

Содержание (сводка)

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

Первые Шаги

1

Не теряя времени

Kotlin впечатляет. С выхода самой первой версии Kotlin впечатляет программистов своим *удобным синтаксисом, компактностью, гибкостью и мощностью.*

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

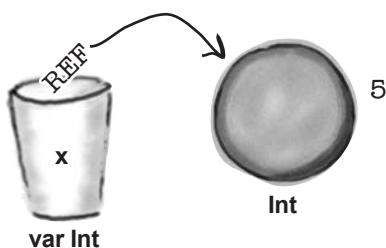
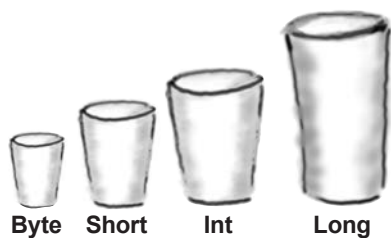


Добро пожаловать в Kotlinville	32
Kotlin может использоваться практически везде	33
Чем мы займемся в этой главе	34
Установка IntelliJ IDEA (Community Edition)	37
Построение простейшего приложения	38
Вы только что создали свой первый проект Kotlin	41
Включение файла Kotlin в проект	42
Анатомия функции 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

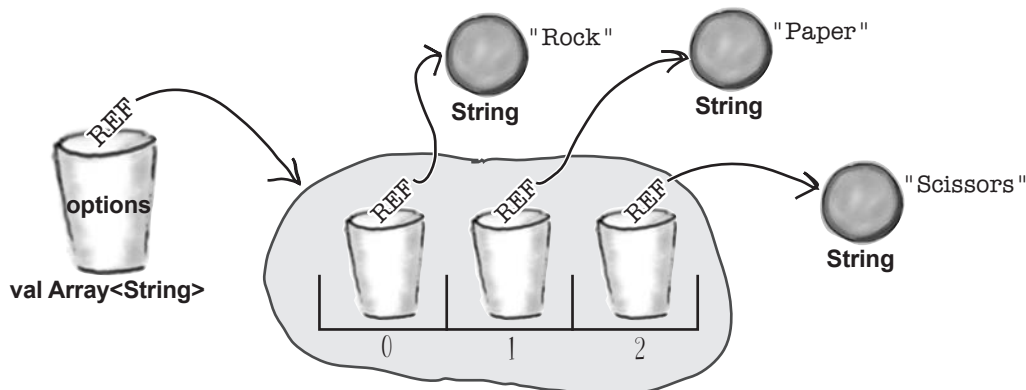
Функции

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 <code>getGameChoice</code> в файл <code>Game.kt</code>	98
Функция <code>getUserChoice</code>	105
Как работают циклы <code>for</code>	106
Запрос выбора пользователя	108
Проверка пользовательского ввода	111
Добавление функции <code>getUserChoice</code> в файл <code>Game.kt</code>	113
Добавление функции <code>printResult</code> в файл <code>Game.kt</code>	117
Ваш инструментарий Kotlin	119



Классы и объекты

4

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.

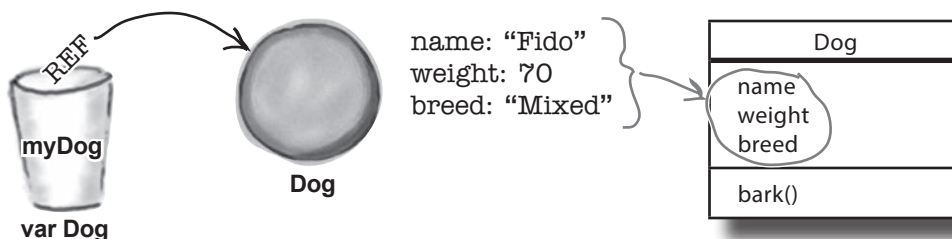
Один класс

Dog
name weight breed
bark()

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

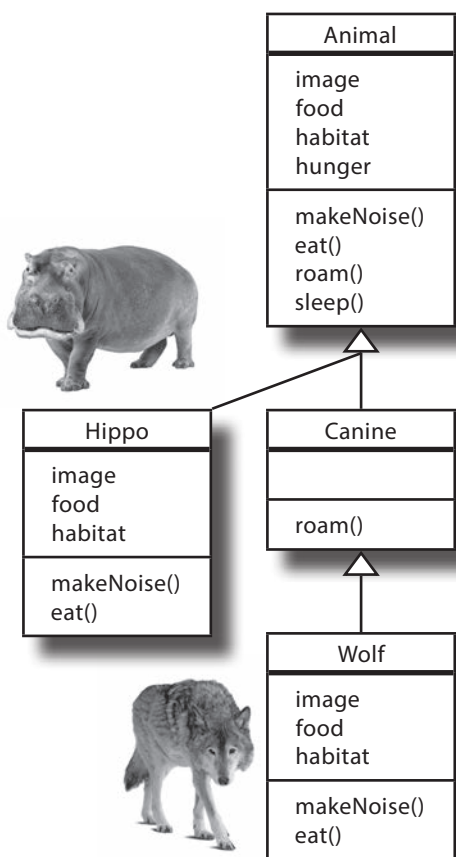


5 Подклассы и суперклассы

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**.

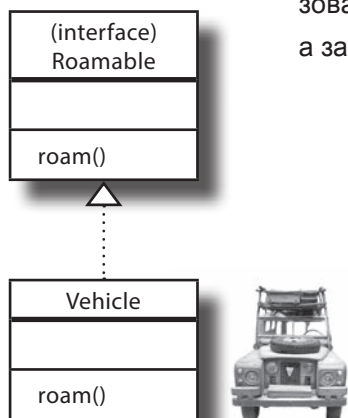


Наследование предотвращает дублирование кода	152
Что мы собираемся сделать	153
Проектирование структуры наследования классов	154
Используйте наследование для предотвращения дублирования кода в подклассах	155
Что должны переопределять подклассы?	156
Некоторых животных можно сгруппировать	157
Добавление 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

6 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*.



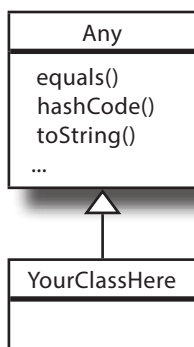
Снова об иерархии классов Animal	186
Некоторые классы не подходят для создания экземпляров	187
Абстрактный или конкретный?	188
Абстрактный класс может содержать абстрактные свойства и функции	189
Класс Animal содержит две абстрактные функции	190
Как реализовать абстрактный класс	192
Вы должны реализовать 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

Классы данных

1

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	242

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.



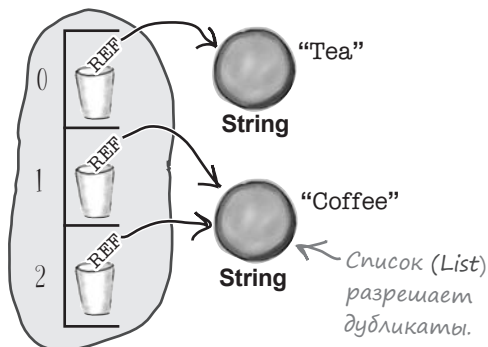
Как удалить ссылку на объект из переменной?	250
Удаление ссылки на объект с использованием null	251
null-совместимые типы могут использоваться везде, где могут использоваться не-null-совместимые	252
Как создать массив null-совместимых типов	253
Как обращаться к функциям и свойствам null-совместимых типов	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

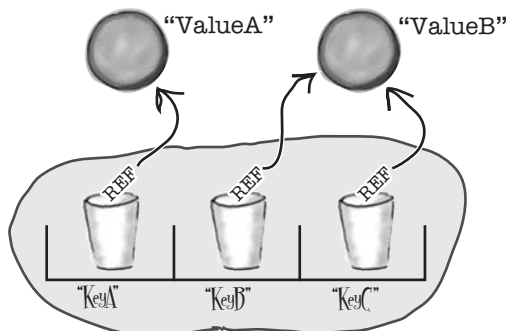
Коллекции

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.



List



Map

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

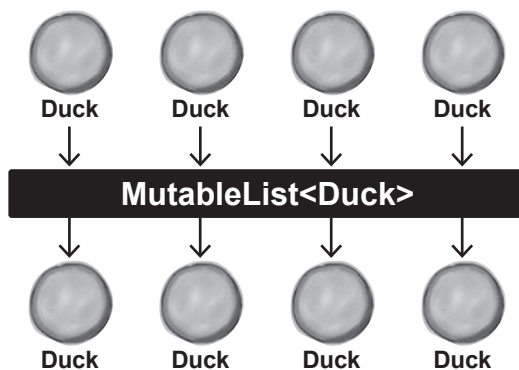
10

Обобщения

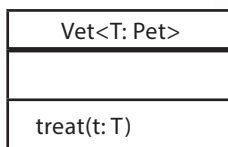
На каждый вход знай свой выход

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 масло. 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.

С обобщениями на ВХОД
поступают только ссылки
на объекты Duck...



...и на ВЫХОДЕ они
остаются ссылками
с типом Duck.



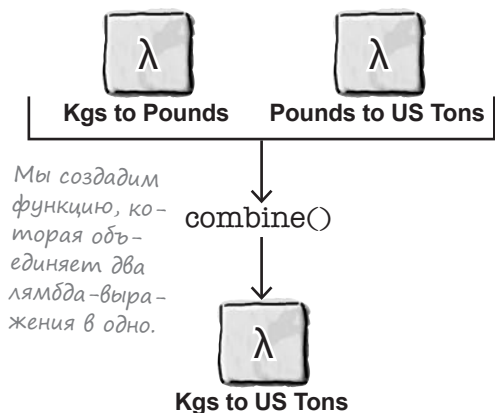
В коллекциях используются обобщения	320
Как определяется MutableList	321
Использование параметров типа с MutableList	322
Что можно делать с обобщенным классом или интерфейсом	323
Что мы собираемся сделать	324
Создание иерархии классов Pet	325
Определение класса Contest	326
Добавление свойства scores	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**.



Я получаю один параметр `Int` с именем `x`. Я прибавляю 5 к `x` и возвращаю результат.



```
{ x: Int, y: Int -> x + y }
```

Lambda

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

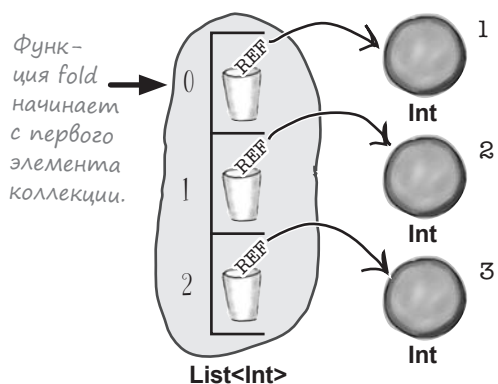
12

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.

У этих предметов не существует естественного порядка. Чтобы найти наименьшее или наибольшее значение, необходимо задать некоторые критерии — например, *unitPrice* или *quantity*.



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

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.



Тестирование

Код под контролем

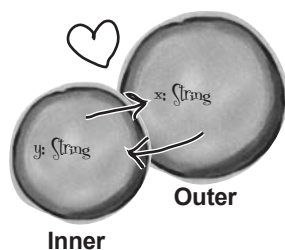
Хороший код должен работать, это все знают. Но при любых изменениях кода появляется опасность внесения новых ошибок, из-за которых код не будет работать так, как положено. Вот почему так важно провести *тщательное тестирование:* вы узнаете о любых проблемах в коде *до того, как он будет развернут в среде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



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.

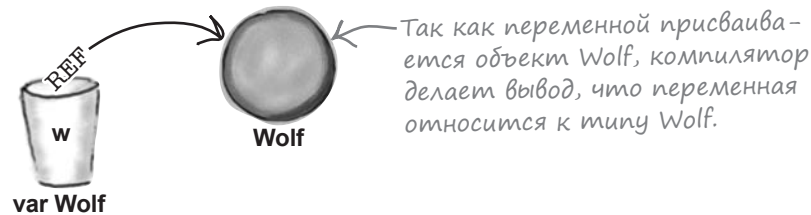
1. Пакеты и импортирование	446
2. Модификаторы видимости	448
3. Классы перечислений	450
4. Изолированные классы	452
5. Вложенные и внутренние классы	454
6. Объявления объектов и выражения	456
7. Расширения	459
8. Return, break и continue	460
9. Дополнительные возможности функций	462
10. Совместимость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

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:

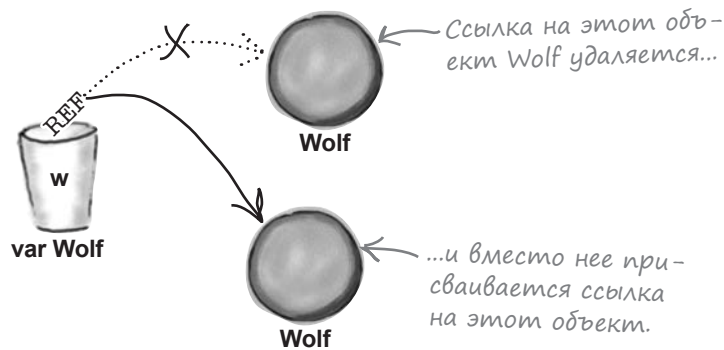
```
var w = Wolf()
```

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.

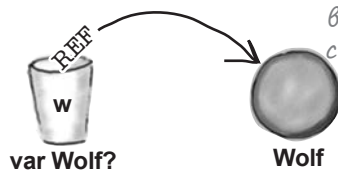
Но тут снова возникает Коварная Ловушка. По умолчанию *типы Kotlin не поддерживают значения null*. Если вам нужна переменная, способная хранить `null`, вы должны явно указать, что ее тип является `null`-совместимым.

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:

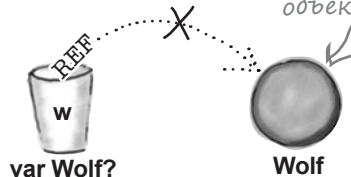
```
var w: Wolf? = Wolf()
```



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.

(Мысли 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`.

Где же используются `null`-совместимые типы?

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`):

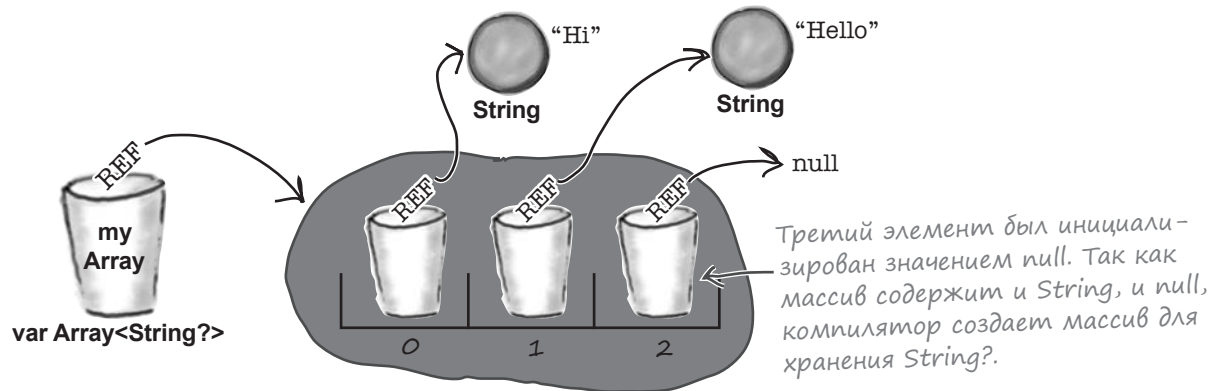
```
var myArray: Array<String?> = arrayOf("Hi", "Hello")
```

← *Array<String?> можем 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?>`:



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 значение `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?
```