



**TECNOLOGICO NACIONAL DE MEXICO
INSTITUTO TECNOLOGICO DE CIUDAD MADERO**

Carrera: Ingeniería en Sistemas Computacionales.

Materia: Programación nativa para móviles

Alumnos:

Zavala Osorio Camilo Alexander

Yañez Herrera Karina Aileen

Números de control: 21070415

21070336

Profesor: Jorge Peralta Escobar

Hora: 2:00 pm – 3:00 pm

Semestre: Enero - Junio 2025

Unidad 1: Tu primera app para Android Ruta de Aprendizaje 1 (Introducción a Kotlin)

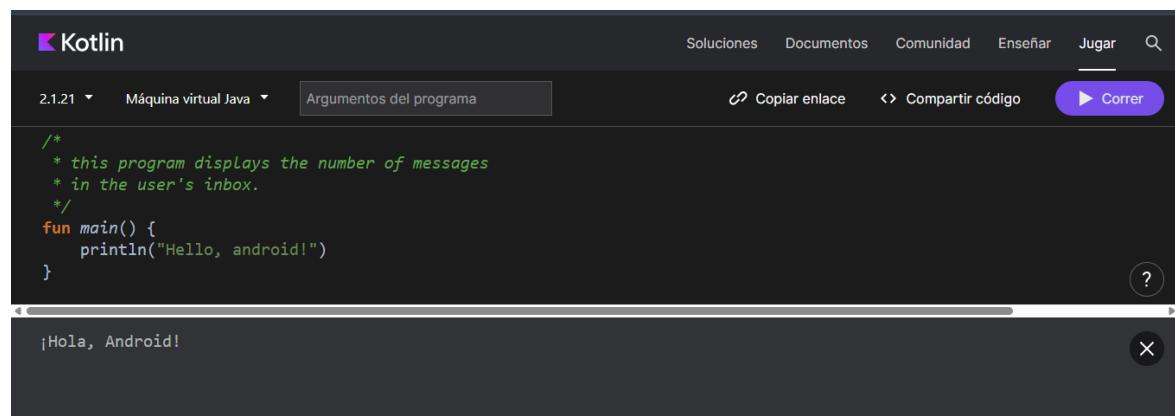
En este codelab se explica la función principal `fun main()` y cómo usar `println()` para imprimir texto. Se enseña cómo modificar el mensaje, por ejemplo, cambiando "Hello, world!" por "Hello, Android!", y cómo repetir líneas para imprimir el mensaje más de una vez.



The screenshot shows the Kotlin playground interface. At the top, it says "Kotlin" with a logo, version "2.1.21", and "Máquina virtual Java". Below that is a text input field labeled "Argumentos del programa". The code area contains the following Kotlin code:

```
/*
 * this program displays the number of messages
 * in the user's inbox.
 */
fun main() {
    println("Hello, world!")
}
```

At the bottom, the output window shows the text "¡Hola Mundo!".



This screenshot shows the same setup as the first one, but with more UI elements visible. At the top right are links for "Soluciones", "Documentos", "Comunidad", "ENSEÑAR", "Jugar", and a search icon. Below the code input field is a "Correr" (Run) button with a play icon. The output window at the bottom shows the text "¡Hola, Android!".

A screenshot of the Kotlin Play website interface. The top navigation bar includes links for 'Aplicaciones', 'Todos los favoritos', 'Soluciones', 'Documentos', 'Comunidad', 'Enseñar', 'Jugar', and a search icon. The main area shows code in a dark-themed editor:

```
2.1.21 ▾ Máquina virtual Java ▾ Argumentos del programa
```

```
/**  
 * You can edit, run, and share this code.  
 * play.kotlinlang.org  
 */  
fun main() {  
    println("Hello, Android!")  
    println("Hello, Android!")  
}  
  
¡Hola, Android! ¡Hola, Android!
```

The output window at the bottom displays the printed text: "¡Hola, Android! ¡Hola, Android!".

A screenshot of the Kotlin Play website interface. The top navigation bar includes links for 'Aplicaciones', 'Todos los favoritos', 'Soluciones', 'Documentos', 'Comunidad', 'Enseñar', 'Jugar', and a search icon. The main area shows code in a dark-themed editor:

```
2.1.21 ▾ Máquina virtual Java ▾ Argumentos del programa
```

```
/**  
 * You can edit, run, and share this code.  
 * play.kotlinlang.org  
 */  
fun main() {  
    println("Today is sunny!")  
}  
  
¡Hoy está soleado!
```

The output window at the bottom displays the printed text: "¡Hoy está soleado!".

A screenshot of the Kotlin Play website interface. The top navigation bar includes links for 'Aplicaciones', 'Todos los favoritos', 'Soluciones', 'Documentos', 'Comunidad', 'Enseñar', 'Jugar', and a search icon. The main area shows code in a dark-themed editor:

```
2.1.21 ▾ Máquina virtual Java ▾ Argumentos del programa
```

```
/**  
 * You can edit, run, and share this code.  
 * play.kotlinlang.org  
 */  
fun main() {  
    println("Coding is fun!")  
}  
  
¡Codificar es divertido!
```

The output window at the bottom displays the printed text: "¡Codificar es divertido!".

Kotlin

Solutions Docs Community Teach Play

2.1.21 JVM Program arguments

Copy link Share code Run

```
fun main() {
    println("I'm")
    println("learning")
    println("Kotlin!")
}
```

I'm
learning
Kotlin!

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
* You can edit, run, and share this code.
* play.kotlinlang.org
*/
fun main() {
    println("Tuesday")
    println("Thursday")
    println("Wednesday")
    println("Friday")
    println("Monday")
}
```

Tuesday
Thursday
Wednesday
Friday
Monday

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/*
 * You can edit, run, and share this code.
 * play.kotlinlang.org
*/
fun main() {
    println("Tomorrow is rainy")
}
```

Tomorrow is rainy

Kotlin

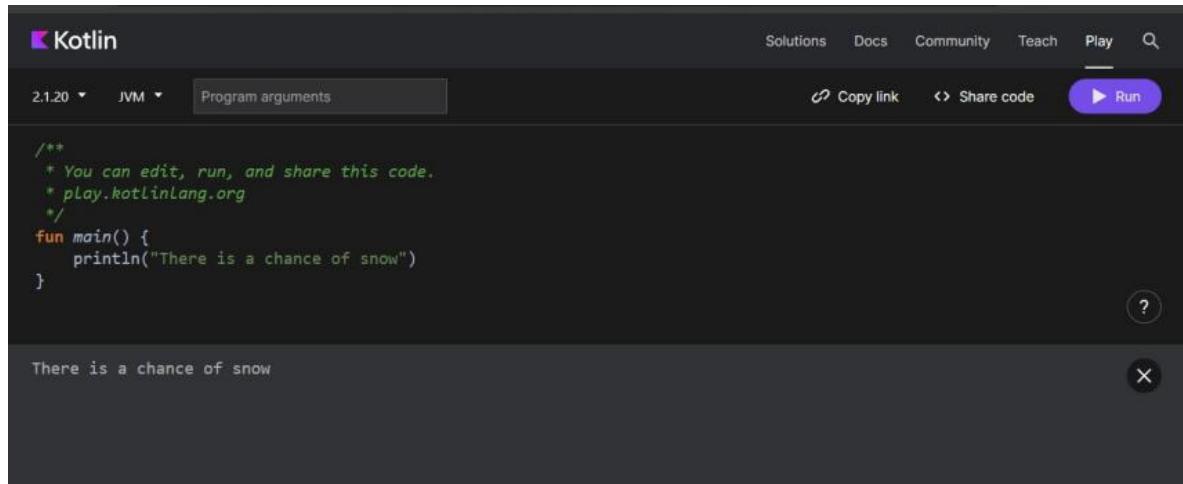
Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/*
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {
    println("There is a chance of snow")
}
```

There is a chance of snow



Kotlin

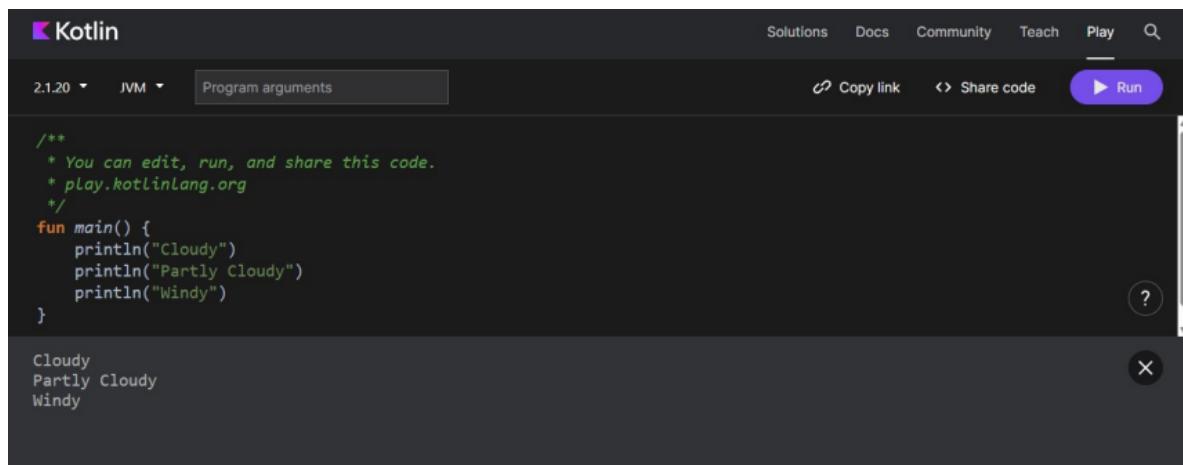
Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/*
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {
    println("Cloudy")
    println("Partly Cloudy")
    println("Windy")
}
```

Cloudy
Partly Cloudy
Windy



Kotlin

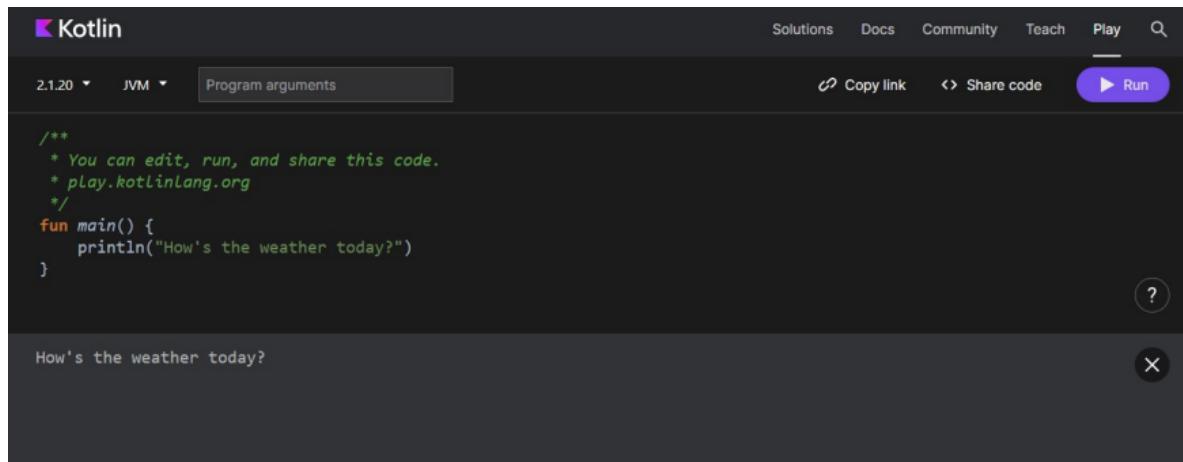
Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

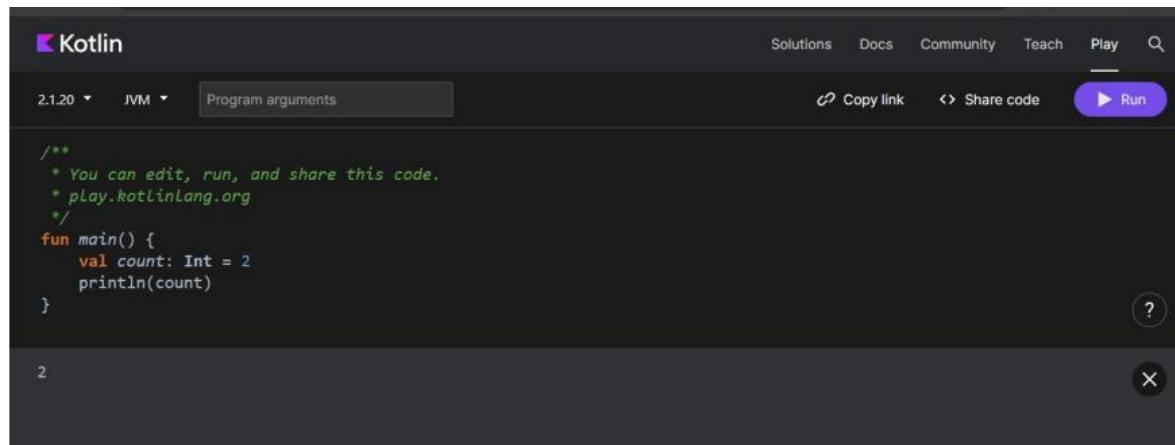
```
/*
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {
    println("How's the weather today?")
}
```

How's the weather today?



Crea y usa variables en Kotlin

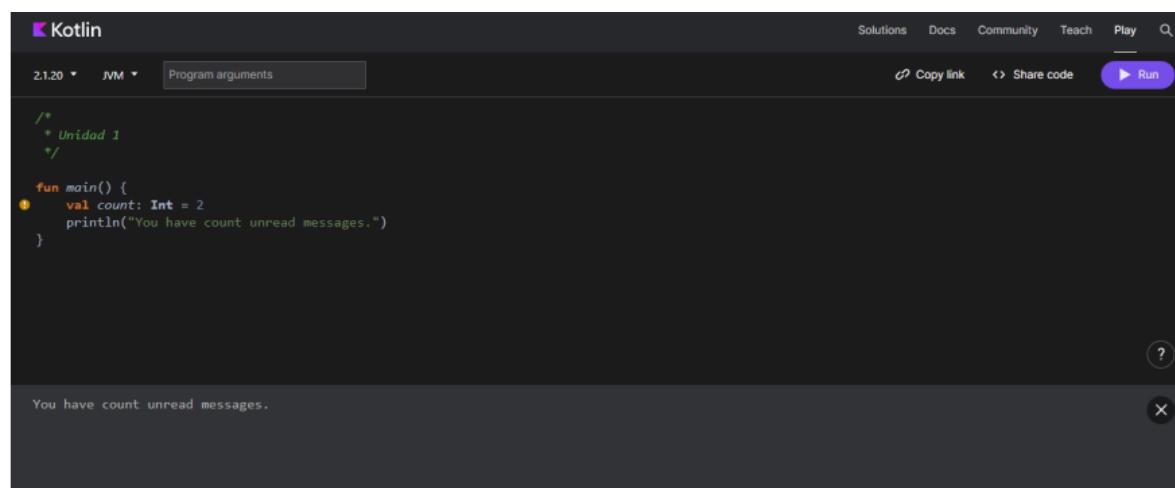
Se explica cómo declarar y utilizar variables, tanto inmutables (val) como mutables (var). Las variables inmutables no pueden cambiar su valor después de ser asignadas, mientras que las mutables sí pueden modificarse durante la ejecución del programa.



The screenshot shows the Kotlin Play IDE interface. The code editor contains the following code:

```
/*
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {
    val count: Int = 2
    println(count)
}
```

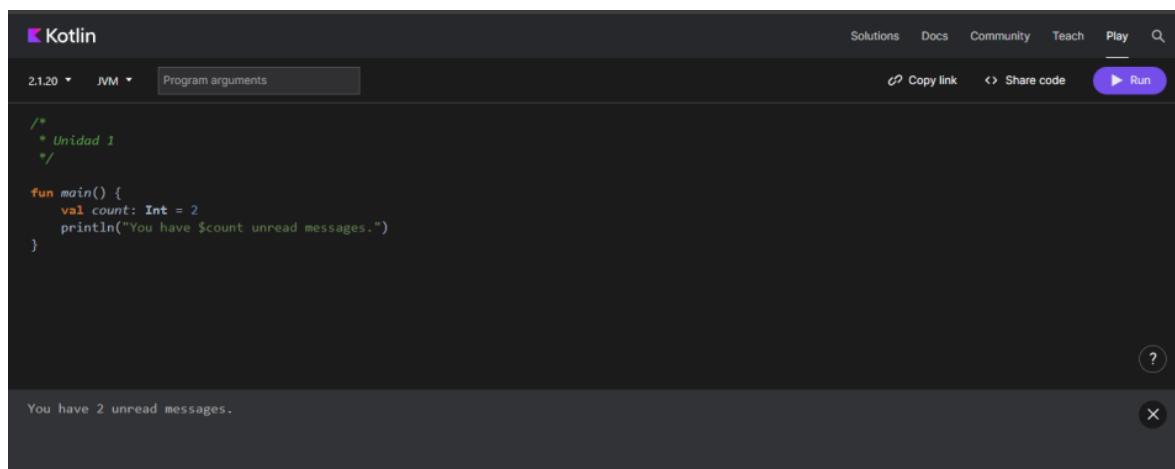
The output window below the editor shows the result of running the program: "2".



The screenshot shows the Kotlin Play IDE interface. The code editor contains the following code:

```
/*
 * Unidad 1
 */
fun main() {
    val count: Int = 2
    println("You have count unread messages.")
}
```

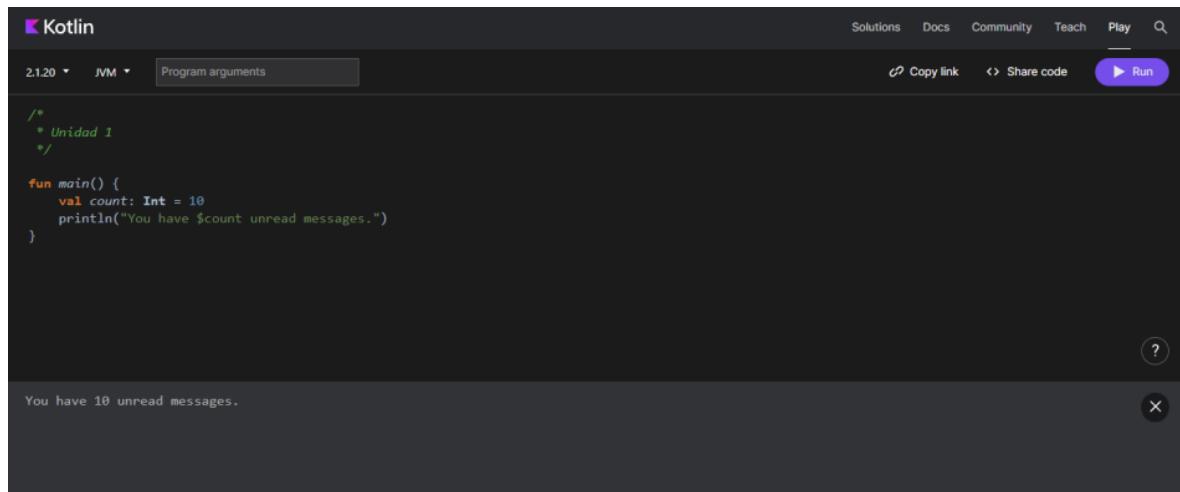
The output window below the editor shows the result of running the program: "You have count unread messages." (with a red circle around the word "count").



The screenshot shows the Kotlin Play IDE interface. The code editor contains the following code:

```
/*
 * Unidad 1
 */
fun main() {
    val count: Int = 2
    println("You have $count unread messages.")
}
```

The output window below the editor shows the result of running the program: "You have 2 unread messages." (with a red circle around the value "2").



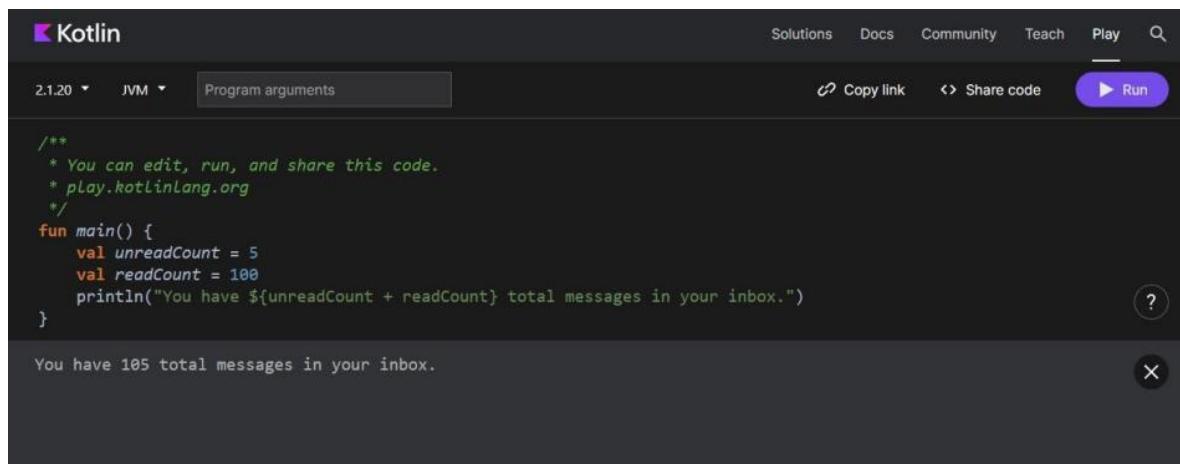
The screenshot shows the Kotlin Play IDE interface. At the top, there's a navigation bar with links for Solutions, Docs, Community, Teach, Play, and a search icon. Below the navigation bar, the version is set to 2.1.20 and the JVM target is selected. A "Program arguments" input field is empty. On the right side of the toolbar, there are "Copy link", "Share code", and a "Run" button. The main area contains the following Kotlin code:

```
/*
 * Unidad 1
 */

fun main() {
    val count: Int = 10
    println("You have $count unread messages.")
}
```

When the code is run, the output window displays the message "You have 10 unread messages." with a copy icon and a close icon.

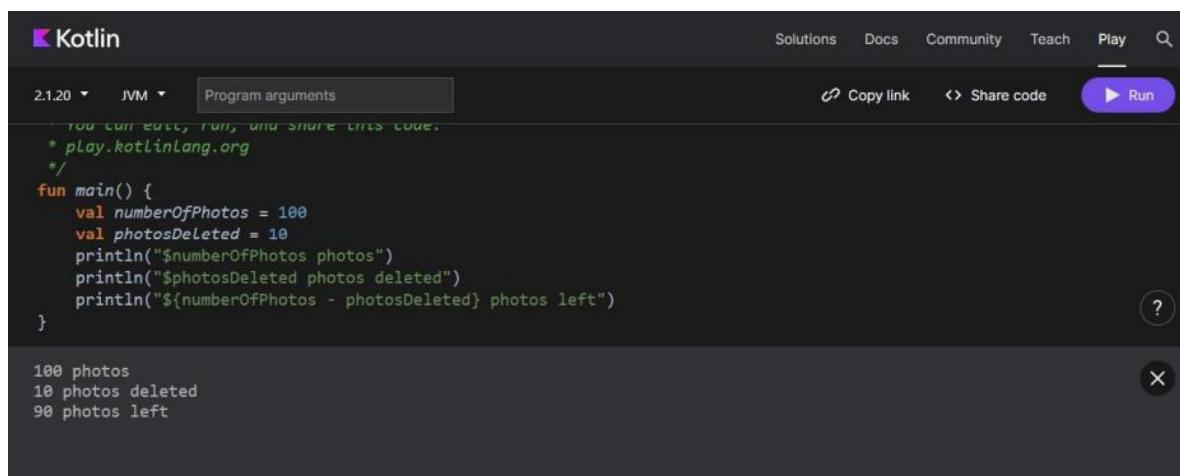
Además, se nos muestra que en Kotlin, el símbolo “\$” se utiliza dentro de cadenas para insertar el valor de una variable, y “\${...}” permite incluir expresiones o cálculos directamente en el texto. Esto facilita combinar texto con datos sin tener que concatenar manualmente.



The screenshot shows the Kotlin Play IDE interface. At the top, there's a navigation bar with links for Solutions, Docs, Community, Teach, Play, and a search icon. Below the navigation bar, the version is set to 2.1.20 and the JVM target is selected. A "Program arguments" input field is empty. On the right side of the toolbar, there are "Copy link", "Share code", and a "Run" button. The main area contains the following Kotlin code:

```
/**
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {
    val unreadCount = 5
    val readCount = 100
    println("You have ${unreadCount + readCount} total messages in your inbox.")
}
```

When the code is run, the output window displays the message "You have 105 total messages in your inbox." with a copy icon and a close icon.

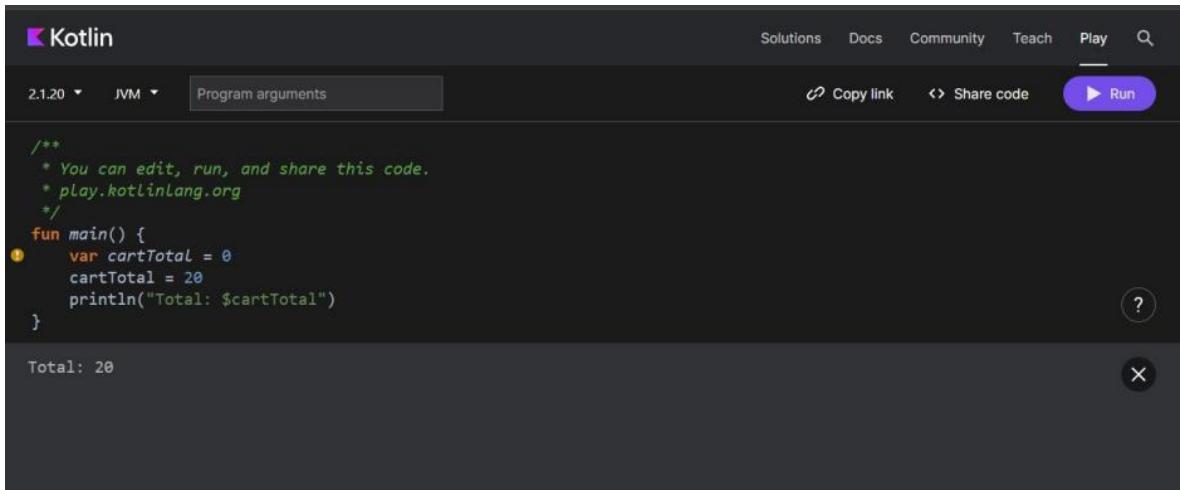


The screenshot shows the Kotlin Play IDE interface. At the top, there's a navigation bar with links for Solutions, Docs, Community, Teach, Play, and a search icon. Below the navigation bar, the version is set to 2.1.20 and the JVM target is selected. A "Program arguments" input field is empty. On the right side of the toolbar, there are "Copy link", "Share code", and a "Run" button. The main area contains the following Kotlin code:

```
/*
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {
    val numberOfPhotos = 100
    val photosDeleted = 10
    println("$numberOfPhotos photos")
    println("$photosDeleted photos deleted")
    println("${numberOfPhotos - photosDeleted} photos left")
}

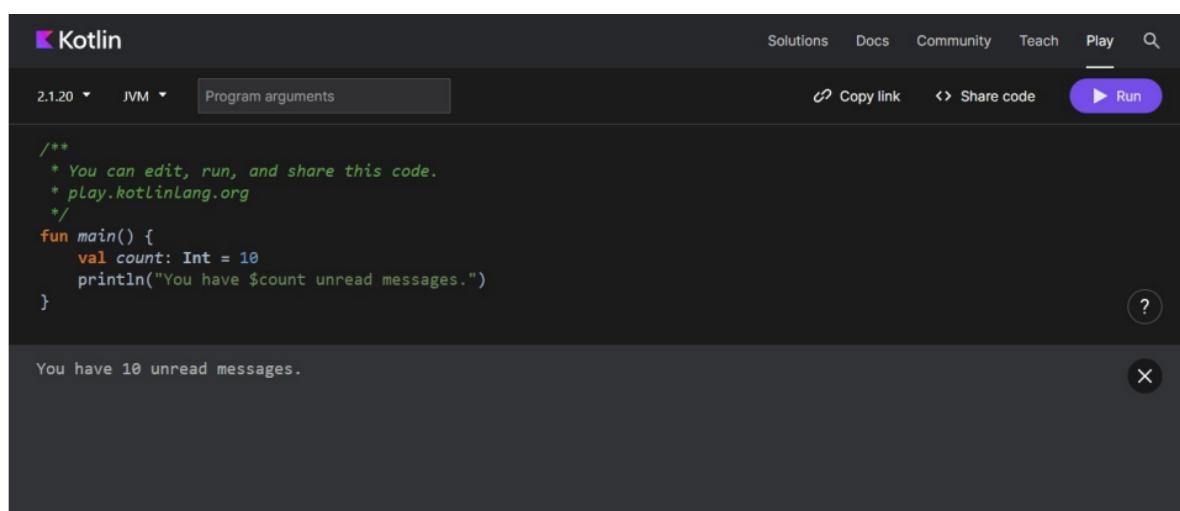
100 photos
10 photos deleted
90 photos left
```

When the code is run, the output window displays the messages "100 photos", "10 photos deleted", and "90 photos left" with a copy icon and a close icon.



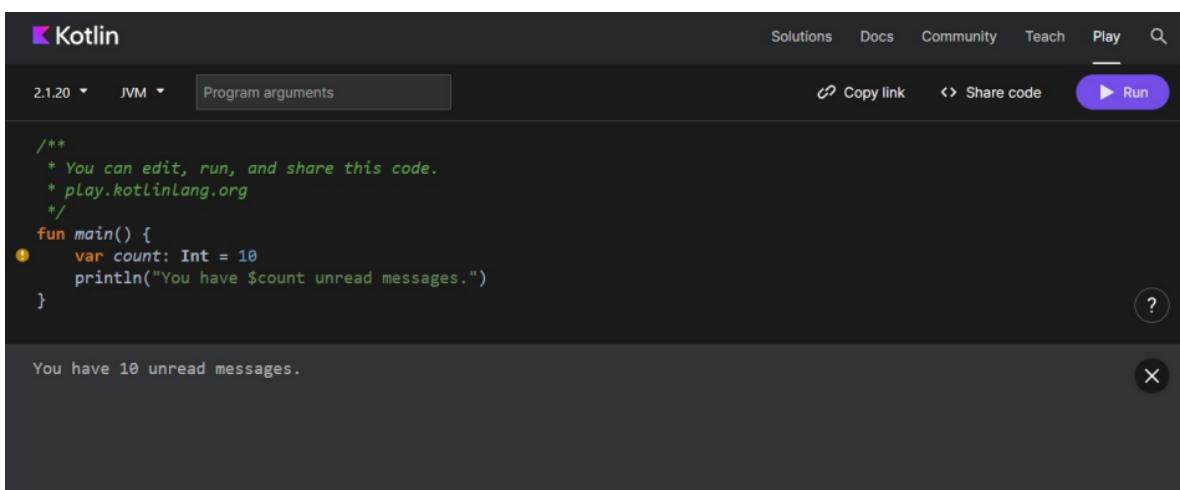
The screenshot shows the Kotlin Play IDE interface. At the top, there's a navigation bar with links for Solutions, Docs, Community, Teach, and Play. Below the navigation bar, there are dropdown menus for version (2.1.20) and JVM, and a "Program arguments" input field. On the right side of the header are buttons for "Copy link", "Share code", and a purple "Run" button. The main area displays a Kotlin script. The script contains a comment block, a function main with a local variable cartTotal initialized to 0, and a println statement outputting "Total: \$cartTotal". The output window below the code shows the result "Total: 20".

```
/**  
 * You can edit, run, and share this code.  
 * play.kotlinlang.org  
 */  
fun main() {  
    var cartTotal = 0  
    cartTotal = 20  
    println("Total: $cartTotal")  
}  
  
Total: 20
```



This screenshot shows the same basic setup as the first one, with the Kotlin Play IDE interface at the top. The code in the editor is identical to the first screenshot, but it uses a `val` declaration instead of a `var` for the variable `count`. The output window shows the result "You have 10 unread messages."

```
/**  
 * You can edit, run, and share this code.  
 * play.kotlinlang.org  
 */  
fun main() {  
    val count: Int = 10  
    println("You have $count unread messages.")  
}  
  
You have 10 unread messages.
```



This screenshot shows the same setup again. The code has been modified to demonstrate increment and decrement operations. It uses a `var` declaration for `count`, initializes it to 10, and then prints its value. The output window shows the result "You have 10 unread messages.", which is likely a mistake or a placeholder for the expected output.

```
/**  
 * You can edit, run, and share this code.  
 * play.kotlinlang.org  
 */  
fun main() {  
    var count: Int = 10  
    println("You have $count unread messages.")  
}  
  
You have 10 unread messages.
```

También se nos muestra cómo modificar el valor de una variable utilizando operaciones de incremento (“`count++`” o “`count = count + 1`”) y decremento (“`count—`”). Estas operaciones ajustan el valor de la variable en una unidad: “`count++`” aumenta el valor de “`count`” en 1, mientras que “`count—`” lo disminuye en 1.

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/*
 * Unidad 1
 */

fun main() {
    var count = 10
    println("You have $count unread messages.")
    count = count + 1
    println("You have $count unread messages.")
}
```

You have 10 unread messages.
You have 11 unread messages.

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/*
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {
    var count = 10
    println("You have $count unread messages.")
    count++
    println("You have $count unread messages.")
}
```

You have 10 unread messages.
You have 11 unread messages.

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/*
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {
    var count = 10
    println("You have $count unread messages.")
    count--
    println("You have $count unread messages.")
}
```

You have 10 unread messages.
You have 9 unread messages.

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/*
 * This program displays the number of messages
 * in the user's inbox.
 */
fun main() {
    // Create a variable for the number of unread messages.
    var count = 10
    println("You have $count unread messages.")

    // Decrease the number of messages by 1.
    count--
    println("You have $count unread messages.")
}
```

You have 10 unread messages.
You have 9 unread messages.

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/*
 * Unidad 1
 */
fun main() {
    val trip1: Double = 3.20
    val trip2: Double = 4.10
    val trip3: Double = 1.72
    val totalTripLength: Double = trip1 + trip2 + trip3
    println("$totalTripLength miles left to destination")
}
```

9.02 miles left to destination

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/*
 * Unidad 1
 */
fun main() {
    val nextMeeting = "Next meeting:"
    val date = "January 1"
    val reminder = nextMeeting + date
    println(reminder)
}
```

Next meeting:January 1

The screenshot shows the Kotlin Play IDE interface. At the top, there's a navigation bar with links for Solutions, Docs, Community, Teach, Play, and a search icon. Below the bar, the version is set to 2.1.20 and the JVM target is selected. A "Program arguments" input field is present. On the right, there are buttons for Copy link, Share code, and Run. The main area contains the following Kotlin code:

```
/*
 * Unidad 1
 */

fun main() {
    val nextMeeting = "Next meeting: "
    val date = "January 1"
    val reminder = nextMeeting + date + " at work"
    println(reminder)
}
```

Below the code, the output window displays the result: "Next meeting: January 1 at work".

se abordan distintos tipos de datos como numéricos, textos y booleanos, y se muestra cómo se pueden asignar y usar en el código, así como la importancia de los comentarios, tanto de una sola línea (//) como de varias líneas (* *), para mejorar la legibilidad del código y facilitar su comprensión. Además, se destaca la buena práctica de utilizar nombres descriptivos para las variables y agregar comentarios que expliquen su propósito, especialmente cuando el código es complejo. Finalmente, se muestra cómo las variables pueden ser utilizadas en el flujo del programa, como en cálculos o para mostrar resultados en pantalla

The screenshot shows the Kotlin Play IDE interface. At the top, there's a navigation bar with links for Solutions, Docs, Community, Teach, Play, and a search icon. Below the bar, the version is set to 2.1.20 and the JVM target is selected. A "Program arguments" input field is present. On the right, there are buttons for Copy link, Share code, and Run. The main area contains the following Kotlin code:

```
/*
 * Unidad 1
 */

fun main() {
    println("Say \"hello\"")
}
```

Below the code, the output window displays the result: "Say hello".

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/*
 * Unidad 1
 */

fun main() {
    val notificationsEnabled: Boolean = true
    println(notificationsEnabled)
}
```

true

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/*
 * Unidad 1
 */

fun main() {
    val notificationsEnabled: Boolean = false
    println(notificationsEnabled)
}
```

false

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/*
 * Unidad 1
 */

fun main() {
    val notificationsEnabled: Boolean = false
    println("Are notifications enabled? " + notificationsEnabled)
}
```

Are notifications enabled? false

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for 'Solutions', 'Docs', 'Community', 'Teach', 'Play', and a search icon. Below the tabs, there are dropdown menus for '2.1.20' and 'JVM'. A 'Program arguments' input field is present. On the right, there are buttons for 'Copy link', 'Share code', and 'Run'. The main area contains the following Kotlin code:

```
/*
 * This program displays the number of messages
 * in the user's inbox.
 */
fun main() {
    // Create a variable for the number of unread messages.
    var count = 10
    println("You have $count unread messages.")

    // Decrease the number of messages by 1.
    count--
    println("You have $count unread messages.")
}
```

Below the code, the output window shows two lines of text: "You have 10 unread messages." and "You have 9 unread messages.", each followed by a close button.

En Kotlin, una función se define utilizando la palabra clave `fun`, y permite agrupar fragmentos de código reutilizable, lo que facilita el mantenimiento de programas grandes y evita la repetición innecesaria. Las funciones pueden recibir parámetros, que son variables definidas entre paréntesis y separadas por comas, y pueden devolver valores que pueden almacenarse para su uso posterior. Al llamar una función, se deben pasar argumentos que coincidan con los tipos y el orden de los parámetros, aunque Kotlin también permite el uso de argumentos con nombre, lo cual facilita la lectura del código y permite cambiar el orden en que se pasan los valores sin alterar el resultado.

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for 'Solutions', 'Docs', 'Community', 'Teach', 'Play', and a search icon. Below the tabs, there are dropdown menus for '2.1.20' and 'JVM'. A 'Program arguments' input field is present. On the right, there are buttons for 'Copy link', 'Share code', and 'Run'. The main area contains the following Kotlin code:

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

fun birthdayGreeting() {
    println("Happy Birthday, Rover!")
    println("You are now 5 years old!")
}
```

Below the code, the output window shows two lines of text: "Happy Birthday, Rover!" and "You are now 5 years old!", each followed by a close button.

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
fun main() {
    println(birthdayGreeting())
}

fun birthdayGreeting(): String {
    val nameGreeting = "Happy Birthday, Rover!"
    val ageGreeting = "You are now 5 years old!"
    return "$nameGreeting\n$ageGreeting"
}
```

Happy Birthday, Rover!
You are now 5 years old!

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
fun main() {
    println(birthdayGreeting("Rover"))
    println(birthdayGreeting("Rex"))
}

fun birthdayGreeting(name: String): String {
    val nameGreeting = "Happy Birthday, $name!"
    val ageGreeting = "You are now 5 years old!"
    return "$nameGreeting\n$ageGreeting"
}
```

Happy Birthday, Rover!
You are now 5 years old!
Happy Birthday, Rover!
You are now 5 years old!

The screenshot shows the Kotlin Play IDE interface. At the top, there are dropdown menus for 'Solutions', 'Docs', 'Community', 'Teach', 'Play', and a search bar. Below the menu bar, the version '2.1.20' and 'JVM' are selected. A 'Program arguments' input field contains the text 'Rover Rex'. To the right of the code editor are buttons for 'Copy link', 'Share code', and 'Run'. The code in the editor is:

```
fun main() {
    println(birthdayGreeting("Rover", 5))
    println(birthdayGreeting("Rex", 2))
}

fun birthdayGreeting(name: String, age: Int): String {
    val nameGreeting = "Happy Birthday, $name!"
    val ageGreeting = "You are now $age years old!"
    return "$nameGreeting\n$ageGreeting"
}
```

The output window below the code editor displays the results of the program execution:

```
Happy Birthday, Rover!
You are now 5 years old!
Happy Birthday, Rex!
You are now 2 years old!
```

This screenshot shows the same setup as the first one, but with a modification in the main function call. The 'Program arguments' input field now contains 'Rex 5'. The code and its output remain the same as in the first screenshot.

```
fun main() {
    println(birthdayGreeting(name = "Rover", age = 5))
    println(birthdayGreeting(age = 2, name = "Rex"))
}

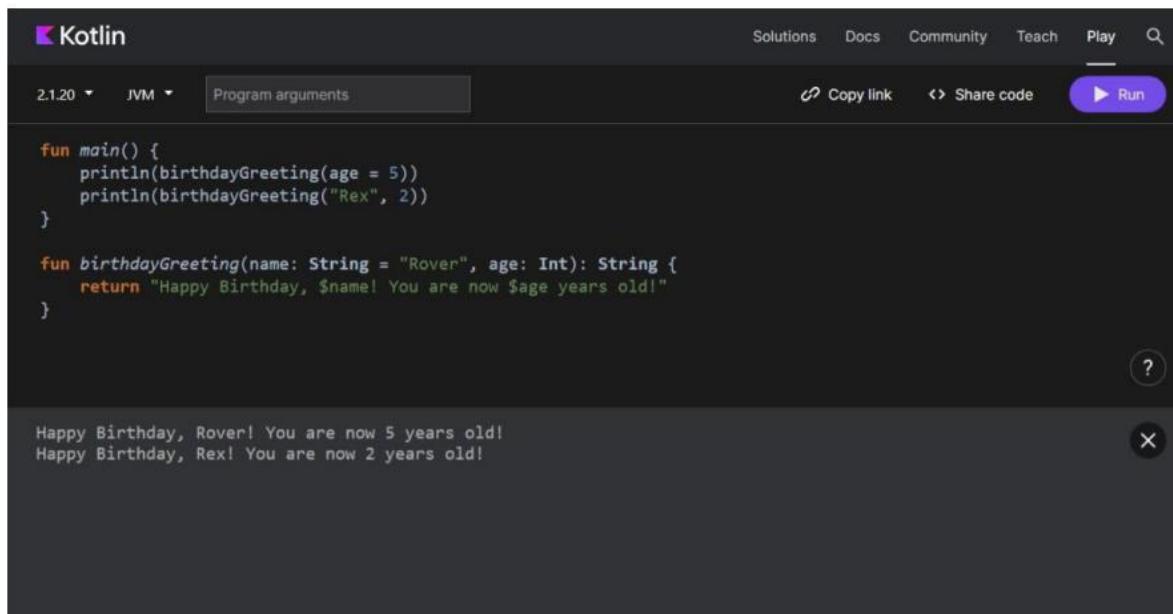
fun birthdayGreeting(name: String, age: Int): String {
    val nameGreeting = "Happy Birthday, $name!"
    val ageGreeting = "You are now $age years old!"
    return "$nameGreeting\n$ageGreeting"
}
```

Output:

```
Happy Birthday, Rover!
You are now 5 years old!
Happy Birthday, Rex!
You are now 2 years old!
```

En este caso se define una función llamada “birthdayGreeting” como se mencionaba anteriormente que toma un nombre (con un valor predeterminado de "Rover") y una edad como entrada, y devuelve una cadena formateada con un mensaje de cumpleaños personalizado. La función principal (main) llama a birthdayGreeting dos veces: la primera vez proporcionando solo la edad (5), lo que hace que se utilice el nombre por defecto "Rover", y la segunda vez proporcionando tanto el nombre ("Rex") como la edad (2). Las cadenas de cumpleaños resultantes de estas

llamadas se imprimen en la consola, mostrando los mensajes "Happy Birthday, ¡Rover! You are now 5 years old!" y "Happy Birthday, Rex! You are now 2 years old!".

A screenshot of the Kotlin Play IDE interface. At the top, there's a navigation bar with 'Solutions', 'Docs', 'Community', 'Teach', 'Play', and a search icon. Below the navigation bar, there are dropdown menus for '2.1.20' and 'JVM', and a 'Program arguments' input field. To the right of these are 'Copy link', 'Share code', and a 'Run' button. The main area contains the following Kotlin code:

```
fun main() {
    println(birthdayGreeting(age = 5))
    println(birthdayGreeting("Rex", 2))
}

fun birthdayGreeting(name: String = "Rover", age: Int): String {
    return "Happy Birthday, $name! You are now $age years old!"
}
```

The output window below the code shows the printed messages:

```
Happy Birthday, Rover! You are now 5 years old!
Happy Birthday, Rex! You are now 2 years old!
```

Problemas prácticos:

Conceptos básicos de Kotlin Se muestra cómo imprimir mensajes en consola utilizando `println()`, así como el uso de plantillas de cadenas, lo cual permite insertar valores de variables dentro de una cadena de texto de manera directa utilizando el símbolo `"$"`, lo que hace que el código sea más limpio y legible. Se profundiza también en la concatenación de cadenas, una operación común en la programación, y cómo el uso de operadores como `“+”` que puede ayudar a combinar textos y variables.

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
fun main() {  
    println("Use the val keyword when the value doesn't change.")  
    println("Use the var keyword when the value can change.")  
    println("When you define a function, you define the parameters that can be passed to it.")  
    println("When you call a function, you pass arguments for the parameters.")  
}
```

Use the val keyword when the value doesn't change.
Use the var keyword when the value can change.
When you define a function, you define the parameters that can be passed to it.
When you call a function, you pass arguments for the parameters.

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
fun main() {  
    println("New chat message from a friend")  
}
```

New chat message from a friend

Kotlin

Solutions Docs Community Teach Play

2.1.20 ▾ JVM ▾ Program arguments

Copy link Share code Run

```
fun main() {
    val discountPercentage = 20
    val item = "Google Chromecast"
    val offer = "Sale - Up to $discountPercentage% discount off $item! Hurry Up!"

    println(offer)
}
```

Sale - Up to 20% discount off Google Chromecast! Hurry Up!

Kotlin

Solutions Docs Community Teach Play

2.1.20 ▾ JVM ▾ Program arguments

Copy link Share code Run

```
fun main() {
    val numberOfAdults = 20
    val numberOfKids = 30
    val total = numberOfAdults + numberOfKids
    println("The total party size is: $total")
}
```

The total party size is: 50

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for Solutions, Docs, Community, Teach, Play, and a search icon. Below the tabs, the version is listed as 2.1.20 and the JVM target is selected. A "Program arguments" input field contains no text. On the right, there are buttons for Copy link, Share code, and Run, with the Run button highlighted in purple. The main area displays the following Kotlin code:

```
fun main() {
    val baseSalary = 5000
    val bonusAmount = 1000
    val totalSalary = "$baseSalary + $bonusAmount"
    println("Congratulations for your bonus! You will receive a total of $totalSalary (additional bonus).")
}
```

Below the code, a message box appears with the text "Congratulations for your bonus! You will receive a total of 5000 + 1000 (additional bonus.)".

También se explora cómo realizar operaciones matemáticas básicas, como suma y resta, y cómo encapsular estas operaciones en funciones para reutilizarlas. Además, se presenta la creación de funciones con parámetros predeterminados, lo que permite que ciertos argumentos tengan valores por defecto si no se especifican al llamar a la función.

Se destaca también la importancia de seguir las convenciones de nomenclatura de Kotlin, como usar notación camelCase para las variables y funciones, lo cual mejora la claridad y mantenibilidad del código. Se toca el tema de la comparación de valores, utilizando operadores como “>”, y cómo Kotlin evalúa las comparaciones como valores booleanos.

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for Solutions, Docs, Community, Teach, Play, and a search icon. Below the tabs, the version is listed as 2.1.20 and the JVM target is selected. A "Program arguments" input field contains no text. On the right, there are buttons for Copy link, Share code, and Run, with the Run button highlighted in purple. The main area displays the following Kotlin code:

```
fun main() {
    val firstNumber = 10
    val secondNumber = 5
    val thirdNumber = 8

    val result = add(firstNumber, secondNumber)
    val anotherResult = subtract(firstNumber, thirdNumber)

    println("$firstNumber + $secondNumber = $result")
    println("$firstNumber - $thirdNumber = $anotherResult")
}

fun add(firstNumber: Int, secondNumber: Int): Int {
    return firstNumber + secondNumber
}

fun subtract(firstNumber: Int, secondNumber: Int): Int {
    return firstNumber - secondNumber
}
```

Below the code, the output window shows the results of the calculations: "10 + 5 = 15" and "10 - 8 = 2".

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
fun main() {
    val firstUserEmailId = "user_one@gmail.com"

    // The following line of code assumes that you named your parameter as emailId.
    // If you named it differently, feel free to update the name.
    println(displayAlertMessage(emailId = firstUserEmailId))
    println()

    val secondUserOperatingSystem = "Windows"
    val secondUserEmailId = "user_two@gmail.com"

    println(displayAlertMessage(secondUserOperatingSystem, secondUserEmailId))
    println()

    val thirdUserOperatingSystem = "Mac OS"
    val thirdUserEmailId = "user_three@gmail.com"

    println(displayAlertMessage(thirdUserOperatingSystem, thirdUserEmailId))
    println()
}

fun displayAlertMessage(
    operatingSystem: String = "Unknown OS",
    emailId: String
): String {
    return "There is a new sign-in request on $operatingSystem for your Google Account $emailId."
}
```

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
fun main() {
    val firstUserEmailId = "user_one@gmail.com"

    // The following line of code assumes that you named your parameter as emailId.
    // If you named it differently, feel free to update the name.
    println(displayAlertMessage(emailId = firstUserEmailId))
    println()

    val secondUserOperatingSystem = "Windows"
    val secondUserEmailId = "user_two@gmail.com"

    println(displayAlertMessage(secondUserOperatingSystem, secondUserEmailId))
}

There is a new sign-in request on Unknown OS for your Google Account user_one@gmail.com.

There is a new sign-in request on Windows for your Google Account user_two@gmail.com.

There is a new sign-in request on Mac OS for your Google Account user_three@gmail.com.
```

En este codelab también se representó un podómetro, que es un dispositivo usado para contar los pasos que una persona da al caminar, comúnmente integrado en teléfonos y relojes inteligentes.

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for Solutions, Docs, Community, Teach, and Play, with a search icon. Below the tabs, it says "2.1.20" and "JVM". A "Program arguments" input field contains nothing. To the right are "Copy link", "Share code", and a "Run" button. The code area contains the following Kotlin code:

```
fun main() {
    val steps = 4000
    val caloriesBurned = pedometerStepsToCalories(steps)
    println("Walking $steps steps burns $caloriesBurned calories")
}

fun pedometerStepsToCalories(numberOfSteps: Int): Double {
    val caloriesBurnedForEachStep = 0.04
    val totalCaloriesBurned = numberOfSteps * caloriesBurnedForEachStep
    return totalCaloriesBurned
}
```

The output window below shows the result of running the program: "Walking 4000 steps burns 160.0 calories".

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for Solutions, Docs, Community, Teach, and Play, with a search icon. Below the tabs, it says "2.1.20" and "JVM". A "Program arguments" input field contains nothing. To the right are "Copy link", "Share code", and a "Run" button. The code area contains the following Kotlin code:

```
fun main() {
    println("Have I spent more time using my phone today: ${compareTime(300, 250)}")
    println("Have I spent more time using my phone today: ${compareTime(300, 300)}")
    println("Have I spent more time using my phone today: ${compareTime(200, 220)}")
}

fun compareTime(timeSpentToday: Int, timeSpentYesterday: Int): Boolean {
    return timeSpentToday > timeSpentYesterday
}
```

The output window below shows the results of the three comparisons: "Have I spent more time using my phone today: true", "Have I spent more time using my phone today: false", and "Have I spent more time using my phone today: false".

Por último, se enseña a mover código duplicado a funciones, lo que mejora la reutilización y organización del código. La centralización de tareas repetitivas en una sola función permite que, si se necesita modificar la lógica, solo sea necesario hacerlo en un único lugar, evitando la duplicación de código y mejorando la eficiencia del mantenimiento del proyecto.

The screenshot shows the Kotlin Play IDE interface. At the top, there's a navigation bar with links for Solutions, Docs, Community, Teach, Play, and a search icon. Below the navigation bar, there are dropdown menus for version (2.1.20) and JVM, and a "Program arguments" input field. On the right side of the header, there are "Copy link", "Share code", and a "Run" button. The main area displays a Kotlin code snippet. The code defines a `main` function that calls a `printWeatherForCity` function for four cities: Ankara, Tokyo, Cape Town, and Guatemala City. The `printWeatherForCity` function takes a city name and three integers representing low temperature, high temperature, and chance of rain, then prints a message for each. The output window at the bottom shows the results for each city.

```
fun main() {
    printWeatherForCity("Ankara", 27, 31, 82)
    printWeatherForCity("Tokyo", 32, 36, 10)
    printWeatherForCity("Cape Town", 59, 64, 2)
    printWeatherForCity("Guatemala City", 50, 55, 7)
}

fun printWeatherForCity(cityName: String, lowTemp: Int, highTemp: Int, chanceOfRain: Int) {
    println("City: $cityName")
    println("Low temperature: $lowTemp, High temperature: $highTemp")
    println("Chance of rain: $chanceOfRain%")
    println()
}
```

City: Ankara
Low temperature: 27, High temperature: 31
Chance of rain: 82%

City: Tokyo
Low temperature: 32, High temperature: 36
Chance of rain: 10%

City: Cape Town
Low temperature: 59, High temperature: 64
Chance of rain: 2%

Problemas prácticos:

Conceptos básicos de Compose Para el artículo de Compose nos centramos en la creación de una pantalla en la aplicación “Learn Together”, que presenta un artículo sobre Jetpack Compose. Se proporciona una estructura para crear una interfaz de usuario utilizando componentes composable de Jetpack Compose, donde en la clase “MainActivity”, se configura el contenido de la actividad utilizando “setContent”, donde se aplica el tema “ComposeArticleTheme” y se llama a la función “ComposeArticleApp()”, que a su vez muestra un artículo con el título, descripción corta y larga, y una imagen de fondo. La función “ComposeArticleApp” es responsable de pasar los recursos de texto e imagen a la función “ArticleCard”, que organiza estos elementos en una columna vertical (Column). Dentro de esta columna, se coloca una imagen que ocupa todo el ancho de la pantalla, seguida de tres elementos de texto: el título con un tamaño de fuente de 24sp y padding de 16dp, la descripción corta y larga con un texto justificado y padding de 16dp en los lados y en la parte superior e inferior.

The screenshot shows the Android Studio interface with the code editor open to `MainActivity.kt`. The code implements a `ComponentActivity` with a `setContent` block containing a `Column` with `Image`, `Text`, and `Text` elements. The `Image` uses `painterResource(R.drawable.bg_compose_background)`. The `Text` elements have specific styling like bold font and padding. To the right, the "Running Devices" window displays a preview of the app's UI on a "Xiaomi M2101K7BL" device, showing a blue background with a green icon and two text labels: "All tasks completed" and "Nice work!". Below the preview is a descriptive text about Jetpack Compose.

```
40     class MainActivity : ComponentActivity() {
41         override fun onCreate(savedInstanceState: Bundle?) {
42             setContent {
43                 ...
44             }
45         }
46     }
47
48     @Composable
49     fun ComposeArticleApp() {
50         ArticleCard(
51             title = stringResource(R.string.compose_tutorial_title),
52             shortDescription = stringResource(R.string.compose_tutorial_short_desc),
53             longDescription = stringResource(R.string.compose_tutorial_long_desc),
54             imagePainter = painterResource(R.drawable.bg_compose_background)
55         )
56     }
57
58     @Composable
59     private fun ArticleCard(
60         title: String,
61         shortDescription: String,
62         longDescription: String,
63         imagePainter: Painter,
64         modifier: Modifier = Modifier,
65     ) {
66         Column(modifier = modifier) {
67             Image(painter = imagePainter, contentDescription = null)
68             Text(
69                 text = title
70             )
71             Text(
72                 text = shortDescription
73             )
74             Text(
75                 text = longDescription
76             )
77         }
78     }
79 }
```

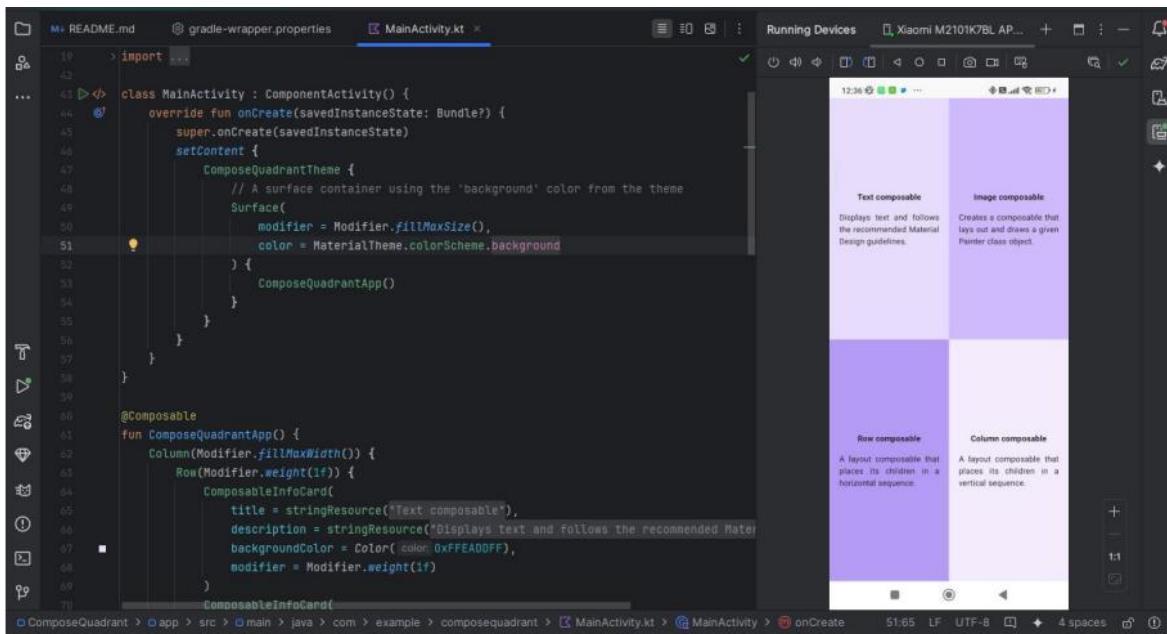
Para el administrador de tareas se construye una pantalla en donde se muestra cuando el usuario ha completado todas sus tareas del día. La interfaz muestra de manera clara y visual que no quedan tareas pendientes. Para ello, se proporcionan recursos como una imagen ilustrativa y dos cadenas de texto: "All tasks completed" y "Nice work!". En el código se implementa la pantalla usando Jetpack Compose. Dentro de "MainActivity", se configura la UI con "setContent" y se llama a "TaskCompletedApp()", que contiene el contenido principal de la pantalla. El diseño se basa en un `Column`, centrado tanto vertical como horizontalmente usando "`verticalArrangement = Arrangement.Center`" y "`horizontalAlignment = Alignment.CenterHorizontally`", para cumplir con las especificaciones de alineación. Dentro de esta columna se inserta primero una imagen (`Image`) usando "`painterResource`", seguida de dos elementos de texto (`Text`). El primero, "All tasks completed", se muestra con estilo de fuente en negrita y paddings superior e inferior de 24dp y 8dp, respectivamente. El segundo texto de "Nice work!", se muestra con un tamaño de fuente de 16sp.

The screenshot shows the Android Studio interface. On the left, the code editor displays `MainActivity.kt` with the following content:

```
43 class MainActivity : ComponentActivity() {
44     override fun onCreate(savedInstanceState: Bundle?) {
45         setContent {
46             TaskCompletedTheme {
47                 modifier = Modifier.fillMaxSize(),
48                 color = MaterialTheme.colorScheme.background
49             }
50             } {
51                 TaskCompletedScreen()
52             }
53         }
54     }
55 }
56 }
57 }
58 }
59 }
60 @Composable
61 fun TaskCompletedScreen() {
62     Column(
63         modifier = Modifier
64             .fillMaxWidth()
65             .fillMaxHeight(),
66         verticalArrangement = Arrangement.Center,
67         horizontalAlignment = Alignment.CenterHorizontally
68     ) {
69         val image = painterResource(R.drawable.ic_task_completed)
70         Image(painter = image, contentDescription = null)
71         Text(
72             text = stringResource("All tasks completed"),
73             modifier = Modifier.padding(top = 24.dp, bottom = 8.dp),
74             fontWeight = FontWeight.Bold
75         )
76     }
77 }
```

On the right, the "Running Devices" window shows a preview of the app running on a "Xiaomi M2101K7BL API 30" device. The screen displays a large green circle with a blue checkmark inside. Below the icon, the text "All tasks completed" is displayed in bold, followed by "Nice work!" in a smaller font.

En cuanto al cuadrante de Compose se construye una pantalla educativa para una app que muestra información sobre funciones “Composable” en Jetpack Compose, dividiendo la interfaz en cuatro cuadrantes iguales. Cada uno representa una función específica: “Text”, “Image”, “Row” y “Column”, e incluye su nombre en texto en negrita seguido de una breve descripción. El diseño se logra usando un Column principal que contiene dos Row, cada una con dos elementos hijos que representan los cuadrantes. Para dividir la pantalla de manera equitativa, se emplea el modificador “Modifier.weight(1f)” tanto en filas como en columnas, lo que permite que cada sección ocupe el mismo espacio vertical u horizontal según corresponda. Dentro de cada cuadrante, los elementos están alineados al centro en ambas direcciones (“Alignment.CenterHorizontally” y “Arrangement.Center”) y rodeados por un padding de 16dp. Cada tarjeta (cuadrante) está coloreada con uno de los cuatro colores pastel proporcionados, y contiene dos “Text”: el primero, que es el título (como “Text composable”), se muestra en negrita con padding inferior de 16dp, y el segundo presenta una descripción más larga con fuente por defecto.



Unidad 2:

Creación de una IU de una app

Ruta de Aprendizaje 1 (Conceptos Básicos de Kotlin)

Escribe condicionales en Kotlin En Kotlin, se pueden usar los condicionales “if/else” o “when” para realizar decisiones basadas en condiciones específicas, donde el condicional if/else funciona evaluando una expresión booleana. El bloque de código dentro de la rama “if” se ejecutará solo cuando la expresión sea verdadera. Si la condición del “if” es falsa, el bloque dentro de la rama “else if” (si existe) se evaluará, y si todas las condiciones anteriores fallan, se ejecutará la rama “else” final.

A screenshot of the Kotlin Play IDE. The code editor contains a simple program with a single println statement. The output window below shows the result "true".

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/**  
 * You can edit, run, and share this code.  
 * play.kotlinlang.org  
 */  
fun main() {  
    println(1 < 1)  
}  
  
false
```

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
You can edit, run, and share this code.  
play.kotlinlang.org  
*/  
fun main() {  
    val trafficLightColor = "Red"  
  
    if (trafficLightColor == "Red") {  
        println("Stop")  
    }  
}  
  
Stop
```

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
fun main() {  
    val trafficLightColor = "Red"  
  
    if (trafficLightColor == "Red") {  
        println("Stop")  
    } else {  
        println("Go")  
    }  
}  
  
Stop
```

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/*
fun main() {
    val trafficLightColor = "Green"

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

Go

?

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

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

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

Slow

?

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

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

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

Go

?

A screenshot of the Kotlin Play IDE interface. The code editor shows the following Kotlin code:

```
* play.kotlinlang.org
*/
fun main() {
    val trafficLightColor = "Black"

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

The output window below the code editor displays the result of running the code: "Invalid traffic-light color".

Cuando se tienen más de dos posibles ramas se recomienda utilizar el condicional “when”, ya que facilita la lectura y manejo de múltiples condiciones. “When” puede manejar condiciones más complejas utilizando comas (’,’) para evaluar varias condiciones simultáneamente, rangos con “in”, o la palabra clave “is” para verificar el tipo de una variable.

A screenshot of the Kotlin Play IDE interface. The code editor shows the following Kotlin code using a when expression:

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

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

The output window below the code editor displays the result of running the code: "Slow".

Kotlin

Solutions Docs Community Teach Play 

2.1.20 JVM Program arguments

Copy link Share code Run

```
/**  
 * You can edit, run, and share this code.  
 * play.kotlinlang.org  
 */  
fun main() {  
    val x = 3  
  
    when (x) {  
        2 -> println("x is a prime number between 1 and 10.")  
        3 -> println("x is a prime number between 1 and 10.")  
        5 -> println("x is a prime number between 1 and 10.")  
        7 -> println("x is a prime number between 1 and 10.")  
        else -> println("x isn't a prime number between 1 and 10.")  
    }  
}  
  
x is a prime number between 1 and 10.
```

Kotlin

Solutions Docs Community Teach Play 

2.1.20 JVM Program arguments

Copy link Share code Run

```
/**  
 * You can edit, run, and share this code.  
 * play.kotlinlang.org  
 */  
fun main() {  
    val x = 3  
  
    when (x) {  
        2, 3, 5, 7 -> println("x is a prime number between 1 and 10.")  
        else -> println("x isn't a prime number between 1 and 10.")  
    }  
}  
  
x is a prime number between 1 and 10.
```

Kotlin

Solutions Docs Community Teach Play 

2.1.20 JVM Program arguments

Copy link Share code Run

```
/**  
 * You can edit, run, and share this code.  
 * play.kotlinlang.org  
 */  
fun main() {  
    val x = 4  
  
    when (x) {  
        2, 3, 5, 7 -> println("x is a prime number between 1 and 10.")  
        in 1..10 -> println("x is a number between 1 and 10, but not a prime number.")  
        else -> println("x isn't a prime number between 1 and 10.")  
    }  
}  
  
x is a number between 1 and 10, but not a prime number.
```

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/**  
 * You can edit, run, and share this code.  
 * play.kotlinlang.org  
 */  
fun main() {  
    val x: Any = 20  
  
    when (x) {  
        2, 3, 5, 7 -> println("x is a prime number between 1 and 10.")  
        in 1..10 -> println("x is a number between 1 and 10, but not a prime number.")  
        is Int -> println("x is an integer number, but not between 1 and 10.")  
        else -> println("x isn't an integer number.")  
    }  
}  
  
x is an integer number, but not between 1 and 10.
```

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/**  
 * You can edit, run, and share this code.  
 * play.kotlinlang.org  
 */  
fun main() {  
    val trafficLightColor = "Amber"  
  
    when (trafficLightColor) {  
        "Red" -> println("Stop")  
        "Yellow", "Amber" -> println("Slow")  
        "Green" -> println("Go")  
        else -> println("Invalid traffic-light color")  
    }  
}  
  
Slow
```

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/**  
 * You can edit, run, and share this code.  
 * play.kotlinlang.org  
 */  
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)  
}  
  
Invalid traffic-light color
```

The screenshot shows the Kotlin Play IDE interface. The code editor contains the following Kotlin code:

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

    val message = when(trafficLightColor) {
        "Red" -> "Stop"
        "Yellow", "Amber" -> "Proceed with caution."
        "Green" -> "Go"
        else -> "Invalid traffic-light color"
    }
    println(message)
}

Proceed with caution.
```

The output window below the code editor displays the result of the program execution: "Proceed with caution.". The IDE has a dark theme with light-colored text and icons.

Nulabilidad en Kotlin

En Kotlin, las variables pueden ser anulables, lo que significa que pueden contener “null”. Las variables no anulables no pueden tener “null”.

The screenshot shows the Kotlin Play IDE interface. The code editor contains the following Kotlin code:

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

The output window below the code editor displays the result of the program execution: "null". The IDE has a dark theme with light-colored text and icons.

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/** * 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

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/** * 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

Kotlin

Solutions Docs Community Teach Play

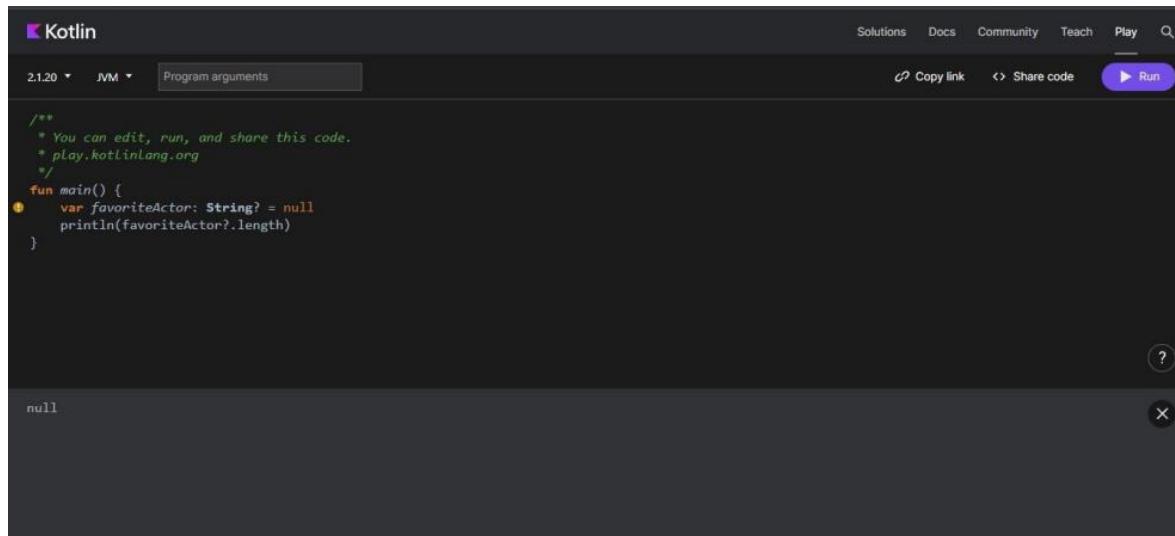
2.1.20 JVM Program arguments

Copy link Share code Run

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

9

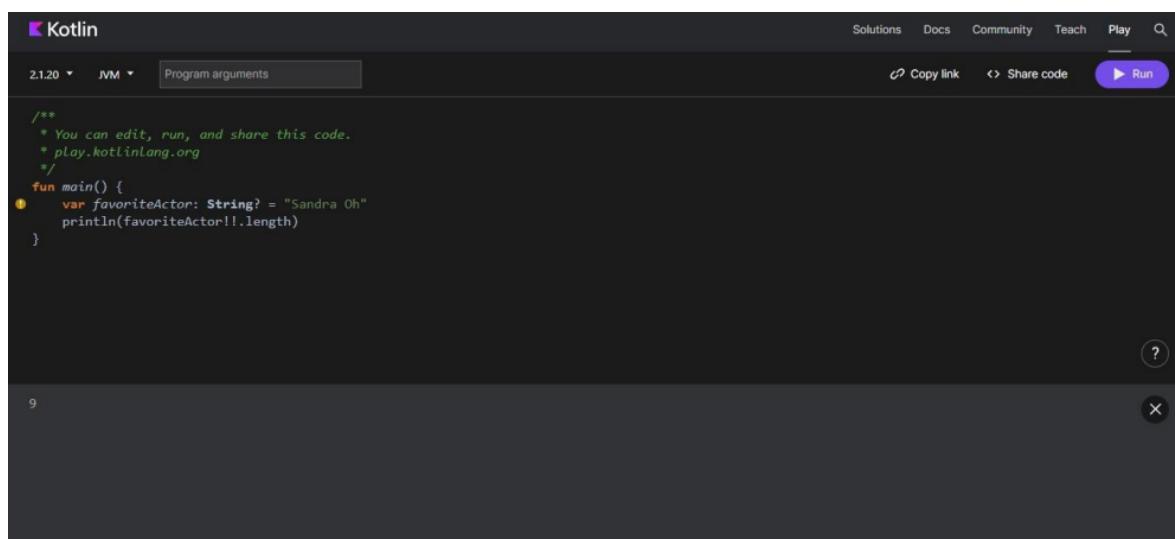
Para acceder a propiedades de variables anulables sin errores, se usan operadores como “?.” (llamada segura) o “!!” (aserción no nula).



The screenshot shows the Kotlin Play IDE interface. The code editor contains the following code:

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

The output window below the editor shows the result of running the code: "null".



The screenshot shows the Kotlin Play IDE interface. The code editor contains the following code:

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

The output window below the editor shows the result of running the code: "9".

También se pueden usar “if/else” para verificar si una variable es “null” y manejirla adecuadamente. El operador Elvis (“?:”) permite proporcionar un valor predeterminado si la variable es “null”.

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for 'Solutions', 'Docs', 'Community', 'Teach', 'Play' (which is selected), and a search bar. Below the tabs, it says '2.1.20' and 'JVM'. A 'Program arguments' input field is present. On the right, there are buttons for 'Copy link', 'Share code', and 'Run' (highlighted in purple). The main area contains the following Kotlin code:

```
/** * 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 ${favoriteActor.length}.")} }
```

Below the code, the output window displays the message: "The number of characters in your favorite actor's name is 9." with a close button (X).

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for 'Solutions', 'Docs', 'Community', 'Teach', 'Play' (selected), and a search bar. Below the tabs, it says '2.1.20' and 'JVM'. A 'Program arguments' input field is present. On the right, there are buttons for 'Copy link', 'Share code', and 'Run' (highlighted in purple). The main area contains the same Kotlin code as the first screenshot.

```
/** * 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 ${favoriteActor.length}.")} else { println("You didn't input a name.") } }
```

Below the code, the output window displays the message: "You didn't input a name." with a close button (X).

En el siguiente código Kotlin podemos observar que se comienza declarando una variable mutable y nullable llamada “favoriteActor”, a la cual se le asigna inicialmente la cadena "Sandra Oh". Posteriormente, se declara una variable inmutable llamada “lengthOfName”, cuyo valor se determina mediante una expresión condicional “ifelse”. Esta condición verifica si la variable “favoriteActor” no es nula; en caso afirmativo, se obtiene la longitud de la cadena almacenada en “favoriteActor”, mientras que, si es nula, se asigna el valor 0 a “lengthOfName”.

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for Solutions, Docs, Community, Teach, Play, and a search icon. Below the tabs, it says "2.1.20" and "JVM". There is a "Program arguments" input field containing nothing. On the right, there are buttons for "Copy link", "Share code", and "Run". The main area contains the following Kotlin code:

```
/*
 * 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 $lengthOfName.")
}
```

Below the code, the output window shows the result: "The number of characters in your favorite actor's name is 9." with a close button.

Por último, en la función principal (main) se comienza con la declaración de una variable mutable y que puede contener valores nulos, “favoriteActor”, a la cual se le asigna la cadena “Sandra Oh”. La siguiente línea calcula la longitud del nombre del actor favorito y la asigna a una variable inmutable llamada “lengthOfName”. Para manejar la posibilidad de que “favoriteActor” sea nulo, se emplea el operador elvis (“?:”). La expresión “favoriteActor?.length” realiza una llamada segura: si “favoriteActor” no es nulo, se accede a su propiedad length; de lo contrario, la expresión completa evalúa a null.

The screenshot shows the same setup as the first one, but the code has been modified to use the safe call operator (?.) instead of the Elvis operator (?:). The code is identical to the one above, except for the assignment of lengthOfName.

```
/*
 * 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 $lengthOfName.")
}
```

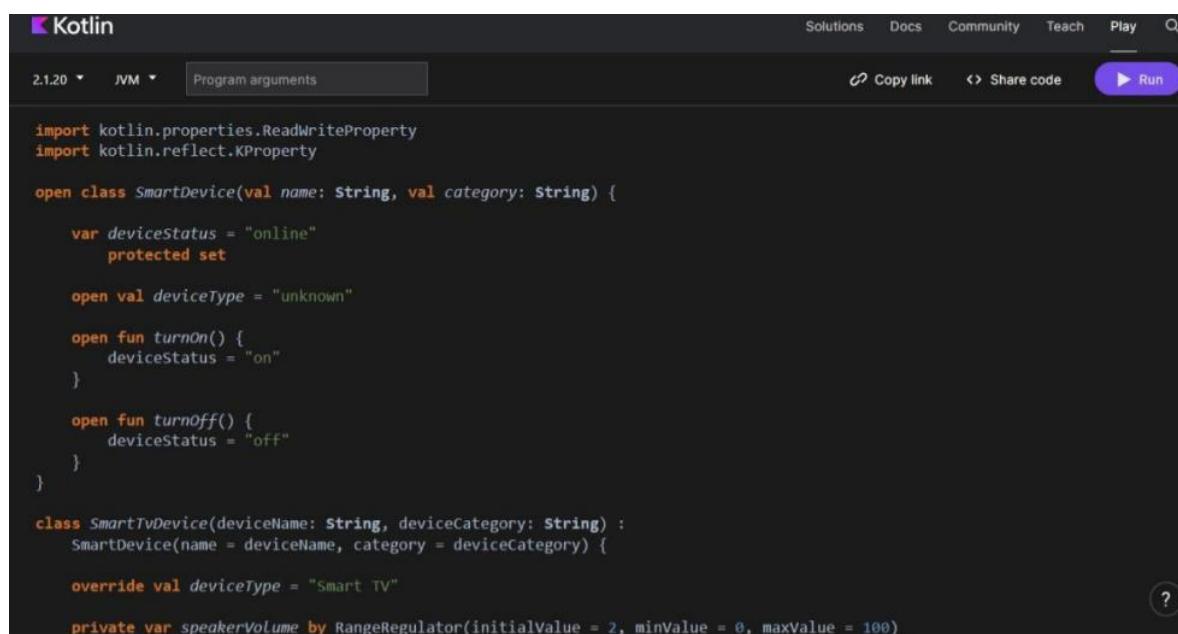
The output window shows the result: "The number of characters in your favorite actor's name is 9." with a close button.

Clases y objetos en Kotlin En Kotlin:

las clases se definen con la palabra clave “class” y contienen propiedades y métodos. Los constructores se usan para crear instancias de objetos, y la herencia permite la reutilización de código (relación IS-A). La composición (relación HAS-A) se usa para combinar objetos dentro de otros objetos. Los modificadores de

visibilidad (public, private, protected, internal) controlan el acceso a las propiedades y métodos. Además, los delegados de propiedades permiten reutilizar el código de los métodos "get" y "set".

La clase SmartTvDevice extiende SmartDevice para representar un televisor inteligente, sobrescribiendo deviceType a "Smart TV". Internamente, gestiona el speakerVolume y el channelNumber mediante la delegación de propiedades a la clase RangeRegulator, lo que asegura que estos valores se mantengan dentro de rangos específicos. Proporciona funciones para incrementar el volumen y cambiar de canal, y redefine los comportamientos de turnOn() y turnOff() para incluir acciones propias de un televisor, como informar del volumen y el canal al encenderse.



The screenshot shows the Kotlin playground interface with the following code:

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

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

    var deviceStatus = "online"
        protected set

    open val deviceType = "unknown"

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

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

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

    override val deviceType = "Smart TV"

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

The screenshot shows the Kotlin playground interface with the following code:

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

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

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

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

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

La clase SmartLightDevice, también heredando de SmartDevice, representa una luz inteligente y establece su deviceType como "Smart Light". Similar a la televisión, utiliza RangeRegulator para controlar el brightnessLevel dentro de un rango definido. Ofrece una función para aumentar el brillo y personaliza las funciones turnOn() y turnOff() para establecer niveles de brillo específicos al cambiar el estado de la luz.

The screenshot shows the Kotlin playground interface with the following code:

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

    override val deviceType = "Smart Light"

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

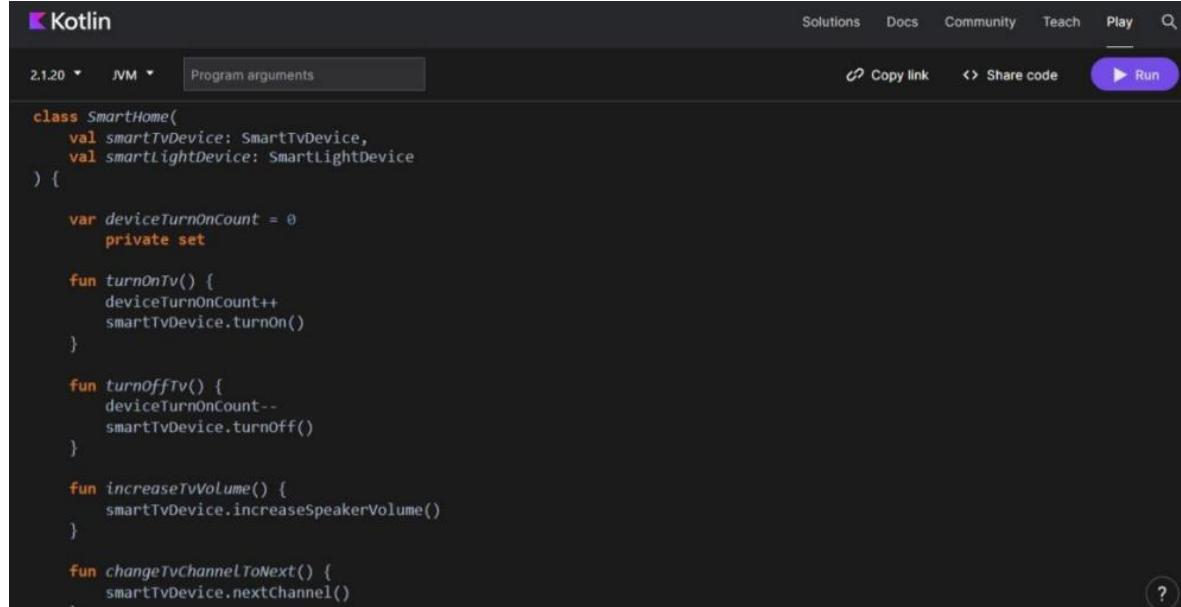
    fun increaseBrightness() {
        brightnessLevel++
        println("Brightness increased to $brightnessLevel.")
    }

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

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

La clase SmartHome actúa como un sistema de control centralizado, manteniendo instancias de SmartTvDevice y SmartLightDevice. Lleva un registro de la cantidad de dispositivos encendidos y ofrece métodos para interactuar con cada dispositivo

individualmente, como encender y apagar la televisión y la luz, ajustar el volumen y el brillo, y cambiar el canal. También incluye una función para apagar todos los dispositivos conectados.



The screenshot shows the Kotlin Play IDE interface with the following code:

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

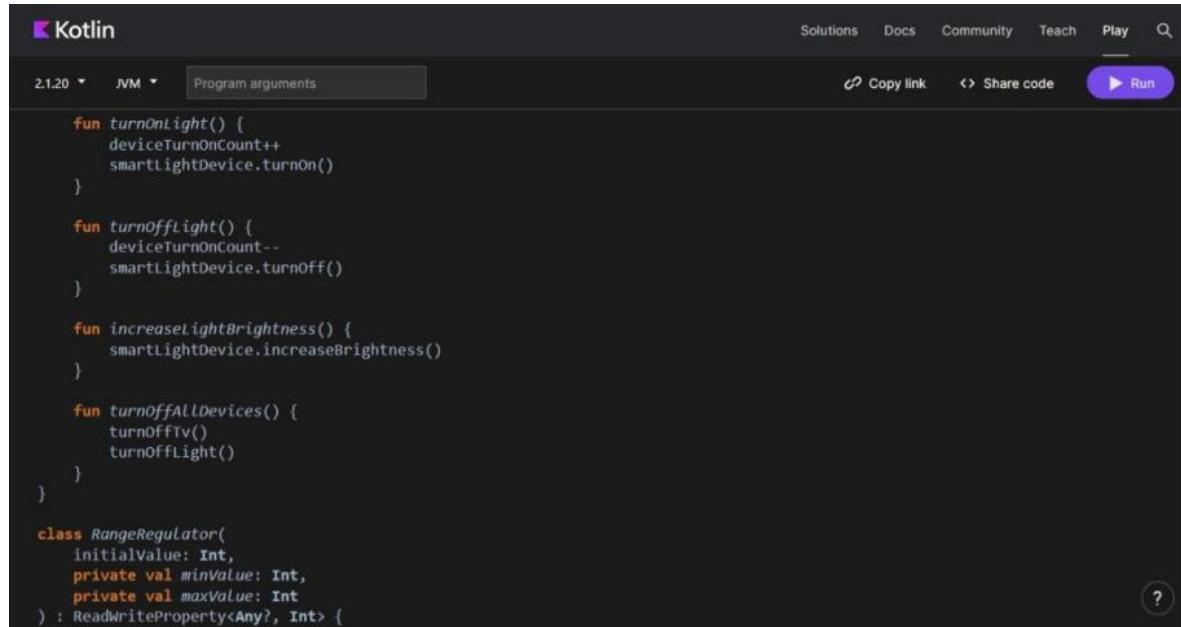
    fun turnOnTv() {
        deviceTurnOnCount++
        smartTvDevice.turnOn()
    }

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

    fun increaseTvVolume() {
        smartTvDevice.increaseSpeakerVolume()
    }

    fun changeTvChannelToNext() {
        smartTvDevice.nextChannel()
    }
}
```

Finalmente, la clase RangeRegulator implementa la interfaz ReadWriteProperty, actuando como un delegado de propiedad para gestionar valores dentro de un rango definido. Mantiene un valor interno (fieldData) y las funciones getValue() y setValue() definen cómo se accede y se modifica este valor, asegurando que cualquier nuevo valor propuesto se encuentre dentro de los límites minValue y maxValue antes de ser aceptado.



The screenshot shows the Kotlin Play IDE interface with the following code:

```
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> {
```

The screenshot shows the Kotlin playground interface. At the top, there's a navigation bar with links for Solutions, Docs, Community, Teach, Play, and a search icon. Below the navigation bar, there are dropdown menus for '2.1.20' and 'JVM', and a 'Program arguments' input field. On the right side of the editor area, there are buttons for 'Copy link', 'Share code', and 'Run'. The code itself is a Kotlin script:

```
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()
}
```

This screenshot shows the same playground interface after the code has been run. The output window at the bottom displays the results of the program execution:

```
fieldData = value
}
}
}

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.
```

Tipos de funciones y expresiones lambda en Kotlin:

En Kotlin, las expresiones lambda ofrecen una sintaxis abreviada para definir funciones, y pueden ser pasadas como parámetros o devueltas desde otras funciones. Cuando se utiliza un solo parámetro en una lambda, se puede hacer referencia a él con “i”. Además, las expresiones lambda pueden ir al final de una llamada de función si el último parámetro es una función, usando una sintaxis más concisa. Mientras tanto, las funciones de orden superior son aquellas que toman otras funciones como parámetros o las devuelven. Un ejemplo es la función “repeat()”, que actúa como un bucle “for”.

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/**  
 * You can edit, run, and share this code.  
 * play.kotlinlang.org  
 */  
fun main() {  
    val trickFunction = trick  
    trick()  
}  
  
val trick = {  
    println("No treats!")  
}
```

No treats!

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
/**  
 * You can edit, run, and share this code.  
 * play.kotlinlang.org  
 */  
fun main() {  
    val trickFunction = trick  
    trick()  
    trickFunction()  
}  
  
val trick = {  
    println("No treats!")  
}
```

No treats!
No treats!

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
* play.kotlinlang.org  
*/  
val trick = {  
    println("No treats!")  
}  
  
val treat: () -> Unit = {  
    println("Have a treat!")  
}  
  
fun main() {  
    val trickFunction = trick  
    trick()  
    trickFunction()  
    treat()  
}  
  
No treats!  
No treats!  
Have a treat!
```

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

* play.kotlinlang.org

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

fun trickOrTreat(isTrick: Boolean): () -> Unit {
    val trick = {
        println("No treats!")
    }

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

Have a treat!
No treats!
```

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

```
/*
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */
fun main() {
    val coins: (Int) -> String = { quantity ->
        "$quantity quarters"
    }

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

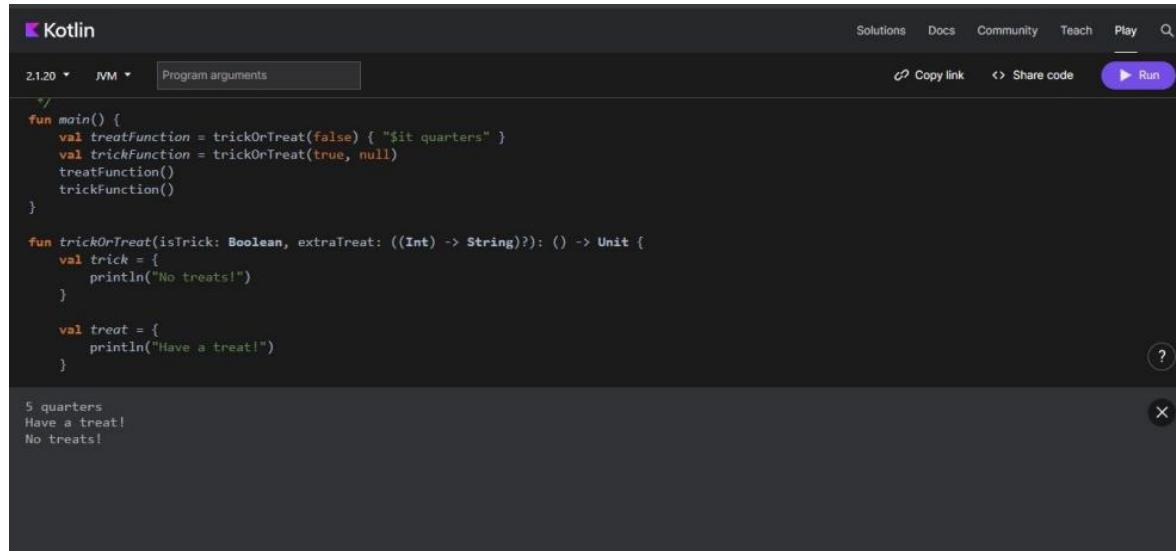
    val treatFunction = trickOrTreat(false, coins)
    val trickFunction = trickOrTreat(true, cupcake)
    treatFunction()
}

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

En este código siguiente se ilustra una forma más elegante de pasar funciones como argumentos, especialmente cuando se trata de expresiones lambda al final de la lista de parámetros. En la función main, la primera llamada a trickOrTreat demuestra esta sintaxis: la lambda { "\$it quarters" } se coloca fuera de los paréntesis de la llamada, actuando como el último argumento. Esta lambda toma un entero y produce una cadena que describe una cantidad de monedas. La segunda llamada a trickOrTreat pasa null para el argumento extraTreat, que ahora es un tipo de función nullable. Dentro de la función trickOrTreat, si isTrick es falso, se verifica que extraTreat no sea nulo antes de invocarlo con el valor 5, lo que en el primer caso resulta en la impresión de "5 quarters".

Finalmente, trickOrTreat devuelve una función que imprime "Have a treat!" o "No treats!" dependiendo del valor de isTrick, y estas funciones devueltas son luego ejecutadas en main, produciendo la salida "Have a treat!" y "No treats!". El principal

cambio aquí es la sintaxis concisa para la lambda final y el manejo explícito de un parámetro de función que puede ser nulo.



The screenshot shows the Kotlin Play IDE interface. At the top, there's a navigation bar with 'Solutions', 'Docs', 'Community', 'Teach', 'Play', and a search icon. Below the bar, it says '2.1.20' and 'JVM'. There's a 'Program arguments' input field. On the right, there are 'Copy link', 'Share code', and a 'Run' button. The main area contains the following Kotlin code:

```
/*
fun main() {
    val treatFunction = trickOrTreat(false) { "5 quarters" }
    val trickFunction = trickOrTreat(true, null)
    treatFunction()
    trickFunction()
}

fun trickOrTreat(isTrick: Boolean, extraTreat: ((Int) -> String)?): () -> Unit {
    val trick = {
        println("No treats!")
    }

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

Below the code, the output window displays the results of running the program:

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

También se introduce la función repeat(), donde en la función main, se definen treatFunction y trickFunction de manera similar al ejemplo anterior, con treatFunction configurada para imprimir "Have a treat!" después de imprimir "5 quarters" debido a la llamada inicial a trickOrTreat, y trickFunction configurada para imprimir "No treats!". La novedad radica en el uso de repeat(4) { treatFunction() }, que ejecuta la función referenciada por treatFunction cuatro veces consecutivas, resultando en la impresión repetida del mensaje "Have a treat!". Finalmente, se llama a trickFunction(), lo que produce la impresión de "No treats!". La función trickOrTreat en sí mantiene su lógica previa para determinar qué función devolver basándose en la condición isTrick y la presencia de extraTreat.

The screenshot shows the Kotlin Play IDE interface. The code in the editor is:

```
/*
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 {
    val trick = {
        println("No treats!")
    }
    val treat = {
        5 quarters
        Have a treat!
        Have a treat!
        Have a treat!
        Have a treat!
        No treats!
    }
    if (isTrick) trick
    else treat
}
```

The run output window shows the following output:

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

Práctica: Conceptos básicos de Kotlin

Aquí se cubren conceptos fundamentales como sentencias condicionales (“if/else” y “when”), funciones de orden superior, expresiones lambda, y manejo de nulos con operadores seguros (“?.” y “?:”).

The screenshot shows the Kotlin Play IDE interface. The code in the editor is:

```
fun main() {
    val morningNotification = 51
    val eveningNotification = 135

    printNotificationSummary(morningNotification)
    printNotificationSummary(eveningNotification)
}

fun printNotificationSummary(numberOfMessages: Int) {
    if (numberOfMessages < 100) {
        println("You have ${numberOfMessages} notifications.")
    } else {
        println("Your phone is blowing up! You have 99+ notifications.")
    }
}
```

The run output window shows the following output:

```
You have 51 notifications.
Your phone is blowing up! You have 99+ notifications.
```

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for Solutions, Docs, Community, Teach, Play, and a search bar. Below the tabs, it says "2.1.20" and "JVM". There is a "Program arguments" input field. On the right, there are buttons for "Copy link", "Share code", and "Run". The main area contains the following Kotlin code:

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

    val isMonday = true

    println("The movie ticket price for a person aged $child is $$${ticketPrice(child, isMonday)}")
    println("The movie ticket price for a person aged $adult is $$${ticketPrice(adult, isMonday)}")
    println("The movie ticket price for a person aged $senior is $$${ticketPrice(senior, isMonday)}")
}

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
    }
}
```

Below the code, the output of the program is displayed:

```
The movie ticket price for a person aged 5 is $15.
The movie ticket price for a person aged 28 is $25.
The movie ticket price for a person aged 87 is $20.
```

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for Solutions, Docs, Community, Teach, Play, and a search bar. Below the tabs, it says "2.1.20" and "JVM". There is a "Program arguments" input field. On the right, there are buttons for "Copy link", "Share code", and "Run". The main area contains the following Kotlin code:

```
fun main() {
    printFinalTemperature(27.0, "Celsius", "Fahrenheit") { 9.0 / 5.0 * it + 32 }
    printFinalTemperature(350.0, "Kelvin", "Celsius") { it - 273.15 }
    printFinalTemperature(10.0, "Fahrenheit", "Kelvin") { 5.0 / 9.0 * (it - 32) + 273.15 }
}

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

Below the code, the output of the program is displayed:

```
27.0 degrees Celsius is 80.60 degrees Fahrenheit.
350.0 degrees Kelvin is 76.85 degrees Celsius.
10.0 degrees Fahrenheit is 260.93 degrees Kelvin.
```

Se enseña el uso de clases con propiedades personalizadas y métodos, como en la clase “Song”, y cómo manejar la herencia con clases como “Phone” y “FoldablePhone”.

En el código se incluye una clase llamada Song que demuestra la creación de un tipo de dato personalizado con sus propias propiedades (title, artist, yearPublished, playCount) y métodos (printDescription). Además, introduce el concepto de una propiedad calculada (isPopular), cuyo valor no se almacena directamente, sino que se calcula cada vez que se accede a ella, basándose en el valor de playCount. Esto ilustra cómo las clases en Kotlin pueden encapsular datos (propiedades) y comportamientos (métodos y propiedades calculadas) relacionados, permitiendo

crear modelos más ricos y significativos para representar entidades del mundo real dentro del programa.

The screenshot shows the Kotlin Play IDE interface. The code in the editor is:

```
2.1.20 * JVM * Program arguments

fun main() {
    val brunoSong = Song("We Don't Talk About Bruno", "Encanto Cast", 2022, 1_000_000)
    brunoSong.printDescription()
    println(brunoSong.isPopular)
}

class Song(
    val title: String,
    val artist: String,
    val yearPublished: Int,
    val playCount: Int
) {
    val isPopular: Boolean
        get() = playCount >= 1000

    fun printDescription() {
        println("$title, performed by $artist, was released in $yearPublished.")
    }
}

We Don't Talk About Bruno, performed by Encanto Cast, was released in 2022.
true
```

The output window at the bottom shows the results of the run:

```
We Don't Talk About Bruno, performed by Encanto Cast, was released in 2022.
true
```

The screenshot shows the Kotlin Play IDE interface. The code in the editor is:

```
2.1.20 * JVM * Program arguments

fun main() {
    val amanda = Person("Amanda", 33, "play tennis", null)
    val atiqah = Person("Atiqah", 28, "climb", amanda)

    amanda.showProfile()
    atiqah.showProfile()
}

class Person(val name: String, val age: Int, val hobby: String?, val referrer: Person?) {
    fun showProfile() {
        println("Name: $name")
        println("Age: $age")
        if(hobby != null) {
            print("Likes to $hobby. ")
        }
        if(referrer != null) {
            print("Has a referrer named ${referrer.name}")
            if(referrer.hobby != null) {
                print(", who likes to ${referrer.hobby}. ")
            } else {
                print(".")
            }
        } else {
            print("Doesn't have a referrer.")
        }
        print("\n\n")
    }
}
```

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for Solutions, Docs, Community, Teach, and Play, along with a search bar. Below the tabs, there are dropdown menus for '2.1.20' and 'JVM', and a 'Program arguments' input field. On the right side, there are buttons for 'Copy link', 'Share code', and a purple 'Run' button.

```

fun main() {
    val amanda = Person("Amanda", 33, "play tennis", null)
    val atiqah = Person("Atiqah", 28, "climb", amanda)

    amanda.showProfile()
    atiqah.showProfile()
}

class Person(val name: String, val age: Int, val hobby: String?, val referrer: Person?) {
    fun showProfile() {
        println("Name: $name")
        println("Age: $age")
        if(hobby != null) {
            print("Likes to $hobby. ")
        }
        if(referrer != null) {
            print("Has a referrer named ${referrer.name}")
            if(referrer.hobby != null) {
                print("Who likes to ${referrer.hobby}. ")
            }
        }
    }
}

```

Output window:

```

Name: Amanda
Age: 33
Likes to play tennis. Doesn't have a referrer.

Name: Atiqah
Age: 28
Likes to climb. Has a referrer named Amanda, who likes to play tennis.

```

La clase Phone actúa como una clase base (marcada como open para permitir la herencia), definiendo propiedades y funciones comunes a los teléfonos, como el estado de la pantalla (isScreenLightOn) y las funciones para encenderla (switchOn), apagarla (switchOff) y verificar su estado (checkPhoneScreenLight). La clase FoldablePhone hereda de Phone, indicando una relación "es-un" (un FoldablePhone es un tipo de Phone). La clase FoldablePhone sobrescribe la función switchOn() para proporcionar una implementación específica para los teléfonos plegables: la pantalla solo se enciende si el teléfono no está plegado.

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for Solutions, Docs, Community, Teach, and Play, along with a search bar. Below the tabs, there are dropdown menus for '2.1.20' and 'JVM', and a 'Program arguments' input field. On the right side, there are buttons for 'Copy link', 'Share code', and a purple 'Run' button.

```

open class Phone(var isScreenLightOn: Boolean = false){
    open fun switchOn() {
        isScreenLightOn = true
    }

    fun switchOff() {
        isScreenLightOn = false
    }

    fun checkPhoneScreenLight() {
        val phoneScreenLight = if (isScreenLightOn) "on" else "off"
        println("The phone screen's light is $phoneScreenLight.")
    }
}

class FoldablePhone(var isFolded: Boolean = true): Phone() {
    override fun switchOn() {
        if (!isFolded) {
            isScreenLightOn = true
        }
    }
}

```

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
fun fold() {
    isFolded = true
}

fun unfold() {
    isFolded = false
}

fun main() {
    val newFoldablePhone = FoldablePhone()

    newFoldablePhone.switchOn()
    newFoldablePhone.checkPhoneScreenLight()
    newFoldablePhone.unfold()
    newFoldablePhone.switchOn()
    newFoldablePhone.checkPhoneScreenLight()
}
```

The phone screen's light is off.
The phone screen's light is on.

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
fun main() {
    val winningBid = Bid(5000, "Private Collector")

    println("Item A is sold at ${auctionPrice(winningBid, 2000)}")
    println("Item B is sold at ${auctionPrice(null, 3000)}")
}

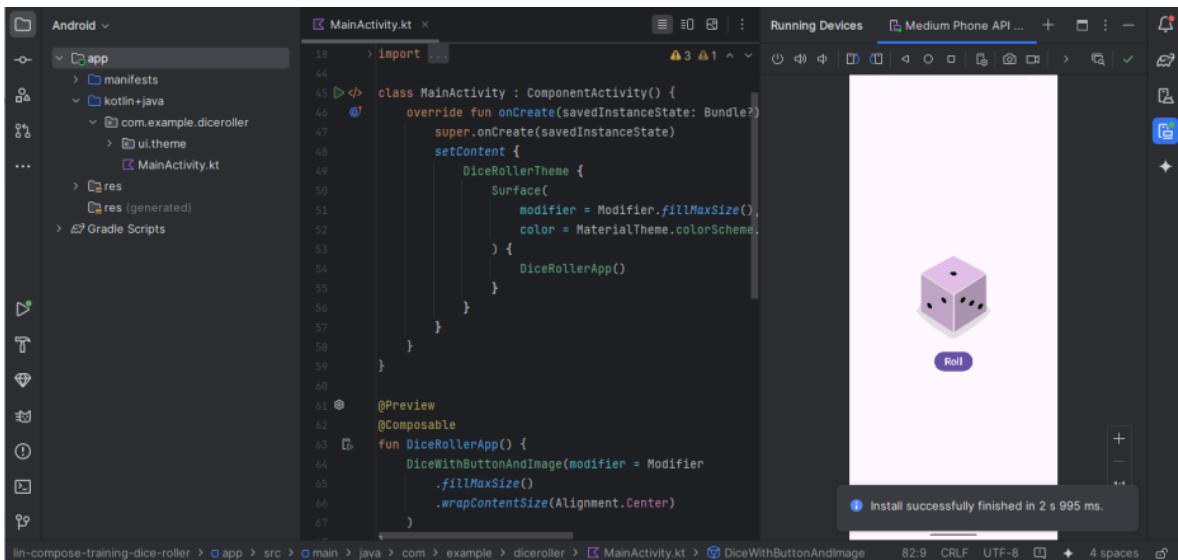
class Bid(val amount: Int, val bidder: String)

fun auctionPrice(bid: Bid?, minimumPrice: Int): Int {
    return bid?.amount ?: minimumPrice
}
```

Item A is sold at 5000.
Item B is sold at 3000.

Ruta de Aprendizaje 2 (Agrega un Botón a una app)

Crear App interactiva de Dice Roller En este codelab se crea una app interactiva de Dice Roller usando Jetpack Compose. Se definen funciones componibles, interfaces con composiciones, y elementos como “Button” e “Image”. Además, se nos enseña a importar recursos gráficos, hacer la interfaz interactiva con “remember” y “mutableStateOf()” para almacenar y observar el estado, permitiendo que la IU se actualice dinámicamente al lanzar el dado



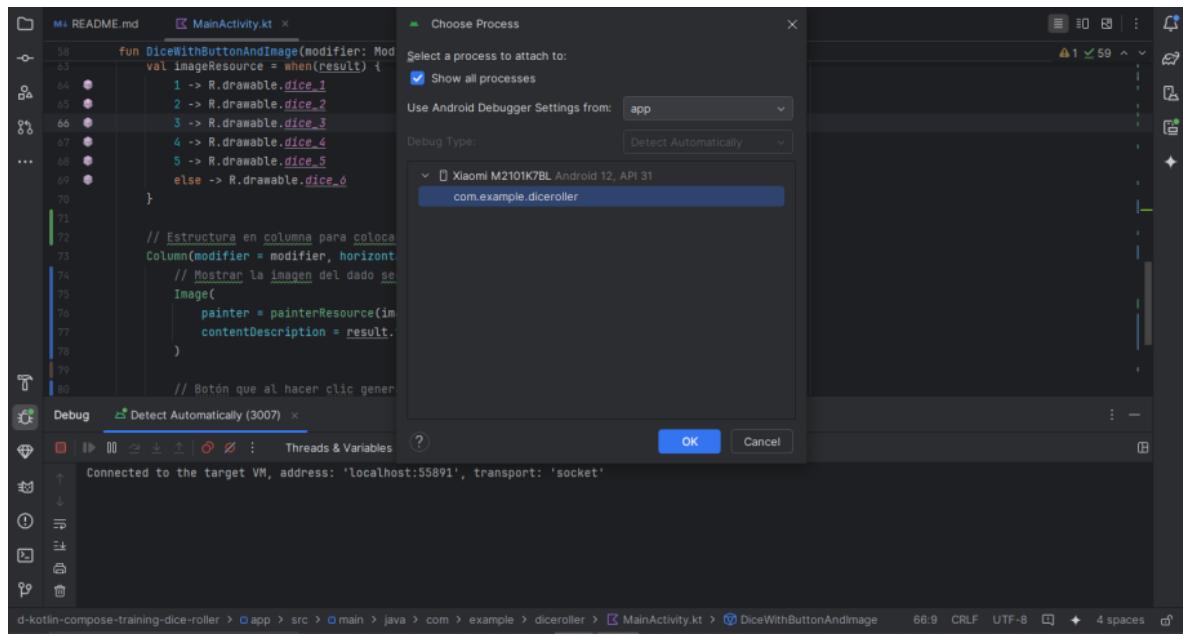
En el archivo principal (MainActivity.kt), se define una función composable llamada DiceRollerApp(), que estructura la interfaz de la app. Dentro de ella, se utiliza remember { mutableStateOf(1) } para almacenar y observar el número del dado que ha salido, de forma que cada vez que cambia, Compose actualiza automáticamente la imagen mostrada. Se usan elementos composables como Image para mostrar la cara del dado (imagen ubicada en res/drawable), y Button para permitir al usuario lanzar el dado. Al presionar el botón, se genera un número aleatorio entre 1 y 6 con Random.nextInt(1, 7), y se actualiza el estado.

Use the debugger in Android Studio

En este codelab, aprendimos a usar el depurador en Android Studio para inspeccionar lo que sucede en la app de Dice Roller durante el tiempo de ejecución. El depurador es una herramienta esencial que te permite inspeccionar la ejecución del código que potencia tu app para Android, de modo que puedas corregir cualquier error que aparezca. Te permite especificar puntos en los cuales suspender la ejecución del código e interactuar manualmente con variables, métodos y otros aspectos del código. Existen dos maneras de ejecutar el depurador:

- Adjunta el depurador a un proceso de la app existente que se ejecuta en un dispositivo o emulador.
- Ejecuta la app con el depurador.

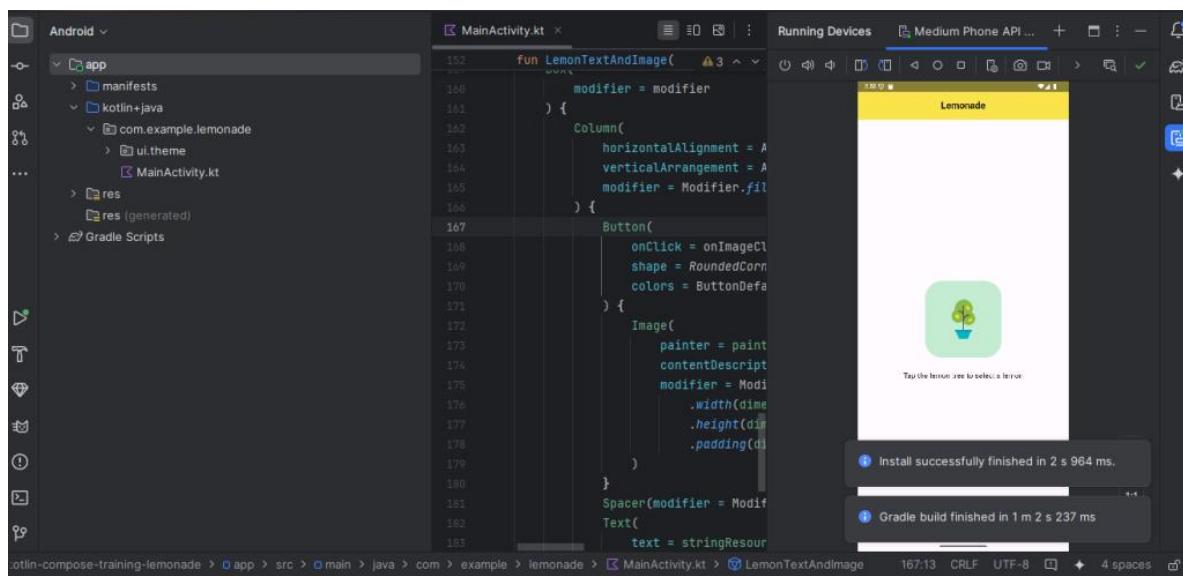
Para adjuntar el depurador a un proceso de la app, seguimos los siguientes pasos:
 1. Haz clic en Attach debugger to Android process. Se abrirá el diálogo Choose Process, en el que podrás elegir el proceso al que deseas adjuntar el depurador.
 2. Selecciona com.example.diceroller y haz clic en OK.



En la parte inferior de Android Studio, aparecerá un panel Debug con un mensaje que indica que el depurador está conectado al dispositivo de destino o al emulador. Habremos adjuntado el depurador a la aplicación con éxito.

Comportamiento de clics (Practice: Click behavior)

Este codelab nos enseña a construir una interfaz interactiva con Jetpack Compose guiando al usuario por cuatro pasos que simulan hacer limonada. Usa estado mutable (“remember” y “mutableStateOf”) para cambiar imagen y texto según el paso: seleccionar un limón, exprimirlo varias veces, tomar limonada y reiniciar. Se usan componentes como “Column”, “Image”, “Text” y “Modifier.clickable” para crear una UI centrada y dinámica.



La app guía al usuario por cuatro etapas visuales: primero ve un limonero, luego un limón que debe exprimir (presionando varias veces), seguido de un vaso de limonada listo para beber, y finalmente un vaso vacío. Tras presionar el vaso vacío, el ciclo vuelve a comenzar.

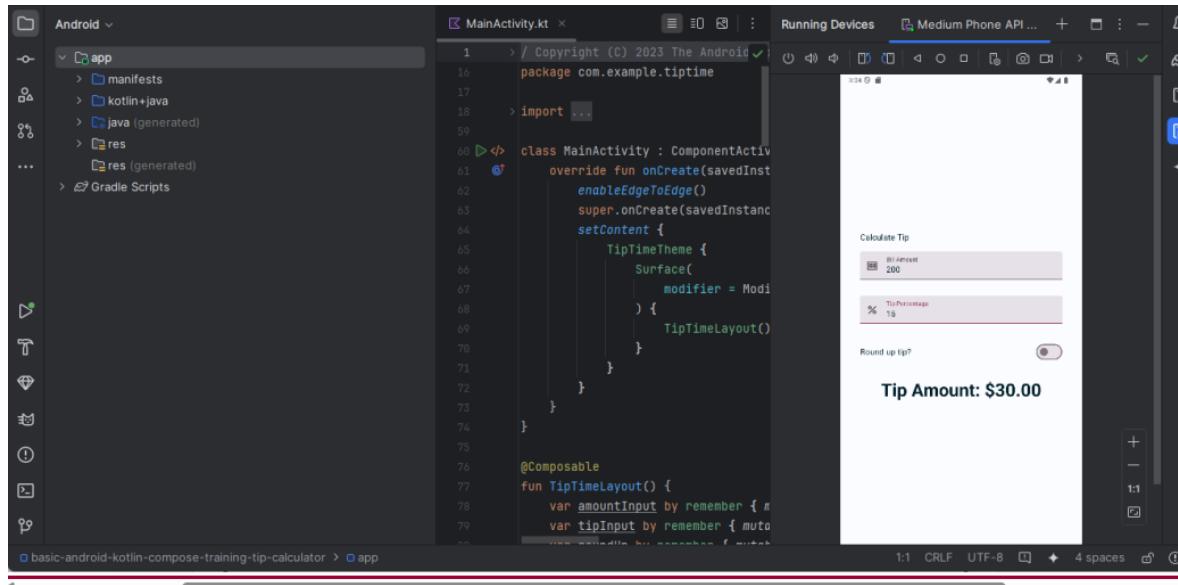
El funcionamiento se basa en un estado mutable (currentStep) que determina qué imagen y texto mostrar. También se usa una variable adicional (squeezeCount) que se inicializa aleatoriamente entre 2 y 4 cuando se selecciona el limón, y decrece con cada clic hasta que el usuario puede avanzar al siguiente paso. La lógica está implementada usando remember y mutableStateOf para que la interfaz se actualice automáticamente al cambiar el estado.

Ruta de Aprendizaje 3 (Interactúa con la IU y el estado)

Introducción al estado en Compose - Calcula una propina personalizada

En este codelab se permite calcular propinas iniciando con una implementación básica donde se ingresa el monto de la factura mediante un campo de texto y la app calcula automáticamente una propina del 15%, mostrando el resultado en pantalla con un componente Text. Sin embargo, cabe mencionar que esta funcionalidad se extiende en una segunda lección para ofrecer una experiencia más completa en la cual se agrega un segundo campo de texto para que el usuario introduzca un porcentaje de propina personalizado, y un interruptor (Switch) que permite redondear el resultado al número entero más cercano. Todos estos componentes

El estado de cada campo (monto de la factura, porcentaje de propina y opción de redondeo) se gestiona con rememberSaveable para asegurar que los datos se conserven ante posibles recomposiciones o cambios de configuración. La función calculateTip() contiene la lógica de cálculo y aplica redondeo si se selecciona, devolviendo el resultado en formato de moneda mediante NumberFormat.getCurrencyInstance().format(). Para mejorar la experiencia del usuario, los campos de texto están optimizados para entrada numérica y etiquetados correctamente con stringResource, y los textos visibles se almacenan en strings.xml para facilitar la localización y el mantenimiento.

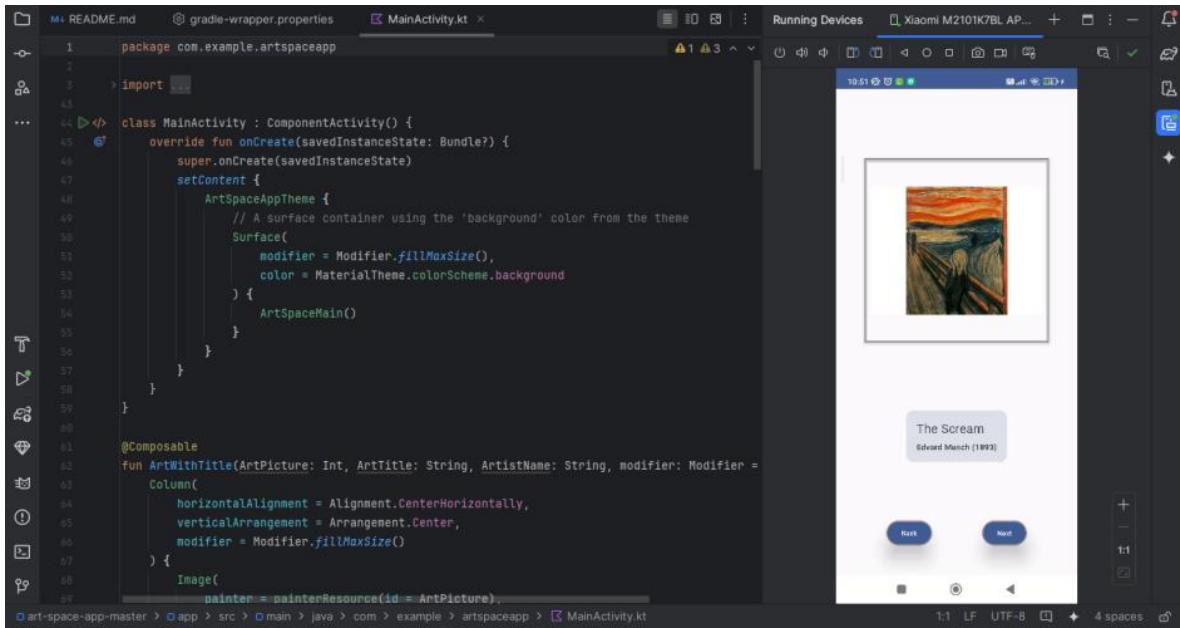


Proyecto: Crea una app de Art Space

Creamos la aplicación para que sea de forma interactiva y dinámica para los usuarios, implementando dos botones para que los usuarios interactúen. Mostrando la obra de arte siguiente o anterior al presionar un botón. Trabajamos en la parte de la IU que muestra la obra de arte siguiente o anterior cuando se presiona un botón:

1. Primero, identificamos los elementos de la IU que deben cambiar según la interacción del usuario. En este caso, los elementos de la IU son la imagen de la obra de arte, el título, el artista y el año.
2. Crea un estado para cada uno de los elementos dinámicos de la IU con el objeto `MutableState`.
3. Reemplazamos los valores codificados por `states` definidos.

Cuando los usuarios presionan el botón Next, deberían ver la siguiente obra de arte en la secuencia. Debido a que no tenemos una cantidad infinita de obras de arte, determinamos el comportamiento del botón Next para cuando se muestre la última obra de la serie para que vuelva a mostrar la primera obra de arte después de la última. Utilizando la sentencia `when` para compilar la lógica condicional, en lugar de las sentencias `if else`, para que el código sea más legible debido a la cantidad de casos de obras de arte.



The screenshot shows the Android Studio interface with the code editor open to `MainActivity.kt`. The code uses Jetpack Compose to display a painting. The preview window shows a painting of 'The Scream' by Edvard Munch. A tooltip below the painting identifies it as 'The Scream' by Edvard Munch (1893). The bottom right corner of the preview shows navigation buttons for 'Next' and 'Prev'.

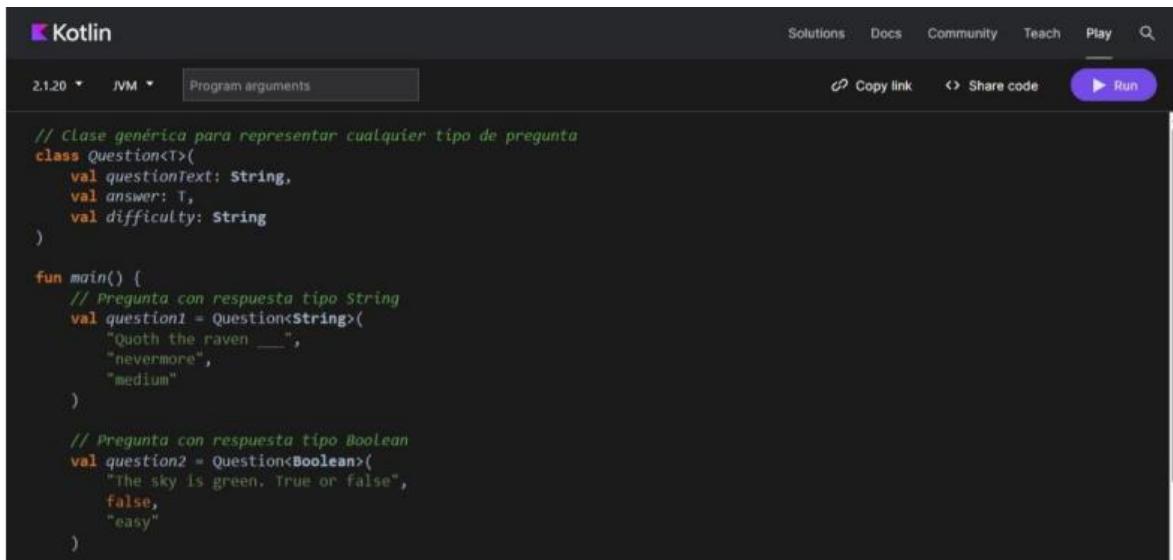
```
1 package com.example.artspaceapp
2
3 import ...
4
5 class MainActivity : ComponentActivity() {
6     override fun onCreate(savedInstanceState: Bundle?) {
7         super.onCreate(savedInstanceState)
8         setContent {
9             ArtSpaceAppTheme {
10                 // A surface container using the 'background' color from the theme
11                 Surface(
12                     modifier = Modifier.fillMaxSize(),
13                     color = MaterialTheme.colorScheme.background
14                 ) {
15                     ArtSpaceMain()
16                 }
17             }
18         }
19     }
20 }
21
22 @Composable
23 fun ArtWithTitle(ArtPicture: Int, ArtTitle: String, ArtistName: String, modifier: Modifier =
24     Column(
25         horizontalAlignment = Alignment.CenterHorizontally,
26         verticalArrangement = Arrangement.Center,
27         modifier = Modifier.fillMaxSize()
28     ) {
29     Image(
30         painter = painterResource(id = ArtPicture),
```

Unidad 3: Cómo mostrar listas y usar Material Design

Ruta de Aprendizaje 1

Parámetros genéricos, objetos y extensiones

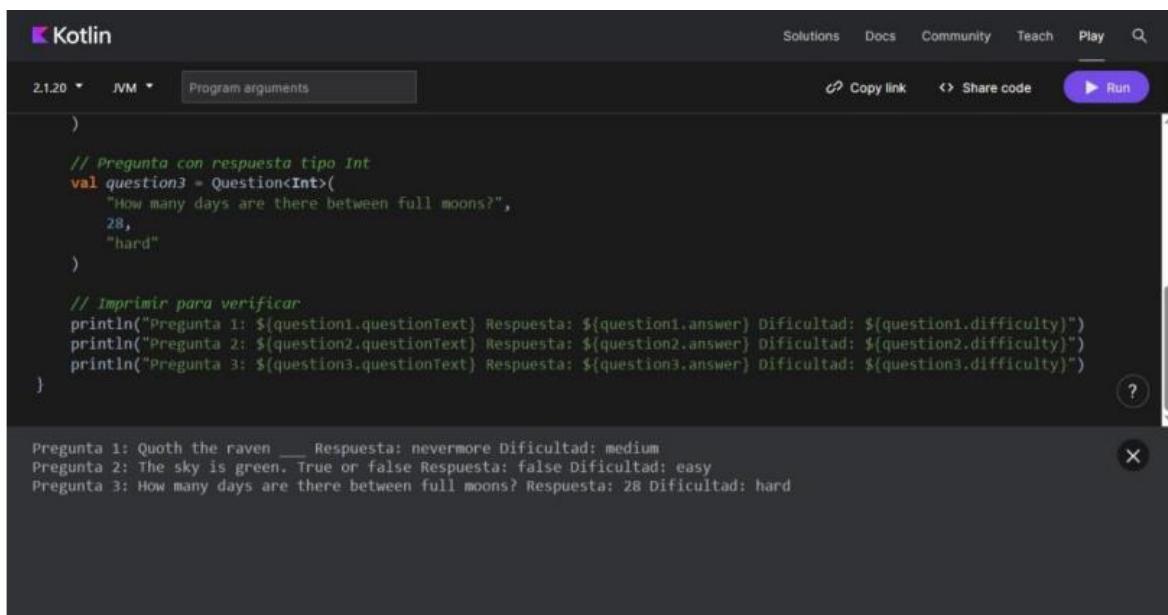
En este codelab se abordan tres conceptos clave de Kotlin que son fundamentales para trabajar con Jetpack Compose y otros componentes de la interfaz de usuario. Los parámetros genéricos permiten crear clases y funciones reutilizables que pueden trabajar con diferentes tipos de datos, como listas o componentes de UI, sin necesidad de especificar un tipo concreto. Los objetos, por su parte, son instancias únicas de una clase que se utilizan para almacenar datos globales o funciones estáticas, como las instancias singleton. Finalmente, las extensiones permiten agregar nuevas funcionalidades a clases existentes sin modificar su código original, lo que facilita la personalización de los componentes de la UI en Jetpack Compose. Estos conceptos mejoran la flexibilidad, la reutilización y la modularidad del código, permitiendo crear aplicaciones más escalables y fáciles de mantener. A continuación, podemos ver la creación y el uso de una clase genérica `Question` para modelar preguntas de un cuestionario donde el tipo de la respuesta puede variar. En lugar de definir clases separadas para preguntas con respuestas de tipo `String`, `Boolean`, o `Int`, se utiliza una única clase genérica `Question` que utiliza un marcador de tipo `T` para la propiedad `answer`.



```
// Clase genérica para representar cualquier tipo de pregunta
class Question<T>(
    val questionText: String,
    val answer: T,
    val difficulty: String
)

fun main() {
    // Pregunta con respuesta tipo String
    val question1 = Question<String>(
        "Quoth the raven ___",
        "nevermore",
        "medium"
    )

    // Pregunta con respuesta tipo Boolean
    val question2 = Question<Boolean>(
        "The sky is green. True or false",
        false,
        "easy"
    )
}
```



```
)
```

```
// Pregunta con respuesta tipo Int
val question3 = Question<Int>(
    "How many days are there between full moons?",
    28,
    "hard"
)

// Imprimir para verificar
println("Pregunta 1: ${question1.questionText} Respuesta: ${question1.answer} Dificultad: ${question1.difficulty}")
println("Pregunta 2: ${question2.questionText} Respuesta: ${question2.answer} Dificultad: ${question2.difficulty}")
println("Pregunta 3: ${question3.questionText} Respuesta: ${question3.answer} Dificultad: ${question3.difficulty}")
```

```
Pregunta 1: Quoth the raven ___ Respuesta: nevermore Dificultad: medium
Pregunta 2: The sky is green. True or false Respuesta: false Dificultad: easy
Pregunta 3: How many days are there between full moons? Respuesta: 28 Dificultad: hard
```

También se representa un conjunto limitado de valores, en este caso, los niveles de dificultad de las preguntas de un cuestionario. Primero, se define una clase enum Difficulty con tres posibles valores: EASY, MEDIUM, y HARD. Luego, en la clase genérica Question, el tipo de la propiedad difficulty se modifica de String a la nueva clase enum Difficulty. Finalmente, al crear instancias de Question en la función main(), se utilizan las constantes enum (como Difficulty.MEDIUM, Difficulty.EASY, y Difficulty.HARD) para asignar la dificultad a cada pregunta.

```
// Enum para representar niveles de dificultad
enum class Difficulty {
    EASY, MEDIUM, HARD
}

// Clase genérica para representar preguntas
class Question<T>{
    val questionText: String,
    val answer: T,
    val difficulty: Difficulty
}

fun main() {
    // Pregunta con respuesta tipo String
    val question1 = Question<String>(
        "Quoth the raven ___",
        "nevermore",
        Difficulty.MEDIUM
    )

    // Pregunta con respuesta tipo Boolean
    val question2 = Question<Boolean>(
        "The sky is green. True or false",
        false,
        Difficulty.EASY
    )
}
```

```
)
```

```
// Pregunta con respuesta tipo Int
val question3 = Question<Int>(
    "How many days are there between full moons?",
    28,
    Difficulty.HARD
)

// Imprimir para verificar
println("Pregunta 1: ${question1.questionText} Respuesta: ${question1.answer} Dificultad: ${question1.difficulty}")
println("Pregunta 2: ${question2.questionText} Respuesta: ${question2.answer} Dificultad: ${question2.difficulty}")
println("Pregunta 3: ${question3.questionText} Respuesta: ${question3.answer} Dificultad: ${question3.difficulty}")
```

```
Pregunta 1: Quoth the raven ___ Respuesta: nevermore Dificultad: MEDIUM
Pregunta 2: The sky is green. True or false Respuesta: false Dificultad: EASY
Pregunta 3: How many days are there between full moons? Respuesta: 28 Dificultad: HARD
```

A continuación, se muestra cómo al llamar a `toString()` en una clase regular (`Question` sin el prefijo `data`) donde la salida es simplemente el nombre de la clase y una referencia de memoria. Luego, se introduce la palabra clave `data` antes de la definición de la clase `Question`. Al convertir `Question` en una `data class`, el compilador de Kotlin genera automáticamente implementaciones para varios métodos útiles, incluyendo `equals()`, `hashCode()`, `toString()`, `componentN()` y `copy()`. La salida después de la modificación muestra que `toString()` ahora imprime una representación legible de las propiedades de la instancia de la clase `Question`.

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
// Enum para representar niveles de dificultad
enum class Difficulty {
    EASY, MEDIUM, HARD
}

// Clase de datos genérica para representar preguntas
data class Question<T>(
    val questionText: String,
    val answer: T,
    val difficulty: Difficulty
)

fun main() {
    // Preguntas con diferentes tipos de respuesta
    val question1 = Question<String>(
        "Quoth the raven ___",
        "nevermore",
        Difficulty.MEDIUM
    )

    val question2 = Question<Boolean>(
        "The sky is green. True or false",
        false,
        Difficulty.EASY
    )
}
```

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
Difficulty.EASY
)

val question3 = Question<Int>(
    "How many days are there between full moons?",
    28,
    Difficulty.HARD
)

// Imprime usando toString() automáticamente implementado por data class
println(question1.toString())
println(question2) // También se puede imprimir directamente
println(question3)
}
```

Question(questionText=Quoth the raven ___, answer=nevermore, difficulty=MEDIUM)
Question(questionText=The sky is green. True or false, answer=false, difficulty=EASY)
Question(questionText=How many days are there between full moons?, answer=28, difficulty=HARD)

Se nos enseña también a crear objetos singleton usando la palabra clave `object`, asegurando que solo exista una instancia de la clase. Estos objetos se utilizan para gestionar estados únicos o servicios. Además, se pueden declarar objetos complementarios dentro de una clase usando `companion object`. Esto asocia el singleton con la clase, permitiendo acceder a sus propiedades directamente a través del nombre de la clase, como si fueran miembros estáticos.

Kotlin

Solutions Docs Community Teach Play

2.1.20 ▾ JVM ▾ Program arguments

Copy link Share code Run

```
// Enum para las dificultades de las preguntas
enum class Difficulty {
    EASY, MEDIUM, HARD
}

// Clase de datos para representar preguntas genéricas
data class Question<T>(
    val questionText: String,
    val answer: T,
    val difficulty: Difficulty
)

// Clase que representa el cuestionario
class Quiz {
    // Preguntas del cuestionario
    val question1 = Question<String>("Quoth the raven __", "nevermore", Difficulty.MEDIUM)
    val question2 = Question<Boolean>("The sky is green. True or false", false, Difficulty.EASY)
    val question3 = Question<Int>("How many days are there between full moons?", 28, Difficulty.HARD)
}
```

Kotlin

Solutions Docs Community Teach Play

2.1.20 ▾ JVM ▾ Program arguments

Copy link Share code Run

```
// Objeto complementario para el progreso del estudiante
companion object StudentProgress {
    var total: Int = 10
    var answered: Int = 3
}

// Función principal
fun main() {
    println("${Quiz.answered} of ${Quiz.total} answered.")
}
```

3 of 10 answered.

También se nos muestra cómo se permite extender clases agregando propiedades y funciones de extensión sin modificar el código original. Se define una propiedad de extensión especificando el tipo a extender seguido de un punto y el nombre de la propiedad (con un getter). De manera similar, una función de extensión se define especificando el tipo a extender seguido de un punto y el nombre de la función.

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
// Enum para las dificultades de las preguntas
enum class Difficulty {
    EASY, MEDIUM, HARD
}

// Clase de datos para representar preguntas genéricas
data class Question<T>{
    val questionText: String,
    val answer: T,
    val difficulty: Difficulty
}

// Clase que representa el cuestionario
class Quiz {
    val question1 = Question<String>("Quoth the raven __", "nevermore", Difficulty.MEDIUM)
    val question2 = Question<Boolean>("The sky is green. True or false", false, Difficulty.EASY)
    val question3 = Question<Int>("How many days are there between full moons?", 28, Difficulty.HARD)
}

// Objeto complementario para el progreso del estudiante
companion object StudentProgress {
    var total: Int = 10
    var answered: Int = 3
}
```

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
// Propiedad de extensión: progreso en texto
val Quiz.StudentProgress.progressText: String
    get() = "$answered of $total answered"

// Función de extensión: imprime una barra de progreso
fun Quiz.StudentProgress.printProgressBar() {
    repeat(answered) { print("█") }
    repeat(total - answered) { print("█") }
    println()
    println(progressText)
}

// Función principal
fun main() {
    Quiz.printProgressBar()
}
```

3 of 10 answered

Kotlin permite definir interfaces (interface) que especifican propiedades y funciones que las clases deben implementar (:) y eso se muestra en el codelab. Se crea un contrato asegurando que las clases que implementan la interfaz proporcionen las funcionalidades definidas. En el ejemplo, se crea una interfaz ProgressPrintable con progressText y printProgressBar().

Las interfaces ofrecen una forma estructurada y modular de compartir funcionalidades entre clases, a diferencia de las extensiones que agregan funcionalidades externamente.

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
// Enum para la dificultad de las preguntas
enum class Difficulty {
    EASY, MEDIUM, HARD
}

// Clase genérica para representar preguntas
data class Question<T>(
    val questionText: String,
    val answer: T,
    val difficulty: Difficulty
)

// INTERFAZ para imprimir progreso
interface ProgressPrintable {
    val progressText: String
    fun printProgressBar()
}

// Clase Quiz que ahora IMPLEMENTA ProgressPrintable
class Quiz : ProgressPrintable {

    val question1 = Question<String>("Quoth the raven ___, \"nevermore\"", Difficulty.MEDIUM)
    val question2 = Question<Boolean>("The sky is green. True or false", false, Difficulty.EASY)
    val question3 = Question<Int>("How many days are there between full moons?", 28, Difficulty.HARD)

    // Objeto complementario que lleva el progreso
    companion object StudentProgress {
        var total: Int = 10
        var answered: Int = 3
    }
}
```

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
// Implementación de la propiedad progressText (obligatoria por la interfaz)
override val progressText: String
    get() = "${answered} of ${total} answered"

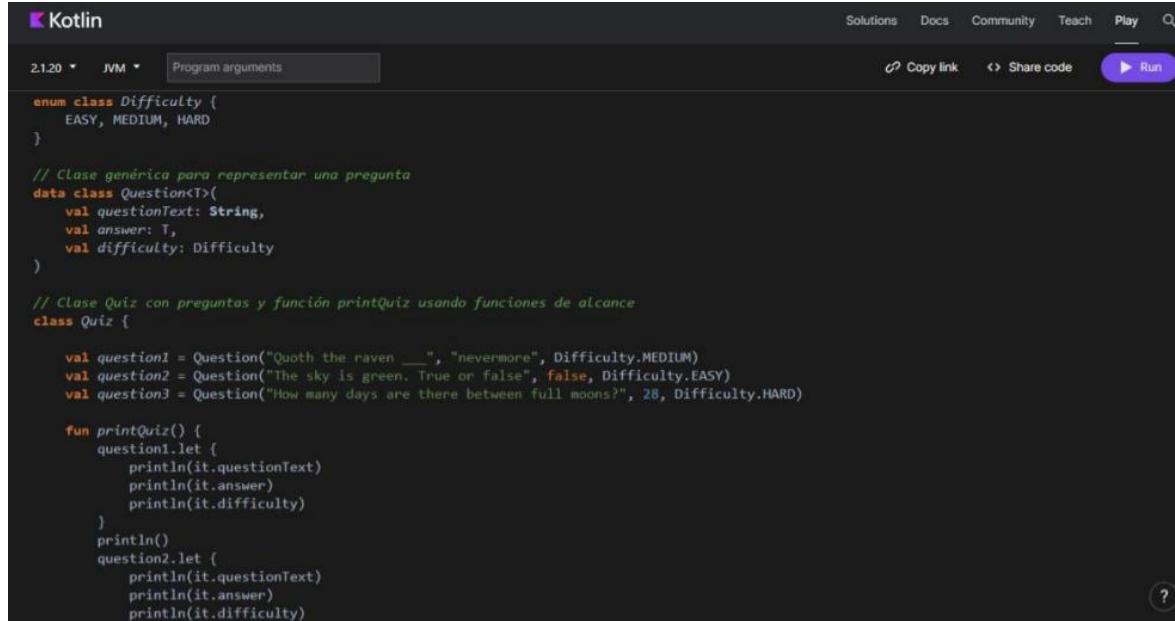
// Implementación de la función printProgressBar() (también requerida)
override fun printProgressBar() {
    repeat(answered) { print("#") }
    repeat(total - answered) { print("-") }
    println()
    println(progressText)
}

// Función principal
fun main() {
    Quiz().printProgressBar()
}
```

3 of 10 answered

Por último se nos habla acerca de las funciones de alcance en Kotlin que permiten acceder a propiedades y métodos de un objeto de forma concisa dentro de una lambda, sin repetir el nombre del objeto. Se explica la función let(), que permite referirse al objeto dentro de la lambda usando el identificador it. En el ejemplo, se agrega una función printQuiz() a la clase Quiz que originalmente accedía a las propiedades de question1, question2 y question3 usando sus nombres completos. Luego, se refactoriza esta función para usar let() en cada pregunta, reemplazando el nombre de la variable por it dentro de la lambda. También se explica la función apply(), que permite llamar métodos en un objeto incluso antes de asignarlo a una variable. La lambda dentro de apply() tiene el objeto como receptor (this), aunque en este ejemplo no se usa explícitamente. Se muestra cómo crear una instancia de

Quiz y llamar a printQuiz() dentro de un bloque apply(), incluso omitiendo la creación de una variable para la instancia de Quiz.



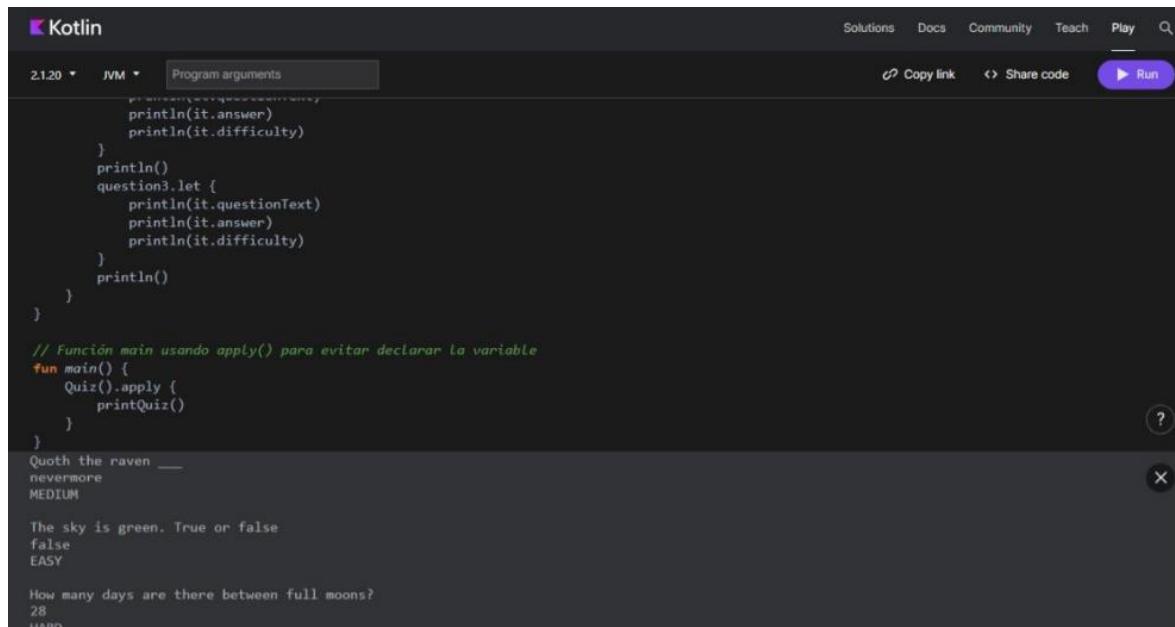
```
enum class Difficulty {
    EASY, MEDIUM, HARD
}

// Clase genérica para representar una pregunta
data class Question<T>{
    val questionText: String,
    val answer: T,
    val difficulty: Difficulty
}

// Clase Quiz con preguntas y función printQuiz usando funciones de alcance
class Quiz {

    val question1 = Question("Quoth the raven ___", "nevermore", Difficulty.MEDIUM)
    val question2 = Question("The sky is green. True or False", false, Difficulty.EASY)
    val question3 = Question("How many days are there between full moons?", 28, Difficulty.HARD)

    fun printQuiz() {
        question1.let {
            println(it.questionText)
            println(it.answer)
            println(it.difficulty)
        }
        println()
        question2.let {
            println(it.questionText)
            println(it.answer)
            println(it.difficulty)
        }
        println()
    }
}
```



```
// Función main usando apply() para evitar declarar la variable
fun main() {
    Quiz().apply {
        printQuiz()
    }
}

```

Quoth the raven ___
nevermore
MEDIUM

The sky is green. True or false
false
EASY

How many days are there between full moons?
28
HARD

Usa colecciones en Kotlin

Se resumen las características principales de los arrays, que almacenan datos ordenados del mismo tipo con un tamaño fijo, y cómo sirven de base para otras colecciones. Las listas se describen como colecciones ordenadas de tamaño variable, mientras que los conjuntos son colecciones no ordenadas que no admiten duplicados. Finalmente, los mapas se presentan como estructuras que almacenan pares de clave-valor, similares a los conjuntos en la unicidad de sus claves.

The screenshot shows the Kotlin Play IDE interface. The code in the editor is:

```
/*
 * Unidad 3
 */
fun main() {
    // Creamos un array de planetas rocosos
    val rockPlanets = arrayOf("Mercury", "Venus", "Earth", "Mars")

    // Creamos un array de planetas gaseosos (sin especificar tipo explícito)
    val gasPlanets = arrayOf("Jupiter", "Saturn", "Uranus", "Neptune")

    // Unimos los dos arrays en uno nuevo llamado solarSystem
    val solarSystem = rockPlanets + gasPlanets
}
```

The screenshot shows the Kotlin Play IDE interface. The code in the editor is:

```
/*
 * Unidad 3
 */
fun main() {
    // Creamos arrays como listas mutables
    val solarSystem = mutableListOf("Mercury", "Venus", "Earth", "Mars",
        "Jupiter", "Saturn", "Uranus", "Neptune")

    // Imprimimos todos los elementos
    println(solarSystem[0])
    println(solarSystem[1])
    println(solarSystem[2])
    println(solarSystem[3])
    println(solarSystem[4])
    println(solarSystem[5])
    println(solarSystem[6])
}
```

Below the code, the terminal window shows the output:

```
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
```

En este ejemplo que se muestra a continuación, dentro de la función main, se declara una variable inmutable llamada solarSystem a la cual se le asigna una lista inmutable que contiene los nombres de los ocho planetas del sistema solar, creada mediante la función listOf(). Posteriormente, se imprime en la consola el tamaño de esta lista utilizando la propiedad size. La salida del programa muestra el número 8, lo que indica que la lista solarSystem contiene ocho elementos, correspondientes a los ocho planetas definidos al crear la lista inmutable.

Kotlin

Solutions Docs Community Teach Play

2.1.20 ▾ JVM ▾ Program arguments

Copy link Share code Run

```
/*
 * Unidad 3
 */
fun main() {
    // Creamos una lista inmutable con los 8 planetas
    val solarSystem = listOf("Mercury", "Venus", "Earth", "Mars",
        "Jupiter", "Saturn", "Uranus", "Neptune")

    // Imprimimos el tamaño de la lista
    println(solarSystem.size)
}
```

8

Kotlin

Solutions Docs Community Teach Play

2.1.20 ▾ JVM ▾ Program arguments

Copy link Share code Run

```
/*
 * Unidad 3
 */
fun main() {
    // Creamos una lista inmutable con los 8 planetas
    val solarSystem = listOf("Mercury", "Venus", "Earth", "Mars",
        "Jupiter", "Saturn", "Uranus", "Neptune")

    // Accedemos al elemento en el índice 2 usando subíndice
    println(solarSystem[2]) // Debería imprimir: Earth

    // Accedemos al elemento en el índice 3 usando el método get()
    println(solarSystem.get(3)) // Debería imprimir: Mars
}
```

Earth
Mars

Kotlin

Solutions Docs Community Teach Play

2.1.20 ▾ JVM ▾ Program arguments

Copy link Share code Run

```
/*
 * Unidad 3
 */
fun main() {
    // Creamos una lista inmutable con los 8 planetas
    val solarSystem = listOf("Mercury", "Venus", "Earth", "Mars",
        "Jupiter", "Saturn", "Uranus", "Neptune")

    // Buscar el índice de "Earth"
    println(solarSystem.indexOf("Earth")) // Debería imprimir: 2

    // Buscar el índice de "Pluto"
    println(solarSystem.indexOf("Pluto")) // Debería imprimir: -1
}
```

2
-1

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for Solutions, Docs, Community, Teach, Play, and a search icon. Below the tabs, it says "2.1.20" and "JVM". There is a "Program arguments" input field. On the right, there are buttons for "Copy link", "Share code", and "Run". The main area contains the following Kotlin code:

```
/*
 * Unidad 3
 */
fun main() {
    // Lista de planetas del sistema solar
    val solarSystem = listOf("Mercury", "Venus", "Earth", "Mars",
        "Jupiter", "Saturn", "Uranus", "Neptune")

    // Recorremos la lista con un bucle for y mostramos cada planeta
    for (planet in solarSystem) {
        println(planet)
    }
}
```

Below the code, the output window displays the names of the planets:

```
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
```

En el fragmento de código Kotlin que se muestra abajo, dentro de la función main, se crea una lista mutable llamada solarSystem inicializada con los nombres de los ocho planetas del sistema solar. Posteriormente, se realizan dos operaciones de modificación en esta lista. Primero, se agrega la cadena "Pluto" al final de la lista utilizando el método add(). Luego, se inserta la cadena "Theia" en la posición de índice 3 de la lista, lo que significa que se coloca como el cuarto elemento, desplazando los elementos siguientes.

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for Solutions, Docs, Community, Teach, Play, and a search icon. Below the tabs, it says "2.1.20" and "JVM". There is a "Program arguments" input field. On the right, there are buttons for "Copy link", "Share code", and "Run". The main area contains the following Kotlin code:

```
/*
 * Unidad 3
 */
fun main() {
    // Lista mutable de planetas
    val solarSystem = mutableListOf(
        "Mercury", "Venus", "Earth", "Mars",
        "Jupiter", "Saturn", "Uranus", "Neptune"
    )

    // Agregamos a Plutón al final de la lista
    solarSystem.add("Pluto")

    // Insertamos a Theia en la posición 3 (antes de Marte)
    solarSystem.add(3, "Theia")
}
```

Below the code, the output window displays the names of the planets:

```
Mercury
Venus
Earth
Theia
Mars
Jupiter
Saturn
Uranus
Neptune
Pluto
```

En este otro código Kotlin, dentro de la función main, se crea una lista mutable llamada solarSystem que contiene los nombres de algunos planetas, incluyendo "Theia" y "Pluto". Posteriormente, se realiza una modificación en esta lista: se actualiza el valor del elemento que se encuentra en el índice 3 (la cuarta posición en la lista) a la cadena "Future Moon". Finalmente, se imprimen en la consola los elementos que se encuentran en los índices 3 y 9 de la lista solarSystem. La salida

del programa muestra "Future Moon", que es el valor asignado al índice 3, y "Pluto", que se encuentra en el índice 9 de la lista.

The screenshot shows the Kotlin Play IDE interface. The code in the editor is:

```
fun main() {
    // Lista mutable con planetas
    val solarSystem = mutableListOf(
        "Mercury", "Venus", "Earth", "Theia", "Mars",
        "Jupiter", "Saturn", "Uranus", "Neptune", "Pluto"
    )

    // Actualizamos el valor en el indice 3
    solarSystem[3] = "Future Moon"

    // Imprimimos los elementos en el indice 3 y 9
    println(solarSystem[3])
    println(solarSystem[9])
}
```

The output window below shows the results of the run:

```
Future Moon
Pluto
```

The screenshot shows the Kotlin Play IDE interface. The code in the editor is:

```
/*
 * Unidad 3
 */
fun main() {
    // Lista mutable con planetas incluyendo Theia y Pluto
    val solarSystem = mutableListOf(
        "Mercury", "Venus", "Earth", "Future Moon", "Mars",
        "Jupiter", "Saturn", "Uranus", "Neptune", "Pluto"
    )

    // Quitar a "Pluto" por indice
    solarSystem.removeAt(9)

    // Quitar a "Future Moon" por nombre
    solarSystem.remove("Future Moon")
}
```

The output window below shows the results of the run:

```
false
false
```

The screenshot shows the Kotlin Play IDE interface. The code in the editor is:

```
/*
 * Unidad 3
 */
fun main() {
    // Crear un conjunto mutable de planetas
    val solarSystem = mutableSetOf("Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune")

    // Imprimir el tamaño inicial del conjunto
    println(solarSystem.size) // Debería imprimir 8

    // Agregar "Pluto" al conjunto
    solarSystem.add("Pluto")

    // Imprimir el tamaño después de agregar "Pluto"
    println(solarSystem.size) // Debería imprimir 9
}
```

The output window below shows the results of the run:

```
8
9
true
9
```

The screenshot shows the Kotlin Play IDE interface. The code editor contains the following Kotlin code:

```
/*
 * Unidad 3
 */
fun main() {
    // Crear un conjunto mutable de planetas
    val solarSystem = mutableSetOf("Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune")

    // Agregar "Pluto" al conjunto
    solarSystem.add("Pluto")

    // Usar remove() para quitar "Pluto" del conjunto
    solarSystem.remove("Pluto")

    // Imprimir el tamaño después de quitar "Pluto"
    println(solarSystem.size) // Debería imprimir 8
}
```

The output window below the code editor shows the results of the program's execution:

```
8
false
```

Podemos observar a continuación que dentro de la función main, se inicializa un mapa mutable llamado solarSystem donde las claves son los nombres de los planetas y los valores son sus respectivos números de lunas. Luego, se realizan varias operaciones de impresión. Primero, se imprime el tamaño del mapa, que es 8. Después, se intenta acceder a las lunas de "Pluto" y "Theia", que no están en el mapa, resultando en la impresión de null para ambos. A continuación, se imprimen las lunas de "Earth" (1) y "Mars" (2). Se vuelve a intentar acceder a las lunas de "Pluto" (imprimiendo null).

The screenshot shows the Kotlin Play IDE interface. The code editor contains the following Kotlin code:

```
/*
 * Unidad 3
 */
fun main() {
    // Crear un mapa mutable de planetas y sus Lunas
    val solarSystem = mutableMapOf(
        "Mercury" to 0,
        "Venus" to 0,
        "Earth" to 1,
        "Mars" to 2,
        "Jupiter" to 79,
        "Saturn" to 82,
        "Uranus" to 27,
        "Neptune" to 14
    )
}
```

The output window below the code editor shows the results of the program's execution:

```
8
9
5
null
```

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for Solutions, Docs, Community, Teach, Play, and a search bar. Below the tabs, the version is listed as 2.1.20 and the JVM target is selected. A "Program arguments" input field is present. On the right, there are buttons for Copy link, Share code, and Run.

```
/*
 * Unidad 3
 */
fun main() {
    // Crear un mapa mutable de planetas y sus Lunas
    val solarSystem = mutableMapOf(
        "Mercury" to 0,
        "Venus" to 0,
        "Earth" to 1,
        "Mars" to 2,
        "Jupiter" to 79,
        "Saturn" to 82,
        "Uranus" to 27,
        "Neptune" to 14
    )
}

null
8
78
```

Funciones de orden superior con colecciones

En este codelab se define una clase llamada Cookie con propiedades como nombre, si está horneada suave, si tiene relleno y su precio. Luego, se crea una lista inmutable llamada cookies que contiene varias instancias de la clase Cookie, representando diferentes tipos de galletas con sus respectivas características y precios.

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for Solutions, Docs, Community, Teach, Play, and a search bar. Below the tabs, the version is listed as 2.1.20 and the JVM target is selected. A "Program arguments" input field is present. On the right, there are buttons for Copy link, Share code, and Run.

```
/*
 * Unidad 3
 */
class Cookie(
    val name: String,
    val softBaked: Boolean,
    val hasfilling: Boolean,
    val price: Double
)

val cookies = listOf(
    Cookie(
        name = "Galleta de chocolate",
        softBaked = true,
        hasfilling = false,
        price = 1.5
    ),
    Cookie(
        name = "Galleta de vainilla",
        softBaked = false,
        hasfilling = false,
        price = 1.0
    ),
    Cookie(
        name = "Galleta de limón",
        softBaked = false,
        hasfilling = true,
        price = 1.2
    ),
    Cookie(
        name = "Galleta de mantequilla",
        softBaked = true,
        hasfilling = false,
        price = 1.8
    ),
    Cookie(
        name = "Galleta de galletas",
        softBaked = false,
        hasfilling = true,
        price = 2.0
    ),
    Cookie(
        name = "Galleta de jengibre",
        softBaked = true,
        hasfilling = false,
        price = 1.7
    ),
    Cookie(
        name = "Galleta de canela",
        softBaked = false,
        hasfilling = true,
        price = 1.9
    )
)
```

En el siguiente, dentro de la lambda, en lugar de imprimir el objeto Cookie completo, se accede específicamente a la propiedad name de cada galleta y se imprime. La salida será una lista de solo los nombres de cada galleta.

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
2.1.20 JVM
Program arguments

    name = "Sugar and Sprinkles",
    softBaked = false,
    hasFilling = false,
    price = 1.39
)
)

fun main() {
    cookies.forEach {
        println("Menu item: ${it.name}")
    }
}

Menu item: Cookie@6e2c634b.name
Menu item: Cookie@76fb509a.name
Menu item: Cookie@300ffa5d.name
Menu item: Cookie@1f17ae12.name
Menu item: Cookie@4d405ef7.name
Menu item: Cookie@6193b845.name
Menu item: Cookie@2e817b38.name
```

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
2.1.20 JVM
Program arguments

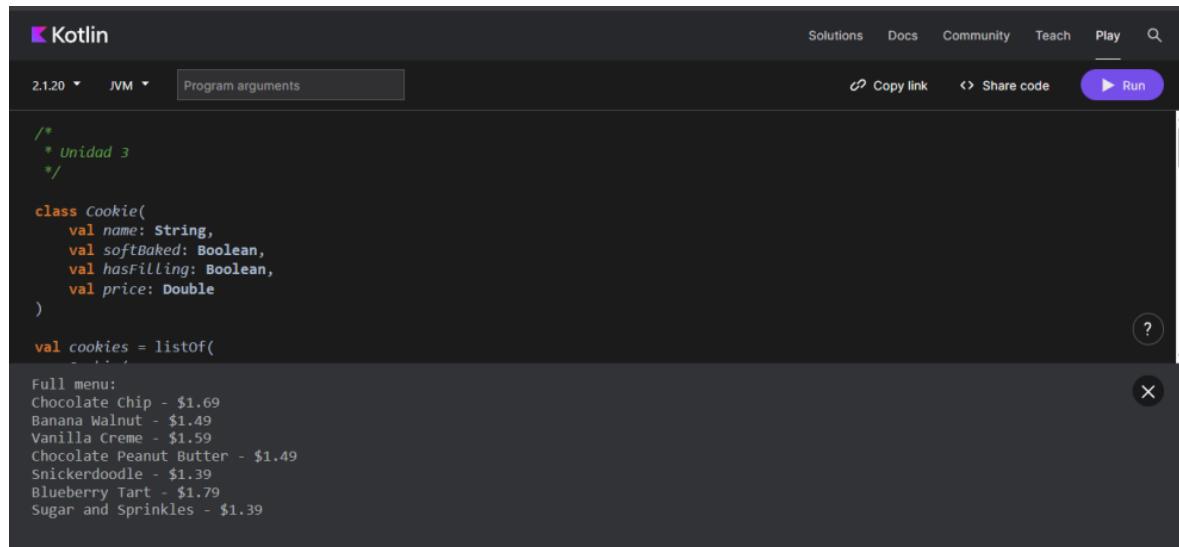
/*
 * Unidad 3
 */
class Cookie(
    val name: String,
    val softBaked: Boolean,
    val hasFilling: Boolean,
    val price: Double
)

val cookies = listOf(
    Cookie(
        name = "Sugar and Sprinkles",
        softBaked = false,
        hasFilling = false,
        price = 1.39
    ),
    Cookie(
        name = "Chocolate Chip",
        softBaked = true,
        hasFilling = false,
        price = 1.50
    ),
    Cookie(
        name = "Banana Walnut",
        softBaked = true,
        hasFilling = false,
        price = 1.50
    ),
    Cookie(
        name = "Vanilla Creme",
        softBaked = true,
        hasFilling = true,
        price = 1.75
    ),
    Cookie(
        name = "Chocolate Peanut Butter",
        softBaked = true,
        hasFilling = true,
        price = 1.75
    ),
    Cookie(
        name = "Snickerdoodle",
        softBaked = true,
        hasFilling = false,
        price = 1.25
    ),
    Cookie(
        name = "Blueberry Tart",
        softBaked = false,
        hasFilling = true,
        price = 2.00
    )
)

Menu item: Chocolate Chip
Menu item: Banana Walnut
Menu item: Vanilla Creme
Menu item: Chocolate Peanut Butter
Menu item: Snickerdoodle
Menu item: Blueberry Tart
Menu item: Sugar and Sprinkles
```

También se introduce la función de extensión map, la cual transforma cada elemento de la lista original en un nuevo elemento según la lambda proporcionada y devuelve

una nueva lista con estos elementos transformados. En este caso, la lambda toma cada objeto Cookie y crea una cadena que contiene el nombre de la galleta y su precio, formateado como "nombre - \$precio". La lista resultante fullMenu se imprime, mostrando el menú completo con el nombre y el precio de cada galleta.



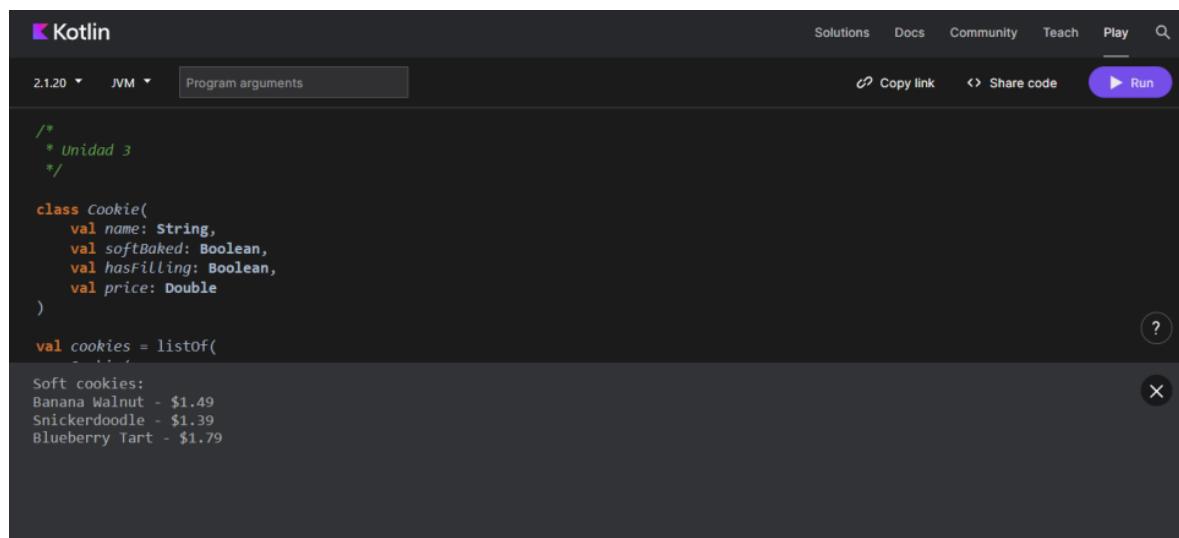
The screenshot shows the Kotlin playground interface. The code defines a `Cookie` class and creates a list of `Cookie` objects. A lambda expression is used to map each `Cookie` to a string representing its name and price. The resulting list, `fullMenu`, is then printed to the console, showing the full menu with all items and their prices.

```
/*
 * Unidad 3
 */

class Cookie(
    val name: String,
    val softBaked: Boolean,
    val hasFilling: Boolean,
    val price: Double
)

val cookies = listOf(
    ...
)

Full menu:
Chocolate Chip - $1.69
Banana Walnut - $1.49
Vanilla Creme - $1.59
Chocolate Peanut Butter - $1.49
Snickerdoodle - $1.39
Blueberry Tart - $1.79
Sugar and Sprinkles - $1.39
```



The screenshot shows the Kotlin playground interface. The code is identical to the first one, defining a `Cookie` class and creating a list of `Cookie` objects. However, the lambda expression uses the `filter` function to keep only the soft-baked cookies. The resulting list, `fullMenu`, is then printed to the console, showing only the soft-baked cookies and their prices.

```
/*
 * Unidad 3
 */

class Cookie(
    val name: String,
    val softBaked: Boolean,
    val hasFilling: Boolean,
    val price: Double
)

val cookies = listOf(
    ...
)

Soft cookies:
Banana Walnut - $1.49
Snickerdoodle - $1.39
Blueberry Tart - $1.79
```

```
2.1.20 ▾ JVM ▾ Program arguments

)
fun main() {
    val groupedMenu = cookies.groupBy { it.softBaked }

    val softBakedMenu = groupedMenu[true] ?: emptyList()
    val crunchyMenu = groupedMenu[false] ?: emptyList()

    println("Soft cookies:")
    softBakedMenu.forEach {
        println("${it.name} - ${it.price}")
    }
    println("Crunchy cookies:")
    Crunchy cookies:
    Chocolate Chip - $1.69
    Vanilla Creme - $1.59
    Chocolate Peanut Butter - $1.49
    Sugar and Sprinkles - $1.39
```

La función de extensión fold se utiliza para acumular un valor a través de todos los elementos de la lista. Toma un valor inicial (aquí 0.0) y una lambda que se ejecuta para cada elemento de la lista, tomando el valor acumulado actual y el elemento actual como argumentos y devolviendo el nuevo valor acumulado. En este caso, la lambda suma el precio de cada Cookie al total acumulado. El resultado final totalPrice es la suma de los precios de todas las galletas en la lista, y se imprime.

```
2.1.20 ▾ JVM ▾ Program arguments

class Cookie(
    val name: String,
    val softbaked: Boolean,
    val hasFilling: Boolean,
    val price: Double
)

val cookies = listOf(
    Cookie(
        name = "Chocolate Chip",
        softbaked = false,
        hasFilling = false,
        price = 1.69
    ),
    Cookie(
        name = "Vanilla Creme",
        softbaked = true,
        hasFilling = false,
        price = 1.59
    ),
    Cookie(
        name = "Chocolate Peanut Butter",
        softbaked = false,
        hasFilling = true,
        price = 1.49
    ),
    Cookie(
        name = "Sugar and Sprinkles",
        softbaked = true,
        hasFilling = false,
        price = 1.39
    )
)

Total price: $10.83
```

Finalmente, la función de extensión sortedBy se utiliza para ordenar los elementos de la lista según el valor devuelto por la lambda proporcionada. Devuelve una nueva lista con los elementos ordenados. Aquí, la lambda it.name indica que la lista de galletas debe ordenarse alfabéticamente por su nombre. La lista resultante alphabeticalMenu se imprime, mostrando solo el nombre de cada galleta en orden alfabético.

The screenshot shows the Kotlin Play IDE interface. At the top, there are dropdown menus for 'Solutions', 'Docs', 'Community', 'Teach', 'Play', and a search bar. Below the header, there are buttons for 'Copy link', 'Share code', and a purple 'Run' button. The main area contains the following Kotlin code:

```
class Cookie(
    val name: String,
    val softBaked: Boolean,
    val hasFilling: Boolean,
    val price: Double
)

val cookies = listOf(
    cookie(
        name = "Chocolate Chip",
        softBaked = false,
        hasFilling = false,
    )
)
```

Below the code, a tooltip displays an 'Alphabetical menu:' with the following items:

- Banana Walnut
- Blueberry Tart
- Chocolate Chip
- Chocolate Peanut Butter
- Snickerdoodle
- Sugar and Sprinkles
- Vanilla Creme

Práctica: Clases y colecciones

Se trabaja con clases, propiedades, constructores, y colecciones como listas y mapas, resolviendo una serie de tareas donde cada una plantea un pequeño reto, como crear clases con atributos, recorrer listas, filtrar datos o transformar colecciones, y luego se ofrece una solución guiada. A lo largo de las tareas

aprendemos a crear clases de datos como Event, refactorizar atributos utilizando enumeraciones (enum class), y trabajar con listas mutables.

The screenshot shows the Kotlin Play IDE interface. At the top, there are dropdown menus for 'Solutions', 'Docs', 'Community', 'Teach', 'Play', and a search bar. Below the header, there are buttons for 'Copy link', 'Share code', and a purple 'Run' button. The main area contains the following Kotlin code:

```
data class Event(
    val title: String,
    val description: String? = null,
    val daypart: String,
    val durationInMinutes: Int
)

fun main() {
    val event = Event(
        title = "Study Kotlin",
        description = "Commit to studying Kotlin at least 15 minutes per day.",
        daypart = "Evening",
        durationInMinutes = 15
    )

    println(event)
}
```

At the bottom of the code editor, the output of the run command is shown:

```
Event(title=Study Kotlin, description=Commit to studying Kotlin at least 15 minutes per day., daypart=Evening, durationInMinutes=15)
```

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
enum class Daypart {
    MORNING,
    AFTERNOON,
    EVENING
}

data class Event(
    val title: String,
    val description: String? = null,
    val daypart: Daypart,
    val durationInMinutes: Int
)

fun main() {
    val event = Event(
        title = "Study Kotlin",
        description = "Commit to studying Kotlin at least 15 minutes per day.",
        daypart = Daypart.EVENING,
        durationInMinutes = 15
    )

    println(event)
}
```

Event(title=Study Kotlin, description=Commit to studying Kotlin at least 15 minutes per day., daypart=EVENING, durationInMinutes=15)

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
// Enum para representar las franjas del día
enum class Daypart {
    MORNING,
    AFTERNOON,
    EVENING,
}

// Clase de datos para eventos
data class Event(
    val title: String,
    val description: String? = null,
    val daypart: Daypart,
    val durationInMinutes: Int,
)

fun main() {
    // Crear eventos individuales
    val event1 = Event(title = "Wake up", description = "Time to get up", daypart = Daypart.MORNING, durationInMinutes = 0)
    val event2 = Event(title = "Eat breakfast", daypart = Daypart.MORNING, durationInMinutes = 15)
    val event3 = Event(title = "Learn about Kotlin", daypart = Daypart.AFTERNOON, durationInMinutes = 30)
    val event4 = Event(title = "Practice Compose", daypart = Daypart.AFTERNOON, durationInMinutes = 60)
    val event5 = Event(title = "Watch latest DevBytes video", daypart = Daypart.AFTERNOON, durationInMinutes = 10)
    val event6 = Event(title = "Check out latest Android Jetpack library", daypart = Daypart.EVENING, durationInMinutes = 45)
}

// Lista mutable de eventos
```

```
val event1 = Event(title = "Watch latest DevBytes video", daypart = Daypart.AFTERNOON, durationInMinutes = 10)
val event2 = Event(title = "Check out latest Android Jetpack library", daypart = Daypart.EVENING, durationInMinutes = 45)

// Lista mutable de eventos
val events = mutableListOf(event1, event2, event3, event4, event5, event6)

// Mostrar cantidad de eventos
println("Total events scheduled: ${events.size}")

// Mostrar todos los eventos
events.forEach {
    println(it)
}

Total events scheduled: 6
Event(title=Wake up, description=Time to get up, daypart=MORNING, durationInMinutes=0)
Event(title=Eat breakfast, description=null, daypart=MORNING, durationInMinutes=15)
Event(title=Learn about Kotlin, description=null, daypart=AFTERNOON, durationInMinutes=30)
Event(title=Practice Compose, description=null, daypart=AFTERNOON, durationInMinutes=60)
Event(title=Watch latest DevBytes video, description=null, daypart=AFTERNOON, durationInMinutes=10)
Event(title=Check out latest Android Jetpack library, description=null, daypart=EVENING, durationInMinutes=45)
```

Kotlin

Solutions Docs Community Teach Play ?

2.1.20 ▾ JVM ▾ Program arguments

Copy link Share code Run

```
// Mostrar cantidad total de eventos
println("Total events scheduled: ${events.size}")

// Mostrar todos los eventos
events.forEach {
    println(it)
}

println() // Línea en blanco para separar

// Filtrar eventos cortos (menos de 60 min) y mostrar cantidad
val shortEvents = events.filter { it.durationInMinutes < 60 }
println("You have ${shortEvents.size} short events.")

Total events scheduled: 6
Event(title=Wake up, description=Time to get up, daypart=MORNING, durationInMinutes=0)
Event(title=Eat breakfast, description=null, daypart=MORNING, durationInMinutes=15)
Event(title=Learn about Kotlin, description=null, daypart=AFTERNOON, durationInMinutes=30)
Event(title=Practice Compose, description=null, daypart=AFTERNOON, durationInMinutes=60)
Event(title=Watch latest DevBytes video, description=null, daypart=AFTERNOON, durationInMinutes=10)
Event(title=Check out latest Android Jetpack library, description=null, daypart=EVENING, durationInMinutes=45)

You have 5 short events.
```

Kotlin

Solutions Docs Community Teach Play

2.1.20 ▾ JVM ▾ Program arguments

Copy link Share code Run

```
// Mostrar resumen de eventos cortos
val shortEvents = events.filter { it.durationInMinutes < 60 }
println("You have ${shortEvents.size} short events.\n")

// Agrupar eventos por franja horaria y mostrar resumen
val groupedEvents = events.groupBy { it.daypart }
groupedEvents.forEach { (daypart, eventsByDaypart) ->
    println("${daypart.name.lowercase().replaceFirstChar { it.uppercase() }}: ${eventsByDaypart.size} events")
}
}
```

You have 5 short events.

Morning: 2 events
Afternoon: 3 events
Evening: 1 events

Kotlin

Solutions Docs Community Teach Play

2.1.20 ▾ JVM ▾ Program arguments

Copy link Share code Run

```
// Mostrar resumen de eventos cortos
val shortEvents = events.filter { it.durationInMinutes < 60 }
println("You have ${shortEvents.size} short events.\n")

// Agrupar eventos por franja horaria y mostrar resumen
val groupedEvents = events.groupBy { it.daypart }
groupedEvents.forEach { (daypart, eventsByDaypart) ->
    println("${daypart.name.lowercase().replaceFirstChar { it.uppercase() }}: ${eventsByDaypart.size} events")
}

println("\nLast event of the day: ${events.last().title}")
}
```

You have 5 short events.

Morning: 2 events
Afternoon: 3 events
Evening: 1 events

Last event of the day: Check out latest Android Jetpack library

The screenshot shows the Kotlin playground interface. At the top, there's a navigation bar with links for Solutions, Docs, Community, Teach, Play, and a search icon. Below the navigation bar, there are dropdown menus for '2.1.20' and 'JVM', and a 'Program arguments' input field. To the right of these are 'Copy link', 'Share code', and a purple 'Run' button. The main area contains code in Spanish and English. The code defines a class with a property 'durationOfEvent' that returns 'short' if the duration is less than 60 minutes, and 'long' otherwise. It also prints the total number of events and the last event of the day. The output window shows the results: 'You have 5 short events.', 'Morning: 2 events', 'Afternoon: 3 events', 'Evening: 1 events', 'Last event of the day: Check out latest Android Jetpack library', and 'Duration of first event of the day: short'.

```
// Llama a la propiedad de extensión que se creó para obtener si ese evento es "short" o "Long"
// (según su duración).
    println("\nDuration of first event of the day: ${events[0].durationOfEvent}")

}

// Se crea una propiedad de extensión llamada durationOfEvent para la clase Event para darle una descripción
// (en este caso "short" o "Long")
val Event.durationOfEvent: String
    get() = if (this.durationInMinutes < 60) {
        "short"
    } else {
        "long"
    }

You have 5 short events.

Morning: 2 events
Afternoon: 3 events
Evening: 1 events

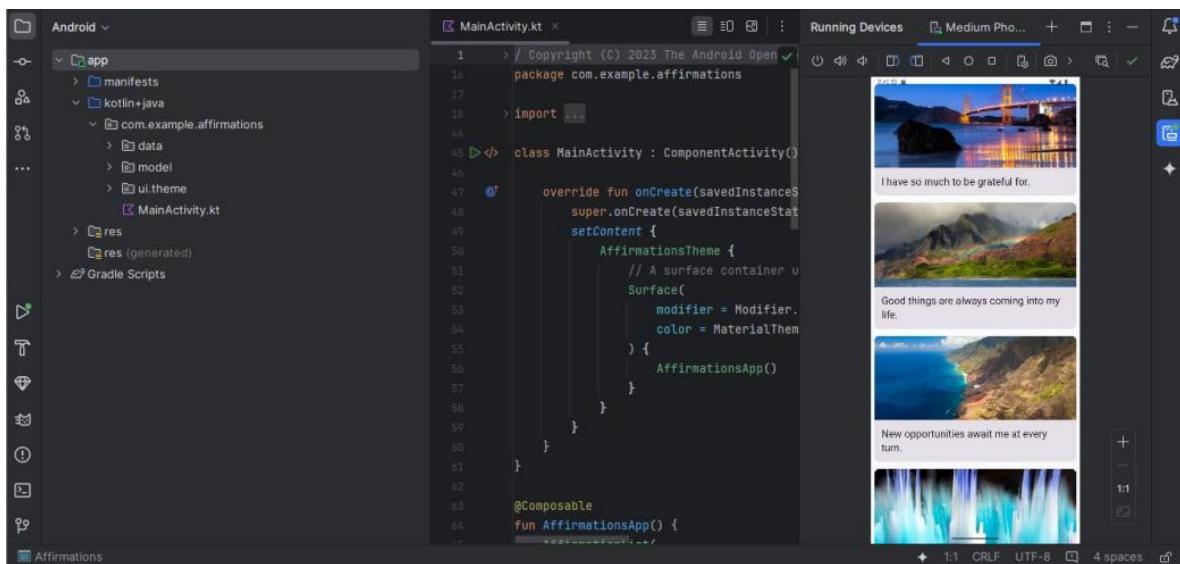
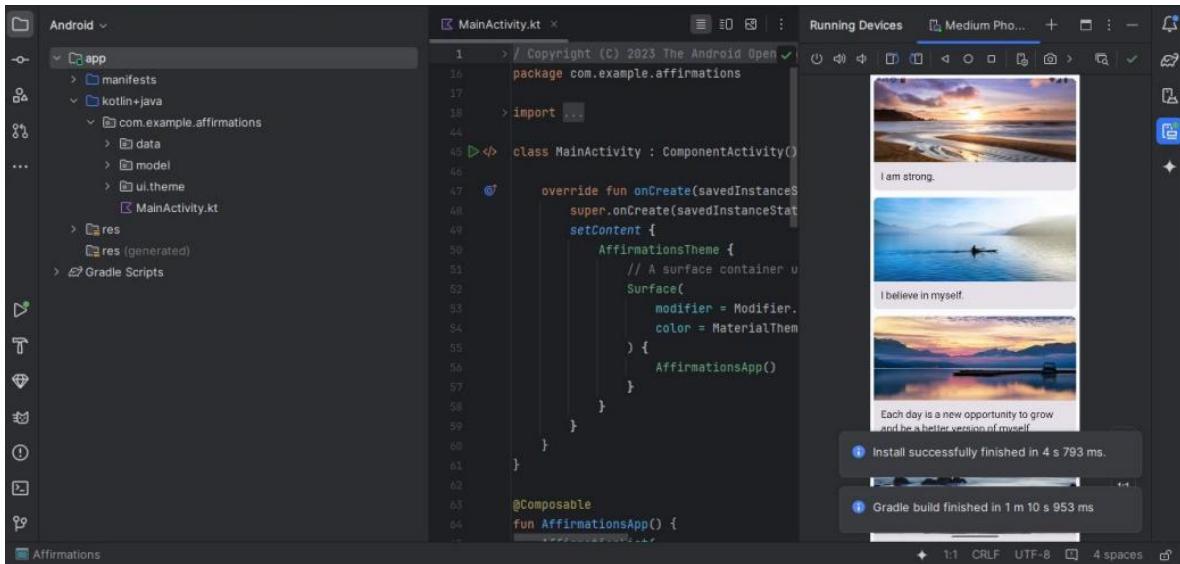
Last event of the day: Check out latest Android Jetpack library

Duration of first event of the day: short
```

Ruta de Aprendizaje 2 (Crea una lista desplazable)

¿Cómo agregar una lista desplazable?

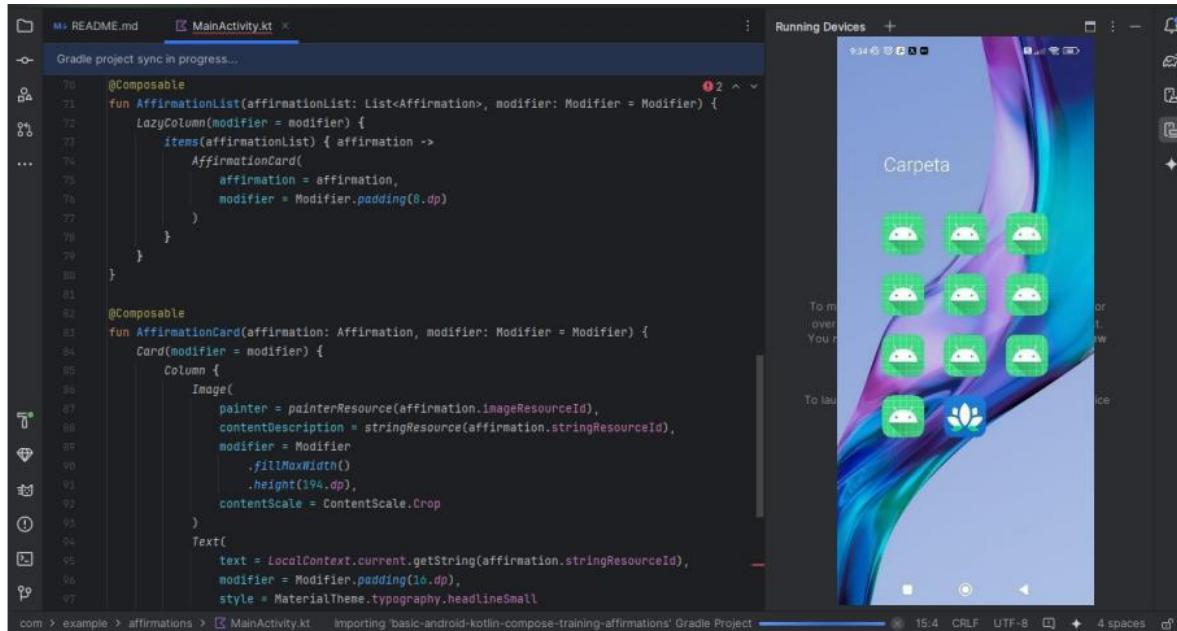
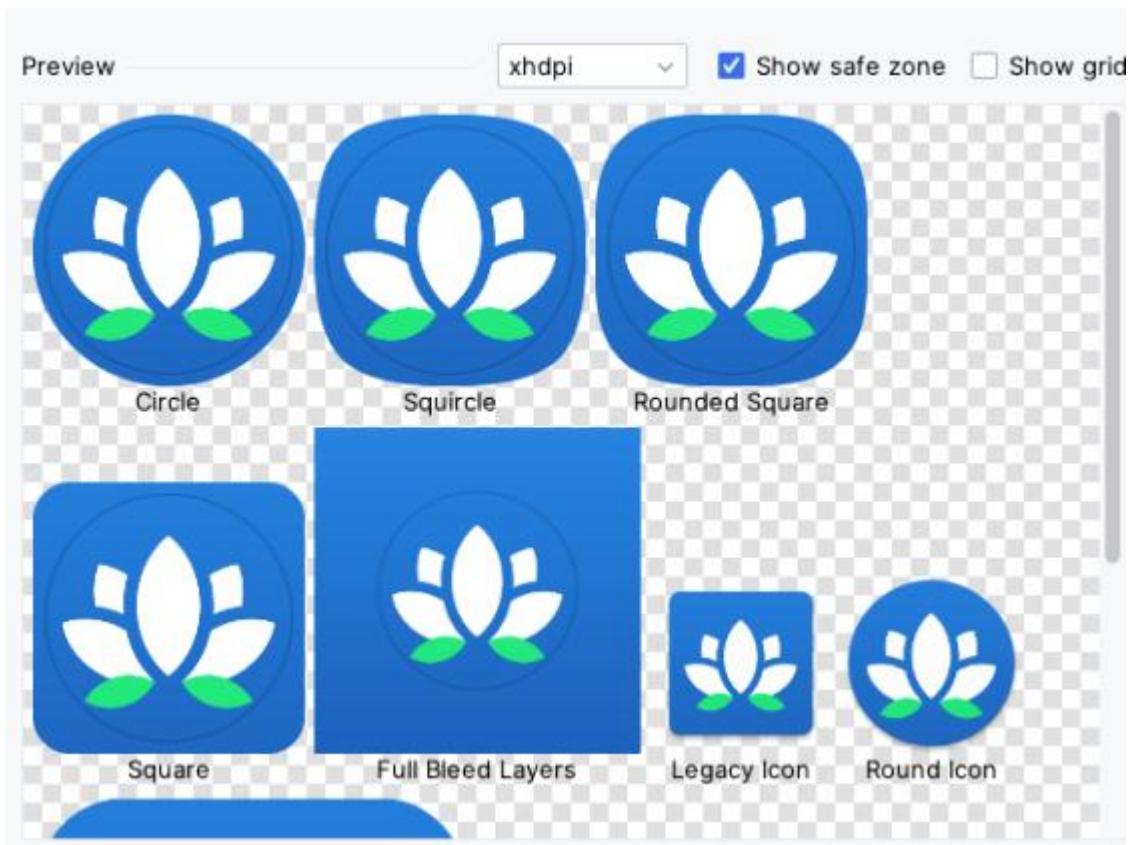
En este codelab se utiliza Jetpack Compose para mostrar una lista desplazable de afirmaciones, donde cada afirmación consiste en una imagen y un texto. La clase de datos `Affirmation` define la estructura de cada elemento de la lista, conteniendo IDs de recursos para la cadena y la imagen. La clase `Datasource` se encarga de crear y proporcionar la lista de objetos `Affirmation`, cargando las referencias a los recursos de la aplicación. En la interfaz de usuario (`MainActivity.kt`), el composable `AffirmationCard` define cómo se muestra cada elemento individual dentro de una tarjeta, incluyendo una imagen y un texto con estilos definidos. El composable `AffirmationList` utiliza `LazyColumn` para mostrar de manera eficiente la lista de afirmaciones, solo componiendo los elementos visibles en pantalla y permitiendo el desplazamiento. Finalmente, el composable `AffirmationsApp` ensambla toda la interfaz, obteniendo los datos del `Datasource` y mostrando la lista dentro de una `Surface` con el tema de la aplicación aplicado.



Cómo cambiar el ícono de la app:

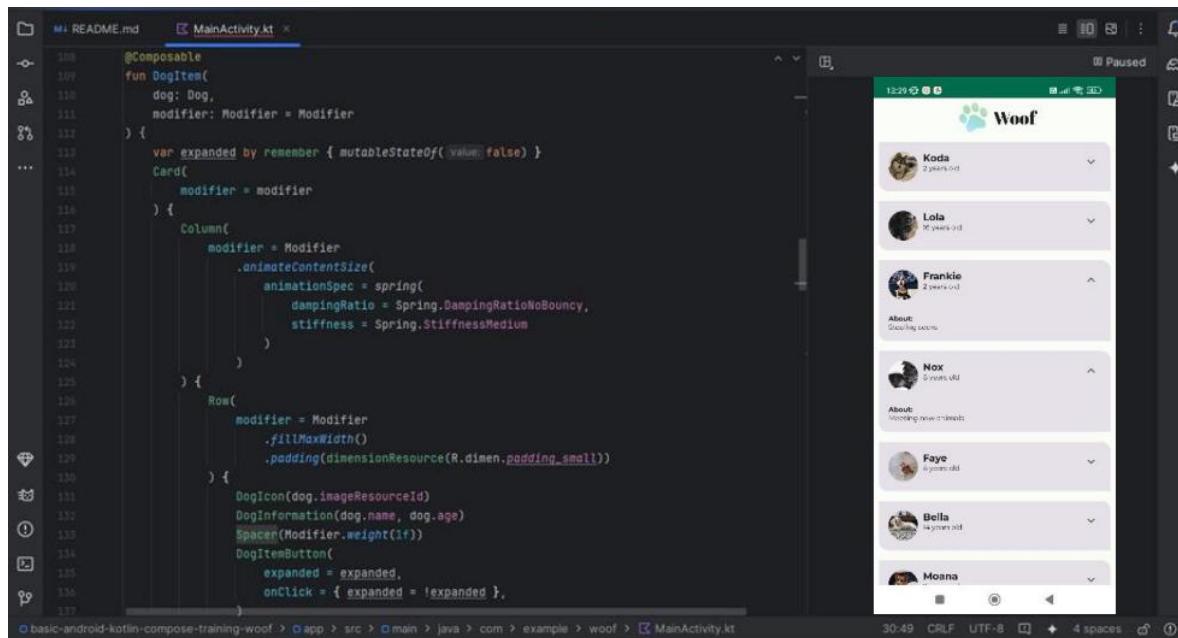
Para personalizar el ícono de la aplicación y reemplazar el logo predeterminado de Android, se llevó a cabo el proceso de configuración utilizando los recursos gráficos adecuados. Primero, se colocaron los archivos de íconos en las carpetas mipmap correspondientes, generando distintas versiones del ícono (mdpi, hdpi, xhdpi, xxhdpi y xxxhdpi) para asegurar compatibilidad con distintos tamaños y densidades de pantalla. También se incluyeron calificadores de recursos (v24, v26, etc.) para aplicar versiones específicas del ícono en dispositivos con diferentes niveles de API.

Se manejó el uso de Image Asset Studio, una herramienta integrada en Android Studio, para generar de forma sencilla los íconos heredados y adaptables, asegurando así que la aplicación tenga un ícono personalizado de alta calidad.



Ruta de Aprendizaje 3 (Compila apps fabulosas) Temas de Material con Jetpack Compose - Animación simple con Jetpack Compose

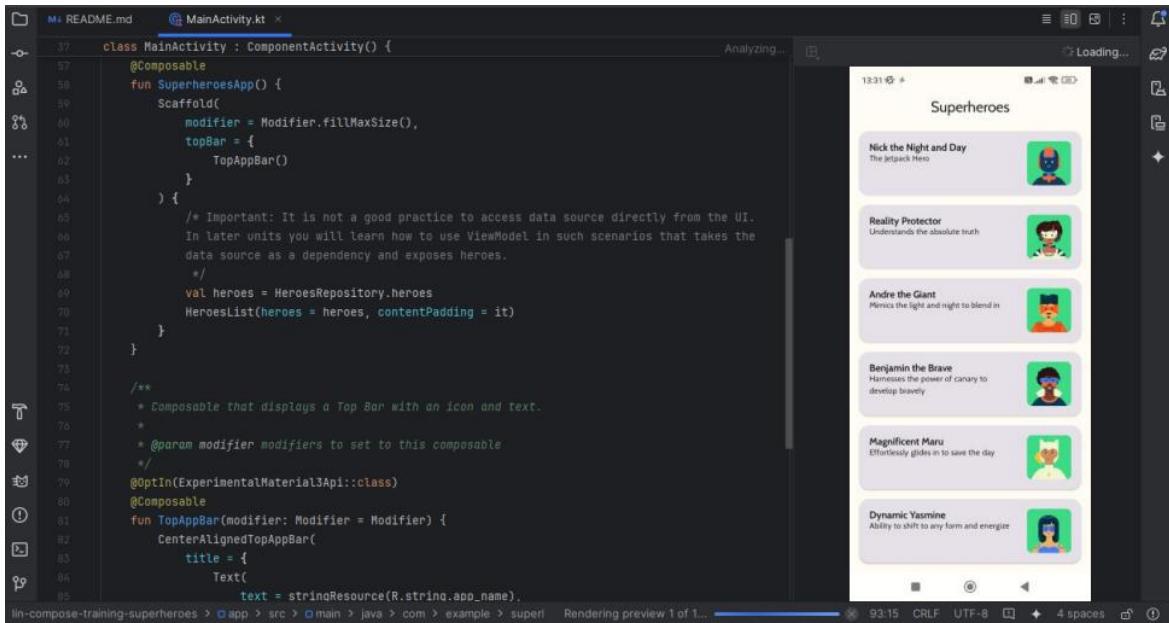
En este codelab se ve acerca de la app Woof, donde se muestra una lista de perros usando Material Design para crear una experiencia de usuario atractiva. El codelab se enfoca en el uso de Temas de Material para mejorar el aspecto de las apps. El archivo Theme.kt contiene la información del tema de la app, definido por color, tipografía y forma. Dentro de este archivo, el elemento WoofTheme() establece los colores, tipografía y formas de la app. Además, la app utiliza paletas de colores para temas claros y oscuros, y define estilos de tipo específicos. El archivo Color.kt incluye los colores usados, que luego se asignan en Theme.kt a ranuras específicas para los temas claros y oscuros, mientras que Shape.kt define las formas de la app (pequeña, mediana y grande) con opciones para redondear las esquinas.



Esta práctica se enfoca en crear una app de superhéroes con una interfaz atractiva, aplicando **Material Design**. Se personaliza la apariencia usando temas claro y oscuro, una fuente personalizada (Cabin) y formas definidas para la UI.

La estructura incluye una **barra superior** usando Scaffold, con título estilizado (DisplayLarge) y alineado. También se modifica el color de la **barra de estado** para pantalla completa con setUpEdgeToEdge en Theme.kt.

La **lista de superhéroes** se implementa en HeroesScreen.kt, mostrando cada héroe con imagen redondeada, nombre (DisplaySmall) y descripción (BodyLarge), todo con diseño en Box y uso de padding y recorte. Finalmente, se crea una **lista desplazable eficiente** con LazyColumn, aplicando el diseño anterior a cada elemento.



Unidad 4: Navegación y arquitectura de la app

Ruta de Aprendizaje 1 (Componentes de la arquitectura)

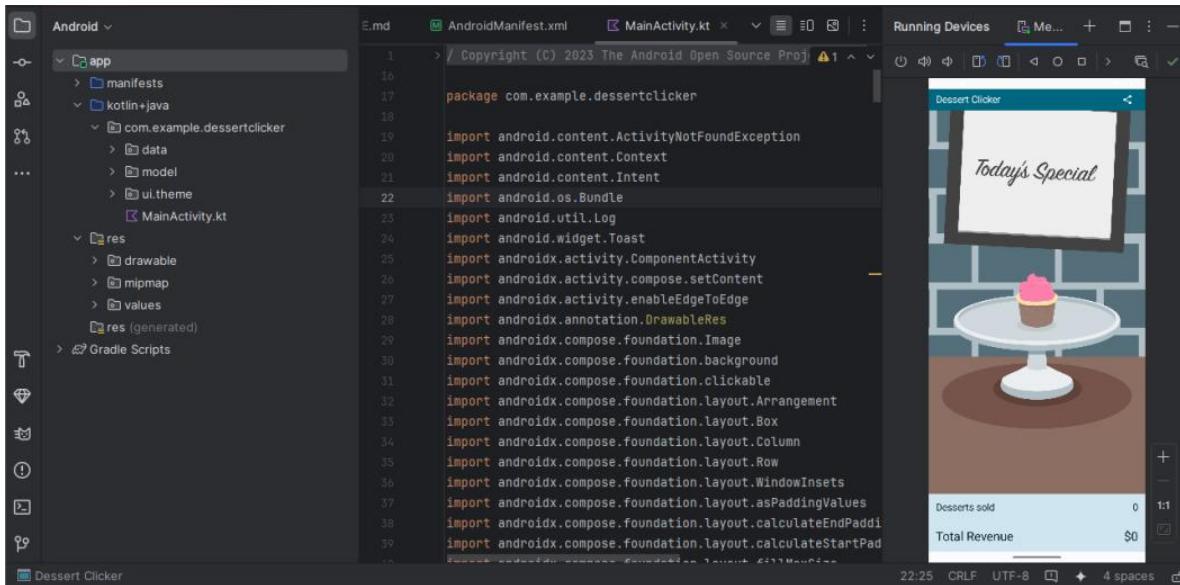
Etapas del ciclo de vida de la actividad

Este codelab explora el **ciclo de vida de las actividades** en Android, mostrando cómo una app transita por distintos estados desde su creación hasta su destrucción. Usando **Logcat**, se observa el orden en que se ejecutan los métodos del ciclo en situaciones como inicio, cierre, paso a segundo plano e interacción entre actividades.

Se destaca cómo los **cambios de configuración**, como la **rotación de pantalla**, provocan la destrucción y recreación de la actividad, lo que puede causar pérdida del estado de la interfaz.

Para resolver esto en **Jetpack Compose**, se explica la diferencia entre `remember` y `rememberSaveable`:

- `remember` guarda el estado solo durante las recomposiciones.
- `rememberSaveable` lo conserva incluso tras cambios de configuración, como la rotación.



Resumen: ViewModel y el estado en Compose

Este codelab explica cómo se gestiona el estado en una app Jetpack Compose usando el juego **Unscramble** como ejemplo.

La **MainActivity** carga la interfaz definida en GameScreen.kt, que muestra la palabra desordenada, un campo de entrada, botones de acción ("Submit" y "Skip") y la puntuación actual.

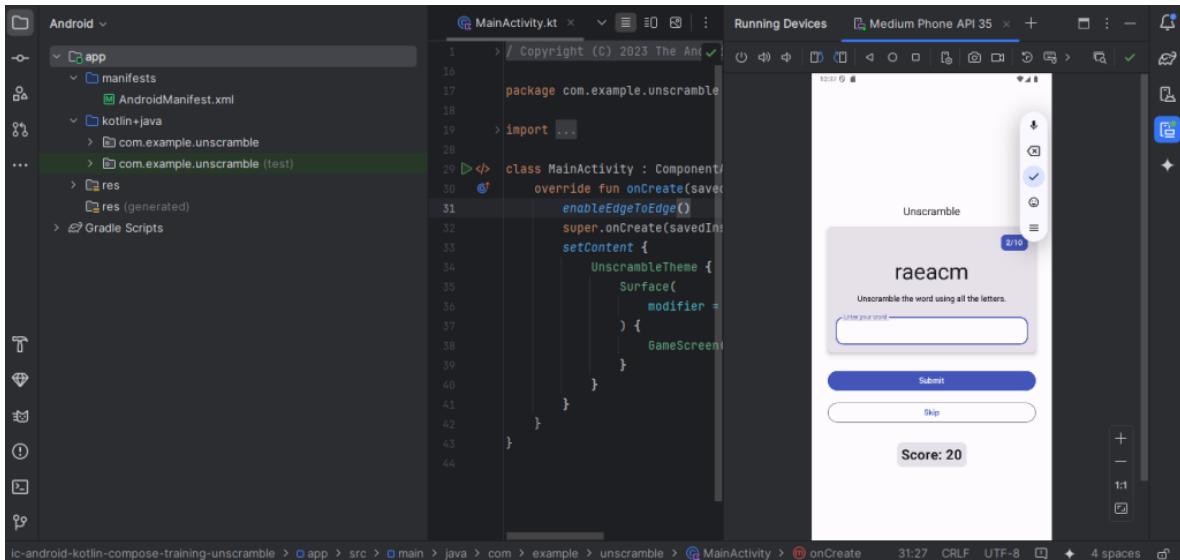
La lógica del juego está en el **GameViewModel**, que:

- Toma palabras de WordsData.kt,
- Las desordena,
- Mantiene el estado del juego (palabra actual, puntuación y progreso).

Cuando el usuario interactúa (por ejemplo, al enviar una respuesta), esa acción se envía al **ViewModel**, que:

- Verifica la respuesta,
- Actualiza el estado (como la puntuación o la siguiente palabra),
- Y notifica a GameScreen.

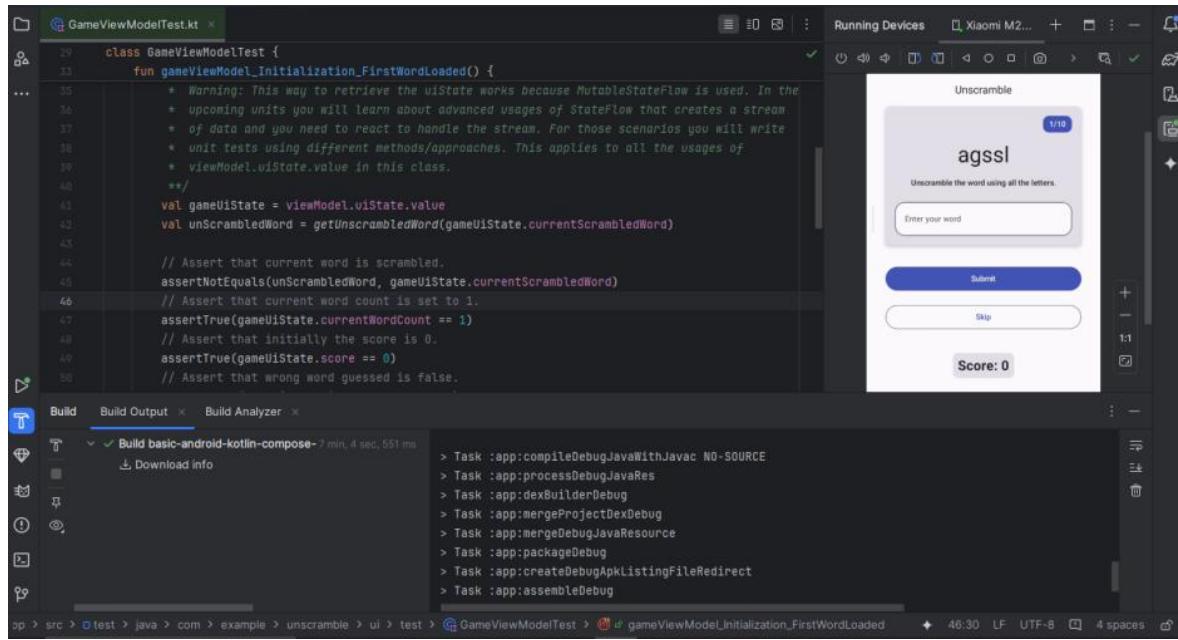
Gracias a que **Jetpack Compose es reactivo**, cualquier cambio en el estado del ViewModel se refleja automáticamente en la UI, sin necesidad de actualizarla manualmente.



Cómo escribir pruebas de unidades para ViewModel

Para garantizar la calidad y confiabilidad de la lógica del juego en la aplicación Unscramble, se implementaron pruebas unitarias sobre la clase GameViewModel. Estas pruebas siguen una estrategia centrada en cubrir distintas rutas de ejecución del código, abordando los siguientes escenarios clave:

- Ruta de éxito: Validamos que el flujo principal del juego se comporta como se espera cuando el usuario proporciona una palabra correcta. Esto incluye la actualización correcta de la puntuación, la cantidad de palabras mostradas y el reinicio del indicador de error.
- Ruta de error: Se verifica que, al ingresar una palabra incorrecta, la app no modifique la puntuación y establezca correctamente una bandera de error (isGuessedWordWrong).
- Caso límite: Se comprueba que el estado inicial del juego esté configurado correctamente al iniciar una nueva partida. Esto incluye asegurarse de que el contador de palabras comience en 1, la puntuación sea 0, y que no haya errores ni se haya alcanzado el fin del juego.



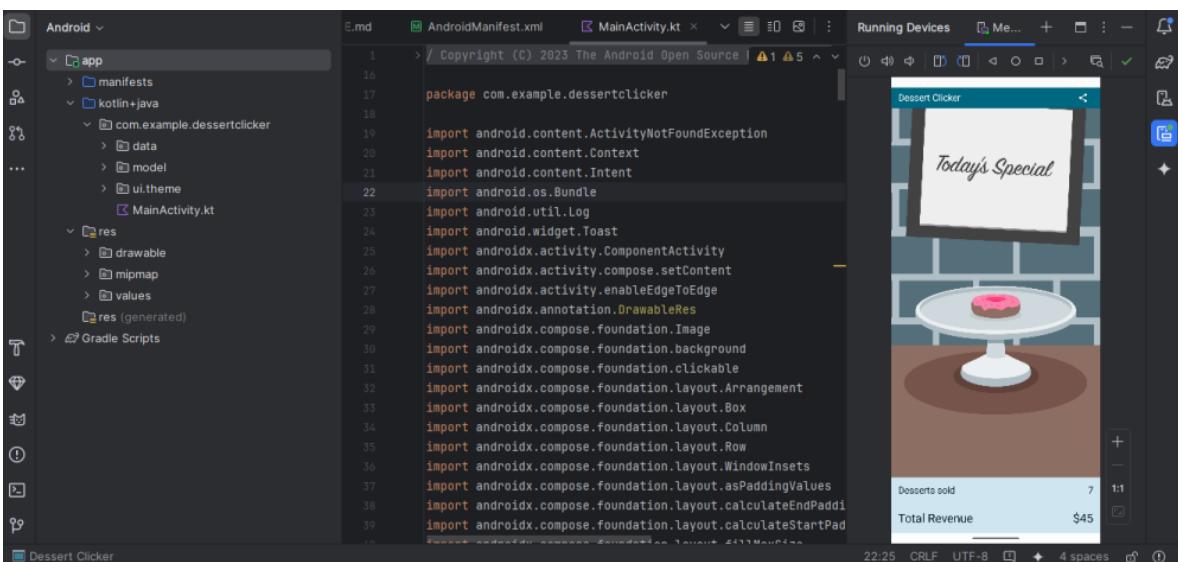
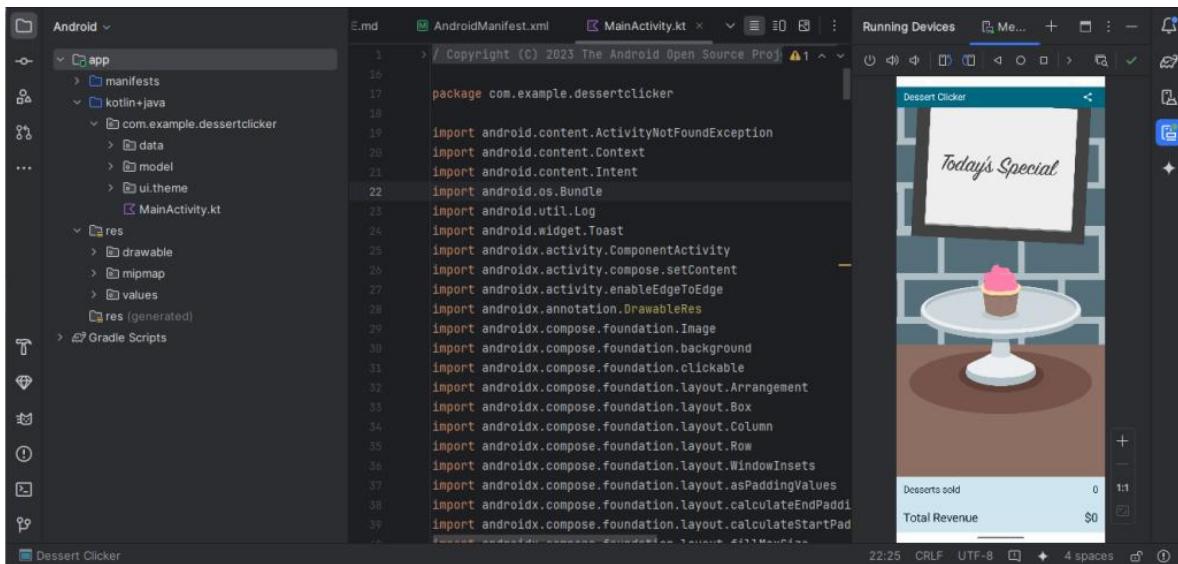
Práctica: Agrega un ViewModel a Dessert Clicker

En esta práctica se refactoriza la arquitectura de la aplicación Dessert Clicker para adoptar el patrón **ViewModel**, mejorando la gestión del estado en relación con el ciclo de vida de la UI. Inicialmente, el estado y la lógica de negocio estaban directamente incrustados en MainActivity, lo cual no es ideal para aplicaciones escalables o resistentes a cambios de configuración como la rotación de pantalla.

El primer paso es agregar la dependencia de ViewModel en el proyecto y definir una clase de datos (data class) que encapsule el estado de la UI, brindando una estructura clara para su gestión.

Posteriormente, se crea la clase DessertViewModel, que centraliza el manejo del estado y la lógica de negocio. Esta clase sobrevive a cambios de configuración y desacopla la lógica de la interfaz, lo cual es un principio clave en arquitecturas modernas como MVVM (Model-View-ViewModel).

La MainActivity queda ahora enfocada únicamente en renderizar la interfaz gráfica y observar el estado expuesto por el ViewModel, así como delegar en él las acciones del usuario (como clics). Esta separación mejora la mantenibilidad, reutilización y testeo del código.

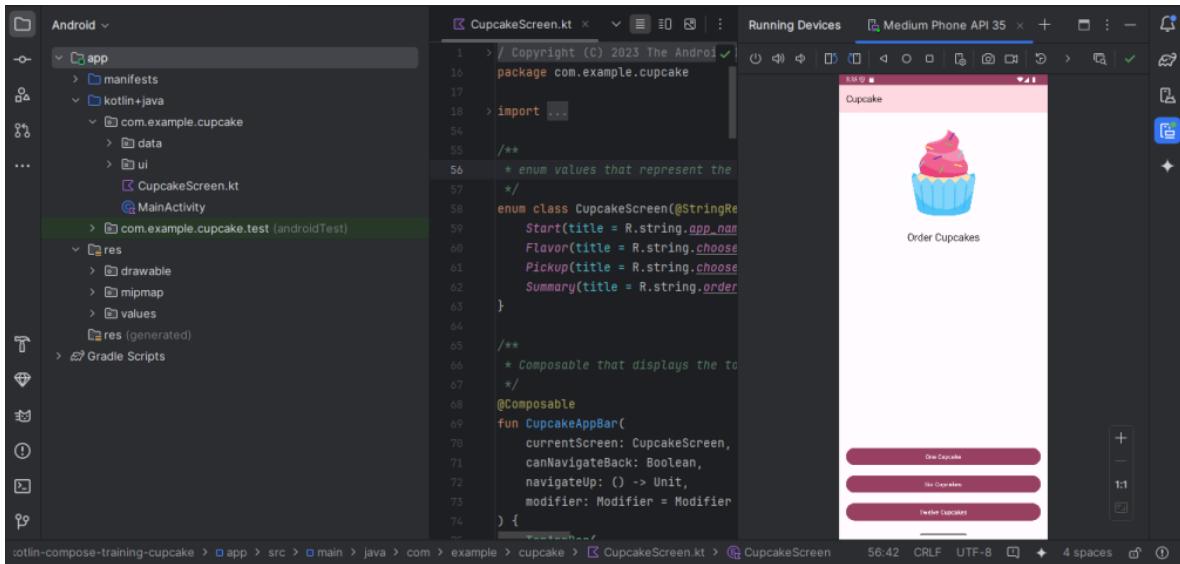


Ruta de Aprendizaje 2

(Navigation en Jetpack Compose)

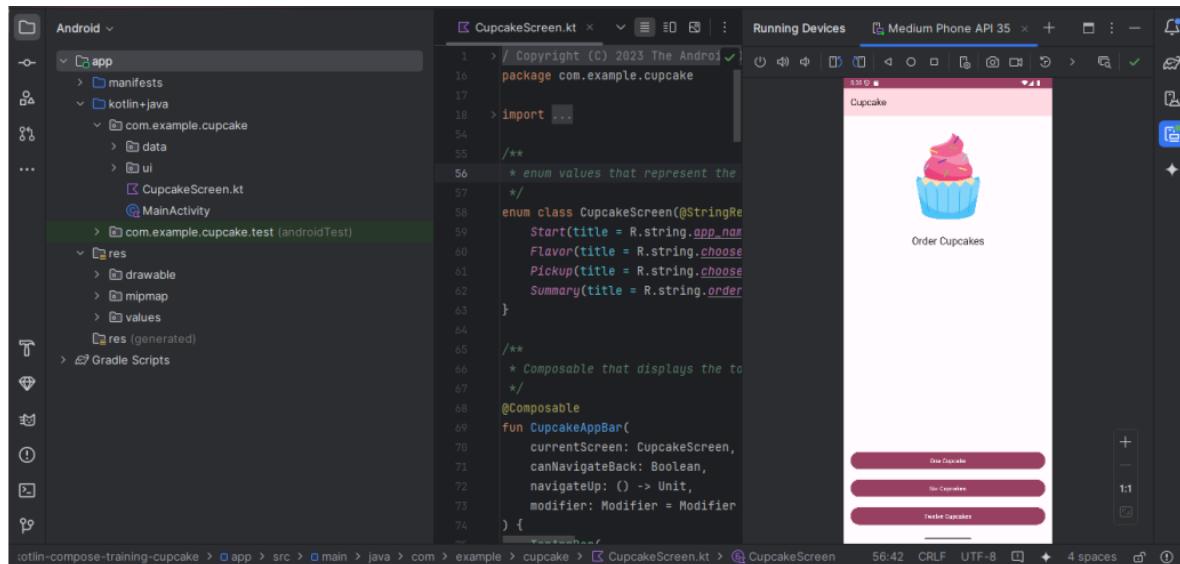
Cómo navegar entre pantallas con Compose

Aquí se explica cómo el componente Navigation de Jetpack Compose facilita la definición de rutas únicas para cada pantalla y cómo asociar cada ruta con un elemento componible específico mediante el uso de un NavHost. El NavController se presenta como el responsable de gestionar la navegación entre estas rutas. En lugar de pasar directamente el NavController a las pantallas individuales, el codelab enfatiza la importancia de utilizar callbacks de eventos para desacoplar la lógica de navegación de la interfaz de usuario de cada pantalla, manteniendo la gestión de la navegación centralizada dentro del NavHost.



```
1 > / Copyright (C) 2023 The Android
2 package com.example.cupcake
3
4 > import ...
5
6 /**
7 * enum values that represent the
8 */
9 enum class CupcakeScreen(@StringRes
10     Start(title = R.string.app_name)
11     Flavor(title = R.string.choose)
12     Pickup(title = R.string.choose)
13     Summary(title = R.string.order)
14 )
15
16 /**
17 * Composable that displays the top
18 */
19 @Composable
20 fun CupcakeAppBar(
21     currentScreen: CupcakeScreen,
22     canNavigateBack: Boolean,
23     navigateUp: () -> Unit,
24     modifier: Modifier = Modifier
25 ) {
```

Inicialmente aquí se establece la configuración necesaria para las pruebas de navegación, lo que implica la creación y gestión de un TestNavController dentro de una `AndroidComposeTestRule`, permitiendo así la inspección del flujo de navegación de la aplicación. Posteriormente, el codelab guía en la implementación de pruebas específicas para verificar la correcta navegación entre las diferentes pantallas de la aplicación Cupcake, simulando las interacciones del usuario con los botones de navegación y asegurando que la aplicación se dirige a la pantalla esperada en cada caso, incluyendo la navegación hacia adelante, hacia atrás y la cancelación de flujos.



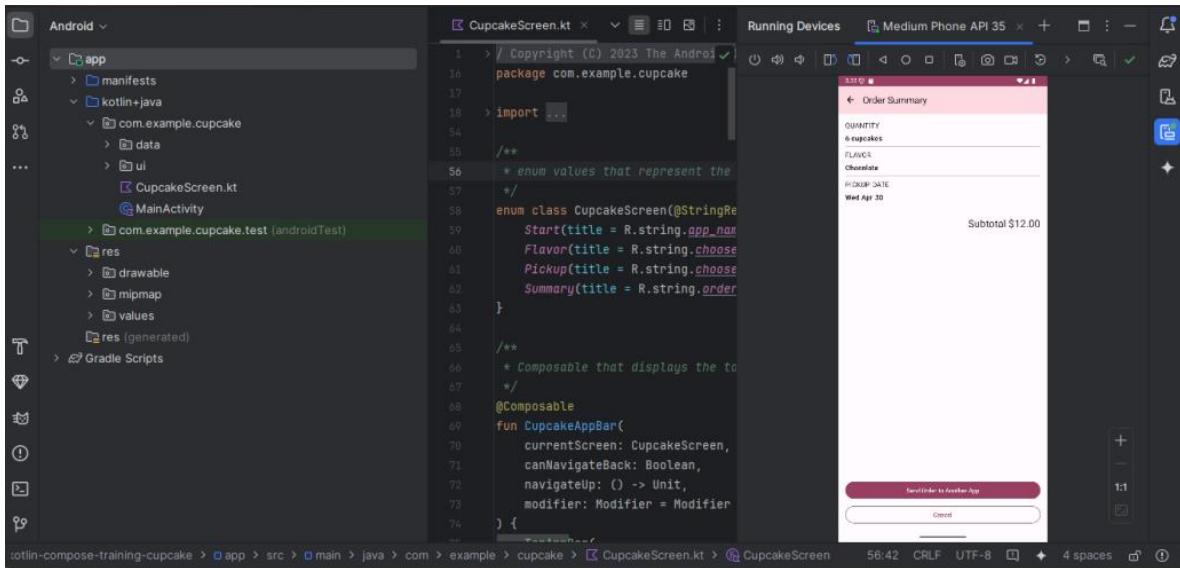
```
1 > / Copyright (C) 2023 The Android
2 package com.example.cupcake
3
4 > import ...
5
6 /**
7 * enum values that represent the
8 */
9 enum class CupcakeScreen(@StringRes
10     Start(title = R.string.app_name)
11     Flavor(title = R.string.choose)
12     Pickup(title = R.string.choose)
13     Summary(title = R.string.order)
14 )
15
16 /**
17 * Composable that displays the top
18 */
19 @Composable
20 fun CupcakeAppBar(
21     currentScreen: CupcakeScreen,
22     canNavigateBack: Boolean,
23     navigateUp: () -> Unit,
24     modifier: Modifier = Modifier
25 ) {
```

The screenshot shows the Android Studio interface with the code editor open to `CupcakeScreen.kt`. The code defines an enum for cupcake flavors and a Composable function for displaying the flavor selection screen. In the bottom right corner, a preview window displays a "Choose Flavor" dialog with five options: Vanilla, Chocolate, Red Velvet, Salted Caramel, and Coffee. The "Chocolate" option is selected. Below the dialog, the subtotal is listed as \$12.00.

```
1 > / Copyright (C) 2023 The Android
16 package com.example.cupcake
17
18 > import ...
19
20 /**
21 * enum values that represent the
22 */
23 enum class CupcakeScreen(@StringRes
24     Start(title = R.string.app_name),
25     Flavor(title = R.string.choose),
26     Pickup(title = R.string.choose),
27     Summary(title = R.string.order)
28 )
29
30 /**
31 * Composable that displays the to
32 */
33 @Composable
34 fun CupcakeAppBar(
35     currentScreen: CupcakeScreen,
36     canNavigateBack: Boolean,
37     navigateUp: () -> Unit,
38     modifier: Modifier = Modifier
39 ) {
```

The screenshot shows the Android Studio interface with the code editor open to `CupcakeScreen.kt`. The code is identical to the previous screenshot. In the bottom right corner, a preview window displays a "Choose Pickup Date" dialog with four options: Tue Apr 29, Wed Apr 30, Thu May 1, and Fri May 2. The "Wed Apr 30" option is selected. Below the dialog, the subtotal is listed as \$12.00.

```
1 > / Copyright (C) 2023 The Android
16 package com.example.cupcake
17
18 > import ...
19
20 /**
21 * enum values that represent the
22 */
23 enum class CupcakeScreen(@StringRes
24     Start(title = R.string.app_name),
25     Flavor(title = R.string.choose),
26     Pickup(title = R.string.choose),
27     Summary(title = R.string.order)
28 )
29
30 /**
31 * Composable that displays the to
32 */
33 @Composable
34 fun CupcakeAppBar(
35     currentScreen: CupcakeScreen,
36     canNavigateBack: Boolean,
37     navigateUp: () -> Unit,
38     modifier: Modifier = Modifier
39 ) {
```



Unidad 5: Cómo conectarse a Internet

Ruta de Aprendizaje 1 (Cómo obtener datos de Internet)

Introducción a las corrotinas en el Playground de Kotlin

Este codelab de Android tiene como objetivo el enseñar cómo mejorar el rendimiento de una aplicación utilizando programación asíncrona y concurrente. El enfoque inicial compara una ejecución secuencial básica sin corrotinas con otra que utiliza `delay()` para simular un retardo, mostrando cómo el uso de `runBlocking` permite introducir esta pausa sin necesidad de bloquear completamente el hilo principal. En este caso este código de Kotlin define una función principal (`main`) que contiene dos instrucciones. La primera instrucción utiliza la función `println()` para mostrar la frase "Weather forecast" en la consola. Acto seguido, la segunda instrucción también emplea `println()` para imprimir la palabra "Sunny" en la línea siguiente.

The screenshot shows the Kotlin playground interface. At the top, there's a navigation bar with links for Solutions, Docs, Community, Teach, Play, and a search icon. Below the navigation bar, there are dropdown menus for version (2.1.20) and platform (JVM). A "Program arguments" input field is present. On the right side of the header, there are "Copy link", "Share code", and a "Run" button. The main area contains the following code:

```
/*
 * Unidad 5
 */

fun main() {
    println("Weather forecast")
    println("sunny")
}
```

Below the code, the output window displays the results of the execution:

```
Weather forecast
sunny
```

Aquí podemos ver cómo se está imprimiendo "Weather forecast". Acto seguido, se invoca una función suspendida llamada printForecast(), la cual introduce una pausa de un segundo, antes de imprimir "Sunny". Simultáneamente o justo después, se llama a la función printTemperature() que imprime "30°C". La función runBlocking asegura que el programa principal espere a que tanto la impresión de "Sunny" (con su retardo) como la impresión de la temperatura se completen antes de finalizar su ejecución.

The screenshot shows the Kotlin playground interface. At the top, there's a navigation bar with links for Solutions, Docs, Community, Teach, Play, and a search icon. Below the navigation bar, there are dropdown menus for version (2.1.20) and platform (JVM). A "Program arguments" input field is present. On the right side of the header, there are "Copy link", "Share code", and a "Run" button. The main area contains the following code:

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        printForecast()
        printTemperature()
    }
}

suspend fun printForecast() {
    delay(1000)
    ...
}
```

Below the code, the output window displays the results of the execution:

```
Weather forecast
sunny
30°C
```

A screenshot of the Kotlin Play IDE interface. The code editor shows a Kotlin script with the following content:

```
2.1.20 ▾ JVM ▾ Program arguments
}
    println("Execution time: ${time / 1000.0} seconds")
}
suspend fun printForecast() {
    delay(1000)
    println("Sunny")
}

suspend fun printTemperature() {
    delay(1000)
    println("30°C")
}

Weather forecast
Sunny
30°C
Execution time: 2.106 seconds
```

The output window displays the results of running the script: "Weather forecast", "Sunny", "30°C", and "Execution time: 2.106 seconds".

Además, también se introducen las funciones suspendidas (`suspend fun`), que permiten que una función se ejecute de manera asíncrona. Estas funciones, como `printForecast` y `printTemperature`, permiten simular tareas que se suspenden y reanudan de forma eficiente.

A screenshot of the Kotlin Play IDE interface. The code editor shows a Kotlin script with the following content:

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        launch {
            printForecast()
        }
        launch {
            printTemperature()
        }
    }
}

Weather forecast
Sunny
30°C
```

The output window displays the results of running the script: "Weather forecast", "Sunny", and "30°C".

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
import kotlinx.coroutines.*
```

```
fun main() {
    runBlocking {
        println("Weather forecast")
        launch {
            printForecast()
        }
        launch {
            printTemperature()
        }
        println("Have a good day!")
    }
}
```

Weather forecast
Have a good day!
Sunny
30°C

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
import kotlinx.coroutines.*
```

```
fun main() {
    runBlocking {
        println("Weather forecast")
        val forecast: Deferred<String> = async {
            getForecast()
        }
        val temperature: Deferred<String> = async {
            getTemperature()
        }
        println("${forecast.await()} ${temperature.await()}")
    }
}
```

Weather forecast
Sunny 30°C
Have a good day!

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

Copy link Share code Run

```
import kotlinx.coroutines.*
```

```
fun main() {
    runBlocking {
        println("Weather forecast")
        try {
            println(getWeatherReport())
        } catch (e: AssertionError) {
            println("Caught exception in runBlocking(): $e")
            println("Report unavailable at this time")
        }
        println("Have a good day!")
    }
}
```

Weather forecast
Caught exception in runBlocking(): java.lang.AssertionError: Temperature is invalid
Report unavailable at this time
Have a good day!

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

```
    }
    suspend fun getForecast(): String {
        delay(1000)
        return "Sunny"
    }
    suspend fun getTemperature(): String {
        delay(500)
        throw AssertionError("Temperature is invalid")
    }
}

Weather forecast
Caught exception java.lang.AssertionError: Temperature is invalid
Sunny { No temperature found }
Have a good day!
```

Copy link Share code Run

Kotlin

Solutions Docs Community Teach Play

2.1.20 JVM Program arguments

```
val temperature = async { getTemperature() }

delay(200)
temperature.cancel()

${forecast.await()}"
}

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

Weather forecast
Sunny
Have a good day!
```

Copy link Share code Run

En este caso se vuelve a hacer uso de corrutinas para ejecutar tareas de manera asíncrona, donde dentro de la función principal (main), se inicia un bloque runBlocking que espera la finalización de todas las corrutinas que se lancen dentro de él. Inmediatamente, se lanza una nueva corrutina utilizando launch. Dentro de esta corrutina, se cambia el contexto de ejecución a un grupo de hilos predeterminado (Dispatchers.Default) utilizando withContext.

The screenshot shows the Kotlin Play IDE interface. At the top, there are tabs for Solutions, Docs, Community, Teach, Play, and a search bar. Below the tabs, there are dropdown menus for version (2.1.20) and JVM. A "Program arguments" input field is present. On the right, there are buttons for Copy link, Share code, and Run. The main area displays the following Kotlin code:

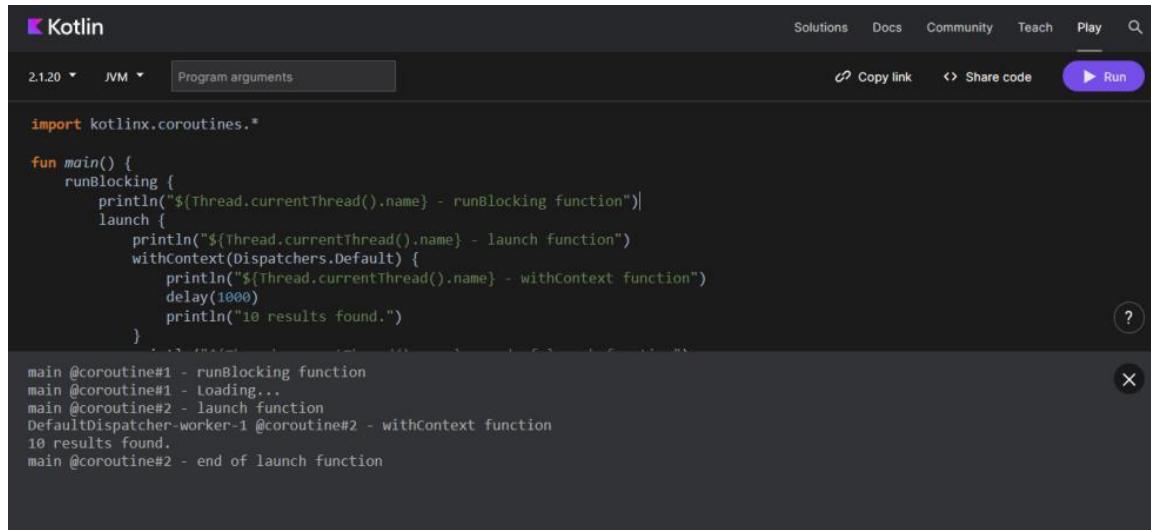
```
import kotlinx.coroutines.*  
  
fun main() {  
    runBlocking {  
        launch {  
            withContext(Dispatchers.Default) {  
                delay(1000)  
                println("10 results found.")  
            }  
        }  
        println("Loading...")  
    }  
}  
  
Loading...  
10 results found.
```

The output window shows the text "Loading..." followed by "10 results found.". There are also a question mark icon and a close button in the output window.

Resumen: Corrutinas en Kotlin

Este programa en Kotlin demuestra cómo lanzar y controlar **corrutinas**.

- Se inicia con un bloque runBlocking, que **espera a que terminen todas las corrutinas internas**. Ahí se imprime un mensaje con el nombre del hilo principal.
- Luego, se lanza una corrutina con launch, que también imprime el hilo en el que se ejecuta.
- Dentro de esa corrutina, se usa withContext(Dispatchers.Default) para cambiar el contexto a un grupo de hilos optimizado para tareas intensivas. Se imprime el hilo actual y se hace una **pausa de un segundo**.
- Después de la pausa, se muestra el mensaje: "10 resultados encontrados".
- Al finalizar la corrutina, se imprime un mensaje marcando su **fin**.

A screenshot of the Kotlin Play IDE interface. At the top, there's a navigation bar with links for Solutions, Docs, Community, Teach, Play, and a search icon. Below the navigation bar, there are dropdown menus for '2.1.20' and 'JVM', and a 'Program arguments' input field. To the right of these are 'Copy link', 'Share code', and a 'Run' button. The main area shows a piece of Kotlin code and its execution output. The code imports `kotlinx.coroutines.*` and defines a `main` function. Inside `main`, it uses `runBlocking` to print the current thread name, then launches a new coroutine using `launch`. This coroutine prints the current thread name again, sets the context to `Dispatchers.Default`, delays for 1000ms, and prints "10 results found.". The output window shows the execution of the code: it prints "main @coroutine#1 - runBlocking function", "main @coroutine#1 - Loading...", "main @coroutine#2 - launch function", "DefaultDispatcher-worker-1 @coroutine#2 - withContext function", "10 results found.", and finally "main @coroutine#2 - end of launch function".

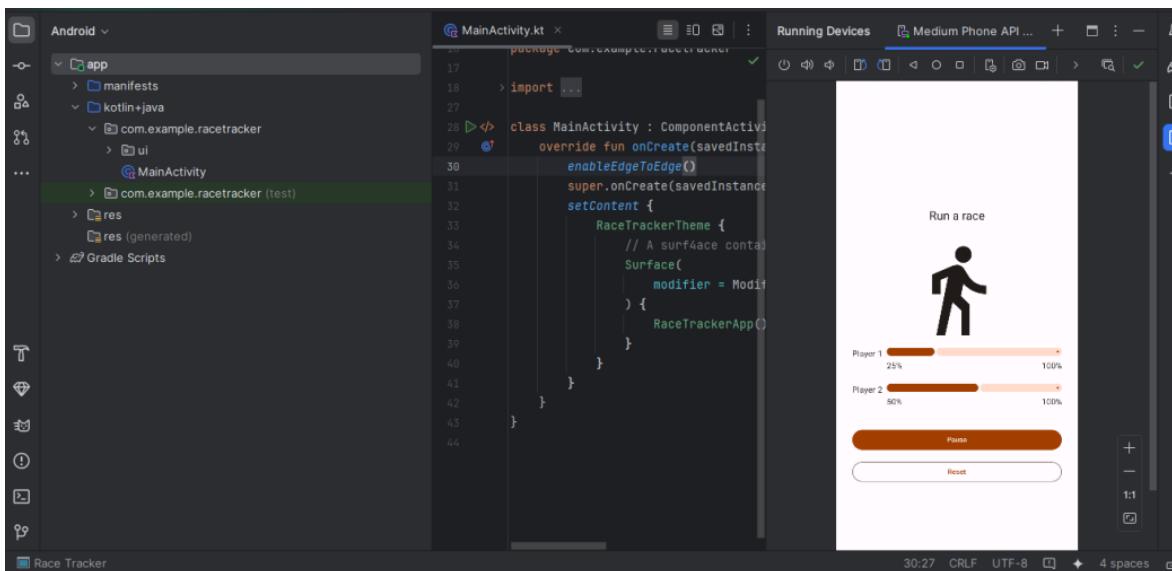
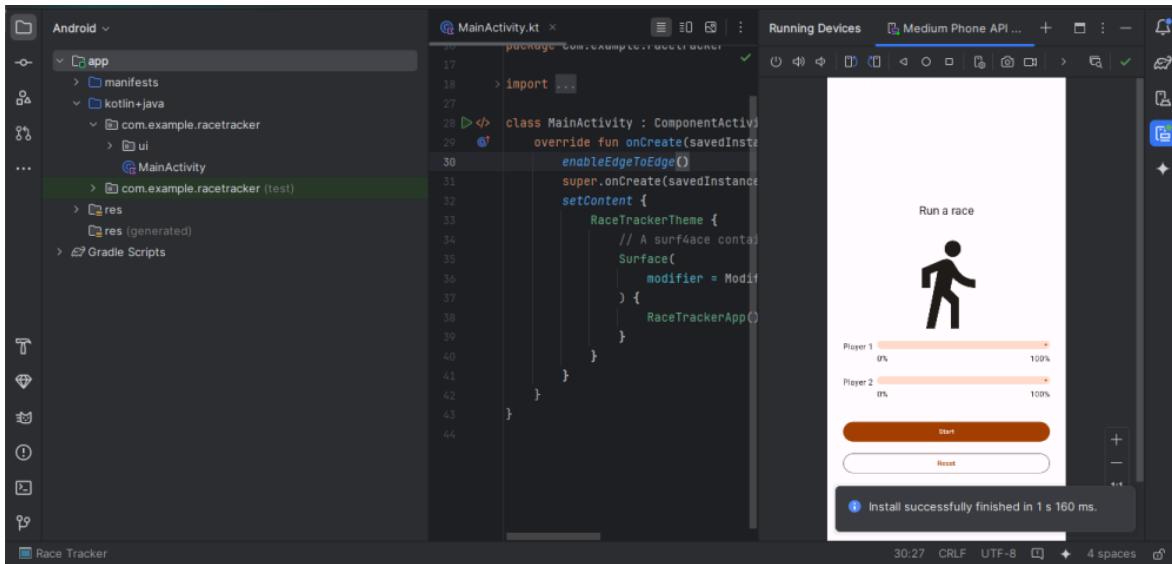
```
import kotlinx.coroutines.*  
  
fun main() {  
    runBlocking {  
        println("${Thread.currentThread().name} - runBlocking function")  
        launch {  
            println("${Thread.currentThread().name} - launch function")  
            withContext(Dispatchers.Default) {  
                println("${Thread.currentThread().name} - withContext function")  
                delay(1000)  
                println("10 results found.")  
            }  
        }  
    }  
}  
  
main @coroutine#1 - runBlocking function  
main @coroutine#1 - Loading...  
main @coroutine#2 - launch function  
DefaultDispatcher-worker-1 @coroutine#2 - withContext function  
10 results found.  
main @coroutine#2 - end of launch function
```

Este codelab introduce el uso de **corrutinas en Kotlin** dentro de una aplicación que simula una carrera entre dos jugadores. La interfaz incluye botones de inicio/pausa y reinicio, además de barras de progreso que reflejan el avance de cada corredor. El objetivo principal es ejecutar tareas concurrentes sin bloquear el hilo principal, asegurando una experiencia fluida para el usuario.

La clase `RaceParticipant` define el comportamiento individual de cada jugador, incluyendo pausas que simulan su velocidad y actualizaciones de progreso. La función clave es `run()`, una **función suspendida** que permite simular el avance sin interferir con la UI.

En el archivo `RaceTrackerApp.kt`, se usa **LaunchedEffect** para iniciar las corrutinas cuando comienza la carrera. Dentro de este efecto, se lanza una corriputina para cada jugador utilizando `launch`, permitiendo que ambos se ejecuten de manera concurrente. Para sincronizar la finalización de ambas corridas antes de actualizar el estado de la interfaz (como el texto del botón), se utiliza **coroutineScope**.

Además, el codelab enseña cómo manejar la **cancelación de corrutinas**, una habilidad importante cuando el usuario interactúa con los botones de pausa o reinicio, garantizando que no haya procesos colgando o conflictos de estado.



Cómo obtener datos de Internet

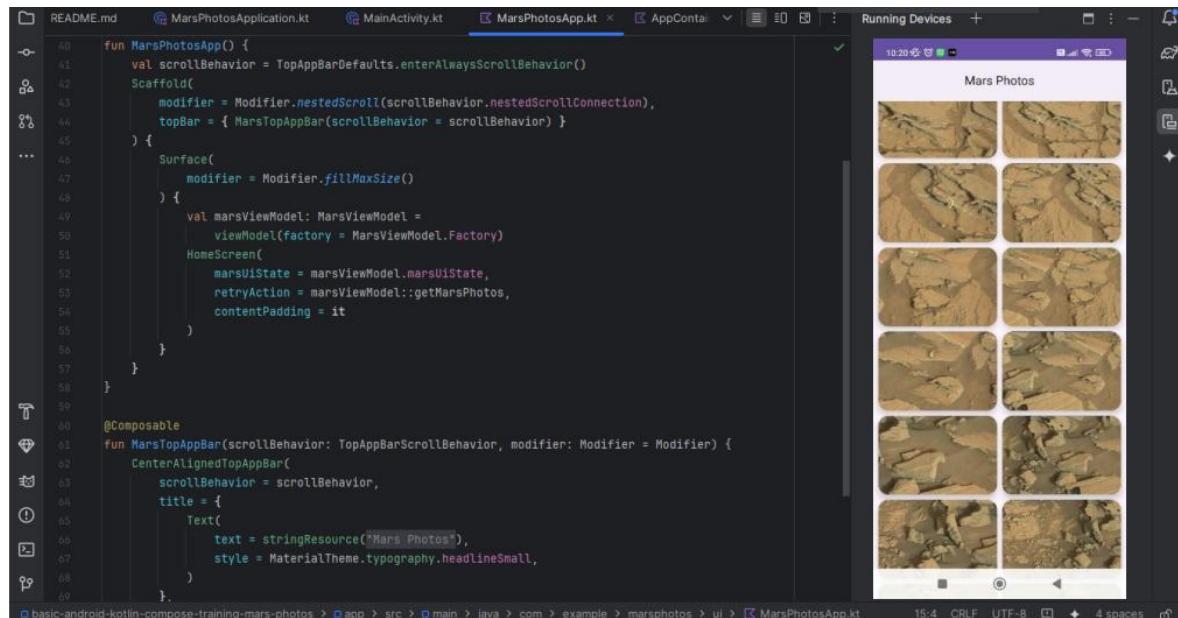
En este codelab se desarrolla la app **Mars Photos**, que muestra imágenes reales de Marte obtenidas mediante una solicitud **HTTP GET** a un servicio web RESTful de la NASA, todo con una interfaz construida en **Jetpack Compose**.

Se implementa una **capa de datos** usando la biblioteca **Retrofit** para facilitar la conexión HTTP. Para manejar las respuestas en formato **JSON**, se utiliza **kotlinx.serialization**, lo que permite convertir los datos en objetos Kotlin.

Un **ViewModel** realiza las llamadas de red y actualiza el estado de la UI cuando llegan nuevas fotos. También se siguen buenas prácticas como:

- Ejecutar operaciones en segundo plano.
- Manejar excepciones de red para informar problemas de conectividad.

Finalmente, se otorgan los permisos necesarios para acceder a Internet y se muestra la cantidad de imágenes recibidas como prueba del funcionamiento.



The screenshot shows the Android Studio interface with the MarsPhotosApplication.kt file open in the code editor. The code defines a MarsPhotosApp class that contains a MarsPhotosApp() function. This function sets up a scroll behavior, a scaffold with a top bar, and a surface that displays a HomeScreen. The HomeScreen uses a MarsViewModel to get Mars photos and displays them in a grid. A MarsTopAppBar is also defined to handle the scroll behavior. To the right of the code editor is a preview window showing a 4x3 grid of Mars surface images.

```

40 fun MarsPhotosApp() {
41     val scrollBehavior = TopAppBarDefaults.enterAlwaysScrollBehavior()
42     Scaffold(
43         modifier = Modifier.nestedScroll(scrollBehavior.nestedScrollConnection),
44         topBar = { MarsTopAppBar(scrollBehavior = scrollBehavior) }
45     ) {
46         Surface(
47             modifier = Modifier.fillMaxSize()
48         ) {
49             val marsViewModel: MarsViewModel =
50                 viewModel(factory = MarsViewModel.Factory)
51             HomeScreen(
52                 marsUiState = marsViewModel.marsUiState,
53                 retryAction = marsViewModel::getMarsPhotos,
54                 contentPadding = it
55             )
56         }
57     }
58 }
59
60 @Composable
61 fun MarsTopAppBar(scrollBehavior: TopAppBarScrollBehavior, modifier: Modifier = Modifier) {
62     CenterAlignedTopAppBar(
63         scrollBehavior = scrollBehavior,
64         title = {
65             Text(
66                 text = stringResource("Mars Photos"),
67                 style = MaterialTheme.typography.headlineSmall,
68             )
69         }
70     )
71 }

```

Ruta de Aprendizaje 2 (Cómo cargar y mostrar imágenes de Internet)

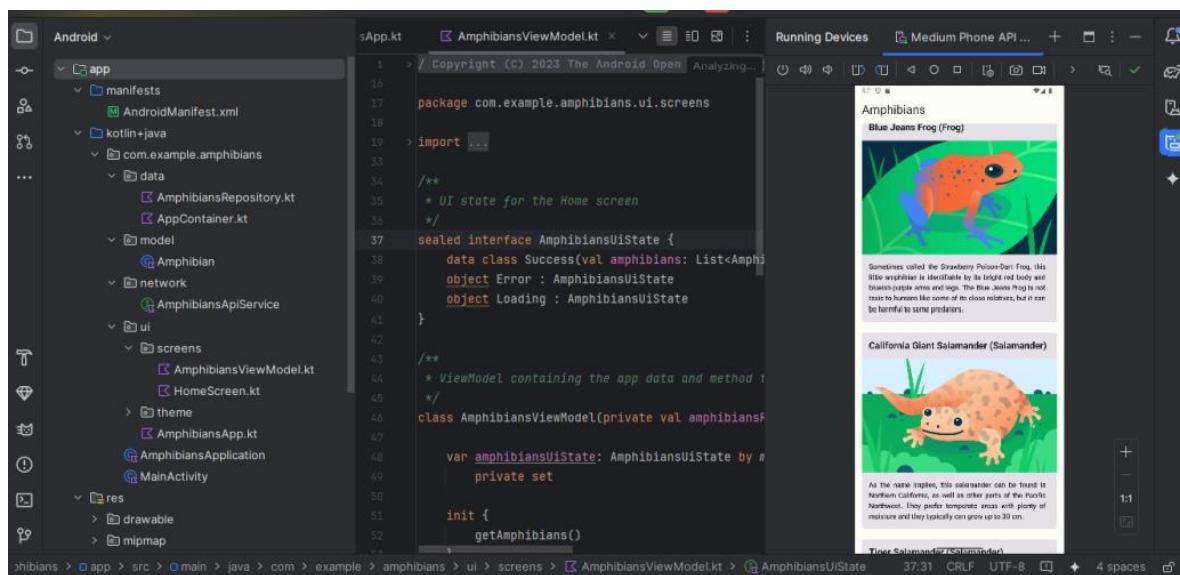
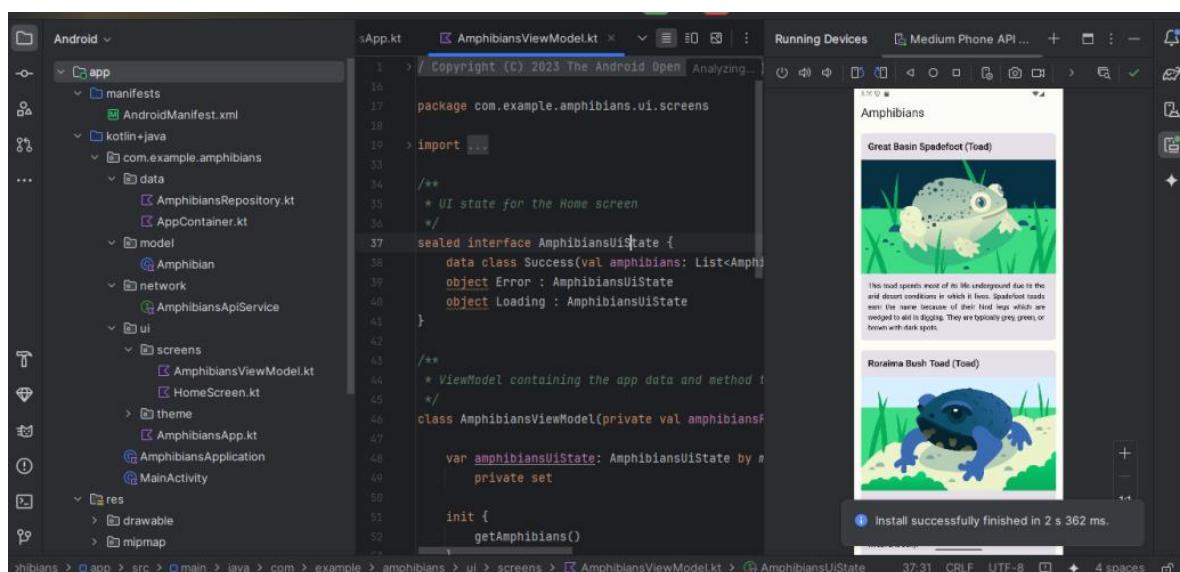
Cómo agregar el repositorio y la inserción manual de dependencias

En esta práctica se desarrolla una app que **recupera y muestra información sobre anfibios** desde una **API web**, incluyendo su nombre, tipo, descripción e imagen. La lista de anfibios se presenta en una **interfaz desplazable** usando Jetpack Compose.

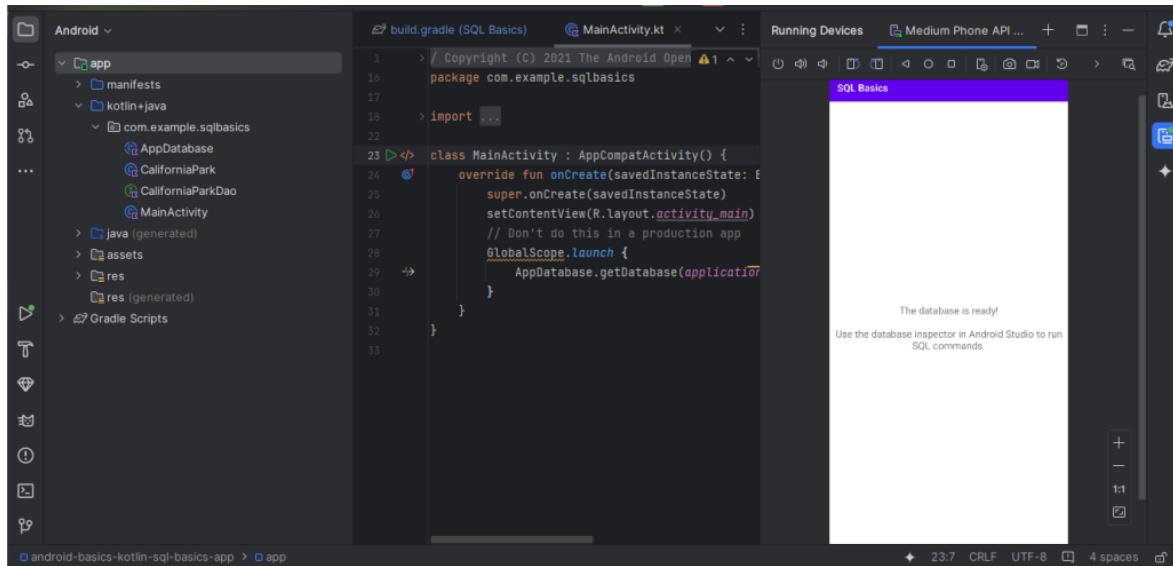
El desarrollo se organiza en **capas**, aplicando buenas prácticas de arquitectura:

- **Capa de datos:**
 - Amphibian.kt: modelo de datos.
 - Amphibian ApiService.kt: usa **Retrofit** para hacer solicitudes HTTP.
 - Se usa **kotlinx.serialization** para analizar las respuestas JSON.
 - Un **repositorio** gestiona el acceso a los datos.
- **Interfaz de usuario:**

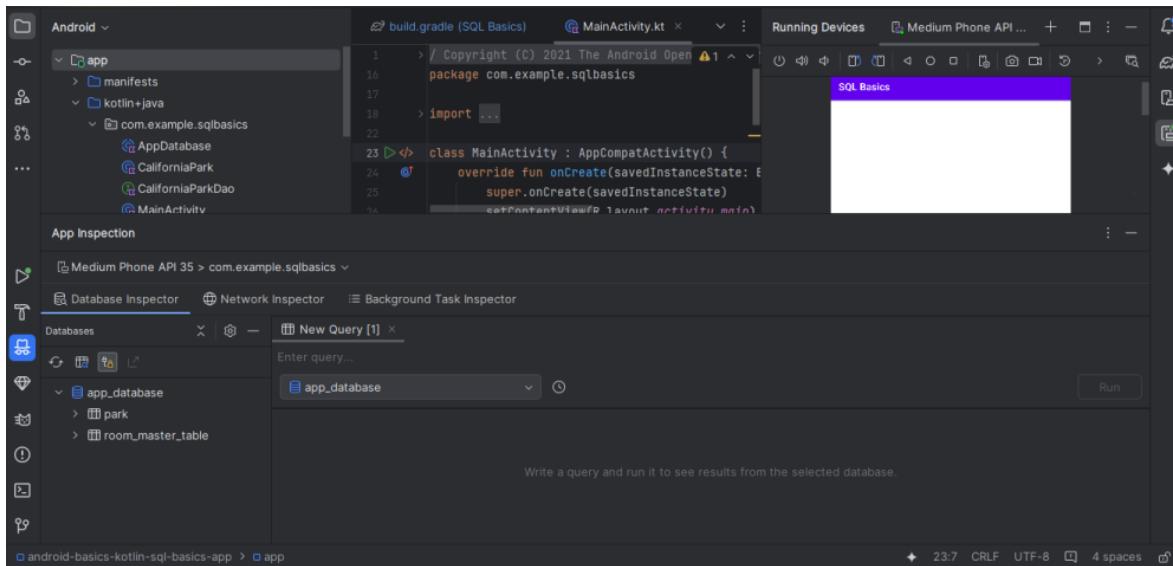
- MainActivity.kt: inicia la UI con funciones componibles.
 - AmphibianCard.kt: muestra cada anfibio en una tarjeta con su imagen e información.
 - AmphibianListScreen.kt: organiza la lista completa de tarjetas.
- Gestión de estado y lógica:
- AmphibianViewModel.kt: maneja la lógica de negocio y llama a la API.
 - AmphibianUiState.kt: representa estados como cargando, error o datos cargados.



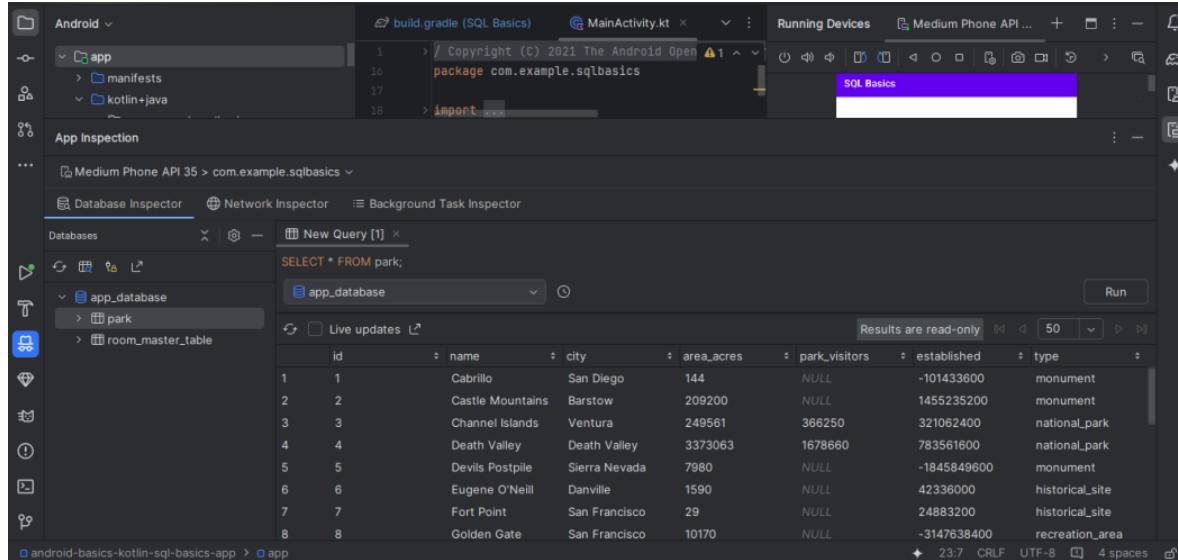
Unidad 6: Persistencia de Datos Ruta de Aprendizaje 1 (Introducción a SQL)



Posteriormente nos dirigimos a App Inspection con la pestaña Database Inspector, donde el proyecto actual está configurado para usar una base de datos diferente a la del ejemplo del curso.



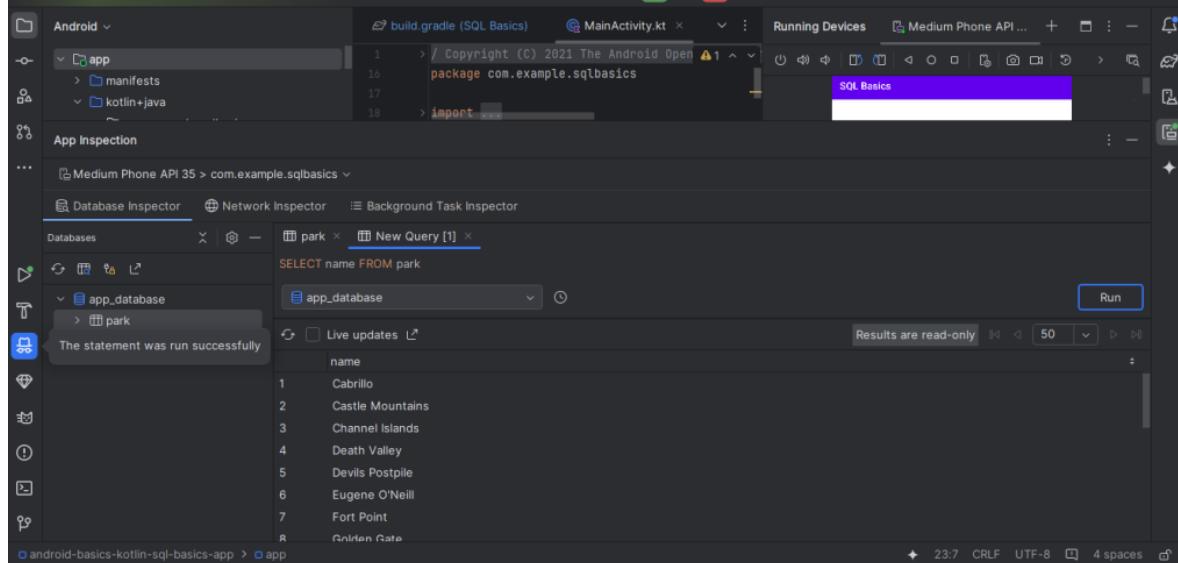
Hacemos uso de la instrucción SELECT de SQL y vemos que se muestra toda la tabla de park (en este caso):



The screenshot shows the Android Studio interface with the Database Inspector tool open. The query `SELECT * FROM park;` is run against the `app_database`. The results are displayed in a table:

	id	name	city	area_acres	park_visitors	established	type
1	1	Cabrillo	San Diego	144	NULL	-101433600	monument
2	2	Castle Mountains	Barstow	209200	NULL	1455235200	monument
3	3	Channel Islands	Ventura	249561	366250	321062400	national_park
4	4	Death Valley	Death Valley	3373063	1678660	783561600	national_park
5	5	Devils Postpile	Sierra Nevada	7980	NULL	-1845849600	monument
6	6	Eugene O'Neill	Danville	1590	NULL	42336000	historical_site
7	7	Fort Point	San Francisco	29	NULL	24883200	historical_site
8	8	Golden Gate	San Francisco	10170	NULL	-3147638400	recreation_area

Ahora seleccionamos solo una columna. En el curso es SELECT subject FROM email; en nuestro caso es SELECT name FROM park



The screenshot shows the Android Studio interface with the Database Inspector tool open. The query `SELECT name FROM park;` is run against the `app_database`. The results are displayed in a table:

	name
1	Cabrillo
2	Castle Mountains
3	Channel Islands
4	Death Valley
5	Devils Postpile
6	Eugene O'Neill
7	Fort Point
8	Golden Gate

A message in the bottom left corner says "The statement was run successfully".

Y para seleccionar varias columnas en el ejemplo es SELECT subject, sender FROM email; y en el nuestro es SELECT name, city FROM park

```
build.gradle (SQL Basics) MainActivity.kt
```

```
1 > / Copyright (C) 2021 The Android Open
16 package com.example.sqlbasics
17
18 > import ...
```

The statement was run successfully

	name	city
1	Cabrillo	San Diego
2	Castle Mountains	Barstow
3	Channel Islands	Ventura
4	Death Valley	Death Valley
5	Devils Postpile	Sierra Nevada
6	Eugene O'Neill	Danville
7	Fort Point	San Francisco
8	Golden Gate	San Franiern

Para contar la cantidad de parques en la base de datos usamos `SELECT COUNT(*)` FROM park:

```
build.gradle (SQL Basics) MainActivity.kt
```

```
1 > / Copyright (C) 2021 The Android Open
16 package com.example.sqlbasics
17
18 > import ...
```

The statement was run successfully

	COUNT(*)
1	23

Para obtener el área más grande de los parques (usando MAX) en caso de que se quiera saber cuál es el parque con el área más grande, se usa MAX() sobre la columna area_acres:

```
build.gradle (SQL Basics) MainActivity.kt
1 > / Copyright (C) 2021 The Android Open
16 package com.example.sqlbasics
17
18 > import ...

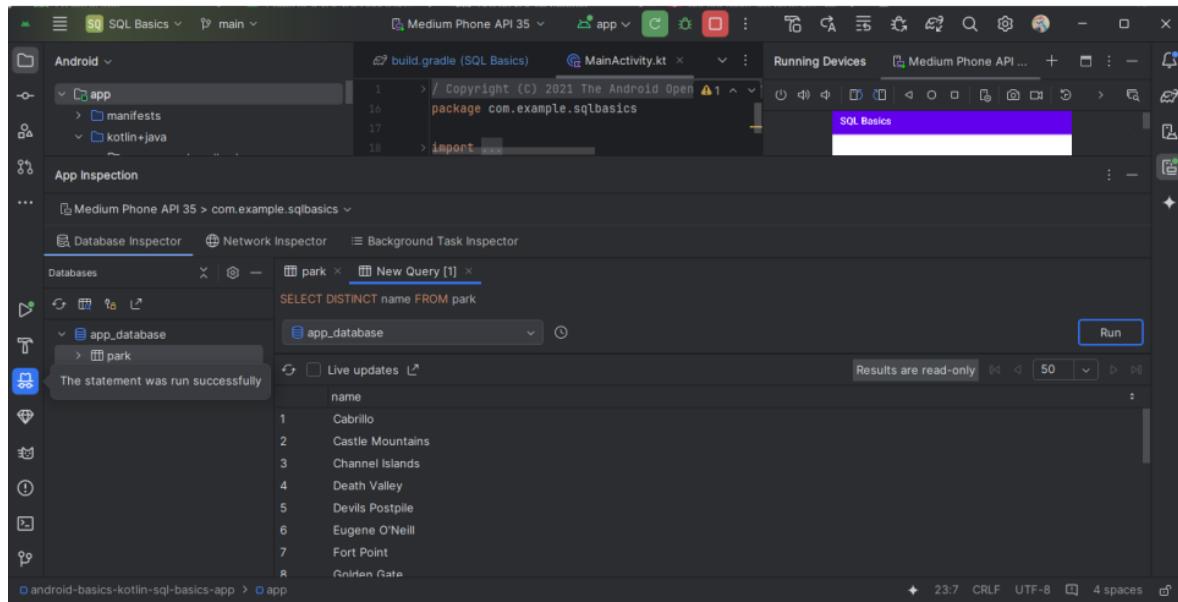
SELECT MAX(area_acres) FROM park
app_database
Run
Results are read-only 50
MAX(area_acres)
1 3373063
```

Ahora para filtrar los resultados duplicados con DISTINCT en el ejemplo original, la consulta SELECT sender FROM email; devuelve todos los valores de la columna sender (aunque haya duplicados). Para adaptarlo a la tabla park se selecciona la columna name.

```
build.gradle (SQL Basics) MainActivity.kt
1 > / Copyright (C) 2021 The Android Open
16 package com.example.sqlbasics
17
18 > import ...

SELECT name FROM park
app_database
Run
Results are read-only 50
name
1 Cabrillo
2 Castle Mountains
3 Channel Islands
4 Death Valley
5 Devils Postpile
6 Eugene O'Neill
7 Fort Point
R Golden Gate
```

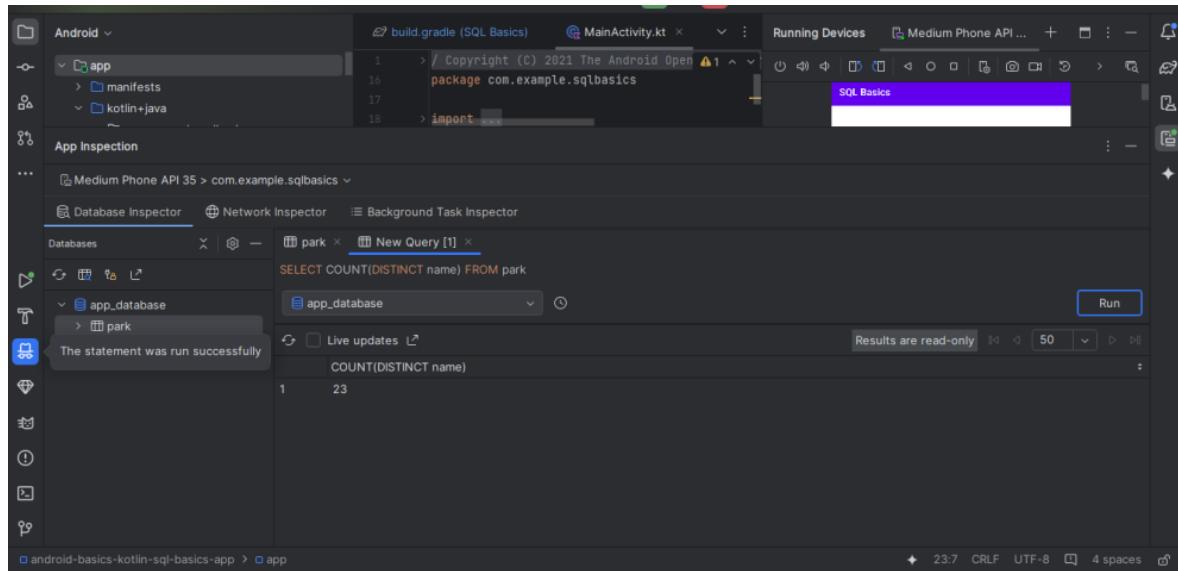
Ahora, en el ejemplo original, SELECT DISTINCT sender FROM email; selecciona los valores únicos de la columna sender. Como nosotros contamos con la tabla park y queremos ver los nombres de parques únicos (sin duplicados) se ejecuta lo siguiente:



The screenshot shows the Android Studio interface with the Database Inspector tool open. The left sidebar shows the project structure with the 'app' module selected. In the Database Inspector, the 'park' table is selected under the 'app_database' database. A query window displays the SQL command: 'SELECT DISTINCT name FROM park'. The results table shows 8 distinct names: Cabrillo, Castle Mountains, Channel Islands, Death Valley, Devils Postpile, Eugene O'Neill, Fort Point, and Golden Gate.

name
Cabrillo
Castle Mountains
Channel Islands
Death Valley
Devils Postpile
Eugene O'Neill
Fort Point
Golden Gate

En el ejemplo original, `SELECT COUNT(DISTINCT sender) FROM email;` cuenta cuántos remitentes únicos (sender) hay en la tabla email. Para la tabla park contamos cuántos nombres únicos de parques hay



The screenshot shows the Android Studio interface with the Database Inspector tool open. The left sidebar shows the project structure with the 'app' module selected. In the Database Inspector, the 'park' table is selected under the 'app_database' database. A query window displays the SQL command: 'SELECT COUNT(DISTINCT name) FROM park'. The results table shows a single row with the value 23.

COUNT(DISTINCT name)
23

Un ejemplo básico con WHERE para la tabla park, si queremos filtrar los parques que están en una ciudad específica (por ejemplo, "San Diego"), la consulta se vería así: `SELECT * FROM park WHERE city = 'San Diego'`

The screenshot shows the Android Studio interface with the Database Inspector tool open. A query has been run against the 'park' table in the 'app_database'. The results show one row for Cabrillo National Monument in San Diego.

```
WHERE city = 'San Diego'
```

	id	name	city	area_acres	park_visitors	established	type
1	1	Cabrillo	San Diego	144	NULL	-101433600	monument

En cambio, para filtrar con varias condiciones usando AND en el ejemplo original se mostró cómo filtrar los correos electrónicos que estén en la carpeta "inbox" y que no hayan sido leídos, utilizando AND.

The screenshot shows the Android Studio interface with the Database Inspector tool open. A query has been run against the 'park' table in the 'app_database'. The results show one row for Channel Islands National Park in Ventura, which has more than 1000 visitors.

```
SELECT * FROM park  
WHERE city = 'Ventura' AND park_visitors > 1000
```

	id	name	city	area_acres	park_visitors	established	type
1	3	Channel Islands	Ventura	249561	366250	321062400	national_park

Para filtrar con OR filtramos los parques que están en "Palms" o en "San Francisco":

The screenshot shows the Android Studio interface with the Database Inspector open. A query is running:

```
SELECT * FROM park  
WHERE city = 'Palms' OR city = 'San Francisco'
```

The results table shows data from the 'park' table:

	id	name	city	area_acres	park_visitors	established	type
	1	Fort Point	San Francisco	29	NULL	24883200	historical_site
	2	Golden Gate	San Francisco	10170	NULL	-3147638400	recreation_area
	3	Joshua Tree	Palms	790636	2942381	783561600	national_park
	4	San Francisco Mariti	San Francisco	50	4223542	583372800	national_park

Si queremos filtrar con NOT para seleccionar los parques que no están en "San Francisco":

The screenshot shows the Android Studio interface with the Database Inspector open. A query is running:

```
SELECT * FROM park  
WHERE NOT city = 'San Francisco'
```

The results table shows data from the 'park' table, excluding the row for San Francisco:

	id	name	city	area_acres	park_visitors	established	type
	1	Cabrillo	San Diego	144	NULL	-101433600	monument
	2	Castle Mountains	Barstow	209200	NULL	1455235200	monument
	3	Channel Islands	Ventura	249561	366250	321062400	national_park
	4	Death Valley	Death Valley	3373063	1678660	783561600	national_park
	5	Devils Postpile	Sierra Nevada	7980	NULL	-1845849600	monument
	6	Eugene O'Neill	Darville	1590	NULL	42336000	historical_site
	7	Inyo Muir	Martinez	3450	NULL	-101413600	historical_site

Para buscar texto con LIKE en este caso buscamos parques cuyo nombre contiene la palabra "Fort".

The screenshot shows the Android Studio interface with the Database Inspector tool open. The query window contains the following SQL code:

```
SELECT * FROM park  
WHERE name LIKE '%Fort%';
```

The results table shows one row of data:

id	name	city	area_acres	park_visitors	established	type
7	Fort Point	San Francisco	29	NULL	24883200	historical_site

Para contar cuántos parques tienen la palabra "fool" en el nombre:

The screenshot shows the Android Studio interface with the Database Inspector tool open. The query window contains the following SQL code:

```
SELECT COUNT(*) FROM park  
WHERE name LIKE '%fool%';
```

The results table shows one row of data:

COUNT(*)
0

Mostrar todos los datos de los parques cuyo nombre termina con "fool":

```
build.gradle (SQL Basics) MainActivity.kt
```

```
Copyright (C) 2021 The Android Open
```

```
package com.example.sqlbasics
```

```
import
```

The statement was run successfully

	id	name	city	area_acres	park_visitors	established	type
Table is empty							

Mostrar los nombres distintos de parques que comienzan con la letra "r":

```
build.gradle (SQL Basics) MainActivity.kt
```

```
Copyright (C) 2021 The Android Open
```

```
package com.example.sqlbasics
```

```
import
```

The statement was run successfully

	name
1	Redwood
2	Rosie the Riveter WWII Home Front