Unit 2: Building app UI

Session Objective:

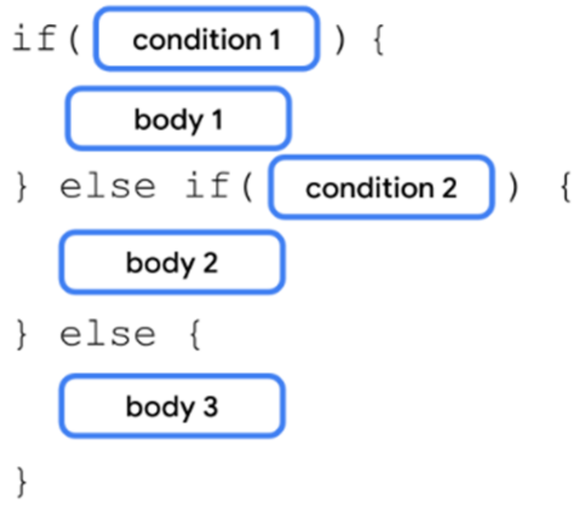
- Learn Kotlin Basics
- Add UI components
- Interact with UI

Kotlin Fundamentals

Kotlin Playground - Kotlin Playground
Open Kotlin Playground and go along with the speaker.
Kotlin docs - https://kotlinlang.org/docs/home.html

Conditional Statements in Kotlin

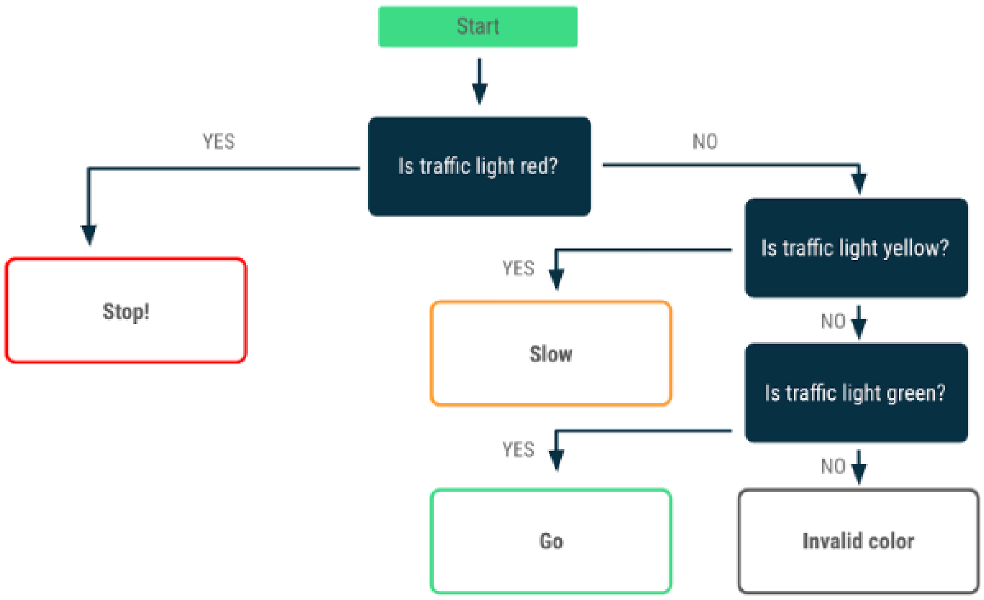
Similar logic like other programming languages.
SYNTAX for basic if/else condition:



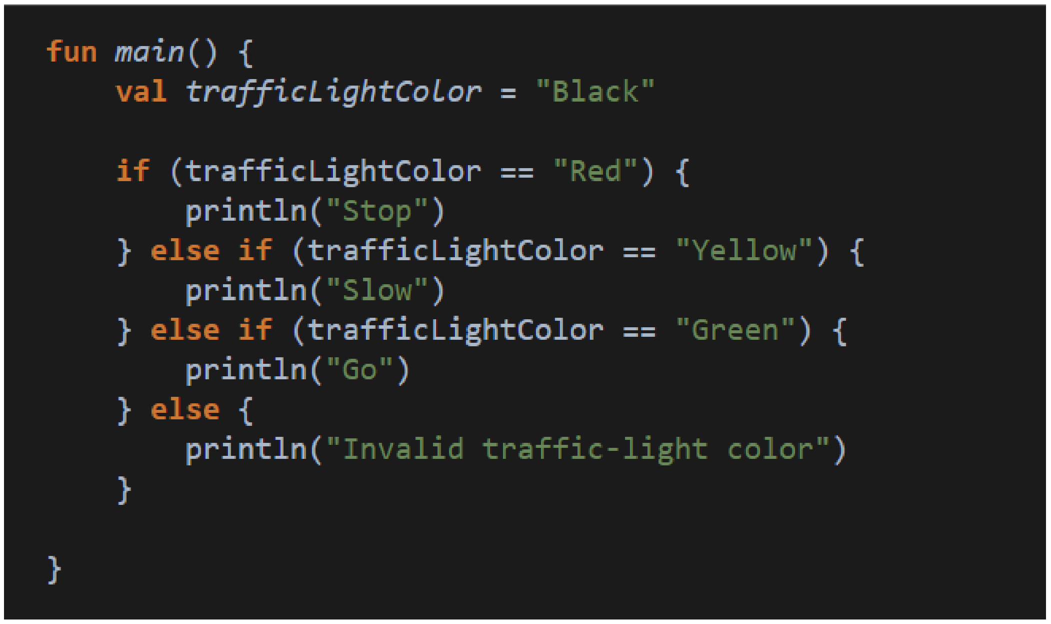
```
if( condition 1 ) {
    body 1
} else if( condition 2 ) {
    body 2
} else {
    body 3
}
```

Discuss with an example :

Flowchart for example - **Show messages based on traffic light**

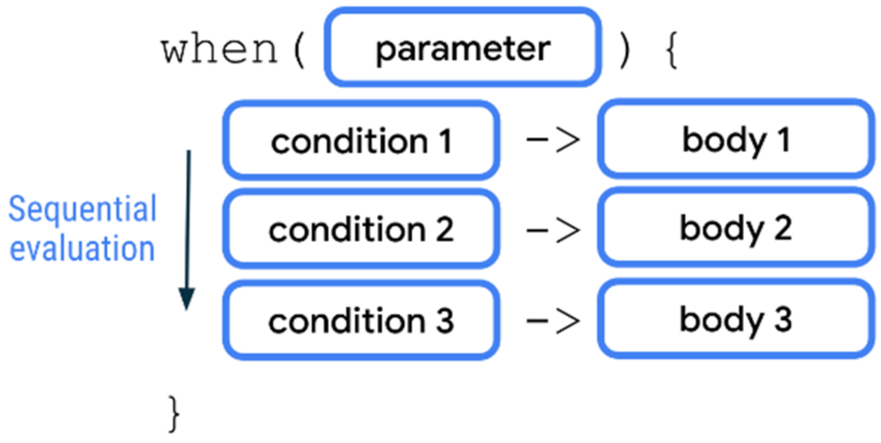


Solution :

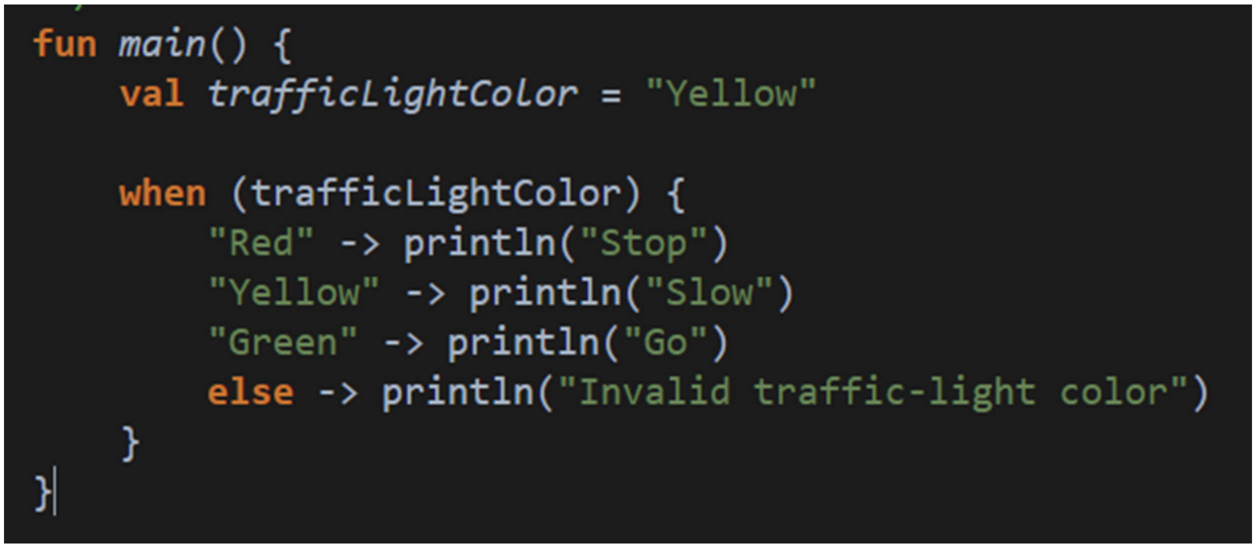


```
fun main() {
    val trafficLightColor = "Black"

    if (trafficLightColor == "Red") {
        println("Stop")
    } else if (trafficLightColor == "Yellow") {
        println("Slow")
    } else if (trafficLightColor == "Green") {
        println("Go")
    } else {
        println("Invalid traffic-light color")
    }

}
```

When statement
SYNTAX

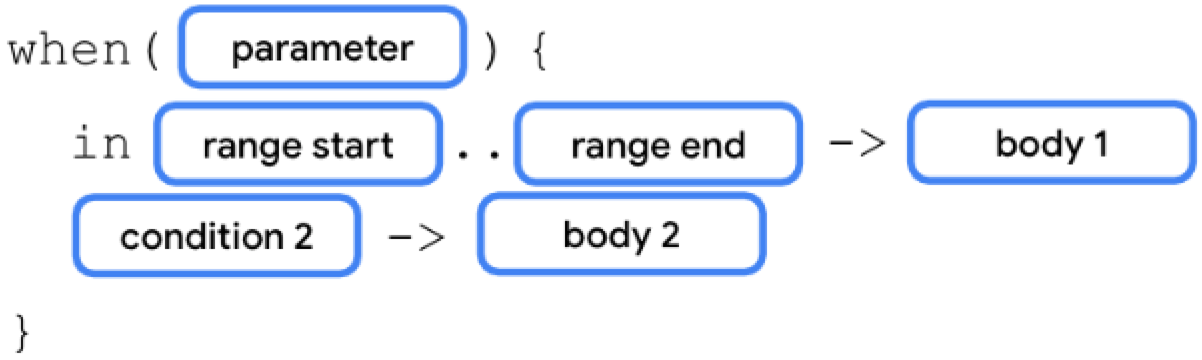


Same problem solution using when

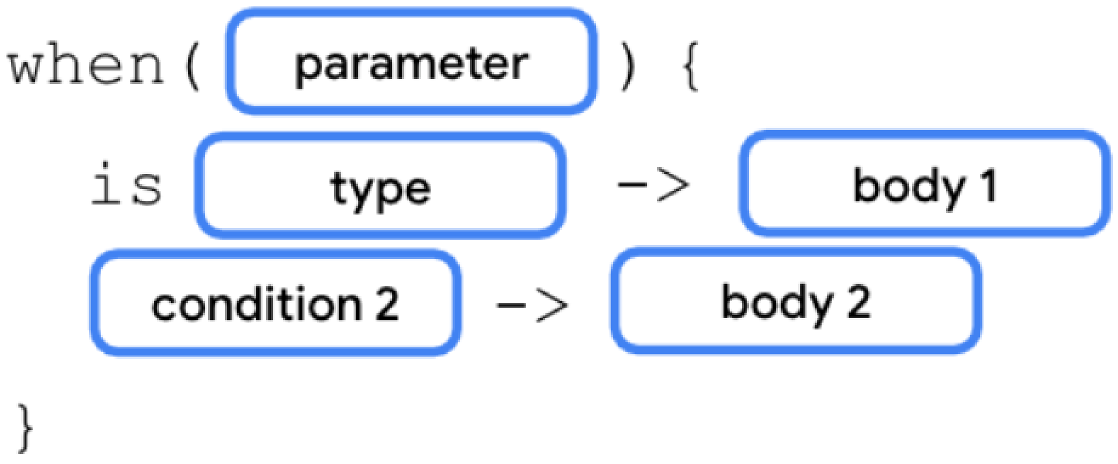


```
fun main() {
    val trafficLightColor = "Yellow"

    when (trafficLightColor) {
        "Red" -> println("Stop")
        "Yellow" -> println("Slow")
        "Green" -> println("Go")
        else -> println("Invalid traffic-light color")
    }
}
```

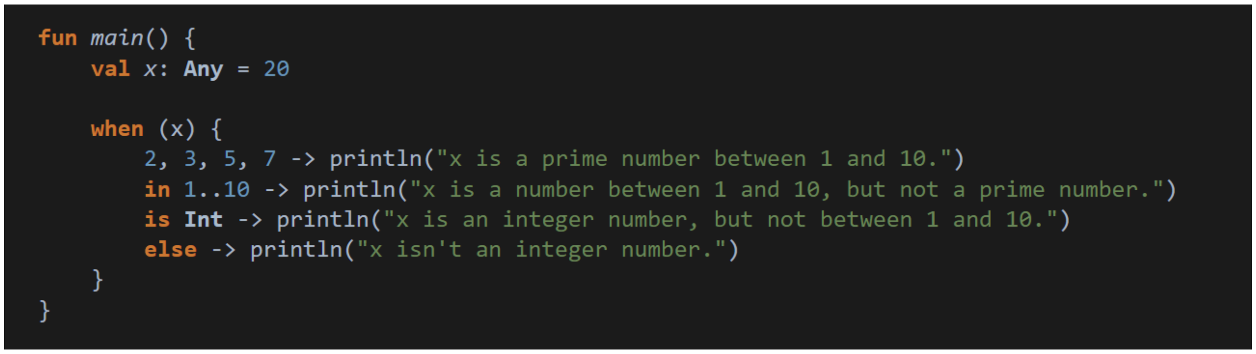
in keyword - used to define a range of values in when branches.



is keyword - a condition to check the data type of an evaluated value



Example:

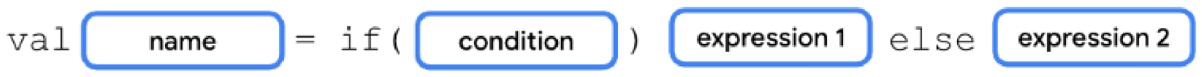


```
fun main() {
    val x: Any = 20

    when (x) {
        2, 3, 5, 7 -> println("x is a prime number between 1 and 10.")
        in 1..10 -> println("x is a number between 1 and 10, but not a prime number.")
        is Int -> println("x is an integer number, but not between 1 and 10.")
        else -> println("x isn't an integer number.")
    }
}
```

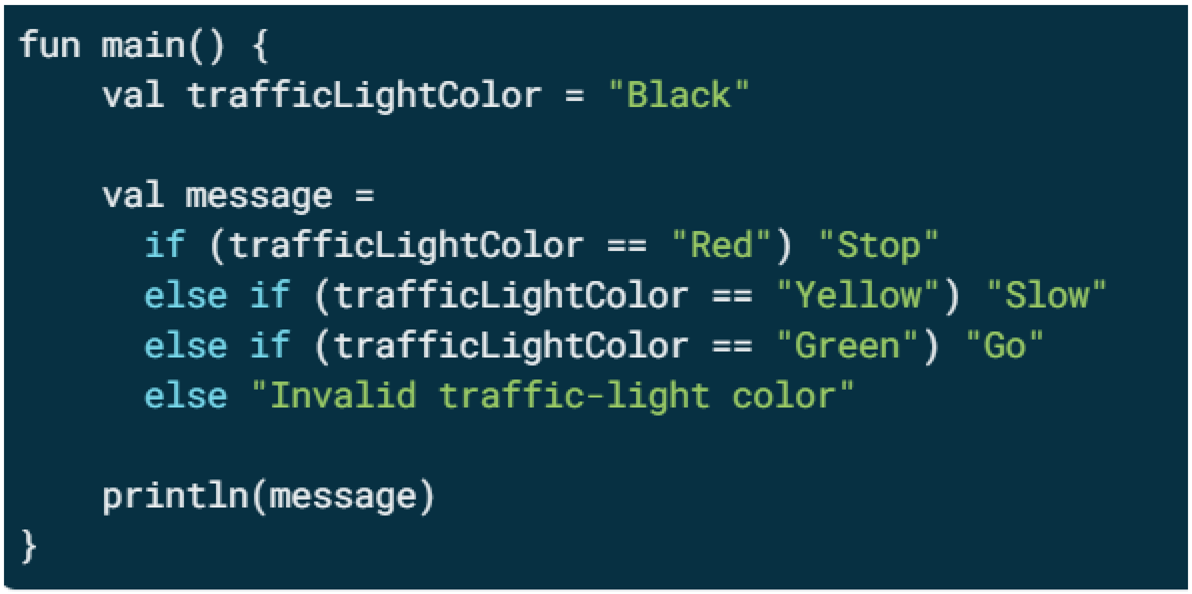
Using Expressions

You learned how to use if/else and when as statements. You can also use conditionals as expressions to return different values for each branch of condition.

SYNTAX:



Example:



```
fun main() {
    val trafficLightColor = "Black"

    val message =
      if (trafficLightColor == "Red") "Stop"
      else if (trafficLightColor == "Yellow") "Slow"
      else if (trafficLightColor == "Green") "Go"
      else "Invalid traffic-light color"

    println(message)
}
```

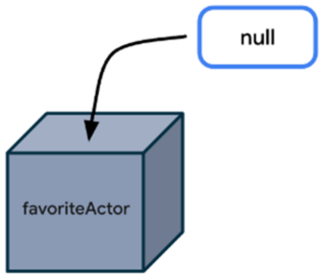
We can also do this using when statements. But How ? (Question to Participants)

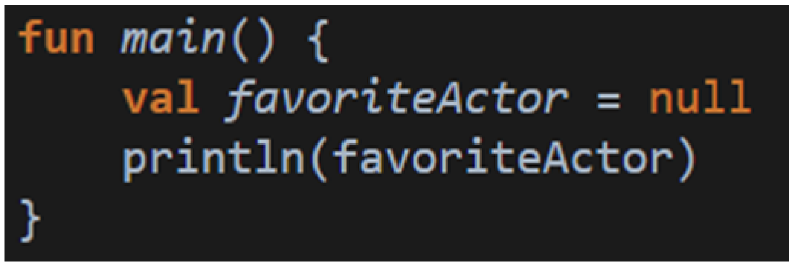
Use nullability in Kotlin

What is null ?

Kotlin null safety is a procedure to eliminate the risk of null reference from the code. Kotlin compiler throws NullPointerException immediately if it is found any null argument is passed without executing any other statements.
In Java this would be the equivalent of a NullPointerException, or an *NPE* for short.

Example:





```
fun main() {
    val favoriteActor = null
    println(favoriteActor)
}
```

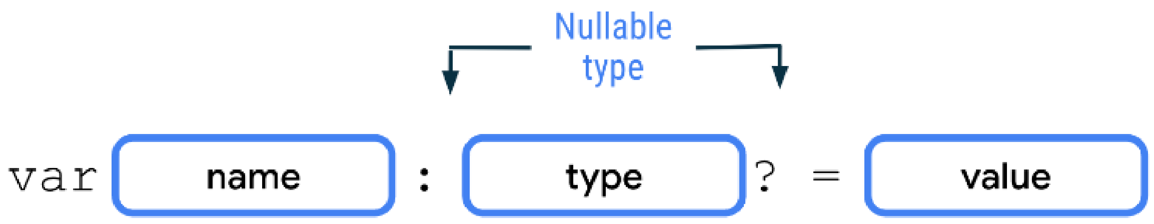
Guess the output :

```kotlin
fun main() {
    var favoriteActor: String = "Sandra Oh"
    favoriteActor = null
}
```

Answer :

```
❗ Null can not be a value of a non-null type String
```

Reason:
Nullable types (can hold null) and non-nullable types(can't hold null)
Declare Nullable type:
SYNTAX:



Now Guess the output:

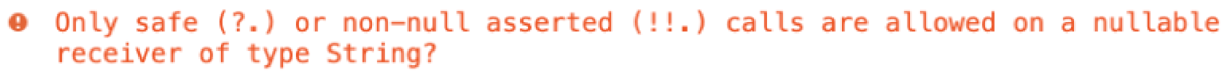
```kotlin
fun main() {
    var favoriteActor: String? = "Sandra Oh"
    favoriteActor = null
}
```

Most common error : Accessing nullable variable

Let's run the code

```kotlin
fun main() {
    var favoriteActor: String? = "Sandra Oh"
    println(favoriteActor.length)
}
```

What's the output ?



```
Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable
receiver of type String?
```

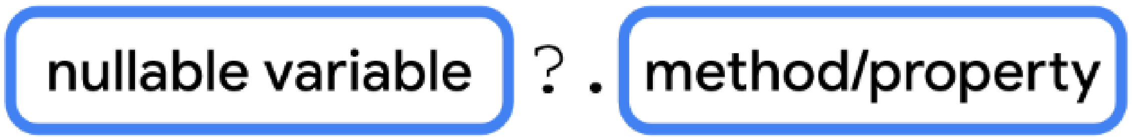
Output
the code fails at compile time because the direct reference to the length property for the favoriteActor variable isn't allowed because there's a possibility that the variable is null.

This causes a critical issue - known as null reference - attempt to access a member of a variable that is null. This might cause the app to crash. This type of crash is known as runtime error.

Due to the null safety nature of Kotlin, such runtime errors are prevented because the Kotlin compiler forces a null *check* for nullable types. Null *check* refers to a process of checking whether a variable could be null before it's accessed and treated as a non-nullable type. If you wish to use a nullable value as its non-nullable type, you need to perform a null check explicitly.

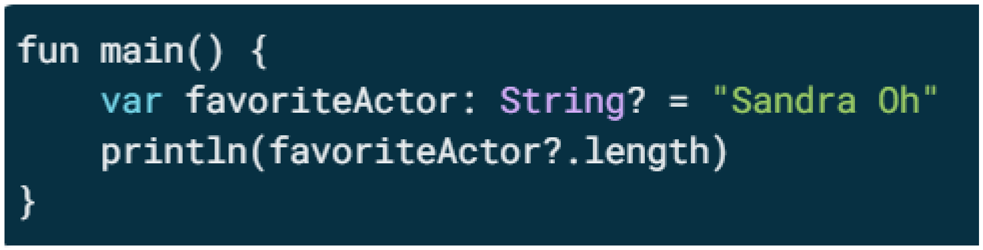
How to fix this ?

1. Using Safe Call operator (?.)
   SYNTAX



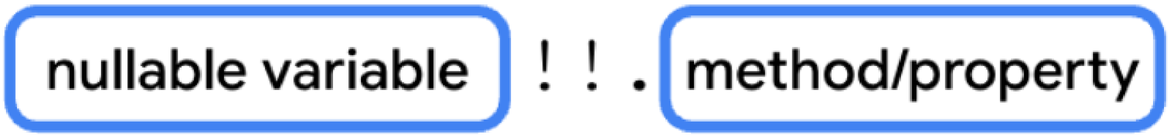
The ?. safe-call operator allows safer access to nullable variables because the Kotlin compiler stops any attempt of member access to null references and returns null for the member accessed.

Now what's the output ?



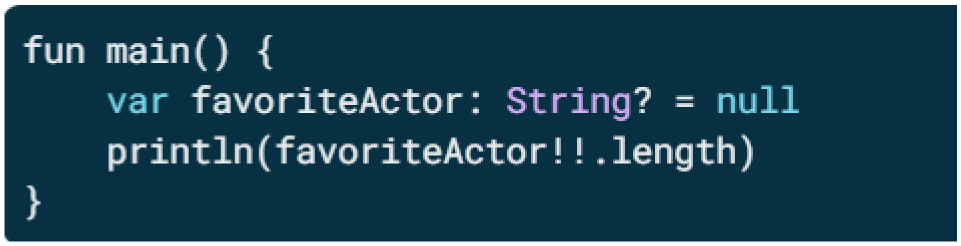
```
fun main() {
    var favoriteActor: String? = "Sandra Oh"
    println(favoriteActor?.length)
}
```

2. Using not-null assertion operator (!!)
   SYNTAX:



It means that you assert that the value of the variable isn't null, regardless of whether it is or isn't.

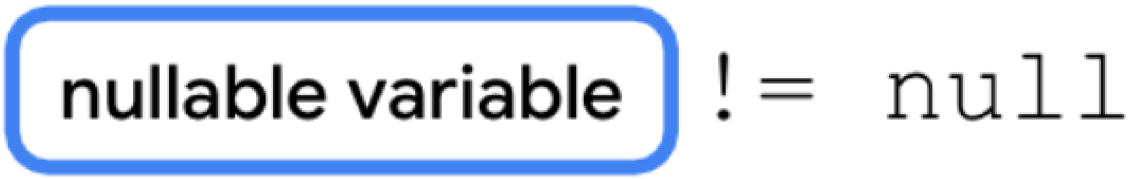
What's the output ?



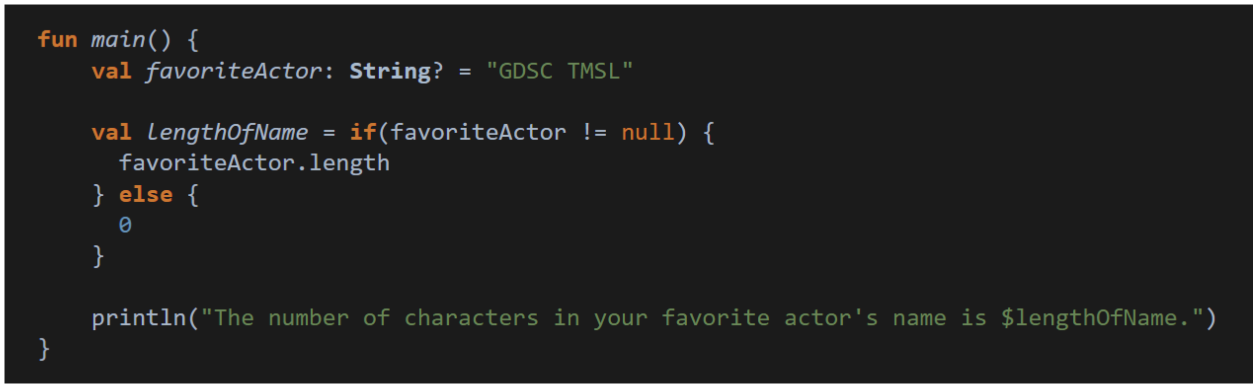
```kotlin
fun main() {
    var favoriteActor: String? = null
    println(favoriteActor!!.length)
}
```

You get NullPointerException

3. Using if/else
   SYNTAX:



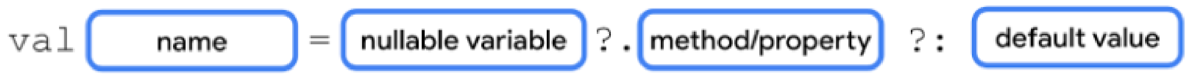
Understanding test :



```kotlin
fun main() {
    val favoriteActor: String? = "GDSC TMSL"

    val lengthOfName = if(favoriteActor != null) {
      favoriteActor.length
    } else {
      0
    }

    println("The number of characters in your favorite actor's name is $lengthOfName.")
}
```

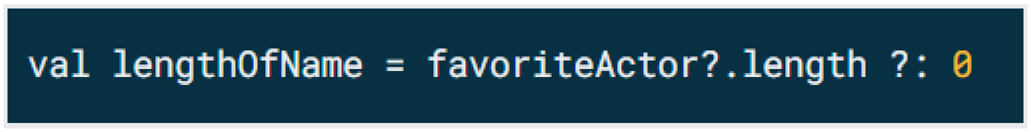
Guess the output ?

4. Elvis operator (?:)
SYNTAX:



If the variable *isn't* null, the expression before the ?: Elvis operator executes. If the variable *is* null, the expression after the ?:

How can we express the previous problem using Elvis operator ?

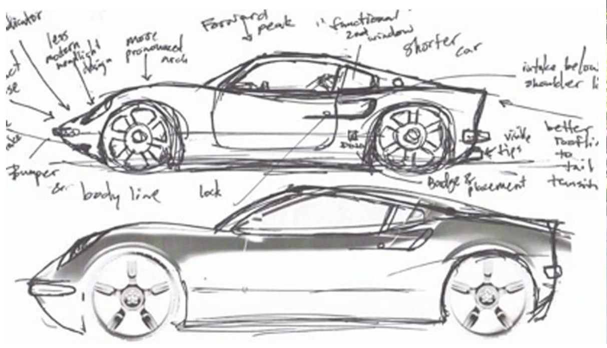


```
val lengthOfName = favoriteActor?.length ?: 0
```

OOPS in Kotlin

Whats is OOPS ?

Suppose we want to build a car then first we have to make a design called ***CLASS*** and the real car called ***OBJECT***.



CLASS

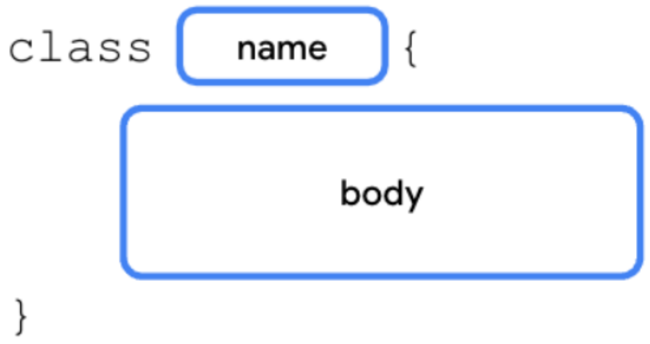
OBJECT

4 basic concepts of OOP -

- ☐ Encapsulation - Wraps the related properties and methods that perform action on those properties in a class.
- ☐ Abstraction - The idea is to hide the internal implementation logic as much as possible.
- ☐ Inheritance - Enables you to build a class upon the characteristics and behavior of other classes by establishing a parent-child relationship.
- ☐ Polymorphism - Ability to use different objects in a single, common way.

Defining a class (Encapsulation)

SYNTAX:

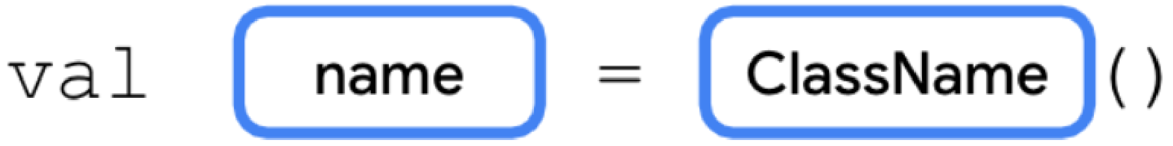


A class consists of three major parts:

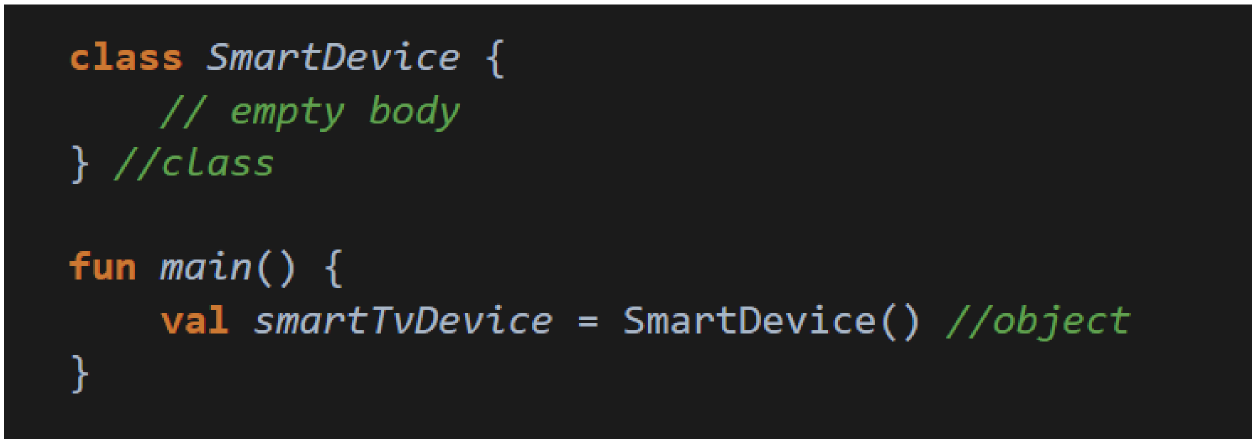
- **Properties**. Variables that specify the attributes of the class's objects.
- **Methods.** Functions that contain the class's behaviors and actions.
- **Constructors.** A special member function that creates instances of the class throughout the program in which it's defined.

Creating an instance of a class

SYNTAX :



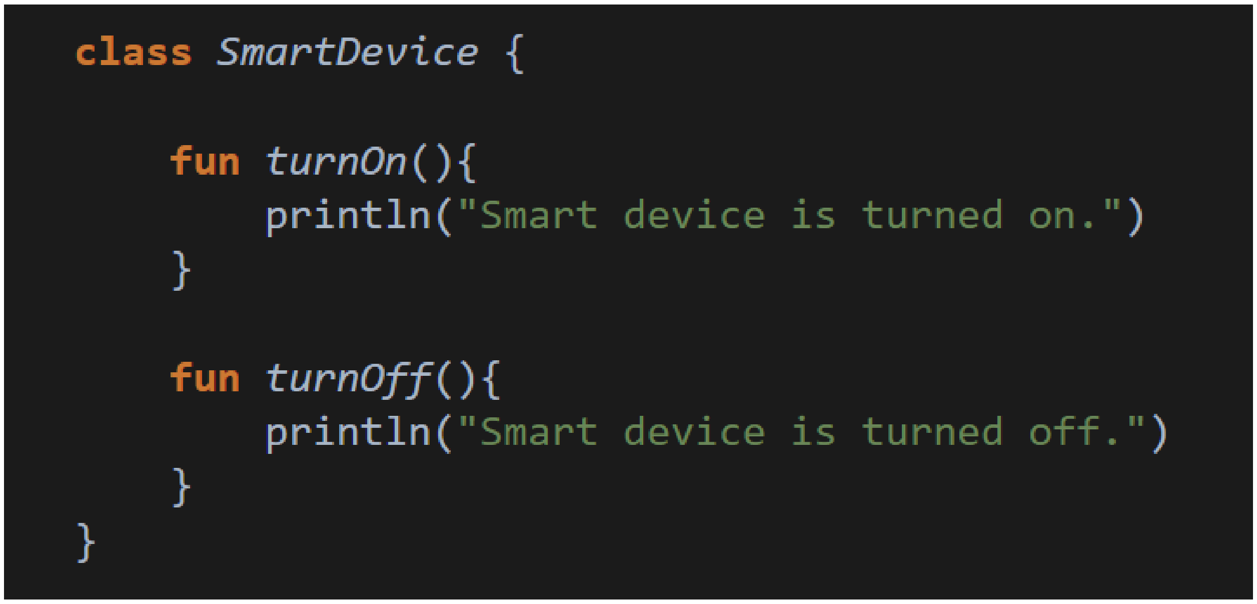
Example:
Create a SmartDevice class and make an object named smartTvDevice



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

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

Defining Methods

Methods are functions. Declared using fun keyword

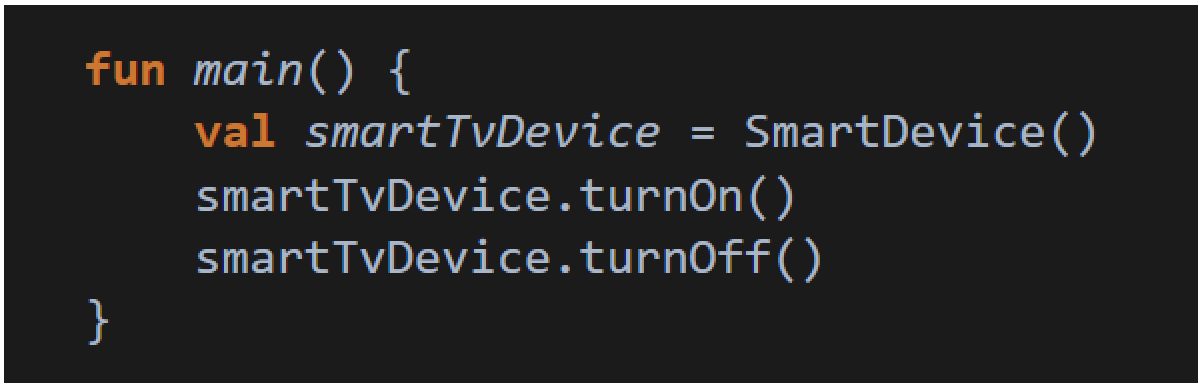


```
class SmartDevice {

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

    fun turnOff(){
        println("Smart device is turned off.")
    }
}
```

Call method from object

SYNTAX:

```
classObject.methodName([Optional] Arguments)
```



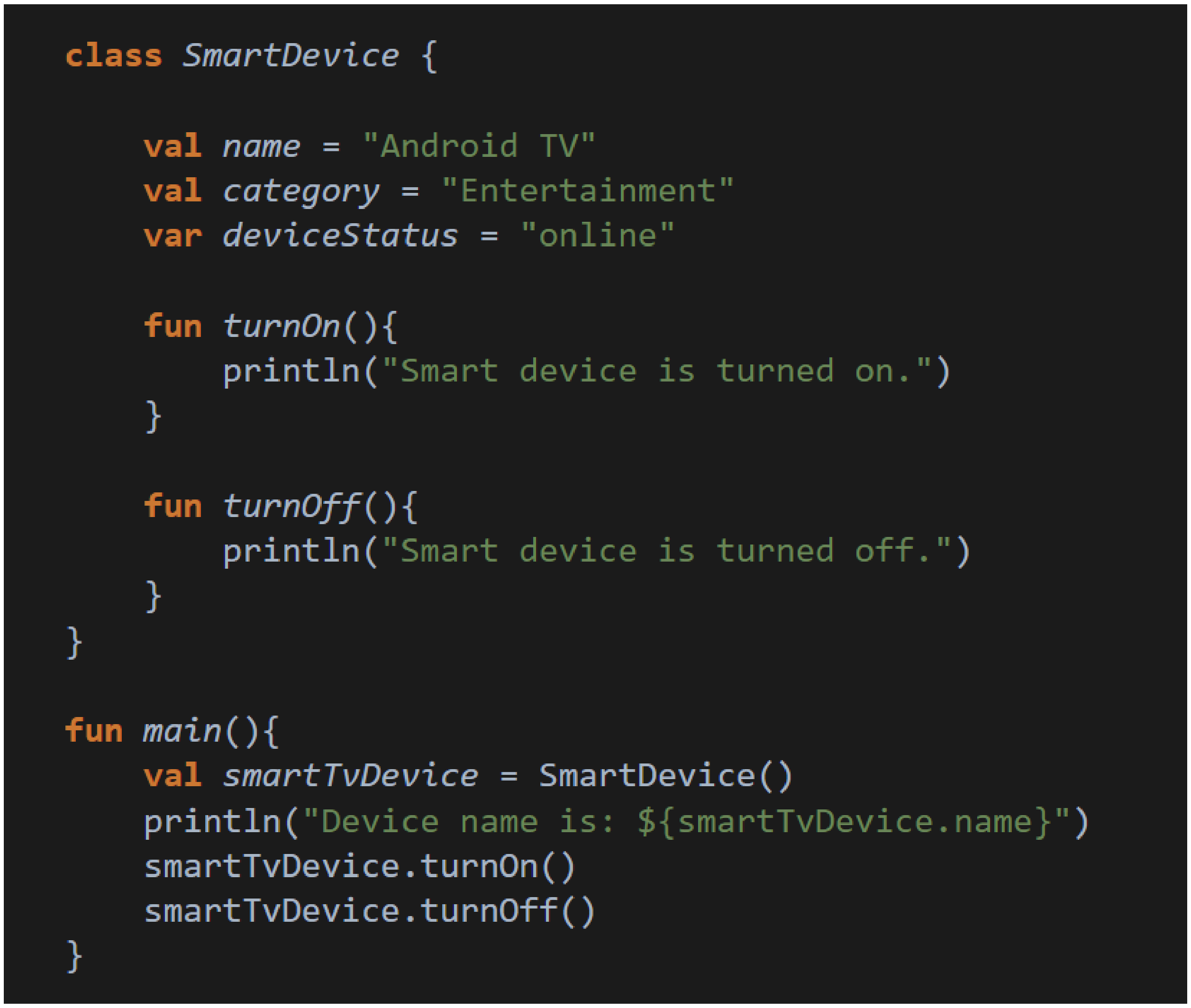
```
fun main() {
    val smartTvDevice = SmartDevice()
    smartTvDevice.turnOn()
    smartTvDevice.turnOff()
}
```

Class properties

We can put variables in a class using the same way we declare variables.
We can access then by

classObject.variablename

Guess the output :



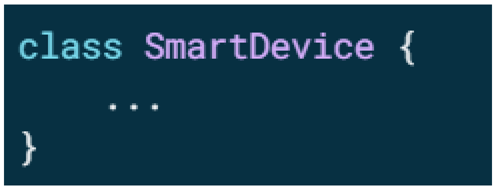
```
class SmartDevice {

    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.")
    }
}

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

Code:

Define a constructor

Constructors initialize an object and make the object ready for use.

1. Default constructor
   A default constructor is a constructor without parameters. Kotlin auto generates the default constructor.
   SYNTAX:



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

2. Parameterized constructor
   User can pass parameters to thai type of constructors.

```kotlin
class SmartDevice(val name: String, val category: String)
```

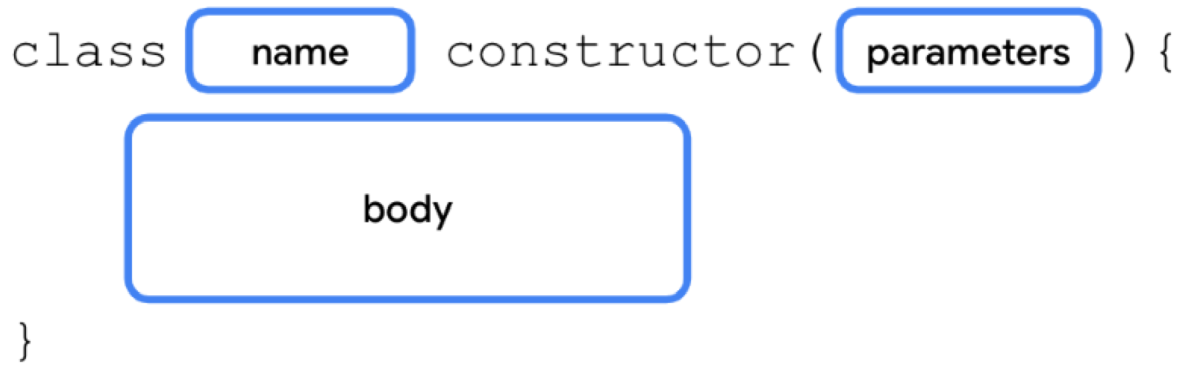
   Init a parameterized constructor

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

There are two main types of constructors in Kotlin:

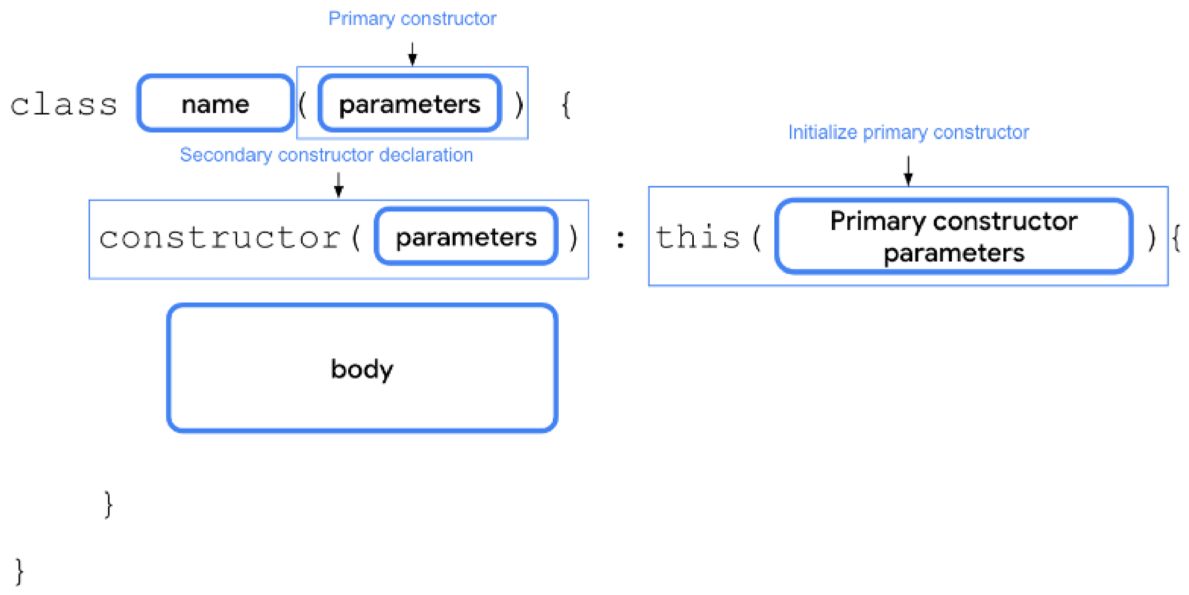
- **Primary constructor.** A class can have only one primary constructor, which is defined as part of the class header. A primary constructor can be a default or parameterized constructor. The primary constructor doesn't have a body. That means that it can't contain any code.

  SYNTAX:

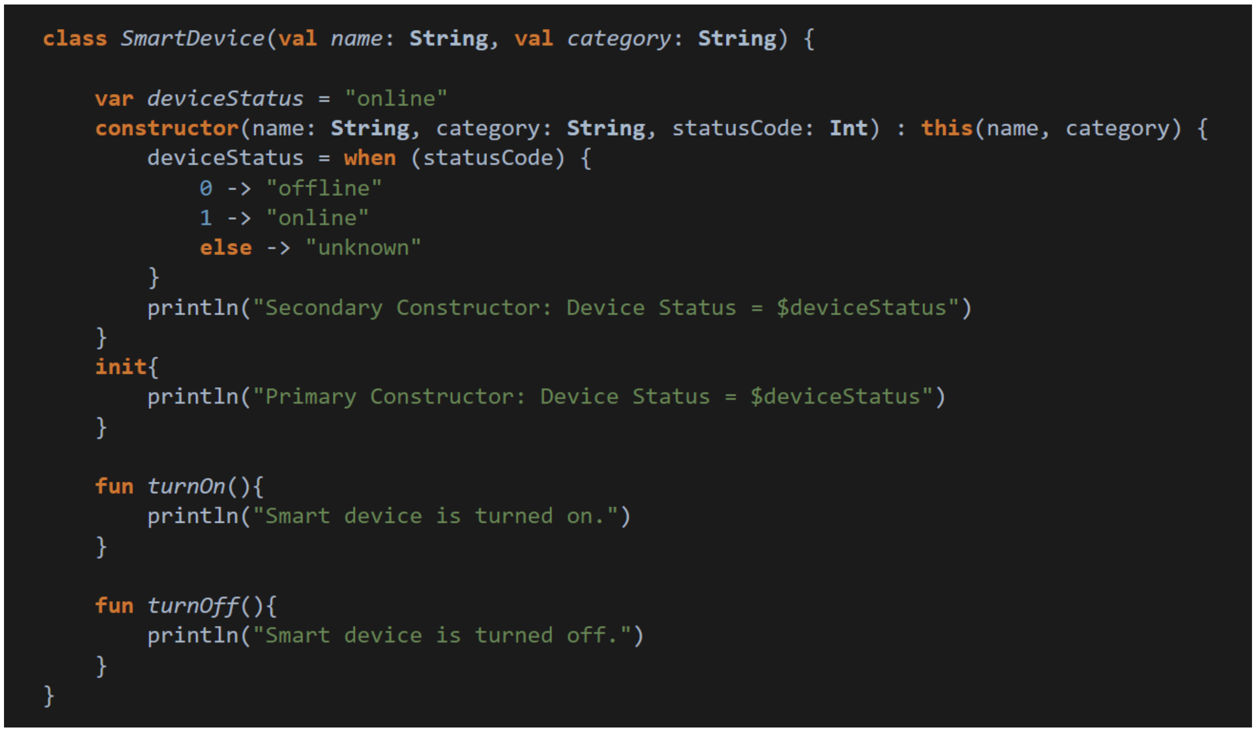


- **Secondary constructor.** A class can have multiple secondary constructors. You can define the secondary constructor with or without parameters. The secondary constructor can initialize the class and has a body, which can contain initialization logic. If the class has a primary constructor, each secondary constructor needs to initialize the primary constructor.

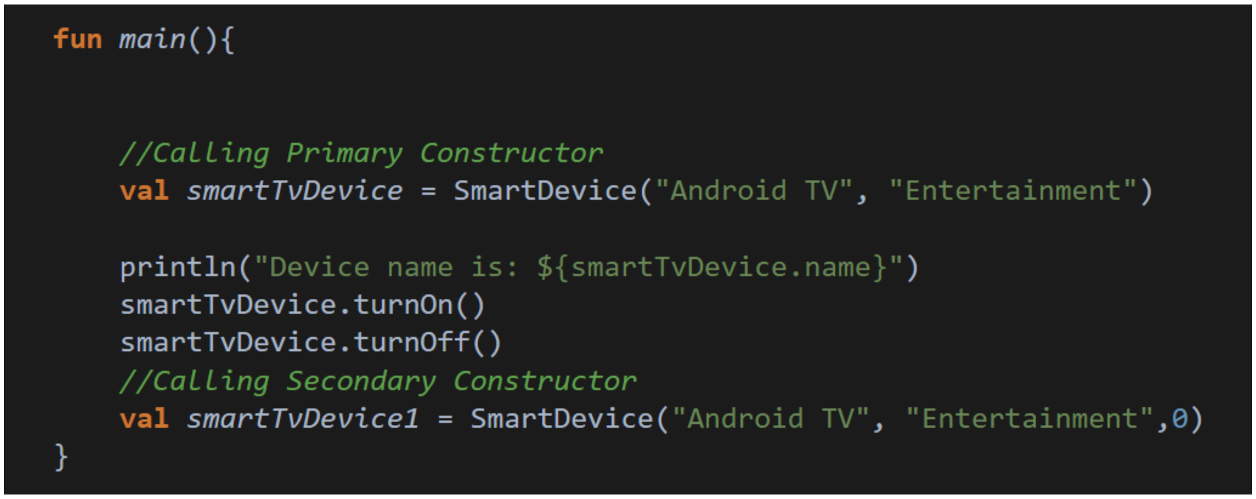
  SYNTAX:


Primary constructor
class
name
(
parameters
)
{
Secondary constructor declaration
Initialize primary constructor
constructor(
parameters
)
:
this(
Primary constructor parameters
){
body
}
}

***Note:* *val or var is NOT ALLOWED in secondary constructor arguments.*
Let’s feel it

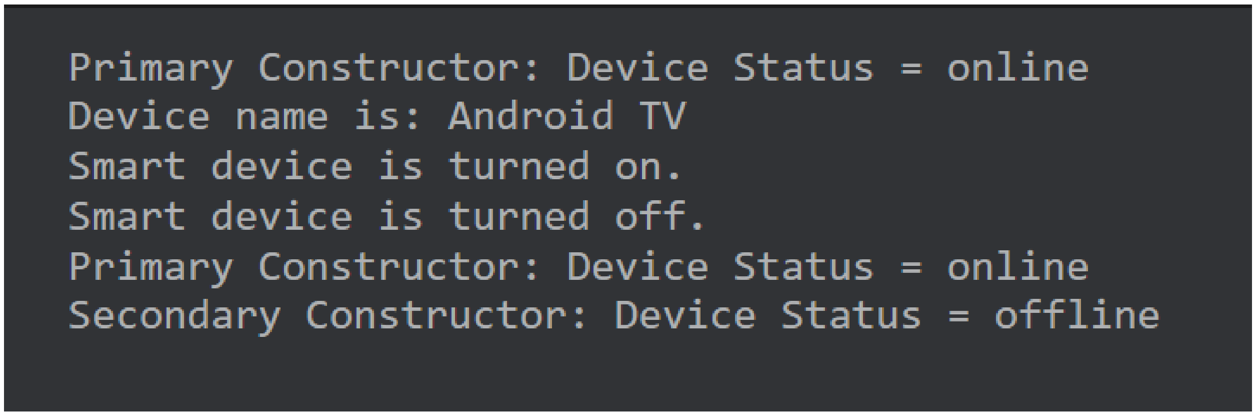


```kotlin
class SmartDevice(val name: String, val category: String) {

    var deviceStatus = "online"
    constructor(name: String, category: String, statusCode: Int) : this(name, category) {
        deviceStatus = when (statusCode) {
            0 -> "offline"
            1 -> "online"
            else -> "unknown"
        }
        println("Secondary Constructor: Device Status = $deviceStatus")
    }
    init{
        println("Primary Constructor: Device Status = $deviceStatus")
    }

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

    fun turnOff(){
        println("Smart device is turned off.")
    }
}
```



```kotlin
fun main(){


    //Calling Primary Constructor
    val smartTvDevice = SmartDevice("Android TV", "Entertainment")

    println("Device name is: ${smartTvDevice.name}")
    smartTvDevice.turnOn()
    smartTvDevice.turnOff()
    //Calling Secondary Constructor
    val smartTvDevice1 = SmartDevice("Android TV", "Entertainment",0)
}
```

**We also get to see polymorphism. SmartDevice used in two ways.
Output Explanation

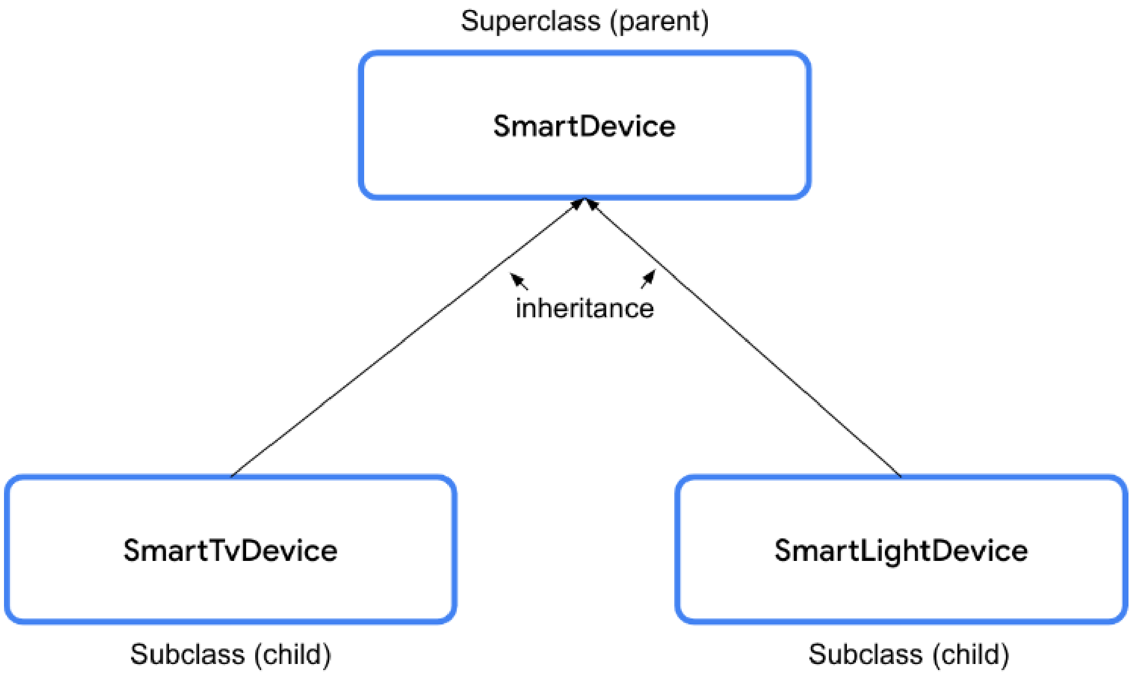


```
Primary Constructor: Device Status = online
Device name is: Android TV
Smart device is turned on.
Smart device is turned off.
Primary Constructor: Device Status = online
Secondary Constructor: Device Status = offline
```

Code Link: https://pl.kotl.in/uyyz1teKY?theme=darcula&readOnly=true
https://gist.github.com/priyasu-cx/3a079bab1942b6ce4dda3c3632a1eb19

Relationship between classes (inheritance)

1. ***open*** → To inherit from any class we have to make it open as by default every class is public and final in Kotlin.
2. ***super*** → It calls to upper level of Class currently we are precent at.
3. ***override*** → It is used to change/override the functionality of any function or variable of Parent class from child class.
4. ***private*** → It is used to make any item to be accessed within that class only.

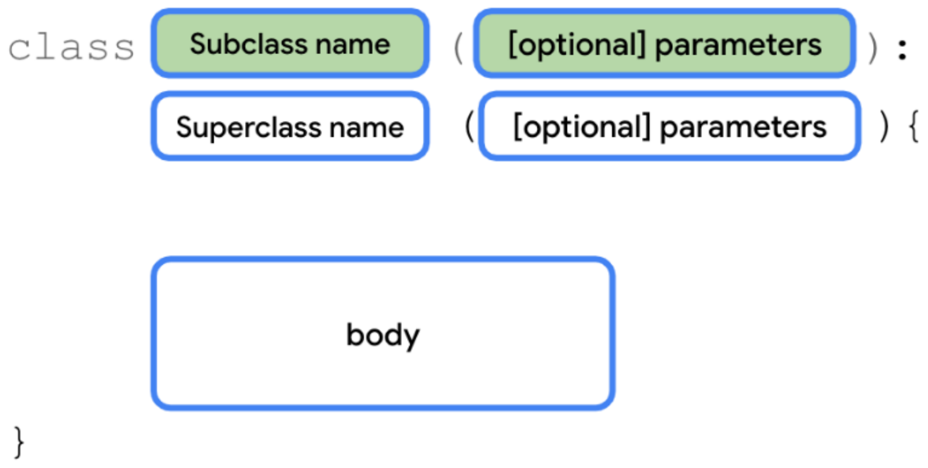


SmartDevice as parent (Super) class whereas SmartTvDevice and SmartLightDevice as child class, which inherit the properties of the parent class.
In Kotlin, all the classes are final, which means that you can't extend them, so you have to define the relationships between them.

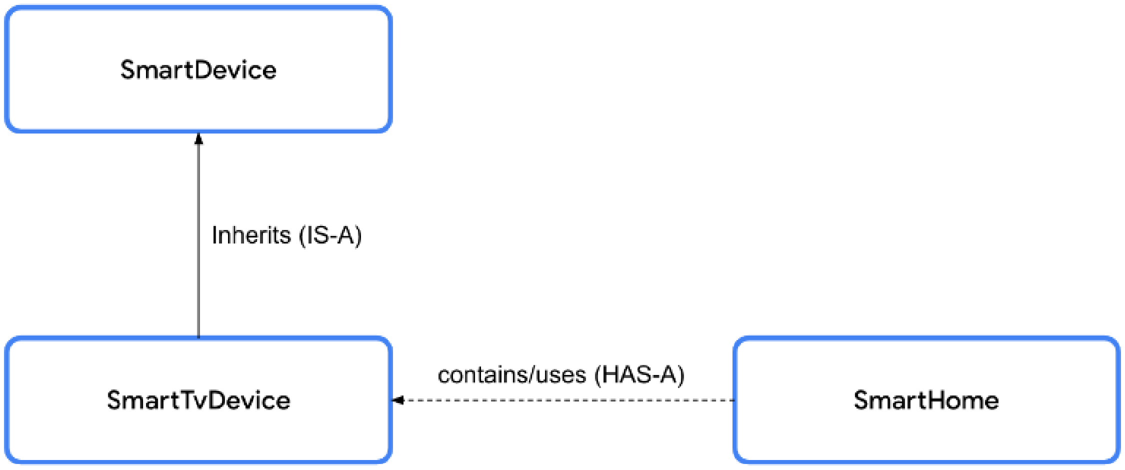
Use **open** keyword to declare parent class
SYNTAX:

open class className(parameters) { }

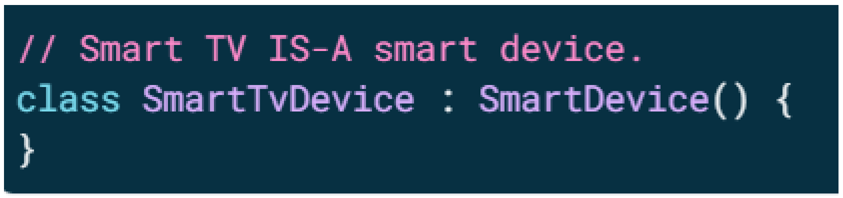
Declare child class:
SYNTAX:



When you use inheritance, you establish a relationship between two classes in something called an ***IS-A relationship***. An object is also an instance of the class from which it inherits. In a ***HAS-A relationship***, an object can own an instance of another class without actually being an instance of that class itself.

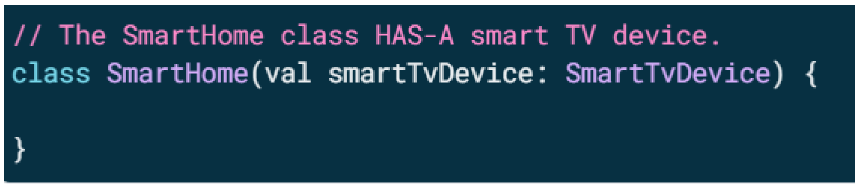


IS-A relationship -



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

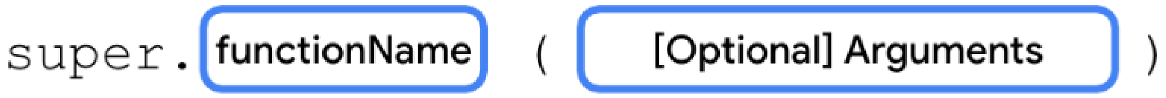
HAS-A relationship -



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

}
```

Overriding

- **superclass methods from subclass**
  Add override keyword in subclass.
- **Use superclass code in subclass with super keyword**



- **Override superclass properties from subclass**

Example (Code snippet):
https://pl.kotl.in/3bqP3khIp?theme=darcula&readOnly=true
https://gist.github.com/priyasu-cx/eca99949b54633d66b7b4f9b665ee972

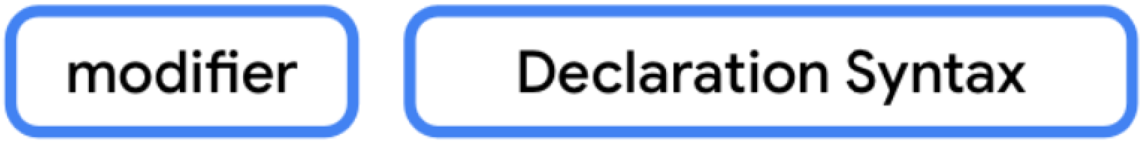
Code Explanation on the basis of Inheritance, Method overriding.

Visibility Modifiers (Abstraction)

Kotlin provides four visibility modifiers:

- public. Default visibility modifier. Makes the declaration accessible everywhere.
- private. Makes the declaration accessible in the same class or source file.
- protected. Makes the declaration accessible in subclasses.
- internal. Makes the declaration accessible in the same module. The internal modifier is similar to private, but you can access internal properties and methods from outside the class as long as it's being accessed in the same module.

SYNTAX



What changes should be done on the previous code to keep the output the same ?

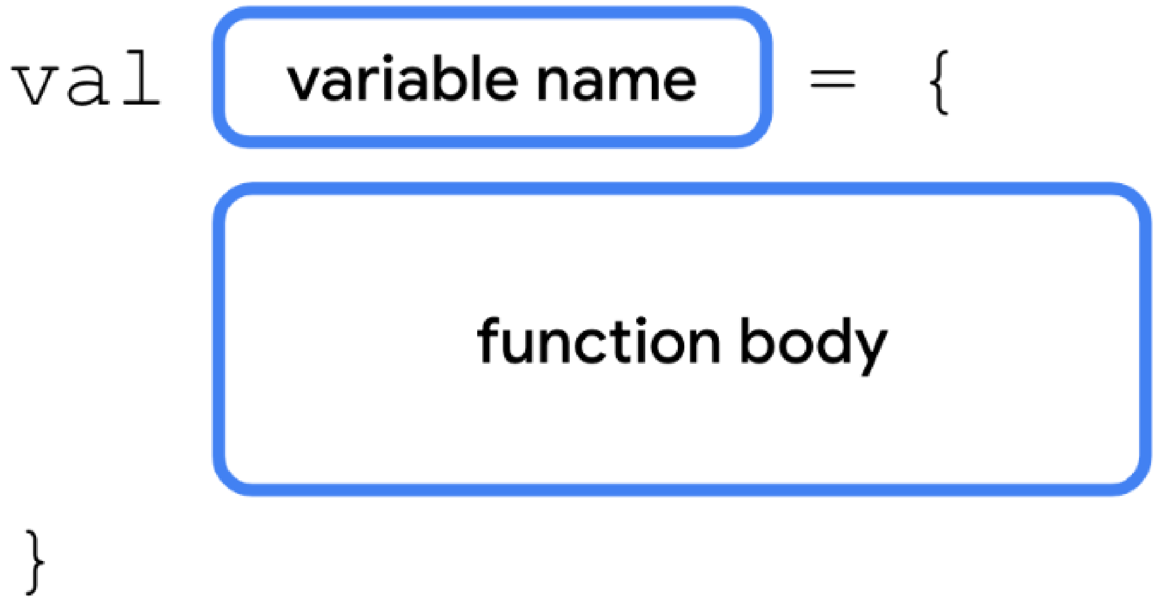
Sol : We shouldn’t allow the class properties to be controlled from outside the class. Add private to class properties.

Function types

Kotlin supports functions as a type.

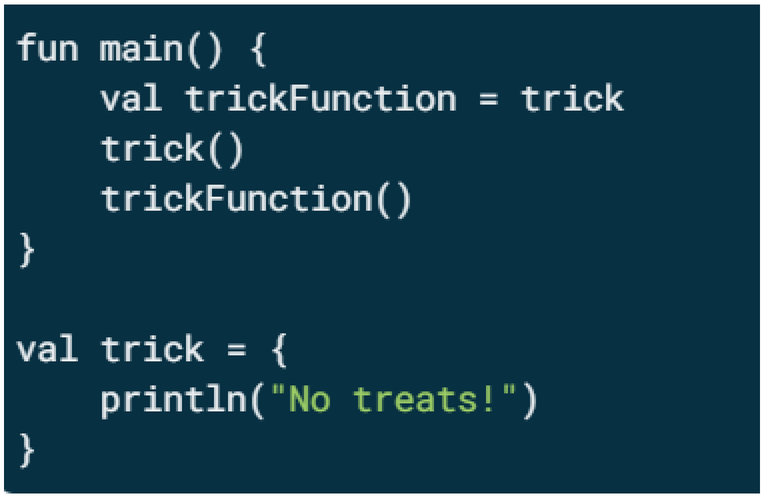
You can store functions in variables, pass them to functions, and return them from functions.

Lambda expression
SYNTAX:



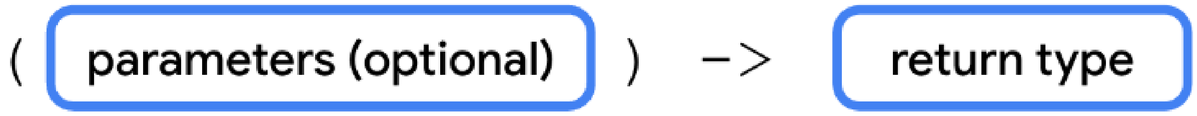
Lambda expressions provide a concise syntax to define a function without the fun keyword. You can store a lambda expression directly in a variable without a function reference on another function.

Example:

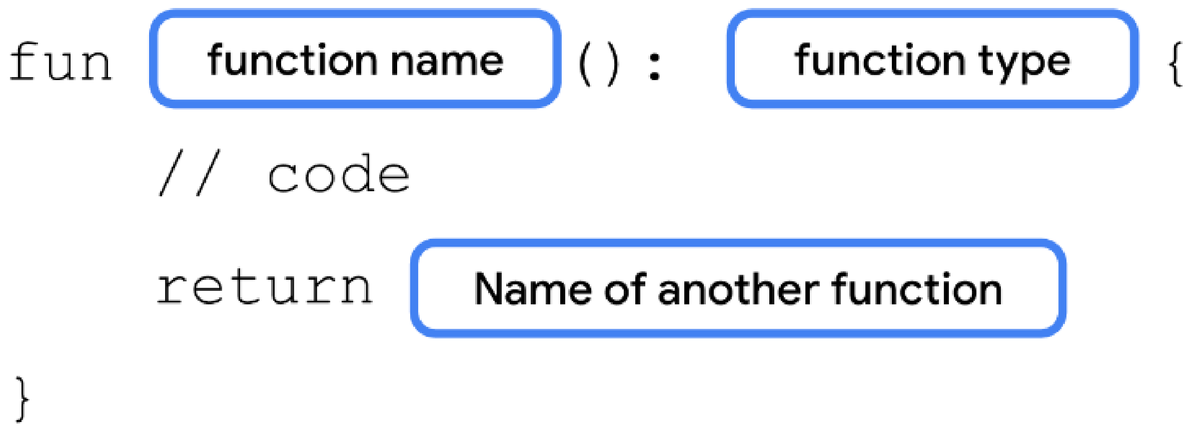


```
fun main() {
    val trickFunction = trick
    trick()
    trickFunction()
}

val trick = {
    println("No treats!")
}
```

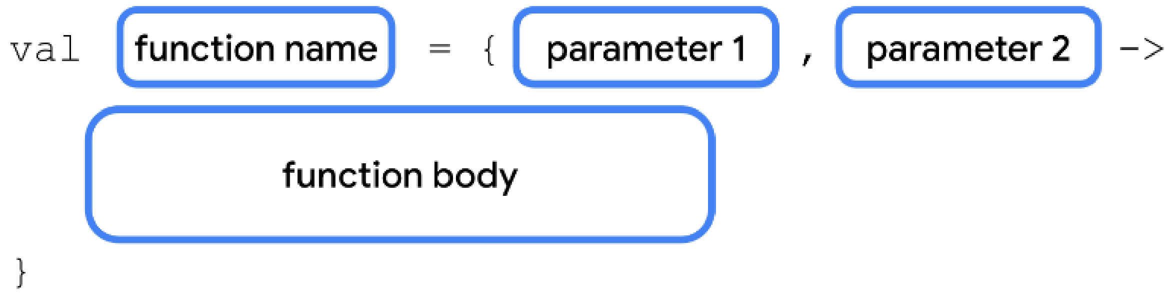
Using functions as a data type
SYNTAX:



Use a function as a return type
SYNTAX:

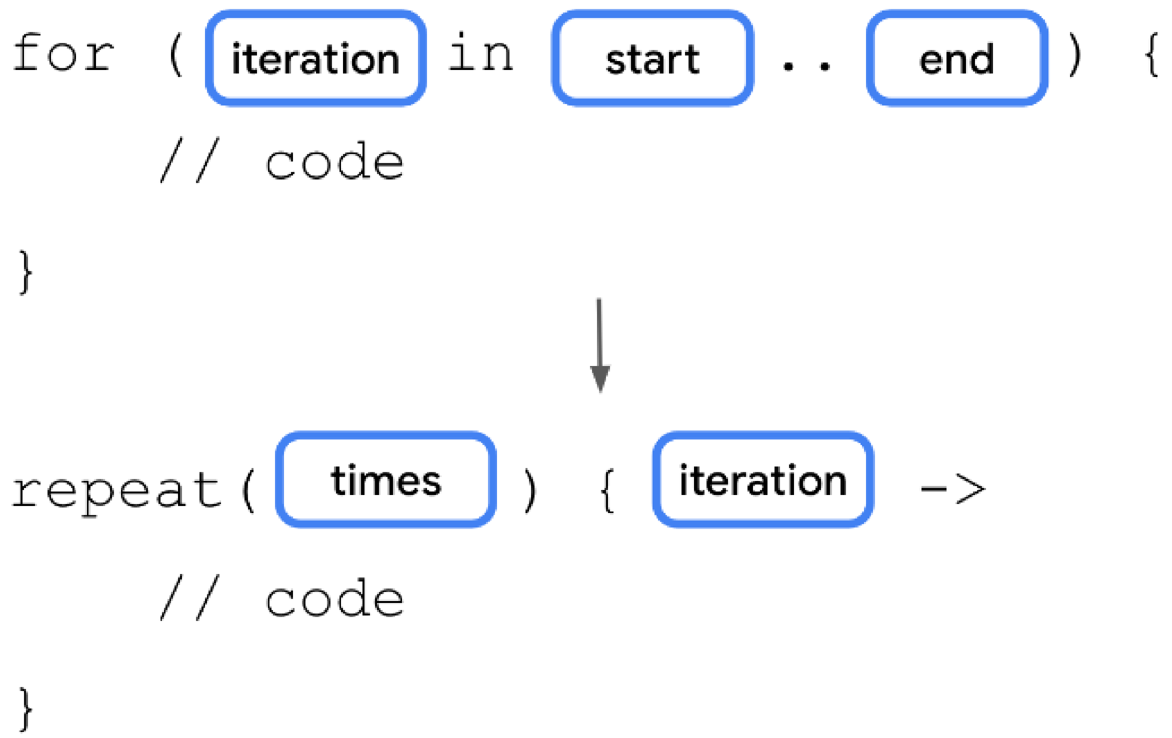


Pass a function to another function as an argument

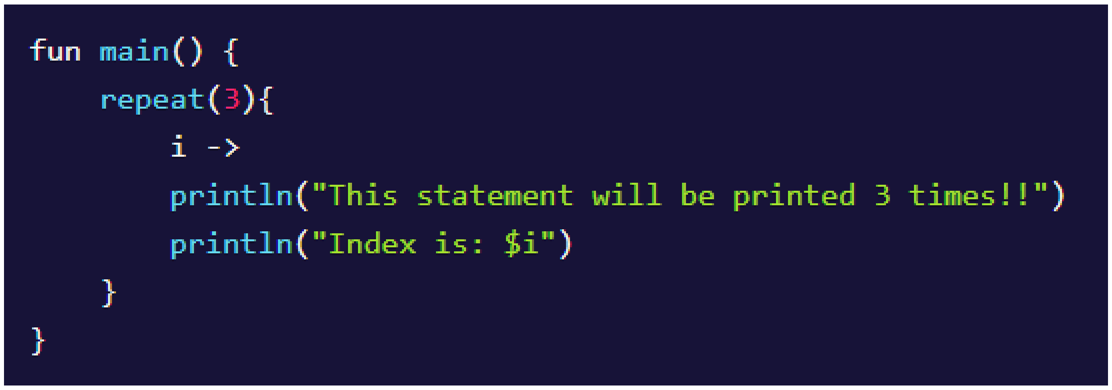


Higher-order-function
When a function returns a function *or* takes a function as an argument, it's called a higher-order function.

Example: repeat() function
It is similar to for loop


for (iteration in start .. end) {
// code
}
repeat(times) { iteration ->
// code
}

// trailing lambda syntax
Example:



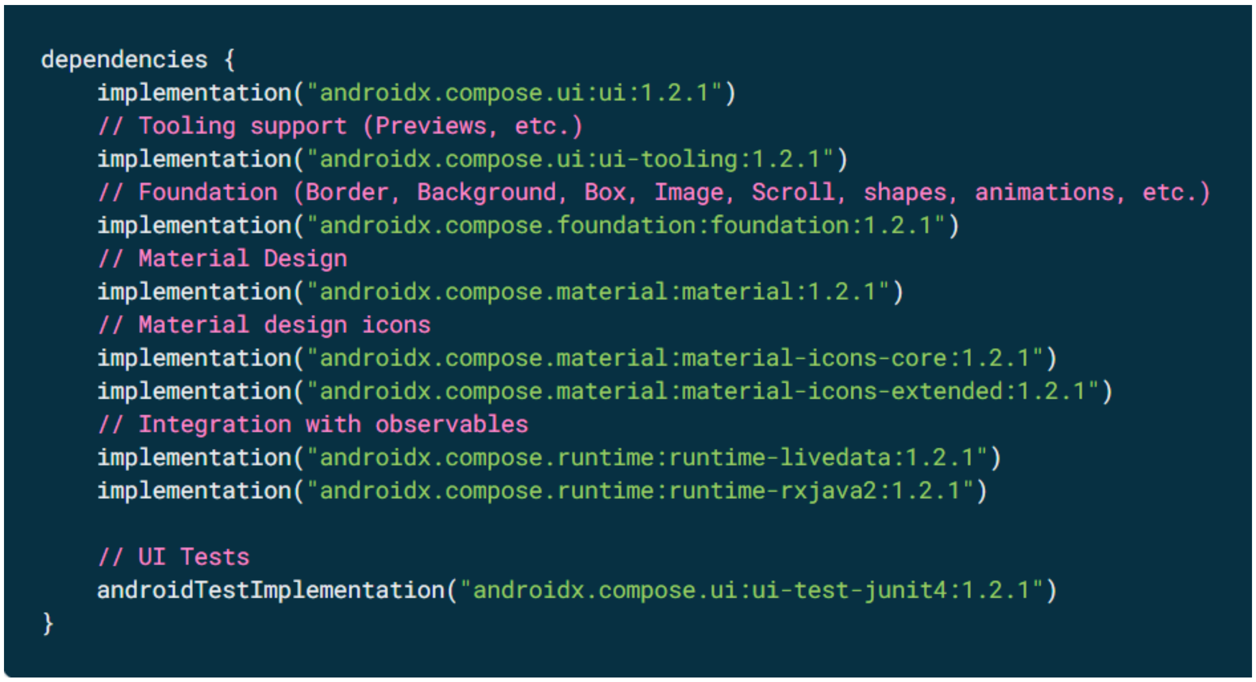
```
fun main() {
    repeat(3){
        i ->
        println("This statement will be printed 3 times!!")
        println("Index is: $i")
    }
}
```

Hands on session - Dice Roller App with Compose

What is Jetpack Compose ?

Jetpack Compose is a modern toolkit for building native Android UI. Jetpack Compose simplifies and accelerates UI development on Android with less code, powerful tools, and intuitive Kotlin APIs.

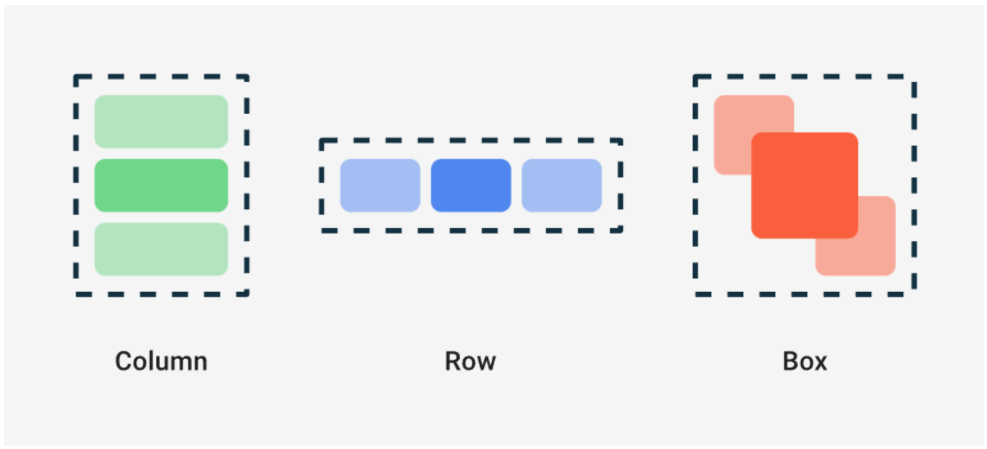
Compose Dependencies



```
dependencies {
    implementation("androidx.compose.ui:ui:1.2.1")
    // Tooling support (Previews, etc.)
    implementation("androidx.compose.ui:ui-tooling:1.2.1")
    // Foundation (Border, Background, Box, Image, Scroll, shapes, animations, etc.)
    implementation("androidx.compose.foundation:foundation:1.2.1")
    // Material Design
    implementation("androidx.compose.material:material:1.2.1")
    // Material design icons
    implementation("androidx.compose.material:material-icons-core:1.2.1")
    implementation("androidx.compose.material:material-icons-extended:1.2.1")
    // Integration with observables
    implementation("androidx.compose.runtime:runtime-livedata:1.2.1")
    implementation("androidx.compose.runtime:runtime-rxjava2:1.2.1")

    // UI Tests
    androidTestImplementation("androidx.compose.ui:ui-test-junit4:1.2.1")
}
```

Compostable functions

Composable functions are the basic building block of Compose. A composable function is a function emitting Unit that describes some part of your UI.



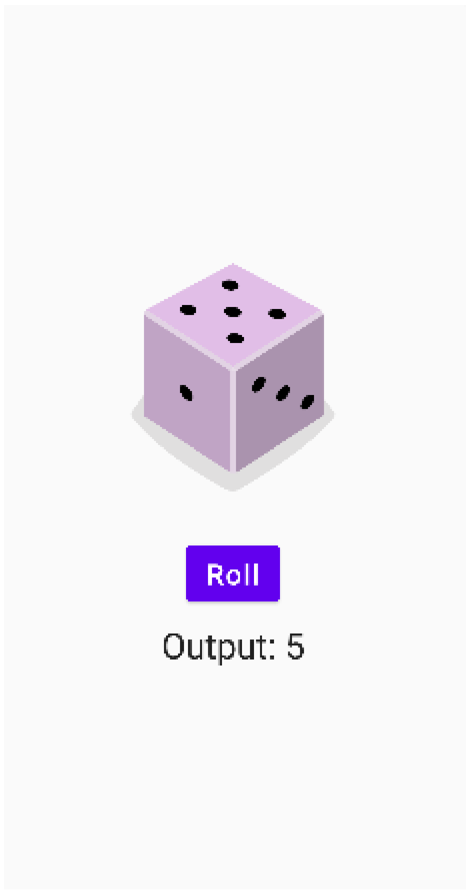
Basic Layout

Using Modifiers in Layouts

Modifiers allow you to decorate or augment a composable. Modifiers let you do these sorts of things:

- Change the composable's size, layout, behavior, and appearance
- Add information, like accessibility labels
- Process user input
- Add high-level interactions, like making an element clickable, scrollable, draggable, or zoomable

App UI -



Resources -
[Download Images]

Hands on session - Tip Calculator App with Compose