично `fold`, не считая того, что ей не нужно передавать исходное значение. В качестве исходного значения автоматически используется первый элемент коллекции.

В: Функция `fold` перебирает коллекции в определенном порядке? А можно выбрать обратный порядок перебора?

О: Функции `fold` и `reduce` перебирают элементы коллекций слева направо, начиная с первого элемента.

Если вы хотите перебирать элементы в обратном порядке, используйте функции **`foldRight`** и **`reduceRight`**. Эти функции работают с массивами и списками `List`, но не с `Set` или `Map`.

В: Я могу обновлять переменные в замыкании лямбда-выражения?

О: Да.

Напомним, что замыканием лямбда-выражения называется совокупность переменных, определенных вне тела лямбда-выражения, доступных для лямбда-выражения. В отличие от некоторых языков (таких, как Java), эти переменные можно обновлять в теле лямбда-выражения — при условии, что они были определены с ключевым словом `var`.

В: В Kotlin есть и другие функции высшего порядка?

О: Да. В Kotlin слишком много функций высшего порядка, чтобы их можно было описать в одной главе, поэтому мы решили ограничиться только теми, которые, на наш взгляд, являются самыми важными или полезными. Впрочем, после того как вы научились пользоваться этими функциями, вы наверняка сможете воспользоваться полученными знаниями и применить их с другими функциями.

Полный список функций Kotlin (включая функции высшего порядка) приведен в электронной документации:

https://kotlinlang.org/api/latest/jvm/stdlib/index.html

Возьми в руку карандаш

Следующий код определяет класс данных `Grocery` и список `List<Grocery>` с именами `groceries`:

```
data class Grocery(val name: String, val category: String,
                   val unit: String, val unitPrice: Double,
                   val quantity: Int)

val groceries = listOf(Grocery("Tomatoes", "Vegetable", "lb", 3.0, 3),
                       Grocery("Mushrooms", "Vegetable", "lb", 4.0, 1),
                       Grocery("Bagels", "Bakery", "Pack", 1.5, 2),
                       Grocery("Olive oil", "Pantry", "Bottle", 6.0, 1),
                       Grocery("Ice cream", "Frozen", "Pack", 3.0, 2))
```

Напишите код для определения суммы, которая должна быть потрачена на овощи (категория `Vegetable`).

.....

Создайте список `List` с названиями (`name`) всех товаров, общие затраты на которые ниже 5.0.

.....

Выведите общие затраты по каждой категории.

.....

.....

.....

Выведите названия (`name`) всех товаров, которые не разливаются в бутылки (`"Bottle"`), сгруппированные по типу упаковки (`unit`).

.....

.....

.....

.....

Ответы на с. 422.

Путаница с сообщениями

Ниже приведена короткая программа Kotlin. Один блок в программе пропущен. Ваша задача — сопоставить блоки-кандидаты (слева) с выводом, который вы получите при подстановке этого блока. Используются не все строки вывода, а некоторые могут использоваться более одного раза. Проведите линию от каждого блока к подходящему варианту вывода.

```
fun main(args: Array<String>) {
    val myMap = mapOf("A" to 4, "B" to 3, "C" to 2, "D" to 1, "E" to 2)
    var x1 = ""
    var x2 = 0
    println("$x1$x2")
}
```

Код блока подставляется сюда.

Соедините каждый блок с одним из вариантов вывода.

Блоки:

```
x1 = myMap.keys.fold("") { x, y -> x + y}
x2 = myMap.entries.fold(0) { x, y -> x * y.value }
```

```
x2 = myMap.values.groupBy { it }.keys.sumBy { it }
```

```
x1 = "ABCDE"
x2 = myMap.values.fold(12) { x, y -> x - y }
```

```
x2 = myMap.entries.fold(1) { x, y -> x * y.value }
```

```
x1 = myMap.values.fold("") { x, y -> x + y }
```

```
x1 = myMap.values.fold(0) { x, y -> x + y }
                 .toString()
x2 = myMap.keys.groupBy { it }.size
```

Варианты вывода:

```
10
```

```
ABCDE0
```

```
ABCDE48
```

```
43210
```

```
432120
```

```
48
```

```
125
```

Ответы на с. 423.

Возьми в руку карандаш
Решение

Следующий код определяет класс данных `Grocery` и список `List<Grocery>` с именами `groceries`:

```
data class Grocery(val name: String, val category: String,
                   val unit: String, val unitPrice: Double,
                   val quantity: Int)

val groceries = listOf(Grocery("Tomatoes", "Vegetable", "lb", 3.0, 3),
                       Grocery("Mushrooms", "Vegetable", "lb", 4.0, 1),
                       Grocery("Bagels", "Bakery", "Pack", 1.5, 2),
                       Grocery("Olive oil", "Pantry", "Bottle", 6.0, 1),
                       Grocery("Ice cream", "Frozen", "Pack", 3.0, 2))
```

Напишите код для определения суммы, которая должна быть потрачена на овощи (категория `Vegetable`).

Фильтрация по category и последующее суммирование общих затрат.

```
groceries.filter { it.category == "Vegetable" }.sumByDouble { it.unitPrice * it.quantity }
```

Создайте список `List` с названиями (`name`) всех товаров, общие затраты на которые ниже 5.0.

```
groceries.filter { it.unitPrice * it.quantity < 5.0 }.map { it.name }
```

*Фильтрация по unitPrice * quantity и последующий вызов map для преобразования результата.*

Выведите общие затраты по каждой категории.

Для каждой категории...

```
groceries.groupBy { it.category }.forEach {
    println("${it.key}: ${it.value.sumByDouble { it.unitPrice * it.quantity }}")
}
```

...вывести ключ, за которым следует результат sumByDouble для каждого значения.

Выведите названия (`name`) всех товаров, которые не разливаются в бутылки (`"Bottle"`), сгруппированные по типу упаковки (`unit`).

Результаты группируются по unit.

```
groceries.filterNot { it.unit == "Bottle" }.groupBy { it.unit }.forEach {
    println(it.key)
    it.value.forEach { println("    ${it.name}") }
}
```

Получить элементы, у которых значение unit отлично от "Bottle".

Вывести каждый ключ полученной карты Map.

Каждое значение в Map представляет собой список List<Grocery>, поэтому мы можем перебрать каждый список List при помощи forEach и вывести значение name для каждого элемента.

Путаница с сообщениями. Решение

Ниже приведена короткая программа Kotlin. Один блок в программе пропущен. Ваша задача — сопоставить блоки-кандидаты (слева) с выводом, который вы получите при подстановке этого блока. Используются не все строки вывода, а некоторые могут использоваться более одного раза. Проведите линию от каждого блока к подходящему варианту вывода.

```
fun main(args: Array<String>) {
    val myMap = mapOf("A" to 4, "B" to 3, "C" to 2, "D" to 1, "E" to 2)
    var x1 = ""
    var x2 = 0
    println("$x1$x2")
}
```

Код блока подставляется сюда.

Блоки:

```
x1 = myMap.keys.fold("") { x, y -> x + y}
x2 = myMap.entries.fold(0) { x, y -> x * y.value }
```

```
x2 = myMap.values.groupBy { it }.keys.sumBy { it }
```

```
x1 = "ABCDE"
x2 = myMap.values.fold(12) { x, y -> x - y }
```

```
x2 = myMap.entries.fold(1) { x, y -> x * y.value }
```

```
x1 = myMap.values.fold("") { x, y -> x + y }
```

```
x1 = myMap.values.fold(0) { x, y -> x + y }
                 .toString()
x2 = myMap.keys.groupBy { it }.size
```

Возможный вывод:

```
10
```

```
ABCDE0
```

```
ABCDE48
```

```
43210
```

```
432120
```

```
48
```

```
125
```

Ваш инструментарий Kotlin

Глава 12 осталась позади, а ваш инструментарий пополнился встроенными функциями высшего порядка.

Весь код для этой главы можно загрузить по адресу https://tinyurl.com/HFKotlin.

ГЛАВА 12

КЛЮЧЕВЫЕ МОМЕНТЫ

- Функции `minBy` и `maxBy` используются для определения наименьшего или наибольшего значения в коллекции. Каждая из этих функций получает один параметр — лямбда-выражение, тело которого задает критерии функции. Возвращаемый тип соответствует типу элементов коллекции.
- Функции `sumBy` или `sumByDouble` для возвращения суммы элементов коллекции. Лямбда-выражение, передаваемое в параметре, определяет суммируемую величину. Если суммируются данные типа `Int`, используйте функцию `sumBy`, а для данных типа `Double` — функцию `sumByDouble`.
- Функция `filter` выполняет поиск (фильтрацию) коллекции по заданному критерию. Этот критерий задается лямбда-выражением, тело которого должно возвращать `Boolean`. `filter` обычно возвращает `List`. При использовании этой функции с `Map` возвращается `Map`.
- Функция `map` преобразует элементы коллекции по критерию, определяемому лямбда-выражением. Она возвращает список `List`.
- Функция `forEach` работает по принципу цикла `for`. Она позволяет выполнить одну или несколько операций для каждого элемента коллекции.
- Функция `groupBy` используется для разбиения коллекции на группы. Она получает один параметр — лямбда-выражение, которое определяет, как должны группироваться элементы. Функция возвращает карту `Map`, у которой ключи определяются критерием лямбда-выражения, а каждое значение представлено списком `List`.
- Функция `fold` позволяет задать исходное значение и выполнить некоторую операцию с каждым элементом коллекции. Она получает два параметра: исходное значение и лямбда-выражение, определяющее выполняемую операцию.

Пара слов на прощание...

Надеемся, вы хорошо провели время в мире Kotlin.

Конечно, жаль расставаться, но новые знания заслуживают того, чтобы применить их на практике. В конце книги еще осталось несколько приложений, просмотрите их и переходите к самостоятельной работе. Счастливого пути!

Параллельный запуск

Некоторые задачи лучше выполнять в фоновом режиме. Если вы загружаете данные с медленного внешнего сервера, то вряд ли захотите, чтобы остальной код простаивал и дожидался завершения загрузки. В подобных ситуациях **на помощь приходят сопрограммы**. Сопрограммы позволяют писать код, предназначенный для ***асинхронного выполнения***. А это означает *сокращение времени простоя, более удобное взаимодействие с пользователем и улучшенная масштабируемость приложений*. Продолжайте читать, и вы узнаете, как говорить с Бобом, одновременно слушая Сьюзи.

Построение драм-машины

Сопрограммы, или корутины, позволяют создавать фрагменты кода, которые могут запускаться **асинхронно**. Вместо того чтобы выполнять все блоки кода последовательно, один за другим, сопрограммы позволяют организовать их одновременное выполнение.

Использование сопрограмм означает, что вы можете запустить фоновую задачу (например, загрузку данных с внешнего сервера), и коду не придется дожидаться завершения этого задания прежде, чем делать что-либо еще. Приложение работает более плавно, к тому же оно лучше масштабируется.

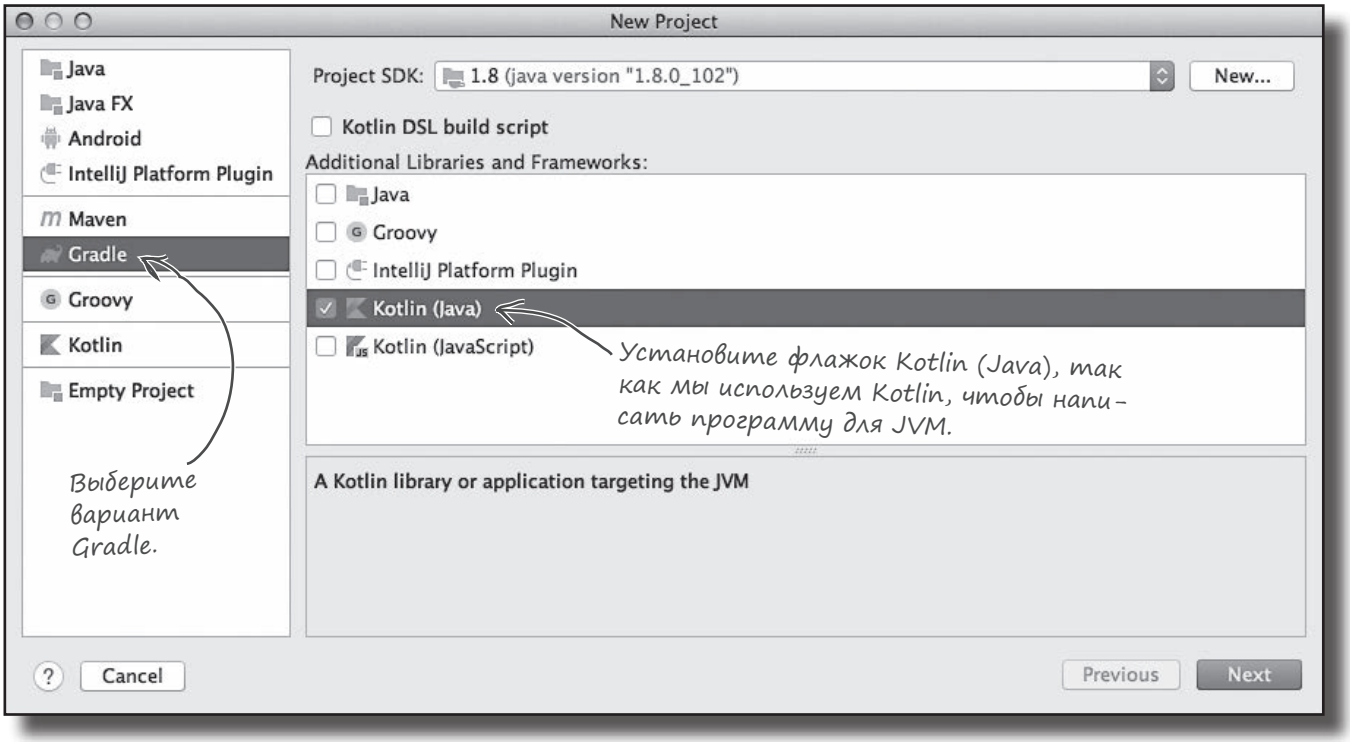
Чтобы вы увидели, к каким различиям может привести применение сопрограмм в вашем коде, представьте, что вы хотите построить драм-машину на основании кода, воспроизводящего ритмические последовательности. Начнем с создания проекта Drum Machine. Выполните следующие действия:

Код в этом приложении предназначен для Kotlin 1.3 и выше. В более ранних версиях сопрограммы имели экспериментальный статус.

Gradle — система сборки, предназначенная для компиляции и развертывания кода, а также включения любых сторонних библиотек, необходимых для вашего кода. Здесь мы используем Gradle, чтобы добавить сопрограммы в свой проект (это произойдет через несколько страниц).

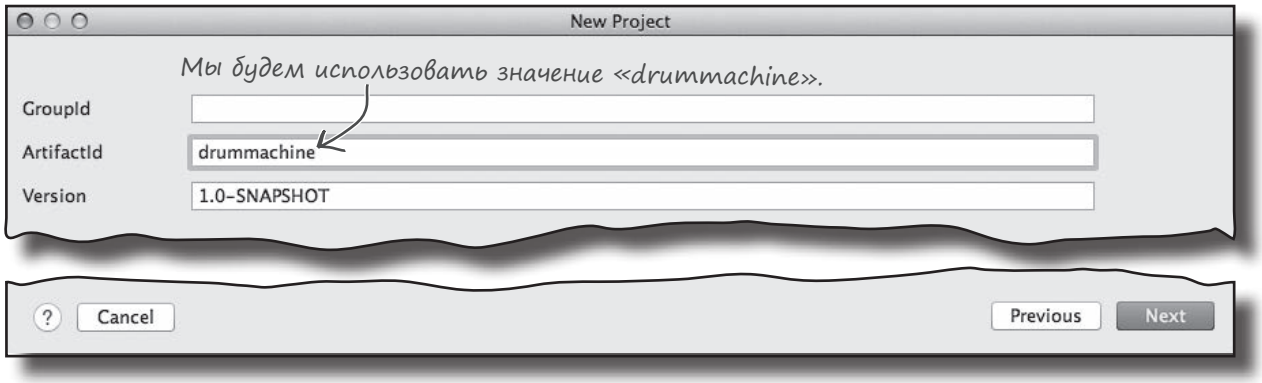
1. Создание нового проекта GRADLE

Чтобы написать код, использующий сопрограммы, необходимо создать новый проект **Gradle**, чтобы настроить его для сопрограмм. Создайте новый проект, выберите вариант Gradle и установите флажок Kotlin (Java). Затем щелкните на кнопке Next.



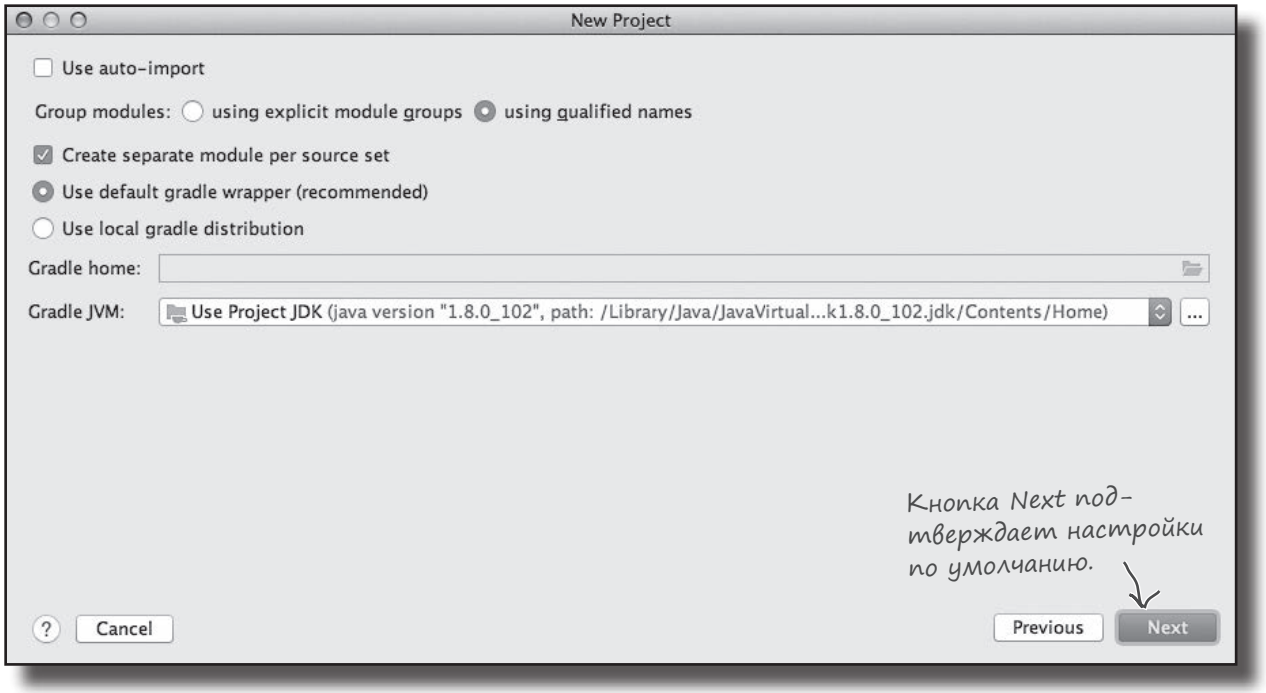
2. Ввод Artifact ID

При создании проекта Gradle необходимо задать Artifact ID. Фактически это имя проекта, но по общепринятым соглашениям записанное в нижнем регистре. Введите в поле Artifactid строку «drummachine» и щелкните на кнопке Next.



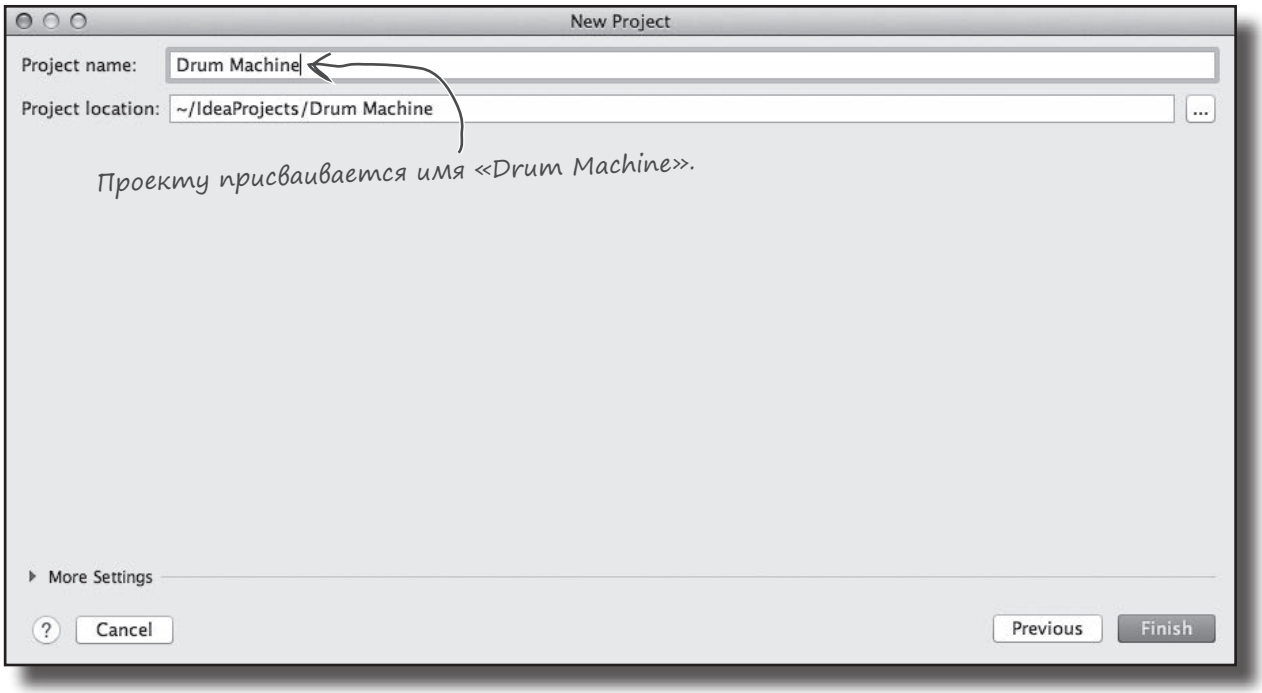
3. Настройка конфигурации

На следующем шаге вносятся изменения в стандартную конфигурацию проекта. Щелкните на кнопке Next, чтобы подтвердить настройки по умолчанию.



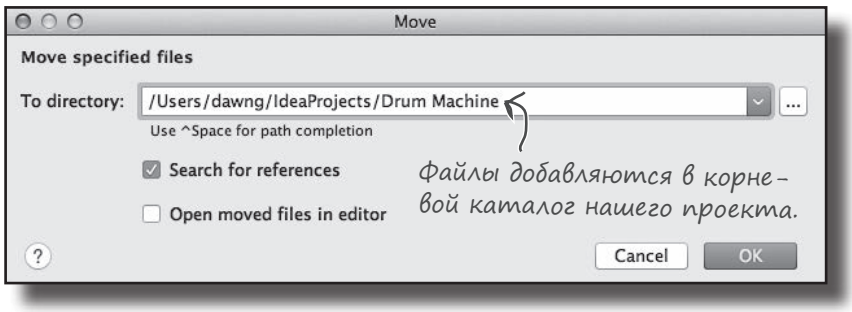
4. Определение имени проекта

Остается задать имя проекта. Введите имя «Drum Machine» и щелкните на кнопке Finish. IntelliJ IDEA создает ваш проект.



Добавление аудиофайлов

После того как проект Drum Machine будет создан, в него необходимо добавить пару аудиофайлов. Загрузите файлы *crash_cymbal.aiff* и *toms.aiff* по адресу *https://tinyurl.com/HFKotlin*, после чего перетащите их в свой проект. По запросу подтвердите, что вы хотите переместить файлы в папку *Drum Machine*.



Добавление кода в проект

Мы получили готовый код для воспроизведения ритмических последовательностей; теперь этот код нужно добавить в проект. Создайте новый файл Kotlin с именем *Beats.kt*: выделите папку *src/main/kotlin*, откройте меню File и выберите команду New → Kotlin File/Class. Введите имя файла «Beats» и выберите вариант File в группе Kind. Затем обновите свою версию *Beats.kt*, чтобы она соответствовала нашей:

```
import java.io.File
import javax.sound.sampled.AudioSystem

fun playBeats(beats: String, file: String) {
    val parts = beats.split("x")
    var count = 0
    for (part in parts) {
        count += part.length + 1
        if (part == "") {
            playSound(file)
        } else {
            Thread.sleep(100 * (part.length + 1L))
            if (count < beats.length) {
                playSound(file)
            }
        }
    }
}

fun playSound(file: String) {
    val clip = AudioSystem.getClip()
    val audioInputStream = AudioSystem.getAudioInputStream(
        File(
            file
        )
    )
    clip.open(audioInputStream)
    clip.start()
}

fun main() {
    playBeats("x-x-x-x-x-x-", "toms.aiff")
    playBeats("x-----x-----", "crash_cymbal.aiff")
}
```

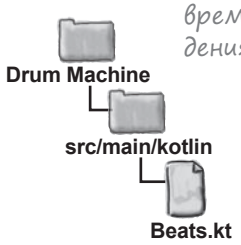
В коде используются две библиотеки Java, которые необходимо импортировать. О командах import подробнее рассказано в приложении III.

Параметр beats задает ритм воспроизведения, а параметр file — воспроизводимый звуковой файл.

playSound вызывается по одному разу для каждого символа "x" в параметре beats.

Приостанавливает текущий поток выполнения, чтобы дать время для воспроизведения звукового файла.

Воспроизводит заданный аудиофайл.



Воспроизводятся указанные звуковые файлы.

Посмотрим, что происходит при выполнении этого кода.

Тест-драйв

При запуске приложения сначала воспроизводится звук барабана (*toms.aiff*), а потом звук тарелок (*crash_cymbal.aiff*). Это происходит последовательно, так что сначала воспроизводится звук барабана, а потом начинают звучать тарелки:

Бам! Бам! Бам! Бам! Бам! Бам! Дынц! Дынц!

Код воспроизводит звуковой файл toms шесть раз.

После этого звук тарелок воспроизводится дважды.

Но что если барабан и тарелки должны звучать параллельно?

Использование сопрограмм для параллельного воспроизведения

Как упоминалось ранее, сопрограммы позволяют асинхронно выполнять несколько блоков кода. В нашем примере это означает, что код воспроизведения звука барабана можно разместить в сопрограмме, чтобы он воспроизводился одновременно с тарелками.

Для этого необходимо сделать две вещи:

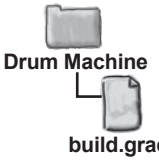
1. **Включить сопрограммы в проект в качестве зависимости.**
   Сопрограммы реализуются отдельной библиотекой Kotlin. Чтобы пользоваться ими, необходимо включить эту библиотеку в проект.

2. **Запустить сопрограмму.**
   Сопрограмма включает код воспроизведения звука барабана.

Давайте займемся этим.

1. Добавление зависимости для сопрограмм

Если вы хотите использовать сопрограммы в своем проекте, сначала необходимо добавить их в проект в виде зависимости. Для этого откройте файл *build.gradle* и обновите раздел зависимостей `dependencies` следующим образом:

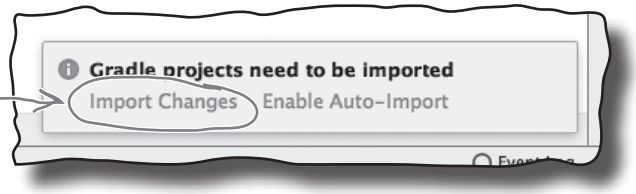



```
dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.0.1'
}
```

Добавьте эту строку в build.gradle, чтобы добавить библиотеку coroutines в проект.

Щелкните на ссылке Import Changes, чтобы изменения вступили в силу:

Щелкните на ссылке Import Changes, когда вам будет предложено.



На следующем шаге необходимо внести изменения в функцию `main`, чтобы в ней использовалась сопрограмма.

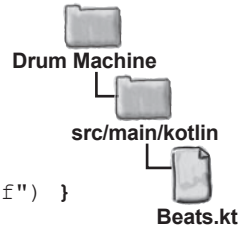
2. Запуск сопрограммы

Наш код должен запускать файл со звуком барабана в отдельной сопрограмме в фоновом режиме. Для этого код воспроизведения будет заключен в вызов **`GlobalScope.launch`** из библиотеки `kotlinx.coroutines`. Это приводит к тому, что код, воспроизводящий файл со звуком барабана, будет выполняться в фоновом режиме, чтобы два звука воспроизводились параллельно.

Ниже приведена новая функция `main` — обновите свою версию кода и приведите ее в соответствие с нашей (изменения выделены жирным шрифтом):

```
...
import kotlinx.coroutines.*
...

fun main() {
    GlobalScope.launch { playBeats("x-x-x-x-x-x-", "toms.aiff") }
    playBeats("x-----x-----", "crash_cymbal.aiff")
}
```

Добавьте эту строку, чтобы мы могли использовать функции из библиотеки coroutines в своем коде.



Сопрограмма запускается в фоновом режиме.

Посмотрим, как работает фоновое выполнение сопрограмм, и применим новый код на практике.

Тест-драйв

Когда мы запускаем код, звуки барабанов и тарелок воспроизводятся одновременно. Звук барабана воспроизводится в отдельной сопрограмме в фоновом режиме.

Бам! Бам! Бам! Бам! Бам! Бам!
Дынц! Дынц!

На этот раз барабаны и тарелки звучат одновременно.

Итак, вы увидели, как запустить сопрограмму в фоновом режиме и к какому эффекту это приводит. Теперь давайте углубимся в эту тему.

Сопрограмма напоминает облегченный программный поток

Во внутренней реализации запуск сопрограммы напоминает запуск отдельного программного потока (или просто **потока**). Потоки часто используются в других языках (например, Java); сопрограммы и потоки могут выполняться параллельно и взаимодействовать друг с другом. Однако принципиальное различие заключается в том, что сопрограммы в коде **работают эффективнее потоков.**

Запуск потока и его дальнейшее выполнение обходятся достаточно дорого в плане быстродействия. Обычно максимальное количество потоков, одновременно выполняемых процессором, ограничено, поэтому эффективнее будет свести количество потоков к минимуму. С другой стороны, сопрограммы по умолчанию используют общий пул потоков, и один поток может обслуживать несколько сопрограмм. Так как в этом случае задействовано меньшее количество потоков, для асинхронного выполнения операций более эффективно использовать сопрограммы.

В нашем коде функция `GlobalScope.launch` используется для запуска новой сопрограммы в фоновом режиме. При этом создается новый поток, в котором выполняется сопрограмма, чтобы файлы *toms.aiff* и *crash_cymbal.aiff* воспроизводились с отдельных потоках. Так как в приложении более эффективно свести количество потоков к минимуму, давайте посмотрим, как организовать воспроизведение звуковых файлов в разных сопрограммах, но в одном потоке.

Использование runBlocking для выполнения сопрограмм в одной области видимости

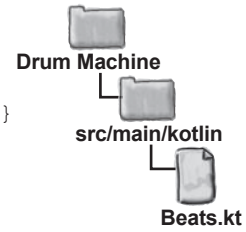
Если вы хотите, чтобы ваш код выполнялся в том же потоке, но в других сопрограммах, используйте функцию **runBlocking**. Это функция высшего порядка, которая блокирует текущий поток до того, как завершится выполнение переданного ей кода. Функция `runBlocking` определяет область видимости, которая наследуется переданным ей кодом. В нашем примере эта область видимости используется для выполнения других сопрограмм в том же потоке.

Ниже приведена новая версия функции `main`, в которой используется эта возможность. Обновите свою версию кода (изменения выделены жирным шрифтом):

```
fun main() {
    runBlocking {
        GlobalScope.launch { playBeats("x-x-x-x-x-x-", "toms.aiff") }
        playBeats("x-----x-----", "crash_cymbal.aiff")
    }
}
```

Выполняемый код заключается в вызов runBlocking.

Удалите ссылку на GlobalScope.



Обратите внимание: новая сопрограмма запускается вызовом `launch` вместо `GlobalScope.launch`. Это объясняется тем, что мы хотим запустить сопрограмму, выполняемую в том же, а не в отдельном фоновом потоке, а исключение этой ссылки на `GlobalScope` позволяет сопрограмме использовать ту же область видимости, что и `runBlocking`.

А теперь посмотрим, что происходит при выполнении кода.

Тест-драйв

При выполнении этого кода звуковые файлы воспроизводятся, но последовательно, а не параллельно.

Дынц! Дынц! Бам! Бам! Бам! Бам! Бам! Бам!

Сначала звук тарелок воспроизводится дважды.

Затем звук барабана воспроизводится шесть раз.

Что же пошло не так?

Thread.sleep приостанавливает текущий ПОТОК

Как вы, возможно, заметили, при добавлении функции `playBeats` в проект была включена следующая строка:

```
Thread.sleep(100 * (part.length + 1L))
```

Этот вызов использует библиотеку Java для приостановки текущего потока, чтобы у воспроизводимого звукового файла было время для воспроизведения, и не позволяет потоку делать что-либо еще. Так как звуковые файлы воспроизводятся в том же потоке, они уже не могут звучать параллельно, хотя и воспроизводятся в разных сопрограммах.

Функция delay приостанавливает текущую СОПРОГРАММУ

В таких ситуациях лучше использовать функцию **`delay`** из библиотеки сопрограмм. По своему действию она похожа на `Thread.sleep`, но вместо приостановки текущего *потока* она приостанавливает текущую *сопрограмму*. Сопрограмма приостанавливается на заданное время, позволяя выполниться другому коду в том же потоке. Например, следующий код приостанавливает сопрограмму на 1 секунду:

```
delay(1000)
```

Функция delay создает задержку, но работает эффективнее Thread.sleep.

Функция `delay` может использоваться в следующих ситуациях.

- **Из сопрограммы.**
  В следующем примере функция `delay` вызывается из сопрограммы:

```
GlobalScope.launch {
    delay(1000)
    //Код, который выполняется через 1 с.
}
```

Здесь мы запускаем сопрограмму, а затем задерживаем выполнение ее кода на 1 секунду.

- **Из функций, которые могут быть приостановлены компилятором.**

В нашем примере функция `delay` должна использоваться внутри функции `playBeats`; это означает, что компилятору нужно сообщить, что функция `playBeats` — а также функция `main`, которая вызывает ее, — могут приостанавливаться. Для этого обе функции должны быть помечены префиксом `suspend`:

```
suspend fun playBeats(beats: String, file: String) {
    ...
}
```

Префикс suspend сообщает компилятору, что функция может приостанавливаться.

Когда вы вызываете функцию, которая может быть приостановлена (например, delay) из другой функции, эта функция должна быть помечена ключевым словом suspend.

Полный код проекта приведен на следующей странице.

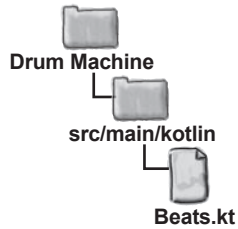
Полный код проекта

Ниже приведен полный код проекта Drum Machine — обновите свою версию *Beats.kt* и включите в нее изменения, выделенные жирным шрифтом:

```
import java.io.File
import javax.sound.sampled.AudioSystem
import kotlinx.coroutines.*

suspend fun playBeats(beats: String, file: String) {
    val parts = beats.split("x")
    var count = 0
    for (part in parts) {
        count += part.length + 1
        if (part == "") {
            playSound(file)
        } else {
            Thread.sleep delay(100 * (part.length + 1L))
            if (count < beats.length) {
                playSound(file)
            }
        }
    }
}

fun playSound(file: String) {
    val clip = AudioSystem.getClip()
    val audioInputStream = AudioSystem.getAudioInputStream(
        File(
            file
        )
    )
    clip.open(audioInputStream)
    clip.start()
}

suspend fun main() {
    runBlocking {
        launch { playBeats("x-x-x-x-x-x-", "toms.aiff") }
        playBeats("x-----x-----", "crash_cymbal.aiff")
    }
}
```

Пометьте функцию playBeats префиксом suspend, чтобы она могла вызывать функцию delay.

Замените Thread.sleep на delay.



Пометьте функцию main префиксом suspend, чтобы она могла вызывать функцию playBeats.

Посмотрим, что происходит при выполнении этого кода.

Тест-драйв

При выполнении этого кода барабан и тарелки, как и прежде, звучат параллельно. Но на этот раз звуковые файлы воспроизводятся в разных сопрограммах в одном потоке.

Бам! Бам! Бам! Бам! Бам! Бам!
Дынц! Дынц!

Барабан и тарелки по-прежнему звучат параллельно, но на этот раз мы используем более эффективный способ воспроизведения звуковых файлов.

Об использовании сопрограмм можно больше узнать здесь:

https://kotlinlang.org/docs/reference/coroutines-overview.html

КЛЮЧЕВЫЕ МОМЕНТЫ

- Сопрограммы позволяют выполнять код асинхронно. В частности, они могут пригодиться для выполнения фоновых операций.
- Сопрограмма представляет собой нечто вроде облегченного потока. Сопрограммы по умолчанию используют общий пул потоков, и в одном потоке могут выполняться несколько сопрограмм.
- Чтобы использовать сопрограммы, создайте проект Gradle и добавьте библиотеку coroutines в файл *build.gradle* как зависимость.
- Используйте функцию `launch` для запуска новой сопрограммы.
- Функция `runBlocking` блокирует текущий поток до завершения выполнения содержащегося в ней кода.
- Функция `delay` приостанавливает код на заданный промежуток времени. Она может использоваться внутри сопрограммы или внутри функции, помеченной префиксом `suspend`.

Весь код для этого приложения можно загрузить по адресу https://tinyurl.com/HFKotlin.

Код под контролем

Хороший код должен работать, это все знают. Но при любых изменениях кода появляется опасность внесения новых ошибок, из-за которых код не будет работать так, как положено. Вот почему так важно провести *тщательное тестирование*: вы узнаете о любых проблемах в коде *до того, как он будет развернут в среде реальной эксплуатации*. В этом приложении рассматриваются ***JUnit*** и ***KotlinTest*** — библиотеки **модульного тестирования**, которые дадут вам *дополнительные гарантии безопасности*.

В Kotlin можно использовать существующие библиотеки тестирования

Как вы уже знаете, код Kotlin может компилироваться в Java, JavaScript или в машинный код, поэтому вы можете использовать существующие библиотеки для выбранной платформы. Что касается тестирования, это означает, что для тестирования кода Kotlin можно использовать самые популярные библиотеки из Java и JavaScript.

Посмотрим, как организовать модульное тестирование кода Kotlin с применением JUnit.

Добавление библиотеки JUnit

Модульное тестирование используется для тестирования отдельных блоков исходного кода (например, классов или функций).

Библиотека **JUnit** (*https://junit.org*) — самая популярная библиотека модульного тестирования Java.

Чтобы использовать JUnit в своем проекте Kotlin, сначала необходимо включить библиотеки JUnit в проект. Для этого откройте меню File и выберите команду Project Structure → Libraries — или, если вы используете проект Gradle, добавьте следующие строки в файл *build.gradle*:

```
dependencies {
    ....
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.3.1'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.3.1'
    test { useJUnitPlatform() }
    ....
}
```

Эти строки включают в проект версию 5.3.1 библиотек JUnit. Измените номер, если вы хотите использовать другую версию.

После того как код будет скомпилирован, вы сможете провести тестирование — щелкните правой кнопкой мыши на имени класса или функции и выберите команду Run.

Чтобы продемонстрировать использование JUnit с Kotlin, мы напишем тест для класса с именем `Totaller`: класс инициализируется значением `Int` и вычисляет накапливаемую сумму значений, которые прибавляются к ней при помощи функции `add`:

```
class Totaller(var total: Int = 0) {
    fun add(num: Int): Int {
        total += num
        return total
    }
}
```

Посмотрим, как может выглядеть тест JUnit для этого класса.

Создание тестового класса JUnit

Ниже приведен класс теста JUnit с именем `TotallerTest`, предназначенный для тестирования `Totaller`:

```
import org.junit.jupiter.api.Assertions.*
import org.junit.jupiter.api.Test

class TotallerTest {
    @Test
    fun shouldBeAbleToAdd3And4() {
        val totaller = Totaller()

        assertEquals(3, totaller.add(3))
        assertEquals(7, totaller.add(4))
        assertEquals(7, totaller.total)
    }
}
```

При использовании кода из пакетов JUnit необходимо их импортировать. За дополнительной информацией о командах import обращайтесь к приложению III.

Класс TotallerTest используется для тестирования Totaller.

Аннотация помечает следующую функцию как тестируемую.

Создание объекта Totaller.

Проверяем, что при добавлении 3 возвращаемое значение будет равно 3.

Если теперь добавить 4, то возвращаемое значение должно быть равно 7.

Проверить, что возвращаемое значение соответствует значению переменной total.

Каждый тест хранится в функции, префиксом которой является аннотация `@Test`. Аннотации используются для включения расширенной информации о коде; аннотация `@Test` сообщает программным инструментам: «Это тестовая функция».

Тесты состоят из *действий* и *проверочных утверждений*. Действия представляют собой фрагменты кода, которые что-то *делают*, тогда как проверочные утверждения — блоки кода, которые что-то *проверяют*. В приведенном выше коде проверочное утверждение с именем `assertEquals` используется для проверки того, что два переданных ему значения равны. Если они не равны, `assertEquals` выдает исключение, и тест завершается неудачей.

За дополнительной информацией об использовании JUnit обращайтесь по адресу https://junit.org

В приведенном выше примере тестовой функции присвоено имя `shouldBeAbleToAdd3And4`. Однако мы можем воспользоваться одной редко встречающейся возможностью Kotlin: разработчик может заключить имя функции в обратные апострофы (`), а затем включить в имя функции пробелы и другие знаки, чтобы оно выглядело более информативно. Пример:

```
....
@Test
fun `should be able to add 3 and 4 - and it mustn't go wrong`() {
    val totaller = Totaller()
...
```

Выглядит необычно, но это допустимое имя функции Kotlin.

В основном JUnit используется в Kotlin почти так же, как в проектах Java. Но если вы предпочитаете что-то в большей степени ориентированное на Kotlin, к вашим услугам другая библиотека — KotlinTest.

Библиотека KotlinTest

Библиотека **KotlinTest** (*https://github.com/kotlintest/kotlintest*) была разработана для возможности написания более выразительных тестов. Как и JUnit, это сторонняя библиотека, которую необходимо отдельно включить в проект, если вы собираетесь ее использовать.

KotlinTest — очень большая библиотека, которая позволяет писать тесты в разных стилях. Тем не менее один из способов написания версии кода JUnit, описанного выше, в KotlinTest может выглядеть так:

```
import io.kotlintest.shouldBe
import io.kotlintest.specs.StringSpec

class AnotherTotallerTest : StringSpec({
    "should be able to add 3 and 4 - and it mustn't go wrong" {
        val totaller = Totaller()

        totaller.add(3) shouldBe 3
        totaller.add(4) shouldBe 7
        totaller.total shouldBe 7
    }
})
```

Мы используем эти функции из библиотек KotlinTest, поэтому их нужно импортировать.

Функция JUnit test заменяется строкой.

Мы используем shouldBe вместо assertEquals.

Этот тест похож на тест JUnit, который приводился ранее, не считая того, что функция `test` заменена строкой, а вызовы `assertEquals` были переписаны в виде выражений `shouldBe`. Это пример стиля **строковых спецификаций** KotlinTest (или стиля **StringSpec**). KotlinTest поддерживает несколько стилей тестирования, и вы должны выбрать тот стиль, который лучше подходит для вашего кода.

Тем не менее KotlinTest не является простой переработкой JUnit (хотя KotlinTest использует JUnit во внутренней реализации). KotlinTest поддерживает много возможностей, которые упрощают создание тестов и позволяют с меньшим объемом кода делать больше, чем с простой библиотекой Java. Например, можно воспользоваться строками данных (rows), чтобы протестировать код для целого набора данных. Рассмотрим пример.

Тестирование для наборов данных

В следующем примере теста строки данных (rows) используются для суммирования множества разных чисел (изменения выделены жирным шрифтом):

```
import io.kotlintest.data.forall
import io.kotlintest.shouldBe
import io.kotlintest.specs.StringSpec
import io.kotlintest.tables.row

class AnotherTotallerTest : StringSpec({
    "should be able to add 3 and 4 - and it mustn't go wrong" {
        val totaller = Totaller()

        totaller.add(3) shouldBe 3
        totaller.add(4) shouldBe 7
        totaller.total shouldBe 7
    }

    "should be able to add lots of different numbers" {
        forall(
                row(1, 2, 3),
                row(19, 47, 66),
                row(11, 21, 32)
        ) { x, y, expectedTotal ->
            val totaller = Totaller(x)
            totaller.add(y) shouldBe expectedTotal
        }
    }
})
```

Мы используем две дополнительные функции из библиотек KotlinTest.

Второй тест.

Выполняем тест для каждой строки данных.

Значения каждой строки присваиваются переменным x, y и expectedTotal.

Эти две строки выполняются для каждой строки данных.

Библиотека KotlinTest также может использоваться для следующих целей:

- Параллельное выполнение тестов.
- Создание тестов с генерируемыми свойствами.
- Динамическое включение/отключение тестов. Допустим, некоторые тесты должны запускаться только в Linux, а другие — только на компьютерах Mac.
- Объединение тестов в группы.

...и многих, многих других. Если вы собираетесь писать большой объем кода Kotlin, то KotlinTest определенно заслуживает вашего внимания.

Подробнее о KotlinTest можно узнать по адресу *https://github.com/kotlintest/kotlintest*

Топ-10 тем, которые мы не рассмотрели

Но и это еще не все. Осталось еще несколько тем, о которых, как нам кажется, нужно рассказать. Делать вид, что их не существует, было бы неправильно — как, впрочем, и выпускать книгу, которую поднимет разве что культурист. Прежде чем откладывать книгу, ознакомьтесь с этими **лакомыми кусочками**, которые мы оставили напоследок.

1. Пакеты и импортирование

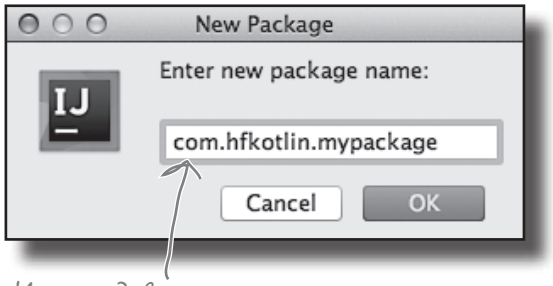
Как упоминалось в главе 9, классы и функции стандартной библиотеки Kotlin группируются в пакеты. Но при этом мы *не* сказали, что вы можете группировать в пакетах *свой собственный* код.

Размещение кода в пакетах полезно по двум причинам:

- **Улучшенная организация кода.**
  Пакеты могут использоваться для объединения кода по конкретным областям функциональности – например, структуры данных или операции с базами данных.
- **Предотвращение конфликтов имен.**
  Если вы написали класс с именем `Duck` и разместили его в пакете, это позволит вам отличить его от любого другого класса с именем `Duck`, добавленного в ваш проект.

Как добавить пакет

Чтобы включить пакет в проект Kotlin, выделите папку *src* и выполните команду File→New→Package. Введите имя пакета (например, *com.hfkotlin.mypackage*) и щелкните OK.



Имя создаваемого пакета.

Объявления пакетов

Когда вы добавляете с пакет файл Kotlin (выделяя имя пакета и выбирая команду File→New→Kotlin File/Class), в начало исходного файла автоматически добавляется объявление **`package`**:

```
package com.hfkotlin.mypackage
```

Объявление `package` сообщает компилятору, что все в этом исходном файле принадлежит этому пакету. Например, следующий код означает, что пакет *com.hfkotlin.mypackage* содержит класс `Duck` и функцию `doStuff`:

```
package com.hfkotlin.mypackage

class Duck
fun doStuff() {
    ...
}
```

Класс Duck и функция doStuff добавляются в пакет com.hfkotlin.mypackage.

Если исходный файл не содержит объявления пакета, код добавляется в анонимный пакет по умолчанию.

Ваш проект может содержать несколько пакетов, и каждый пакет может состоять из нескольких исходных файлов. Тем не менее каждый исходный файл может содержать только одно объявление пакета.

Полное имя

При включении класса в пакет добавляется полное — или *полностью определенное* — имя класса с префиксом в виде имени пакета. Таким образом, если пакет *com.hfkotlin.mypackage* содержит класс с именем `Duck`, то полное имя класса `Duck` имеет вид `com.hfkotlin.mypackage.Duck`. Вы по-прежнему можете обращаться к нему по имени `Duck` в любом коде, принадлежащем тому же пакету, но если вы захотите использовать его в другом пакете, придется указать компилятору его полное имя.

Существуют два способа указания полного имени класса: вы либо используете полное имя во всех местах вашего кода, либо импортируете его.

Укажите полное имя...

Первый способ — указывать полное имя класса каждый раз, когда вы хотите использовать его за пределами своего пакета, например:

```
package com.hfkotlin.myotherpackage

fun main(args: Array<String>) {
    val duck = com.hfkotlin.mypackage.Duck()
    ...
}
```

Другой пакет.

Полное имя.

Однако если вам нужно многократно обращаться к классу или ссылаться на несколько элементов в одном пакете, такой подход может быть неудобным.

...или импортируйте его

Альтернативный способ основан на **импортировании** класса или пакета, чтобы к нему можно было обращаться в классе `Duck` без постоянного ввода его полного имени. Пример:

```
package com.hfkotlin.myotherpackage
import com.hfkotlin.mypackage.Duck

fun main(args: Array<String>) {
    val duck = Duck()
    ...
}
```

Эта строка импортирует класс Duck...

...поэтому к нему можно обращаться без указания полного имени.

Также можно использовать следующий код для импортирования всего пакета:

```
import com.hfkotlin.mypackage.*
```

* означает «импортировать все из этого пакета».

А если возникнет конфликт имен классов, используйте ключевое слово **as**:

```
import com.hfkotlin.mypackage.Duck
import com.hfKotlin.mypackage2.Duck as Duck2
```

К классу Duck из пакета mypackage2 можно обращаться по имени «Duck2».

Импортирование по умолчанию

Следующие пакеты автоматически импортируются в каждый файл Kotlin по умолчанию:

- *kotlin.**
- *kotlin.annotation.**
- *kotlin.collections.**
- *kotlin.comparisons.**
- *kotlin.io.**
- *kotlin.ranges.**
- *kotlin.sequences.**
- *kotlin.text.**

Если вашей целевой платформой является JVM, также импортируются следующие пакеты:

- *java.lang.**
- *kotlin.jvm.**

А если вы выбрали JavaScript, то вместо этого будет импортирован пакет:

- *kotlin.js.**

2. Модификаторы видимости

Модификаторы видимости позволяют определить уровень видимости любого создаваемого кода (например, классов или функций). Например, можно объявить, что класс используется только в коде своего исходного файла или что функция класса используется только внутри своего класса.

В Kotlin определены четыре модификатора видимости: **`public`**, **`private`**, **`protected`** и **`internal`**. Посмотрим, как они работают.

Модификаторы видимости и код верхнего уровня

Как вы уже знаете, классы, переменные и функции могут быть объявлены непосредственно внутри исходного файла или пакета. По умолчанию весь этот код обладает открытым уровнем видимости и может использоваться в любом пакете, в котором он импортируется. Тем не менее это поведение можно изменить — поставьте перед объявлением один из следующих модификаторов видимости:

Помните: если пакет не задан, код по умолчанию автоматически добавляется в пакет без имени.

Модификатор:	Что делает:
public	Объявление видимо в любом месте. Этот уровень действует по умолчанию, поэтому его можно не указывать.
private	Объявление видимо для кода внутри своего исходного файла, но невидимо во всех остальных местах.
internal	Объявление видимо внутри того же модуля, но невидимо во всех остальных местах. Модуль представляет собой набор файлов Kotlin, компилируемых совместно — как, например, модули IntelliJ IDEA.

Обратите внимание: уровень protected недоступен для объявлений на верхнем уровне исходного файла или пакета.

Например, следующий код указывает, что класс `Duck` является открытым (`public`) и виден в любых местах, тогда как функция `doStuff` является приватной (`private`) и видна только внутри своего исходного файла:

```
package com.hfkotlin.mypackage

class Duck

private fun doStuff() {
    println("hello")
}
```

Duck не имеет модификатора видимости, а следовательно, является открытым (public).

Функция doStuff() помечена как приватная, поэтому она может использоваться только внутри того исходного файла, в котором она определяется.

Модификаторы видимости также могут применяться к компонентам классов и интерфейсов. Посмотрим, как они работают.

Модификаторы видимости и классы/интерфейсы

Следующие модификаторы видимости могут применяться к свойствам, функциям и другим компонентам, принадлежащим классам или интерфейсам:

Модификатор:	Что делает:
public	Объявление видимо везде, где видим сам класс. Этот уровень действует по умолчанию, поэтому его можно не указывать.
private	Объявление видимо внутри класса и невидимо в любых других местах.
protected	Объявление видимо внутри класса и любых из его подклассов.
internal	Объявление видимо для любого кода в границах модуля, в котором видим класс.

Пример класса с модификаторами видимости свойств и подкласса, в котором они переопределяются:

```
open class Parent {
    var a = 1
    private var b = 2
    protected open var c = 3
    internal var d = 4
}

class Child: Parent() {
    override var c = 6
}
```

Так как свойство b имеет уровень видимости private, оно может использоваться только внутри этого класса. Оно невидимо для любых субклассов Parent.

Класс Child видит свойства a и c, но он также может обратиться к свойству d, если Parent и Child определяются в одном модуле. Однако для Child свойство b остается невидимым, так как оно имеет модификатор видимости private.

Учтите, что при переопределении компонента с видимостью `protected`, как в приведенном примере, версия этого компонента в подклассе по умолчанию тоже будет иметь видимость `protected`. Тем не менее его видимость можно изменить, как в следующем примере:

```
class Child: Parent() {
    public override var c = 6
}
```

Свойство с видимо везде, где видим класс Child.

По умолчанию конструкторы классов имеют открытый уровень видимости, поэтому они видны везде, где видим сам класс. Тем не менее вы можете изменить видимость конструктора, указав модификатор видимости и поставив перед конструктором ключевое слово `constructor`. Если, например, у вас имеется класс, определенный следующим образом:

```
class MyClass(x: Int)
```

По умолчанию первичный конструктор MyClass имеет уровень видимости public,

то вы можете назначить его конструктору уровень видимости `private` при помощи следующего кода:

```
class MyClass private constructor(x: Int)
```

Этот код делает первичный конструктор приватным.

3. Классы перечислений

Класс перечисления (enum) позволяет создать набор значений, представляющих ***единственно*** допустимые значения переменной.

Допустим, вы хотите создать приложение для музыкальной группы. Нужно удостовериться, что переменной selectedBandMember может быть присвоен только допустимый участник. Для этого можно создать класс перечисления с именем BandMember, содержащий все допустимые значения:

```
enum class BandMember { JERRY, BOBBY, PHIL }
```

Класс перечисления содержит три значения: JERRY, BOBBY и PHIL.

Чтобы после этого ограничить переменную selectedBandMember одним из этих значений, назначьте ей тип BandMember:

```
fun main(args: Array<String>) {
    var selectedBandMember: BandMember
    selectedBandMember = BandMember.JERRY
}
```

Переменная имеет тип BandMember...

...поэтому ей можно присвоить одно из значений из BandMember.

Каждое значение класса перечисления является константой.

Каждая константа перечисления существует как одиночный экземпляр этого класса перечисления.

Конструкторы классов перечислений

Класс перечисления может иметь конструктор, который используется для инициализации каждого значения перечисления. Этот механизм работает, потому что **каждое значение, определенное классом перечисления, является экземпляром этого класса**.

Допустим, вы хотите задать инструмент, на котором играет каждый участник группы. Для этого можно добавить в конструктор BandMember строковую переменную с именем instrument и инициализировать каждое значение в классе подходящим значением. Код выглядит так:

```
enum class BandMember(val instrument: String) {
    JERRY("lead guitar"),
    BOBBY("rhythm guitar"),
    PHIL("bass")
}
```

Определяет свойство с именем instrument в конструкторе BandMember. Каждое значение класса перечисления является экземпляром BandMember, поэтому каждое значение содержит это свойство.

Чтобы после этого узнать, на каком инструменте играет выбранный участник группы, следует обратиться к его свойству instrument:

```
fun main(args: Array<String>) {
    var selectedBandMember: BandMember
    selectedBandMember = BandMember.JERRY
    println(selectedBandMember.instrument)
}
```

Выдает строку «lead guitar».

Свойства и функции перечислений

В предыдущем примере мы добавили свойство в `BandMember`, включив его в конструктор класса перечисления. Свойства и функции также могут добавляться в основное тело класса. Например, следующий код добавляет функцию `sings` в класс перечисления `BandMember`:

```
enum class BandMember(val instrument: String) {
    JERRY("lead guitar"),
    BOBBY("rhythm guitar"),
    PHIL("bass");

    fun sings() = "occasionally"
}
```

Обратите внимание: функция sings() должна отделяться от значений перечисления символом «;».

У каждого значения перечисления имеется функция с именем sings(), которая возвращает строку «occasionally».

Каждое значение, определенное в классе перечисления, может определять свойства и функции, унаследованные от определения класса. В следующем примере функция `sings` определяется для `JERRY` и `BOBBY`:

```
enum class BandMember(val instrument: String) {
    JERRY("lead guitar") {
        override fun sings() = "plaintively"
    },
    BOBBY("rhythm guitar") {
        override fun sings() = "hoarsely"
    },
    PHIL("bass");

    open fun sings() = "occasionally"
}
```

JERRY и BOBBY имеют собственную реализацию sings().

Так как мы переопределяем функцию sings() для двух значений, ее необходимо пометить префиксом open.

После этого вы сможете вызвать функцию `sings` для выбранного участника группы:

```
fun main(args: Array<String>) {
    var selectedBandMember: BandMember
    selectedBandMember = BandMember.JERRY
    println(selectedBandMember.instrument)
    println(selectedBandMember.sings())
}
```

Эта строка вызывает функцию sings() участника JERRY и выдает результат «plaintively».

4. Изолированные классы

Как упоминалось ранее, классы перечислений позволяют создать ограниченный набор фиксированных значений, но в некоторых ситуациях желательно иметь чуть больше гибкости.

Предположим, вы хотите использовать в приложении два разных типа сообщений: для успеха и для неудачи. Все сообщения должны ограничиваться этими двумя типами.

Если вы захотите смоделировать эту конфигурацию с классами перечислений, код может выглядеть так:

```
enum class MessageType(var msg: String) {
    SUCCESS("Yay!"),
    FAILURE("Boo!")
}
```

Класс перечисления MessageType содержит два значения: SUCCESS и FAILURE.

Тем не менее у такого решения есть пара недостатков:

- **Каждое значение — константа, которая существует в единственном экземпляре.** Скажем, вы не сможете изменить свойство `msg` значения `SUCCESS` в одной конкретной ситуации, так как это изменение отразится во всем коде вашего приложения.
- **Каждое значение должно содержать одинаковый набор свойств и функций.** Например, в значение `FAILURE` будет удобно добавить свойство `Exception`, чтобы вы могли проанализировать, что именно пошло не так, но класс перечисления не даст такой возможности.

Что же делать?

На помощь приходят изолированные классы!

Для решения подобных проблем используются **изолированные классы**. Изолированный (sealed) класс напоминает расширенную версию класса перечисления. Он позволяет ограничить иерархию классов конкретным множеством подтипов, каждый из которых может определять собственные свойства и функции. Кроме того, в отличие от класса перечисления, вы можете создать несколько экземпляров каждого типа.

Чтобы создать изолированный класс, поставьте перед именем класса префикс **`sealed`**. Например, следующий фрагмент создает изолированный класс с именем `MessageType`, с двумя подтипами — `MessageSuccess` и `MessageFailure`. У каждого подтипа имеется строковое свойство `msg`, а подтип `MessageFailure` содержит дополнительное свойство `Exception` с именем `e`:

```
sealed class MessageType
class MessageSuccess(var msg: String) : MessageType()
class MessageFailure(var msg: String, var e: Exception) : MessageType()
```

Класс MessageType является изолированным.

MessageSuccess и MessageFailure наследуются от MessageType и определяют собственные свойства в своих конструкторах.

Как использовать изолированные классы

Как мы уже говорили, изолированный класс позволяет создать несколько экземпляров каждого подтипа. Например, следующий код создает два экземпляра `MessageSuccess` и один экземпляр `MessageFailure`:

```
fun main(args: Array<String>) {
    val messageSuccess = MessageSuccess("Yay!")
    val messageSuccess2 = MessageSuccess("It worked!")
    val messageFailure = MessageFailure("Boo!", Exception("Gone wrong."))
}
```

После этого вы можете создать переменную `MessageType` и присвоить ей одно из сообщений:

```
fun main(args: Array<String>) {
    val messageSuccess = MessageSuccess("Yay!")
    val messageSuccess2 = MessageSuccess("It worked!")
    val messageFailure = MessageFailure("Boo!", Exception("Gone wrong."))

    var myMessageType: MessageType = messageFailure
}
```

messageFailure является подклассом MessageType, поэтому его экземпляр можно присвоить myMessageType.

А поскольку `MessageType` является изолированным классом с ограниченным набором подтипов, для проверки всех подтипов можно воспользоваться конструкцией `when` без дополнительной секции `else`:

```
fun main(args: Array<String>) {
    val messageSuccess = MessageSuccess("Yay!")
    val messageSuccess2 = MessageSuccess("It worked!")
    val messageFailure = MessageFailure("Boo!", Exception("Gone wrong."))

    var myMessageType: MessageType = messageFailure
    val myMessage = when (myMessageType) {
        is MessageSuccess -> myMessageType.msg
        is MessageFailure -> myMessageType.msg + " " + myMessageType.e.message
    }
    println(myMessage)
}
```

myMessageType может относиться только к типу MessageSuccess или MessageFailure, поэтому в дополнительной секции else нет необходимости.

За дополнительной информацией о создании и использовании изолированных классов обращайтесь по адресу *https://kotlinlang.org/docs/reference/sealed-classes.html*

5. Вложенные и внутренние классы

Вложенный класс определяется внутри другого класса. Это может быть полезно, если вы хотите наделить внешний класс расширенной функциональностью, выходящей за пределы его основного предназначения, или определить код ближе к месту его использования.

Вложенные классы в Kotlin похожи на статические вложенные классы в языке Java.

Чтобы определить вложенный класс, заключите его в фигурные скобки внутри внешнего класса. Например, следующий код определяет класс с именем `Outer`, внутри которого находится вложенный класс с именем `Nested`:

```
class Outer {
    val x = "This is in the Outer class"

    class Nested {
        val y = "This is in the Nested class"
        fun myFun() = "This is the Nested function"
    }
}
```

Вложенный класс размещается полностью внутри внешнего класса.

После этого вы можете обращаться к классу `Nested` и его свойствам и функциям следующим образом:

```
fun main(args: Array<String>) {
    val nested = Outer.Nested()
    println(nested.y)
    println(nested.myFun())
}
```

Создает экземпляр Nested и присваивает его переменной.

Учтите, что вы не сможете обратиться к вложенному классу из экземпляра внешнего класса без предварительного создания свойства этого типа внутри внешнего класса. Например, следующий код компилироваться не будет:

```
val nested = Outer().Nested()
```

Не компилируется, так как мы используем Outer(), а не Outer.

Другое ограничение заключается в том, что экземпляр внешнего класса недоступен для вложенного класса, поэтому вложенный класс не сможет обращаться к его компонентам. Например, вы не сможете обратиться к свойству `x` класса `Outer` из класса `Nested`, поэтому следующая строка не компилируется:

```
class Outer {
    val x = "This is in the Outer class"

    class Nested {
        fun getX() = "Value of x is: $x"
    }
}
```

Вложенный класс не видит x, так как переменная определяется в классе Outer, поэтому эта строка не компилируется.

Внутренний класс может обращаться к компонентам внешнего класса

Если вы хотите, чтобы вложенный класс мог обращаться к свойствам и функциям, определенным его внешним классом, преобразуйте его во **внутренний класс**. Для этого вложенный класс помечается префиксом **`inner`**. Пример:

```
class Outer {
    val x = "This is in the Outer class"

    inner class Inner {
        val y = "This is in the Inner class"
        fun myFun() = "This is the Inner function"
        fun getX() = "The value of x is: $x"
    }
}
```

Внутренний класс представляет собой вложенный класс, который может обращаться к компонентам внешнего класса. Таким образом, в этом примере класс Inner может обратиться к свойству x класса Outer.

Чтобы обратиться к внутреннему классу, создайте экземпляр внешнего класса, а затем на его основе создайте экземпляр внутреннего класса. В следующем примере используются классы `Outer` и `Inner`, определенные выше:

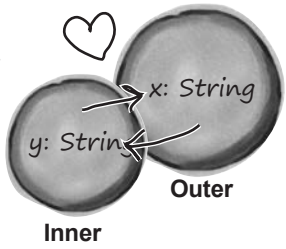
```
fun main(args: Array<String>) {
    val inner = Outer().Inner()
    println(inner.y)
    println(inner.myFun())
    println(inner.getX())
}
```

Inner — внутренний класс, поэтому необходимо использовать Outer(), а не Outer.

Также можно обратиться к внутреннему классу, создав экземпляр свойства этого типа во внешнем классе, как в следующем примере:

```
class Outer {
    val myInner = Inner()

    inner class Inner {
        ...
    }
}

fun main(args: Array<String>) {
    val inner = Outer().myInner
}
```

Свойство myInner класса Outer содержит ссылку на экземпляр его класса Inner.



Объекты Inner и Outer имеют особую связь. Inner может использовать переменные Outer, и наоборот.

Ключевой момент заключается в том, что экземпляр внутреннего класса *всегда* связывается с конкретным экземпляром внешнего класса, поэтому вы не сможете создать объект `Inner` без предварительного создания объекта `Outer`.

6. Объявления объектов и выражения

В некоторых случаях нужна гарантия, что может быть создан только один экземпляр заданного типа, например, когда один объект должен координировать некие действия в масштабах приложения. В таких ситуациях можно воспользоваться ключевым словом **`object`** для создания **объявления объекта**.

Если вы знакомы с паттернами проектирования, объявление объекта является аналогом паттерна «Одиночный объект» («Синглтон») в Kotlin.

Объявление объекта определяет объявление класса и создает его экземпляр в одной команде. А когда оно включается на верхнем уровне исходного файла или пакета, может быть создан только один экземпляр этого типа.

Объявление объекта выглядит примерно так:

```
package com.hfkotlin.mypackage

object DuckManager {
    val allDucks = mutableListOf<Duck>()

    fun herdDucks() {
        //Код для работы с объектами Duck
    }
}
```

DuckManager — объект.

Содержит свойство с именем allDucks и функцию с именем herdDucks().

Объявление объекта определяет класс и создает его экземпляр в одной команде.

Как видите, объявление объекта похоже на определение класса, если не считать того, что перед ним ставится префикс `object`, а не `class`. Как и класс, объект может обладать свойствами, функциями и блоками инициализации, может наследоваться от классов и интерфейсов. Впрочем, в объявление объекта нельзя добавить конструктор. Дело в том, что объект автоматически создается сразу же после обращения, поэтому конструктор будет лишним.

К объекту, созданному с использованием объявления объекта, вы обращаетесь по имени; через него также можно обращаться к компонентам объекта. Например, если вы хотите вызвать функцию `herdDucks` объекта `DuckManager`, это можно сделать так:

```
DuckManager.herdDucks()
```

Наряду с добавлением объявления объекта на верхнем уровне исходного файла или пакета, вы также можете добавить его в класс. Давайте посмотрим, как это делается.

Объекты классов...

В следующем коде объявление объекта `DuckFactory` добавляется в класс с именем `Duck`:

```
class Duck {
    object DuckFactory {
        fun create(): Duck = Duck()
    }
}
```

Объявление объекта размещается в основном теле класса.

Добавьте объявление объекта в класс, чтобы создать одиночный экземпляр этого типа, принадлежащий классу.

При добавлении объявления объекта в класс создается объект, принадлежащий этому классу. Для класса создается один экземпляр объекта, совместно используемый всеми экземплярами этого класса.

Добавив объявление объекта, вы можете обращаться к объекту из класса с использованием точечной записи. Например, следующий код вызывает функцию `create` объекта `DuckFactory` и присваивает результат новой переменной с именем `newDuck`:

```
val newDuck = Duck.DuckFactory.create()
```

Обратите внимание: для обращения к объекту используется запись Duck, а не Duck().

...и объекты-компаньоны

Один объект для класса может быть помечен как объект-**компаньон** при помощи префикса **`companion`**. Объект-компаньон похож на объект класса, не считая того, что его имя можно не указывать. Например, следующий код превращает объект `DuckFactory` из приведенного выше примера в безымянный объект-компаньон:

```
class Duck {
    companion object DuckFactory {
        fun create(): Duck = Duck()
    }
}
```

Если поставить перед объявлением объекта префикс companion, вам уже не придется указывать имя объекта. Впрочем, при желании имя можно указать.

Если вы создали объект-компаньон, для обращения к нему достаточно указать имя класса. Например, следующий код вызывает функцию `create()`, определенную объектом-компаньоном `Duck`:

```
val newDuck = Duck.create()
```

Для получения ссылки на безымянный объект-компаньон используется ключевое слово `Companion`. Например, следующий код создает новую переменную с именем `x` и присваивает ей ссылку на объект-компаньон класса `Duck`:

```
val x = Duck.Companion
```

После знакомства с объявлениями объектов и объектами-компаньонами обратимся к объектным выражениям.

Объект-компаньон может использоваться как Kotlin-аналог статических методов в языке Java. Любые функции, добавленные в объект-компаньон, совместно используются всеми экземплярами класса.

Объектные выражения

Объектным выражением называется выражение, которое «на ходу» создает анонимный объект без заранее определенного типа.

Допустим, вы хотите создать объект для хранения исходного значения координат x и y. Чтобы не определять класс `Coordinate` и не создавать экземпляр этого класса, вы можете создать объект, использующий свойства для хранения значений координат x и y. Например, следующий код создает новую переменную с именем `startingPoint` и присваивает ей такой объект:

```
val startingPoint = object {
    val x = 0
    val y = 0
}
```

Создает объект с двумя свойствами, x и y.

После этого для обращения к компонентам объекта можно использовать следующую запись:

```
println("starting point is ${startingPoint.x}, ${startingPoint.y}")
```

Объектные выражения обычно используются как аналоги анонимных внутренних классов языка Java. Если вы пишете некий GUI-код и вдруг понимаете, что вам нужен экземпляр класса, реализующий абстрактный класс `MouseAdapter`, используйте объектные выражения для создания экземпляра на ходу. Например, следующий код передает объект функции с именем `addMouseListener`; объект реализует интерфейс `MouseAdapter` и переопределяет его функции `mouseClicked` и `mouseEntered`:

Эта команда...

```
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        //Код, выполняемый при щелчке кнопкой мыши
    }

    override fun mouseReleased(e: MouseEvent) {
        //Код, выполняемый при отпускании кнопки мыши
    }
})
```

...заканчивается здесь.

Объектное выражение, выделенное жирным шрифтом, фактически означает «создать экземпляр класса (без имени), который реализует MouseAdapter — а вот его реализации функций mouseClicked и mouseReleased». Другими словами, мы предоставляем функции addMouseListener реализацию класса и экземпляр этого класса именно там, где он нужен.

Дополнительную информацию об объявлениях объектов и выражениях см. здесь:

https://kotlinlang.org/docs/reference/object-declarations.html

7. Расширения

Расширения позволяют добавлять новые функции и свойства в существующий тип без создания нового подтипа.

Также существуют библиотеки расширения Kotlin, которые могут использоваться для упрощения программирования, — такие, как Anko и Android KTX для разработки приложений Android.

Представьте, что вы пишете приложение, в котором часто приходится снабжать значение `Double` префиксом $, чтобы отформатировать его в виде денежной суммы. Чтобы не повторять одно и то же действие снова и снова, можно написать функцию расширения с именем `toDollar`, которая может использоваться с `Double`. Эта задача решается так:

Определяет функцию с именем toDollar(), которая расширяет Double.

Вернуть текущее значение с префиксом $.

```
fun Double.toDollar(): String {
    return "$$this"
}
```

Приведенный выше код указывает, что функция с именем `toDollar`, которая возвращает строку, может использоваться со значениями `Double`. Функция получает текущий объект (обозначаемый `this`), ставит перед ним префикс $ и возвращает результат.

Созданная функция расширения используется так же, как любая другая функция. Например, в следующем фрагменте функция `toDollar` вызывается для переменной `Double` со значением 45.25:

```
var dbl = 45.25
println(dbl.toDollar())     //Выводит $45.25
```

Расширения свойств создаются так же, как расширения функций. В следующем коде создается свойство расширения для `String` с именем `halfLength`, которое возвращает длину текущей строки, разделенную на 2.0:

Определяет свойство halfLength, которое может использоваться со строками.

```
val String.halfLength
    get() = length / 2.0
```

Пример кода, в котором используется новое свойство:

```
val test = "This is a test"
println(test.halfLength)     //Выводит 7.0
```

Дополнительная информация об использовании расширений, в том числе и о добавлении их в объекты-компаньоны, доступна по адресу *https://kotlinlang.org/docs/reference/extensions.html*

Об использовании `this` можно подробнее узнать здесь:

https://kotlinlang.org/docs/reference/this-expressions.html

Паттерны проектирования

Паттерны проектирования представляют собой универсальные решения типичных задач. Kotlin предоставляет простые и удобные средства для реализации некоторых из этих паттернов.

Объявления объектов предоставляют возможность реализации паттерна «Одиночный объект», так как каждое объявление создает одиночный экземпляр этого объекта. **Расширения** могут использоваться вместо паттерна «Декоратор», поскольку они позволяют расширять поведение классов и объектов. А если вас интересует паттерн «Делегирование» как альтернатива наследованию, обращайтесь за дополнительной информацией по адресу

https://kotlinlang.org/docs/reference/delegation.html

8. Return, break и continue

В Kotlin предусмотрены три способа выхода из цикла:

- **return**
  Как вы уже знаете, эта команда осуществляет возврат из включающей функции.

- **break**
  Команда завершает вмещающий цикл (осуществляет переход к точке после его завершения), например:

```
var x = 0
var y = 0
while (x < 10) {
    x++
    break
    y++
}
```

Этот код увеличивает x, после чего завершает цикл без выполнения строки y++. x имеет итоговое значение 1, а значение y остается равным 0.

- **continue**
  Команда переходит к следующей итерации цикла, например:

```
var x = 0
var y = 0
while (x < 10) {
    x++
    continue
    y++
}
```

Увеличивает x, после чего возвращается к строке while (x < 10) без выполнения строки y++. Переменная x продолжает увеличиваться, пока условие (x < 10) не станет равным false. Итоговое значение x равно 10, а значение y остается равным 0.

Использование меток с break и continue

Если вы используете вложенные циклы, можно явно указать, из какого цикла вы хотите выйти, — для этого следует указать **метку**. Метка состоит из имени, за которым следует символ `@`. Например, в следующем коде представлены два цикла, один из которых вложен в другой. Внешний цикл имеет метку с именем `myloop@`, которая используется в выражении `break`:

```
myloop@ while (x < 20) {
    while (y < 20) {
        x++
        break@myloop
    }
}
```

Означает «выйти из цикла с меткой myloop@ (внешнего цикла)».

При использовании `break` с меткой команда осуществляет переход к концу вмещающего цикла с указанной меткой, так что в приведенном примере будет завершен внешний цикл. При использовании `continue` с меткой происходит переход к следующей итерации этого цикла.

Использование меток с return

При помощи меток также можно управлять поведением кода во вложенных функциях, включая функции высшего порядка.

Предположим, имеется следующая функция, которая включает вызов `forEach` — встроенной функции высшего порядка, получающей лямбда-выражение:

```
fun myFun() {
    listOf("A", "B", "C", "D").forEach {
        if (it == "C") return
        println(it)
    }
    println("Finished myFun()")
}
```

Здесь return используется внутри лямбда-выражения. При достижении return происходит выход из функции myFun().

В этом примере программа выходит из функции `myFun` при достижении выражения `return`, поэтому строка

```
println("Finished myFun()")
```

никогда не выполняется.

Если вы хотите выйти из лямбда-выражения, но при этом продолжить выполнение `myFun`, добавьте метку в лямбда-выражение и укажите ее в `return`. Пример:

```
fun myFun() {
    listOf("A", "B", "C", "D").forEach myloop@{
        if (it == "C") return@myloop
        println(it)
    }
    println("Finished myFun()")
}
```

Лямбда-выражение, которое передается функции forEach, снабжено меткой myloop@. В выражении return лямбда-выражения используется эта метка, поэтому при его достижении происходит выход из лямбда-выражения, а управление передается на сторону вызова (цикл forEach).

Это можно заменить **неявной** меткой, имя которой соответствует функции, которой было передано лямбда-выражение:

```
fun myFun() {
    listOf("A", "B", "C", "D").forEach {
        if (it == "C") return@forEach
        println(it)
    }
    println("Finished myFun()")
}
```

Здесь при помощи неявной метки мы говорим программе выйти из лямбда-выражения и вернуться на сторону его вызова (цикл forEach).

За дополнительной информацией об использовании меток для передачи управления в программе обращайтесь по адресу

https://kotlinlang.org/docs/reference/returns.html

9. Дополнительные возможности функций

О функциях в этой книге было сказано достаточно много, но у них есть еще несколько возможностей, о которых, как нам кажется, вам стоило бы знать.

vararg

Если вы хотите, чтобы функция получала несколько аргументов одного типа, но точное количество этих аргументов неизвестно, поставьте перед этим параметром префикс **vararg**. Он сообщает компилятору, что параметр может получать переменное количество аргументов. Пример:

Только один параметр может быть помечен ключевым словом vararg. Этот параметр обычно стоит на последнем месте.

Префикс vararg означает, что в параметре ints могут передаваться несколько значений.

```
fun <T> valuesToList(vararg vals: T): MutableList<T> {
    val list: MutableList<T> = mutableListOf()
    for (i in vals) {
        list.add(i)
    }
    return list
}
```

Значения vararg передаются функции в виде массива, так что мы можем перебрать все значения. В следующем фрагменте каждое значение добавляется в MutableList.

При вызове функции с передачей `vararg` ей передаются значения — точно так же, как при вызове любых других функций. Например, в следующем примере функции `valuesToList` передаются пять значений `Int`:

```
val mList = valuesToList(1, 2, 3, 4, 5)
```

Если у вас имеется готовый массив значений, вы можете передать их функции, поставив перед именем массива префикс `*` — так называемый **оператор распространения**. Пара примеров его использования:

```
val myArray = arrayOf(1, 2, 3, 4, 5)
val mList = valuesToList(*myArray)
val mList2 = valuesToList(0, *myArray, 6, 7)
```

Значения, содержащиеся в myArray, передаются функции valuesToList.

функции передается 0...

...за ним следует содержимое myArray...

...и наконец, 6 и 7.

infix

Если поставить перед функцией префикс **infix**, ее можно будет вызывать без использования точечной записи. Пример функции с префиксом `infix`:

Помечаем функцию bark() префиксом infix.

```
class Dog {
    infix fun bark(x: Int): String {
        //Код, выполняющий операцию bark для Dog x раз
    }
}
```

Так как функция была помечена ключевым словом `infix`, ее вызов может выглядеть так:

```
Dog() bark 6
```

Создает экземпляр Dog и вызывает его функцию bark(), передавая функции значение 6.

Функция может быть помечена ключевым словом `infix`, если она является функцией класса или функцией расширения, имеет один параметр без значения по умолчанию и не имеет пометки `vararg`.

inline

Функции высшего порядка иногда работают медленнее обычных, но достаточно часто их быстродействие можно повысить, поставив перед ними префикс **inline**, как в следующем примере:

```
inline fun convert(x: Double, converter: (Double) -> Double) : Double {
    val result = converter(x)
    println("$x is converted to $result")
    return result
}
```

Перед вами функция, которая была создана в главе 11, но здесь она помечена префиксом inline.

Когда функция объявляется подобным образом, в сгенерированном коде вызов функции заменяется ее непосредственным содержимым. При этом устраняются дополнительные затраты на вызов функции, что часто ускоряет выполнение функции, но во внутренней реализации при этом генерируется больший объем кода.

За дополнительной информацией об этих (и многих других) возможностях обращайтесь по адресу *https://kotlinlang.org/docs/reference/functions.html*

10. Совместимость

Как было сказано в начале книги, код Kotlin совместим с Java и может транспилироваться в JavaScript. Если вы собираетесь использовать свой код Kotlin с другими языками, рекомендуем ознакомиться в электронной документации Kotlin с разделами, посвященными совместимости.

Совместимость с Java

Практически любой код Java может вызваться в Kotlin без малейших проблем. Просто импортируйте любые библиотеки, которые не были импортированы автоматически, и используйте их. Обо всех дополнительных факторах — а также о том, как Kotlin работает со значениями null, получаемыми из Java, — можно прочитать здесь:

https://kotlinlang.org/docs/reference/java-interop.html

Использование кода Kotlin из Java рассматривается здесь:

https://kotlinlang.org/docs/reference/java-to-kotlin-interop.html

Использование Kotlin с JavaScript

Электронная документация также включает разнообразную информацию об использовании Kotlin с JavaScript. Например, если ваше приложение ориентировано на JavaScript, вы можете воспользоваться типом Kotlin `dynamic`, который фактически отключает проверку типов Kotlin:

```
val myDynamicVariable: dynamic = ...
```

О типе `dynamic` более подробно рассказано здесь:

https://kotlinlang.org/docs/reference/dynamic-type.html

Следующая страница содержит информацию об использовании JavaScript из Kotlin:

https://kotlinlang.org/docs/reference/js-interop.html

Наконец, информацию об обращении к коду Kotlin из JavaScript смотрите здесь:

https://kotlinlang.org/docs/reference/js-to-kotlin-interop.html

Написание платформенного кода на Kotlin

Режим Kotlin/Native позволяет компилировать код Kotlin в платформенные двоичные файлы. Если вам захочется больше узнать об этой возможности, обращайтесь по следующему адресу:

https://kotlinlang.org/docs/reference/native-overview.html

Если вы хотите иметь возможность использовать свой код на нескольких целевых платформах, мы рекомендуем ознакомиться с поддержкой многоплатформенных проектов в Kotlin. За дополнительной информацией о многоплатформенных проектах обращайтесь по адресу

https://kotlinlang.org/docs/reference/multiplatform.html