# **Классы и объекты в Котлин**

1. <https://developer.android.com/courses/pathways/android-basics-compose-unit-2-pathway-2>

Классы предоставляют 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 взаимозаменяемо.

В качестве аналогии представьте, что вы строите дом. Класс похож на проектный план архитектора, также известный как чертеж. План — это не дом; это инструкция о том, как построить дом. Дом – это реальная вещь или объект, построенный по чертежу.

Точно так же, как в проекте дома указано несколько комнат, и каждая комната имеет свой дизайн и предназначение, так и каждый класс имеет свой собственный дизайн и назначение. Чтобы знать, как проектировать классы, вам необходимо познакомиться с *объектно-ориентированным программированием (ООП)* — инфраструктурой, которая учит вас заключать данные, логику и поведение в объекты.

ООП помогает вам упростить сложные реальные проблемы до более мелких объектов. Существует четыре основные концепции ООП, о каждой из которых вы узнаете больше позже в этой лаборатории:

* **Инкапсуляция.**Обертывает связанные свойства и методы, которые выполняют действия над этими свойствами в классе. Например, рассмотрим ваш мобильный телефон. Он включает в себя камеру, дисплей, карты памяти и ряд других аппаратных и программных компонентов. Вам не нужно беспокоиться о том, как компоненты подключены внутри.
* **Абстракция.**Расширение инкапсуляции. Идея состоит в том, чтобы максимально скрыть внутреннюю логику реализации. Например, чтобы сделать снимок на мобильный телефон, все, что вам нужно сделать, это открыть приложение камеры, навести 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 оборудование камеры на вашем мобильном телефоне. Короче говоря, внутренняя механика приложения камеры и то, как мобильная камера снимает фотографии, абстрагированы, чтобы вы могли выполнять важные задачи.
* **Наследование.**Позволяет построить класс на основе характеристик и поведения других классов путем установления отношений родитель-потомок. Например, есть разные производители, которые выпускают различные мобильные устройства под управлением ОС Android, но пользовательский интерфейс у каждого из устройств разный. Другими словами, производители наследуют функции ОС Android и создают на их основе свои настройки.
* **Полиморфизм.**Это слово представляет собой адаптацию греческого корня *поли-* , что означает множество, и *-морфизм* , что означает формы. Полиморфизм — это способность использовать разные объекты одним и тем же способом. Например, когда вы подключаете динамик Bluetooth к мобильному телефону, телефону достаточно знать, что существует устройство, которое может воспроизводить звук через Bluetooth. Однако вы можете выбирать из множества динамиков Bluetooth, и вашему телефону не обязательно знать, как работать с каждым из них конкретно.

Наконец, вы узнаете о делегатах свойств, которые предоставляют повторно используемый код для управления значениями свойств с кратким синтаксисом. В этой лабораторной работе вы изучите эти концепции при построении структуры классов для приложения «умный дом».

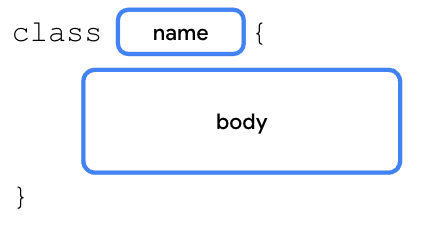
Что вы узнаете

* Обзор ООП.
* Какие есть классы.
* Как определить класс с помощью конструкторов, функций и свойств.
* Как создать экземпляр объекта.
* Что такое наследование.
* Разница между отношениями IS-A и HAS-A.
* Как переопределить свойства и функции.
* Что такое модификаторы видимости.
* Что такое делегат и как его использовать by.

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

Когда вы определяете класс, вы указываете свойства и методы, которые должны иметь 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 вы можете указать свойства и функции класса. Вскоре вы узнаете о свойствах и функциях. Синтаксис определения класса вы можете увидеть на этой диаграмме:



Ниже приведены рекомендуемые соглашения об именах для класса:

* Вы можете выбрать любое имя класса, но не используйте [ключевые слова](https://kotlinlang.org/docs/keyword-reference.html) Kotlin в качестве имени класса, например ключевое слово fun.
* Имя класса пишется 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, а между словами нет пробела.

Класс состоит из трех основных частей:

* **Характеристики.**Переменные, задающие атрибуты объектов класса.
* **Методы.**Функции, содержащие поведение и действия класса.
* **Конструкторы.**Специальная функция-член, которая создает экземпляры класса во всей программе, в которой он определен.

Вы уже не первый раз работаете с классами. На предыдущих занятиях по коду вы узнали о типах данных, таких как Int, Float, Stringи Double. Эти типы данных определяются как классы в Kotlin. Когда вы определяете переменную, как показано в этом фрагменте кода, вы создаете объект класса Int, экземпляр которого создается со  значением 1:

val number: Int = 1

Определите SmartDevice класс:

1. В [Kotlin Playground](https://developer.android.com/training/kotlinplayground" \t "_blank) замените содержимое пустой main() функцией:

fun main() {  
}

1. В строке перед функцией main() определите класс SmartDevice с телом, содержащим комментарий:*//* *empty* *body*

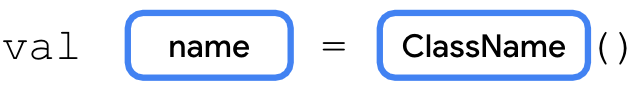
class SmartDevice {  
    // empty body  
}  
  
fun main() {  
}

## [3. Создайте экземпляр класса](https://developer.android.com/codelabs/basic-android-kotlin-compose-classes-and-objects?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-classes-and-objects" \l "2)

Как вы узнали, класс — это проект объекта. Среда выполнения Kotlin использует класс или схему для создания объекта этого конкретного типа. На занятии вы получите представление о том, что такое интеллектуальное устройство. Чтобы использовать в вашей программе настоящее интеллектуальное устройство, вам необходимо создать экземпляр объекта SmartDevice. Синтаксис создания экземпляра начинается с имени класса, за которым следует набор круглых скобок, как вы можете видеть на этой диаграмме:

## 

Чтобы использовать объект, вы создаете объект и присваиваете его переменной аналогично тому, как вы определяете переменную. Вы используете val ключевое слово для создания неизменяемой переменной и var ключевое слово для изменяемой переменной. За ключевым словом val или var следует имя переменной, затем оператор =присваивания, а затем экземпляр объекта класса. Вы можете увидеть синтаксис на этой диаграмме:



**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 переменную другой объект, но можете изменить состояние объекта при обновлении значений его свойств.

Создайте экземпляр класса SmartDevice как объекта:

* В функции main()используйте ключевое слово val , чтобы создать переменную с именем smartTvDevice и инициализировать ее как экземпляр класса SmartDevice:

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

## [4. Определите методы класса](https://developer.android.com/codelabs/basic-android-kotlin-compose-classes-and-objects?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-classes-and-objects#3)

Ранее вы узнали, что:

* В определении функции используется ключевое слово fun, за которым следует набор круглых и фигурных скобок. Фигурные скобки содержат код, который представляет собой инструкции, необходимые для выполнения задачи.
* Вызов функции приводит к выполнению кода, содержащегося в этой функции.

Действия, которые может выполнять класс, определяются как функции в классе. Например, у вас есть интеллектуальное устройство, смарт-телевизор или интеллектуальный светильник, который вы можете включать и выключать с помощью мобильного телефона. Интеллектуальное устройство преобразуется в класс SmartDevice, а действие по его включению и выключению представлено функциями turnOn() и turnOff(), которые включают поведение включения и выключения.

Синтаксис определения функции в классе идентичен тому, что вы узнали ранее. Единственное отличие состоит в том, что функция помещается в тело класса. Когда вы определяете функцию в теле класса, она называется функцией-членом или методом и представляет поведение класса. В оставшейся части этой лаборатории функции называются методами всякий раз, когда они появляются в теле класса.

Определите метод turnOn()and turnOff() в классе SmartDevice:

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

class SmartDevice {  
    fun turnOn() {  
  
    }  
}

1. В тело метода turnOn()добавьте println()оператор и передайте ему "Smart device is turned on."строку:

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

1. После turnOn()метода добавьте turnOff()метод, который печатает "Smart device is turned off."строку:

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

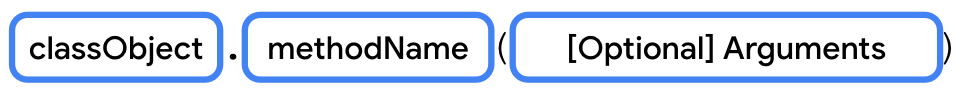
## Вызов метода ****объекта****

Итак, вы определили класс, который служит основой для интеллектуального устройства, создали экземпляр класса и присвоили этот экземпляр переменной. Теперь вы используете SmartDevice методы класса для включения и выключения устройства.

Вызов метода в классе аналогичен тому, как вы вызывали другие функции из main()функции в предыдущей лаборатории кода. Например, если вам нужно вызвать turnOff()метод из turnOn()метода, 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 следуют .оператор, имя функции и набор круглых скобок. Если применимо, круглые скобки содержат аргументы, необходимые методу. Вы можете увидеть синтаксис на этой диаграмме:



Вызовите методы turnOn()и turnOff()объекта:

1. В main()функции в строке после smartTvDeviceпеременной вызовите turnOn()метод:

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

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

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

1. Запустите код.

Результат следующий:

Smart device is turned on.

Smart device is turned off.

## [5. Определите свойства класса](https://developer.android.com/codelabs/basic-android-kotlin-compose-classes-and-objects?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-classes-and-objects#4)

В модуле 1 вы узнали о переменных, которые представляют собой контейнеры для отдельных фрагментов данных. Вы узнали, как создать переменную, доступную только для чтения, с val ключевым словом и изменяемую переменную с var ключевым словом.

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

* **Имя.**Название устройства.
* **Категория.**Тип интеллектуального устройства, например развлекательного, служебного или кулинарного.
* **Состояние устройства** . Независимо от того, включено ли устройство, выключено, онлайн или офлайн. Устройство считается подключенным к Интернету, когда оно подключено к Интернету. В противном случае это считается оффлайн.

Свойства — это, по сути, переменные, которые определены в теле класса, а не в теле функции. Это означает, что синтаксис определения свойств и переменных идентичен. Вы определяете неизменяемое свойство с помощью valключевого слова и изменяемое свойство с помощью varключевого слова.

Реализуйте вышеупомянутые характеристики как свойства класса SmartDevice:

1. В строке перед методом turnOn()определите свойство name и присвойте его строке "Android TV":

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

1. В строке после свойства name определите свойство category и присвойте его строке "Entertainment", а затем определите свойство deviceStatus и присвойте его строке "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.")  
    }  
}

1. В строке после  переменной smartTvDevice вызовите функцию println()и передайте ей "Device name is: ${smartTvDevice.name}"строку:

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

1. Запустите код.

Результат следующий:

Device name is: Android TV

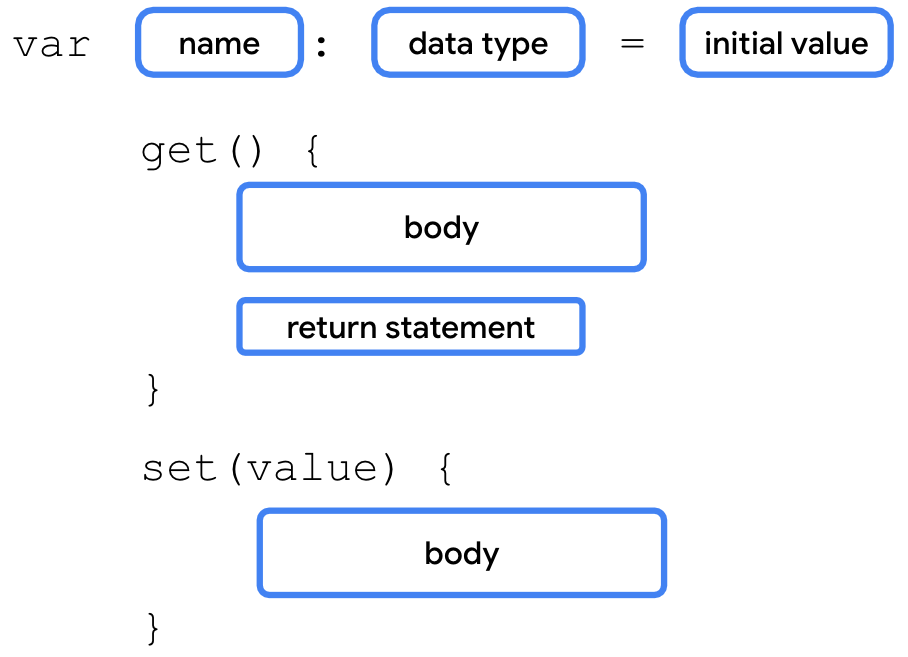
Smart device is turned on.

Smart device is turned off.

## Функции получения и установки в свойствах

Свойства могут делать больше, чем переменная. Например, представьте, что вы создаете структуру классов для представления смарт-телевизора. Одним из распространенных действий, которые вы выполняете, является увеличение и уменьшение громкости. Чтобы представить это действие в программировании, вы можете создать свойство с именем speakerVolume, которое содержит текущий уровень громкости, установленный на динамике телевизора, но существует диапазон, в котором находится значение громкости. Минимальный объем, который можно установить, равен 0, а максимальный — 100. Чтобы гарантировать, что speakerVolumeзначение свойства никогда не превысит 100 и не упадет ниже 0, вы можете написать функцию установки . Когда вы обновляете значение свойства, вам необходимо проверить, находится ли значение в диапазоне от 0 до 100. В качестве другого примера представьте, что существует требование, чтобы имя всегда было в верхнем регистре. Вы можете реализовать функцию получения , чтобы преобразовать nameсвойство в верхний регистр.

Прежде чем углубляться в реализацию этих свойств, вам необходимо понять полный синтаксис их объявления. Полный синтаксис определения изменяемого свойства начинается с определения переменной, за которой следуют необязательные функции get()и set(). Вы можете увидеть синтаксис на этой диаграмме:



Если вы не определяете функции получения и установки для свойства, компилятор Kotlin создает функции внутри себя.

Например, если вы используете ключевое слово var для определения  свойства speakerVolume и присвоения ему значения 2, компилятор автоматически генерирует функции получения и установки, как вы можете видеть в этом фрагменте кода:

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

Вы не увидите эти строки в своем коде, поскольку они добавляются компилятором в фоновом режиме.

Полный синтаксис неизменяемого свойства имеет два отличия:

* Оно начинается с ключевого слова val.
* Переменные типа val доступны только для чтения, поэтому у них нет set() функций.

Свойства Kotlin используют резервное поле для хранения значения в памяти. Резервное поле — это, по сути, переменная класса, определенная внутри свойств. Область резервного поля ограничена свойством, а это означает, что вы можете получить к нему доступ только через функции свойства get()или set().

Чтобы прочитать значение свойства в  функции get() или обновить значение в функции set() , вам необходимо использовать резервное поле свойства . Оно автоматически генерируется компилятором Kotlin и на него ссылается идентификатор .field

Например, если вы хотите обновить значение свойства в  функции set(), вы используете параметр этой функции - value, и присваиваете его переменной field.

Например::

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 войдет в бесконечный цикл, поскольку среда выполнения Kotlin пытается обновить значение свойства speakerVolume, что вызывает повторный вызов функции установки.

Пример:

Чтобы гарантировать, что значение, присвоенное свойству speakerVolume, находится в диапазоне от 0 до 100, вы можете реализовать функцию установки , как показано в этом фрагменте кода:

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

Функция set() проверяет, находится ли значение в диапазоне от 0 до 100, используя ключевое слово in, за которым следует диапазон значения. Если значение находится в ожидаемом диапазоне, поле field обновляется. В противном случае стоимость имущества остается неизменной.

Вы включаете это свойство в класс в разделе «Реализация связи между классами» этой лаборатории кода, поэтому вам не нужно сейчас добавлять функцию установки в код.

## [6. Определите конструктор](https://developer.android.com/codelabs/basic-android-kotlin-compose-classes-and-objects?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-classes-and-objects#5)

Основная цель конструктора — указать, как создаются объекты класса. Другими словами, конструкторы инициализируют объект и подготавливают его к использованию. Вы сделали это, когда создали экземпляр объекта. Код внутри конструктора выполняется при создании экземпляра объекта класса. 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 быть кратким, поэтому вы можете удалить  ключевое слово constructor, если в конструкторе нет аннотаций или модификаторов видимости, о которых вы скоро узнаете. Вы также можете удалить круглые скобки, если у конструктора нет параметров, как показано в этом фрагменте кода:

class SmartDevice {  
    ...  
}

Компилятор Kotlin автоматически генерирует конструктор по умолчанию. Вы не увидите автоматически созданный конструктор по умолчанию в своем коде, поскольку он добавляется компилятором в фоновом режиме.

## Определить 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 запрограммированы. Это означает, что все интеллектуальные устройства именуются с помощью этой строки "Android TV" и классифицируются с помощью этой строки  "Entertainment".

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

* В классе SmartDevice переместите свойства name и category в конструктор, не присваивая значений 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 создания экземпляра объекта для такого класса также меняется. Полный синтаксис создания экземпляра объекта вы можете увидеть на этой диаграмме:



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

Это представление кода:

SmartDevice("Android TV", "Entertainment")

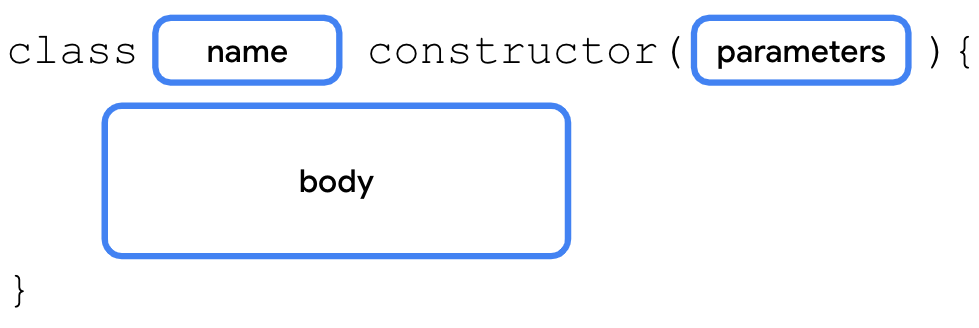
Оба аргумента конструктора являются строками. Немного неясно, какому параметру следует присвоить значение. Чтобы исправить это, аналогично тому, как вы передавали аргументы функции, вы можете создать конструктор с именованными аргументами, как показано в этом фрагменте кода:

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

В Котлине есть два основных типа конструкторов:

* **Первичный конструктор.**Класс может иметь только один основной 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.

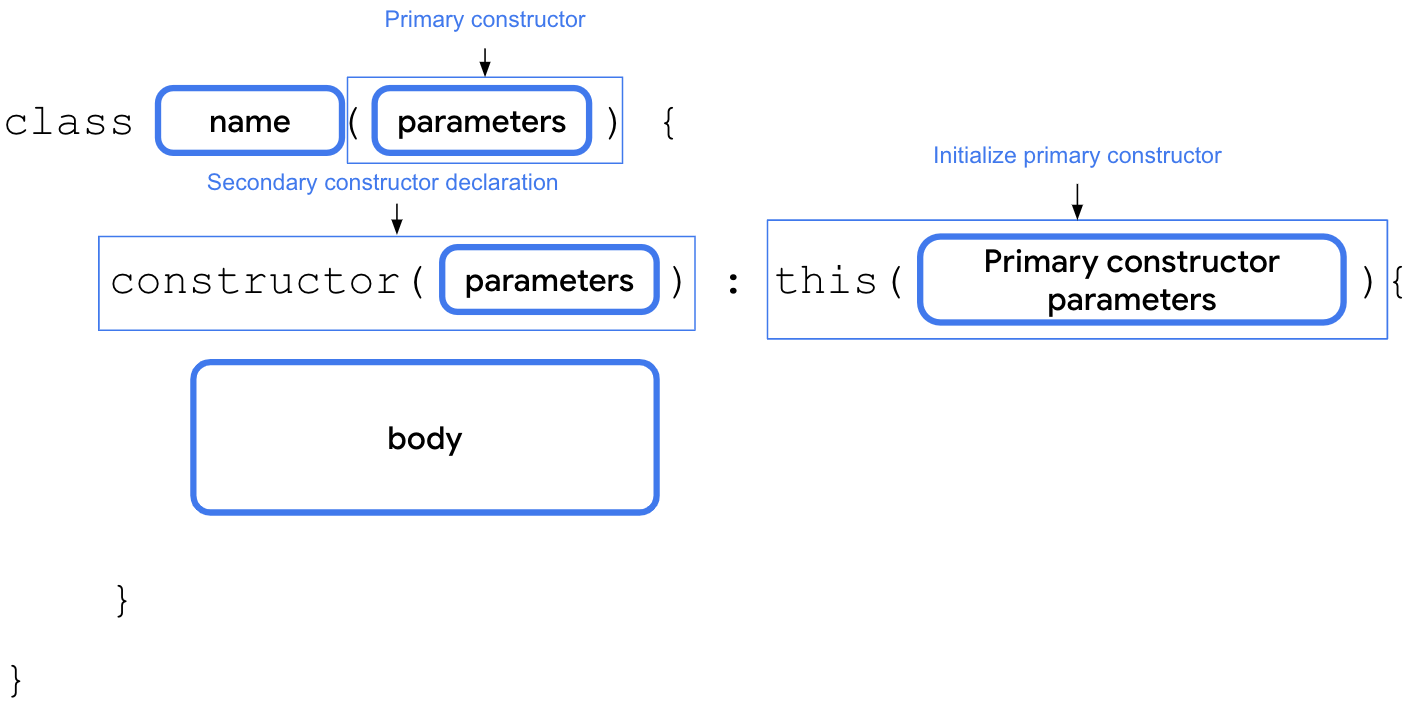
Вы можете использовать основной конструктор для инициализации свойств в заголовке класса. Аргументы, передаваемые конструктору, присваиваются свойствам. Синтаксис определения первичного конструктора начинается с имени класса, за которым следует  ключевое слово constructor и набор круглых скобок. В скобках указаны параметры первичного конструктора. Если имеется более одного параметра, определения параметров разделяются запятыми. Полный синтаксис определения первичного конструктора можно увидеть на этой диаграмме:



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, круглые скобки содержат параметры, необходимые вторичному конструктору.
* **Инициализация первичного конструктора.**Инициализация начинается с двоеточия, за которым следует  ключевое слово this и набор круглых скобок. Если применимо, круглые скобки содержат параметры, необходимые основному конструктору.
* **Вторичное тело конструктора.**За инициализацией первичного конструктора следует набор фигурных скобок, содержащих тело вторичного конструктора.

Вы можете увидеть синтаксис на этой диаграмме:



Например, вы хотите интегрировать API, разработанный поставщиком интеллектуальных устройств. Однако API возвращает код состояния типа Int, указывающий начальное состояние устройства. API возвращает значение 0, если устройство находится в автономном режиме, и значение 1, если устройство находится в сети. Для любого другого целочисленного значения статус считается неизвестным .

Вы можете создать в классе SmartDevice вторичный конструктор  для преобразования этого параметра statusCode в строковое представление, как вы можете видеть 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"  
        }  
    }  
    ...  
}

## [7. Реализовать связь между классами](https://developer.android.com/codelabs/basic-android-kotlin-compose-classes-and-objects?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-classes-and-objects" \l "6)

Наследование позволяет построить класс на основе характеристик и поведения другого класса. Это мощный механизм, который помогает писать код многократного использования и устанавливать связи между классами.

Например, на рынке представлено множество интеллектуальных устройств, таких как смарт-телевизоры, интеллектуальные фонари и интеллектуальные выключатели. Когда вы представляете интеллектуальные устройства в программировании, они имеют некоторые общие свойства, такие как имя, категория и статус. У них также есть общее поведение, например, возможность включать и выключать их.

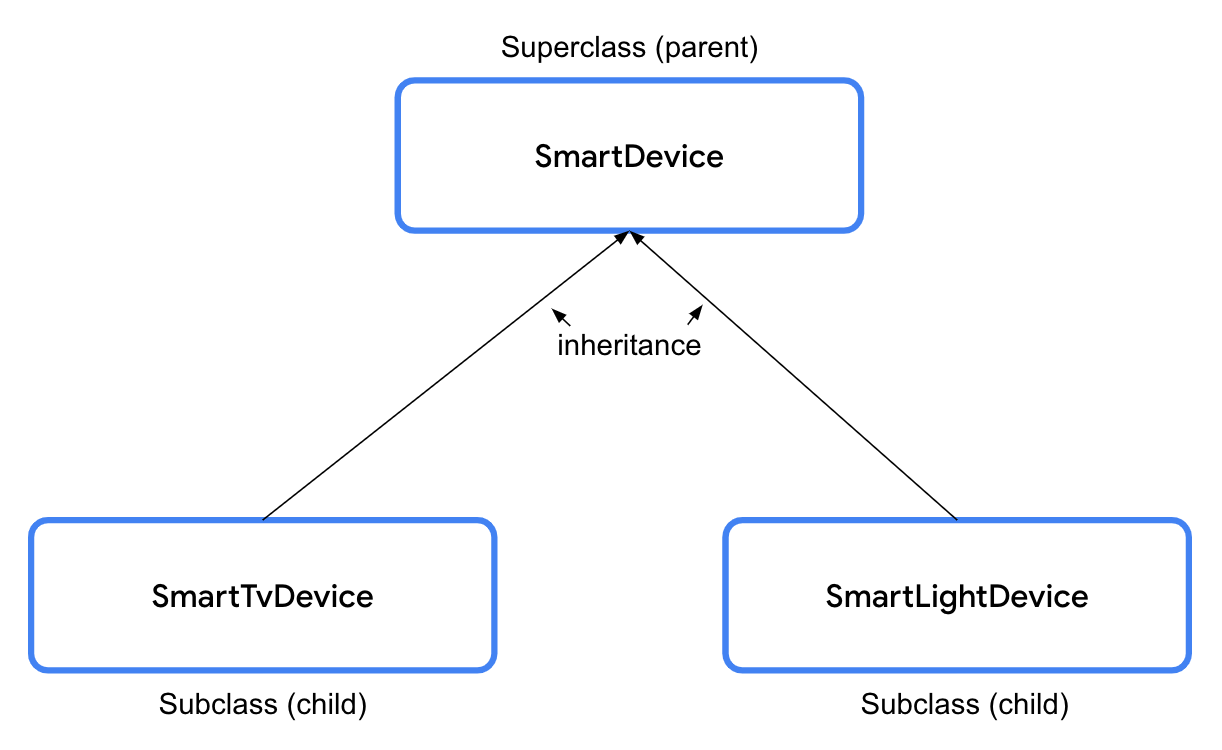
Однако способ включения и выключения каждого смарт-устройства различен. Например, чтобы включить телевизор, вам может потребоваться включить дисплей, а затем настроить последний известный уровень громкости и канал. С другой стороны, чтобы включить свет, вам может потребоваться всего лишь увеличить или уменьшить яркость.

Кроме того, каждое из умных устройств имеет больше функций и действий, которые они могут выполнять. Например, с помощью телевизора можно регулировать громкость и переключать канал. С помощью света вы можете регулировать яркость или цвет.

Короче говоря, все интеллектуальные устройства имеют разные функции, но имеют некоторые общие характеристики. Вы можете либо продублировать эти общие характеристики для каждого класса интеллектуальных устройств, либо сделать код пригодным для повторного использования с помощью наследования.

Для этого вам необходимо создать SmartDeviceродительский класс и определить эти общие свойства и поведение. Затем вы можете создавать дочерние классы, такие как классы SmartTvDeviceи SmartLightDevice, которые наследуют свойства родительского класса.

В терминах программирования мы говорим, что классы SmartTvDeviceи расширяют родительский класс. Родительский класс также называется суперклассом , а дочерний класс — подклассом . Вы можете увидеть связь между ними на этой диаграмме:SmartLightDeviceSmartDevice



Однако в Котлине все классы по умолчанию являются окончательными, а это значит, что вы не можете их расширять, поэтому вам нужно определить отношения между ними.

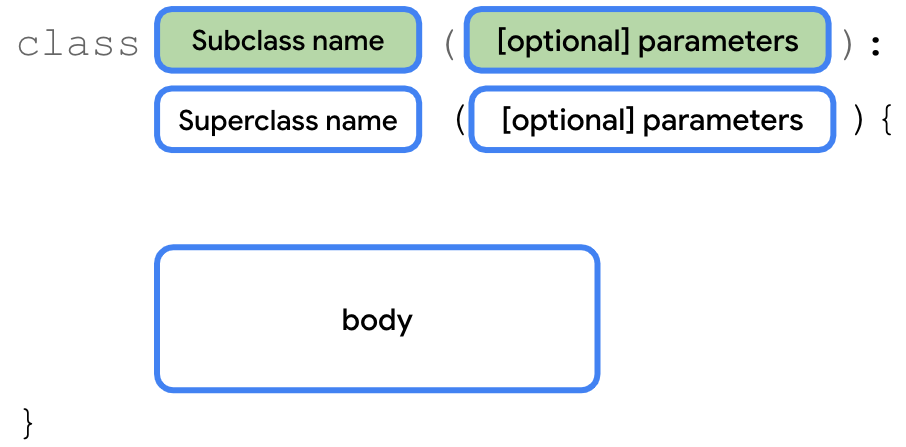
Определите отношения между суперклассом SmartDeviceи его подклассами:

1. В SmartDeviceсуперклассе добавьте openключевое слово перед classключевым словом, чтобы сделать его расширяемым:

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

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

Синтаксис создания подкласса начинается с создания заголовка класса, как вы делали до сих пор. За закрывающей круглой скобкой конструктора следует пробел, двоеточие, еще один пробел, имя суперкласса и набор круглых скобок. При необходимости в круглые скобки включаются параметры, необходимые конструктору суперкласса. Вы можете увидеть синтаксис на этой диаграмме:



1. Создайте SmartTvDeviceподкласс, расширяющий SmartDeviceсуперкласс:

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

В constructorопределении SmartTvDeviceне указывается, являются ли свойства изменяемыми или неизменяемыми. Это означает, что параметры deviceNameи deviceCategoryявляются просто constructorпараметрами, а не свойствами класса. Вы не сможете использовать их в классе, а просто передадите конструктору суперкласса.

1. В SmartTvDeviceтело подкласса добавьте speakerVolumeсвойство, которое вы создали, когда узнали о функциях получения и установки:

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

1. Определите channelNumberсвойство, присвоенное значению, 1с помощью функции установки, которая указывает 0..200диапазон:

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  
            }  
        }  
}

1. Определите increaseSpeakerVolume()метод, который увеличивает громкость и печатает "Speaker volume increased to $speakerVolume."строку:

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.")  
    }   
}

1. Добавьте nextChannel()метод, который увеличивает номер канала и печатает "Channel number increased to $channelNumber." строку:

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.")  
    }  
}

1. В строке после SmartTvDeviceподкласса определите SmartLightDeviceподкласс, расширяющий SmartDeviceсуперкласс:

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

1. В SmartLightDeviceтеле подкласса определите brightnessLevelсвойство, присвоенное значению, 0с помощью функции установки, задающей 0..100диапазон:

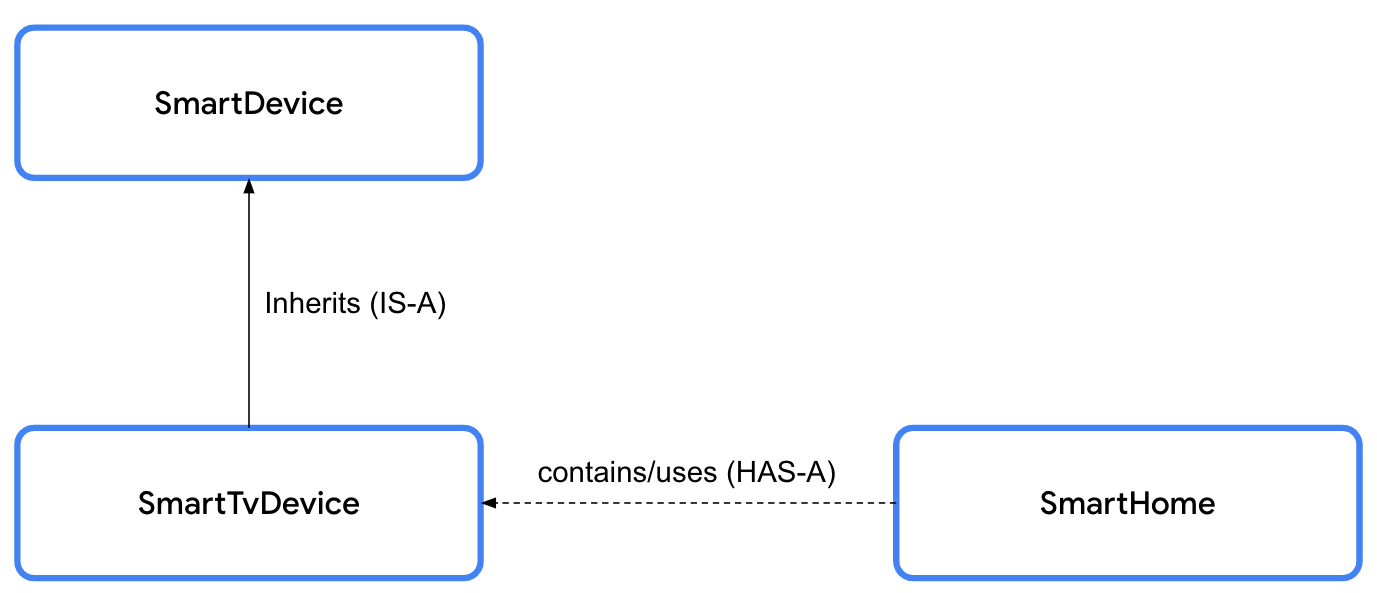
class SmartLightDevice(deviceName: String, deviceCategory: String) :  
    SmartDevice(name = deviceName, category = deviceCategory) {  
  
    var brightnessLevel = 0  
        set(value) {  
            if (value in 0..100) {  
                field = value  
            }  
        }  
}

1. Определите increaseBrightness()метод, который увеличивает яркость света и печатает "Brightness increased to $brightnessLevel."строку:

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 связь между двумя классами в так называемом отношении IS-A . Объект также является экземпляром класса, от которого он наследуется. В отношениях HAS-A объект может владеть экземпляром другого класса, но на самом деле не является экземпляром этого класса. Вы можете увидеть высокоуровневое представление этих отношений на этой диаграмме:



### **Отношения IS-A**

Когда вы указываете отношение IS-A между суперклассом SmartDeviceи SmartTvDeviceподклассом, это означает, что все, что SmartDeviceможет делать суперкласс, SmartTvDeviceможет делать и подкласс. Связь однонаправленная, поэтому можно сказать, что каждый смарт-телевизор — это смарт-устройство, но нельзя сказать, что каждое смарт-устройство — это смарт-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 обеспечения возможности повторного использования кода. Прежде чем принять решение, проверьте, связаны ли эти два класса друг с другом. Если они демонстрируют некоторую связь, проверьте, действительно ли они соответствуют отношениям IS-A. Спросите себя: «Могу ли я сказать, что подкласс — это суперкласс?». Например, Android — это операционная система.

### **HAS-A отношения**

Отношения HAS-A — это еще один способ указать отношения между двумя классами. Например, вы, вероятно, будете использовать Smart TV у себя дома. В данном случае между смарт-телевизором и домом существует связь. В доме есть умное устройство или, другими словами, в доме есть умное устройство. Отношения HAS -A между двумя классами также называются композицией .

На данный момент вы создали пару умных устройств. Теперь вы создаете SmartHomeкласс, содержащий интеллектуальные устройства. Класс SmartHomeпозволяет взаимодействовать со смарт-устройствами.

Используйте отношение HAS-A для определения SmartHomeкласса:

1. Между SmartLightDeviceклассом и main()функцией определите SmartHomeкласс:

class SmartLightDevice(deviceName: String, deviceCategory: String) :  
    SmartDevice(name = deviceName, category = deviceCategory) {  
  
    ...  
  
}  
  
class SmartHome {  
}  
  
fun main() {   
    ...  
}

1. В SmartHomeконструкторе класса используйте valключевое слово для создания smartTvDeviceсвойства SmartTvDeviceтипа:

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

1. В теле класса SmartHomeопределите turnOnTv()метод, который вызывает turnOn()метод свойства smartTvDevice:

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

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

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

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

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

1. В SmartHomeконструкторе класса переместите smartTvDeviceпараметр свойства в отдельную строку, после которой поставьте запятую:

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

1. В строке после smartTvDeviceсвойства используйте valключевое слово для определения smartLightDeviceсвойства SmartLightDeviceтипа:

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

1. В SmartHomeтеле определите turnOnLight()метод, вызывающий turnOn()метод объекта smartLightDevice, и turnOffLight()метод, вызывающий turnOff()метод объекта smartLightDevice:

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

1. В строке после turnOffLight()метода определите increaseLightBrightness()метод, который вызывает increaseBrightness()метод 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()  
    }  
}

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

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

## Переопределить методы суперкласса из подклассов

Как обсуждалось ранее, хотя функции включения и выключения поддерживаются всеми интеллектуальными устройствами, способы их выполнения различаются. Чтобы обеспечить такое поведение, специфичное для устройства, вам необходимо переопределить методы turnOn()и turnOff(), определенные в суперклассе. Переопределить означает перехватить действие, обычно для того, чтобы взять на себя ручное управление. Когда вы переопределяете метод, метод подкласса прерывает выполнение метода, определенного в суперклассе, и обеспечивает собственное выполнение.

Переопределить SmartDeviceкласс turnOn()и turnOff()методы:

1. В теле суперкласса SmartDeviceперед funключевым словом каждого метода добавьте openключевое слово:

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

1. В теле класса SmartLightDeviceопределите turnOn()метод 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() {  
    }  
}

1. В теле метода turnOn()задайте deviceStatusдля свойства строку " on", задайте brightnessLevelдля свойства значение 2и добавьте println()оператор, а затем передайте ему "$name turned on. The brightness level is $brightnessLevel."строку:

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

1. В теле класса SmartLightDeviceопределите turnOff()метод 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() {  
    }  
}

1. В теле метода turnOff()задайте deviceStatusдля свойства строку " off", задайте brightnessLevelдля свойства значение 0и добавьте println()оператор, а затем передайте ему "Smart Light turned off"строку:

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")  
    }  
}

1. В SmartLightDeviceподклассе перед funключевым словом методов turnOn()и turnOff()добавьте overrideключевое слово:

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 overrideслово сообщает среде выполнения Kotlin о необходимости выполнения кода, заключенного в методе, определенном в подклассе.

1. В теле класса SmartTvDeviceопределите turnOn()метод 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() {  
    }  
}

1. В теле метода turnOn()задайте deviceStatusдля свойства строку " on", добавьте println()оператор, а затем передайте ему "$name is turned on. Speaker volume is set to $speakerVolume and channel number is " + "set to $channelNumber."строку:

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."  
        )  
    }  
}

1. В теле класса SmartTvDeviceпосле turnOn()метода определите turnOff()метод с пустым телом:

class SmartTvDevice(deviceName: String, deviceCategory: String) : SmartDevice(name = deviceName, category = deviceCategory) {  
  
    ...  
  
    fun turnOn() {  
        ...  
    }  
  
    fun turnOff() {  
    }  
}

1. В теле метода turnOff()задайте deviceStatusдля свойства строку " off", добавьте println()оператор, а затем передайте ему "$name turned off"строку:

class SmartTvDevice(deviceName: String, deviceCategory: String) : SmartDevice(name = deviceName, category = deviceCategory) {  
  
    ...  
  
    fun turnOn() {  
        ...  
    }  
  
    fun turnOff() {  
        deviceStatus = "off"  
        println("$name turned off")  
    }  
}

1. В SmartTvDeviceклассе перед funключевым словом методов turnOn()и turnOff()добавьте overrideключевое слово:

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")  
    }  
}

1. В main()функции используйте varключевое слово, чтобы определить smartDeviceпеременную SmartDeviceтипа, которая создает экземпляр SmartTvDeviceобъекта, принимающего "Android TV"аргумент и "Entertainment"аргумент:

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

1. В строке после smartDeviceпеременной вызовите turnOn()метод объекта smartDevice:

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

1. Запустите код.

Результат следующий:

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

1. В строке после вызова метода turnOn()переназначьте smartDeviceпеременную для создания экземпляра SmartLightDeviceкласса, который принимает "Google Light"аргумент и "Utility"аргумент, а затем вызовите turnOn()метод по smartDeviceссылке на объект:

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

1. Запустите код.

Результат следующий:

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 turnOn()метод для переменной SmartDeviceтипа, и в зависимости от фактического значения переменной turnOn()могут быть выполнены различные реализации метода.

### **Повторно используйте код суперкласса в подклассах с super ключевым словом**

Если вы внимательно посмотрите на методы turnOn()и turnOff(), то заметите сходство в том, как deviceStatusобновляется переменная при каждом вызове методов в подклассах SmartTvDeviceи SmartLightDevice: код дублируется. Вы можете повторно использовать код при обновлении статуса в SmartDeviceклассе.

Чтобы вызвать переопределенный метод суперкласса из подкласса, вам нужно использовать superключевое слово. Вызов метода из суперкласса аналогичен вызову метода извне класса. Вместо использования .оператора между объектом и методом вам нужно использовать superключевое слово, которое сообщает компилятору Kotlin о необходимости вызова метода в суперклассе, а не в подклассе.

Синтаксис вызова метода из суперкласса начинается с superключевого слова, за которым следуют .оператор, имя функции и набор круглых скобок. Если применимо, в круглые скобки включаются аргументы. Вы можете увидеть синтаксис на этой диаграмме:



Повторно используйте код из SmartDeviceсуперкласса:

1. Удалите println()операторы из методов turnOn()и turnOff()и переместите дублированный код из подклассов SmartTvDeviceи SmartLightDeviceв SmartDeviceсуперкласс:

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

1. Используйте superключевое слово для вызова методов класса SmartDeviceв подклассах SmartTvDeviceи 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")  
    }  
}

## Переопределить свойства суперкласса из подклассов

Подобно методам, вы также можете переопределить свойства, выполнив те же действия.

Переопределить deviceTypeсвойство:

1. В SmartDeviceсуперклассе в строке после свойства deviceStatusиспользуйте ключевые слова open и valдля определения deviceTypeсвойства, заданного в виде "unknown"строки:

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

1. В SmartTvDeviceклассе используйте ключевые слова overrideи valдля определения deviceTypeсвойства, заданного в виде "Smart TV"строки:

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

1. В SmartLightDeviceклассе используйте ключевые слова overrideи valдля определения deviceTypeсвойства, заданного в виде "Smart Light"строки:

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

## [8. Модификаторы видимости](https://developer.android.com/codelabs/basic-android-kotlin-compose-classes-and-objects?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-classes-and-objects#7)

Модификаторы видимости играют важную роль в достижении инкапсуляции:

* В классе они позволяют скрыть ваши 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 декларацию доступной повсюду. Свойства и методы, которые вы хотите использовать вне класса, помечаются как общедоступные.
* private**.**Делает объявление доступным в том же классе или исходном файле.

Вероятно, существуют некоторые свойства и методы, которые используются только внутри класса, и вы не обязательно хотите, чтобы их использовали другие классы. Эти свойства и методы можно пометить модификатором privateвидимости, чтобы гарантировать, что другой класс не сможет случайно получить к ним доступ.

* protected**.**Делает объявление доступным в подклассах. Свойства и методы, которые вы хотите использовать в определяющем их классе и подклассах, помечены модификатором protectedвидимости.
* internal**.**Делает объявление доступным в том же модуле. Модификатор Internal аналогичен модификатору Private, но вы можете получить доступ к внутренним свойствам и методам вне класса, если доступ к ним осуществляется в том же модуле.

**Примечание.** Модуль — это набор исходных файлов и настроек сборки, которые позволяют разделить проект на отдельные функциональные единицы [.](https://developer.android.com/studio/projects#ApplicationModules)Ваш проект может иметь один или несколько модулей. Вы можете независимо собирать, тестировать и отлаживать каждый модуль.

Пакет похож на каталог или папку, в которой группируются связанные классы, тогда как модуль предоставляет контейнер для исходного кода вашего приложения, файлов ресурсов и настроек уровня приложения . Модуль может содержать несколько пакетов.

Когда вы определяете класс, он становится общедоступным, и к нему может получить доступ любой пакет, который его импортирует. Это 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, чтобы скрыть свойства и методы, к которым другим классам не нужен доступ.

Например, рассмотрим, как автомобиль становится доступным для водителя. Подробности о том, из каких частей состоит автомобиль и как он работает внутри, по умолчанию скрыты. Автомобиль должен быть максимально интуитивным в управлении. Вы не хотели бы, чтобы автомобиль был таким же сложным в эксплуатации, как коммерческий самолет, подобно тому, как вы не хотели бы, чтобы другой разработчик или вы в будущем были в замешательстве относительно того, какие свойства и методы класса следует использовать.

Модификаторы видимости помогают вам отображать соответствующие части кода для других классов вашего проекта и гарантировать, что реализация не может быть непреднамеренно использована, что делает код более простым для понимания и менее подверженным ошибкам.

Модификатор видимости должен быть помещен перед синтаксисом объявления при объявлении класса, метода или свойств, как вы можете видеть на этой диаграмме:



## Укажите модификатор видимости для свойств

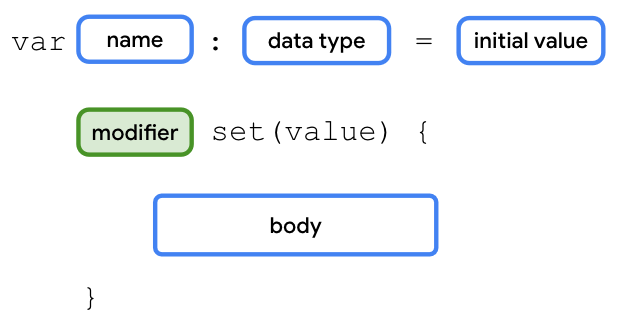
Синтаксис указания модификатора видимости для свойства начинается с модификатора private, protectedили , internalза которым следует синтаксис, определяющий свойство. Вы можете увидеть синтаксис на этой диаграмме:



Например, deviceStatusв этом фрагменте кода вы можете увидеть, как сделать свойство частным:

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

Вы также можете установить модификаторы видимости для функций установки. Модификатор ставится перед setключевым словом. Вы можете увидеть синтаксис на этой диаграмме:



**Примечание.** Если модификатор видимости функции получения не совпадает с модификатором видимости свойства, компилятор сообщает об ошибке.

Для SmartDeviceкласса значение свойства deviceStatusдолжно быть доступно для чтения за пределами класса через объекты класса. Однако только класс и его дочерние элементы должны иметь возможность обновлять или записывать значение. Чтобы реализовать это требование, вам необходимо использовать protectedмодификатор функции set()свойства deviceStatus.

Используйте protectedмодификатор функции set()свойства deviceStatus:

1. В свойстве SmartDeviceсуперкласса deviceStatusдобавьте protectedк set()функции модификатор:

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

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

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

1. В SmartHomeклассе определите deviceTurnOnCountсвойство, для которого установлено 0значение, с помощью частной функции установки:

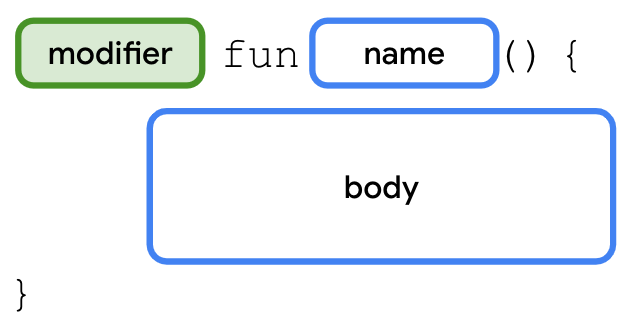
class SmartHome(  
    val smartTvDevice: SmartTvDevice,  
    val smartLightDevice: SmartLightDevice  
) {  
  
    var deviceTurnOnCount = 0  
        private set  
  
    ...  
}

1. Добавьте deviceTurnOnCountсвойство, за которым следует ++арифметический оператор, к методам turnOnTv()and turnOnLight(), а затем добавьте deviceTurnOnCountсвойство, за которым следует --арифметический оператор, к методам turnOffTv()and 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 указания модификатора видимости для метода начинается с модификаторов private, protectedили или, internalза которыми следует синтаксис, определяющий метод. Вы можете увидеть синтаксис на этой диаграмме:



Например, в этом фрагменте кода вы можете увидеть, как указать protectedмодификатор для nextChannel()метода в классе:SmartTvDevice

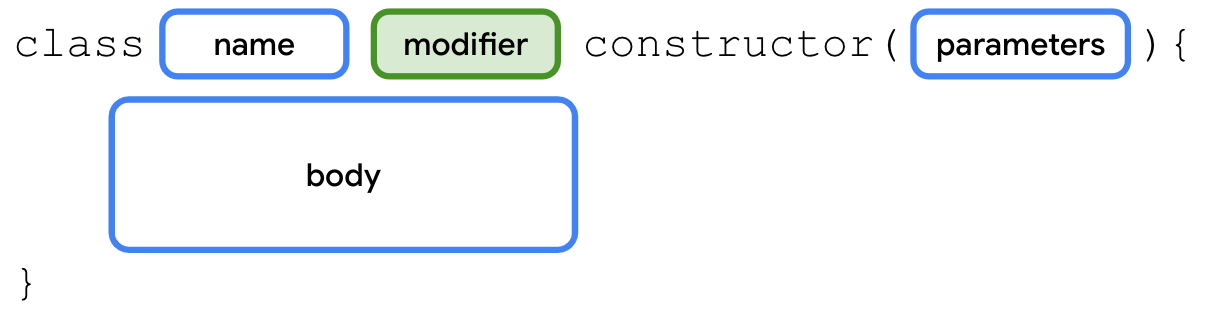
class SmartTvDevice(deviceName: String, deviceCategory: String) :  
    SmartDevice(name = deviceName, category = deviceCategory) {  
  
    ...  
  
    protected fun nextChannel() {  
        channelNumber++  
        println("Channel number increased to $channelNumber.")  
    }        
  
    ...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

## Модификаторы видимости для конструкторов

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

* Модификатор указывается после имени класса, но перед constructorключевым словом.
* Если вам нужно указать модификатор для основного конструктора, необходимо сохранить constructorключевое слово и круглые скобки, даже если параметров нет.

Вы можете увидеть синтаксис на этой диаграмме:

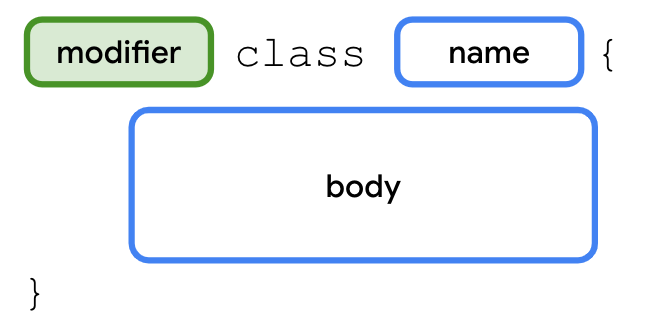


Например, в этом фрагменте кода вы можете увидеть, как добавить protectedмодификатор к конструктору:SmartDevice

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

## Модификаторы видимости классов

Синтаксис указания модификатора видимости для класса начинается с модификаторов private, protectedили , internalза которыми следует синтаксис, определяющий класс. Вы можете увидеть синтаксис на этой диаграмме:



Например, вы можете увидеть, как указать internalмодификатор для SmartDeviceкласса в этом фрагменте кода:

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

В идеале следует стремиться к строгой видимости свойств и методов, поэтому объявляйте их с модификатором privateкак можно чаще. Если вы не можете сохранить их конфиденциальными, используйте protectedмодификатор. Если вы не можете защитить их, используйте internalмодификатор. Если вы не можете сохранить их внутренними, используйте publicмодификатор.

### **Укажите соответствующие модификаторы видимости**

Эта таблица поможет вам определить соответствующие модификаторы видимости в зависимости от того, где должно быть доступно свойство или методы класса или конструктора:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Модификатор** | **Доступно в том же классе** | **Доступно в подклассе** | **Доступно в том же модуле** | **Доступный внешний модуль** |
| private | ✔ | 𝗫 | 𝗫 | 𝗫 |
| protected | ✔ | ✔ | 𝗫 | 𝗫 |
| internal | ✔ | ✔ | ✔ | 𝗫 |
| public | ✔ | ✔ | ✔ | ✔ |

В SmartTvDeviceподклассе вы не должны позволять управлять свойствами speakerVolumeи извне класса. channelNumberЭтими свойствами следует управлять только с помощью методов increaseSpeakerVolume()и nextChannel().

Аналогично, в SmartLightDeviceподклассе brightnessLevelсвойство должно контролироваться только через increaseLightBrightness()метод.

Добавьте соответствующие модификаторы видимости в подклассы SmartTvDeviceи SmartLightDevice:

1. В SmartTvDeviceклассе добавьте privateмодификатор видимости к свойствам speakerVolumeи 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  
            }  
        }  
  
    ...  
}

1. В SmartLightDeviceклассе добавьте privateмодификатор к brightnessLevelсвойству:

class SmartLightDevice(deviceName: String, deviceCategory: String) :  
    SmartDevice(name = deviceName, category = deviceCategory) {  
  
    ...  
  
    private var brightnessLevel = 0  
        set(value) {  
            if (value in 0..100) {  
                field = value  
            }  
        }  
  
    ...  
}

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

В предыдущем разделе вы узнали, что свойства в Kotlin используют резервное поле для хранения своих значений в памяти. Вы используете fieldидентификатор для ссылки на него.

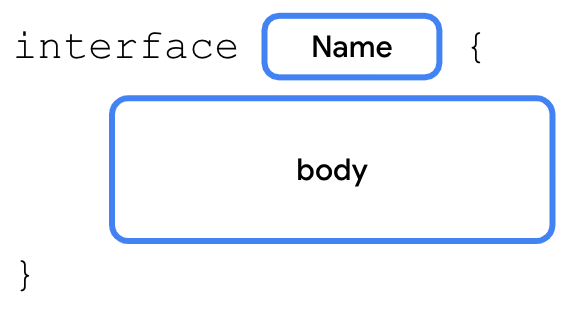
Когда вы посмотрите на код, вы увидите дублированный код, чтобы проверить, находятся ли значения в пределах диапазона для свойств speakerVolume, channelNumberи brightnessLevelв классах SmartTvDeviceи SmartLightDevice. Вы можете повторно использовать код проверки диапазона в функции установки с делегатами . Вместо использования поля и функций получения и установки для управления значением им управляет делегат.

Синтаксис создания делегатов свойств начинается с объявления переменной, за которой следует byключевое слово, и объекта делегата, который обрабатывает функции получения и установки свойства. Вы можете увидеть синтаксис на этой диаграмме:



Прежде чем реализовать класс, которому можно делегировать реализацию, вам необходимо ознакомиться с интерфейсами . Интерфейс — это контракт, которого должны придерживаться реализующие его классы. Он фокусируется на том, что делать, а не на том, как выполнить действие. Короче говоря, интерфейс помогает достичь абстракции .

Например, прежде чем построить дом, вы сообщаете архитектору, чего хотите. Вам нужна спальня, детская, гостиная, кухня и пара ванных комнат. Короче говоря, вы указываете, чего хотите , а архитектор указывает, как этого достичь . Синтаксис создания интерфейса вы можете увидеть на этой диаграмме:



Вы уже узнали, как расширить класс и переопределить его функциональность. Что касается интерфейсов, класс реализует интерфейс. Класс предоставляет сведения о реализации методов и свойств, объявленных в интерфейсе. Вы сделаете нечто подобное с [ReadWriteProperty](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.properties/-read-write-property/" \t "_blank)интерфейсом для создания делегата. Подробнее об интерфейсах вы узнаете в следующем модуле.

Чтобы создать класс делегата для varтипа, вам необходимо реализовать [ReadWriteProperty](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.properties/-read-write-property/" \t "_blank)интерфейс. Аналогично вам необходимо реализовать [ReadOnlyProperty](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.properties/-read-only-property/" \t "_blank)интерфейс для valтипа.

Создайте делегата для varтипа:

1. Перед main()функцией создайте RangeRegulatorкласс, реализующий ReadWriteProperty<Any?, Int>интерфейс:

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

Не беспокойтесь об угловых скобках или содержимом внутри них. Они представляют собой обобщенные типы, и вы узнаете о них в следующем разделе.

1. В RangeRegulatorосновной конструктор класса добавьте initialValue параметр, частное minValueсвойство и частное maxValueсвойство Intтипа:

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

1. В RangeRegulatorтеле класса 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 действуют как функции получения и установки свойств.

**Примечание.** Это [KProperty](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-property/" \l "kproperty" \t "_blank)интерфейс, который представляет объявленное свойство и позволяет получить доступ к метаданным делегированного свойства. Хорошо иметь информацию высокого уровня о том, что это [KProperty](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-property/" \l "kproperty" \t "_blank)такое.

1. В строке перед SmartDeviceклассом импортируйте интерфейсы ReadWritePropertyи 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) {  
    }  
}  
  
...

1. В RangeRegulatorклассе в строке перед getValue()методом определите fieldDataсвойство и инициализируйте его initialValueпараметром:

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 действует как вспомогательное поле для переменной.

1. В getValue()теле метода верните 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) {  
    }  
}

1. В setValue()теле метода проверьте, valueнаходится ли назначаемый параметр в minValue..maxValueдиапазоне, прежде чем назначать 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  
        }  
    }  
}

1. В SmartTvDeviceклассе используйте класс делегата для определения свойств speakerVolumeи 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)  
  
    ...  
  
}

1. В SmartLightDeviceклассе используйте класс делегата для определения brightnessLevelсвойства:

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)  
  
    ...  
  
}

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

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()  
}

Результат следующий:

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.