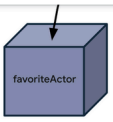
Kotlin fundamentals activity 3:

Use nullability in Kotlin

Português ... Fazer login

1. Before you begin
2. Use nullable variables
3. Handle nullable variables
4. Conclusion



To use `null` in code, follow these steps:

1. In Kotlin Playground, replace the content in the body of the `main()` function with a `favoriteActor` variable set to `null`:

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

2. Print the value of the `favoriteActor` variable with the `println()` function and then run this program:

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

Copiar amostra de código

The output looks like this code snippet:

```
null
```

Kotlin Solutions Docs Community Teach Play

2.1.10 JVM Program arguments

```
/**
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {
    val favoriteActor = null
    println(favoriteActor)
}
```

```
null
```

1. Before you begin
2. Use nullable variables
3. Handle nullable variables
4. Conclusion

There are occasions after you declare a variable when you may want to assign the variable to `null`. For example, after you declare your favorite actor, you decide that you don't want to reveal your favorite actor at all. In this case, it's useful to assign the `favoriteActor` variable to `null`.

Understand non-nullable and nullable variables

To reassign the `favoriteActor` variable to `null`, follow these steps:

1. Change the `val` keyword to a `var` keyword, and then specify that the `favoriteActor` variable is a `String` type and assign it to the name of your favorite actor:

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

2. Remove the `println()` function:

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

3. Reassign the `favoriteActor` variable to `null` and then run this program:

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

Copiar amostra de código

You get this error message:

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

2.1.10 JVM Program arguments

```
/**
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {
    var favoriteActor: String = "Sandra Oh"
    favoriteActor = null
}
```

```
Null cannot be a value of a non-null type 'kotlin.String'.
```

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. To declare a nullable variable, you need to explicitly add the nullable type. Without the nullable type, the Kotlin compiler infers that it's a non-nullable type.

4. Change the `favoriteActor` variable type from a `String` data type to a `String?` data type:

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

5. Print the `favoriteActor` variable before and after the `null` reassignment, and then run this program:

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

    favoriteActor = null
    println(favoriteActor)
}
```

The output looks like this code snippet:

```
Sandra Oh
null
```

The `favoriteActor` variable originally held a string and is then reassigned to `null`.

Try it

2.1.10 JVM Program arguments

```
/**
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {
    var favoriteActor: String? = "Sandra Oh"
    println(favoriteActor)

    favoriteActor = null
println(favoriteActor)

}
```

```
Sandra Oh
null
```


Try it

Now that you can use the nullable `String?` type, can you initialize a variable with an `Int` value and reassign it to `null`?

Write a nullable `Int` value

1. Remove all the code in the `main()` function:

```
fun main() {

}
```

2. Create a `number` variable of a nullable `Int` type and then assign it a `10` value:

```
fun main() {
    var number: Int? = 10
}
```

3. Print the `number` variable and then run this program:

```
fun main() {
    var number: Int? = 10
    println(number)
}
```

The output is as expected:

```
10
```

4. Reassign the `number` variable to `null` to confirm that the variable is nullable:

2.1.10 JVM Program arguments

```
/**
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {
    var number: Int? = 10
    println(number)
}
```

```
10
```


The output is as expected:

```
10
```

4. Reassign the `number` variable to `null` to confirm that the variable is nullable:

```
fun main() {
    var number: Int? = 10
    println(number)

    number = null
}
```

5. Add another `println(number)` statement as the final line of the program and then run it:

```
fun main() {
    var number: Int? = 10
    println(number)

    number = null
    println(number)
}
```

The output is as expected:

```
10
null
```

Note: While you should use nullable variables for variables that can carry `null`, you should use non-nullable variables for variables that can never carry `null` because the access of nullable variables requires more complex handling. You learn about various techniques to handle nullable variables in the next section.

2.1.10 JVM Program arguments

```
/**
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {
    var number: Int? = 10
    println(number)
    number = null
    println(number)


}
```

```
10
null
```

3. Handle nullable variables

Previously, you learned to use the `.` operator to access methods and properties of non-nullable variables. In this section, you learn how to use it to access methods and properties of nullable variables.

To access a property of the non-nullable `favoriteActor` variable, follow these steps:

1. Remove all the code in the `main()` function, and then declare a `favoriteActor` variable of `String` type and assign it to the name of your favorite actor:

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

2. Print the number of characters in the `favoriteActor` variable value with the `length` property and then run this program:

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

The output is as expected:

```
9
```

There are *nine* characters in the value of the `favoriteActor` variable, which includes spaces. The number of characters in your favorite actor's name might be different.

Access a property of a nullable variable

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:

- Change the `favoriteActor` variable type to a nullable type and then run this program:

```
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 error is a *compile error*. As mentioned in a previous codelab, a compile error happens when Kotlin isn't able to compile the code due to a syntax error in your code.

Kotlin intentionally applies syntactic rules so that it can achieve `null` *safety*, which refers to a guarantee that *no accidental calls are made on potentially* `null` *variables*. This doesn't mean that variables can't be `null`. It means that if a member of a variable is accessed, the variable can't be `null`.

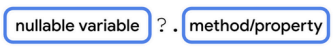
This is critical because if there's an attempt to access a member of a variable that's `null` - known as `null` *reference* - during the running of an app, the app crashes because the `null` variable doesn't contain any property or method. This type of crash is known as a *runtime error* in which the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

Next, you learn various techniques and operators to work with nullable types.

Use the `?.` safe call operator

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



To use the `?.` safe call operator to access a method or property, add a `?` symbol after the variable name and access the method or property with the `.` notation.

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.

To safely access a property of the nullable `favoriteActor` variable, follow these steps:

1. In the `println()` statement, replace the `.` operator with the `?.` safe call operator:

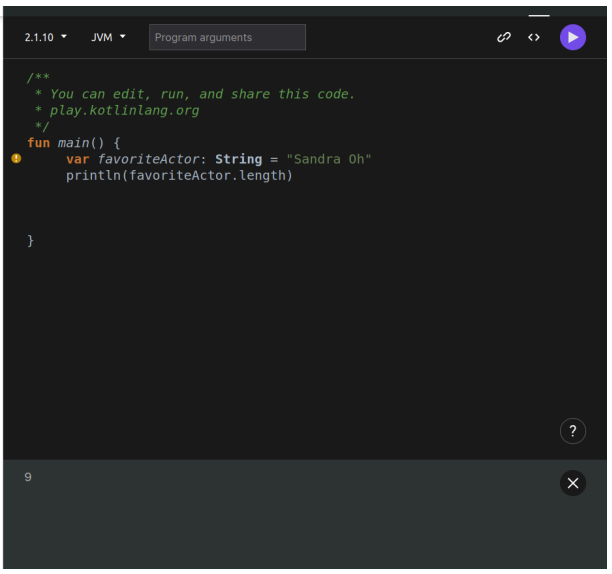
```
fun main() {
    var favoriteActor: String? = "Sandra Oh"
    println(favoriteActor?.length)
}
```

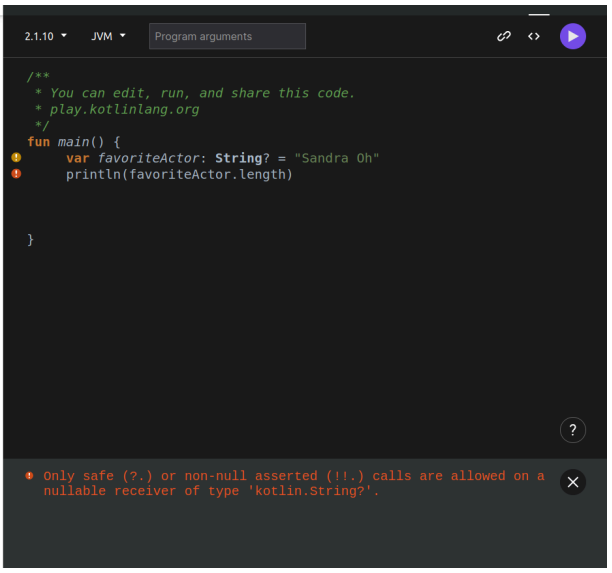
2. Run this program and then verify that the output is as expected:

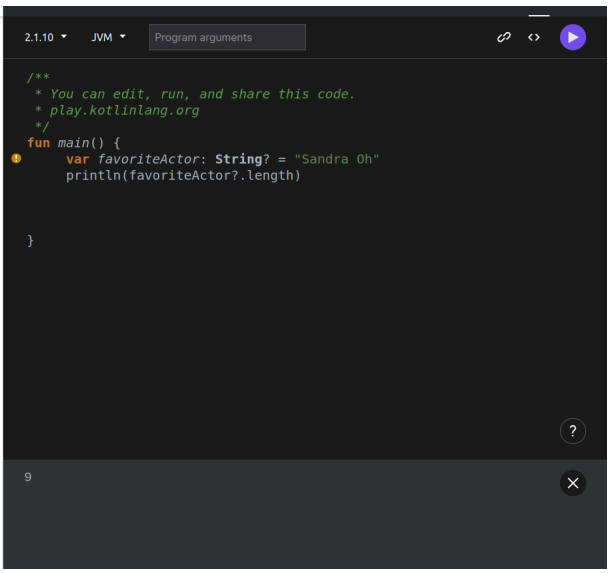
```
9
```

The number of characters of your favorite actor's name might differ.

3. Reassign the `favoriteActor` variable to `null` and then run this program:







```
    println(favoriteActor?.length)
}
```

2. Run this program and then verify that the output is as expected:

```
9
```

The number of characters of your favorite actor's name might differ.

3. Reassign the `favoriteActor` variable to `null` and then run this program:

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

You see this output:

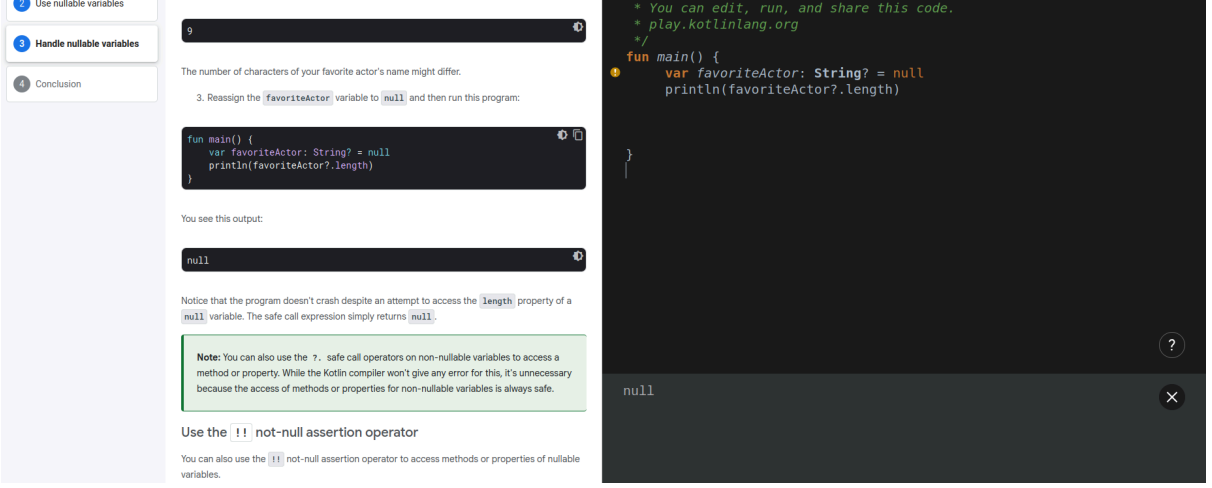
```
null
```

Notice that the program doesn't crash despite an attempt to access the `length` property of a `null` variable. The safe call expression simply returns `null`.

> **Note:** 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.



1 Before you begin
2 Use nullable variables
3 Handle nullable variables
4 Conclusion

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

nullable variable ! ! . method/property

After the nullable variable, you need to add the `!!` not-null assertion operator followed by the `.` operator and then the method or property without any spaces.

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. When not handled, exceptions cause runtime errors. You learn about exception handling in later units of this course.

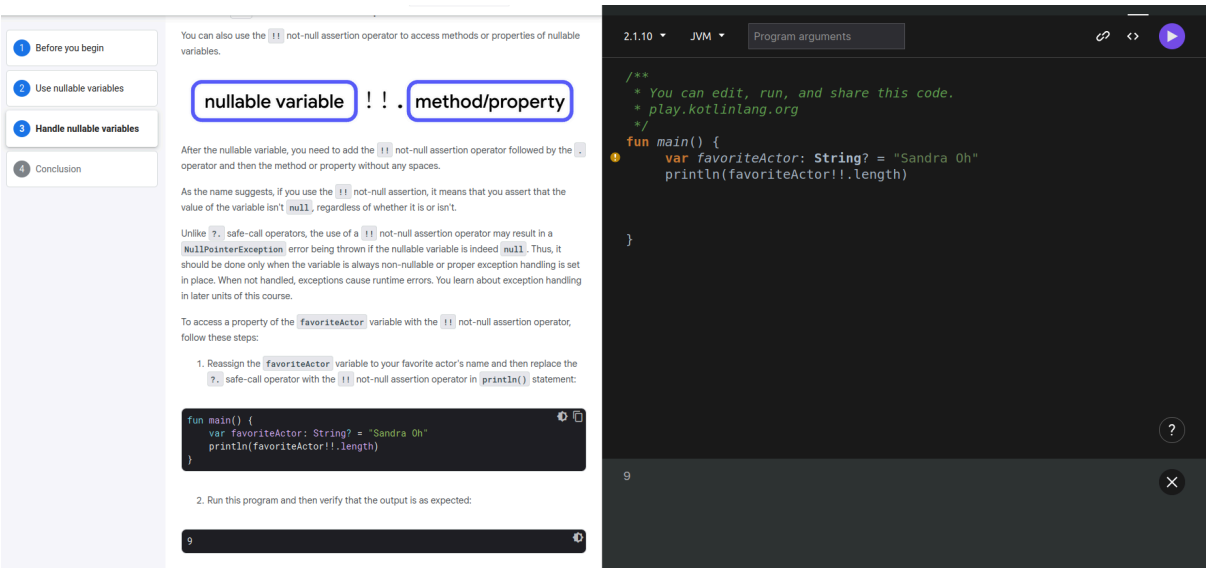
To access a property of the `favoriteActor` variable with the `!!` not-null assertion operator, follow these steps:

1. Reassign the `favoriteActor` variable to your favorite actor's name and then replace the `?.` safe-call operator with the `!!` not-null assertion operator in `println()` statement:

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

2. Run this program and then verify that the output is as expected:

```
9
```



1 Before you begin
2 Use nullable variables
3 Handle nullable variables
4 Conclusion

in later units of this course.

To access a property of the `favoriteActor` variable with the `!!` not-null assertion operator, follow these steps:

1. Reassign the `favoriteActor` variable to your favorite actor's name and then replace the `?.` safe-call operator with the `!!` not-null assertion operator in `println()` statement:

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

2. Run this program and then verify that the output is as expected:

```
9
```

The number of characters of your favorite actor's name might differ.

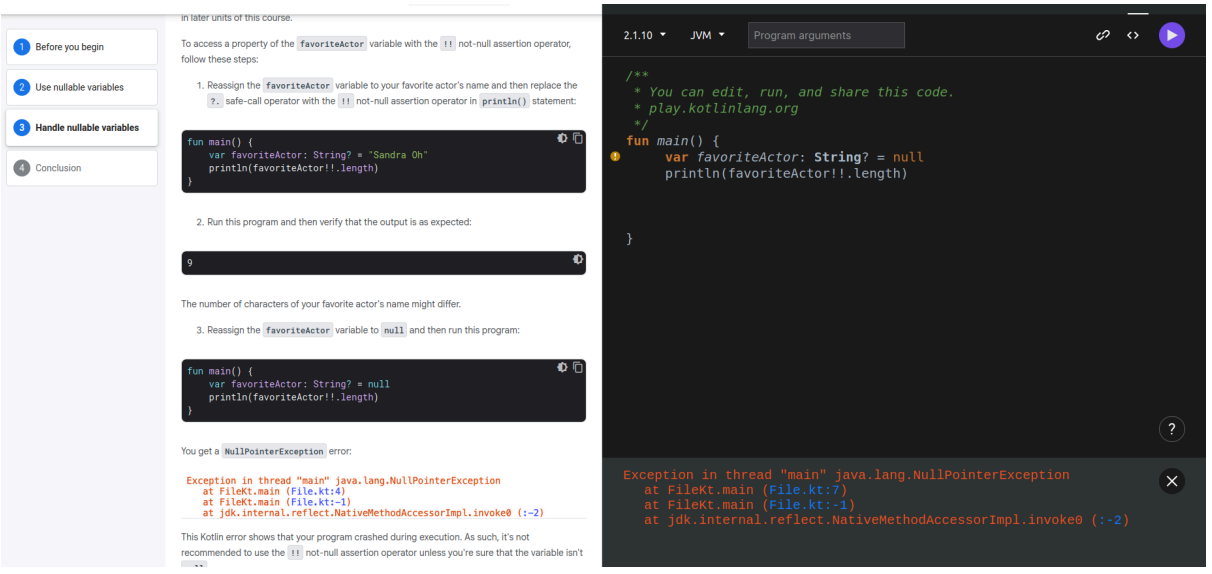
3. Reassign the `favoriteActor` variable to `null` and then run this program:

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

You get a `NullPointerException` error:

```
Exception in thread "main" java.lang.NullPointerException
    at FileKt.main (File.kt:4)
    at FileKt.main (File.kt:-1)
    at jdk.internal.reflect.NativeMethodAccessorImpl.invoke0 (:-2)
```

This Kotlin error shows that your program crashed during execution. As such, it's not recommended to use the `!!` not-null assertion operator unless you're sure that the variable isn't



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

2. Add an `if` branch with a `favoriteActor != null` condition:

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

    if (favoriteActor != null) {

    }
}
```

3. In the body of the `if` branch, add a `println` statement that accepts a `"The number of characters in your favorite actor's name is ${favoriteActor.length}."` string and then run this program:

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

    if (favoriteActor != null) {
      println("The number of characters in your favorite actor's name is
    }
}
```

The output is as expected.

```
The number of characters in your favorite actor's name is 9.
```

The number of characters in your favorite actor's name might differ.

2.1.10 JVM Program arguments

```
/**
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {

    var favoriteActor: String? = "Sandra Oh"

    if (favoriteActor != null) {
      println("The number of characters in your favorite actor's name is $
    }


}
```

```
The number of characters in your favorite actor's name is 9.
```



```
    } else {

    }
}
```

5. In the body of the `else` branch, add a `println` statement that takes a `"You didn't input a name."` string:

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

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

6. Assign the `favoriteActor` variable to `null` and then run this program:

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

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

The output is as expected:

```
You didn't input a name.
```

2.1.10 JVM Program arguments

```
/**
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {

    var favoriteActor: String? = null


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


}
```

```
You didn't input a name.
```

5. In the body of the `else` branch, add a `0` value:

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

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

The `0` value serves as the default value when the name is `null`.

6. At the end of the `main()` function, add a `println` statement that takes a `"The number of characters in your favorite actor's name is $lengthOfName."` string and then run this program:

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

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

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

The output is as expected:

```
The number of characters in your favorite actor's name is 9.
```

The number of characters of the name that you used might differ.

2.1.10 JVM Program arguments

```
/**
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {
    var favoriteActor: String? = "Sandra Oh"

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

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

```
The number of characters in your favorite actor's name is 9.
```


To modify your previous program to use the `?:` Elvis operator, follow these steps:

1. Remove the `if/else` conditional and then set the `lengthOfName` variable to the nullable `favoriteActor` variable and use the `?.` safe-call operator to call its `length` property:

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

    val lengthOfName = favoriteActor?.length

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

2. After the `length` property, add the `?:` Elvis operator followed by a `0` value and then run this program:

Copiar amostra de código

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

    val lengthOfName = favoriteActor?.length ?: 0

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

The output is the same as the previous output:

```
The number of characters in your favorite actor's name is 9.
```

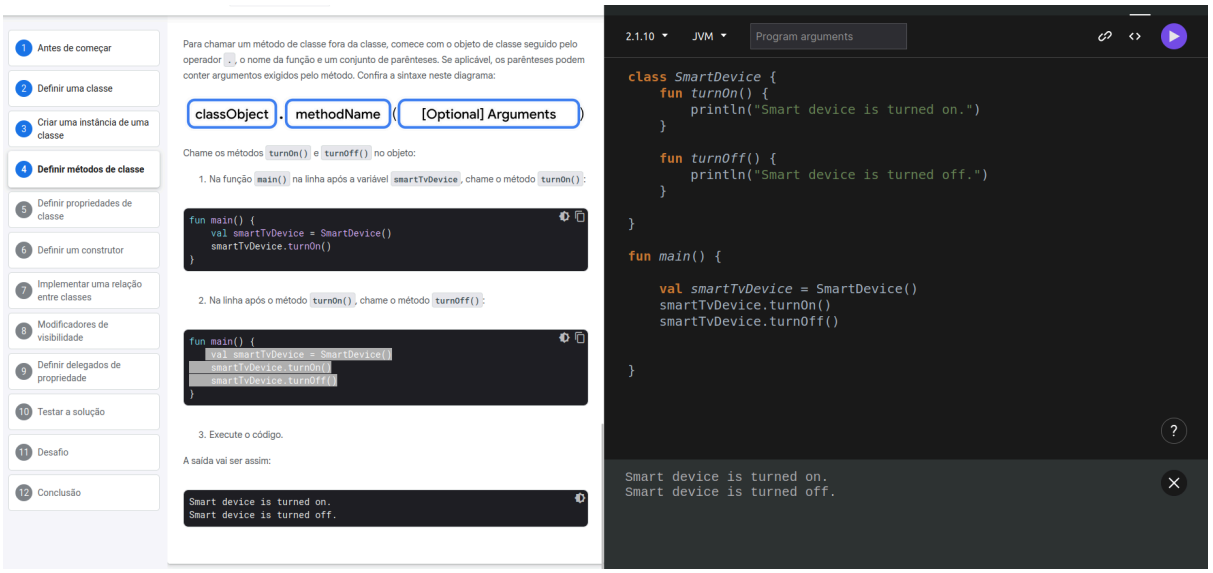
> **Note:** The `?:` Elvis operator is named after Elvis Presley, the rock star, because it resembles an emoticon of his quiff when you view it sideways.

2.1.10 JVM Program arguments

```
/**
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {
    var favoriteActor: String? = "Sandra Oh"

    val lengthOfName = favoriteActor?.length ?: 0

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

```
The number of characters in your favorite actor's name is 9.
```

Kotlin fundamentals activity 4:



1. Antes de começar
2. Definir uma classe
3. Criar uma instância de uma classe
4. Definir métodos de classe
5. Definir propriedades de classe
6. Definir um construtor
7. Implementar uma relação entre classes
8. Modificadores de visibilidade
9. Definir delegados de propriedade
10. Testar a solução
11. Desafio
12. Conclusão

Para chamar um método de classe fora da classe, comece com o objeto de classe seguido pelo operador `.`, o nome da função e um conjunto de parênteses. Se aplicável, os parênteses podem conter argumentos exigidos pelo método. Confira a sintaxe neste diagrama:

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

Chame os métodos `turnOn()` e `turnOff()` no objeto:

1. Na função `main()` na linha após a variável `smartTvDevice`, chame o método `turnOn()`:

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

2. Na linha após o método `turnOn()`, chame o método `turnOff()`:

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

3. Execute o código.

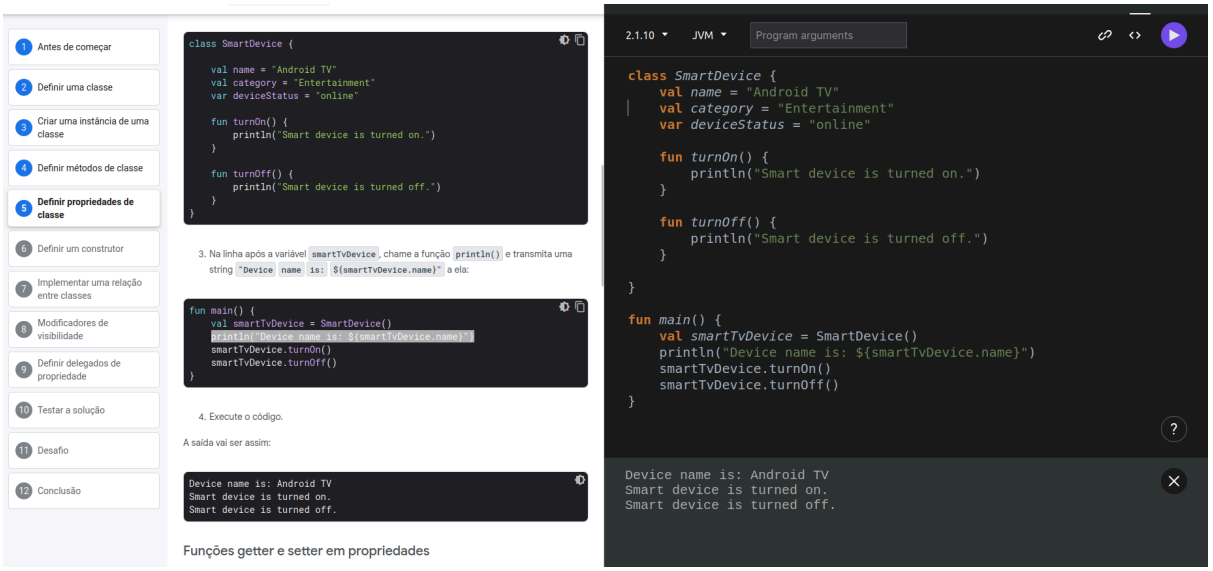
A saída vai ser assim:

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

2.1.10 JVM Program arguments

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

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

}

fun main() {

    val smartTvDevice = SmartDevice()
    smartTvDevice.turnOn()
    smartTvDevice.turnOff()


}
```

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



1. Antes de começar
2. Definir uma classe
3. Criar uma instância de uma classe
4. Definir métodos de classe
5. Definir propriedades de classe
6. Definir um construtor
7. Implementar uma relação entre classes
8. Modificadores de visibilidade
9. Definir delegados de propriedade
10. Testar a solução
11. Desafio
12. Conclusão

```
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.")
    }
}
```

3. Na linha após a variável `smartTvDevice`, chame a função `println()` e transmita uma string `"Device name is: ${smartTvDevice.name}"` a ela:

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

4. Execute o código.

A saída vai ser assim:

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

Funções getter e setter em propriedades

2.1.10 JVM Program arguments

```
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.
```

```
            "set to $channelNumber."
        )
    }

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

12. Na função `main()`, use a palavra-chave `var` para definir uma variável `smartDevice` do tipo `SmartDevice` que instancia um objeto `SmartTvDevice` que usa um argumento `"Android TV"` e um `"Entertainment"`:

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

13. Na linha após a variável `smartDevice`, chame o método `turnOn()` no objeto `smartDevice`:

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

Copiar amostra de código

14. Execute o código.

A saída vai ser assim:

```
d on. Speaker volume is set to 2 and channel number is set to 1.
```

15. Na linha após a chamada do método `turnOn()`, reatribua a variável `smartDevice` para instanciar uma classe `SmartLightDevice` que usa um argumento `"Google Light"`

```
2.1.10  JVM  Program arguments

    smartTvDevice.increaseSpeakerVolume()
}

fun changeTvChannelToNext() {
    smartTvDevice.nextChannel()
}
 fun turnOnLight() {
    smartLightDevice.turnOn()
}

fun turnOffLight() {
    smartLightDevice.turnOff()
}

fun increaseLightBrightness() {
    smartLightDevice.increaseBrightness()
}

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

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

14. Execute o código.

A saída vai ser assim:

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

15. Na linha após a chamada para o método `turnOn()`, reatribua a variável `smartDevice` para instanciar uma classe `SmartLightDevice` que usa um argumento `"Google Light"` e um argumento `"Utility"`. Em seguida, chame o método `turnOn()` na referência do objeto `smartDevice`:

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

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

16. Execute o código.

A saída vai ser assim:

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

Esse é um exemplo de polimorfismo. O código chama o método `turnOn()` em uma variável do tipo `SmartDevice` e, dependendo do valor real da variável, diferentes implementações do método `turnOn()` podem ser executadas.

Reutilizar o código de superclasse em subclasses com a palavra-chave `super`

A Voltar is de perto os métodos `turnOn()` e `turnOff()`, há uma semelhança na fo Avançar

```
2.1.10  JVM  Program arguments  Copy link  Share code  Run

    }

    fun increaseLightBrightness() {
        smartLightDevice.increaseBrightness()
    }

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


}



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

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

```
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.
```

```
    }
}

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()
}
```

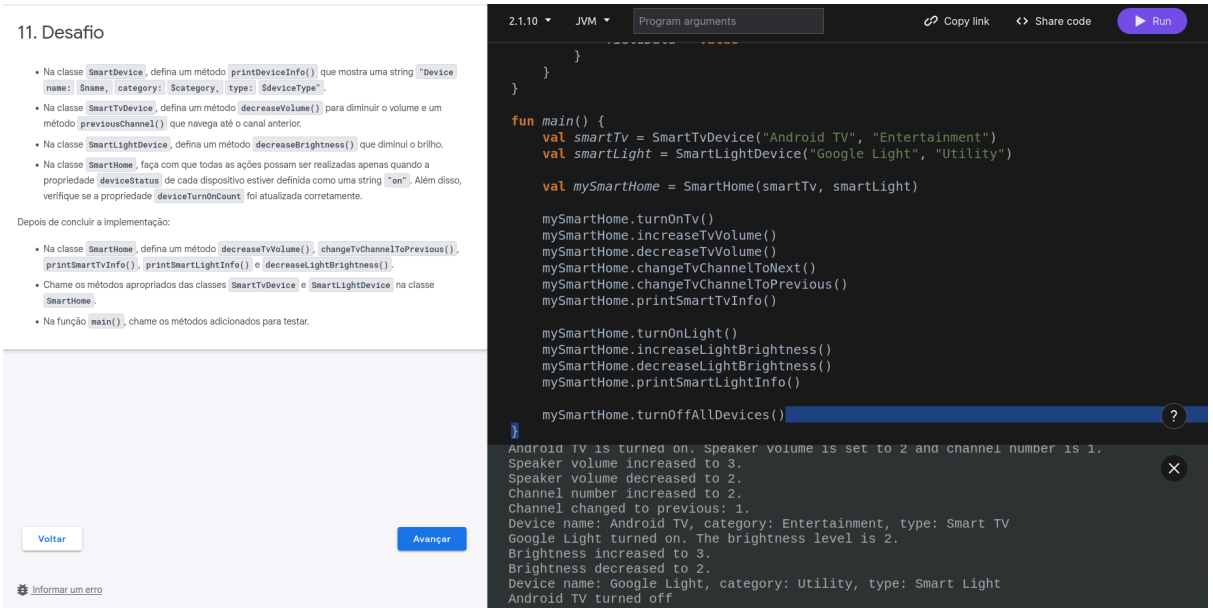
A saída vai ser assim:

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

Voltar Avançar

```
2.1.10  JVM  Program arguments  Copy link  Share code  Run

}

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

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


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

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

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

```
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.
```



Código final:

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

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

        open val deviceType = "unknown"

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

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

        fun printDeviceInfo() {
        println("Device name: $name, category: $category, type: $deviceType")
        }
}

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 decreaseVolume() {
        speakerVolume--
        println("Speaker volume decreased to $speakerVolume.")
        }

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

        fun previousChannel() {
        channelNumber--
        println("Channel changed to previous: $channelNumber.")
        }

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

```
        }

        fun decreaseBrightness() {
        brightnessLevel--
        println("Brightness decreased 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(
        private val smartTvDevice: SmartTvDevice,
        private val smartLightDevice: SmartLightDevice
) {
        var deviceTurnOnCount = 0
        private set

        fun turnOnTv() {
        if (smartTvDevice.deviceStatus != "on") {
        deviceTurnOnCount++
        smartTvDevice.turnOn()
        }
        }

        fun turnOffTv() {
        if (smartTvDevice.deviceStatus == "on") {
        deviceTurnOnCount--
        smartTvDevice.turnOff()
        }
        }

        fun increaseTvVolume() {
        if (smartTvDevice.deviceStatus == "on") smartTvDevice.increaseSpeakerVolume()
        }

        fun decreaseTvVolume() {
        if (smartTvDevice.deviceStatus == "on") smartTvDevice.decreaseVolume()
        }
```

```
        fun changeTvChannelToNext() {
        if (smartTvDevice.deviceStatus == "on") smartTvDevice.nextChannel()
        }

        fun changeTvChannelToPrevious() {
        if (smartTvDevice.deviceStatus == "on") smartTvDevice.previousChannel()
        }

        fun turnOnLight() {
        if (smartLightDevice.deviceStatus != "on") {
        deviceTurnOnCount++
        smartLightDevice.turnOn()
        }
        }

        fun turnOffLight() {
        if (smartLightDevice.deviceStatus == "on") {
        deviceTurnOnCount--
        smartLightDevice.turnOff()
        }
        }

        fun increaseLightBrightness() {
        if (smartLightDevice.deviceStatus == "on") smartLightDevice.increaseBrightness()
        }

        fun decreaseLightBrightness() {
        if (smartLightDevice.deviceStatus == "on") smartLightDevice.decreaseBrightness()
        }

        fun printSmartTvInfo() {
        smartTvDevice.printDeviceInfo()
        }

        fun printSmartLightInfo() {
        smartLightDevice.printDeviceInfo()
        }

        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() {
        val smartTv = SmartTvDevice("Android TV", "Entertainment")
        val smartLight = SmartLightDevice("Google Light", "Utility")

        val mySmartHome = SmartHome(smartTv, smartLight)

        mySmartHome.turnOnTv()
        mySmartHome.increaseTvVolume()
        mySmartHome.decreaseTvVolume()
        mySmartHome.changeTvChannelToNext()
        mySmartHome.changeTvChannelToPrevious()
        mySmartHome.printSmartTvInfo()

        mySmartHome.turnOnLight()
        mySmartHome.increaseLightBrightness()
        mySmartHome.decreaseLightBrightness()
        mySmartHome.printSmartLightInfo()

        mySmartHome.turnOffAllDevices()
}
```

Kotlin fundamentals activity 5:

1. Acesse o Kotlin Playground.
2. Após a função `main()`, defina uma `trick()` sem parâmetros e sem valor de retorno para mostrar a mensagem `"No treats!"` (Sem doces!). A sintaxe é a mesma de outras funções mostradas em codelabs anteriores.

```
fun main() {

}

fun trick() {
    println("No treats!")
}
```

3. No corpo da função `main()`, crie uma variável chamada `trickFunction` e a defina como igual a `trick` (travessuras). Os parênteses depois de `trick` não são incluídos porque você não quer chamar a função, mas sim que ela seja armazenada em uma variável.

Copiar amostra de código

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

fun trick() {
    println("No treats!")
}
```

4. Execute o código. Ele gera um erro porque o compilador Kotlin reconhece `trick` como o nome da função `trick()`, mas espera que você chame a função em vez de a atribuir a uma variável.

```
Function invocation 'trick()' expected
```

Você tentou armazenar `trick` na variável `trickFunction`. No entanto, para se referir a uma função como um valor, é necessário usar o operador de referência de função (`::`). A sintaxe é ilustrada nesta imagem:

Voltar Avançar

Informar um erro

2.1.10 JVM Program arguments Copy link Share code Run

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

fun trick() {
    println("No treats!")
}
```

Function invocation 'trick()' expected.

2. Na função `main()`, remova o operador de referência de função (`::`), já que `trick` agora se refere a uma variável em vez de um nome de função.

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

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

3. Execute o código. Não há erros e você pode referenciar a função `trick()` sem o operador de referência de função (`::`). Não há saída porque você ainda não chamou a função.
4. Na função `main()`, chame a `trick()`, mas agora inclua os parênteses, como faria ao chamar qualquer outra função.

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

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

Copiar amostra de código

5. Execute o código. O corpo da expressão lambda é executado.

```
No treats!
```

6. No função `main()`, chame a variável `trickFunction` como se ela fosse uma função.

Voltar Avançar

```
fun main() {
```

Informar um erro

2.1.10 JVM Program arguments Copy link Share code Run

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

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

No treats!

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

5. Execute o código. O corpo da expressão lambda é executado.

```
No treats!
```

6. Na função `main()`, chame a variável `trickFunction` como se ela fosse uma função.

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

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

Copiar amostra de código

7. Execute o código. A função é chamada duas vezes: uma para a chamada de função `trick()` e outra para a `trickFunction()`.

```
No treats!
No treats!
```

As expressões lambda permitem criar variáveis que armazenam funções, fazer a chamada delas como funções e as armazenar em outras variáveis que podem ser chamadas como funções.

Voltar Avançar

Informar um erro

2.1.10 JVM Program arguments Copy link Share code Run

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

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

```
No treats!
No treats!
```

```
val treat = {
    println("Have a treat!")
}
```

2. Especifique o tipo de dado da variável `treat` como `() -> Unit`.

```
val treat: () -> Unit = {
    println("Have a treat!")
}
```

3. Na função `main()`, chame a `treat()`.

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

4. Execute o código. A função `treat()` se comporta como a `trick()`. As duas variáveis têm o mesmo tipo de dado, mesmo que apenas a variável `treat` o declare explicitamente.

```
No treats!
No treats!
Have a treat!
```

Usar uma função como um tipo de retorno

Uma função é um tipo de dado e, por isso, pode ser usada como qualquer outro. Você pode até mesmo retornar funções de outras funções. A sintaxe está ilustrada nesta imagem:

Voltar function name () : function type Avançar

Informar um erro

2.1.10 JVM Program arguments Copy link Share code Run

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

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

val treat: () -> Unit = {
    println("Have a treat!")
}
```

```
No treats!
No treats!
Have a treat!
```

```
}
```

4. Na função `main()`, crie uma variável chamada `treatFunction` e a atribua ao resultado da chamada de `trickOrTreat()`, transmitindo `false` para o parâmetro `isTrick`. Em seguida, crie uma segunda variável, chamada `trickFunction`, e a atribua ao resultado da chamada de `trickOrTreat()`, desta vez transmitindo `true` para o parâmetro `isTrick`.

```
fun main() {
    val treatFunction = trickOrTreat(false)
    val trickFunction = trickOrTreat(true)
}
```

5. Chame `treatFunction()` e depois `trickFunction()` na próxima linha.

```
fun main() {
    val treatFunction = trickOrTreat(false)
    val trickFunction = trickOrTreat(true)
    treatFunction()
    trickFunction()
}
```

6. Execute o código. Você verá a resposta de cada função. Mesmo que não tenha chamado as funções `trick()` ou `treat()` diretamente, elas ainda podem ser chamadas, já que você armazenou os valores de retorno cada vez que chamou a função `trickOrTreat()` e usou as variáveis `trickFunction` e `treatFunction` para chamar as funções.

```
Have a treat!
No treats!
```

Agora, você sabe como as funções podem retornar outras funções. Você também pode transmitir uma função c Voltar o para outra. Talvez você queira fornecer algum comportamento personaliza Avançar função `trickOrTreat()` faça algo além de retornar uma das duas strings. Uma função que usa outra como argumento permite transmitir uma função diferente sempre que ela é chamada.

Informar um erro

2.1.10 JVM Program arguments Copy link Share code Run

```
fun main() {
    val treatFunction = trickOrTreat(false)
    val trickFunction = trickOrTreat(true)
    treatFunction()
    trickFunction()

}


fun trickOrTreat(isTrick: Boolean): () -> Unit {
    if (isTrick) {
        return trick
    } else {
        return treat
    }
}


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

val treat: () -> Unit = {
    println("Have a treat!")
```

```
Have a treat!
No treats!
```

```
    "Have a cupcake!"
}
```

Observação: na função `coins()`, o parâmetro `Int` é denominado `quantity`. No entanto, ele pode receber qualquer nome, desde que o parâmetro e a variável na string tenham o mesmo nome.

6. Atualize as chamadas para a função `trickOrTreat()`. Para a primeira chamada, quando `isTrick` for `false`, transmita a função `coins()`. Para a segunda chamada, quando `isTrick` for `true`, transmita a função `cupcake()`.

```
fun main() {
    val coins: (Int) -> String = { quantity ->
        "$quantity quarters"
    }

    val cupcake: (Int) -> String = {
        "Have a cupcake!"
    }

    val treatFunction = trickOrTreat(false, coins)
    val trickFunction = trickOrTreat(true, cupcake)
    treatFunction()
    trickFunction()
}
```

7. Execute o código. A função `extraTreat()` só é chamada quando o parâmetro `isTrick` é definido como um argumento `false`. Então, a saída inclui cinco moedas de 25 centavos, mas nenhum cupcake.

```
5 quarters
Have a treat!
No treats!
```

T Voltar ınção anuláveis Avançar

Assim como outros tipos de dados, os tipos de função podem ser declarados como anuláveis. Próxima etapa s,

Informar um erro

2.1.10 JVM Program arguments Copy link Share code Run

```
fun main() {
    val coins: (Int) -> String = { quantity ->
        "$quantity quarters"
    }

    val cupcake: (Int) -> String = {
        "Have a cupcake!"
    }

    val treatFunction = trickOrTreat(false, coins)
    val trickFunction = trickOrTreat(true, cupcake)
    treatFunction()
    trickFunction()
}


fun trickOrTreat(isTrick: Boolean, extraTreat: (Int) -> String): () -> Unit {
    if (isTrick) {
        return trick
    } else {
        println(extraTreat(5))
        return treat
    }
```

```
5 quarters
Have a treat!
No treats!
```

```
fun trickOrTreat(isTrick: Boolean, extraTreat: ((Int) -> String)?): () -> Unit {
    if (isTrick) {
        return trick
    } else {
        if (extraTreat != null) {
            println(extraTreat(5))
        }
        return treat
    }
}
```

3. Remova a função `cupcake()` e substitua o argumento `cupcake` por `null` na segunda chamada para a função `trickOrTreat()`.

```
fun main() {
    val coins: (Int) -> String = { quantity ->
        "$quantity quarters"
    }

    val treatFunction = trickOrTreat(false, coins)
    val trickFunction = trickOrTreat(true, null)
    treatFunction()
    trickFunction()
}
```

4. Execute o código. A saída não será modificada. Agora que você pode declarar tipos de [illegible] anuláveis, não é mais necessário transmitir uma função para o parâmetro `extraTreat`.

```
5 quarters
Have a treat!
No treats!
```

Voltar | Avançar

Informar um erro

2.1.10 | JVM | Program arguments | Copy link | Share code | Run

```
fun main() {
    val coins: (Int) -> String = { quantity ->
        "$quantity quarters"
    }

    val treatFunction = trickOrTreat(false, coins)
    val trickFunction = trickOrTreat(true, null)
    treatFunction()
    trickFunction()
}


fun trickOrTreat(isTrick: Boolean, extraTreat: ((Int) -> String)?): () -> Unit {
    if (isTrick) {
        return trick
    } else {
        if (extraTreat != null) {
            println(extraTreat(5))
        }
        return treat
    }
}
```

```
5 quarters
Have a treat!
No treats!
```

```
    val coins: (Int) -> String = {
        "$it quarters"
    }
```

Atualize a função `coins()` para usar a sintaxe abreviada para parâmetros:

1. Na função `coins()`, remova o nome do parâmetro `quantity` e o símbolo `->`.

```
val coins: (Int) -> String = {
    "$quantity quarters"
}
```

2. Mude o modelo de string `"$quantity quarters"` para se referir ao parâmetro único usando `$it`.

```
val coins: (Int) -> String = {
    "$it quarters"
}
```

3. Execute o código. O Kotlin reconhece o nome do parâmetro `it` do `Int` e ainda mostra o número de moedas de 25 centavos.

```
5 quarters
Have a treat!
No treats!
```

Transmitir uma expressão lambda diretamente para uma função

No momento, a função `coins()` é usada apenas em um lugar. E se você pudesse simplesmente transmitir uma expressão lambda para a função `trickOrTreat()` sem a necessidade de criar uma variável?

A[illegible]ambda são simplesmente literais de função, assim como `0` é um literal de n[illegible] e um literal de string. Você pode transmitir uma expressão lambda diretamente para uma chamada de função. A sintaxe está ilustrada nesta imagem:

Voltar | Avançar

Informar um erro

2.1.10 | JVM | Program arguments | Copy link | Share code | Run

```
fun main() {
    val coins: (Int) -> String = {
    "$it quarters"
}


    val treatFunction = trickOrTreat(false, coins)
    val trickFunction = trickOrTreat(true, null)
    treatFunction()
    trickFunction()
}


fun trickOrTreat(isTrick: Boolean, extraTreat: ((Int) -> String)?): () -> Unit {
    if (isTrick) {
        return trick
    } else {
        if (extraTreat != null) {
            println(extraTreat(5))
        }
        return treat
    }
}
```

```
5 quarters
Have a treat!
No treats!
```

1. Mova a expressão lambda para que ela seja transmitida diretamente para a função `trickOrTreat()` na chamada. Você também pode condensar a expressão lambda em uma única linha.

```
fun main() {
    val coins: (Int) -> String = {
        "$it quarters"
    }
    val treatFunction = trickOrTreat(false, { "$it quarters" })
    val trickFunction = trickOrTreat(true, null)
    treatFunction()
    trickFunction()
}
```

2. Remova a variável `coins`, que não será mais usada.

```
fun main() {
    val treatFunction = trickOrTreat(false, { "$it quarters" })
    val trickFunction = trickOrTreat(true, null)
    treatFunction()
    trickFunction()
}
```

Copiar amostra de código

3. Execute o código. A compilação e execução ainda funcionam como esperado.

```
5 quarters
Have a treat!
No treats!
```

Usar a sintaxe de lambda final

É possível usar outra opção abreviada para programar lambdas quando um tipo de função é o último parâmetro de uma função. Nesse caso, você pode colocar a expressão lambda após os parênteses para c... o. A sintaxe está ilustrada nesta imagem:

Voltar Avançar

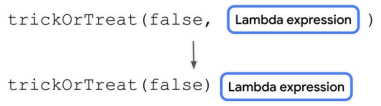
Informar um erro

2.1.10 JVM Program arguments Copy link Share code Run

```
fun main() {
    val treatFunction = trickOrTreat(false, { "$it quarters" })
    val trickFunction = trickOrTreat(true, null)
    treatFunction()
    trickFunction()
}

fun trickOrTreat(isTrick: Boolean, extraTreat: ((Int) -> String)?): () -> Unit {
    if (isTrick) {
        return trick
    } else {
        if (extraTreat != null) {
            println(extraTreat(5))
        }
        return treat
    }
}

val trick = {
```

```
5 quarters
Have a treat!
No treats!
```

chamar a função. A sintaxe está ilustrada nesta imagem:



Isso deixa seu código mais legível, porque ele separa a expressão lambda dos outros parâmetros, mas não muda o que o código faz.

Atualize o código para usar a sintaxe de lambda final:

1. Na variável `treatFunction`, mova a expressão lambda `{"$it quarters"}` após os parênteses na chamada para `trickOrTreat()`.

```
val treatFunction = trickOrTreat(false) { "$it quarters" }
```

2. Execute o código. Tudo continua funcionando.

```
5 quarters
Have a treat!
No treats!
```

Observação: as funções de composição usadas para declarar a IU utilizam funções como parâmetros e geralmente são chamadas com a sintaxe de lambda final.

Voltar Avançar

Informar um erro

2.1.10 JVM Program arguments Copy link Share code Run

```
fun main() {
    val treatFunction = trickOrTreat(false) { "$it quarters" }
    val trickFunction = trickOrTreat(true, null)
    treatFunction()
    trickFunction()
}

fun trickOrTreat(isTrick: Boolean, extraTreat: ((Int) -> String)?): () -> Unit {
    if (isTrick) {
        return trick
    } else {
        if (extraTreat != null) {
            println(extraTreat(5))
        }
        return treat
    }
}

val trick = {
```

```
5 quarters
Have a treat!
No treats!
```

```
fun main() {
    val treatFunction = trickOrTreat(false) { "$it quarters" }
    val trickFunction = trickOrTreat(true, null)
    treatFunction()
    trickFunction()
    repeat(4) {

    }
}
```

2. Mova a chamada para a função `treatFunction()` na expressão lambda da função `repeat()`.

```
fun main() {
    val treatFunction = trickOrTreat(false) { "$it quarters" }
    val trickFunction = trickOrTreat(true, null)
    repeat(4) {
        treatFunction()
    }
    trickFunction()
}
```

3. Execute o código. A string `"Have a treat"` precisa ser mostrada quatro vezes.

```
5 quarters
Have a treat!
Have a treat!
Have a treat!
Have a treat!
No treats!
```

Voltar

Avançar

Informar um erro

2.1.10 JVM Program arguments Copy link Share code Run

```
fun main() {
    val treatFunction = trickOrTreat(false) { "$it quarters" }
    val trickFunction = trickOrTreat(true, null)
    repeat(4) {
        treatFunction()
    }
    trickFunction()
}

fun trickOrTreat(isTrick: Boolean, extraTreat: ((Int) -> String)?): () -> Unit {
    if (isTrick) {
        return trick
    } else {
        if (extraTreat != null) {
            println(extraTreat(5))
        }
        return treat
    }
}
```

```
5 quarters
Have a treat!
Have a treat!
Have a treat!
Have a treat!
No treats!
```