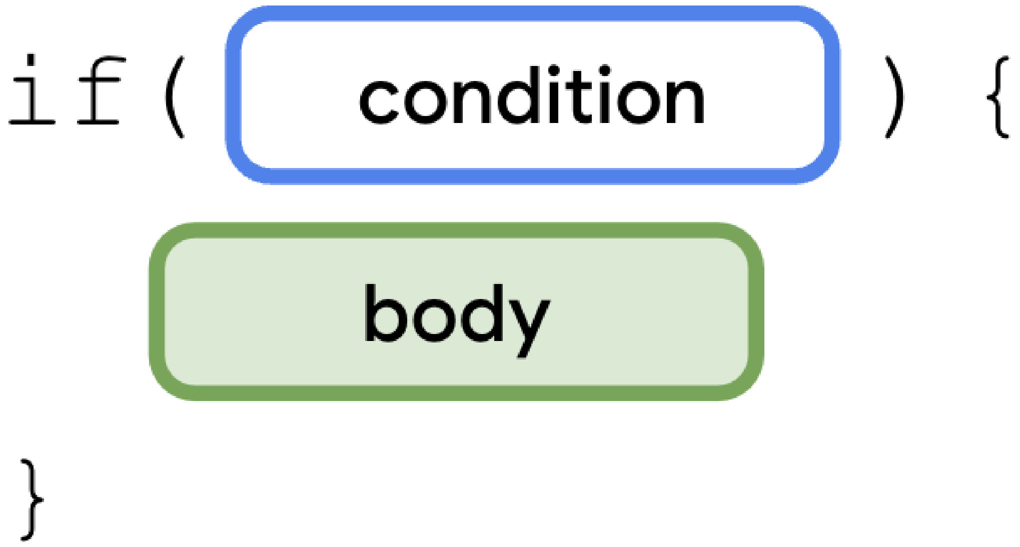
▼ 2.1: Kotlin Fundamentals.

▼ Conditionals in Kotlin.

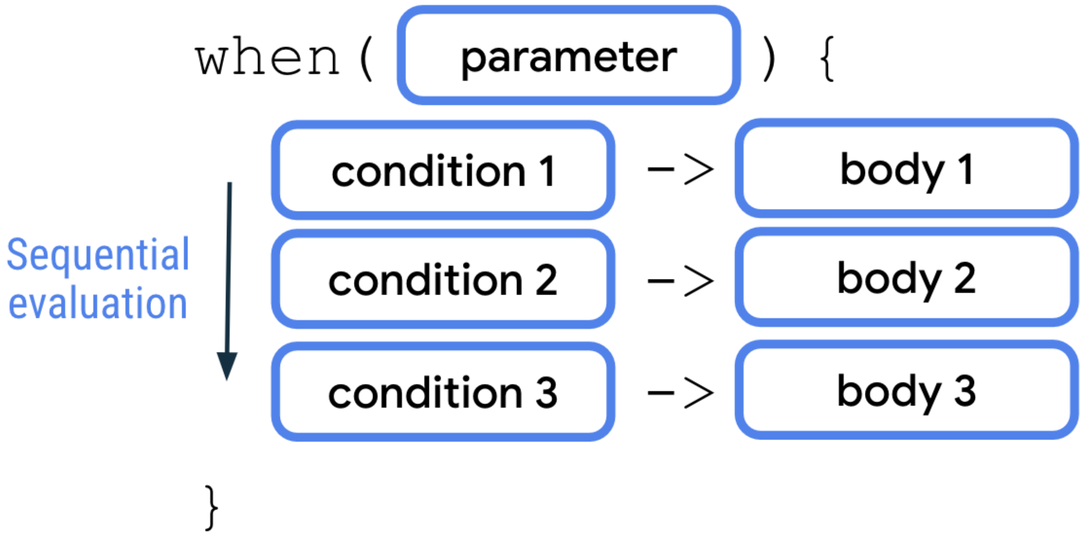
If-Else Statements.



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

    if (trafficLightColor == "Red") {
        println("Stop")
    } else if (trafficLightColor == "Yellow") {
        println("Slow")
    } else if (trafficLightColor == "Green") {
        println("Go")
    }
}
```

Use a When Statement for Multiple Branches.

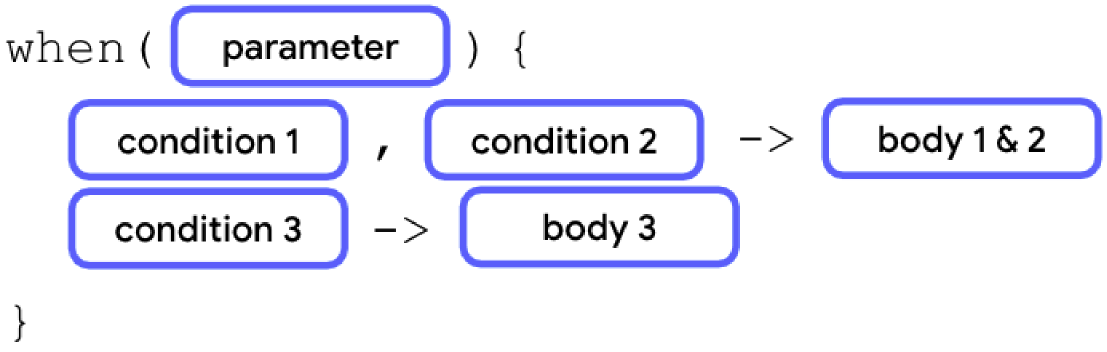
Your `trafficLightColor` program looks more complex with multiple conditions, also known as branching. In Kotlin, when you deal with multiple branches, you can use the `when` statement instead of if-else statements as it improves readability.



A `when` statement accepts a single value through the parameter. The value is then evaluated against each of the conditions sequentially. The corresponding body of the first condition that's met is then executed. Each pair of condition and body is called a branch in `when` statements.

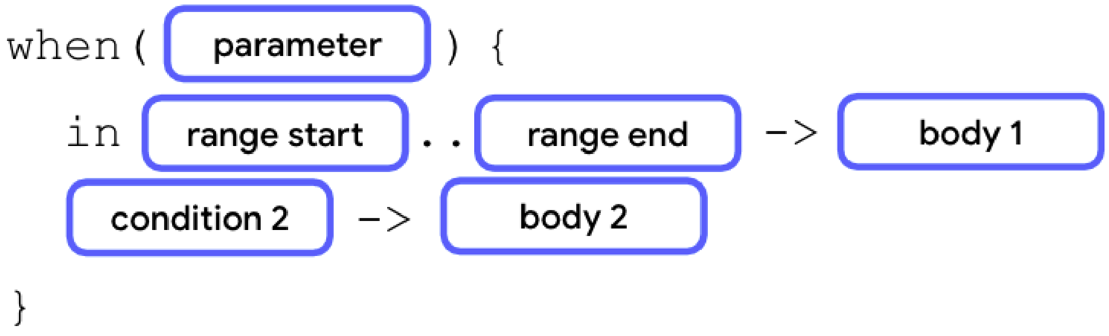
```
fun main() {
    val trafficLightColor = "Yellow"
    when (trafficLightColor) {
        "Red" -> println("Stop")
        "Yellow" -> println("Slow")
        "Green" -> println("Go")
        else -> println("Invalid traffic-light color")
    }
}
```

When you write a `when` statement, you can use a comma to denote multiple conditions that correspond to the same body.



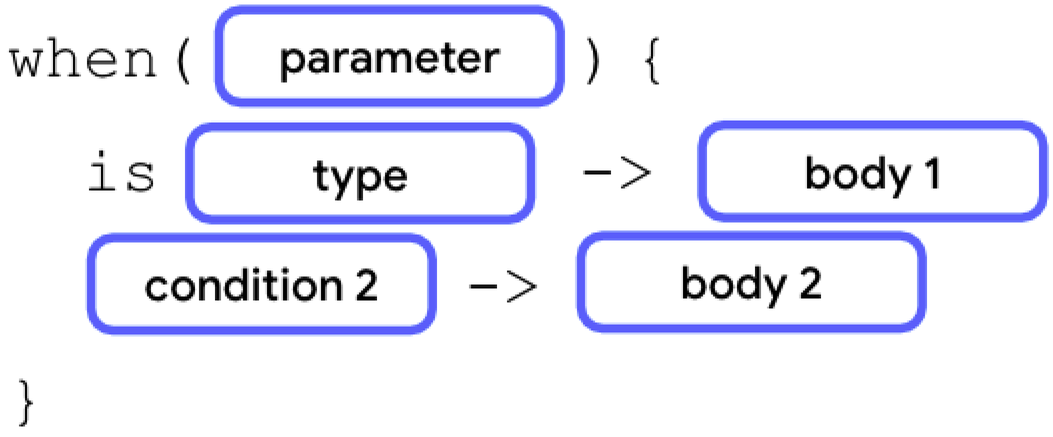
```
fun main() {
    val x = 3
    when (x) {
        2, 3, 5, 7 -> println("x is a prime number between
        else -> println("x isn't a prime number between 1 a
    }
}
```

Besides the comma to denote multiple conditions you can also use the `in` keyword and a range of values in `when` branches.



```
fun main() {
    val x = 3

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

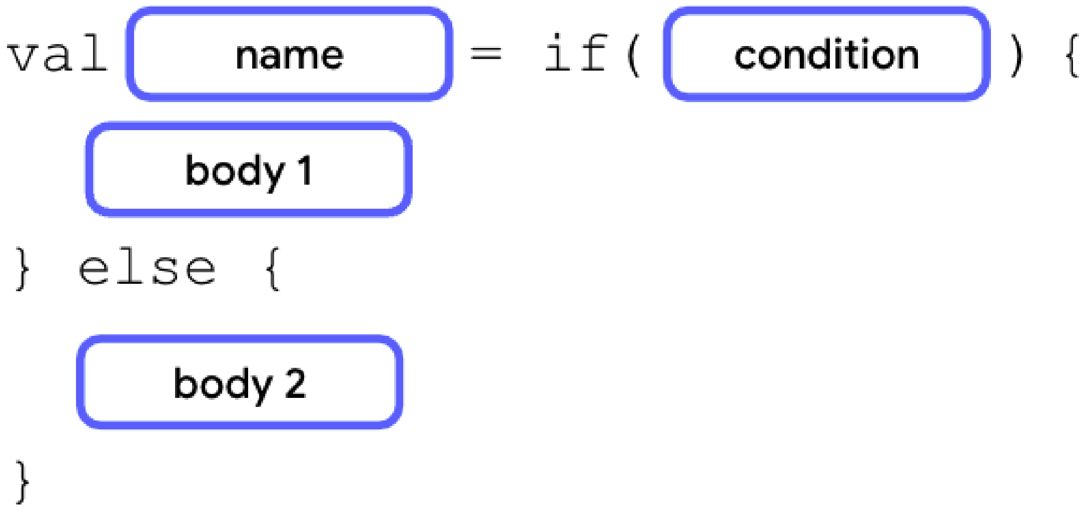
You can use the `is` keyword as a condition to check the data type of an evaluated value.



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

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

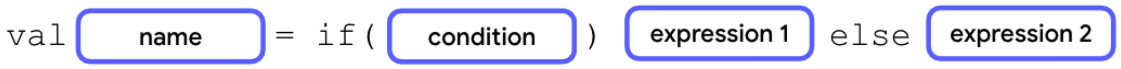
Use `if` / `else` and `when` as Expressions.

You can also use conditionals as expressions to return different values for each branch of condition.



The syntax for conditionals as expressions is similar to statements, but the last line of bodies in each branch need to return a value or an expression, and the conditionals are assigned to a variable.

If the bodies only contain a return value or expression, you can remove the curly braces to make the code more concise.



```
// using if-else statements
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)
}
```

```
// using when statements
fun main() {
    val trafficLightColor = "Amber"

    val message = when(trafficLightColor) {
        "Red" -> "Stop"
        "Yellow", "Amber" -> "Slow"
        "Green" -> "Go"
        else -> "Invalid traffic-light color"
    }
}
```

Conclusion.

- In Kotlin, branching can be achieved with `if/else` or `when` conditionals.
- The body of an `if` branch in an `if/else` conditional is only executed when the boolean expression inside the `if` branch condition returns a `true` value.
- Subsequent `else if` branches in an `if/else` conditional get executed only when previous `if` or `else if` branches return `false` values.
- The final `else` branch in an `if/else` conditional only gets executed when all previous `if` or `else if` branches return `false` values.
- It's recommended to use the `when` conditional to replace an `if/else` conditional when there are more than two branches.
- You can write more complex conditions in `when` conditionals with the comma (`,`), `in` ranges, and the `is` keyword.
- `if/else` and `when` conditionals can work as either statements or expressions.

▼ Nullability in Kotlin.

Nullability is a concept commonly found in many programming languages. It refers to the ability of variables to have an absence of value. In Kotlin, nullability is intentionally treated to achieve `null` safety.

Nullable Variables.

In Kotlin, you can use `null` to indicate that there’s no value associated with the variable.

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

In Kotlin, there’s a distinction between nullable and non-nullable types:

- **Nullable types** are variables that can hold `null`.
- **Non-nullable types** are variables that cannot hold `null`.

A type is only nullable if you explicitly let it hold `null`.

💡 To declare nullable variables in Kotlin, you need to add a `?` operator to the end of the type. For example, a `String?` type can hold either a string or `null`, whereas a `String` type can only hold a string.

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

    favoriteActor = null
    println(favoriteActor)
}
```

Handle Nullable Variables.

Imagine that you want to make the `favoriteActor` variable nullable so that people who don't have a favorite actor can assign the variable to `null`. To access a property of the nullable `favoriteActor` variable, follow these steps:

If you run this function:

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

You get this error message:

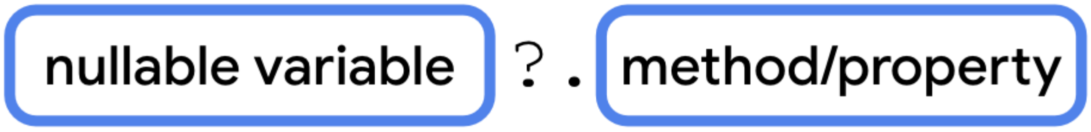
```
❗ Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable
   receiver of type String?
```

This is critical because if there's an attempt to access a member of a variable that's `null` - known as `null` *reference* - during the running of an app, the app crashes because the `null` variable doesn't contain any property. This type of crash is known as a *runtime error* in which the error happens after the code has compiled and runs.

Due to the `null` safety nature of Kotlin, such runtime errors are prevented because the Kotlin compiler forces a `null` *check* for nullable types.

Use the `?.` Safe-call Operator.

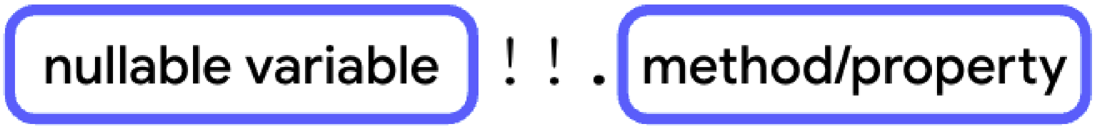
You can use the `?.` safe call operator to access properties of nullable variables.



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

> You can also use the `?.` safe-call operators on non-nullable variables to access a method or property. While the Kotlin compiler won't give any error for this, it's unnecessary because the access of methods or properties for non-nullable variables is always safe.

Use the `!!` not-null Assertion Operator.



You can also use the `!!` not-null assertion operator to access methods or properties of nullable variables.

> As the name suggests, if you use the `!!` not-null assertion, it means that you assert that the value of the variable isn't `null`, *regardless of whether it is or isn't*.

Unlike `?.` safe-call operators, the use of a `!!` not-null assertion operator may result in a `NullPointerException` error being thrown if the nullable variable is indeed `null`. Thus, it should be done only when the variable is always non-nullable or proper exception handling is set in place.

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

You can use the `if` branch in the if-else conditionals to perform `null` checks. To perform `null` checks, you can check that the nullable variable isn't equal to `null` with the `!=` comparison operator.

```
fun main() {
    var favoriteActor: String? = null

    if(favoriteActor != null) {
      println("The number of characters in your favorite ac
    } else {
      println("You didn't input a name.")
    }
}
```

The `?:` Elvis Operator.

With the `?:` Elvis operator, you can add a default value when the `?.` safe-call operator returns `null`. It's similar to an `if/else` expression, but in a more idiomatic way.

If the variable is not `null`, the expression *before* the `?:` executes. If the variable is null, the expression *after* `?:` executes.

```
fun main() {
    var favoriteActor: String? = "Sandra Oh"
    val lengthOfName = favoriteActor?.length ?: 0
    println("The number of characters in your favorite acto
}
// output: The number of characters in your favorite actor
```

Conclusion.

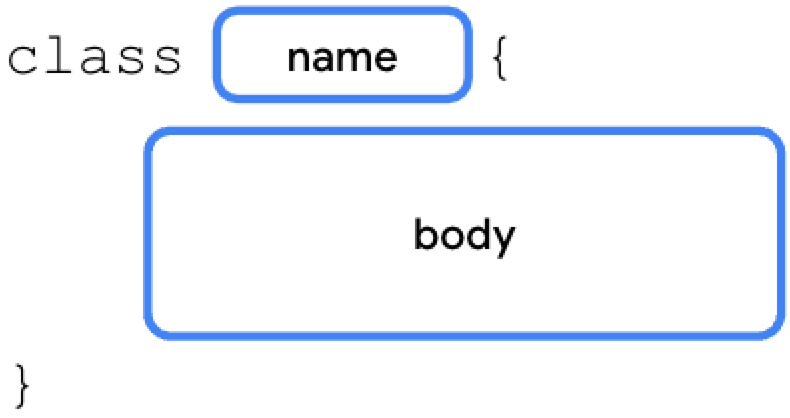
- More about null safety: https://kotlinlang.org/docs/null-safety.html#checking-for-null-in-conditions
- A variable can be set to `null` to indicate that it holds no value.
- Non-nullable variables cannot be assigned `null`.
- Nullable variables can be assigned `null`.

- To access properties of nullable variables, you need to use `?.` safe-call operators or `!!` not-null assertion operators.
- You can use `if/else` statements with `null` checks to access nullable variables in non-nullable contexts.
- You can provide a default value for when a nullable variable is `null` with the `if/else` expression or the `?:` Elvis operator.

▼ Classes and Objects in Kotlin.

Define a Class.

When you define a class, you specify the properties and methods that all objects of that class should have. A class definition starts with the `class` keyword, followed by a name and a set of curly braces. The part of the syntax before the opening curly brace is also referred to as the class header. In the curly braces, you can specify properties and functions for the class.



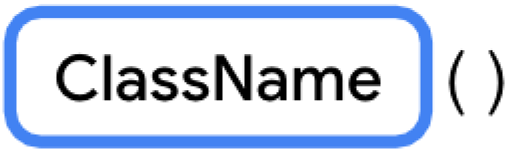
💡 A class consists of three major parts:

- **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.

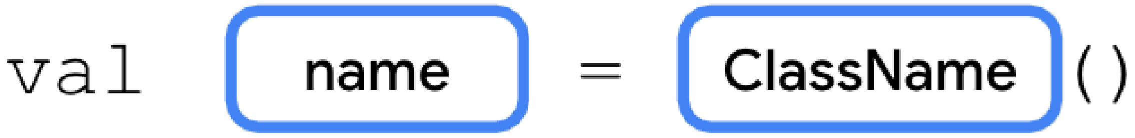
```
class SmartDevice {
    // empty body
}

fun main() {
}
```

Create an Instance of a Class.

With the `SmartDevice` class, you have a blueprint of what a smart device is. To have an *actual* smart device in your program, you need to create a `SmartDevice` object instance. The instantiation syntax starts with the class name followed by a set of parentheses.



To use an object, you create the object and assign it to a variable, similar to how you define a variable.



💡 When you define the variable with the `val` keyword to reference the object, the variable itself is read-only, but the class object remains mutable. This means you can't re-assign another object to the variable, but you can change the object's state when you update its properties' values.

```
class SmartDevice {
    // empty body
}

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

Define Class Methods.

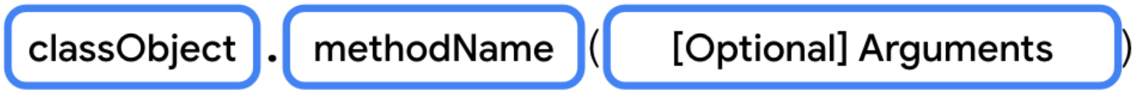
The syntax to define a function in a class is identical to what you learned before. The only difference is that the function is placed in the class body. When you define a *function* in the class body, it's referred to as a member function or a *method*, and it represents the behavior of the class.

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

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

The call to a method in a class is similar to how you called other functions from the `main()` function in the previous codelab. For example:

```
class SmartDevice {
    fun turnOn() {
        // A valid use case to call the turnOff() method co
        turnOff()
        ...
    }
    ...
}
```

💡 To call a class method outside of the class, start with the class object followed by the `.` operator, the name of the function, and a set of parentheses.



```
fun main() {
    val smartTvDevice = SmartDevice()
    smartTvDevice.turnOn()
    smartTvDevice.turnOff()
}
// Output:
// Smart device is turned on.
// Smart device is turned off.
```

Define Class Properties.

While methods define the actions that a class can perform, the properties define the class's characteristics. For example, a smart device has these properties:

- **Name**: Name of the device.
- **Category**: Type of smart device, such as entertainment, utility, or cooking.
- **Device status**: Whether the device is on, off, online, or offline. The device is considered online when it's connected to the internet.

Properties are basically variables that are defined in the class body instead of the function body. This means that the syntax to define properties and variables are identical.

```
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()
}
// Device name is: Android TV
// Smart device is turned on.
// Smart device is turned off.
```

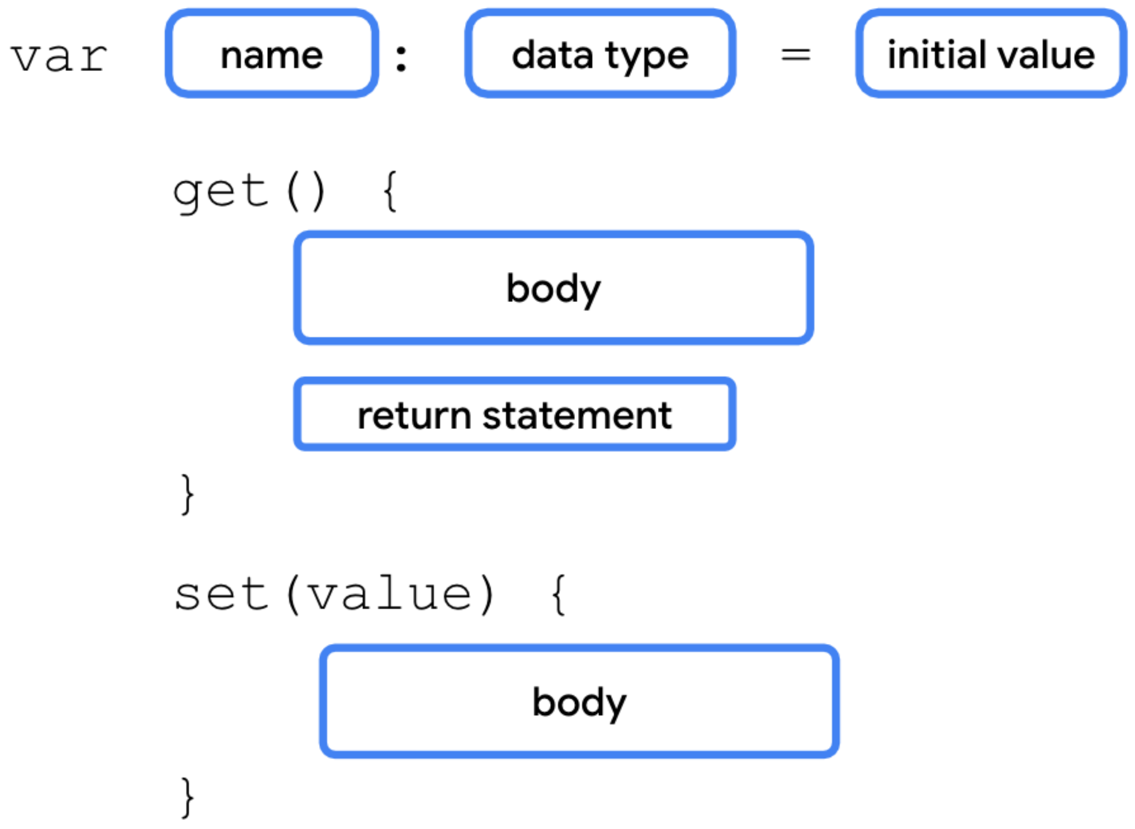
Getter and Setter Functions in Properties.

Properties in a class can do more than just hold values like regular variables. For example, consider a class representing a smart TV.

One common action is adjusting the volume. You could create a property named `speakerVolume` to represent the current volume level, with values ranging from 0 to 100. To ensure the volume stays within this range, you can use a setter function. This function checks that any new value assigned to `speakerVolume` is between 0 and 100, preventing it from going out of bounds.

Similarly, if you want a `name` property to always be stored in uppercase, you can use a getter function to automatically convert any assigned value to uppercase when accessed.

To implement these properties, you need to understand the syntax for defining them. The syntax for a mutable property includes the variable definition and optional `get()` and `set()` functions, as shown in this diagram:



When you don't define the getter and setter function for a property, the Kotlin compiler internally creates the functions. You won't see these lines in your code because they're added by the compiler in the background.

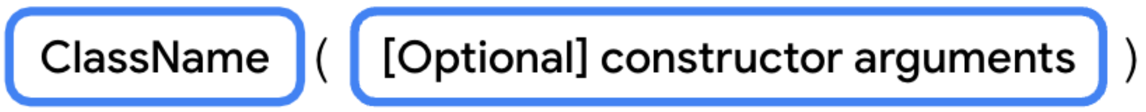
Define a Constructor.

The primary purpose of the constructor is to specify how the objects of the class are created. In other words, constructors initialize an object and make the object ready for use. You can define a constructor with or without parameters.

As it is, the `name` and `category` properties are immutable. You need to ensure that all the instances of the `SmartDevice` class initialize the `name` and `category` properties.

```
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.")
    }
}
```

The constructor now accepts parameters to set up its properties, so the way to instantiate an object for such a class also changes. See:



This is the code representation:

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

There are two main types of constructors in Kotlin:

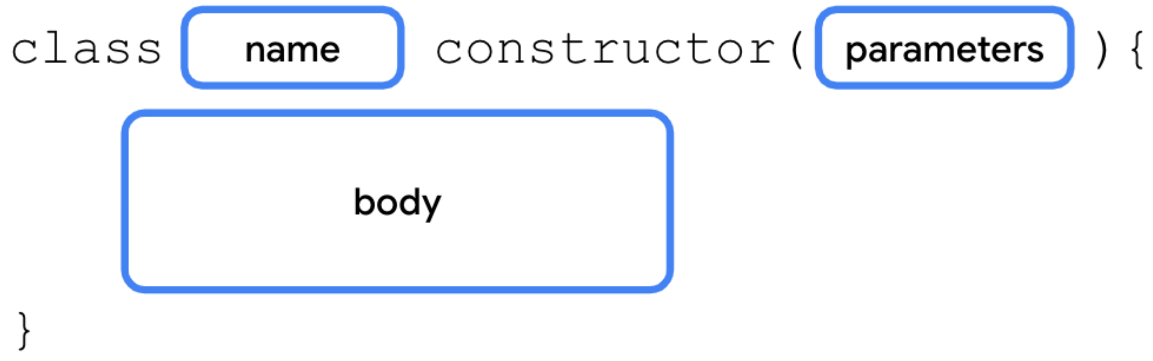
- **Primary constructor**: A class can have only one primary constructor, which is defined as part of the class header. A primary constructor can be a default or parameterized constructor. The primary constructor doesn't have a body, i.e., it can't contain any code.

- **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.

You can use the primary constructor to initialize properties in the class header. The arguments passed to the constructor are assigned to the properties.

The syntax to define a primary constructor starts with the class name followed by the `constructor` keyword and a set of parentheses. The parentheses contain the parameters for the primary constructor. If there's more than one parameter, commas separate the parameter definitions.

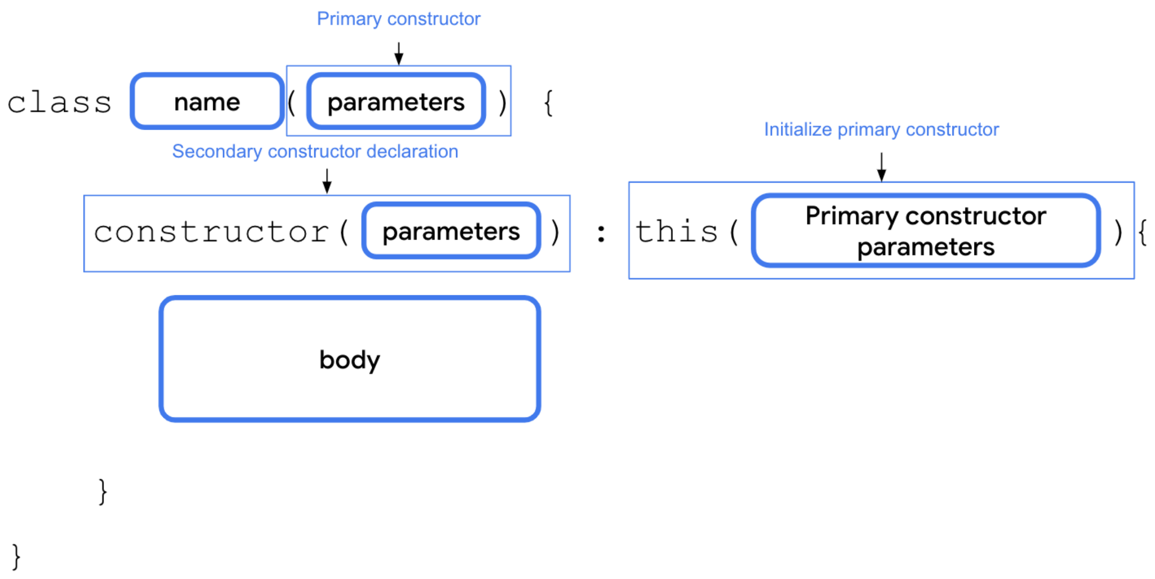


The secondary constructor is enclosed in the body of the class and its syntax includes 3 parts:

- **Secondary constructor declaration**: The secondary constructor definition starts with the `constructor` keyword followed by parentheses. If applicable, the parentheses contain the parameters required by the secondary constructor.
- **Primary constructor initialization**: The initialization starts with a colon followed by the `this` keyword and a set of parentheses. If applicable,

the parentheses contain the parameters required by the primary constructor.

- **Secondary constructor body:** Initialization of the primary constructor is followed by a set of curly braces, which contain the secondary constructor's body.



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

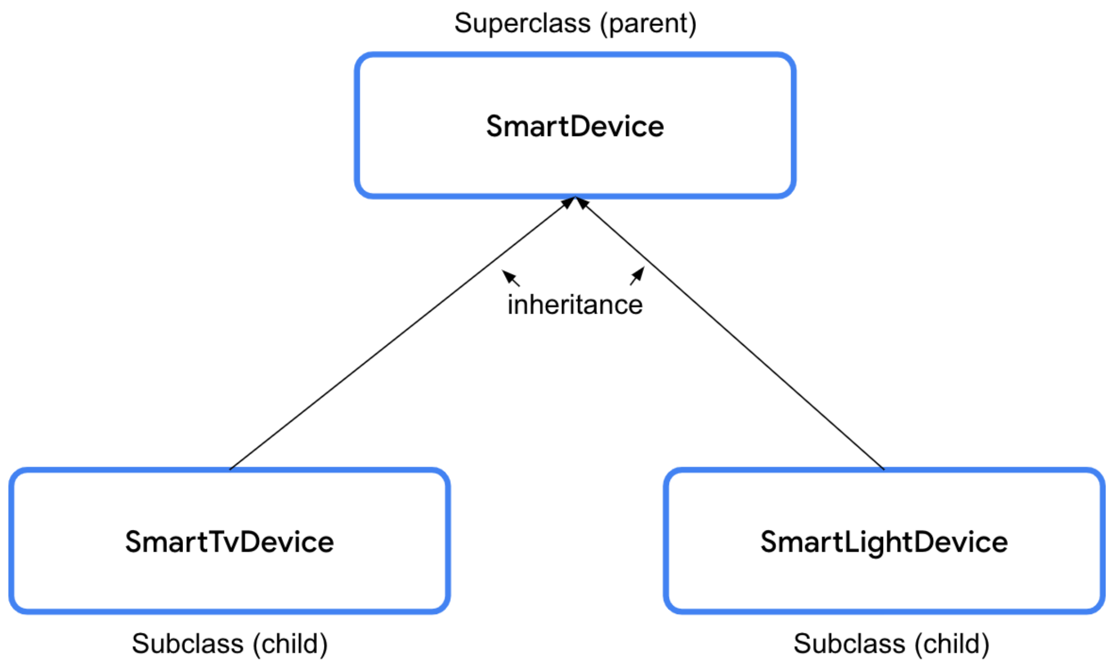
Implement a Relationship Between Classes.

Inheritance lets you build a class upon the characteristics and behavior of another class. It's a powerful mechanism that helps you write reusable code and establish relationships between classes.

In short, all smart devices have different features, yet share some common characteristics. You can either duplicate these common characteristics to each of the smart device classes or make the code reusable with inheritance.

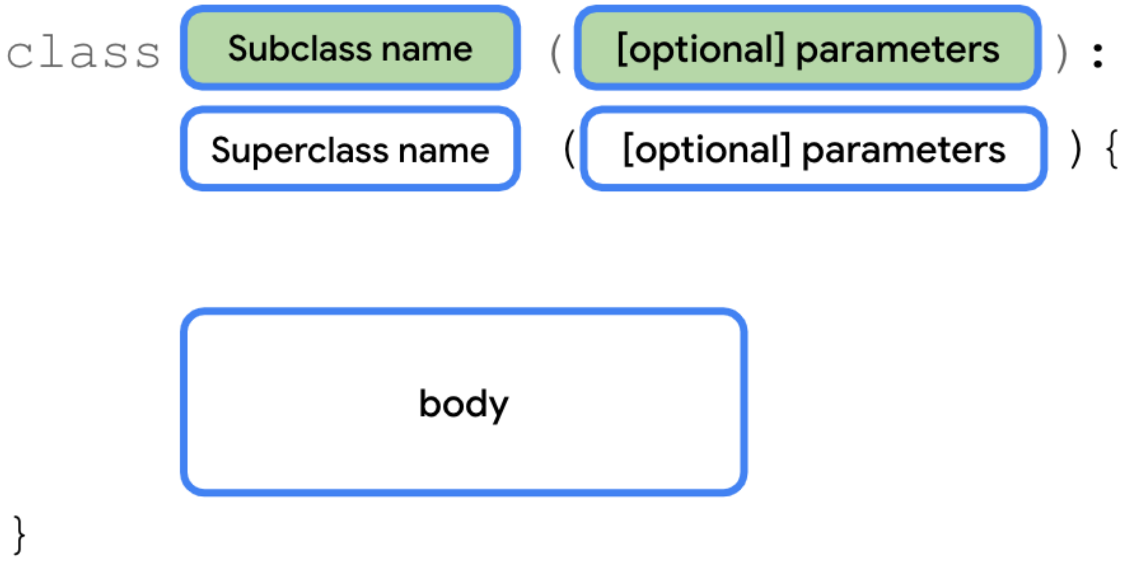
To do so, you need to create a `SmartDevice` parent class, and define these common properties and behaviors. Then, you can create child classes, such as the `SmartTvDevice` and `SmartLightDevice` classes, which inherit the properties of the parent class.

In programming terms, we say that the SmartTvDevice and SmartLightDevice classes extend the SmartDevice parent class. The parent class is referred to as a superclass and the child class as a subclass.



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

The `open` keyword informs the compiler that this class is extendable, so now other classes can extend it.



```
class SmartTvDevice(deviceName: String, deviceCategory: Str
    SmartDevice(name = deviceName, category = deviceCatego
}
```

The `constructor` definition for `SmartTvDevice` doesn't specify whether the properties are mutable or immutable. This means that the `deviceName` and `deviceCategory` parameters are merely `constructor` parameters instead of class properties. You won't be able to use them in the class, but simply pass them to the superclass constructor.

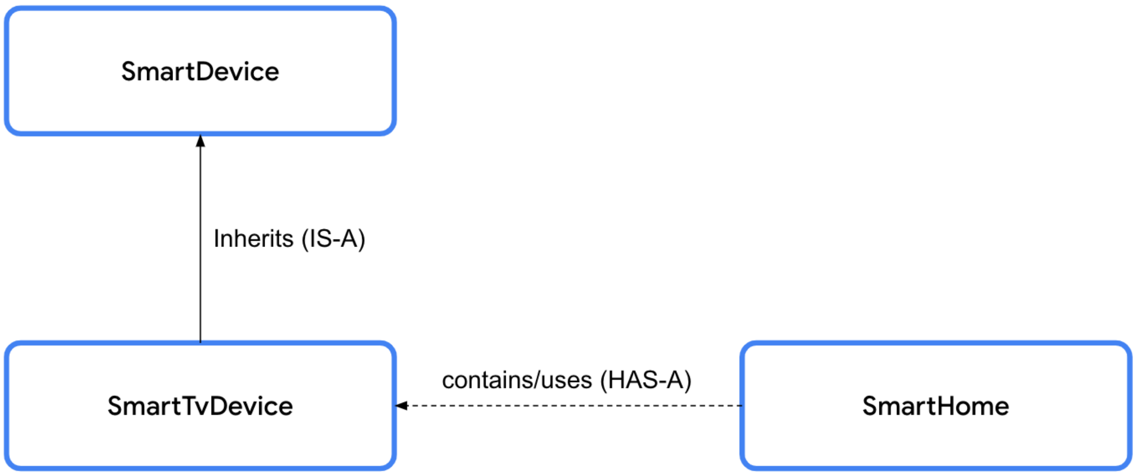
```
class SmartTvDevice(deviceName: String, deviceCategory: Str
    SmartDevice(name = deviceName, category = deviceCatego

    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 $speakerVolum
    }

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

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

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

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

Relationships Between Classes.

When you use inheritance, you establish a relationship between 2 classes in something called a 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 Relationships.

When you specify an IS-A relationship between the `SmartDevice` superclass and `SmartTvDevice` subclass, it means that whatever the `SmartDevice` superclass can do, the `SmartTvDevice` subclass can do. The relationship is unidirectional, so you can say that every smart TV *is a* smart device, but you can't say that every smart device *is a* smart TV.

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

HAS-A Relationships.

For example, you will probably use the smart TV in your home. In this case, there's a relationship between the smart TV and the home. The home contains a smart device or, in other words, the home *has a* smart device. The *HAS-A* relationship between two classes is also referred to as *composition*.

So far, you created a couple of smart devices. Now, you create the `SmartHome` class, which contains smart devices. The `SmartHome` class lets you interact with the smart devices.

```
class SmartLightDevice(deviceName: String, deviceCategory:
    SmartDevice(name = deviceName, category = deviceCategor
    ...

}

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

    fun turnOnTv() {
        smartTvDevice.turnOn()
    }

    fun turnOffTv() {
        smartTvDevice.turnOff()
    }

    fun increaseTvVolume() {
        smartTvDevice.increaseSpeakerVolume()
    }

    fun changeTvChannelToNext() {
        smartTvDevice.nextChannel()
    }

    fun turnOnLight() {
        smartLightDevice.turnOn()
    }

    fun turnOffLight() {
        smartLightDevice.turnOff()
    }

    fun increaseLightBrightness() {
        smartLightDevice.increaseBrightness()
    }

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

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

Override Superclass Methods from Subclasses.

As discussed earlier, even though the turn-on and turn-off functionality is supported by all the smart devices, the way in which they perform the functionality differs.

To provide this device-specific behavior, you need to override the `turnOn()` and `turnOff()` methods defined in the superclass. To override means to intercept the action, typically to take manual control. When you override a method, the method in the subclass interrupts the execution of the method defined in the superclass and provides its own execution.

```
open class SmartDevice(val name: String, val category: Str

    var deviceStatus = "online"

    open fun turnOn() {
        // function body
    }

    open fun turnOff() {
        // function body
    }
}
```

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

    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 $
    }

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

💡 The `override` keyword informs the Kotlin runtime to execute the code enclosed in the method defined in the subclass.

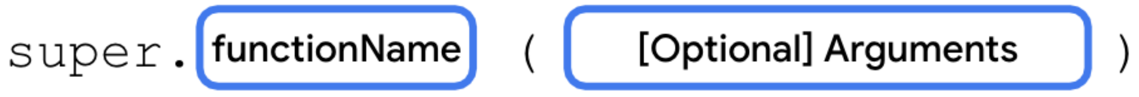
```
class SmartTvDevice(deviceName: String, deviceCategory: Str
        : SmartDevice(name = deviceName, category = deviceC

    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 $
                "set to $channelNumber."
        )
    }
    override fun turnOff() {
        deviceStatus = "off"
        println("$name turned off")
    }
}
```

```
fun main() {
    var smartDevice: SmartDevice = SmartTvDevice("Android 
    smartDevice.turnOn()

    smartDevice = SmartLightDevice("Google Light", "Utilit
    smartDevice.turnOn()
}
// Output:
// Android TV is turned on. Speaker volume is set to 2 and
// Google Light turned on. The brightness level is 2.
```

This is an example of **polymorphism**. The code calls the `turnOn()` method on a variable of `SmartDevice` type and, depending on what the actual value of the variable is, different implementations of the `turnOn()` method can be executed.

Reuse Superclass Code in Subclasses with the `super` Keyword.

> To call the overridden method in the superclass from the subclass, you need to use the `super` keyword. Instead of using a `.` operator between the object and method, you need to use the `super` keyword, which informs the Kotlin compiler to call the method on the superclass instead of the subclass.



```
open class SmartDevice(val name: String, val category: Stri

    var deviceStatus = "online"

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

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

```
class SmartTvDevice(deviceName: String, deviceCategory: Str
    SmartDevice(name = deviceName, category = deviceCatego

    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 $channelNumbe
    }

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

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

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

    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 $
    }

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

Override Superclass Properties from Subclasses.

Similar to methods, you can also override properties with the same steps.

To override the `deviceType`property: In the `SmartDevice` superclass on the line after the `deviceStatus` property, use the `open` and `val`keywords to define a `deviceType` property set to an `"unknown"` string:

```
open class SmartDevice(val name: String, val category: Stri

    var deviceStatus = "online"

    open val deviceType = "unknown"
    ...
}
```

In the `SmartTvDevice` class, use the `override` and `val` keywords to define a `deviceType` property set to a `“Smart TV”` string.

```
class SmartTvDevice(deviceName: String, deviceCategory: Str
    SmartDevice(name = deviceName, category = deviceCategor

    override val deviceType = "Smart TV"
    ...
}
```

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

    override val deviceType = "Smart Light"

    ...

}
```

Visibility Modifiers.

Visibility modifiers play an important role to achieve encapsulation:

- In a *class*, they let you hide your properties and methods from unauthorized access outside the class.
- In a *package*, they let you hide the classes and interfaces from unauthorized access outside the package.

Kotlin provides 4 visibility modifiers:

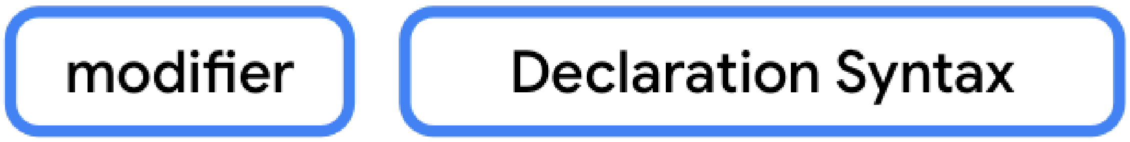
- `public`: Default visibility modifier. Makes the declaration accessible everywhere. The properties and methods that you want used outside the class are marked as public.
- `private`: Makes the declaration accessible in the same class or source file.
- `protected`: Makes the declaration accessible in subclasses. The properties and methods that you want used in the class that defines them & the subclasses are marked with the `protected` visibility modifier.
- `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.

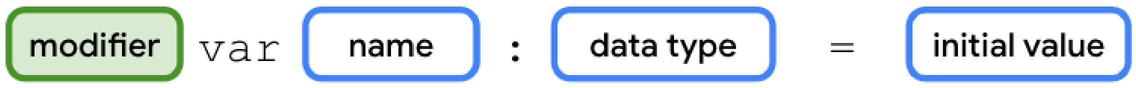
💡 A *module* is a collection of source files and build settings that let you divide your project into discrete units of functionality. Your project can have one or many modules. You can independently build, test, and debug each module.

💡 A *package* is like a directory or a folder that groups related classes, whereas a module provides a container for your app's source code, resource files, and app-level settings. A module can contain multiple packages.

The visibility modifier should be placed before the declaration syntax, while declaring the class, method, or properties as you can see in this diagram:

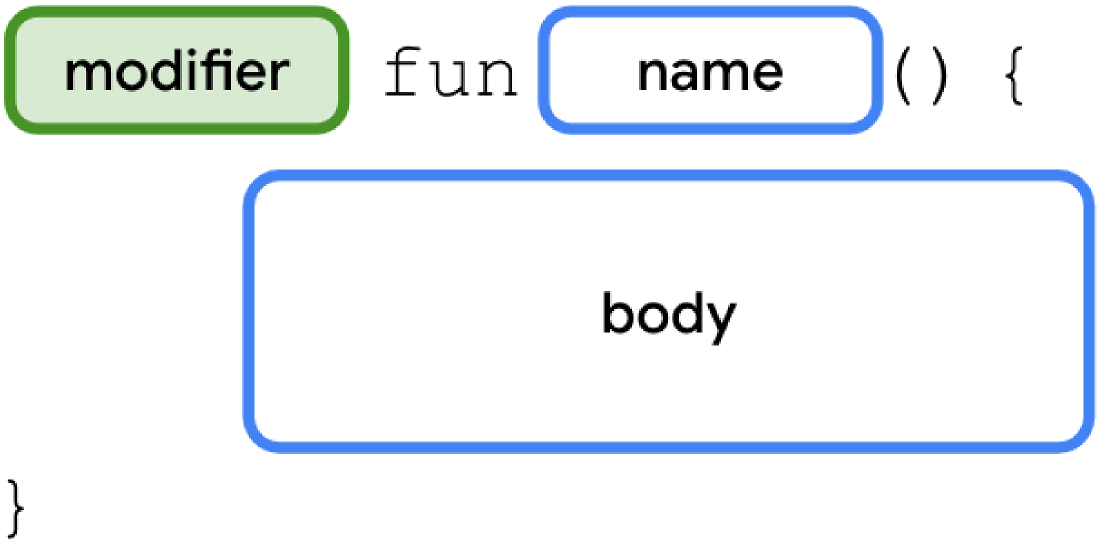


For properties:



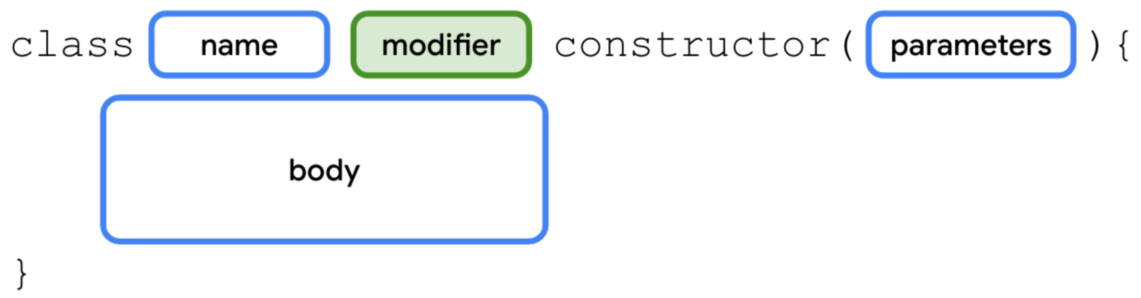
```
open class SmartDevice(val name: String, val category: Str

    ...

    private var deviceStatus = "online"
    ...
}
```

For methods:



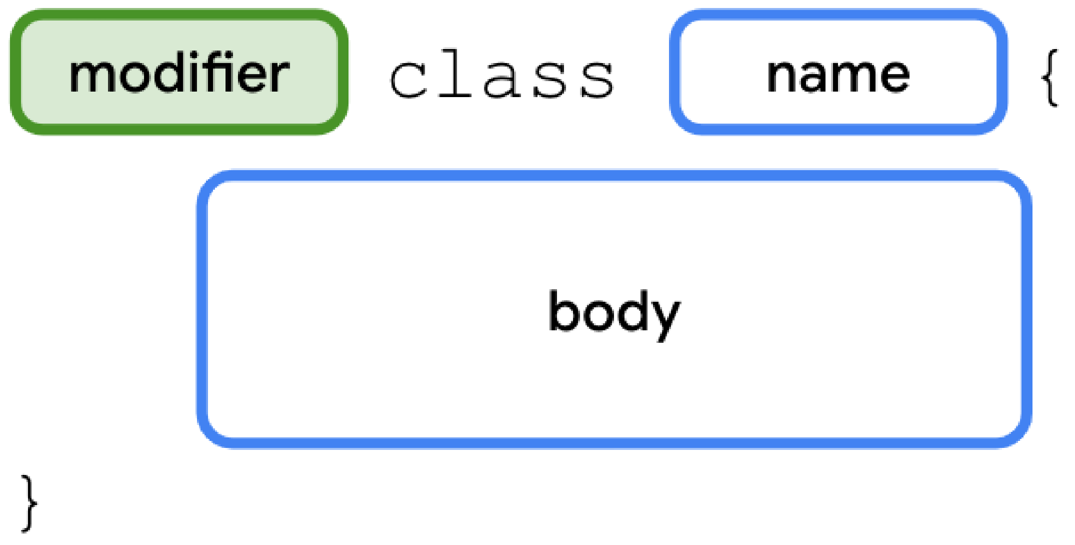
```
class SmartTvDevice(deviceName: String, deviceCategory: Str
    SmartDevice(name = deviceName, category = deviceCatego

    ...

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

    ...
}
```

For constructors:



```
open class SmartDevice protected constructor (val name: Str

    ...

}
```

For classes:



```
internal open class SmartDevice(val name: String, val categ

    ...

}
```

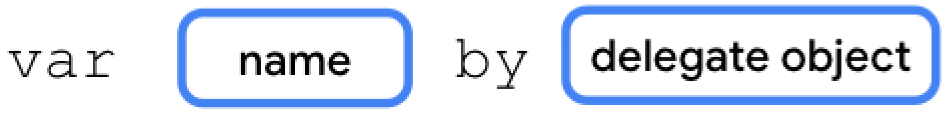
Ideally, you should strive for strict visibility of properties and methods, so declare them with the `private` modifier as often as possible. If you can't keep them private, use the `protected` modifier. If you can't keep them protected, use the `internal` modifier. If you can't keep them internal, use the `public` modifier.

Modifier	Accessible in same class	Accessible in subclass	Accessible in same module	Accessible outside module
`private`	✔	X	X	X
`protected`	✔	✔	X	X
`internal`	✔	✔	✔	X
`public`	✔	✔	✔	✔

Define Property Delegates.

Properties in Kotlin use a backing field to hold their values in memory. You use the `field` identifier to reference it.

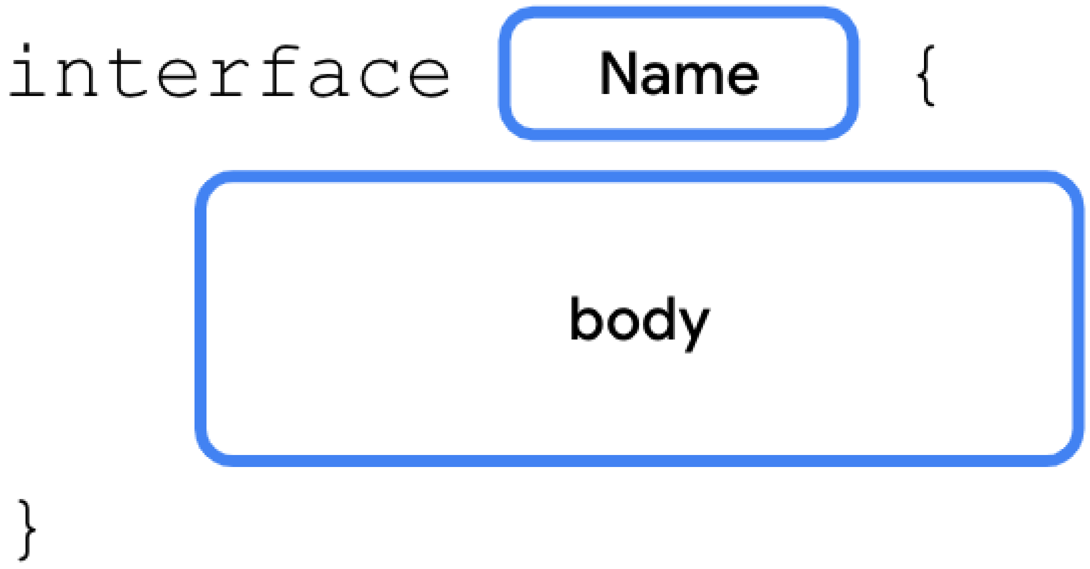
The syntax to create property delegates starts with the declaration of a variable followed by the by keyword, and the delegate object that handles the getter and setter functions for the property.



Before you implement the class to which you can delegate the implementation, you need to be familiar with *interfaces*.

💡 An interface is a contract to which classes that implement it need to adhere. It focuses on *what to do* instead of *how to do* the action. In short, an interface helps you achieve *abstraction*.

For example, before you build a house, you inform the architect about what you want. You want a bedroom, kid's room, living room, kitchen, and a couple of bathrooms. In short, you specify *what you want* and the architect specifies *how to achieve it*.



With interfaces, the class implements the interface. The class provides implementation details for the methods and properties declared in the interface.

To create the delegate class for the `var` type, you need to implement the `ReadWriteProperty` interface. Similarly, you need to implement the `ReadOnlyProperty` for the `val` type.

Here the steps to create the delegate for the `var` type:
1. Before the
`main()` function, create a `RangeRegulator` class that implements the `ReadWriteProperty<Any?,` `Int>` interface:

💡 The `KProperty` is an interface that represents a declared property and lets you access the metadata on a delegated property. It's good to have high-level information about what the `KProperty` is.

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

class SmartTvDevice(deviceName: String, deviceCategory: Str
    SmartDevice(name = deviceName, category = deviceCategor

    override val deviceType = "Smart TV"

    private var speakerVolume by RangeRegulator(initialValu

    private var channelNumber by RangeRegulator(initialValu

    ...

}

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

    var fieldData = initialValue

    override fun getValue(thisRef: Any?, property: KPropert
        return fieldData
    }

    override fun setValue(thisRef: Any?, property: KPropert
        if (value in minValue..maxValue) {
            fieldData = value
        }
    }
}
```

```
...
```

The Entire Solution.

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

open class SmartDevice(val name: String, val category: Stri

    var deviceStatus = "online"
        protected set

    open val deviceType = "unknown"

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

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

class SmartTvDevice(deviceName: String, deviceCategory: St
    SmartDevice(name = deviceName, category = deviceCatego

    override val deviceType = "Smart TV"

    private var speakerVolume by RangeRegulator(initialVal

    private var channelNumber by RangeRegulator(initialVal

    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 $
                "set to $channelNumber."
        )
    }

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

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

    override val deviceType = "Smart Light"

    private var brightnessLevel by RangeRegulator(initialVa

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

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

    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: KProper
        return fieldData
    }

    override fun setValue(thisRef: Any?, property: KProper
        if (value in minValue..maxValue) {
            fieldData = value
        }
    }
}

fun main() {
    var smartDevice: SmartDevice = SmartTvDevice("Android T
    smartDevice.turnOn()

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

Conclusion.

- There are 4 main principles of OOP:
    - Encapsulation.
    - Abstraction.
    - Inheritance.
    - Polymorphism.
- Classes are defined with the `class` keyword, and contain properties and methods.
- Properties are similar to variables except properties can have custom getters and setters.
- A constructor specifies how to instantiate objects of a class.
- You can omit the `constructor` keyword when you define a primary constructor.
- Inheritance makes it easier to reuse code.
- The IS-A relationship refers to inheritance.
- The HAS-A relationship refers to composition.
- Visibility modifiers play an important role in the achievement of encapsulation.

- Kotlin provides four visibility modifiers: the `public`, `private`, `protected`, and `internal` modifiers.
- A property delegate lets you reuse the getter and setter code in multiple classes.

▼ Practice: Kotlin Fundamentals.

Movie-Ticket Price.

Movie tickets are typically priced differently based on the age of moviegoers.

In the initial code provided in the following code snippet, write a program that calculates these age-based ticket prices:

- A children's ticket price of $15 for people 12 years old or younger.
- A standard ticket price of $30 for people between 13 and 60 years old. On Mondays, discount the standard ticket price to $25 for this same age group.
- A senior ticket price of $20 for people 61 years old and older. Assume that the maximum age of a moviegoer is 100 years old.
- A `-1` value to indicate that the price is invalid when a user inputs an age outside of the age specifications.

```
fun main() {
    val child = 5
    val adult = 28
    val senior = 87

    val isMonday = true

    println("The movie ticket price for a person aged $chi
    println("The movie ticket price for a person aged $adu
    println("The movie ticket price for a person aged $sen
}

fun ticketPrice(age: Int, isMonday: Boolean): Int {
    return when(age) {
        in 0..12 -> 15
        in 13..60 -> if (isMonday) 25 else 30
        in 61..100 -> 20
        else -> -1
    }
}
```

Temperature Converter.

There are three main temperature scales used in the world: Celsius, Fahrenheit, and Kelvin.

In the initial code provided in the following code snippet, write a program that converts a temperature from one scale to another with these formulas:

- Celsius to Fahrenheit: ° *F* = 9/5 (° *C*) + 32
- Kelvin to Celsius: ° *C* = *K* - 273.15
- Fahrenheit to Kelvin: *K* = 5/9 (° *F* - 32) + 273.15

Note that the `String.format("%.2f", /* measurement */ )` method is used to convert a number into a `String` type with 2 decimal places.

The solution requires you to pass a function as a parameter to the `printFinalTemperature()` function. The most succinct solution passes lambda expressions as the arguments, uses the `it` parameter reference in place of the parameter names, and makes use of trailing lambda syntax.

```
fun main() {
        printFinalTemperature(27.0, "Celsius", "Fahrenheit
        printFinalTemperature(350.0, "Kelvin", "Celsius")
        printFinalTemperature(10.0, "Fahrenheit", "Kelvin"
}

fun printFinalTemperature(
    initialMeasurement: Double,
    initialUnit: String,
    finalUnit: String,
    conversionFormula: (Double) -> Double
) {
    val finalMeasurement = String.format("%.2f", conversion
    println("$initialMeasurement degrees $initialUnit is $
}
```