Содержание (сводка)

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?

1 Первые шаги

Не теряя времени

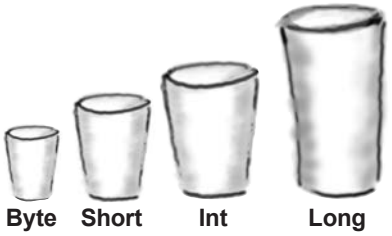
Kotlin впечатляет. С выхода самой первой версии Kotlin впечатляет программистов своим ***удобным синтаксисом, компактностью, гибкостью и мощью***. В этой книге мы научим вас **строить собственные приложения Kotlin**, а для начала покажем, как построить простейшее приложение и запустить его. Попутно вы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

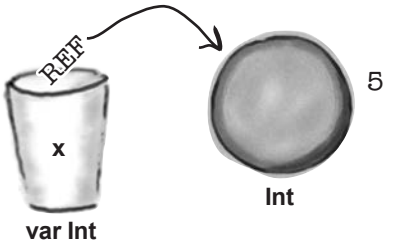
Раз вы можете выбирать платформу, для которой должен компилироваться код, это означает, что ваш код Kotlin может выполняться на серверах, в браузерах, на мобильных устройствах и т. д.

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*.

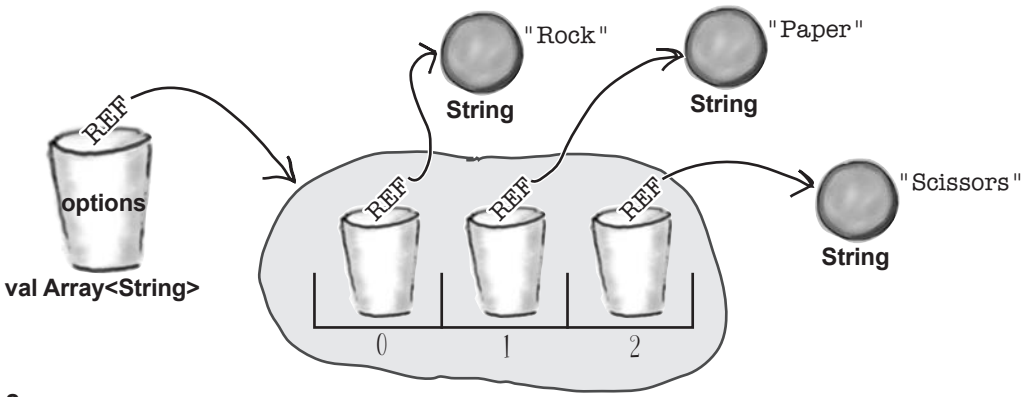




3 Функции

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*.

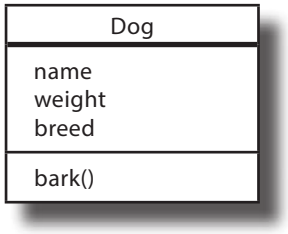


4 Классы и объекты

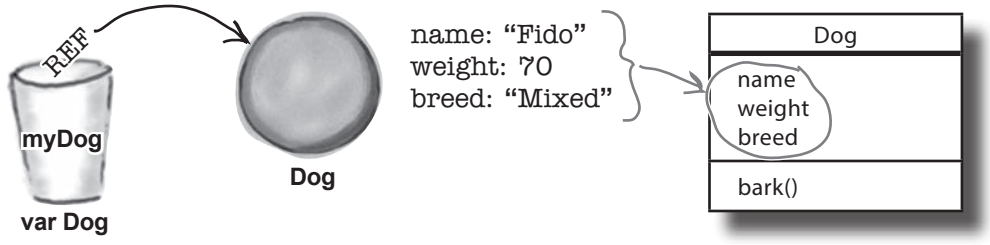
Высокий класс

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

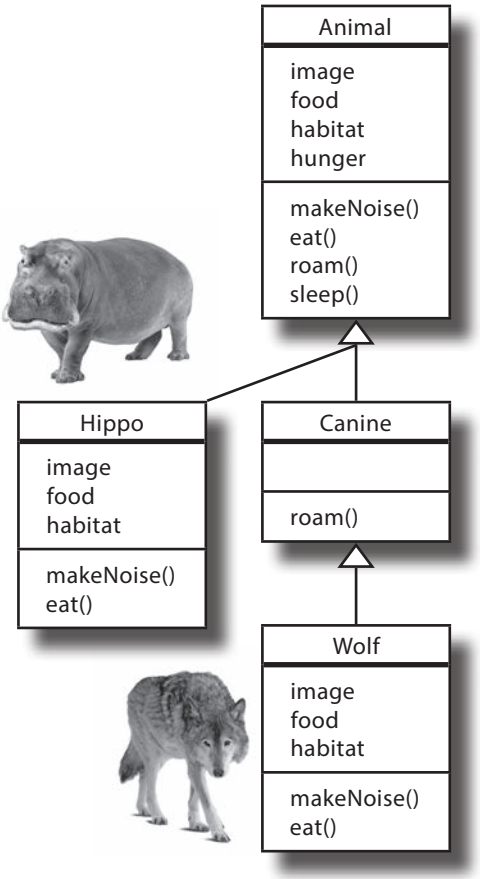


Подклассы и суперклассы

5 Наследование

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

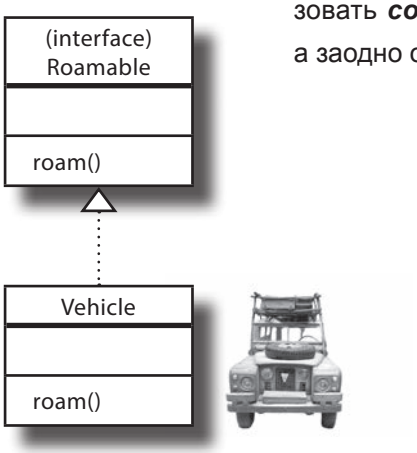
Что ж, это одно из преимуществ **наследования**. В этой главе вы научитесь создавать **подклассы** и наследовать свойства и функции **суперклассов**. Вы узнаете, ***как переопределять функции и свойства,*** чтобы классы работали так, как нужно *вам,* и когда это стоит (или не стоит) делать. Наконец,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



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***.

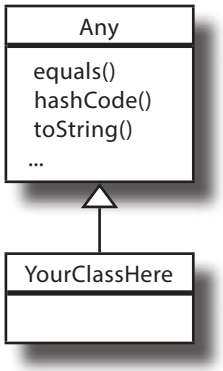


7

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.

8 Null и исключения

В целости и сохранности

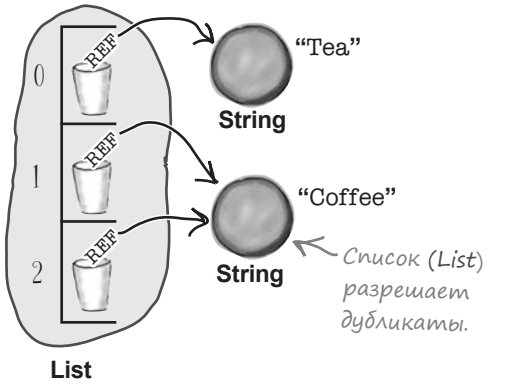
Все мечтают о безопасности кода, и, к счастью, она была заложена в основу языка Kotlin. В этой главе мы сначала покажем, что при использовании **null-совместимых типов** Kotlin вы *вряд ли когда-либо столкнетесь с исключениями NullPointerException за все время программирования на Kotlin*. Вы научитесь 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.

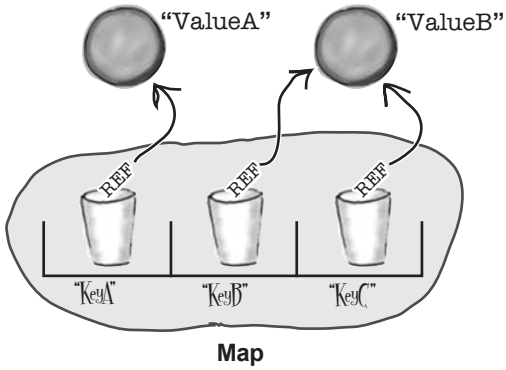
9

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.



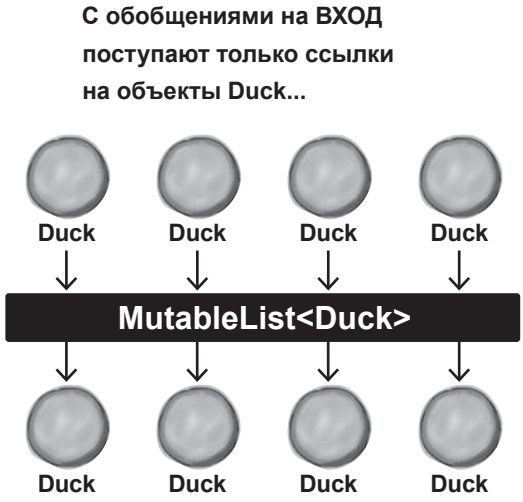


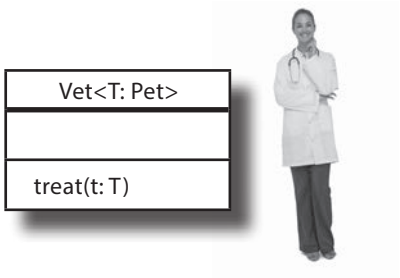
Ассоциативный массив (Map) разрешает дубликаты значений, но не дубликаты ключей.

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.

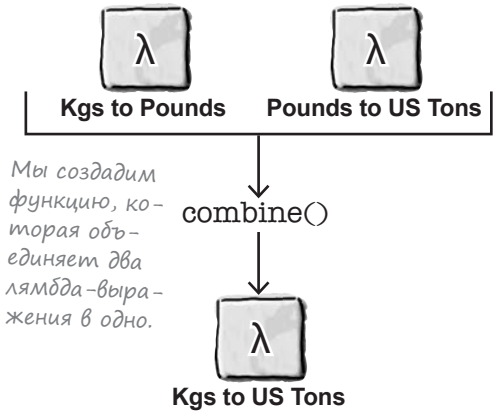


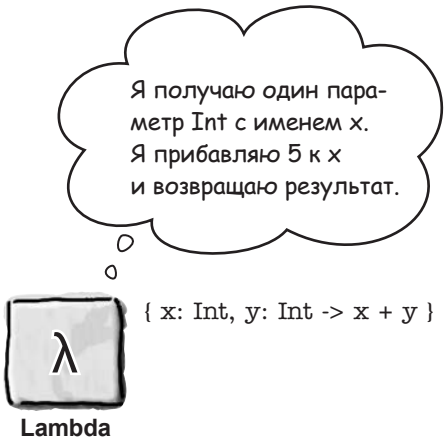


11 Лямбда-выражения и функции высшего порядка

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.***

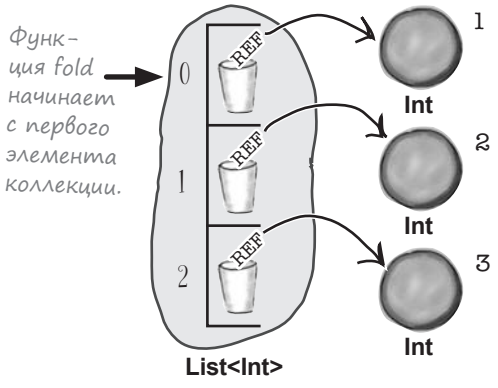




12 Встроенные функции высшего порядка

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.



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

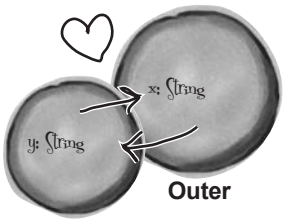
Хороший код должен работать, это все знают. Но при любых изменениях кода появляется опасность внесения новых ошибок, из-за которых код не будет работать так, как положено. Вот почему так важно провести *тщательное тестирование*: вы узнаете о любых проблемах в коде *до того, как он будет развернут в среде 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.



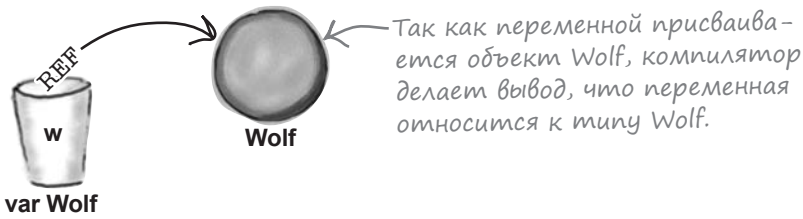
Объекты Inner и Outer объединены связью особого рода. Inner может использовать переменные Outer, и наоборот.

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:

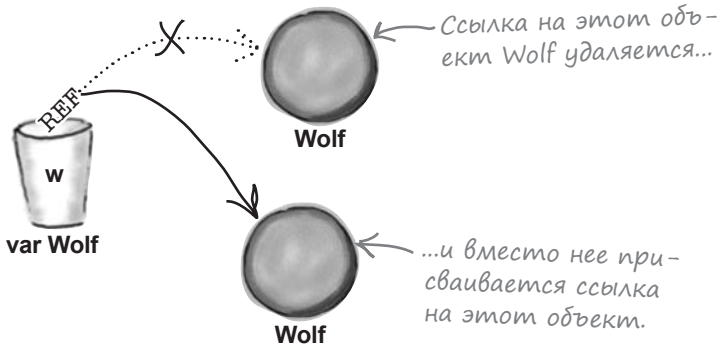
```
var w = Wolf()
```

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.**

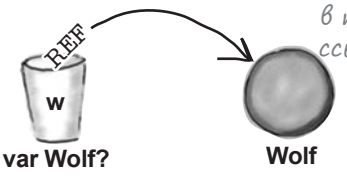
Для чего нужны null-совместимые типы?

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()
```

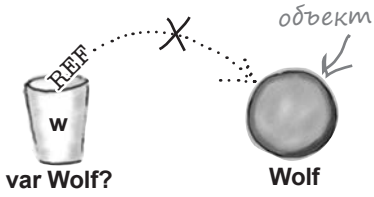
Запись Wolf? означает, что в переменной могут храниться ссылки на объекты Wolf или null.



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`):

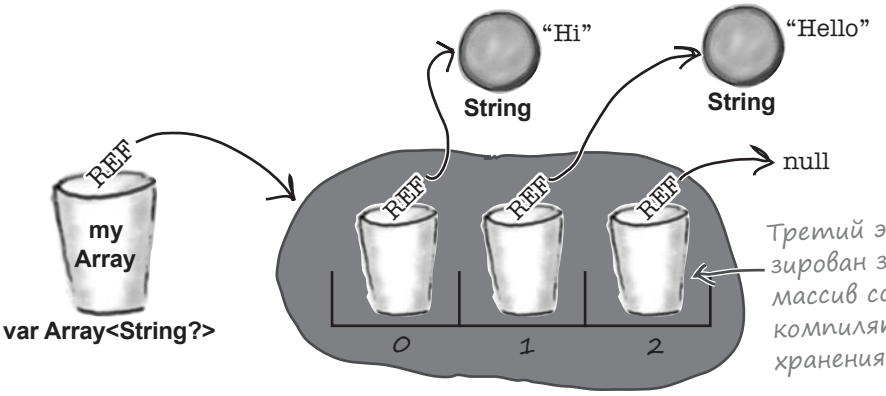
```
var myArray: Array<String?> = arrayOf("Hi", "Hello")
```

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?
```