

Generics, objects и extensions

Введение

На протяжении десятилетий программисты придумали несколько функций языка программирования, которые помогают писать более качественный код: выражение одной и той же идеи меньшим количеством кода, абстракция для выражения сложных идей, написание кода, который не позволяет другим разработчикам случайно совершить ошибки, - вот лишь несколько примеров. Язык Kotlin не является исключением, и в нем есть ряд функций, призванных помочь разработчикам писать более выразительный код.

К сожалению, эти функции могут усложнить работу, если вы программируете впервые. Хотя они могут показаться полезными, степень их полезности и проблемы, которые они решают, не всегда очевидны. Скорее всего, вы уже сталкивались с некоторыми функциями, используемыми в Compose и других библиотеках.

Хотя опыт ничем не заменишь, этот урок знакомит вас с несколькими концепциями Kotlin, которые помогут вам структурировать большие приложения:

- Generics
- Различные виды классов (классы перечислений и классы данных)
- Синглтон и объекты-компаньоны
- Свойства и функции расширения
- Функции области видимости

К концу этого урока вы должны получить более глубокие знания о коде, с которым вы уже познакомились в этом курсе, и узнать несколько примеров того, как вы будете использовать эти концепции в своих собственных приложениях.

Предварительные требования

- Знакомство с концепциями объектно-ориентированного программирования, включая наследование.
- Как определять и реализовывать интерфейсы.

Что вы узнаете

- Как определить параметр общего типа для класса.
- Как инстанцировать общий класс.
- Когда использовать классы перечислений и данных.
- Как определить параметр общего типа, который должен реализовывать интерфейс.
- Как использовать функции области видимости для доступа к свойствам и методам класса.
- Как определить для класса объекты-синглтоны и объекты-компаньоны.
- Как расширить существующие классы новыми свойствами и методами.

Что вам понадобится

- IntelliJ Idea

Создание многократно используемого класса с помощью дженериков

Допустим, вы пишете приложение для онлайн-викторины, похожее на те, что вы видели в этом курсе. Вопросы викторины часто бывают нескольких типов, например «заполни пустое место», «истина 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 ответ. Однако для разных типов вопросов, например true или false, может потребоваться представление ответа с помощью разных типов данных. Давайте определим три различных типа вопросов.

- Вопрос с заполнением пустого места: Ответом является слово, представленное строкой.
- Вопрос «истина или ложь»: Ответ представлен булевым числом.
- Математические задачи: Ответом является числовое значение. Ответ на простую арифметическую задачу представлен числом 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: "легко", "средне" или "тяжело".

Определите классы для представления каждого типа вопросов викторины:

- Над функцией main() определите класс для вопросов «в пустоту» с именем FillInTheBlankQuestion, состоящий из свойства String для текста вопроса, свойства String для ответа и свойства String для сложности.

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

- Ниже класса FillInTheBlankQuestion определите еще один класс TrueOrFalseQuestion для истинных или ложных вопросов, состоящий из свойства String для текста вопроса, свойства Boolean для ответа и свойства String для сложности.

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

Наконец, под двумя другими классами определите класс **NumericQuestion**, состоящий из свойства String для текста вопроса, свойства Int для ответа и свойства String для сложности.

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

Посмотрите на код, который вы написали. Заметили ли вы повторения?

```
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 одинаковые свойства: `questionText`, `answer` и `difficulty`. Единственное различие заключается в типе данных свойства «ответ». Вы можете подумать, что очевидным решением является создание родительского класса с текстом вопроса и сложностью, а каждый подкласс определяет свойство ответа.

Однако при использовании наследования возникает та же проблема, что и выше. Каждый раз, когда вы добавляете новый тип вопроса, вам приходится добавлять свойство ответа. Единственное различие заключается в типе данных. Кроме того, странно выглядит родительский класс `Question`, у которого нет свойства ответа.

Когда вы хотите, чтобы свойство имело разные типы данных, подклассификация - не выход. Вместо этого Kotlin предоставляет так называемые **общие типы**, которые позволяют вам иметь одно свойство, которое может иметь различные типы данных в зависимости от конкретного случая использования.

Что такое общий тип данных?

Общие типы, или дженерики, позволяют типу данных, например классу, указать неизвестный тип данных, который может использоваться в его свойствах и методах. Что именно это означает?

В приведенном выше примере вместо того, чтобы определять свойство ответа для каждого возможного типа данных, вы можете создать один класс для представления любого вопроса и использовать имя-заполнитель для типа данных свойства ответа.

Фактический тип данных - String, Int, Boolean и т. д. указывается при инстанцировании этого класса. Там, где используется имя-заместитель, вместо него используется тип данных, переданный в класс.

Синтаксис для определения общего типа для класса показан ниже:

```
class class name < generic data type > (
    properties
)
```

Общий тип данных предоставляется при инстанцировании класса, поэтому он должен быть определен как часть сигнатуры класса. После имени класса следует угловая скобка (<), обращенная влево, затем имя-заполнитель для типа данных, за которым следует угловая скобка (>), обращенная вправо.

Имя-заместитель можно использовать везде, где в классе используется реальный тип данных, например, в свойстве.

```
class class name < generic data type > (
    val property name : generic data type
)
```

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

Как в конечном итоге ваш класс узнает, какой тип данных использовать? Тип данных, который использует общий тип, передается в качестве параметра в угловых скобках при инстанцировании класса.

```
val instance name = class name < generic data type > ( parameters )
```

После имени класса следует угловая скобка (<), обращенная влево, за ней - тип данных (String, Boolean, Int и т. д.), а за ней - скобка (>), обращенная вправо. Тип данных значения, которое вы передаете в родовое свойство, должен совпадать с типом данных в угловых скобках. Вы сделаете свойство ответа общим, чтобы можно было использовать один класс для представления любого типа вопроса

викторины, независимо от того, является ли ответ строкой, булевой функцией, Int или любым произвольным типом данных.

Примечание: Родовые типы, передаваемые при инстанцировании класса, также называются «параметрами», хотя они являются частью отдельного списка параметров, а не значений свойств, заключенных в круглые скобки.

Примечание: Как и в приведенном выше примере, вы часто будете видеть общий тип с именем **T** (сокращение от **type**) или другие заглавные буквы, если класс имеет несколько общих типов. Однако это не является правилом, и вы можете использовать более описательное имя для общих типов.

Рефакторинг вашего кода для использования дженериков

- Переделайте свой код, чтобы использовать один класс с именем `Question` и общим свойством ответа.
- Удалите определения классов `FillInTheBlankQuestion`, `TrueOrFalseQuestion` и `NumericQuestion`. Создайте новый класс с именем `Question`.

```
class Question()
```

После имени класса, но перед круглыми скобками, добавьте параметр общего типа, используя угловые скобки слева и справа. Назовите общий тип `T`.

```
class Question<T>()
```

Добавьте свойства `questionText`, `answer` и `difficulty`. `questionText` должен иметь тип `String`. `answer` должен иметь тип `T`, поскольку его тип данных задается при инстанцировании класса `Question`. Свойство `difficulty` должно иметь тип `String`.

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

Чтобы увидеть, как это работает с несколькими типами вопросов - «заполнить пустое место», «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")  
}
```

- 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`.

Использование класса перечисления `enum`

В предыдущем разделе вы определили свойство сложности с тремя возможными значениями: "легко", "средне" и "тяжело". Хотя это работает, есть несколько проблем.

Если вы случайно неправильно введете одну из трех возможных строк, это может привести к ошибкам. Если значения изменятся, например, «средний» будет переименован в «средний», то вам придется обновить все варианты использования строки. Ничто не мешает вам или другому разработчику случайно использовать другую строку, которая не является одним из трех допустимых значений. Код становится сложнее поддерживать, если вы добавляете больше уровней сложности.

Kotlin помогает решить эти проблемы с помощью специального типа класса, называемого классом **перечисления**. Класс `enum` используется для создания типов с ограниченным набором возможных значений. Например, в реальном мире четыре кардинальных направления - север, юг, восток и запад - могли бы быть представлены классом `enum`. Нет необходимости, и код не должен допускать использования дополнительных направлений. Синтаксис класса `enum` показан ниже.

```
enum class enum name {  
    Case 1 , Case 2 , Case 3  
}
```

Каждое возможное значение перечисления называется **константой перечисления**. Константы перечислений помещаются внутрь фигурных скобок, разделенных запятыми. Принято писать каждую букву в имени константы с заглавной буквы.

Вы ссылаетесь на константы перечисления с помощью оператора `dot`.

enum name

.

case name

Используйте константу enum

- Измените свой код, чтобы использовать константу **enum** вместо String для представления сложности.
- Ниже класса Question определите класс enum под названием Difficulty.

```
enum class Difficulty {  
    EASY, MEDIUM, HARD  
}
```

- В классе Question измените тип данных свойства difficulty со String на Difficulty.

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

- При инициализации трех вопросов передайте константу enum 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)
```

Используйте класс данных data

Многие классы, с которыми вы работали до сих пор, например подклассы Activity, имеют несколько методов для выполнения различных действий. Эти классы не просто представляют данные, но и содержат много функциональности.

Такие классы, как класс Question, с другой стороны, содержат только данные. У них нет методов, выполняющих какие-либо действия. Их можно определить как класс данных. 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, `toString()` и другие методы реализуются автоматически на основе свойств класса.

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

`data class` **class name** `(...)`

Преобразование Question в класс данных

Сначала вы увидите, что происходит, когда вы пытаетесь вызвать метод типа `toString()` в классе, не являющемся классом данных. Затем вы преобразуете `Question` в класс данных, чтобы этот и другие методы были реализованы по умолчанию.

В `main()` выведите результат вызова `toString()` для `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())
}
```

- Запустите свой код. На выходе вы увидите только имя класса и уникальный идентификатор объекта.

`Question@37f8bb67`

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

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

- Запустите код снова. Пометив этот класс как класс данных, Kotlin может определить, как отображать свойства класса при вызове `toString()`.

```
Question(questionText=Quoth the raven __, answer=nevermore, difficulty=MEDIUM)
```


Когда класс определяется как класс данных, реализуются следующие методы.

- equals()
- hashCode(): Вы увидите этот метод при работе с некоторыми типами коллекций.
- toString()
- componentN(): component1(), component2(), etc.
- copy()

Примечание: Класс данных должен иметь хотя бы один параметр в своем конструкторе, а все параметры конструктора должны быть помечены val или var. Класс данных также не может быть абстрактным, открытым, запечатанным или внутренним.

Используйте объект-синглтон

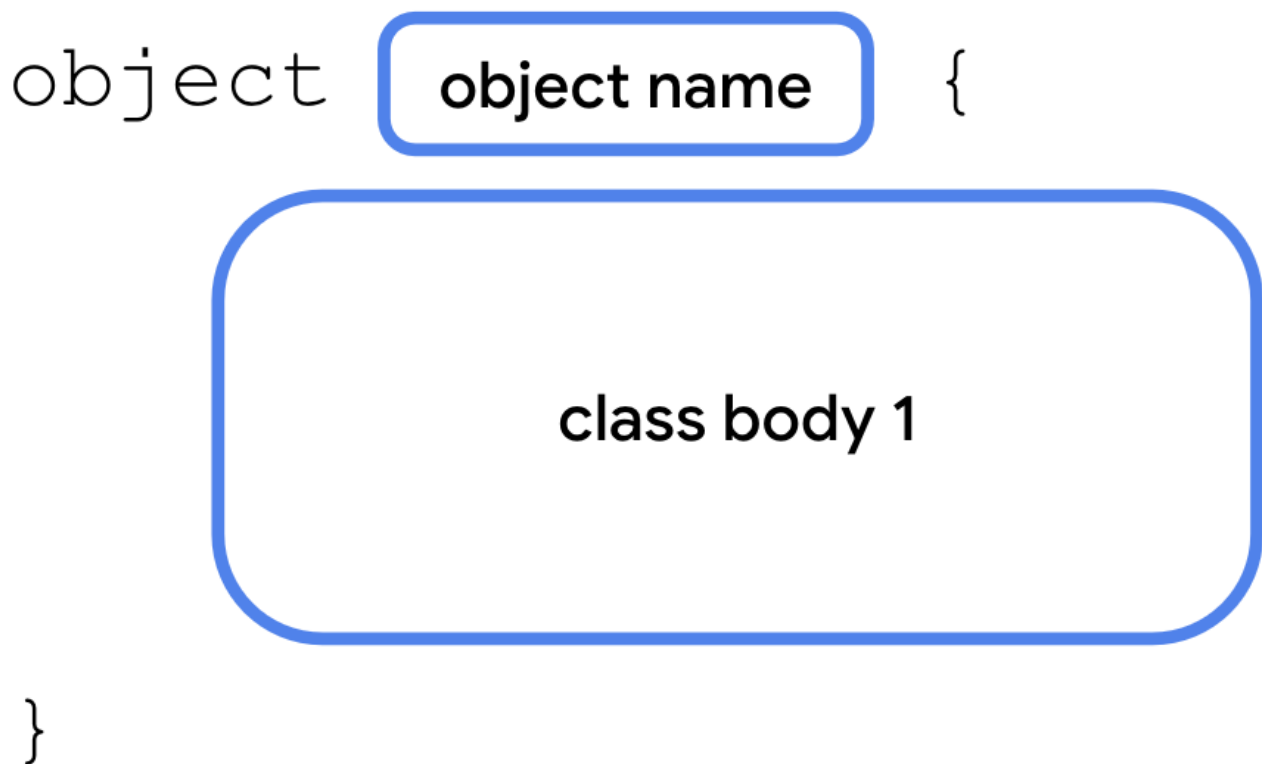
Существует множество сценариев, когда вы хотите, чтобы у класса был только один экземпляр. Например:

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

В вышеперечисленных сценариях вам, вероятно, потребуется использовать класс. Однако вам потребуется создать только **один экземпляр** этого класса. Если есть только одно аппаратное устройство или только один пользователь, входящий в систему одновременно, то нет причин создавать более одного экземпляра. Наличие двух объектов, одновременно обращающихся к одному и тому же аппаратному устройству, может привести к очень странному поведению и ошибкам.

Вы можете четко указать в 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, которая может быть использована для создания класса-синглтона.

Определение объект-одиночку Singleton



Синтаксис объекта аналогичен синтаксису класса. Просто используйте ключевое слово `object` вместо ключевого слова `class`. У объекта-синглтона не может быть конструктора, так как вы не можете создавать экземпляры напрямую. Вместо этого все свойства определяются внутри фигурных скобок и получают начальное значение.

Некоторые из приведенных ранее примеров могут показаться неочевидными, особенно если вы еще не работали с конкретными аппаратными устройствами или не сталкивались с аутентификацией в своих приложениях. Тем не менее, вы увидите, как объекты-синглтоны будут появляться по мере того, как вы будете продолжать изучать разработку Android. Давайте посмотрим на это в действии на простом примере с объектом состояния пользователя, в котором нужен только один экземпляр.

Для викторины было бы здорово иметь возможность отслеживать общее количество вопросов и количество вопросов, на которые ученик ответил на данный момент. Вам понадобится только один экземпляр этого класса, поэтому вместо того, чтобы объявлять его как класс, объявите его как объект-синглтон.

Создайте объект с именем `StudentProgress`.

```
object StudentProgress {  
}
```

Для этого примера мы предположим, что всего вопросов десять, и на три из них уже есть ответы. Добавьте два свойства `Int`: `total` со значением 10 и `answered` со значением 3.

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

Доступ к объекту-синглтону

Помните, что вы не можете создать экземпляр объекта-синглтона напрямую? Как тогда вы можете получить доступ к его свойствам?

Поскольку одновременно существует только один экземпляр `StudentProgress`, вы получаете доступ к его свойствам, обращаясь к имени **самого объекта**, затем к оператору точки (`.`), а затем к имени свойства.

object name

.

property name

- Обновите функцию `main()`, чтобы получить доступ к свойствам объекта-синглтона.

В функции `main()` добавьте вызов `println()`, который выводит ответы и общее количество вопросов из объекта `StudentProgress`.

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

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

```
...  
3 of 10 answered.
```

Объявление объектов как объектов-компаньонов

Классы и объекты в Kotlin могут быть определены внутри других типов, и это отличный способ организовать код. Вы можете определить объект-синглтон внутри другого класса с помощью `companion-object`. **companion-object** позволяет обращаться к его свойствам и методам изнутри класса, если свойства и методы объекта принадлежат этому классу, что позволяет использовать более лаконичный синтаксис.

Чтобы объявить `companion-object`, просто добавьте ключевое слово **companion** перед ключевым словом `object`.

companion object

object name

Вы создадите новый класс Quiz для хранения вопросов викторины и сделаете StudentProgress объектом-компаньоном класса Quiz.

- Ниже перечисления Difficulty определите новый класс с именем Quiz.

```
class Quiz {  
}
```

- Переместите question1, question2 и question3 из main() в класс Quiz. Также необходимо удалить println(question1.toString()), если вы этого еще 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)  
}
```

- 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  
    }  
}
```

- Пометьте объект StudentProgress ключевым словом companion.

```
companion object StudentProgress {  
    var total: Int = 10  
}
```

```
    var answered: Int = 3
}
```

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

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

- Запустите свой код, чтобы проверить результат.

3 of 10 answered.

Расширение классов новыми свойствами и методами

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

padding(16.dp)

Зачем разработчикам языка Kotlin включать свойства и функции для встроенных типов данных, специально для создания пользовательского интерфейса Android? Могли ли они предсказать будущее? Был ли Kotlin разработан для использования с Compose еще до появления Compose?

Конечно же, нет! Когда вы пишете класс, вы часто не знаете, как именно другой разработчик будет использовать или планирует использовать его в своем приложении. Невозможно предугадать все будущие варианты использования, да и неразумно добавлять в код ненужный объем для какого-то непредвиденного случая использования.

Что делает язык Kotlin, так это предоставляет другим разработчикам возможность расширять существующие типы данных, добавляя свойства и методы, к которым можно обращаться с помощью синтаксиса точек, как если бы они были частью этого типа данных. Например, разработчик, который не работал над типами с плавающей точкой в Kotlin, например, создающий библиотеку Compose, может добавить свойства и методы, специфичные для размеров UI.

Поскольку вы видели этот синтаксис при изучении Compose в первых двух частях, пришло время узнать, как это работает «под капотом». Вы добавите несколько свойств и методов, чтобы расширить существующие типы.

Добавление свойства расширения

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

`val` **type name** . **property name** : **data type**

property getter

Вы рефакторите код в функции `main()`, чтобы выводить прогресс теста с помощью свойства расширения.

- В классе `Quiz` определите свойство расширения `Quiz.StudentProgress` с именем `progressText` типа `String`.

```
val Quiz.StudentProgress.progressText: String
```

- Определите геттер для свойства `extension`, который возвращает ту же строку, что и в `main()`.

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

- Замените код в функции `main()` на код, который печатает `progressText`. Поскольку это свойство является расширением объекта-компаньона, вы можете получить к нему доступ с помощью точечной нотации, используя имя класса, `Quiz`.

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

- Запустите свой код, чтобы убедиться в его работоспособности.

3 of 10 answered.

Примечание: свойства расширения не могут хранить данные, поэтому они должны быть только для получения.

Добавить функцию расширения

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

```

fun type name . function name ( parameters ) : return type {
    function body
}

```


Вы добавите функцию расширения, которая будет выводить прогресс викторины в виде прогресс-бара. Вы выведете прогресс-бар в ретро-стиле с помощью текста!

- Добавьте функцию расширения к объекту StudentProgress под названием printProgressBar(). Функция не должна принимать никаких параметров и не должна иметь возвращаемого значения.

```

fun Quiz.StudentProgress.printProgressBar() {
}


```

- С помощью функции repeat() выведите символ  на который вы ответили определенное количество раз. Эта затененная часть индикатора выполнения представляет собой количество ответов на вопросы. Используйте print(), потому что вам не нужна новая строка после каждого символа.

```

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

```

- С помощью функции repeat() выведите символ  количество раз равное разнице между общим количеством и количеством ответов. Эта затененная часть представляет собой оставшиеся вопросы на панели процесса.

```

fun Quiz.StudentProgress.printProgressBar() {
    repeat(Quiz.answered) { print("■") }
    repeat(Quiz.total - Quiz.answered) { print("□") }
}

```

- Выведите новую строку с помощью функции println() без аргументов, а затем выведите progressText.

```

fun Quiz.StudentProgress.printProgressBar() {
    repeat(Quiz.answered) { print("■") }
    repeat(Quiz.total - Quiz.answered) { print("□") }
    println()
}

```

```
println(Quiz.progressText)
}
```

- Обновите код в `main()`, чтобы вызвать `printProgressBar()`.

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

- Запустите свой код, чтобы проверить результат.



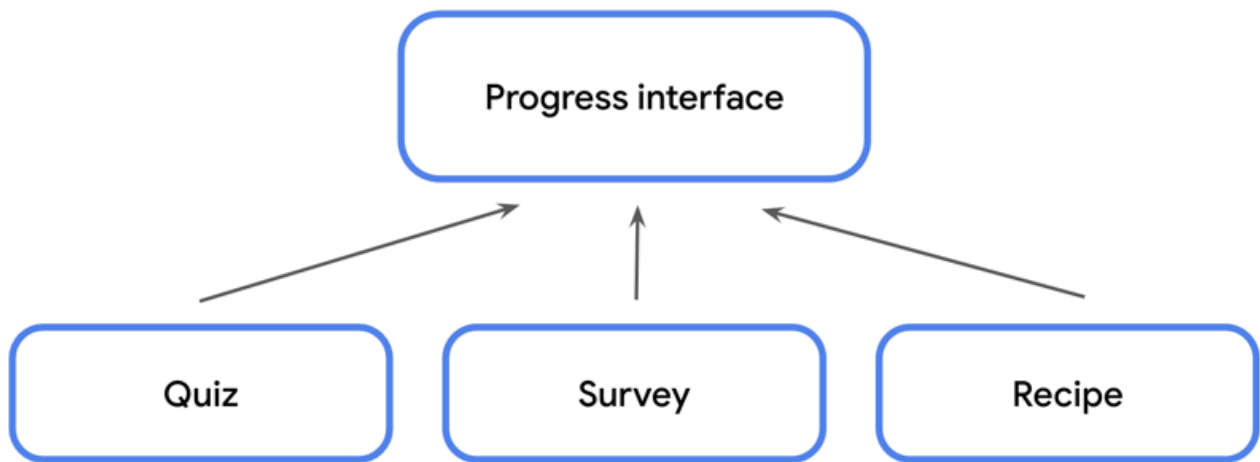
3 of 10 answered.

Обязательно ли делать все это? Конечно же, нет. Однако возможность расширения свойств и методов дает вам больше возможностей для раскрытия своего кода перед другими разработчиками. Использование точечного синтаксиса в других типах может сделать ваш код более легким для чтения как для вас, так и для других разработчиков.

Переписывание функций расширения с помощью интерфейсов

На предыдущей странице вы видели, как добавить свойства и методы в объект `StudentProgress`, не добавляя в него код напрямую, используя **свойства расширения** и **функции расширения**. Хотя это отличный способ добавить функциональность в один класс, который уже определен, расширение класса не всегда необходимо, если у вас есть доступ к исходному коду. Бывают ситуации, когда вы не знаете, какой должна быть реализация, а только то, что определенный метод или свойство должны существовать. 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.

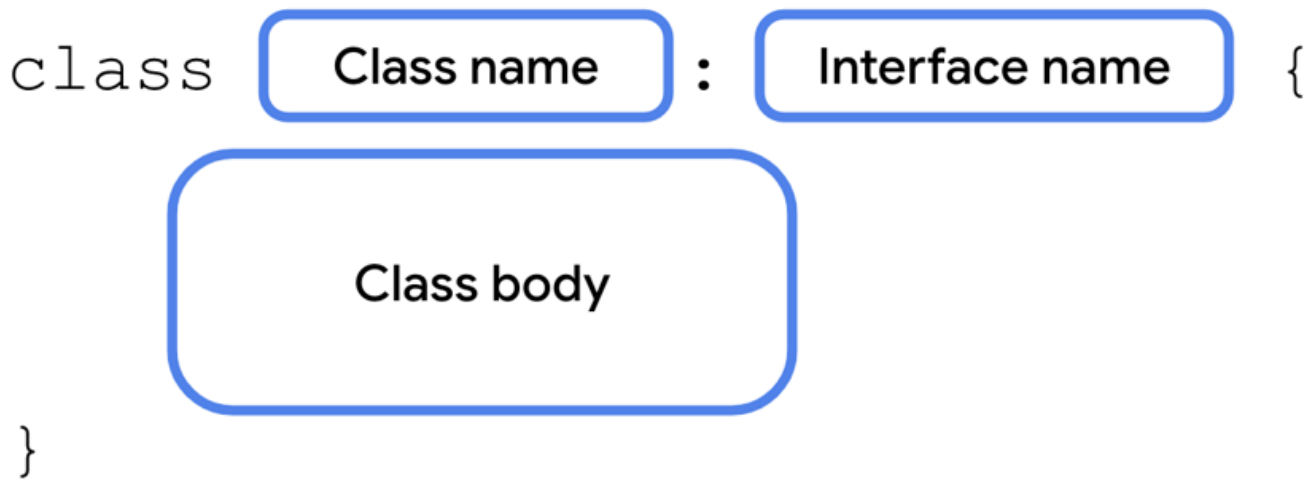
Например, в дополнение к викторинам у вас есть классы для опросов, шагов в рецепте или любых других упорядоченных данных, в которых может использоваться индикатор выполнения. Вы можете определить нечто, называемое **интерфейсом**, который определяет методы и/или свойства, которые должен содержать каждый из этих классов.



Интерфейс определяется с помощью ключевого слова **interface**, за которым следует имя в верхнем регистре, а затем открывающие и закрывающие фигурные скобки. Внутри фигурных скобок можно определить сигнатуры методов или свойства, доступные только для получения, которые должен реализовать любой класс, соответствующий интерфейсу.

```
interface Interface name {  
  
    Interface body  
  
}
```

Интерфейс - это контракт. Класс, который соответствует интерфейсу, считается расширяющим его. Класс может объявить, что он хочет расширить интерфейс, используя двоеточие (👉), за которым следует пробел, а затем 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 точно такие же методы с точно такой же сигнатурой метода. Если вы каким-либо образом модифицируете интерфейс, например добавляете или удаляете свойства или методы или изменяете сигнатуру метода, компилятор потребует от вас обновить любой класс, расширяющий интерфейс, что позволит сохранить последовательность кода и облегчит его сопровождение.

Интерфейсы позволяют варьировать поведение классов, которые их расширяют. Каждый класс должен сам обеспечить его реализацию.

Давайте посмотрим, как можно переписать прогресс-бар, чтобы он использовал интерфейс, и заставить класс Quiz расширять этот интерфейс.

- Над классом Quiz определите интерфейс с именем ProgressPrintable. Мы выбрали название ProgressPrintable, потому что с его помощью любой класс, который его расширяет, сможет выводить индикатор выполнения.

```
interface ProgressPrintable {  
}
```

В интерфейсе ProgressPrintable определите свойство progressText.

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

Измените объявление класса Quiz, чтобы расширить интерфейс ProgressPrintable.

```
class Quiz : ProgressPrintable {  
    ...  
}
```

- В классе Quiz добавьте свойство progressText типа String, как указано в интерфейсе ProgressPrintable. Поскольку свойство происходит от ProgressPrintable, перед val поставьте ключевое слово override.

```
override val progressText: String
```

- Скопируйте геттер свойства из старого свойства расширения progressText.

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

- Удалите старое свойство расширения progressText.

Код для удаления:

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

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

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

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

```
override fun printProgressBar() {
}
```

- Перенесите 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)
}
```

- Удалите функцию расширения `printProgressBar()`. Теперь эта функция принадлежит классу `Quiz`, который расширяет `ProgressPrintable`.

Код для удаления:

```
fun Quiz.StudentProgress.printProgressBar() {
    repeat(Quiz.answered) { print("█") }
    repeat(Quiz.total - Quiz.answered) { print("░") }
    println()
    println(Quiz.progressText)
}
```

- Обновите код в `main()`. Поскольку функция `printProgressBar()` теперь является методом класса `Quiz`, вам нужно сначала инстанцировать объект `Quiz`, а затем вызвать `printProgressBar()`.

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

- Запустите свой код. Результат не изменился, но ваш код стал более модульным. По мере роста кодовой базы вы сможете легко добавлять классы, соответствующие одному и тому же интерфейсу, чтобы повторно использовать код без наследования от суперкласса.



3 of 10 answered.

Существует множество вариантов использования интерфейсов для структурирования кода, и вы начнете часто встречать их в общих блоках. Ниже приведены примеры интерфейсов, с которыми вы можете столкнуться, продолжая работать с Kotlin.

Ручная инъекция зависимостей. Создайте интерфейс, определяющий все свойства и методы зависимости. Укажите интерфейс в качестве типа данных зависимости (активности, тестового случая и т. д.), чтобы можно было использовать экземпляр любого класса, реализующего этот интерфейс. Это позволит вам менять местами базовые реализации.

Мокинг для автоматизированных тестов. И имитируемый класс, и реальный класс соответствуют одному и тому же интерфейсу.

Доступ к одним и тем же зависимостям в мультиплатформенном приложении Compose. Например, создайте интерфейс, который предоставляет общий набор свойств и методов для Android и настольных компьютеров, даже если базовая реализация отличается для каждой платформы.

Некоторые типы данных в Compose, например `Modifier`, являются интерфейсами. Это позволяет добавлять новые модификаторы без необходимости доступа к исходному коду или его изменения.

Использование функций области видимости для доступа к свойствам и методам класса

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

Одна из таких функций, с которой вы столкнетесь по мере изучения разработки под Android, - это функции **области видимости**. Функции области видимости позволяют вам получить доступ к свойствам и методам класса без необходимости многократного обращения к имени переменной. Что именно это значит?

Давайте рассмотрим пример.

Устранение повторяющихся ссылок на объекты с помощью функций области видимости

Функции области видимости - это функции более высокого порядка, которые позволяют обращаться к свойствам и методам объекта, не обращаясь к его имени. Они называются функциями области видимости, потому что тело переданной функции принимает область видимости объекта, для которого вызывается функция области видимости. Например, некоторые функции области видимости позволяют получить доступ к свойствам и

Существует множество вариантов использования интерфейсов для структурирования кода, и вы начнете видеть, как они часто используются в качестве методов в классе, как если бы функции были определены как метод этого класса. Это может сделать ваш код более читабельным, позволяя опускать имя объекта, когда его включение является излишним.

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

Замена длинных имен объектов с помощью **let()**

Функция `let()` позволяет ссылаться на объект в лямбда-выражении с помощью идентификатора **it**, а не его реального имени. Это поможет вам избежать повторного использования длинного, более описательного имени объекта при обращении к нескольким свойствам. Функция `let()` - это функция расширения, которая может быть вызвана для любого объекта Kotlin с использованием точечной нотации.

Попробуйте получить доступ к свойствам `question1`, `question2` и `question3` с помощью `let()`:

- Добавьте в класс `Quiz` функцию `printQuiz()`.

```
fun printQuiz() {  
  
}
```

- Добавьте следующий код, который выводит текст вопроса `QuestionText`, ответ и сложность. Хотя для вопросов `question1`, `question2` и `question3` используется несколько свойств, каждый раз используется полное имя переменной. Если имя переменной изменится, вам придется обновлять его при каждом использовании.

```
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()
}
```

- Окружите код, обращающийся к свойствам `questionText`, `answer` и `difficulty`, вызовом функции `let()` для вопросов `question1`, `question2` и `question3`. Замените ее 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()
}
```

- Обновите код в `main()`, чтобы создать экземпляр класса `Quiz` с именем `quiz`.

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

- Вызовите `printQuiz()`.

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

- 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 apply()

Одна из замечательных особенностей функций расширения заключается в том, что их можно вызывать на объекте еще до того, как он будет присвоен переменной. Например, функция **apply()** - это функция расширения, которая может быть вызвана на объекте с использованием нотации точки. Функция `apply()` также возвращает ссылку на этот объект, чтобы его можно было сохранить в переменной.

Обновите код в `main()`, чтобы вызвать функцию `apply()`.

- Вызовите `apply()` после закрывающей круглой скобки при создании экземпляра класса `Quiz`. Вы можете опустить круглые скобки при вызове `apply()` и использовать синтаксис лямбды в конце.

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

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

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

- Функция `apply()` возвращает экземпляр класса `Quiz`, но поскольку вы больше нигде его не используете, удалите переменную `quiz`. С функцией `apply()` вам даже не нужна переменная для вызова методов экземпляра `Quiz`.

```
Quiz().apply {  
    printQuiz()  
}
```

- Запустите свой код. Обратите внимание, что вы смогли 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 демонстрируют, как они могут сделать ваш код более лаконичным и избежать повторения одного и того же имени переменной.

Приведенный выше код демонстрирует только два примера, но мы рекомендуем вам сделать закладку и обратиться к документации по функциям области видимости, так как вы столкнетесь с их использованием позже в курсе.

Резюме

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

Generics позволяет передавать классу типы данных в качестве параметров, классы **enum** определяют ограниченный набор возможных значений, а классы данных **data class** помогают автоматически генерировать некоторые полезные методы для классов.

Вы также узнали, как создать **объект-синглтон**, ограниченный одним экземпляром, как сделать его **companion-object** другого класса и как расширить существующие классы новыми свойствами get-only 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 в последующих разделах, когда будете узнавать больше о Kotlin, разработке под Android и Compose. Теперь вы лучше понимаете, как они работают и как они могут улучшить многократное использование и читаемость вашего кода.