

Лекция. Использование классов и объектов в Kotlin

Прежде чем начать

Эта практическая работа научит вас использовать классы и объекты в Kotlin.

Классы представляют собой чертежи, на основе которых можно создавать объекты.

Объект - это экземпляр класса, который состоит из данных, специфичных для этого объекта. Вы можете использовать объекты или экземпляры классов как взаимозаменяемые понятия.

В качестве аналогии представьте, что вы строите дом. Класс похож на проектный план архитектора, также известный как чертеж. Чертеж - это не дом, это инструкция по его строительству. Дом - это реальная вещь, или объект, который строится на основе чертежа.

Как на чертеже дома указывается несколько комнат, и каждая комната имеет свой дизайн и назначение, так и каждый класс имеет свой дизайн и назначение. Чтобы знать, как проектировать классы, вам нужно познакомиться с объектно-ориентированным программированием (ООП) - основой, которая учит заключать данные, логику и поведение в объекты.

ООП помогает упростить сложные проблемы реального мира, разбив их на более мелкие объекты. Существует четыре основных концепции ООП, о каждой из которых вы узнаете:

Инкапсуляция. Обертывание связанных свойств и методов, выполняющих действия над этими свойствами, в класс. Например, рассмотрим ваш мобильный телефон. В нем заключены камера, дисплей, карты памяти и некоторые другие аппаратные и программные компоненты. Вам не нужно беспокоиться о том, как компоненты соединены между собой.

Абстракция. Расширение инкапсуляции. Идея заключается в том, чтобы максимально скрыть внутреннюю логику реализации. Например, чтобы сделать фотографию с помощью мобильного телефона, вам достаточно открыть приложение камеры, направить телефон на сцену, которую вы хотите запечатлеть, и нажать кнопку, чтобы сделать снимок. Вам не нужно знать, как устроено приложение камеры или как на самом деле работает аппаратная часть камеры в вашем мобильном телефоне. Короче говоря, внутренняя механика приложения камеры и то, как мобильная камера делает снимки, абстрагированы, чтобы позволить вам выполнять задачи, которые имеют значение.

Наследование. Позволяет создавать класс на основе характеристик и поведения других классов, устанавливая отношения «родитель-ребенок». Например, различные производители выпускают множество мобильных устройств под управлением Android OS, но пользовательский интерфейс для каждого из них отличается. Другими словами, производители наследуют характеристики Android OS и строят свои настройки поверх них.

Полиморфизм. Это слово является адаптацией греческого корня poly-, что означает много, и -morphism, что означает формы. Полиморфизм - это способность использовать различные объекты одним общим способом. Например, когда вы подключаете Bluetooth-динамики к своему мобильному телефону, телефону достаточно знать, что есть устройство, которое может воспроизводить звук по Bluetooth. Однако существует множество Bluetooth-колонок, и телефону не нужно знать, как работать с каждой из них в отдельности.

Наконец, вы узнаете о делегатах свойств, которые предоставляют многократно используемый код для управления значениями свойств с помощью лаконичного синтаксиса. В этой практической работе вы изучите эти концепции при создании структуры классов для приложения **smart-home**.

Примечание: Умные устройства делают нашу жизнь удобнее и проще. На рынке представлено множество решений для «умного дома», позволяющих управлять «умными» устройствами с помощью смартфона. Одним нажатием кнопки на мобильном устройстве вы можете управлять различными устройствами, такими как «умные» телевизоры, светильники, термостаты и другая бытовая техника.

Предварительные условия

- Как открывать, редактировать и запускать код в IntelliJ IDEA
- Знание основ программирования на Kotlin, включая переменные, функции, а также функции `println()` и `main()`.

Что вы узнаете

- Обзор ООП.
- Что такое классы.
- Как определить класс с помощью конструкторов, функций и свойств.
- Как инстанцировать объект.
- Что такое наследование.
- Разница между отношениями IS-A и HAS-A.
- Как переопределять свойства и функции.
- Что такое модификаторы видимости.
- Что такое делегат и как использовать делегат `by`.

Что вы создадите

Структуру классов «умного дома». Классы, представляющие умные устройства, такие как умный телевизор и умный светильник.

Примечание: написанный вами код не будет взаимодействовать с реальными аппаратными устройствами. Вместо этого вы будете выводить действия в консоль с помощью функции `println()`, чтобы имитировать взаимодействие.

Определение класса

Когда вы определяете класс, вы указываете свойства и методы, которыми должны обладать все объекты этого класса.

Определение класса начинается с ключевого слова **class**, за которым следует имя и набор фигурных скобок. Часть синтаксиса перед открывающей фигурной скобкой также называется заголовком класса. В фигурных скобках можно указать свойства и функции класса. О свойствах и функциях вы узнаете позже. Синтаксис определения класса показан на этой диаграмме:



Это рекомендуемые соглашения по именованию классов:

Вы можете выбрать любое имя класса, но не используйте в качестве имени класса ключевые слова Kotlin, например, ключевое слово `fun`. Имя класса записывается в `PascalCase`, поэтому каждое слово начинается с заглавной буквы и между словами нет пробелов. Например, в `SmartDevice` первая буква каждого слова пишется с заглавной буквы, и между словами нет пробела.

Класс состоит из трех основных частей:

- Свойства. Переменные, задающие атрибуты объектов класса.
- Методы. Функции, которые содержат поведение и действия класса.
- Конструкторы. Специальная функция, создающая экземпляры класса во всей программе, в которой он определен.
-

Это не первый раз, когда вы работаете с классами. На предыдущих практических работах вы узнали о типах данных, таких как `Int`, `Float`, `String` и `Double`. В Kotlin эти типы данных определяются как классы. Когда вы определяете переменную, как показано в этом фрагменте кода, вы создаете объект класса `Int`, который инстанцируется со значением `1`:

```
val number: Int = 1
```

Определите класс `SmartDevice`:

- замените его содержимое на пустую функцию `main()`:

```
fun main() {  
}
```

В строке перед функцией `main()` определите класс `SmartDevice` с телом, включающим `//` пустой комментарий `body`:

```
class SmartDevice {  
    // empty body
```

```
}  
  
fun main() {  
}
```

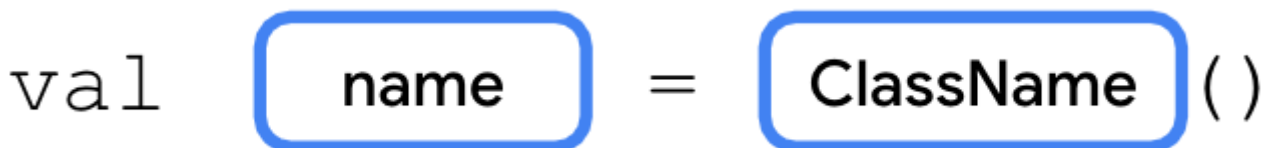
Создание экземпляра класса

Как вы уже узнали, класс - это чертеж объекта. Среда выполнения Kotlin использует класс, или чертеж, для создания объекта определенного типа. С помощью класса `SmartDevice` у вас есть чертеж того, что такое умное устройство. Чтобы в вашей программе появилось реальное «умное» устройство, вам нужно создать экземпляр объекта `SmartDevice`. Синтаксис создания экземпляра начинается с имени класса, за которым следует набор круглых скобок, как показано на этой диаграмме:



ClassName ()

Чтобы использовать объект, вы создаете его и присваиваете переменной, подобно тому, как вы определяете переменную. Ключевое слово `val` используется для создания неизменяемой переменной, а ключевое слово `var` - для изменяемой. За ключевым словом `val` или `var` следует имя переменной, затем оператор присваивания `=`, а затем инстанцирование объекта класса. Синтаксис можно увидеть на этой диаграмме:



val name = ClassName ()

Примечание: Когда вы определяете переменную с ключевым словом `val` для ссылки на объект, сама переменная доступна только для чтения, но объект класса остается изменяемым. Это означает, что вы не можете переназначить другой объект на переменную, но можете изменить состояние объекта при обновлении значений его свойств.

Инстанцируйте класс `SmartDevice` в качестве объекта:

В функции `main()` с помощью ключевого слова `val` создайте переменную с именем `smartTvDevice` и инициализируйте ее как экземпляр класса `SmartDevice`:

```
fun main() {  
    val smartTvDevice = SmartDevice()  
}
```

Определение методов класса

Ранее вы узнали, что:

В определении функции используется ключевое слово `fun`, за которым следует набор круглых скобок и фигурных скобок. Фигурные скобки содержат код - инструкции, необходимые для выполнения задачи.

Вызов функции приводит к выполнению кода, содержащегося в этой функции. Действия, которые может выполнять класс, определяются как функции класса. Например, представьте, что у вас есть умное устройство, умный телевизор или умный светильник, который вы можете включать и выключать с помощью мобильного телефона. В программировании умное устройство транслируется в класс `SmartDevice`, а действия по его включению и выключению представлены функциями `turnOn()` и `turnOff()`, которые обеспечивают включение и выключение.

Синтаксис определения функции в классе идентичен тому, что вы изучали ранее. Единственное отличие заключается в том, что функция помещается в тело класса. Когда вы определяете функцию в теле класса, ее называют методом, и она представляет собой поведение класса. В дальнейшем в этой статье функции будут называться методами всегда, когда они появляются в теле класса.

- Определите метод `turnOn()` и `turnOff()` в классе `SmartDevice`:

В теле класса `SmartDevice` определите метод `turnOn()` с пустым телом:

```
class SmartDevice {  
    fun turnOn() {  
  
    }  
}
```

В теле метода `turnOn()` добавьте оператор `println()` и передайте ему строку «Smart device is turned on.»:

```
class SmartDevice {  
    fun turnOn() {  
        println("Smart device is turned on.")  
    }  
}
```

После метода `turnOn()` добавьте метод `turnOff()`, который выводит строку «Умное устройство выключено»:

```
class SmartDevice {  
    fun turnOn() {  
        println("Smart device is turned on.")  
    }  
  
    fun turnOff() {  
        println("Smart device is turned off.")  
    }  
}
```

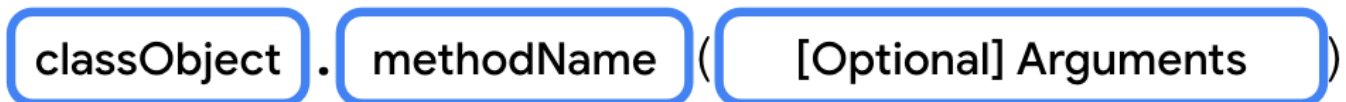
Вызов метода на объекте

Итак, вы определили класс, который служит образцом умного устройства, создали экземпляр класса и присвоили его переменной. Теперь вы используете методы класса SmartDevice для включения и выключения устройства.

Вызов метода в классе похож на то, как вы вызывали другие функции из функции main(). Например, если вам нужно вызвать метод turnOff() из метода turnOn(), вы можете написать что-то похожее на этот фрагмент кода:

```
class SmartDevice {  
    fun turnOn() {  
        // A valid use case to call the turnOff() method could be to turn off the  
        TV when available power doesn't meet the requirement.  
        turnOff()  
        ...  
    }  
    ...  
}
```

Чтобы вызвать метод класса вне класса, начните с объекта класса, за которым следует оператор `.`, имя функции и набор круглых скобок. Если необходимо, в круглых скобках указываются аргументы, требуемые методом. Синтаксис можно увидеть на этой диаграмме:



Вызовите у объекта методы turnOn() и turnOff():

В функции main() в строке после переменной smartTvDevice вызовите метод turnOn():

```
fun main() {  
    val smartTvDevice = SmartDevice()  
    smartTvDevice.turnOn()  
}
```

В строке после метода turnOn() вызовите метод turnOff():

```
fun main() {  
    val smartTvDevice = SmartDevice()  
    smartTvDevice.turnOn()  
    smartTvDevice.turnOff()  
}
```

- Запустите код. На выходе вы получите следующее:

```
Smart device is turned on.  
Smart device is turned off.
```

Определение свойств класса

Вы узнали о переменных, которые являются контейнерами для отдельных фрагментов данных. Вы узнали, как создать переменную, доступную только для чтения, с помощью ключевого слова `val` и переменную, доступную для изменения, с помощью ключевого слова `var`.

В то время как методы определяют действия, которые может выполнять класс, свойства определяют характеристики класса или атрибуты данных. Например, смарт-устройство имеет такие свойства:

- Имя. Название устройства.
- Категория. Тип интеллектуального устройства, например развлекательное, утилитарное или кухонное.
- Состояние устройства. Устройство включено, выключено, находится в сети или в автономном режиме. Устройство считается подключенным к сети, если оно подключено к Интернету. В противном случае оно считается автономным.

Свойства - это, по сути, переменные, которые определяются в теле класса, а не в теле функции. Это означает, что синтаксис для определения свойств и переменных идентичен. Вы определяете неизменяемое свойство с помощью ключевого слова `val` и изменяемое свойство с помощью ключевого слова `var`.

- Реализуйте вышеупомянутые характеристики как свойства класса `SmartDevice`:

В строке перед методом `turnOn()` определите свойство `name` и присвойте ему строку «Android TV»:

```
class SmartDevice {  
  
    val name = "Android TV"  
  
    fun turnOn() {  
        println("Smart device is turned on.")  
    }  
  
    fun turnOff() {  
        println("Smart device is turned off.")  
    }  
}
```

- В строке после свойства `name` определите свойство `category` и присвойте ему строку «Entertainment», а затем определите свойство `deviceStatus` и присвойте ему строку «online»:

```
class SmartDevice {  
  
    val name = "Android TV"  
    val category = "Entertainment"  
    var deviceStatus = "online"  
  
    fun turnOn() {  
        println("Smart device is turned on.")  
    }  
  
    fun turnOff() {  
        println("Smart device is turned off.")  
    }  
}
```

- В строке после переменной smartTvDevice вызовите функцию println(), а затем передайте ей строку «Device name is: \${smartTvDevice.name}»:

```
fun main() {  
    val smartTvDevice = SmartDevice()  
    println("Device name is: ${smartTvDevice.name}")  
    smartTvDevice.turnOn()  
    smartTvDevice.turnOff()  
}
```

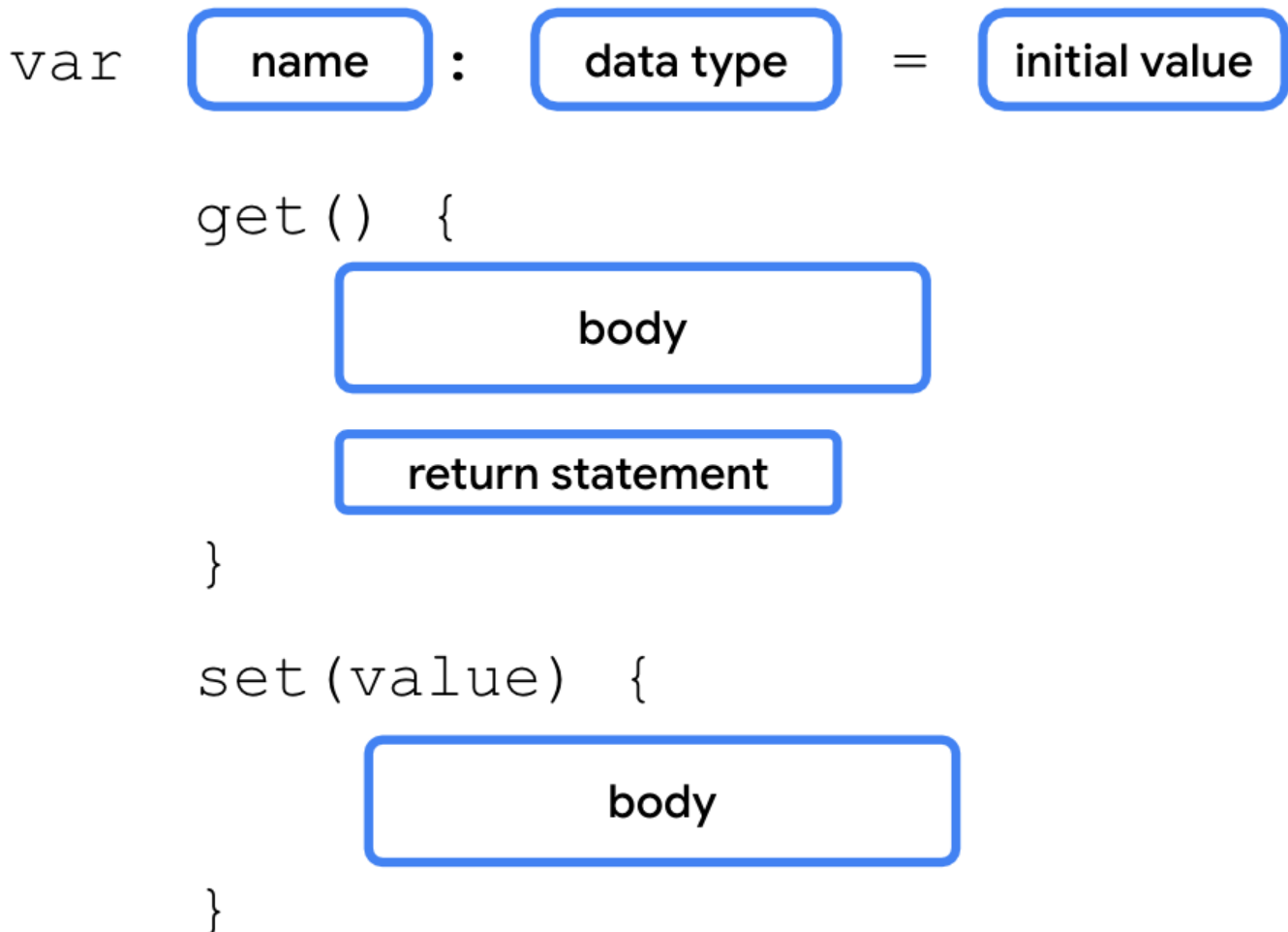
- Запустите код. На выходе вы получите следующее:

```
Device name is: Android TV  
Smart device is turned on.  
Smart device is turned off.
```

Функции Getter и Setter в свойствах

Свойства могут делать больше, чем переменная. Например, представьте, что вы создали структуру класса для представления умного телевизора. Одним из распространенных действий, которые вы выполняете, является увеличение и уменьшение громкости. Чтобы представить это действие в программировании, вы можете создать свойство speakerVolume, которое хранит текущий уровень громкости, установленный на динамике телевизора, но есть диапазон, в котором находится значение громкости. Минимальное значение громкости, которое можно установить, равно 0, а максимальное - 100. Чтобы свойство speakerVolume никогда не превышало 100 или не опускалось ниже 0, можно написать функцию-сеттер. Когда вы обновляете значение свойства, необходимо проверить, находится ли оно в диапазоне от 0 до 100. В качестве другого примера представьте, что требуется, чтобы имя всегда было в верхнем регистре. Вы можете реализовать геттер-функцию для преобразования свойства name в верхний регистр.

Прежде чем углубляться в то, как реализовать эти свойства, необходимо понять полный синтаксис их объявления. Полный синтаксис определения изменяемого свойства начинается с определения переменной, за которым следуют необязательные функции `get()` и `set()`. Синтаксис можно увидеть на этой диаграмме:



Если вы не определяете функции `getter` и `setter` для свойства, компилятор Kotlin сам создает эти функции. Например, если вы используете ключевое слово `var` для определения свойства `speakerVolume` и присваиваете ему значение 2, компилятор автоматически генерирует функции `getter` и `setter`, как показано в этом фрагменте кода:

```
var speakerVolume = 2
    get() = field
    set(value) {
        field = value
    }
```

Вы не увидите этих строк в своем коде, потому что они добавляются компилятором в фоновом режиме.

Полный синтаксис неизменяемого свойства имеет два отличия:

Он начинается с ключевого слова `val`.

Переменные типа `val` - это переменные, доступные только для чтения, поэтому у них нет функции `set()`.

Свойства Kotlin используют опорное поле для хранения значения в памяти. Поле подложки - это, по сути, переменная класса, определенная внутри свойства. Подкрепляющее поле привязано к свойству, что означает, что вы можете получить к нему доступ только через функции свойства `get()` или `set()`.

Чтобы прочитать значение свойства в функции `get()` или обновить его в функции `set()`, необходимо использовать опорное поле свойства. Оно автоматически генерируется компилятором Kotlin и ссылается на него с помощью идентификатора `field`.

Например, когда вы хотите обновить значение свойства в функции `set()`, вы используете параметр функции `set()`, который называется параметром `value`, и присваиваете его переменной `field`, как показано в этом фрагменте кода:

```
var speakerVolume = 2
    set(value) {
        field = value
    }
```

[Предупреждение]: Не используйте имя свойства для получения или установки значения. Например, в функции `set()`, если вы попытаетесь присвоить параметр `value` самому свойству `speakerVolume`, код попадет в бесконечный цикл, поскольку среда выполнения Kotlin попытается обновить значение свойства `speakerVolume`, что вызовет повторный вызов функции `setter`.

Например, чтобы убедиться, что значение, присвоенное свойству `speakerVolume`, находится в диапазоне 0 to 100, можно реализовать функцию `setter`, как показано в этом фрагменте кода:

```
var speakerVolume = 2
    set(value) {
        if (value in 0..100) {
            field = value
        }
    }
```

Функции `set()` проверяют, находится ли значение `Int` в диапазоне от 0 до 100, используя ключевое слово `in`, за которым следует диапазон значений. Если значение находится в ожидаемом диапазоне, значение поля обновляется. Если нет, то значение свойства остается неизменным.

Определите конструктор

Основное назначение конструктора - указать, как создаются объекты класса. Другими словами, конструкторы инициализируют объект и делают его готовым к использованию. Вы сделали это, когда инстанцировали объект. Код внутри конструктора выполняется, когда объект класса инстанцируется. Вы можете определить конструктор с параметрами или без них.

Конструктор по умолчанию

Конструктор по умолчанию - это конструктор без параметров. Вы можете определить конструктор по умолчанию, как показано в этом фрагменте кода:

```
class SmartDevice constructor() {  
    ...  
}
```

Kotlin стремится к краткости, поэтому вы можете убрать ключевое слово `constructor`, если в конструкторе нет аннотаций или модификаторов видимости, о которых вы скоро узнаете. Вы также можете убрать круглые скобки, если конструктор не имеет параметров, как показано в этом фрагменте кода:

```
class SmartDevice {  
    ...  
}
```

Компилятор Kotlin автоматически генерирует конструктор по умолчанию. Вы не увидите автогенерируемый конструктор по умолчанию в вашем коде, потому что он добавляется компилятором в фоновом режиме.

Определите параметризованный конструктор

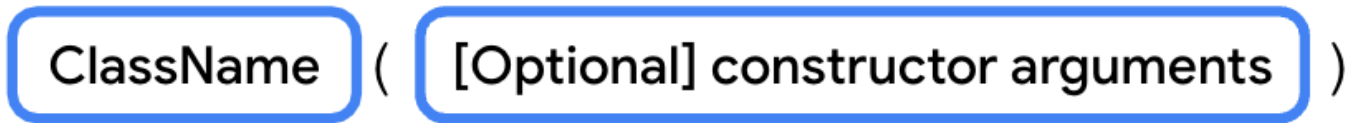
В классе `SmartDevice` свойства `name` и `category` являются неизменяемыми. Вам необходимо убедиться, что все экземпляры класса `SmartDevice` инициализируют свойства `name` и `category`. В текущей реализации значения свойств `name` и `category` жестко закодированы. Это означает, что все смарт-устройства именуется строкой «Android TV» и классифицируются строкой «Entertainment».

Чтобы сохранить неизменяемость, но избежать жестко закодированных значений, используйте параметризованный конструктор для их инициализации:

В классе `SmartDevice` перенесите свойства `name` и `category` в конструктор, не присваивая им значения по умолчанию:

```
class SmartDevice(val name: String, val category: String) {  
  
    var deviceStatus = "online"  
  
    fun turnOn() {  
        println("Smart device is turned on.")  
    }  
  
    fun turnOff() {  
        println("Smart device is turned off.")  
    }  
}
```

Конструктор теперь принимает параметры для настройки своих свойств, поэтому способ инстанцирования объекта для такого класса также меняется. Полный синтаксис инстанцирования объекта можно увидеть на этой диаграмме:



Примечание: Если у класса нет конструктора по умолчанию и вы пытаетесь создать объект без аргументов, компилятор сообщает об ошибке.

Вот представление кода:

```
SmartDevice("Android TV", "Entertainment")
```

Оба аргумента конструктора являются строками. Немного непонятно, какому параметру должно быть присвоено значение. Чтобы исправить это, аналогично тому, как вы передавали аргументы функции, можно создать конструктор с именованными аргументами, как показано в этом фрагменте кода:

```
SmartDevice(name = "Android TV", category = "Entertainment")
```

В Kotlin существует два основных типа конструкторов:

- Первичный конструктор. Класс может иметь только один первичный конструктор, который определяется как часть заголовка класса. Первичный конструктор может быть конструктором по умолчанию или параметризованным конструктором. Первичный конструктор не имеет тела. Это означает, что он не может содержать никакого кода.
- Вторичный конструктор. Класс может иметь несколько вторичных конструкторов. Вы можете определить вторичный конструктор с параметрами или без них. Вторичный конструктор может инициализировать класс и имеет тело, которое может содержать логику инициализации. Если у класса есть первичный конструктор, каждый вторичный конструктор должен инициализировать первичный конструктор. Первичный конструктор можно использовать для инициализации свойств в заголовке класса. Аргументы, переданные конструктору, присваиваются свойствам. Синтаксис определения первичного конструктора начинается с имени класса, за которым следует ключевое слово `constructor` и набор круглых скобок. В круглых скобках указываются параметры первичного конструктора. Если параметров больше одного, определения параметров разделяются запятыми. Полный синтаксис определения первичного конструктора можно увидеть на этой диаграмме:

```
class name constructor ( parameters ) {
    body
}
```

Вторичный конструктор заключен в тело класса, и его синтаксис состоит из трех частей:

Объявление вторичного конструктора

Определение вторичного конструктора начинается с ключевого слова `constructor`, за которым следуют круглые скобки. Если применимо, в круглых скобках указываются параметры, требуемые вторичным конструктором.

- Инициализация первичного конструктора. Инициализация начинается с двоеточия, за которым следует ключевое слово `this` и набор круглых скобок. Если применимо, круглые скобки содержат параметры, требуемые первичным конструктором.
- Тело вторичного конструктора. За инициализацией первичного конструктора следует набор фигурных скобок, которые содержат тело вторичного конструктора. Синтаксис можно увидеть на этой диаграмме:

```
class name ( parameters ) {
    body
    constructor ( parameters ) : this ( Primary constructor parameters ) {
        body
    }
}
```

Например, представьте, что вы хотите интегрировать API, разработанный поставщиком смарт-устройств. Однако API возвращает код состояния типа `Int` для указания начального состояния устройства. API возвращает значение 0, если устройство находится в автономном режиме, и значение 1, если устройство находится в режиме онлайн. При любом другом целочисленном значении статус

считается неизвестным. Вы можете создать дополнительный конструктор в классе SmartDevice, чтобы преобразовать этот параметр statusCode в строковое представление, как показано в этом фрагменте кода:

```
class SmartDevice(val name: String, val category: String) {  
    var deviceStatus = "online"  
  
    constructor(name: String, category: String, statusCode: Int) : this(name,  
category) {  
        deviceStatus = when (statusCode) {  
            0 -> "offline"  
            1 -> "online"  
            else -> "unknown"  
        }  
    }  
    ...  
}
```

Реализация отношений между классами

Наследование позволяет создавать класс на основе характеристик и поведения другого класса. Это мощный механизм, который помогает писать многократно используемый код и устанавливать отношения между классами.

Например, на рынке существует множество «умных» устройств, таких как «умные» телевизоры, «умные» светильники и «умные» выключатели. Когда вы представляете умные устройства в программировании, у них есть общие свойства, такие как имя, категория и статус. У них также есть общее поведение, например возможность включать и выключать их.

Однако способ включения или выключения каждого умного устройства отличается. Например, чтобы включить телевизор, нужно включить дисплей, а затем установить последний известный уровень громкости и канал. С другой стороны, чтобы включить свет, достаточно увеличить или уменьшить яркость.

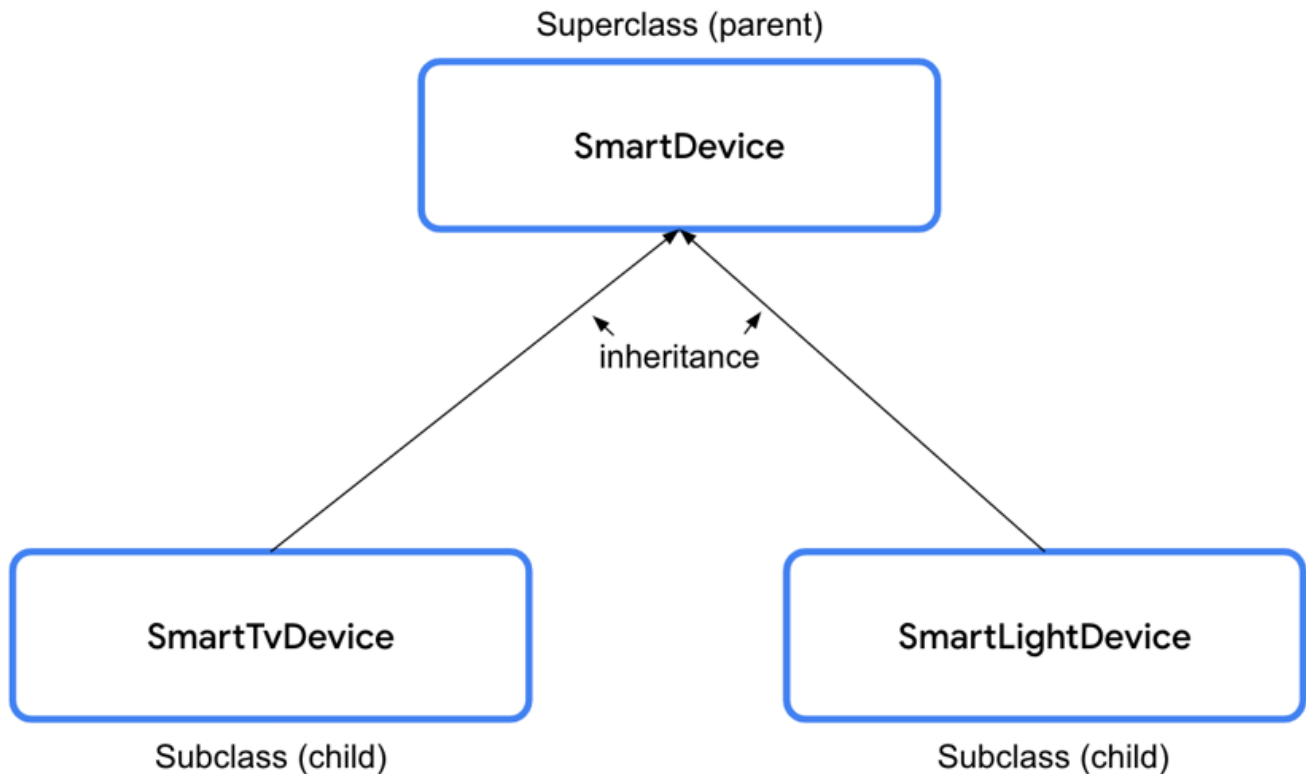
Кроме того, каждое из «умных» устройств имеет больше функций и действий, которые оно может выполнять. Например, с помощью телевизора можно регулировать громкость и переключать каналы. С помощью светильника можно регулировать яркость или цвет.

Одним словом, все умные устройства имеют разные особенности, но при этом обладают некоторыми общими характеристиками. Вы можете либо продублировать эти общие характеристики в каждый класс умного устройства, либо сделать код многократно используемым с помощью наследования.

Для этого необходимо создать родительский класс SmartDevice и определить его общие свойства и поведение. Затем можно создать дочерние классы, такие как SmartTvDevice и SmartLightDevice, которые наследуют свойства родительского класса.

В терминах программирования мы говорим, что классы SmartTvDevice и SmartLightDevice расширяют родительский класс SmartDevice. Родительский класс также называют суперклассом, а дочерний класс -

подклассом. Взаимосвязь между ними можно увидеть на этой диаграмме:



Однако в Kotlin все классы по умолчанию являются финальными, что означает, что вы не можете их расширить, поэтому вам придется определить отношения между ними.

Определение отношения между суперклассом SmartDevice и его подклассами:

В суперклассе SmartDevice добавьте ключевое слово **open** перед ключевым словом **class**, чтобы сделать его расширяемым:

```
open class SmartDevice(val name: String, val category: String) {  
    ...  
}
```

Ключевое слово **open** сообщает компилятору, что этот класс является расширяемым, поэтому теперь другие классы могут расширять его.

Синтаксис создания подкласса начинается с создания заголовка класса, как вы делали до сих пор. За закрывающей скобкой конструктора следует пробел, двоеточие, еще один пробел, имя суперкласса и набор круглых скобок. При необходимости в круглых скобках указываются параметры, требуемые конструктором суперкласса. Синтаксис можно увидеть на этой диаграмме:

```

class Subclass name ( [optional] parameters ) :
    Superclass name ( [optional] parameters ) {

    body

}

```

Создайте подкласс SmartTvDevice, который расширяет суперкласс SmartDevice:

```

class SmartTvDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {
}

```

В определении конструктора SmartTvDevice не указано, являются ли свойства **мутабельными**(var) или **неизменяемыми**(val). Это означает, что параметры deviceName и deviceCategory являются просто параметрами конструктора, а не свойствами класса. Вы не сможете использовать их в классе, а просто передадите их в конструктор суперкласса.

- В тело подкласса SmartTvDevice добавьте свойство speakerVolume, которое вы создали, когда познакомились с функциями getter и setter:

```

class SmartTvDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    var speakerVolume = 2
    set(value) {
        if (value in 0..100) {
            field = value
        }
    }

}

```

- Определите свойство channelNumber, назначенное значению 1, с функцией setter, задающей диапазон 0...200:


```
class SmartTvDevice(deviceName: String, deviceCategory: String) :  
    SmartDevice(name = deviceName, category = deviceCategory) {  
  
    var speakerVolume = 2  
    set(value) {  
        if (value in 0..100) {  
            field = value  
        }  
    }  
  
    var channelNumber = 1  
    set(value) {  
        if (value in 0..200) {  
            field = value  
        }  
    }  
}
```

- Определите метод `increaseSpeakerVolume()`, который увеличивает громкость и выводит строку «Громкость динамика увеличена до `$speakerVolume`.»:

```
class SmartTvDevice(deviceName: String, deviceCategory: String) :  
    SmartDevice(name = deviceName, category = deviceCategory) {  
  
    var speakerVolume = 2  
    set(value) {  
        if (value in 0..100) {  
            field = value  
        }  
    }  
  
    var channelNumber = 1  
    set(value) {  
        if (value in 0..200) {  
            field = value  
        }  
    }  
  
    fun increaseSpeakerVolume() {  
        speakerVolume++  
        println("Speaker volume increased to $speakerVolume.")  
    }  
}
```

- Добавьте метод `nextChannel()`, который увеличивает номер канала и выводит строку «Номер канала увеличился до `$channelNumber`.»:

```

class SmartTvDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    var speakerVolume = 2
        set(value) {
            if (value in 0..100) {
                field = value
            }
        }

    var channelNumber = 1
        set(value) {
            if (value in 0..200) {
                field = value
            }
        }

    fun increaseSpeakerVolume() {
        speakerVolume++
        println("Speaker volume increased to $speakerVolume.")
    }

    fun nextChannel() {
        channelNumber++
        println("Channel number increased to $channelNumber.")
    }
}

```

- В строке после подкласса SmartTvDevice определите подкласс SmartLightDevice, который расширяет суперкласс SmartDevice:

```

class SmartLightDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {
}

```

- В теле подкласса SmartLightDevice определите свойство brightnessLevel, которому присвоено значение 0, с функцией setter, задающей диапазон 0...100:

```

class SmartLightDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    var brightnessLevel = 0
        set(value) {
            if (value in 0..100) {
                field = value
            }
        }
}

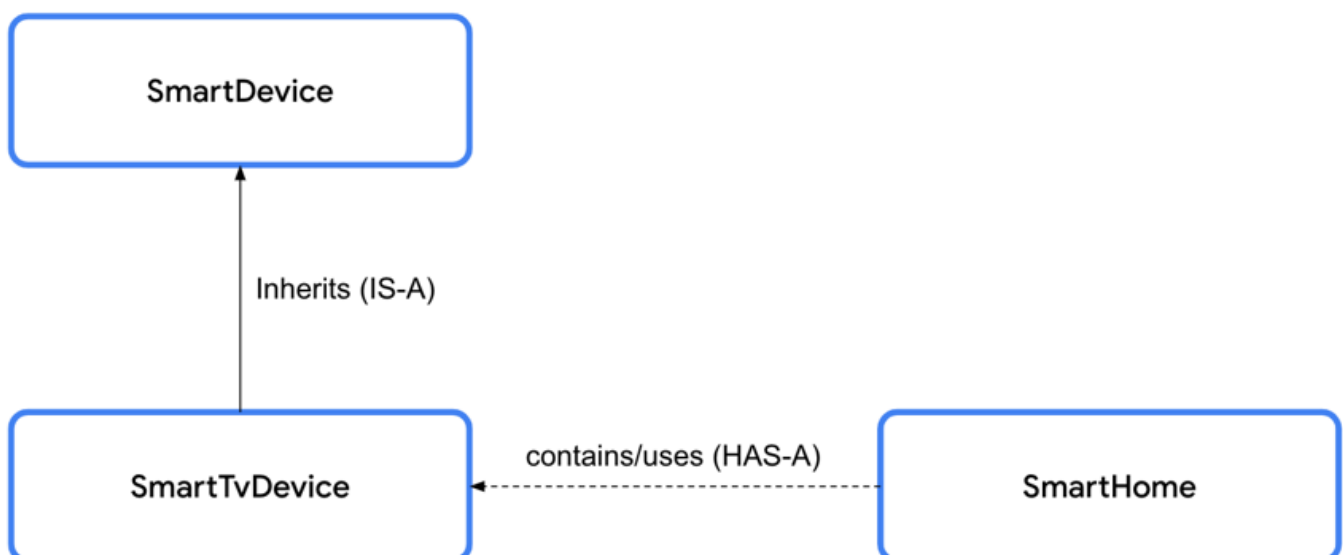
```

- Определите метод `increaseBrightness()`, который увеличивает яркость света и выводит строку «Яркость увеличена до `$brightnessLevel`.»:

```
class SmartLightDevice(deviceName: String, deviceCategory: String) :  
    SmartDevice(name = deviceName, category = deviceCategory) {  
  
    var brightnessLevel = 0  
    set(value) {  
        if (value in 0..100) {  
            field = value  
        }  
    }  
  
    fun increaseBrightness() {  
        brightnessLevel++  
        println("Brightness increased to $brightnessLevel.")  
    }  
}
```

Отношения между классами

Когда вы используете наследование, вы устанавливаете отношения между двумя классами, которые называются отношениями IS-A. Объект также является экземпляром класса, от которого он наследуется. В отношениях HAS-A объект может владеть экземпляром другого класса, не являясь при этом самим экземпляром этого класса. Вы можете увидеть высокоуровневое представление этих отношений на этой диаграмме:



Отношения IS-A

Когда вы указываете отношения IS-A между суперклассом **SmartDevice** и подклассом **SmartTvDevice**, это означает, что все, что может делать суперкласс **SmartDevice**, может делать и подкласс **SmartTvDevice**. Отношение является однонаправленным, поэтому можно сказать, что каждый умный телевизор - это

умное устройство, но нельзя сказать, что каждое умное устройство - это умный телевизор. Представление кода для отношения IS-A показано в этом фрагменте кода:

```
// Smart TV IS-A smart device.
class SmartTvDevice : SmartDevice() {
}
```

Не используйте наследование только для того, чтобы добиться многократного использования кода. Прежде чем принять решение, проверьте, связаны ли два класса друг с другом. Если они демонстрируют какие-то отношения, проверьте, действительно ли они удовлетворяют требованиям IS-A. Спросите себя: «Могу ли я сказать, что подкласс является суперклассом?». Например, Android - это операционная система.

Отношения HAS-A

Отношения HAS-A - это еще один способ указать связь между двумя классами. Например, вы, вероятно, будете использовать смарт-телевизор в своем доме. В этом случае существует связь между «умным» телевизором и домом. Дом **содержит** умное устройство или, другими словами, в доме есть умное устройство. Отношения HAS-A между двумя классами также называются **композицией**.

До сих пор вы создали несколько умных устройств. Теперь вы создаете класс SmartHome, который содержит умные устройства. Класс SmartHome позволяет вам взаимодействовать с умными устройствами.

Для определения класса SmartHome используйте отношение HAS-A:

- Между классом SmartLightDevice и функцией main() определите класс SmartHome:

```
class SmartLightDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    ...

}

class SmartHome {

}

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

- В конструкторе класса SmartHome с помощью ключевого слова val создайте свойство smartTvDevice типа SmartTvDevice:

```
// The SmartHome class HAS-A smart TV device.
class SmartHome(val smartTvDevice: SmartTvDevice) {
```

```
}
```

- В теле класса SmartHome определите метод turnOnTv(), который вызывает метод turnOn() для свойства smartTvDevice:

```
class SmartHome(val smartTvDevice: SmartTvDevice) {  
  
    fun turnOnTv() {  
        smartTvDevice.turnOn()  
    }  
}
```

- В строке после метода turnOnTv() определите метод turnOffTv(), который вызывает метод turnOff() для свойства smartTvDevice:

```
class SmartHome(val smartTvDevice: SmartTvDevice) {  
  
    fun turnOnTv() {  
        smartTvDevice.turnOn()  
    }  
  
    fun turnOffTv() {  
        smartTvDevice.turnOff()  
    }  
}
```

В строке после метода turnOffTv() определите метод increaseTvVolume(), который вызывает метод increaseSpeakerVolume() на свойстве smartTvDevice, а затем определите метод changeTvChannelToNext(), который вызывает метод nextChannel() на свойстве smartTvDevice:

```
class SmartHome(val smartTvDevice: SmartTvDevice) {  
  
    fun turnOnTv() {  
        smartTvDevice.turnOn()  
    }  
  
    fun turnOffTv() {  
        smartTvDevice.turnOff()  
    }  
  
    fun increaseTvVolume() {  
        smartTvDevice.increaseSpeakerVolume()  
    }  
  
    fun changeTvChannelToNext() {
```

```
        smartTvDevice.nextChannel()  
    }  
}
```

В конструкторе класса SmartHome перенесите параметр свойства smartTvDevice в отдельную строку, а затем поставьте запятую:

```
class SmartHome(  
    val smartTvDevice: SmartTvDevice,  
) {  
  
    ...  
  
}
```

В строке после свойства smartTvDevice используйте ключевое слово val, чтобы определить свойство smartLightDevice типа SmartLightDevice:

```
// The SmartHome class HAS-A smart TV device and smart light.  
class SmartHome(  
    val smartTvDevice: SmartTvDevice,  
    val smartLightDevice: SmartLightDevice  
) {  
  
    ...  
  
}
```

В теле SmartHome определите метод turnOnLight(), который вызывает метод turnOn() на объекте smartLightDevice, и метод turnOffLight(), который вызывает метод turnOff() на объекте smartLightDevice:

```
class SmartHome(  
    val smartTvDevice: SmartTvDevice,  
    val smartLightDevice: SmartLightDevice  
) {  
  
    ...  
  
    fun changeTvChannelToNext() {  
        smartTvDevice.nextChannel()  
    }  
  
    fun turnOnLight() {  
        smartLightDevice.turnOn()  
    }  
  
    fun turnOffLight() {
```

```
        smartLightDevice.turnOff()  
    }  
}
```

В строке после метода `turnOffLight()` определите метод `increaseLightBrightness()`, который вызывает метод `increaseBrightness()` для свойства `smartLightDevice`:

```
class SmartHome(  
    val smartTvDevice: SmartTvDevice,  
    val smartLightDevice: SmartLightDevice  
) {  
  
    ...  
  
    fun changeTvChannelToNext() {  
        smartTvDevice.nextChannel()  
    }  
  
    fun turnOnLight() {  
        smartLightDevice.turnOn()  
    }  
  
    fun turnOffLight() {  
        smartLightDevice.turnOff()  
    }  
  
    fun increaseLightBrightness() {  
        smartLightDevice.increaseBrightness()  
    }  
}
```

В строке после метода `increaseLightBrightness()` определите метод `turnOffAllDevices()`, который вызывает методы `turnOffTv()` и `turnOffLight()`:

```
class SmartHome(  
    val smartTvDevice: SmartTvDevice,  
    val smartLightDevice: SmartLightDevice  
) {  
  
    ...  
  
    fun turnOffAllDevices() {  
        turnOffTv()  
        turnOffLight()  
    }  
}
```

Переопределите методы суперкласса из подклассов

Как уже говорилось ранее, несмотря на то, что функции включения и выключения поддерживаются всеми умными устройствами, способ их выполнения различается. Чтобы обеспечить такое поведение, специфичное для конкретного устройства, необходимо **переопределить** методы `turnOn()` и `turnOff()`, определенные в суперклассе.

Переопределить - значит перехватить действие, обычно для того, чтобы взять управление на себя. Когда вы переопределяете метод, метод в подклассе прерывает выполнение метода, определенного в суперклассе, и обеспечивает свое собственное выполнение.

Переопределите методы `turnOn()` и `turnOff()` класса `SmartDevice`:

- В теле суперкласса `SmartDevice` перед ключевым словом `fun` каждого метода добавьте ключевое слово **open**:

```
open class SmartDevice(val name: String, val category: String) {  
  
    var deviceStatus = "online"  
  
    open fun turnOn() {  
        // function body  
    }  
  
    open fun turnOff() {  
        // function body  
    }  
}
```

- В теле класса `SmartLightDevice` определите метод `turnOn()` с пустым телом:

```
class SmartLightDevice(deviceName: String, deviceCategory: String) :  
    SmartDevice(name = deviceName, category = deviceCategory) {  
  
    var brightnessLevel = 0  
    set(value) {  
        if (value in 0..100) {  
            field = value  
        }  
    }  
  
    fun increaseBrightness() {  
        brightnessLevel++  
        println("Brightness increased to $brightnessLevel.")  
    }  
  
    fun turnOn() {  
    }  
}
```


- В теле метода `turnOn()` установите свойство `deviceStatus` в строку «on», установите свойство `brightnessLevel` в значение 2 и добавьте оператор `println()`, а затем передайте ему строку «`$name` включено. Уровень яркости составляет `$brightnessLevel`»:

```
fun turnOn() {
    deviceStatus = "on"
    brightnessLevel = 2
    println("$name turned on. The brightness level is $brightnessLevel.")
}
```

В теле класса `SmartLightDevice` определите метод `turnOff()` с пустым телом:

```
class SmartLightDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    var brightnessLevel = 0
    set(value) {
        if (value in 0..100) {
            field = value
        }
    }

    fun increaseBrightness() {
        brightnessLevel++
        println("Brightness increased to $brightnessLevel.")
    }

    fun turnOn() {
        deviceStatus = "on"
        brightnessLevel = 2
        println("$name turned on. The brightness level is $brightnessLevel.")
    }

    fun turnOff() {
    }
}
```

- В теле метода `turnOff()` установите свойство `deviceStatus` в строку «off», свойство `brightnessLevel` в значение 0 и добавьте оператор `println()`, а затем передайте ему строку «Smart Light выключен»:

```
class SmartLightDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    var brightnessLevel = 0
    set(value) {
        if (value in 0..100) {
            field = value
        }
    }
}
```

```

    }

    fun increaseBrightness() {
        brightnessLevel++
        println("Brightness increased to $brightnessLevel.")
    }

    fun turnOn() {
        deviceStatus = "on"
        brightnessLevel = 2
        println("$name turned on. The brightness level is $brightnessLevel.")
    }

    fun turnOff() {
        deviceStatus = "off"
        brightnessLevel = 0
        println("Smart Light turned off")
    }
}

```

- В подклассе SmartLightDevice перед ключевым словом fun методов turnOn() и turnOff() добавьте ключевое слово **override**:

```

class SmartLightDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    var brightnessLevel = 0
        set(value) {
            if (value in 0..100) {
                field = value
            }
        }

    fun increaseBrightness() {
        brightnessLevel++
        println("Brightness increased to $brightnessLevel.")
    }

    override fun turnOn() {
        deviceStatus = "on"
        brightnessLevel = 2
        println("$name turned on. The brightness level is $brightnessLevel.")
    }

    override fun turnOff() {
        deviceStatus = "off"
        brightnessLevel = 0
        println("Smart Light turned off")
    }
}

```

Ключевое слово **override** сообщает среде выполнения Kotlin, что нужно выполнить код, заключенный в методе, определенном в подклассе.

- В теле класса SmartTvDevice определите метод turnOn() с пустым телом:

```
class SmartTvDevice(deviceName: String, deviceCategory: String) : SmartDevice(name = deviceName, category = deviceCategory) {

    var speakerVolume = 2
    set(value) {
        if (value in 0..100) {
            field = value
        }
    }

    var channelNumber = 1
    set(value) {
        if (value in 0..200) {
            field = value
        }
    }

    fun increaseSpeakerVolume() {
        speakerVolume++
        println("Speaker volume increased to $speakerVolume.")
    }

    fun nextChannel() {
        channelNumber++
        println("Channel number increased to $channelNumber.")
    }

    fun turnOn() {
    }
}
```

- В теле метода turnOn() установите свойство deviceStatus в строку «on» и добавьте оператор println(), а затем передайте ему сообщение «\$name включено. Громкость динамика установлена на \$speakerVolume, а номер канала « + „установлен на \$channelNumber.“:

```
class SmartTvDevice(deviceName: String, deviceCategory: String) : SmartDevice(name = deviceName, category = deviceCategory) {

    ...

    fun turnOn() {
        deviceStatus = "on"
        println(
            "$name is turned on. Speaker volume is set to $speakerVolume and
            channel number is " +

```

```

        "set to $channelNumber."
    )
}

```

- В теле класса SmartTvDevice после метода turnOn() определите метод turnOff() с пустым телом:

```

class SmartTvDevice(deviceName: String, deviceCategory: String) : SmartDevice(name
= deviceName, category = deviceCategory) {

    ...

    fun turnOn() {
        ...
    }

    fun turnOff() {
    }
}

```

- В теле метода turnOff() установите свойство deviceStatus в строку «off» и добавьте оператор println(), а затем передайте ему строку «\$name turned off»:

```

class SmartTvDevice(deviceName: String, deviceCategory: String) : SmartDevice(name
= deviceName, category = deviceCategory) {

    ...

    fun turnOn() {
        ...
    }

    fun turnOff() {
        deviceStatus = "off"
        println("$name turned off")
    }
}

```

- В классе SmartTvDevice перед ключевым словом fun методов turnOn() и turnOff() добавьте ключевое слово override:

```

class SmartTvDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    var speakerVolume = 2
        set(value) {
            if (value in 0..100) {

```

```

        field = value
    }
}

var channelNumber = 1
    set(value) {
        if (value in 0..200) {
            field = value
        }
    }

fun increaseSpeakerVolume() {
    speakerVolume++
    println("Speaker volume increased to $speakerVolume.")
}

fun nextChannel() {
    channelNumber++
    println("Channel number increased to $channelNumber.")
}

override fun turnOn() {
    deviceStatus = "on"
    println(
        "$name is turned on. Speaker volume is set to $speakerVolume and
channel number is " +
        "set to $channelNumber."
    )
}

override fun turnOff() {
    deviceStatus = "off"
    println("$name turned off")
}
}

```

- В функции main() с помощью ключевого слова var определите переменную smartDevice типа SmartDevice, которая инстанцирует объект SmartTvDevice, принимающий аргумент «Android TV» и аргумент «Entertainment»:

```

fun main() {
    var smartDevice: SmartDevice = SmartTvDevice("Android TV", "Entertainment")
}

```

- В строке после переменной smartDevice вызовите метод turnOn() для объекта smartDevice:

```

fun main() {
    var smartDevice: SmartDevice = SmartTvDevice("Android TV", "Entertainment")
    smartDevice.turnOn()
}

```

```
smartDevice.turnOn()  
}
```

- Запустите код. На выходе вы получите следующее:

```
Android TV is turned on. Speaker volume is set to 2 and channel number is set to 1.
```

- В строке после вызова метода turnOn() переназначьте переменную smartDevice, чтобы создать класс SmartLightDevice, который принимает аргумент «Google Light» и аргумент «Utility», а затем вызовите метод turnOn() для ссылки на объект smartDevice:

```
fun main() {  
    var smartDevice: SmartDevice = SmartTvDevice("Android TV", "Entertainment")  
    smartDevice.turnOn()  
  
    smartDevice = SmartLightDevice("Google Light", "Utility")  
    smartDevice.turnOn()  
}
```

- Запустите код. На выходе вы получите следующее:

```
Android TV is turned on. Speaker volume is set to 2 and channel number is set to 1.  
Google Light turned on. The brightness level is 2.
```

Это пример **полиморфизма**. Код вызывает метод turnOn() на переменной типа SmartDevice, и в зависимости от фактического значения переменной могут быть выполнены различные реализации метода turnOn().

Повторное использование кода суперкласса в подклассах с помощью ключевого слова **super**.

При внимательном рассмотрении методов turnOn() и turnOff() можно заметить сходство в том, как обновляется переменная deviceStatus при каждом вызове этих методов в подклассах SmartTvDevice и SmartLightDevice: код дублируется. Вы можете повторно использовать этот код при обновлении статуса в классе SmartDevice.

Чтобы вызвать переопределенный метод суперкласса из подкласса, нужно использовать ключевое слово **super**. Вызов метода из суперкласса аналогичен вызову метода извне класса.

Вместо того чтобы использовать оператор . между объектом и методом, нужно использовать ключевое слово super, которое сообщает компилятору Kotlin, что нужно вызвать метод из суперкласса, а не из подкласса.

Синтаксис вызова метода из суперкласса начинается с ключевого слова `super`, за которым следует оператор `.`, имя функции и набор круглых скобок. Если применимо, в круглых скобках указываются аргументы. Синтаксис можно увидеть на этой диаграмме:

`super.` **functionName** (**[Optional] Arguments**)

- Повторно используйте код из суперкласса `SmartDevice`:
- Удалите операторы `println()` из методов `turnOn()` и `turnOff()` и перенесите дублирующийся код из подклассов `SmartTvDevice` и `SmartLightDevice` в суперкласс `SmartDevice`:

```
open class SmartDevice(val name: String, val category: String) {  
  
    var deviceStatus = "online"  
  
    open fun turnOn() {  
        deviceStatus = "on"  
    }  
  
    open fun turnOff() {  
        deviceStatus = "off"  
    }  
}
```

- Используйте ключевое слово `super` для вызова методов класса `SmartDevice` в подклассах `SmartTvDevice` и `SmartLightDevice`:

```
class SmartTvDevice(deviceName: String, deviceCategory: String) :  
    SmartDevice(name = deviceName, category = deviceCategory) {  
  
    var speakerVolume = 2  
    set(value) {  
        if (value in 0..100) {  
            field = value  
        }  
    }  
  
    var channelNumber = 1  
    set(value) {  
        if (value in 0..200) {  
            field = value  
        }  
    }  
  
    fun increaseSpeakerVolume() {  
        speakerVolume++  
        println("Speaker volume increased to $speakerVolume.")  
    }  
}
```

```

    }

    fun nextChannel() {
        channelNumber++
        println("Channel number increased to $channelNumber.")
    }

    override fun turnOn() {
        super.turnOn()
        println(
            "$name is turned on. Speaker volume is set to $speakerVolume and
channel number is " +
            "set to $channelNumber."
        )
    }

    override fun turnOff() {
        super.turnOff()
        println("$name turned off")
    }
}

```

```

class SmartLightDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    var brightnessLevel = 0
        set(value) {
            if (value in 0..100) {
                field = value
            }
        }

    fun increaseBrightness() {
        brightnessLevel++
        println("Brightness increased to $brightnessLevel.")
    }

    override fun turnOn() {
        super.turnOn()
        brightnessLevel = 2
        println("$name turned on. The brightness level is $brightnessLevel.")
    }

    override fun turnOff() {
        super.turnOff()
        brightnessLevel = 0
        println("Smart Light turned off")
    }
}

```


Переопределение свойств суперкласса из подклассов

Как и методы, вы можете переопределить свойства, выполнив те же действия.

Переопределите свойство deviceType:

В суперклассе SmartDevice в строке после свойства deviceStatus используйте ключевые слова open и val, чтобы определить свойство deviceType, установленное в строку «unknown»:

```
open class SmartDevice(val name: String, val category: String) {  
    var deviceStatus = "online"  
  
    open val deviceType = "unknown"  
    ...  
}
```

В классе SmartTvDevice с помощью ключевых слов override и val определите свойство deviceType, установленное на строку «Smart TV»:

```
class SmartTvDevice(deviceName: String, deviceCategory: String) :  
    SmartDevice(name = deviceName, category = deviceCategory) {  
  
    override val deviceType = "Smart TV"  
  
    ...  
}
```

В классе SmartLightDevice с помощью ключевых слов override и val определите свойство deviceType, установленное на строку «Smart Light»:

```
class SmartLightDevice(deviceName: String, deviceCategory: String) :  
    SmartDevice(name = deviceName, category = deviceCategory) {  
  
    override val deviceType = "Smart Light"  
  
    ...  
}
```

Модификаторы видимости

Модификаторы видимости играют важную роль в достижении **инкапсуляции**:

- В классе они позволяют скрыть свойства и методы от несанкционированного доступа за пределами класса.

- В пакете они позволяют скрыть классы и интерфейсы от несанкционированного доступа за пределами пакета.

Kotlin предоставляет четыре модификатора видимости:

- **public**. Модификатор видимости по умолчанию. Делает объявление доступным везде. Свойства и методы, которые вы хотите использовать за пределами класса, помечаются как **public**.
- **private**. Делает объявление доступным в том же классе или исходном файле. Вероятно, есть свойства и методы, которые используются только внутри класса и которые вы не хотите, чтобы использовали другие классы. Эти свойства и методы можно пометить модификатором видимости **private**, чтобы гарантировать, что другой класс не сможет случайно получить к ним доступ.
- **protected**. Делает объявление доступным для подклассов. Свойства и методы, которые вы хотите использовать в определяющем их классе и подклассах, помечаются модификатором видимости **protected**.
- **internal**. Делает объявление доступным в том же модуле. Модификатор **internal** аналогичен **private**, но вы можете получить доступ к внутренним свойствам и методам извне класса, если они доступны в том же модуле.

Примечание: **Модуль** - это набор исходных файлов и настроек сборки, которые позволяют разделить проект на отдельные функциональные единицы. В вашем проекте может быть один или много модулей. Вы можете независимо собирать, тестировать и отлаживать каждый модуль.

Пакет - это как каталог или папка, в которой сгруппированы связанные классы, в то время как **модуль** - это контейнер для исходного кода, файлов ресурсов и настроек приложения.

Модуль может содержать несколько пакетов.

Когда вы определяете класс, он становится общедоступным и может быть доступен любому пакету, который его импортирует, что означает, что он является общедоступным по умолчанию, если вы не укажете модификатор видимости. Аналогично, когда вы определяете или объявляете свойства и методы в классе, по умолчанию к ним можно получить доступ вне класса через объект класса. Очень важно определить правильную видимость кода, прежде всего для того, чтобы скрыть свойства и методы, доступ к которым не нужен другим классам.

Например, рассмотрим, как автомобиль становится доступным для водителя. Конкретные детали, из которых состоит автомобиль, и его внутреннее устройство по умолчанию скрыты. Автомобиль должен быть максимально интуитивно понятным в управлении. Вы же не хотите, чтобы автомобиль был таким же сложным в управлении, как коммерческий самолет, точно так же, как вы не хотите, чтобы другой разработчик или вы сами в будущем запутались в том, какие свойства и методы класса должны быть использованы.

Модификаторы видимости помогают раскрыть соответствующие части кода для других классов в вашем проекте и гарантировать, что реализация не может быть непреднамеренно использована, что делает код более понятным и менее склонным к ошибкам.

Модификатор видимости следует помещать перед синтаксисом объявления, при объявлении класса, метода или свойства, как показано на этой диаграмме:

modifier

Declaration Syntax

Указание модификатора видимости для свойств

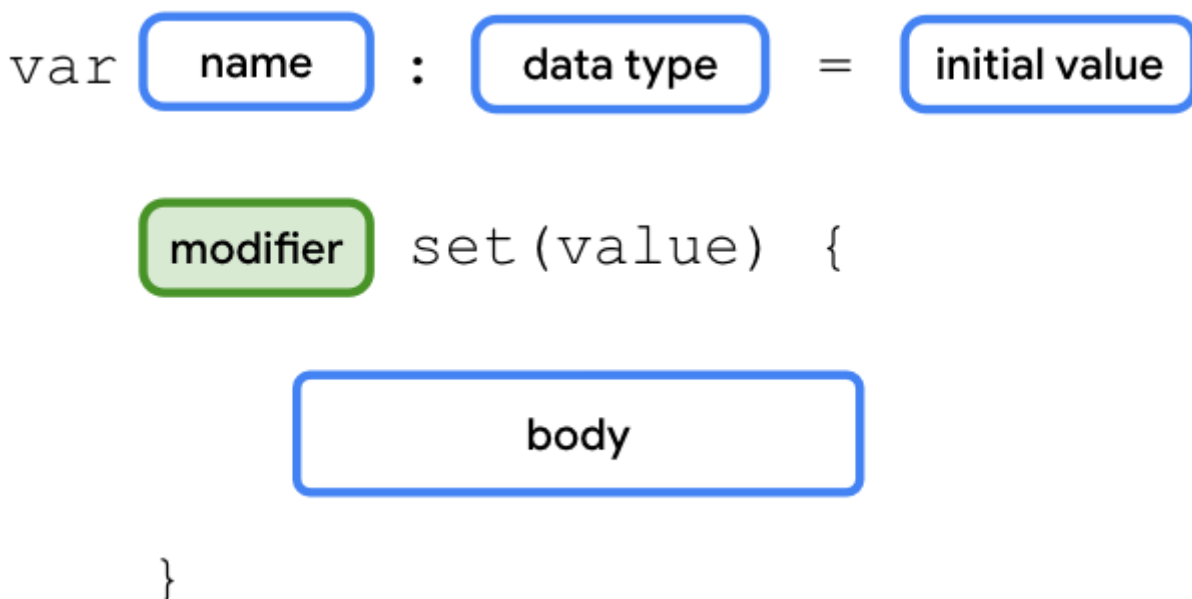
Синтаксис для указания модификатора видимости свойства начинается с модификатора `private`, `protected` или `internal`, за которым следует синтаксис, определяющий свойство. Синтаксис можно увидеть на этой диаграмме:



Например, в этом фрагменте кода показано, как сделать свойство `deviceStatus` приватным:

```
open class SmartDevice(val name: String, val category: String) {  
    ...  
    private var deviceStatus = "online"  
    ...  
}
```

Вы также можете установить модификаторы видимости в функции `setter`. Модификатор помещается перед ключевым словом `set`. Синтаксис можно увидеть на этой диаграмме:



Примечание: Если модификатор видимости для функции getter не совпадает с модификатором видимости для свойства, компилятор сообщает об ошибке.

Для класса SmartDevice значение свойства deviceStatus должно быть доступно для чтения вне класса через объекты класса. Однако только класс и его дочерние объекты должны иметь возможность обновлять или записывать это значение. Чтобы реализовать это требование, необходимо использовать модификатор protected для функции set() свойства deviceStatus.

Используйте модификатор protected для функции set() свойства deviceStatus:

- В свойстве deviceStatus суперкласса SmartDevice добавьте модификатор protected к функции set():

```
open class SmartDevice(val name: String, val category: String) {  
  
    ...  
  
    var deviceStatus = "online"  
        protected set(value) {  
            field = value  
        }  
  
    ...  
}
```

В функции set() вы не выполняете никаких действий или проверок. Вы просто присваиваете параметр value переменной field. Как вы уже узнали, это похоже на реализацию по умолчанию для сеттеров свойств. В этом случае вы можете опустить круглые скобки и тело функции set():

```
open class SmartDevice(val name: String, val category: String) {  
  
    ...  
  
    var deviceStatus = "online"  
        protected set  
  
    ...  
}
```

- В классе SmartHome определите свойство deviceTurnOnCount, установленное на значение 0, с приватной функцией setter:

```
class SmartHome(  
    val smartTvDevice: SmartTvDevice,  
    val smartLightDevice: SmartLightDevice  
) {  
  
    var deviceTurnOnCount = 0
```

```
        private set

        ...
    }
```

- Добавьте свойство `deviceTurnOnCount` с последующим арифметическим оператором `++` в методы `turnOnTv()` и `turnOnLight()`, а затем добавьте свойство `deviceTurnOnCount` с последующим арифметическим оператором `--` в методы `turnOffTv()` и `turnOffLight()`:

```
class SmartHome(
    val smartTvDevice: SmartTvDevice,
    val smartLightDevice: SmartLightDevice
) {

    var deviceTurnOnCount = 0
        private set

    fun turnOnTv() {
        deviceTurnOnCount++
        smartTvDevice.turnOn()
    }

    fun turnOffTv() {
        deviceTurnOnCount--
        smartTvDevice.turnOff()
    }

    ...

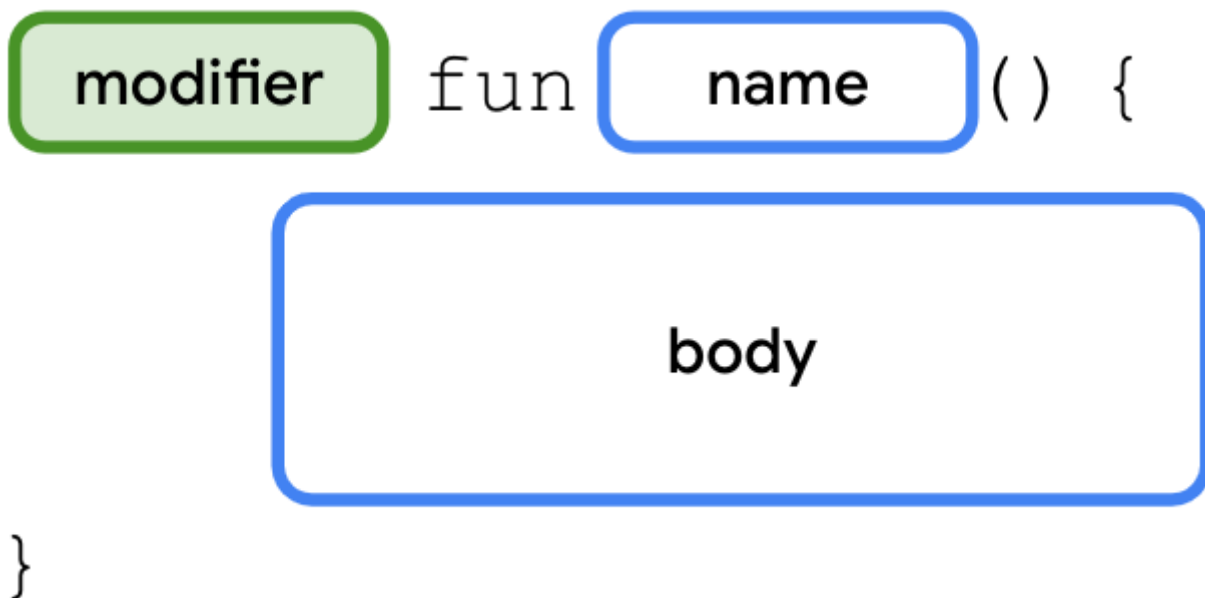
    fun turnOnLight() {
        deviceTurnOnCount++
        smartLightDevice.turnOn()
    }

    fun turnOffLight() {
        deviceTurnOnCount--
        smartLightDevice.turnOff()
    }

    ...
}
```

Модификаторы видимости для методов

Синтаксис для указания модификатора видимости для метода начинается с модификаторов `private`, `protected` или `internal`, за которыми следует синтаксис, определяющий метод. Синтаксис можно увидеть на этой диаграмме:



Например, в этом фрагменте кода показано, как указать защищенный модификатор для метода `nextChannel()` в классе `SmartTvDevice`:

```
class SmartTvDevice(deviceName: String, deviceCategory: String) :  
    SmartDevice(name = deviceName, category = deviceCategory) {  
  
    ...  
  
    protected fun nextChannel() {  
        channelNumber++  
        println("Channel number increased to $channelNumber.")  
    }  
  
    ...  
}
```

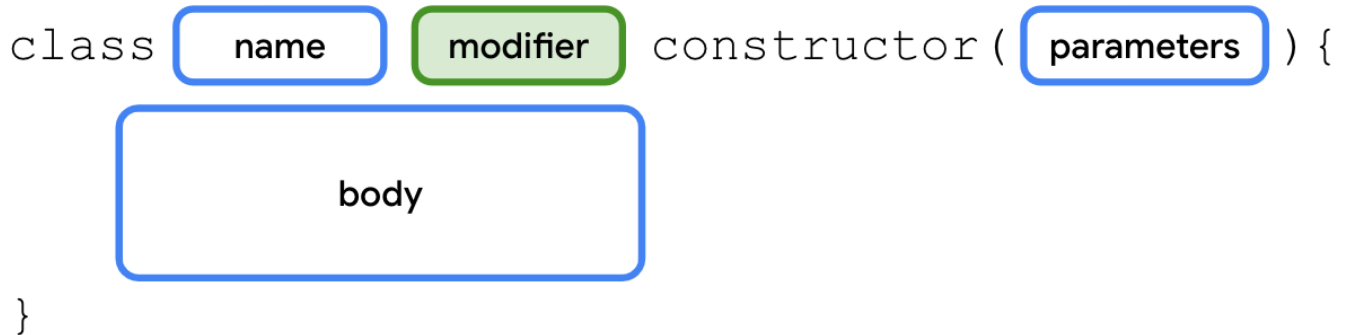
Модификаторы видимости для конструкторов

Синтаксис указания модификатора видимости для конструктора аналогичен определению основного конструктора с некоторыми отличиями:

Модификатор указывается после имени класса, но перед ключевым словом `constructor`.

Если вам нужно указать модификатор для первичного конструктора, необходимо сохранить ключевое слово `constructor` и круглые скобки, даже если нет никаких параметров.

Синтаксис можно увидеть на этой диаграмме:



The diagram illustrates the syntax for a Kotlin class constructor. It shows the keyword `class` followed by a box labeled `name`, then a box labeled `modifier`, then the keyword `constructor`, followed by a box labeled `parameters` enclosed in parentheses, then an opening curly brace `{`, then a large box labeled `body`, and finally a closing curly brace `}`.

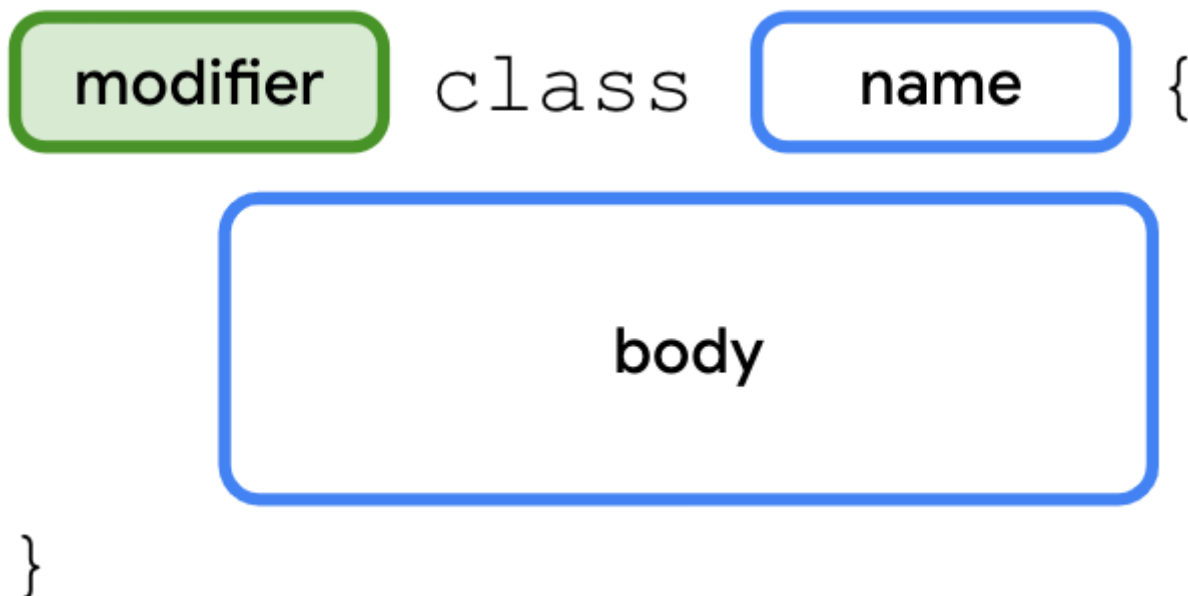
```
class name modifier constructor ( parameters ) {  
    body  
}
```

Например, в этом фрагменте кода показано, как добавить модификатор `protected` в конструктор `SmartDevice`:

```
open class SmartDevice protected constructor (val name: String, val category:  
String) {  
    ...  
}
```

Модификаторы видимости для классов

Синтаксис для указания модификатора видимости для класса начинается с модификаторов `private`, `protected` или `internal`, за которыми следует синтаксис, определяющий класс. Синтаксис можно увидеть на этой диаграмме:



The diagram illustrates the syntax for a Kotlin class with a visibility modifier. It shows a box labeled `modifier`, then the keyword `class`, then a box labeled `name`, then an opening curly brace `{`, then a large box labeled `body`, and finally a closing curly brace `}`.

```
modifier class name {  
    body  
}
```

Например, в этом фрагменте кода показано, как задать внутренний модификатор для класса `SmartDevice`:

```
internal open class SmartDevice(val name: String, val category: String) {
```

```
...

}
```

В идеале вы должны стремиться к строгой видимости свойств и методов, поэтому объявляйте их с модификатором `private` как можно чаще. Если вы не можете сделать их приватными, используйте модификатор `protected`. Если вы не можете сохранить их защищенными, используйте модификатор `internal`. Если вы не можете сохранить их внутренними, используйте модификатор `public`.

Определение подходящих модификаторов видимости

Эта таблица поможет вам определить подходящие модификаторы видимости в зависимости от того, где должны быть доступны свойства или методы класса или конструктора:

Modifier	Accessible in same class	Accessible in subclass	Accessible in same module	Accessible outside module
private	+	-	-	-
protected	+	+	-	-
internal	+	+	+	-
public	+	+	+	+

В подклассе `SmartTvDevice` не следует позволять управлять свойствами `SpeakerVolume` и `ChannelNumber` извне класса. Этими свойствами можно управлять только с помощью методов `increaseSpeakerVolume()` и `nextChannel()`.

Аналогично, в подклассе `SmartLightDevice` свойство `brightnessLevel` должно управляться только методом `increaseLightBrightness()`.

- Добавьте соответствующие модификаторы видимости в подклассы `SmartTvDevice` и `SmartLightDevice`:

В классе `SmartTvDevice` добавьте приватный модификатор видимости к свойствам `speakerVolume` и `channelNumber`:

```
class SmartTvDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    private var speakerVolume = 2
        set(value) {
            if (value in 0..100) {
                field = value
            }
        }

    private var channelNumber = 1
        set(value) {
            if (value in 0..200) {
```



```

        field = value
    }
}
...
}

```

- В классе SmartLightDevice добавьте приватный модификатор к свойству brightnessLevel:

```

class SmartLightDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    ...

    private var brightnessLevel = 0
        set(value) {
            if (value in 0..100) {
                field = value
            }
        }

    ...
}

```

Определение делегатов свойств

В предыдущем разделе вы узнали, что свойства в Kotlin используют базовое поле для хранения своих значений в памяти. Для ссылки на него используется идентификатор поля.

Если посмотреть на код, то можно заметить дублирование кода для проверки того, находятся ли значения в диапазоне для свойств speakerVolume, channelNumber и brightnessLevel в классах SmartTvDevice и SmartLightDevice. Код проверки диапазона в функции setter можно повторно использовать в делегатах. Вместо того чтобы использовать поле и функции getter и setter для управления значением, им управляет делегат.

Синтаксис создания делегатов свойств начинается с объявления переменной, за которым следует ключевое слово **by**, и объекта делегата, который управляет функциями getter и setter для свойства. Синтаксис можно увидеть на этой диаграмме:

```

var name by delegate object

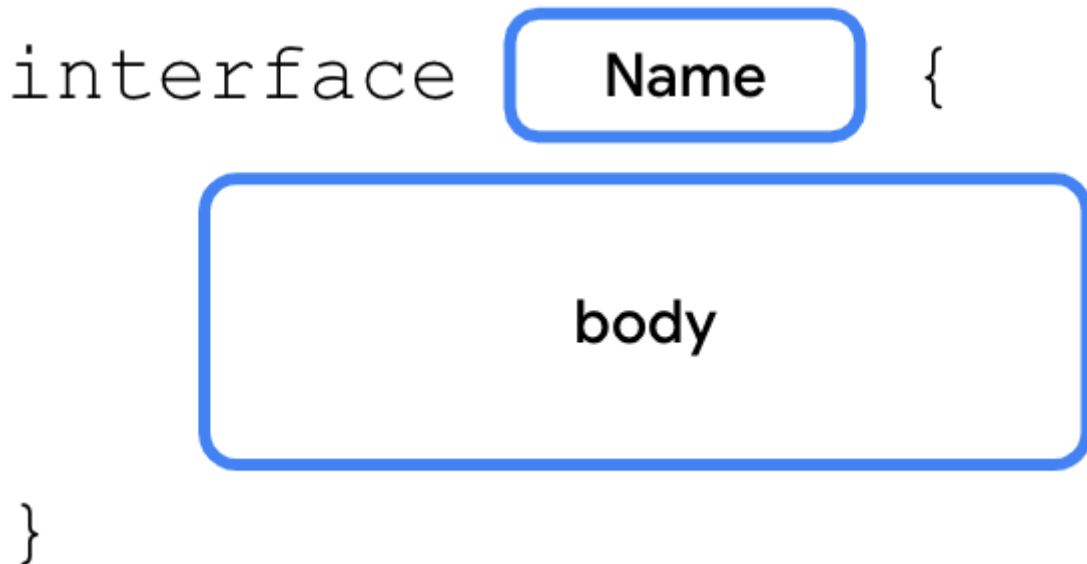
```

Прежде чем реализовать класс, которому можно делегировать реализацию, необходимо познакомиться с интерфейсами.

Интерфейс - это контракт, которого должны придерживаться классы, реализующие его. Он фокусируется на том, что делать, а не на том, как выполнять действие.

Короче говоря, интерфейс помогает вам достичь абстракции.

Например, перед тем как построить дом, вы сообщаете архитектору о том, что вам нужно. Вы хотите спальню, детскую, гостиную, кухню и пару ванных комнат. Короче говоря, вы указываете, чего хотите, а архитектор указывает, как этого добиться. Синтаксис создания интерфейса можно увидеть на этой диаграмме:



Вы уже узнали, как расширять класс и переопределять его функциональность. В интерфейсах класс реализует интерфейс. Класс предоставляет детали реализации методов и свойств, объявленных в интерфейсе. Для создания делегата вы проделаете нечто подобное с интерфейсом `ReadWriteProperty`. Подробнее об интерфейсах вы узнаете в следующем разделе.

Чтобы создать класс делегата для типа `var`, вам нужно реализовать интерфейс `ReadWriteProperty`. Аналогичным образом необходимо реализовать интерфейс `ReadOnlyProperty` для типа `val`.

Создайте делегат для типа `var`:

- Перед функцией `main()` создайте класс `RangeRegulator`, который реализует интерфейс `ReadWriteProperty<Any?, Int>`:

```
class RangeRegulator() : ReadWriteProperty<Any?, Int> {  
  
}  
  
fun main() {  
    ...  
}
```

Не обращайте внимания на угловые скобки и содержимое внутри них. Они представляют собой общие типы, и вы узнаете о них в следующем разделе.

В первичном конструкторе класса `RangeRegulator` добавьте параметр `initialValue`, `private` свойство `minValue` и `private` свойство `maxValue`, все типа `Int`:

```
class RangeRegulator(
    initialValue: Int,
    private val minValue: Int,
    private val maxValue: Int
) : ReadWriteProperty<Any?, Int> {

}
```

- В теле класса RangeRegulator переопределите методы getValue() и setValue():

```
class RangeRegulator(
    initialValue: Int,
    private val minValue: Int,
    private val maxValue: Int
) : ReadWriteProperty<Any?, Int> {

    override fun getValue(thisRef: Any?, property: KProperty<*>): Int {
    }

    override fun setValue(thisRef: Any?, property: KProperty<*>, value: Int) {
    }

}
```

Эти методы выступают в качестве функций получения и установки свойств.

Примечание: KProperty - это интерфейс, который представляет объявленное свойство и позволяет получить доступ к метаданным делегированного свойства. Полезно иметь высокоуровневую информацию о том, что такое KProperty.

В строке перед классом SmartDevice импортируйте интерфейсы ReadWriteProperty и KProperty:

```
import kotlin.properties.ReadWriteProperty
import kotlin.reflect.KProperty

open class SmartDevice(val name: String, val category: String) {
    ...
}

...

class RangeRegulator(
    initialValue: Int,
    private val minValue: Int,
    private val maxValue: Int
) : ReadWriteProperty<Any?, Int> {

    override fun getValue(thisRef: Any?, property: KProperty<*>): Int {
    }

}
```

```

        override fun setValue(thisRef: Any?, property: KProperty<*>, value: Int) {
        }
    }

    ...

```

- В классе RangeRegulator в строке перед методом getValue() определите свойство fieldData и инициализируйте его параметром initialValue:

```

class RangeRegulator(
    initialValue: Int,
    private val minValue: Int,
    private val maxValue: Int
) : ReadWriteProperty<Any?, Int> {

    var fieldData = initialValue

    override fun getValue(thisRef: Any?, property: KProperty<*>): Int {
    }

    override fun setValue(thisRef: Any?, property: KProperty<*>, value: Int) {
    }
}

```

Это свойство служит опорным полем для переменной.

В теле метода getValue() возвратите свойство fieldData:

```

class RangeRegulator(
    initialValue: Int,
    private val minValue: Int,
    private val maxValue: Int
) : ReadWriteProperty<Any?, Int> {

    var fieldData = initialValue

    override fun getValue(thisRef: Any?, property: KProperty<*>): Int {
        return fieldData
    }

    override fun setValue(thisRef: Any?, property: KProperty<*>, value: Int) {
    }
}

```

- В теле метода setValue() проверьте, находится ли присваиваемый параметр значения в диапазоне minValue...maxValue, прежде чем присваивать его свойству fieldData:

```

class RangeRegulator(
    initialValue: Int,
    private val minValue: Int,
    private val maxValue: Int
) : ReadWriteProperty<Any?, Int> {

    var fieldData = initialValue

    override fun getValue(thisRef: Any?, property: KProperty<*>): Int {
        return fieldData
    }

    override fun setValue(thisRef: Any?, property: KProperty<*>, value: Int) {
        if (value in minValue..maxValue) {
            fieldData = value
        }
    }
}

```

- В классе SmartTvDevice используйте класс делегата для определения свойств speakerVolume и channelNumber:

```

class SmartTvDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    override val deviceType = "Smart TV"

    private var speakerVolume by RangeRegulator(initialValue = 2, minValue = 0,
        maxValue = 100)

    private var channelNumber by RangeRegulator(initialValue = 1, minValue = 0,
        maxValue = 200)

    ...
}

```

- В классе SmartLightDevice используйте класс делегата для определения свойства brightnessLevel:

```

class SmartLightDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    override val deviceType = "Smart Light"

    private var brightnessLevel by RangeRegulator(initialValue = 0, minValue = 0,
        maxValue = 100)

    ...
}

```

```
}
```

Протестируйте решение

Вы можете увидеть код решения в этом фрагменте кода:

```
import kotlin.properties.ReadWriteProperty
import kotlin.reflect.KProperty

open class SmartDevice(val name: String, val category: String) {

    var deviceStatus = "online"
        protected set

    open val deviceType = "unknown"

    open fun turnOn() {
        deviceStatus = "on"
    }

    open fun turnOff() {
        deviceStatus = "off"
    }
}

class SmartTvDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    override val deviceType = "Smart TV"

    private var speakerVolume by RangeRegulator(initialValue = 2, minValue = 0,
        maxValue = 100)

    private var channelNumber by RangeRegulator(initialValue = 1, minValue = 0,
        maxValue = 200)

    fun increaseSpeakerVolume() {
        speakerVolume++
        println("Speaker volume increased to $speakerVolume.")
    }

    fun nextChannel() {
        channelNumber++
        println("Channel number increased to $channelNumber.")
    }

    override fun turnOn() {
        super.turnOn()
        println(
```

```

        "$name is turned on. Speaker volume is set to $speakerVolume and
channel number is " +
            "set to $channelNumber."
    )
}

override fun turnOff() {
    super.turnOff()
    println("$name turned off")
}
}

class SmartLightDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    override val deviceType = "Smart Light"

    private var brightnessLevel by RangeRegulator(initialValue = 0, minValue = 0,
maxValue = 100)

    fun increaseBrightness() {
        brightnessLevel++
        println("Brightness increased to $brightnessLevel.")
    }

    override fun turnOn() {
        super.turnOn()
        brightnessLevel = 2
        println("$name turned on. The brightness level is $brightnessLevel.")
    }

    override fun turnOff() {
        super.turnOff()
        brightnessLevel = 0
        println("Smart Light turned off")
    }
}

class SmartHome(
    val smartTvDevice: SmartTvDevice,
    val smartLightDevice: SmartLightDevice
) {

    var deviceTurnOnCount = 0
    private set

    fun turnOnTv() {
        deviceTurnOnCount++
        smartTvDevice.turnOn()
    }

    fun turnOffTv() {
        deviceTurnOnCount--
        smartTvDevice.turnOff()
    }
}

```

```
    }

    fun increaseTvVolume() {
        smartTvDevice.increaseSpeakerVolume()
    }

    fun changeTvChannelToNext() {
        smartTvDevice.nextChannel()
    }

    fun turnOnLight() {
        deviceTurnOnCount++
        smartLightDevice.turnOn()
    }

    fun turnOffLight() {
        deviceTurnOnCount--
        smartLightDevice.turnOff()
    }

    fun increaseLightBrightness() {
        smartLightDevice.increaseBrightness()
    }

    fun turnOffAllDevices() {
        turnOffTv()
        turnOffLight()
    }
}

class RangeRegulator(
    initialValue: Int,
    private val minValue: Int,
    private val maxValue: Int
) : ReadWriteProperty<Any?, Int> {

    var fieldData = initialValue

    override fun getValue(thisRef: Any?, property: KProperty<*>): Int {
        return fieldData
    }

    override fun setValue(thisRef: Any?, property: KProperty<*>, value: Int) {
        if (value in minValue..maxValue) {
            fieldData = value
        }
    }
}

fun main() {
    var smartDevice: SmartDevice = SmartTvDevice("Android TV", "Entertainment")
    smartDevice.turnOn()

    smartDevice = SmartLightDevice("Google Light", "Utility")
}
```



```
smartDevice.turnOn()  
}
```

На выходе получаем следующее:

```
Android TV is turned on. Speaker volume is set to 2 and channel number is set to  
1.  
Google Light turned on. The brightness level is 2.
```

Попробуйте решить эту задачу

- В классе SmartDevice определите метод printDeviceInfo(), который печатает строку «Device name: \$name, category: \$category, type: \$deviceType».
- В классе SmartTvDevice определите метод decreaseVolume(), который уменьшает громкость, и метод previousChannel(), который переходит на предыдущий канал.
- В классе SmartLightDevice определите метод decreaseBrightness(), который уменьшает яркость.
- В классе SmartHome убедитесь, что все действия могут быть выполнены только в том случае, если свойство deviceStatus каждого устройства установлено в строку «on». Также убедитесь, что свойство deviceTurnOnCount обновляется корректно.

После того как вы закончите с реализацией:

- В классе SmartHome определите метод reduceTvVolume(), changeTvChannelToPrevious(), printSmartTvInfo(), printSmartLightInfo() и decreaseLightBrightness().
- Вызовите соответствующие методы из классов SmartTvDevice и SmartLightDevice в классе SmartHome.
- В функции main() вызовите эти добавленные методы для их проверки.

Заключение

Вы узнали, как определять классы и инстанцировать объекты. Вы также узнали, как создавать отношения между классами и создавать делегаты свойств.

Резюме

- Существует четыре основных принципа ООП: инкапсуляция, абстракция, наследование и полиморфизм.
- Классы определяются с помощью ключевого слова class и содержат свойства и методы.
- Свойства похожи на переменные, за исключением того, что свойства могут иметь собственные геттеры и сеттеры.
- Конструктор определяет, как создавать объекты класса.
- Вы можете опустить ключевое слово constructor, когда определяете первичный конструктор.
- Наследование облегчает повторное использование кода.
- Отношение IS-A относится к наследованию.
- Отношение HAS-A относится к композиции.
- Модификаторы видимости играют важную роль в достижении инкапсуляции.

- Kotlin предоставляет четыре модификатора видимости: public, private, protected и internal.
- Делегат свойства позволяет повторно использовать код геттера и сеттера в нескольких классах.