Create and use functions in Kotlin

1. Before you begin

In an earlier codelab, you saw a simple program that printed `Hello, world!`. In the programs you've written so far, you've seen two functions:

- a `main()` function, which is required in every Kotlin program. It is the entry point, or starting point, of the program.
- a `println()` function, which you called from `main()` to output text.

In this codelab, you will learn more about functions.

Functions let you break up your code into reusable pieces, rather than include everything in `main()`. Functions are an essential building block of Android apps and learning how to define and use them is a major step on your journey to become an Android developer.

Prerequisites

- Knowledge of Kotlin programming basics, including variables, and the `println()` and `main()` functions

What you'll learn

- How to define and call your own functions.
- How to return values from a function that you can store in a variable.
- How to define and call functions with multiple parameters.
- How to call functions with named arguments.
- How to set default values for function parameters.

What you'll need

- A web browser with access to Kotlin Playground

2. Define and call a function

Before exploring functions in-depth, let's review some basic terminology.

- Declaring (or defining) a function uses the `fun` keyword and includes code within the curly braces which contains the instructions needed to execute a task.
- Calling a function causes all the code contained in that function to execute.

So far, you've written all your code in the `main()` function. The `main()` function doesn't actually get called anywhere in your code; the Kotlin compiler uses it as a starting point. The `main()` function is intended only to include other code you want to execute, such as calls to the `println()` function.

The `println()` function is part of the Kotlin language. However, you can define your own functions. This allows your code to be reused if you need to call it more than once. Take the following program as an example.

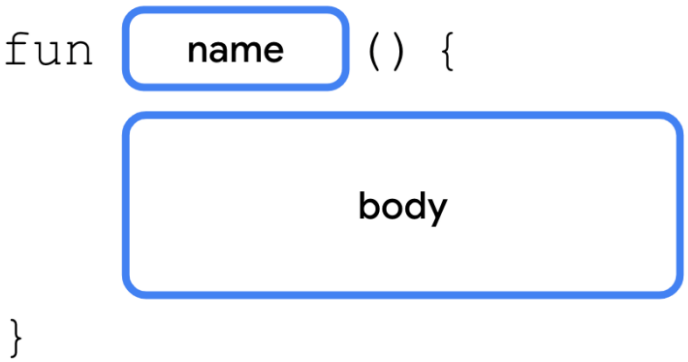
```
fun main() {
    println("Happy Birthday, Rover!")
    println("You are now 5 years old!")
}
```

The `main()` function consists of two `println()` statements—one to wish Rover a happy birthday, and another stating Rover's age.

While Kotlin allows you to put all your code in the `main()` function, you might not always want to. For example, if you also want your program to contain a New Year's greeting, the main function would have to include those calls to `println()` as well. Or perhaps you want to greet Rover multiple times. You could simply copy and paste the code, or you could create a separate function for the birthday greeting. You'll do the latter. Creating separate functions for specific tasks has a number of benefits.

- Reusable code: Rather than copying and pasting code that you need to use more than once, you can simply call a function wherever needed.
- Readability: Ensuring functions do one and only one specific task helps other developers and teammates, as well as your future self to know exactly what a piece of code does.

The syntax for defining a function is shown in the following diagram.



A function definition starts with the `fun` keyword, followed by the name of the function, a set of opening and closing parentheses, and a set of opening and closing curly braces. Contained in curly braces is the code that will run when the function is called.

You'll create a new function to move the two `println()` statements out of the `main()` function.

1. In your browser, open the Kotlin Playground (https://developer.android.com/training/kotlinplayground) and replace the contents with the following code.

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

2. After the `main()` function, define a new function named `birthdayGreeting()`. This function is declared with the same syntax as the `main()` function.

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

fun birthdayGreeting() {

}
```

3. Move the two `println()` statements from `main()` into the curly braces of the `birthdayGreeting()` function.

```
fun main() {

}
```

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

4. In the `main()` function, call the `birthdayGreeting()` function. Your finished code should look as follows:

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

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

5. Run your code. You should see the following output:

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

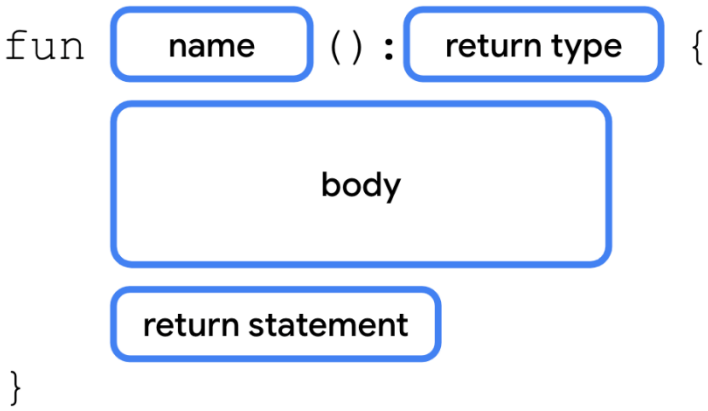
3. Return a value from a function

In more complex apps, functions do more than print text.

Kotlin functions can also generate data called a *return value* which is stored in a variable that you can use elsewhere in your code.

When defining a function, you can specify the data type of the value you want it to return. The return type is specified by placing a colon (`:`) after the parentheses, a single blank space, and then the name of the type (`Int`, `String`, etc). A single space is then placed between the return type and the opening curly brace. Within the function body, after all the statements, you use a return statement to specify the value you want the function to return. A return statement consists of the `return` keyword followed by the value, such as a variable, you want the function to return as an output.

The syntax for declaring a function with a return type is as follows.



The Unit type

By default, if you don't specify a return type, the default return type is `Unit`. `Unit` means the function doesn't return a value. Unit is equivalent to void return types in other languages (`void` in Java and C; `Void`/empty tuple `()` in Swift; `None` in Python, etc.). Any function that doesn't return a value implicitly returns `Unit`. You can observe this by modifying your code to return `Unit`.

1. In the function declaration for `birthdayGreeting()`, add a colon after the closing parenthesis and specify the return type as `Unit`.

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

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

2. Run the code and observe that everything still works.

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

It's optional to specify the `Unit` return type in Kotlin. For functions that don't return anything, or returning `Unit`, you don't need a return statement.

Note: You'll see the `Unit` type again when you learn about a Kotlin feature called lambdas in a later codelab.

Return a String from `birthdayGreeting()`

To demonstrate how a function can return a value, you'll modify the `birthdayGreeting()` function to return a string, rather than simply print the result.

1. Replace the `Unit` return type with `String`.

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

2. Run your code. You'll get an error. If you declare a return type for a function (e.g., `String`), that function must include a `return` statement.

```
A 'return' expression required in a function with a block body
('{...}')
```

3. You can only return one string from a function, not two. Replace the `println()` statements with two variables, `nameGreeting` and `ageGreeting`, using the `val` keyword. Because you removed the calls to `println()` from `birthdayGreeting()`, calling `birthdayGreeting()` won't print anything.

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

4. Using the string formatting syntax you learned in an earlier codelab, add a `return` statement to return a string from the function consisting of both greetings.

In order to format the greetings on a separate line, you also need to use the `\n` escape character. This is just like the `\"` escape character you learned about in a previous codelab. The `\n` character is replaced for a newline so that the two greetings are each on a separate line.

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

5. In `main()`, because `birthdayGreeting()` returns a value, you can store the result in a string variable. Declare a `greeting` variable using `val` to store the result of calling `birthdayGreeting()`.

```
fun main() {
    val greeting = birthdayGreeting()
}
```

6. In `main()`, call `println()` to print the `greeting` string. The `main()` function should now look as follows.

```
fun main() {
   val greeting = birthdayGreeting()
   println(greeting)
}
```

7. Run your code and then observe that the result is the same as before. Returning a value lets you store the result in a variable, but what do you think happens if you call the `birthdayGreeting()` function inside the `println()` function?

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

8. Remove the variable and then pass the result of calling the `birthdayGreeting()` function into the `println()` function:

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

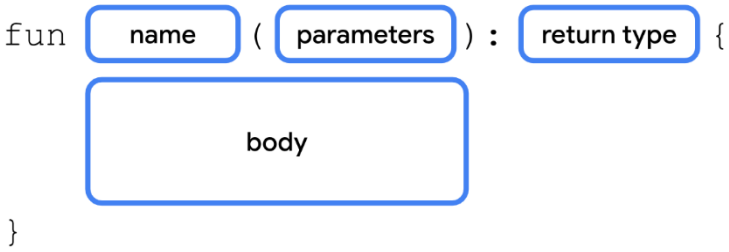
9. Run your code and observe the output. The return value of calling `birthdayGreeting()` is passed directly into `println()`.

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

4. Add a parameter to the `birthdayGreeting()` function

As you've seen, when you call `println()`, you can include a string within the parentheses or pass a value to the function. You can do the same with your `birthdayGreeting()` function. However, you first need to add a parameter to `birthdayGreeting()`.

A parameter specifies the name of a variable and a data type that you can pass into a function as data to be accessed inside the function. Parameters are declared within the parentheses after the function name.



Each parameter consists of a variable name and data type, separated by a colon and a space. Multiple parameters are separated by a comma.

Right now, the `birthdayGreeting()` function can only be used to greet Rover. You'll add a parameter to the `birthdayGreeting()` function so that you can greet any name that you pass into the function.

1. Within the parentheses of the `birthdayGreeting()` function, add a `name` parameter of type `String`, using the syntax `name: String`.

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

The parameter defined in the previous step works like a variable declared with the `val` keyword. Its value can be used anywhere in the `birthdayGreeting()` function. In an earlier codelab, you learned about how you can insert the value of a variable into a string.

2. Replace `Rover` in the `nameGreeting` string with the `$` symbol followed by the `name` parameter.

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

3. Run your code and observe the error. Now that you declared the `name` parameter, you need to pass in a `String` when you call `birthdayGreeting()`. When you call a function that takes a parameter, you pass an argument to the function. The argument is the value that you pass, such as `"Rover"`.

```
No value passed for parameter 'name'
```

4. Pass `"Rover"` into the `birthdayGreeting()` call in `main()`.

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

5. Run your code and observe the output. The name Rover comes from the `name` parameter.

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

6. Because `birthdayGreeting()` takes a parameter, you can call it with a name other than Rover. Add another call to `birthdayGreeting()` inside the call to `println()`, passing in the argument `"Rex"`.

```
println(birthdayGreeting("Rover"))
println(birthdayGreeting("Rex"))
```

7. Run the code again and then observe that the output differs based on the argument passed into `birthdayGreeting()`.

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

Note: Although you often find them used interchangeably, a parameter and an argument aren't the same thing. When you define a function, you define the parameters that are to be passed to it when the function is called. When you call a function, you pass arguments for the parameters. Parameters are the variables accessible to the function, such as a `name` variable, while arguments are the actual values that you pass, such as the `"Rover"` string.

Warning: Unlike in some languages, such as Java, where a function can change the value passed into a parameter, parameters in Kotlin are immutable. You cannot reassign the value of a parameter from within the function body.

5. Functions with multiple parameters

Previously, you added a parameter to change the greeting based on the name. However, you can also define more than one parameter for a function, even

parameters of different data types. In this section, you'll modify the greeting so that it also changes based on the dog's age.

Parameter definitions are separated by commas. Similarly, when you call a function with multiple parameters, you separate the arguments passed in with commas as well. Let's see this in action.

1. After the `name` parameter, add an `age` parameter of type `Int`, to the `birthdayGreeting()` function. The new function declaration should have the two parameters, `name` and `age`, separated by a comma:

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

2. The new greeting string should use the `age` parameter. Update the `birthdayGreeting()` function to use the value of the `age` parameter in the `ageGreeting` string.

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

3. Run the function and then notice the errors in the output:

```
No value passed for parameter 'age'
No value passed for parameter 'age'
```

4. Modify the two calls to the `birthdayGreeting()` function in `main()` to pass in a different age for each dog. Pass in `5` for Rover's age and `2` for Rex's age.

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

5. Run your code. Now that you passed in values for both parameters, the output should reflect the name and age of each dog when you call the function.

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

Function Signature

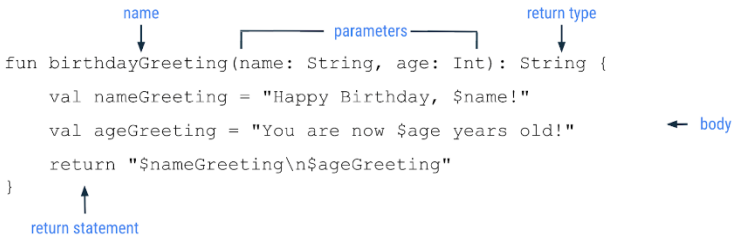
So far you've seen how to define the function name, inputs (parameters), and outputs. The function name with its inputs (parameters) are collectively known as the *function signature*. The function signature consists of everything before the return type and is shown in the following code snippet.

```
fun birthdayGreeting(name: String, age: Int)
```

The parameters, separated by commas, are sometimes called the parameter list.

You'll often see these terms in documentation for code written by other developers. The function signature tells you the name of the function and what data types can be passed in.

You've learned a lot of new syntax around defining functions. Take a look at the following diagram for a recap of function syntax.



6. Named arguments

In the previous examples, you didn't need to specify the parameter names, `name` or `age`, when you called a function. However, you're able to do so if you choose. For example, you may call a function with many parameters or you may want to pass your arguments in a different order, such as putting the `age` parameter before the `name` parameter. When you include the parameter name when you call a function, it's called a *named argument* (https://kotlinlang.org/docs/functions.html#named-arguments). Try using a named argument with the `birthdayGreeting()` function.

1. Modify the call for Rex to use named arguments as shown in this code snippet. You can do this by including the parameter name followed by an equal sign, and then the value (e.g. `name = "Rex"`).

```
println(birthdayGreeting(name = "Rex", age = 2))
```

2. Run the code and then observe that the output is unchanged:

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

3. Reorder the named arguments. For example, put the `age` named argument before the `name` named argument.

```
println(birthdayGreeting(age = 2, name = "Rex"))
```

4. Run the code and observe that the output remains the same. Even though you changed the order of the arguments, the same values are passed in for the same parameters.

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

7. Default arguments

Function parameters can also specify default arguments. Maybe Rover is your favorite dog, or you expect a function to be called with specific arguments in most cases. When you call a function, you can choose to omit arguments for which there is a default, in which case, the default is used.

To add a default argument, you add the assignment operator (=) after the data type for the parameter and set it equal to a value. Modify your code to use a default argument.

1. In the `birthdayGreeting()` function, set the `name` parameter to the default value `"Rover"`.

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

2. In the first call to `birthdayGreeting()` for Rover in `main()`, set the `age` named argument to `5`. Because the `age` parameter is defined after the `name`,

you need to use the named argument `age`. Without named arguments, Kotlin assumes the order of arguments is the same order in which parameters are defined. The named argument is used to ensure Kotlin is expecting an `Int` for the `age` parameter.

```
println(birthdayGreeting(age = 5))
println(birthdayGreeting("Rex", 2))
```

3. Run your code. The first call to the `birthdayGreeting()` function prints "Rover" as the name because you never specified the name. The second call to `birthdayGreeting()` still uses the `Rex` value, which you passed in for the `name`.

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

4. Remove the name from the second call to the `birthdayGreeting()` function. Again, because `name` is omitted, you need to use a named argument for the age.

```
println(birthdayGreeting(age = 5))
println(birthdayGreeting(age = 2))
```

5. Run your code and then observe that now both calls to `birthdayGreeting()` print "Rover" as the name because no name argument is passed in.

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

8. Conclusion

Congratulations! You learned how to define and call functions in Kotlin.

Summary

- Functions are defined with the `fun` keyword and contain reusable pieces of code.
- Functions help make larger programs easier to maintain and prevent the unnecessary repetition of code.
- Functions can return a value that you can store in a variable for later use.

- Functions can take parameters, which are variables available inside a function body.
- Arguments are the values that you pass in when you call a function.
- You can name arguments when you call a function. When you use named arguments, you can reorder the arguments without affecting the output.
- You can specify a default argument that lets you omit the argument when you call a function.