Generics, objects, and extensions

Изучим концепции Kotlin, которые помогут вам структурировать более крупные приложения:

* Дженерики
* Различные виды классов (классы перечислений и классы данных)
* Синглтон и сопутствующие объекты
* Свойства и функции расширения
* Функции области

К концу этой лабораторной работы вы должны иметь более глубокие знания о коде, который вы уже видели в этом курсе, и узнать несколько примеров того, когда вы столкнетесь или будете использовать эти концепции в своих собственных приложениях.

## [2. Создайте повторно используемый класс с дженериками](https://developer.android.com/codelabs/basic-android-kotlin-compose-generics?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-3-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-generics#1)

Допустим, вы пишете приложение для онлайн-викторины, похожее на тесты. Часто существует несколько типов вопросов викторины, например «заполнение пропусков», «правда» или «ложь». Отдельный вопрос викторины может быть представлен классом с несколькими свойствами.

Текст вопроса в викторине может быть представлен строкой. Вопросы викторины также должны представлять собой ответ. Однако разные типы вопросов, например «истина» или «ложь», могут потребовать представления ответа с использованием другого типа данных. Давайте определим три различных типа вопросов.

* **Вопрос с заполнением пропусков** : ответом является слово, обозначенное знаком String.
* **Истина или Ложь вопрос** : Ответ представлен значком Boolean.
* **Математические задачи** : Ответом является числовое значение. Ответ на простую арифметическую задачу представлен знаком Int.

Кроме того, вопросы викторины в нашем примере, независимо от типа вопроса, также будут иметь рейтинг сложности. Рейтинг сложности представлен строкой с тремя возможными значениями: "easy", "medium"или "hard".

Определите классы для представления каждого типа вопросов викторины:

1. Перейдите на [игровую площадку Kotlin](https://developer.android.com/training/kotlinplayground) .
2. Над функцией main() определите класс для вопросов с заполнением пропусков с именем FillInTheBlankQuestion.

class FillInTheBlankQuestion(  
    val questionText: String,  
    val answer: String,  
    val difficulty: String  
)

1. Под классом FillInTheBlankQuestion определите другой класс, названный TrueOrFalseQuestion для истинных или ложных вопросов.

class TrueOrFalseQuestion(  
    val questionText: String,  
    val answer: Boolean,  
    val difficulty: String  
)

1. Наконец, под двумя другими классами определите класс NumericQuestion.

class NumericQuestion(  
    val questionText: String,  
    val answer: Int,  
    val difficulty: String  
)

1. Взгляните на код, который вы написали. Вы замечаете повторение?

class FillInTheBlankQuestion(  
    val questionText: String,  
    val answer: String,  
    val difficulty: String  
)  
  
class TrueOrFalseQuestion(  
    val questionText: String,  
    val answer: Boolean,  
    val difficulty: String  
)  
class NumericQuestion(  
    val questionText: String,  
    val answer: Int,  
    val difficulty: String  
)

Все три класса имеют одни и те же свойства: questionText, answer и difficulty. Единственная разница заключается в типе данных свойства answer. Вы можете подумать, что очевидным решением является создание родительского класса с questionText и difficulty, и каждый подкласс определяет свойство answer.

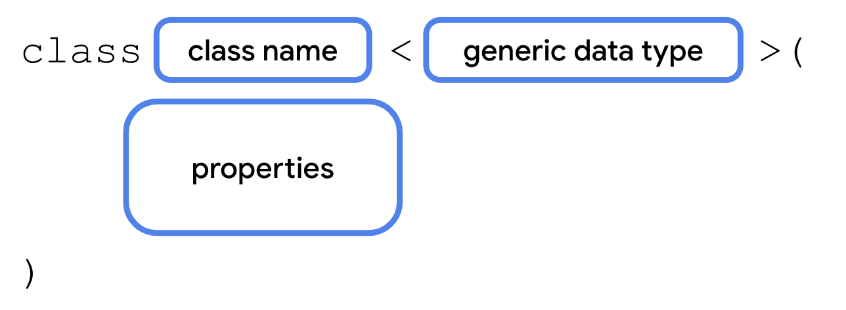
Однако использование наследования имеет ту же проблему, что и выше. Каждый раз, когда вы добавляете новый тип вопроса, вам необходимо добавить свойство answer. Единственное отличие — это тип данных. Также выглядит странно иметь родительский класс Question,  у которого нет свойства ответа.

Если вы хотите, чтобы свойство имело разные типы данных, создание подклассов не является решением. Вместо этого Kotlin предоставляет так называемые универсальные типы, которые позволяют вам иметь одно свойство, которое может иметь разные типы данных в зависимости от конкретного варианта использования.

## Что такое универсальный тип данных?

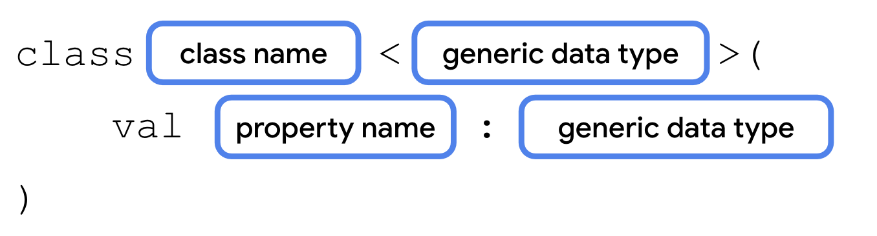
Универсальные типы, или сокращенно **дженерики** , позволяют типу данных, например классу, указывать неизвестный тип данных-заполнителя, который можно использовать с его свойствами и методами. Что именно это значит?

В приведенном выше примере вместо определения свойства ответа для каждого возможного типа данных вы можете создать один класс для представления любого вопроса и использовать имя-заполнитель для типа данных свойства answer. Фактический тип данных String— , Int, Boolean и т. д. — указывается при создании экземпляра этого класса. Везде, где используется имя-заполнитель, вместо него используется тип данных, переданный в класс. Синтаксис определения универсального типа для класса показан ниже:



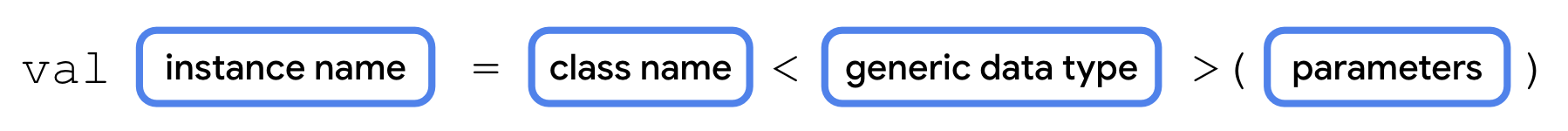
Общий тип данных предоставляется при создании экземпляра класса, поэтому его необходимо определить как часть сигнатуры класса. После имени класса идет левая угловая скобка ( <), за которой следует имя-заполнитель для типа данных, а затем правая угловая скобка ( >).

Имя-заполнитель затем можно использовать везде, где вы используете реальный тип данных в классе, например, для свойства.



Это идентично объявлению любого другого свойства, за исключением того, что вместо типа данных используется имя-заполнитель.

Как ваш класс в конечном итоге узнает, какой тип данных использовать? Тип данных, используемый универсальным типом, передается в качестве параметра в угловых скобках при создании экземпляра класса.



После имени класса идет левая угловая скобка ( <), за которой следует фактический тип данных , String, Boolean и Int т. д., а затем правая скобка ( >). Тип данных значения, которое вы передаете для универсального свойства, должен соответствовать типу данных в угловых скобках. Вы сделаете свойство ответа универсальным, чтобы можно было использовать один класс для представления вопросов викторины любого типа, независимо от того, является ли ответ String, Boolean, Intили любым произвольным типом данных.

**Примечание.** Универсальные типы, передаваемые при создании экземпляра класса, также называются «параметрами», хотя они являются частью отдельного списка параметров, а не значений свойств, помещенных в круглые скобки.

**Примечание.** Как и в приведенном выше примере, вы часто увидите имя универсального типа T(сокращение от типа) или другие заглавные буквы, если класс имеет несколько универсальных типов. Однако правила определенно не существует, и вы можете использовать более описательное имя для универсальных типов.

## Рефакторинг вашего кода для использования дженериков

Выполните рефакторинг вашего кода, чтобы использовать один класс Question с общим свойством ответа.

1. Удалите определения классов для FillInTheBlankQuestion, TrueOrFalseQuestionи NumericQuestion.
2. Создайте новый класс с именем Question.

class Question()

1. После имени класса, но перед круглыми скобками, добавьте параметр универсального типа, используя левые и правые угловые скобки. Вызов универсального типа T.

class Question<T>()

1. Добавьте свойства questionText, answer и difficulty. questionText должно  быть типа String. answer должен  иметь тип T,  поскольку его тип данных указывается при создании экземпляра класса Question. Свойство difficulty должно быть типа String.

class Question<T>(  
    val questionText: String,  
    val answer: T,  
    val difficulty: String  
)

1. Чтобы увидеть, как это работает с несколькими типами вопросов — «заполните пробелы», «истина» или «ложь» и т. д. — создайте три экземпляра класса Question в main(), как показано ниже.

fun main() {  
    val question1 = Question<String>("Quoth the raven \_\_\_", "nevermore", "medium")  
    val question2 = Question<Boolean>("The sky is green. True or false", false, "easy")  
    val question3 = Question<Int>("How many days are there between full moons?", 28, "hard")  
}

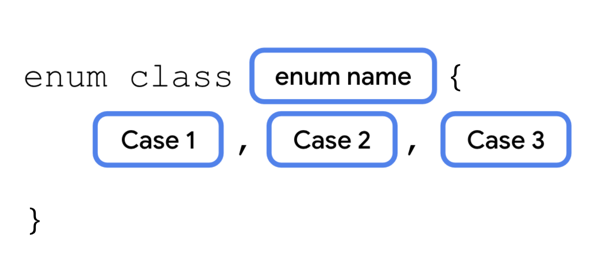
1. Запустите свой код, чтобы убедиться, что все работает. Теперь у вас должно быть три экземпляра класса Question — каждый с разными типами данных для ответа — вместо трех разных классов или вместо использования наследования. Если вы хотите обрабатывать вопросы с другим типом ответа, вы можете повторно использовать тот же класс Question.

## [3. Используйте класс перечисления](https://developer.android.com/codelabs/basic-android-kotlin-compose-generics?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-3-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-generics#2) enum

Выше вы определили свойство сложности с тремя возможными значениями: «легкий», «средний» и «сложный». Хотя это работает, есть пара проблем.

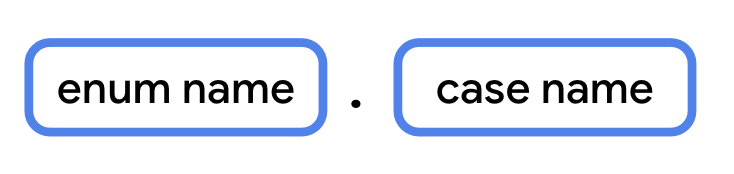
1. Если вы случайно напечатаете одну из трех возможных строк, это может привести к ошибке.
2. Если значения изменяются, например, "medium" переименовываются в "average", вам необходимо обновить все варианты использования строки.
3. Ничто не мешает вам или другому разработчику случайно использовать другую строку, которая не является одним из трех допустимых значений.
4. Код сложнее поддерживать, если вы добавите больше уровней сложности.

Kotlin помогает решить эти проблемы с помощью специального типа класса, называемого **классом перечисления** ***enum***. Класс перечисления используется для создания типов с ограниченным набором возможных значений. Например, в реальном мире четыре основных направления — север, юг, восток и запад — могут быть представлены классом перечисления. Нет необходимости, и код не должен позволять использовать какие-либо дополнительные указания. Синтаксис класса перечисления показан ниже.



Каждое возможное значение перечисления называется **константой перечисления** . Константы перечисления помещаются внутри фигурных скобок, разделенных запятыми. По соглашению каждая буква в имени константы должна писаться с заглавной буквы.

Вы ссылаетесь на константы перечисления, используя оператор точки.



## Используйте константу перечисления

Измените свой код, чтобы использовать константу перечисления вместо String, чтобы обозначить сложность.

1. Под классом Question определите класс enum с именем Difficulty.

enum class Difficulty {  
    EASY, MEDIUM, HARD  
}

1. В классе Question измените тип данных свойства difficulty с String на Difficulty.

class Question<T>(  
    val questionText: String,  
    val answer: T,  
    val difficulty: **Difficulty**  
)

1. При инициализации трех вопросов передайте константу перечисления для сложности.

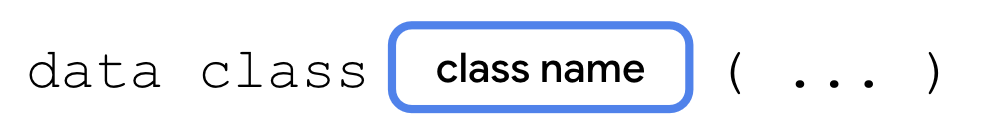
val question1 = Question<String>("Quoth the raven \_\_\_", "nevermore", Difficulty.MEDIUM)  
val question2 = Question<Boolean>("The sky is green. True or false", false, Difficulty.EASY)  
val question3 = Question<Int>("How many days are there between full moons?", 28, Difficulty.HARD)

## [4. Используйте класс данных](https://developer.android.com/codelabs/basic-android-kotlin-compose-generics?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-3-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-generics#3)

Многие из классов, с которыми вы работали до сих пор, например, подклассы Activity, имеют несколько методов для выполнения различных действий. Эти классы не просто представляют данные, но также содержат множество функций.

С другой стороны, классы, подобные классу Question, содержат только данные. У них нет методов, выполняющих действие. Их можно определить как **класс данных** (**data class** ). Определение класса как класса данных позволяет компилятору Kotlin делать определенные предположения и автоматически реализовывать некоторые методы. Например, toString() вызывается за кулисами функцией println(). Когда вы используете класс данных,  другие методы реализуются автоматически на основе свойств класса.

Чтобы определить класс данных, просто добавьте  ключевое слово data перед  ключевым словом class.



## Преобразование Question в класс данных

Во-первых, вы увидите, что происходит, когда вы пытаетесь вызвать метод, например, toString() в классе, который не является классом данных. Затем вы преобразуете его в класс данных Question, чтобы этот и другие методы были реализованы по умолчанию.

1. В main(), распечатайте результат toString() вызова question1.

fun main() {  
    val question1 = Question<String>("Quoth the raven \_\_\_", "nevermore", Difficulty.MEDIUM)  
    val question2 = Question<Boolean>("The sky is green. True or false", false, Difficulty.EASY)  
    val question3 = Question<Int>("How many days are there between full moons?", 28, Difficulty.HARD)  
**println(question1.toString())**}

1. Запустите свой код. В выходных данных отображается только имя класса и уникальный идентификатор объекта.

Question@37f8bb67

1. Превратите Question в класс данных, используя  ключевое слово data.

**data** class Question<T>(  
    val questionText: String,  
    val answer: T,  
    val difficulty: Difficulty  
)

1. Запустите свой код еще раз. Пометив это как класс данных, Kotlin может определить, как отображать свойства класса при вызове toString().

Question(questionText=Quoth the raven \_\_\_, answer=nevermore, difficulty=MEDIUM)

Когда класс определен как класс данных, реализуются следующие методы.

* equals()
* hashCode(): вы увидите этот метод при работе с определенными типами коллекций.
* toString()
* [componentN()](https://kotlinlang.org/docs/destructuring-declarations.html#example-returning-two-values-from-a-function): component1(), component2(), и т. д.
* copy()

**Примечание.** Класс данных должен иметь хотя бы один параметр в своем конструкторе, и все параметры конструктора должны быть отмечены значком val или var. Класс данных также не может быть abstract, open, sealedили inner.

## [5. Используйте одноэлементный объект](https://developer.android.com/codelabs/basic-android-kotlin-compose-generics?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-3-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-generics#4)

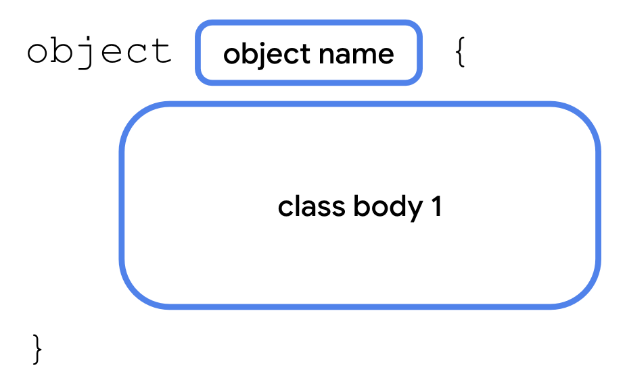
Существует множество сценариев, в которых вы хотите, чтобы класс имел только один экземпляр. Например:

1. Статистика игрока в мобильной игре для текущего пользователя.
2. Взаимодействие с одним аппаратным устройством, например передача звука через динамик.
3. Объект для доступа к удаленному источнику данных (например, базе данных Firebase).
4. Аутентификация, при которой одновременно должен входить только один пользователь.

В приведенных выше сценариях вам, вероятно, потребуется использовать класс. Однако вам понадобится создать экземпляр только одного экземпляра этого класса. Если имеется только одно аппаратное устройство или только один пользователь вошел в систему одновременно, нет смысла создавать более одного экземпляра. Наличие двух объектов, которые одновременно обращаются к одному и тому же аппаратному устройству, может привести к действительно странному и ошибочному поведению.

Вы можете четко указать в своем коде, что объект должен иметь только один экземпляр, определив его как синглтон. **Синглтон** — это класс , который может иметь только один экземпляр. Kotlin предоставляет специальную конструкцию, называемую **объектом** , которую можно использовать для создания одноэлементного класса.

## Определить одноэлементный объект



Синтаксис объекта аналогичен синтаксису класса. Просто используйте ключевое слово object вместо ключевого слова class. У одноэлементного объекта не может быть конструктора, поскольку вы не можете создавать экземпляры напрямую. Вместо этого все свойства определяются в фигурных скобках и получают начальное значение.

Некоторые из приведенных ранее примеров могут показаться неочевидными, особенно если вы еще не работали с конкретными аппаратными устройствами и не сталкивались с аутентификацией в своих приложениях. Однако по мере изучения разработки под Android вы увидите появление одноэлементных объектов. Давайте посмотрим на это в действии на простом примере с использованием объекта пользовательского состояния, для которого необходим только один экземпляр.

Для викторины было бы здорово иметь возможность отслеживать общее количество вопросов и количество вопросов, на которые студент ответил на данный момент. Вам понадобится только один экземпляр этого класса, поэтому вместо объявления его как класса объявите его как одноэлементный объект.

1. Создайте объект с именем StudentProgress.

object StudentProgress {  
}

1. В этом примере мы предположим, что всего вопросов десять, и на три из них уже даны ответы. Добавьте два Int свойства: total со значением 10 и answered со значением 3.

object StudentProgress {  
    var total: Int = 10  
    var answered: Int = 3  
}

## Доступ к одноэлементному объекту

Помните, что вы не можете напрямую создать экземпляр одноэлементного объекта? Как же тогда вы сможете получить доступ к его свойствам?

Поскольку одновременно существует только один экземпляр StudentProgress, вы получаете доступ к его свойствам, ссылаясь на имя самого объекта, за которым следует оператор точки (.), а затем имя свойства.



Обновите свою функцию main(), чтобы получить доступ к свойствам одноэлементного объекта.

1. В main() добавьте вызов println(), который выводит answered и total вопросы из объекта StudentProgress.

fun main() {  
 ...

    println("${StudentProgress.answered} of ${StudentProgress.total} answered.")  
}

1. Запустите свой код, чтобы убедиться, что все работает.

...

3 of 10 answered.

## Объявляйте объекты как сопутствующие объекты

Классы и объекты в Kotlin могут быть определены внутри других типов и могут стать отличным способом организации вашего кода. Вы можете определить одноэлементный объект внутри другого класса, используя **сопутствующий объект**. Объект-компаньон позволяет получить доступ к его свойствам и методам изнутри класса, если свойства и методы объекта принадлежат этому классу, что позволяет использовать более краткий синтаксис.

Чтобы объявить сопутствующий объект, просто добавьте ключевое слово companion перед ключевым словом object.



Вы создадите новый класс Quiz для хранения вопросов викторины и создадите StudentProgress сопутствующий объект класса Quiz.

1. Под перечислением Difficulty определите новый класс с именем Quiz.

**class Quiz {  
}**

1. Переместите question1, question2 и question3 из main() в класс Quiz. Вам также необходимо удалить println(question1.toString()),  если вы еще этого не сделали.

class Quiz {  
**val question1 = Question<String>("Quoth the raven \_\_\_", "nevermore", Difficulty.MEDIUM)  
    val question2 = Question<Boolean>("The sky is green. True or false", false, Difficulty.EASY)  
    val question3 = Question<Int>("How many days are there between full moons?", 28, Difficulty.HARD)**  
}

1. Переместите объект StudentProgress в класс Quiz.

class Quiz {  
    val question1 = Question<String>("Quoth the raven \_\_\_", "nevermore", Difficulty.MEDIUM)  
    val question2 = Question<Boolean>("The sky is green. True or false", false, Difficulty.EASY)  
    val question3 = Question<Int>("How many days are there between full moons?", 28, Difficulty.HARD)  
  
    **object StudentProgress {  
        var total: Int = 10  
        var answered: Int = 3  
    }**}

1. Отметьте объект StudentProgress ключевым словом companion.

**companion** object StudentProgress {  
    var total: Int = 10  
    var answered: Int = 3  
}

1. Обновите вызов println() для ссылки на свойства с помощью Quiz.answered и Quiz.total. Несмотря на то, что эти свойства объявлены в объекте StudentProgress, доступ к ним можно получить с помощью точечной записи, используя только имя класса Quiz.

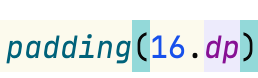
fun main() {  
    println("${**Quiz**.answered} of ${**Quiz**.total} answered.")  
}

1. Запустите свой код, чтобы проверить результат.

3 of 10 answered.

## [6. Расширьте классы новыми свойствами и методами.](https://developer.android.com/codelabs/basic-android-kotlin-compose-generics?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-3-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-generics#5)

При работе с Compose вы могли заметить интересный синтаксис при указании размера элементов пользовательского интерфейса. Числовые типы, такие как Double, имеют такие свойства, как определение размеров dp и sp.



Зачем разработчикам языка Kotlin включать свойства и функции во встроенные типы данных, особенно для создания пользовательского интерфейса Android? Смогли ли они предсказать будущее? Был ли Kotlin разработан для использования с Compose еще до того, как Compose появился?

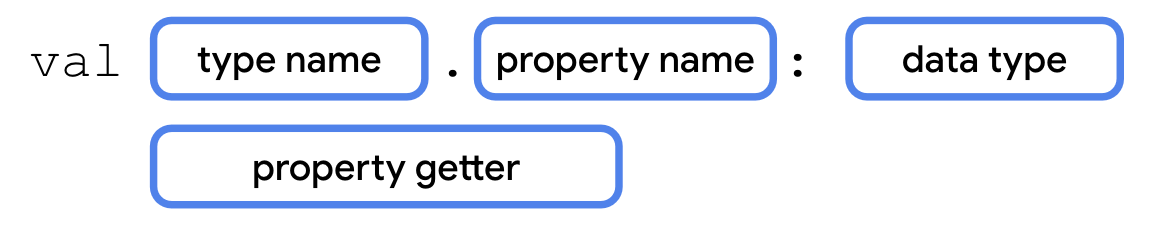
Конечно, нет! Когда вы пишете класс, вы часто не знаете точно, как другой разработчик будет использовать его или планирует использовать в своем приложении. Невозможно предсказать все будущие варианты использования, и неразумно добавлять ненужное раздувание кода в некоторых непредвиденных случаях использования.

Язык Kotlin дает другим разработчикам возможность расширять существующие типы данных, добавляя свойства и методы, к которым можно получить доступ с помощью точечного синтаксиса, как если бы они были частью этого типа данных. Например, разработчик, который не работал над типами с плавающей запятой в Kotlin, например, кто-то, создавший библиотеку Compose, может добавить свойства и методы, специфичные для измерений пользовательского интерфейса.

Поскольку вы видели этот синтаксис при изучении Compose ранее, пришло время узнать, как это работает «под капотом». Вы добавите некоторые свойства и методы для расширения существующих типов.

## Добавить свойство расширения

Чтобы определить свойство расширения, добавьте имя типа и оператор точки ( .) перед именем переменной.



Вы проведете рефакторинг кода в функции main(), чтобы распечатать ход выполнения теста с помощью свойства расширения.

1. Под  классом Quiz определите свойство расширения имени Quiz.StudentProgress progressText типа String.

val Quiz.StudentProgress.progressText: String

1. Определите метод получения для свойства расширения, который возвращает ту же строку, которая использовалась ранее в main().

val Quiz.StudentProgress.progressText: String

**get()** = "${answered} of ${total} answered"

1. Замените код в функции main() кодом, который печатает progressText. Поскольку это свойство расширения сопутствующего объекта, вы можете получить к нему доступ с помощью точечной записи, используя имя класса Quiz.

fun main() {  
    println(**Quiz**.progressText)  
}

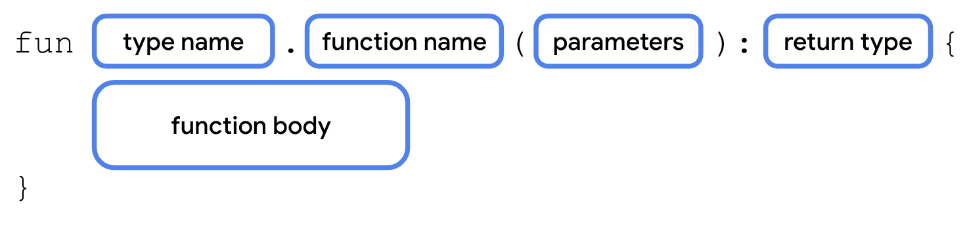
1. Запустите свой код, чтобы убедиться, что он работает.

3 of 10 answered.

**Примечание.** Свойства расширения не могут хранить данные, поэтому они должны быть доступны только для получения.

## Добавьте функцию расширения

Чтобы определить функцию расширения, добавьте имя типа и оператор точки ( .) перед именем функции.



Вы добавите функцию расширения для вывода хода выполнения теста в виде индикатора выполнения. Поскольку вы на самом деле не можете создать индикатор выполнения на игровой площадке Kotlin, вы распечатаете индикатор выполнения в стиле ретро, используя текст!

1. Добавьте функцию расширения к объекту StudentProgress с именем printProgressBar(). Функция не должна принимать никаких параметров и возвращать значение. Функцию добавляем после класса Quiz

fun Quiz.StudentProgress.printProgressBar() {  
}

1. Распечатайте символ ▓ answered несколько раз, используя repeat(). Эта темная часть индикатора выполнения представляет количество ответов на вопросы. Используйте print(), потому что вам не нужна новая строка после каждого символа.

fun Quiz.StudentProgress.printProgressBar() {  
**repeat(Quiz.answered) { print("▓") }**}

1. Распечатайте символ ▒, количество раз, равное разнице между total и answered, используя repeat(). Эта светлая часть представляет собой оставшиеся вопросы на панели процесса.

fun Quiz.StudentProgress.printProgressBar() {  
    repeat(Quiz.answered) { print("▓") }  
**repeat(Quiz.total - Quiz.answered) { print("▒") }**}

1. Выведите новую строку, используя команду println() без аргументов, а затем напечатайте progressText.

fun Quiz.StudentProgress.printProgressBar() {  
    repeat(Quiz.answered) { print("▓") }  
    repeat(Quiz.total - Quiz.answered) { print("▒") }  
**println()  
    println(Quiz.progressText)**}

1. Обновите код main() для вызова printProgressBar().

fun main() {  
**Quiz.printProgressBar()**}

1. Запустите свой код, чтобы проверить результат.

▓▓▓▒▒▒▒▒▒▒

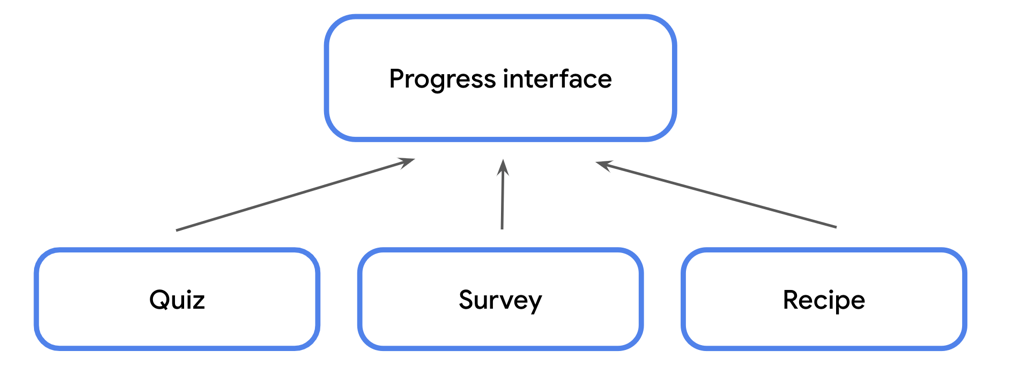
3 of 10 answered.

Обязательно ли что-то из этого делать? Конечно, нет. Однако наличие возможности расширения свойств и методов дает вам больше возможностей предоставить свой код другим разработчикам. Использование точечного синтаксиса в других типах может облегчить чтение вашего кода как для вас, так и для других разработчиков.

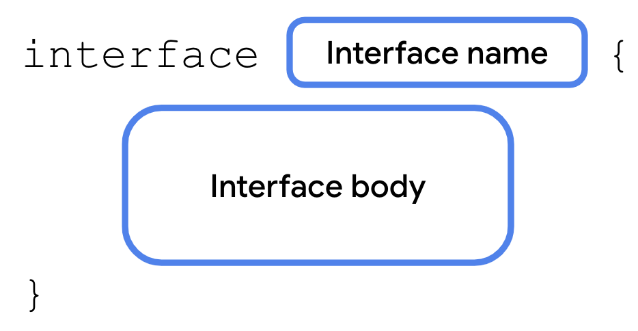
## [7. Перепишите функции расширения, используя интерфейсы.](https://developer.android.com/codelabs/basic-android-kotlin-compose-generics?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-3-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-generics#6)

На предыдущей странице вы видели, как добавлять свойства и методы к объекту StudentProgress без непосредственного добавления к нему кода, используя свойства расширения и функции расширения. Хотя это отличный способ добавить функциональность к уже определенному классу, расширение класса не всегда необходимо, если у вас есть доступ к исходному коду. Бывают также ситуации, когда вы не знаете, какой должна быть реализация, а знаете только, что должен существовать определенный метод или свойство. Если вам нужно, чтобы несколько классов имели одинаковые дополнительные свойства и методы, возможно, с разным поведением, вы можете определить эти свойства и методы с помощью **интерфейса** .

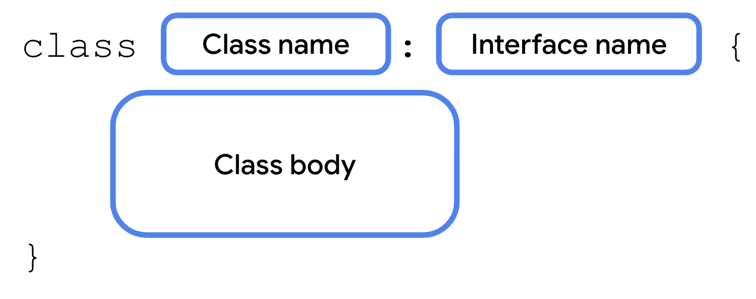
Например, в дополнение к тестам, предположим, у вас также есть классы для опросов, шагов в рецепте или любых других упорядоченных данных, которые могут использовать индикатор выполнения**. Вы можете определить нечто, называемое интерфейсом, который определяет методы и/или свойства, которые должен включать каждый из этих классов.**



Интерфейс определяется с помощью  ключевого слова interface, за которым следует имя в UpperCamelCase, а затем открывающие и закрывающие фигурные скобки. В фигурных скобках вы можете определить любые сигнатуры методов или свойства, доступные только для чтения, которые должен реализовать любой класс, соответствующий интерфейсу.



Интерфейс — это контракт. Говорят, что класс, соответствующий интерфейсу, расширяет интерфейс. Класс может объявить о своем желании расширить интерфейс, используя двоеточие ( :), за которым следует пробел и имя интерфейса.



В свою очередь, **класс должен реализовать все свойства и методы, указанные в интерфейсе**. Это позволяет легко гарантировать, что любой класс, которому необходимо расширить интерфейс, реализует одни и те же методы с одинаковой сигнатурой метода. Если вы каким-либо образом измените интерфейс, например, добавите или удалите свойства или методы или измените сигнатуру метода, компилятор потребует от вас обновить любой класс, расширяющий интерфейс, сохраняя ваш код согласованным и простым в обслуживании.

**Интерфейсы допускают изменение поведения классов, которые их расширяют**. **Каждый класс должен обеспечить реализацию.**

Давайте посмотрим, как можно переписать индикатор выполнения, чтобы использовать интерфейс, и расширить этот интерфейс классом Quiz.

1. Над классом Quiz определите интерфейс с именем ProgressPrintable. Мы выбрали это имя ProgressPrintable,  потому что оно позволяет любому расширяющему его классу печатать индикатор выполнения.

interface ProgressPrintable {  
}

1. В интерфейсе ProgressPrintable определите свойство с именем progressText.

interface ProgressPrintable {  
**val progressText: String**}

1. Измените объявление класса Quiz, чтобы расширить интерфейс ProgressPrintable.

class Quiz : **ProgressPrintable** {  
    ...   
}

1. В классе Quiz добавьте свойство с именем progressText типа String, как указано в интерфейсе ProgressPrintable. Поскольку свойство получено из ProgressPrintable, перед val следует указать ключевое слово override.

override val progressText: String

1. Скопируйте метод получения свойства из старого свойства расширения progressText.

override val progressText: String  
        get() = "${answered} of ${total} answered"

1. Удалите старое свойство расширения progressText.

**Код для удаления:**

val Quiz.StudentProgress.progressText: String  
    get() = "${answered} of ${total} answered"

1. В интерфейсе ProgressPrintable добавьте метод с именем printProgressBar, который не принимает параметров и не имеет возвращаемого значения.

interface ProgressPrintable {  
    val progressText: String  
**fun printProgressBar()**}

1. В  классе Quiz добавьте метод printProgressBar(), используя ключевое слово override.

**override** fun printProgressBar() {  
}

1. Переместите код из старой функции расширения printProgressBar() в новую printProgressBar() из интерфейса. Измените последнюю строку, чтобы она ссылалась на новую переменную progressText из интерфейса, удалив ссылку на Quiz.

override fun printProgressBar() {  
    repeat(Quiz.answered) { print("▓") }  
    repeat(Quiz.total - Quiz.answered) { print("▒") }  
    println()  
    println(progressText)  
}

1. Удалите функцию расширения printProgressBar(). Эта функциональность теперь принадлежит классу Quiz, расширяющему ProgressPrintable.

**Код для удаления:**

fun Quiz.StudentProgress.printProgressBar() {  
    repeat(Quiz.answered) { print("▓") }  
    repeat(Quiz.total - Quiz.answered) { print("▒") }  
    println()  
    println(Quiz.progressText)  
}

1. Обновите код в main(). Поскольку функция printProgressBar()теперь является методом класса Quiz, вам необходимо сначала создать экземпляр объекта Quiz, а затем вызвать printProgressBar().

fun main() {  
    Quiz().printProgressBar()  
}

1. Запустите свой код. Вывод не изменился, но ваш код теперь стал более модульным. По мере роста вашей кодовой базы вы можете легко добавлять классы, соответствующие одному и тому же интерфейсу, для повторного использования кода без наследования от суперкласса.

▓▓▓▒▒▒▒▒▒▒

3 of 10 answered.

Существует множество вариантов использования интерфейсов, помогающих структурировать ваш код, и вы начнете видеть, что они часто используются в общих модулях.

## [8. Используйте функции области видимости для доступа к свойствам и методам класса.](https://developer.android.com/codelabs/basic-android-kotlin-compose-generics?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-3-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-generics#7)

Как вы уже видели, Kotlin включает в себя множество функций, позволяющих сделать ваш код более кратким.

Одна из таких функций, с которой вы столкнетесь, продолжая изучать разработку под Android, — это **функции области видимости**. Функции области позволяют вам лаконично получать доступ к свойствам и методам класса без необходимости многократного доступа к имени переменной. Что именно это значит? Давайте посмотрим на пример.

## Устраните повторяющиеся ссылки на объекты с помощью функций области видимости.

**Функции области** — это функции более высокого порядка, которые позволяют вам получать доступ к свойствам и методам объекта, не обращаясь к имени объекта. Они называются функциями области видимости, поскольку тело переданной функции принимает область действия объекта, с которым вызывается функция области видимости. Например, некоторые функции области видимости позволяют получить доступ к свойствам и методам класса, как если бы функции были определены как метод этого класса. Это может сделать ваш код более читабельным, позволяя опускать имя объекта, если его включение избыточно.

Чтобы лучше это проиллюстрировать, давайте взглянем на несколько различных функций области видимости, с которыми вы столкнетесь позже в этом курсе.

## Замените длинные имена объектов, используяlet()

Функция let() позволяет ссылаться на объект в лямбда-выражении it, используя идентификатор вместо фактического имени объекта. Это поможет вам избежать повторного использования длинного и более описательного имени объекта при доступе к нескольким свойствам. Эта функция let() представляет собой функцию расширения, которую можно вызвать для любого объекта Kotlin с использованием точечной записи.

Попробуйте получить доступ к свойствам question1, question2и question3 использовать let():

1. Добавьте функцию в класс Quiz с именем printQuiz().

fun printQuiz() {  
      
}

1. Добавьте следующий код, который печатает вопросы questionText, answer и difficulty. Хотя доступ к нескольким свойствам осуществляется для question1, question2 и question3, каждый раз используется все имя переменной. Если имя переменной изменилось, вам придется обновлять каждое использование.

fun printQuiz() {  
    println(question1.questionText)  
    println(question1.answer)  
    println(question1.difficulty)  
    println()  
    println(question2.questionText)  
    println(question2.answer)  
    println(question2.difficulty)  
    println()  
    println(question3.questionText)  
    println(question3.answer)  
    println(question3.difficulty)  
    println()  
}

1. Окружите код доступа к свойствам questionText, answer и difficulty, вызовом функции let() question1, question2 и question3. Замените на него имя переменной в каждом лямбда-выражении.

fun printQuiz() {  
    question1.let {  
        println(it.questionText)  
        println(it.answer)  
        println(it.difficulty)  
    }  
    println()  
    question2.let {  
        println(it.questionText)  
        println(it.answer)  
        println(it.difficulty)  
    }  
    println()  
    question3.let {  
        println(it.questionText)  
        println(it.answer)  
        println(it.difficulty)  
    }  
    println()  
}

1. Обновите код main() чтобы создать экземпляр класса Quiz с именем quiz.

fun main() {  
    val quiz = Quiz()  
}

1. Вызов printQuiz().

fun main() {  
    val quiz = Quiz()  
    quiz.printQuiz()  
}

1. Запустите свой код, чтобы убедиться, что все работает.

Quoth the raven \_\_\_

nevermore

MEDIUM

The sky is green. True or false

false

EASY

How many days are there between full moons?

28

HARD

## Вызов методов объекта без переменной с помощью метода apply()

Одной из замечательных особенностей функций области видимости является то, что вы можете вызывать их для объекта еще до того, как этот объект будет присвоен переменной. Например,  функция apply() является функцией расширения, которую можно вызывать для объекта с использованием точечной записи. Функция apply() также возвращает ссылку на этот объект, чтобы ее можно было сохранить в переменной.

Обновите код main() для вызова функции apply().

1. Вызов apply() после закрывающей скобки при создании экземпляра класса Quiz. Вы можете опустить круглые скобки при вызове apply()и использовать конечный лямбда-синтаксис.

val quiz = Quiz().apply {  
}

1. Переместите вызов printQuiz() внутрь лямбда-выражения. Вам больше не нужно ссылаться на quiz переменную или использовать точечную запись.

val quiz = Quiz().apply {  
    printQuiz()  
}

1. Функция apply() возвращает экземпляр класса Quiz, но, поскольку вы его больше нигде не используете, удалите quiz переменную. Благодаря этой apply() функции вам даже не понадобится переменная для вызова методов экземпляра Quiz.

Quiz().apply {  
    printQuiz()  
}

1. Запустите свой код. Обратите внимание, что вы могли вызвать этот метод без ссылки на экземпляр Quiz. Функция apply() вернула объекты, которые хранились в  quiz.

Quoth the raven \_\_\_

nevermore

MEDIUM

The sky is green. True or false

false

EASY

How many days are there between full moons?

28

HARD

Хотя использование функций области видимости не является обязательным для достижения желаемого результата, приведенные выше примеры иллюстрируют, как они могут сделать ваш код более кратким и избежать повторения одного и того же имени переменной.

Приведенный выше код демонстрирует всего два примера, но вам рекомендуется добавить в закладки и обратиться к [документации по функциям области видимости](https://kotlinlang.org/docs/scope-functions.html) , когда вы столкнетесь с их использованием позже в этом курсе.

## [9. Резюме](https://developer.android.com/codelabs/basic-android-kotlin-compose-generics?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-3-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-generics#8)

Вы только что получили возможность увидеть в действии несколько новых функций Kotlin.

**Обобщенные типы данных** позволяют передавать типы данных в качестве параметров классам, **классы перечислений** определяют ограниченный набор возможных значений, а **классы данных** помогают автоматически генерировать некоторые полезные методы для классов.

Вы также увидели, как создать **одноэлементный объект**, ограниченный одним экземпляром, как сделать его **сопутствующим объектом другого класса** и как расширить существующие классы новыми свойствами, доступными только для получения, и новыми методами. Наконец, вы увидели несколько примеров того, как **функции области видимости** могут обеспечить более простой синтаксис при доступе к свойствам и методам.

Вы увидите эти концепции далее, когда узнаете больше о Kotlin, разработке для Android и Compose. Теперь вы лучше понимаете, как они работают и как они могут улучшить возможность повторного использования и читабельность вашего кода.

**Больше информации**

https://developer.android.com/codelabs/basic-android-kotlin-compose-generics?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-3-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-generics#9