Kotlin Introduction

What is Kotlin?

Kotlin is a modern, trending programming language that was released in 2016 by JetBrains.

It has become very popular since it is compatible with Java (one of the most popular programming languages out there), which means that Java code (and libraries) can be used in Kotlin programs.

Kotlin is used for:

- Mobile applications (specially Android apps)
- Web development
- Server side applications
- Data science
- And much, much more!

Why Use Kotlin?

- Kotlin is fully compatible with Java
- Kotlin works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
- Kotlin is concise and safe
- Kotlin is easy to learn, especially if you already know Java
- Kotlin is free to use
- Big community/support

Get Started

This tutorial will teach you the very basics of Kotlin.

It is not necessary to have any prior programming experience.

Kotlin IDE

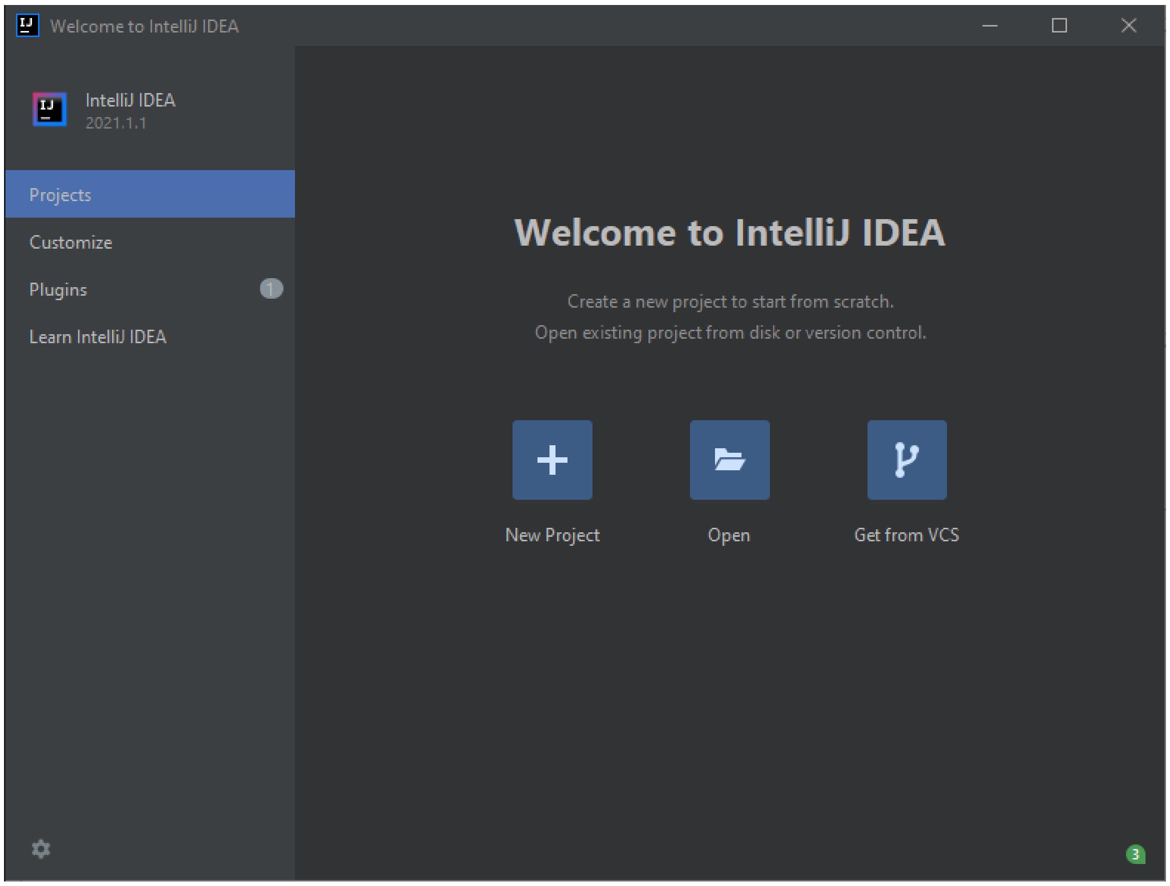
The easiest way to get started with Kotlin, is to use an IDE.

An IDE (Integrated Development Environment) is used to edit and compile code.

In this chapter, we will use IntelliJ (developed by the same people that created Kotlin) which is free to download from https://www.jetbrains.com/idea/download/.

Kotlin Install

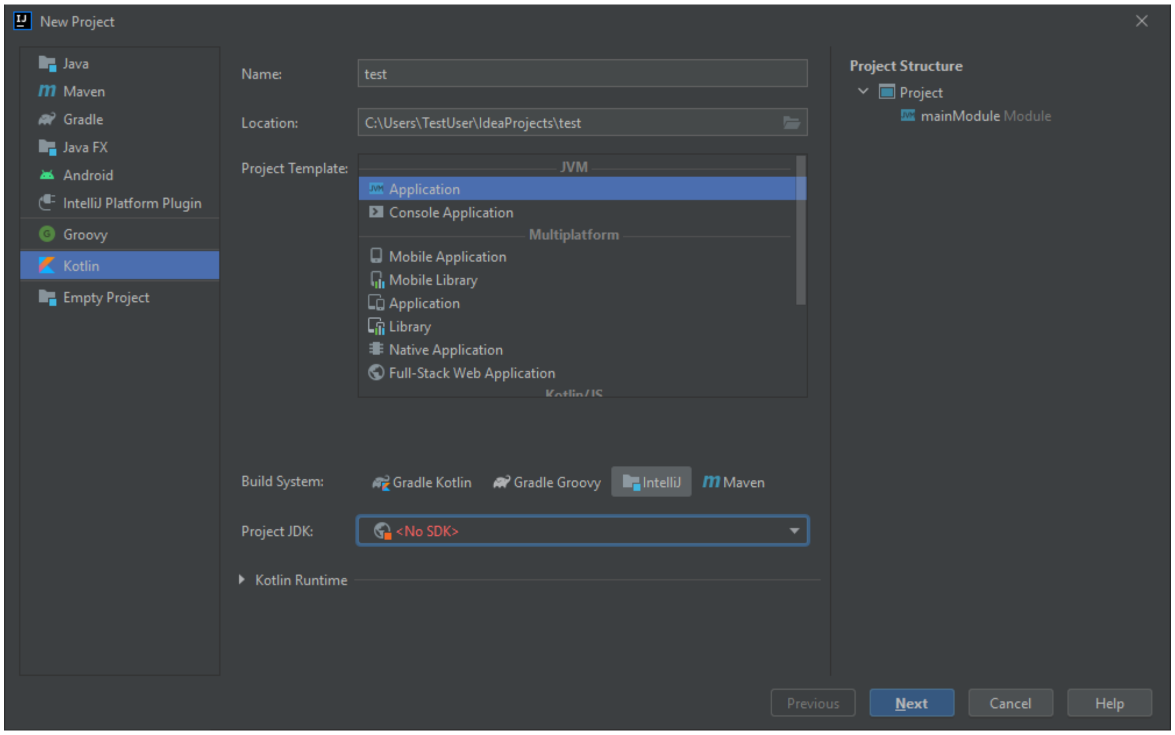
Once IntelliJ is downloaded and installed, click on the **New Project** button to get started with IntelliJ:



Welcome to IntelliJ IDEA

Create a new project to start from scratch.
Open existing project from disk or version control.

Then click on "Kotlin" in the left side menu, and enter a name for your project:



Name: test

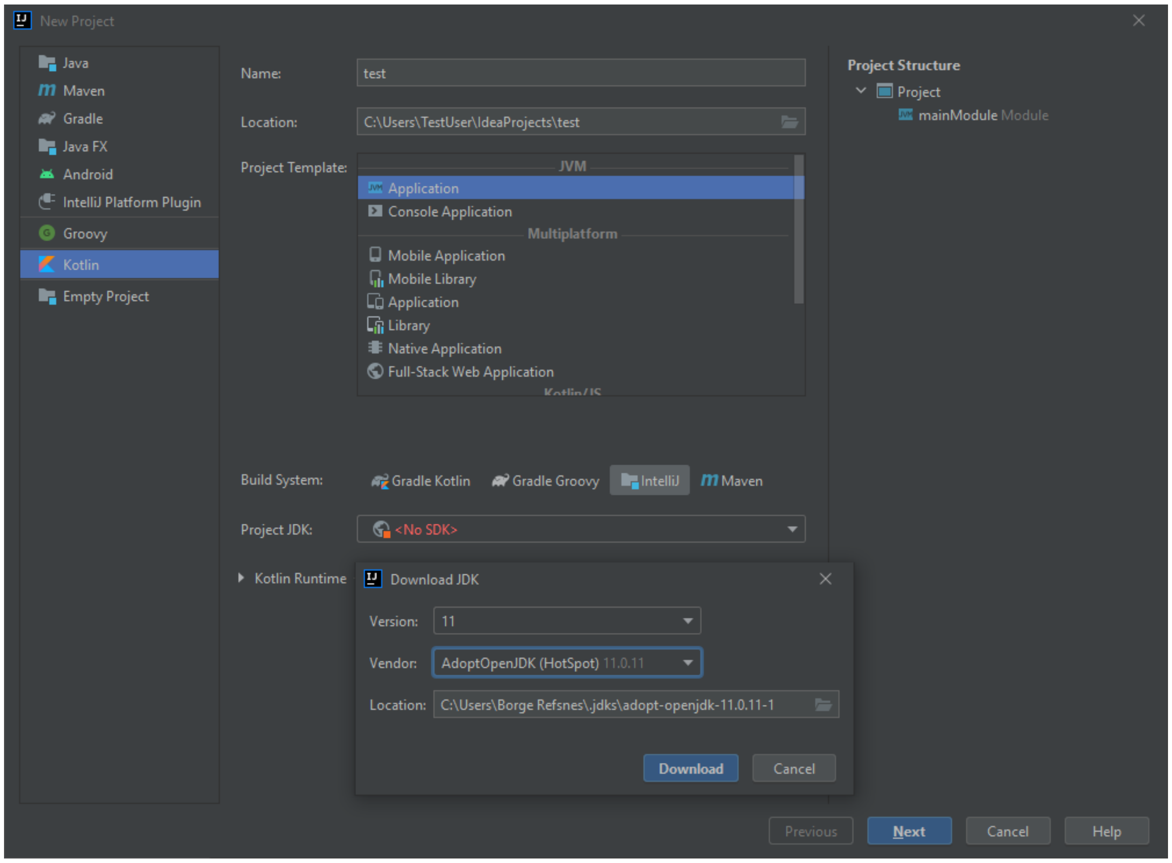
Location: C:\Users\TestUser\IdeaProjects\test

Project Template: Application

Build System: IntelliJ

Project JDK: <No SDK>

Next, we need to install something called JDK (Java Development Kit) to get our Kotlin project up and going. Click on the "Project JDK" menu, select "Download JDK" and select a version and vendor (e.g. AdoptOpenJDK 11) and click on the "Download" button:



Name: test

Location: C:\Users\TestUser\IdeaProjects\test

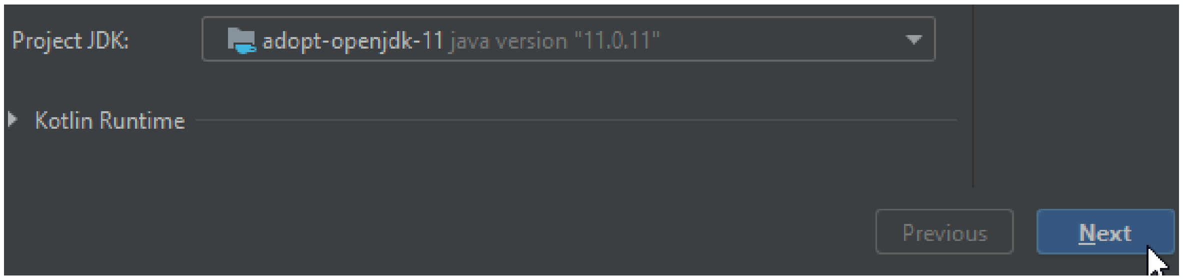
Project JDK: <No SDK>

Version: 11

Vendor: AdoptOpenJDK (HotSpot) 11.0.11

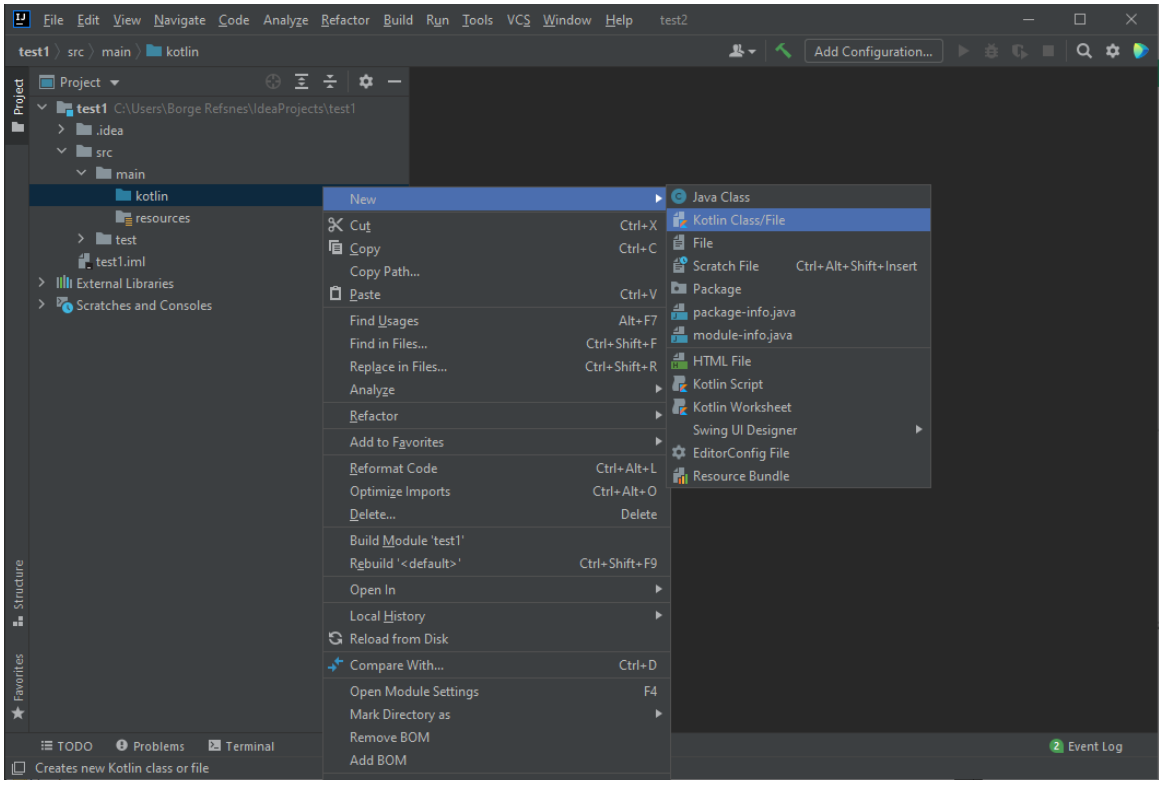
Location: C:\Users\Borge Refsnes\.jdks\adopt-openjdk-11.0.11-1

When the JDK is downloaded and installed, choose it from the select menu and then click on the "Next" button and at last "Finish":

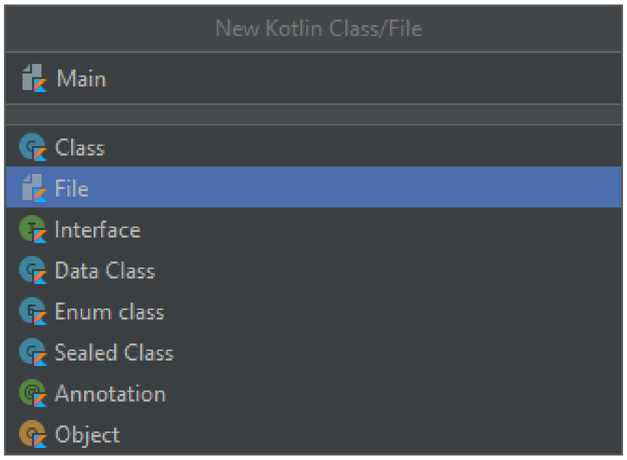


Project JDK: adopt-openjdk-11 java version "11.0.11"

Now we can start working with our Kotlin project. Do not worry about all of the different buttons and functions in IntelliJ. For now, just open the src (source) folder, and follow the same steps as in the image below, to create a kotlin file:



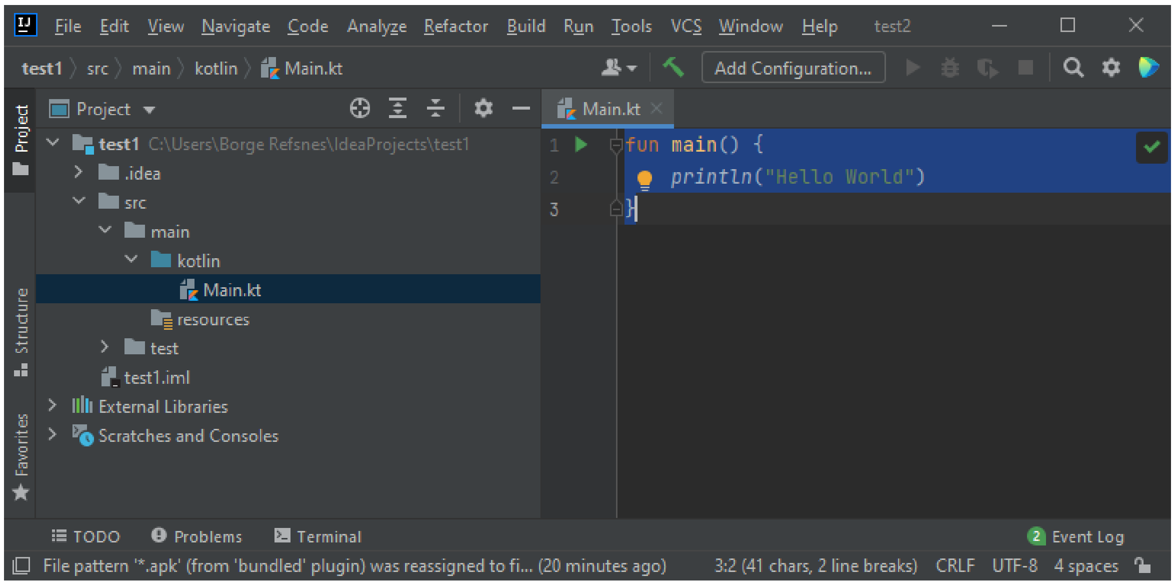
Select the "File" option and add a name to your Kotlin file, for example "Main":



You have now created your first Kotlin file (Main.kt). Let's add some Kotlin code to it, and run the program to see how it works. Inside the Main.kt file, add the following code:

Main.kt

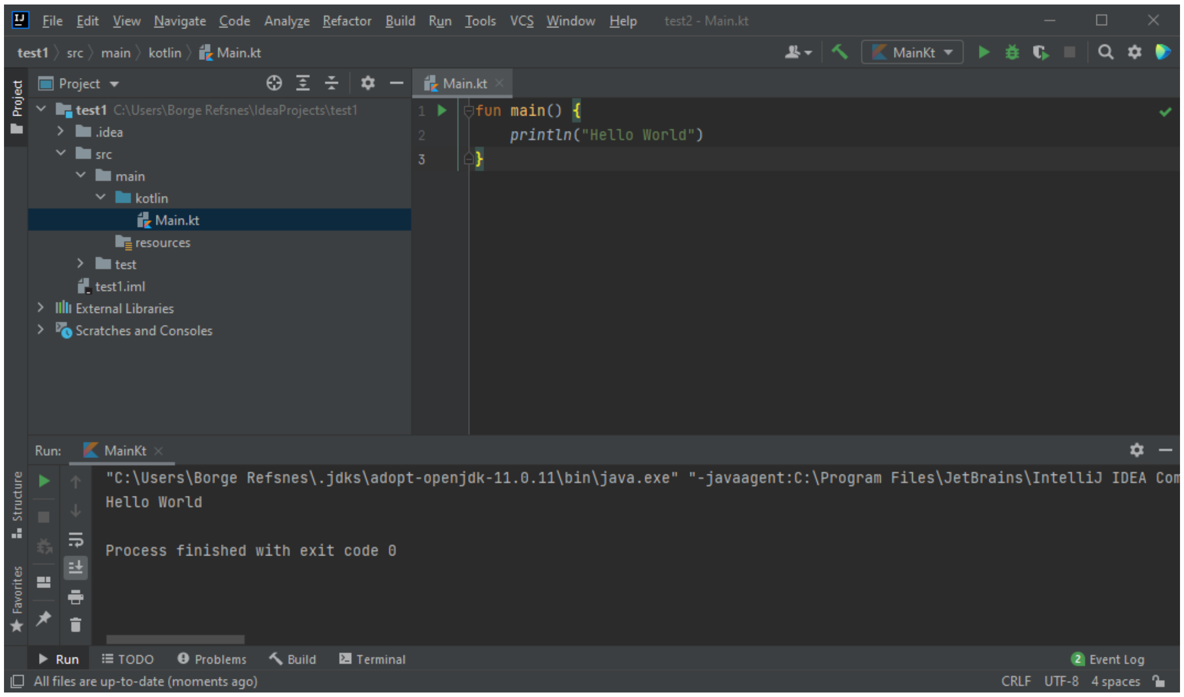
```
fun main() {
  println("Hello World")
}
```


File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help test2
test1 src main kotlin Main.kt
Add Configuration...
Project
test1 C:\Users\Borge Refsnes\IdeaProjects\test1
.idea
src
main
kotlin
Main.kt
resources
test
test1.iml
External Libraries
Scratches and Consoles
TODO Problems Terminal Event Log
File pattern '*.apk' (from 'bundled' plugin) was reassigned to fi... (20 minutes ago) 3:2 (41 chars, 2 line breaks) CRLF UTF-8 4 spaces

```
fun main() {
    println("Hello World")
}
```

Don't worry if you don't understand the code above - we will discuss it in detail in later chapters. For now, lets focus on how to run the code. Click on the Run button at the top navigation bar, then click "Run", and select "Mainkt".

Next, IntelliJ will build your project, and run the Kotlin file. The output will look something like this:


File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help test2 - Main.kt
test1 src main kotlin Main.kt
MainKt
Project
test1 C:\Users\Borge Refsnes\IdeaProjects\test1
.idea
src
main
kotlin
Main.kt
resources
test
test1.iml
External Libraries
Scratches and Consoles
Run: MainKt
Run TODO Problems Build Terminal Event Log
All files are up-to-date (moments ago) CRLF UTF-8 4 spaces

```
fun main() {
    println("Hello World")
}
```

```
"C:\Users\Borge Refsnes\.jdks\adopt-openjdk-11.0.11\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Com
Hello World

Process finished with exit code 0
```

As you can see, the output of the code was "Hello World", meaning that you have now written and executed your first Kotlin program!

Kotlin Syntax

In the previous chapter, we created a Kotlin file called `Main.kt`, and we used the following code to print "Hello World" to the screen:

Example

```
fun main() {
  println("Hello World")
}
```

Example explained

The `fun` keyword is used to declare a function. A function is a block of code designed to perform a particular task. In the example above, it declares the `main()` function.

The `main()` function is something you will see in every Kotlin program. This function is used to **execute** code. Any code inside the `main()` function's curly brackets `{}` will be **executed**.

For example, the `println()` function is inside the `main()` function, meaning that this will be executed. The `println()` function is used to output/print text, and in our example it will output "Hello World".

Good To Know: In Kotlin, code statements do not have to end with a semicolon (`;`) (which is often required for other programming languages, such as Java, C++, C#, etc.).

Main Parameters

Before Kotlin version 1.3, it was required to use the `main()` function with parameters, like: `fun main(args : Array<String>)`. The example above had to be written like this to work:

Example

```
fun main(args : Array<String>) {
  println("Hello World")
}
```

Note: This is no longer required, and the program will run fine without it. However, it will not do any harm if you have been using it in the past, and will continue to use it.

Kotlin Output (Print)

The `println()` function is used to output values/print text:

Example

```
fun main() {
  println("Hello World")
}
```

You can add as many `println()` functions as you want. Note that it will add a new line for each function:

Example

```
fun main() {
  println("Hello World!")
  println("I am learning Kotlin.")
  println("It is awesome!")
}
```

You can also print numbers, and perform mathematical calculations:

Example
```
fun main() {
  println(3 + 3)
}
```

The print() function

There is also a `print()` function, which is similar to `println()`. The only difference is that it does not insert a new line at the end of the output:

Example
```
fun main() {
  print("Hello World! ")
  print("I am learning Kotlin. ")
  print("It is awesome!")
}
```

Note that we have added a space character to create a space between the sentences.

Kotlin Comments

Comments can be used to explain Kotlin code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

Single-line Comments

Single-line comments starts with two forward slashes (`//`).

Any text between `//` and the end of the line is ignored by Kotlin (will not be executed).

This example uses a single-line comment before a line of code:

Example
```
// This is a comment
println("Hello World")
```

This example uses a single-line comment at the end of a line of code:

Example
```
println("Hello World")  // This is a comment
```

Multi-line Comments

Multi-line comments start with `/*` and ends with `*/`.

Any text between `/*` and `*/` will be ignored by Kotlin.

This example uses a multi-line comment (a comment block) to explain the code:

Example
```
/* The code below will print the words Hello World
to the screen, and it is amazing */
println("Hello World")
```

Kotlin Variables

Variables are containers for storing data values.

To create a variable, use `var` or `val`, and assign a value to it with the equal sign (`=`):

Syntax
```
var variableName = value
val variableName = value
```
Example
```
var name = "John"
val birthyear = 1975

println(name)          // Print the value of name
println(birthyear)     // Print the value of birthyear
```

The difference between `var` and `val` is that variables declared with the `var` keyword **can be changed/modified**, while `val` variables **cannot**.

Variable Type

Unlike many other programming languages, variables in Kotlin do not need to be declared with a specified *type* (like "String" for text or "Int" for numbers, if you are familiar with those).

To create a variable in Kotlin that should store text and another that should store a number, look at the following example:

Example
```
var name = "John"      // String (text)
val birthyear = 1975   // Int (number)

println(name)          // Print the value of name
println(birthyear)     // Print the value of birthyear
```

Kotlin is smart enough to understand that **"John"** is a `String` (text), and that **1975** is an `Int` (number) variable.

However, it is possible to specify the type if you insist:

Example
```
var name: String = "John" // String
val birthyear: Int = 1975 // Int

println(name)
println(birthyear)
```

You can also declare a variable without assigning the value, and assign the value later. **However**, this is only possible when you specify the type:

Example

This works fine:

```
var name: String
name = "John"
println(name)
```

Example

This will generate an error:

```
var name
name = "John"
println(name)
```

Notes on `val`

When you create a variable with the `val` keyword, the value **cannot** be changed/reassigned.

The following example will generate an error:

Example

```
val name = "John"
name = "Robert"  // Error (Val cannot be reassigned)
println(name)
```

When using `var`, you can change the value whenever you want:

Example

```
var name = "John"
name = "Robert"
println(name)
```

So When To Use `val`?

The `val` keyword is useful when you want a variable to always store the same value, like PI (3.14159...):

Example

```
val pi = 3.14159265359
println(pi)
```

Display Variables

Like you have seen with the examples above, the `println()` method is often used to display variables.

To combine both text and a variable, use the `+` character:

Example

```
val name = "John"
println("Hello " + name)
```

You can also use the `+` character to add a variable to another variable:

Example

```
val firstName = "John "
val lastName = "Doe"
val fullName = firstName + lastName
println(fullName)
```

For numeric values, the `+` character works as a mathematical operator:

Example

```
val x = 5
val y = 6
println(x + y) // Print the value of x + y
```

From the example above, you can expect:

- x stores the value 5
- y stores the value 6
- Then we use the `println()` method to display the value of x + y, which is **11**

Variable Names

A variable can have a short name (like x and y) or more descriptive names (age, sum, totalVolume).

The general rule for Kotlin variables are:

- Names can contain letters, digits, underscores, and dollar signs
- Names should start with a letter
- Names can also begin with $ and _ (but we will not use it in this tutorial)
- Names are case sensitive ("myVar" and "myvar" are different variables)
- Names should start with a lowercase letter and it cannot contain whitespace
- Reserved words (like Kotlin keywords, such as `var` or `String`) cannot be used as names

camelCase variables

You might notice that we used **firstName** and **lastName** as variable names in the example above, instead of firstname and lastname. This is called "camelCase", and it is considered as good practice as it makes it easier to read when you have a variable name with different words in it, for example "myFavoriteFood", "rateActionMovies" etc.

Kotlin Data Types

In Kotlin, the *type* of a variable is decided by its value:

Example

```
val myNum = 5             // Int
val myDoubleNum = 5.99    // Double
val myLetter = 'D'        // Char
val myBoolean = true      // Boolean
val myText = "Hello"      // String
```

However, you learned from the previous chapter that it is possible to specify the type if you want:

Example

```
val myNum: Int = 5                // Int
val myDoubleNum: Double = 5.99    // Double
val myLetter: Char = 'D'          // Char
val myBoolean: Boolean = true     // Boolean
val myText: String = "Hello"      // String
```

Sometimes you have to specify the type, and often you don't. Anyhow, it is good to know what the different types represent.

You will learn more about **when you need** to specify the type later.

Data types are divided into different groups:

- Numbers
- Characters
- Booleans
- Strings
- Arrays

Numbers

Number types are divided into two groups:

Integer types store whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are `Byte`, `Short`, `Int` and `Long`.

Floating point types represent numbers with a fractional part, containing one or more decimals. There are two types: `Float` and `Double`.

If you don't specify the type for a numeric variable, it is most often returned as `Int` for whole numbers and `Double` for floating point numbers.

Integer Types

Byte

The `Byte` data type can store whole numbers from -128 to 127. This can be used instead of `Int` or other integer types to save memory when you are certain that the value will be within -128 and 127:

Example

```
val myNum: Byte = 100
println(myNum)
```

Short

The `Short` data type can store whole numbers from -32768 to 32767:

Example

```
val myNum: Short = 5000
println(myNum)
```

Int

The `Int` data type can store whole numbers from -2147483648 to 2147483647:

Example

```
val myNum: Int = 100000
println(myNum)
```

Long

The `Long` data type can store whole numbers from -9223372036854775807 to 9223372036854775807. This is used when `Int` is not large enough to store the value. Optionally, you can end the value with an "L":

Example

```
val myNum: Long = 15000000000L
println(myNum)
```

Difference Between Int and Long

A whole number is an `Int` as long as it is up to 2147483647. If it goes beyond that, it is defined as `Long`:

Example

```
val myNum1 = 2147483647  // Int
val myNum2 = 2147483648  // Long
```

Floating Point Types

Floating point types represent numbers with a decimal, such as 9.99 or 3.14515.

The `Float` and `Double` data types can store fractional numbers:

Float Example

```
val myNum: Float = 5.75F
println(myNum)
```

Double Example

```
val myNum: Double = 19.99
println(myNum)
```

Use `Float` or `Double`?

The **precision** of a floating point value indicates how many digits the value can have after the decimal point. The precision of `Float` is only six or seven decimal digits, while `Double` variables have a precision of about 15 digits. Therefore it is safer to use `Double` for most calculations.

Also note that you should end the value of a `Float` type with an "F".

Scientific Numbers

A floating point number can also be a scientific number with an "e" or "E" to indicate the power of 10:

Example
```
val myNum1: Float = 35E3F
val myNum2: Double = 12E4
println(myNum1)
println(myNum2)
```

Booleans

The `Boolean` data type and can only take the values `true` or `false`:

Example
```
val isKotlinFun: Boolean = true
val isFishTasty: Boolean = false
println(isKotlinFun)   // Outputs true
println(isFishTasty)   // Outputs false
```

Boolean values are mostly used for conditional testing, which you will learn more about in a later chapter.

Characters

The `Char` data type is used to store a **single** character. A char value must be surrounded by **single** quotes, like 'A' or 'c':

Example
```
val myGrade: Char = 'B'
println(myGrade)
```

Unlike Java, you cannot use ASCII values to display certain characters. The value 66 would output a "B" in Java, but will generate an error in Kotlin:

Example
```
val myLetter: Char = 66
println(myLetter) // Error
```

Strings

The `String` data type is used to store a sequence of characters (text). String values must be surrounded by **double** quotes:

Example
```
val myText: String = "Hello World"
println(myText)
```

Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

Type Conversion

Type conversion is when you convert the value of one data type to another type.

In Kotlin, numeric type conversion is different from Java. For example, it is not possible to convert an `Int` type to a `Long` type with the following code:

Example
```
val x: Int = 5
val y: Long = x
println(y) // Error: Type mismatch
```

To convert a numeric data type to another type, you must use one of the following functions: `toByte()`, `toShort()`, `toInt()`, `toLong()`, `toFloat()`, `toDouble()` or `toChar()`:

Example
```
val x: Int = 5
val y: Long = x.toLong()
println(y)
```

Kotlin Operators

Operators are used to perform operations on variables and values.

The value is called an operand, while the operation (to be performed between the two operands) is defined by an **operator**:

Operand	Operator	Operand
100	+	50

In the example below, the numbers 100 and 50 are **operands**, and the + sign is an **operator**:

Example
```
var x = 100 + 50
```

Although the + operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and a variable:

Example
```
var sum1 = 100 + 50       // 150 (100 + 50)
var sum2 = sum1 + 250     // 400 (150 + 250)
var sum3 = sum2 + sum2    // 800 (400 + 400)
```

Kotlin divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value from another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value by 1	++x
--	Decrement	Decreases the value by 1	--x

Kotlin Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

Example
```
var x = 10
```

The **addition assignment** operator (+=) adds a value to a variable:

Example
```
var x = 10
x += 5
```

A list of all assignment operators:

Operator	Example	Same As
=	x = 5	x = 5
+=	*x += 3*	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3

Kotlin Comparison Operators

Comparison operators are used to compare two values, and returns a `Boolean` value: either `true` or `false`.

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Kotlin Logical Operators

Logical operators are used to determine the logic between variables or values:

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	

Kotlin Strings

Strings are used for storing text.

A string contains a collection of characters surrounded by double quotes:

Example
```
var greeting = "Hello"
```

Unlike Java, you do not have to specify that the variable should be a `String`. Kotlin is smart enough to understand that the greeting variable in the example above is a `String` because of the double quotes.

However, just like with other data types, you can specify the type if you insist:

Example

```
var greeting: String = "Hello"
```

Note: If you want to create a `String` without assigning the value (and assign the value later), you must specify the type while declaring the variable:

Example

This works fine:

```
var name: String
name = "John"
println(name)
```

Example

This will generate an error:

```
var name
name = "John"
println(name)
```

Access a String

To access the characters (elements) of a string, you must refer to the **index number** inside **square brackets.**

String indexes start with 0. In the example below, we access the first and third element in `txt`:

Example

```
var txt = "Hello World"
println(txt[0]) // first element (H)
println(txt[2]) // third element (l)
```

[0] is the first element. [1] is the second element, [2] is the third element, etc.

String Length

A String in Kotlin is an object, which contain properties and functions that can perform certain operations on strings, by writing a dot character (`.`) after the specific string variable. For example, the length of a string can be found with the `length` property:

Example

```
var txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
println("The length of the txt string is: " + txt.length)
```

String Functions

There are many string functions available, for example `toUpperCase()` and `toLowerCase()`:

Example

```
var txt = "Hello World"
println(txt.toUpperCase())   // Outputs "HELLO WORLD"
println(txt.toLowerCase())   // Outputs "hello world"
```

Comparing Strings

The `compareTo(string)` function compares two strings and returns 0 if both are equal:

Example

```
var txt1 = "Hello World"
var txt2 = "Hello World"
println(txt1.compareTo(txt2))  // Outputs 0 (they are equal)
```

Finding a String in a String

The `indexOf()` function returns the **index** (the position) of the first occurrence of a specified text in a string (including whitespace):

Example

```
var txt = "Please locate where 'locate' occurs!"
println(txt.indexOf("locate"))  // Outputs 7
```

Remember that Kotlin counts positions from zero.
0 is the first position in a string, 1 is the second, 2 is the third ...

Quotes Inside a String

To use quotes inside a string, use single quotes (`'`):

Example

```
var txt1 = "It's alright"
var txt2 = "That's great"
```

String Concatenation

The `+` operator can be used between strings to add them together to make a new string. This is called **concatenation**:

Example

```
var firstName = "John"
var lastName = "Doe"
println(firstName + " " + lastName)
```

Note that we have added an empty text (" ") to create a space between firstName and lastName on print.

You can also use the `plus()` function to concatenate two strings:

Example

```
var firstName = "John "
var lastName = "Doe"
println(firstName.plus(lastName))
```

String Templates/Interpolation

Instead of concatenation, you can also use "string templates", which is an easy way to add variables and expressions inside a string.

Just refer to the variable with the `$` symbol:

Example
```
var firstName = "John"
var lastName = "Doe"
println("My name is $firstName $lastName")
```

"String Templates" is a popular feature of Kotlin, as it reduces the amount of code. For example, you do not have to specify a whitespace between firstName and lastName, like we did in the concatenation example.

Kotlin Booleans

Very often, in programming, you will need a data type that can only have one of two values, like:

- YES / NO
- ON / OFF
- TRUE / FALSE

For this, Kotlin has a `Boolean` data type, which can take the values `true` or `false`.

Boolean Values

A boolean type can be declared with the `Boolean` keyword and can only take the values `true` or `false`:

Example
```
val isKotlinFun: Boolean = true
val isFishTasty: Boolean = false
println(isKotlinFun)   // Outputs true
println(isFishTasty)   // Outputs false
```

Just like you have learned with other data types in the previous chapters, the example above can also be written without specifying the type, as Kotlin is smart enough to understand that the variables are Booleans:

Example
```
val isKotlinFun = true
val isFishTasty = false
println(isKotlinFun)   // Outputs true
println(isFishTasty)   // Outputs false
```

Boolean Expression

A Boolean expression **returns** a Boolean value: `true` or `false`.

You can use a comparison operator, such as the **greater than** (`>`) operator to find out if an expression (or a variable) is true:

Example
```
val x = 10
val y = 9
println(x > y) // Returns true, because 10 is greater than 9
```

Or even easier:

Example
```
println(10 > 9) // Returns true, because 10 is greater than 9
```

In the examples below, we use the **equal to** (==) operator to evaluate an expression:

Example
```
val x = 10;
println(x == 10); // Returns true, because the value of x is equal to 10
```
Example
```
println(10 == 15); // Returns false, because 10 is not equal to 15
```

The Boolean value of an expression is the basis for all Kotlin comparisons and conditions.

Kotlin If ... Else

Kotlin Conditions and If..Else

Kotlin supports the usual logical conditions from mathematics:

- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b
- Equal to a == b
- Not Equal to: a != b

You can use these conditions to perform different actions for different decisions.

Kotlin has the following conditionals:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `when` to specify many alternative blocks of code to be executed

Note: Unlike Java, `if..else` can be used as a **statement** or as an **expression** (to assign a value to a variable) in Kotlin. See an example at the bottom of the page to better understand it.

Kotlin if

Use `if` to specify a block of code to be executed if a condition is `true`.

Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

Note that `if` is in lowercase letters. Uppercase letters (If or IF) will generate an error.

In the example below, we test two values to find out if 20 is greater than 18. If the condition is `true`, print some text:

Example

```
if (20 > 18) {
  println("20 is greater than 18")
}
```

We can also test variables:

Example

```
val x = 20
val y = 18
if (x > y) {
  println("x is greater than y")
}
```

Example explained

In the example above we use two variables, **x** and **y**, to test whether x is greater than y (using the > operator). As x is 20, and y is 18, and we know that 20 is greater than 18, we print to the screen that "x is greater than y".

Kotlin else

Use `else` to specify a block of code to be executed if the condition is `false`.

Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
} else {
  // block of code to be executed if the condition is false
}
```

Example

```
val time = 20
if (time < 18) {
  println("Good day.")
} else {
  println("Good evening.")
}
// Outputs "Good evening."
```

Example explained

In the example above, time (20) is greater than 18, so the condition is `false`, so we move on to the `else` condition and print to the screen "Good evening". If the time was less than 18, the program would print "Good day".

Kotlin else if

Use `else if` to specify a new condition if the first condition is `false`.

Syntax

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and condition2 is true
} else {
  // block of code to be executed if the condition1 is false and condition2 is false
}
```

Example

```
val time = 22
if (time < 10) {
  println("Good morning.")
} else if (time < 20) {
  println("Good day.")
} else {
  println("Good evening.")
}
// Outputs "Good evening."
```

Example explained

In the example above, time (22) is greater than 10, so the **first condition** is `false`. The next condition, in the `else if` statement, is also `false`, so we move on to the `else` condition since **condition1** and **condition2** is both `false` - and print to the screen "Good evening".

However, if the time was 14, our program would print "Good day."

Kotlin If..Else Expressions

In Kotlin, you can also use `if..else` statements as expressions (assign a value to a variable and return it):

Example

```
val time = 20
val greeting = if (time < 18) {
  "Good day."
} else {
  "Good evening."
}
println(greeting)
```

When using `if` as an expression, you must also include `else` (required).

Note: You can ommit the curly braces `{}` when `if` has only one statement:

Example

```
fun main() {
  val time = 20
  val greeting = if (time < 18) "Good day." else "Good evening."
  println(greeting)
}
```

Tip: This example is similar to the "ternary operator" (short hand if...else) in Java.

Kotlin when

Instead of writing many `if..else` expressions, you can use the `when` expression, which is much easier to read.

It is used to select one of many code blocks to be executed:

Example

Use the weekday number to calculate the weekday name:

```
val day = 4

val result = when (day) {
  1 -> "Monday"
  2 -> "Tuesday"
  3 -> "Wednesday"
  4 -> "Thursday"
  5 -> "Friday"
  6 -> "Saturday"
  7 -> "Sunday"
  else -> "Invalid day."
}
println(result)

// Outputs "Thursday" (day 4)
```

The `when` expression is similar to the `switch` statement in Java.

This is how it works:

- The `when` variable (**day**) is evaluated once
- The value of the **day** variable is compared with the values of each "branch"
- Each branch starts with a value, followed by an arrow (->) and a result
- If there is a match, the associated block of code is executed
- `else` is used to specify some code to run if there is no match
- In the example above, the value of `day` is `4`, meaning "Thursday" will be printed

Kotlin While Loop

Loops

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

Kotlin While Loop

The `while` loop loops through a block of code as long as a specified condition is `true`:

Syntax

```
while (condition) {
  // code block to be executed
}
```

In the example below, the code in the loop will run, over and over again, as long as the counter variable (i) is less than 5:

Example
```
var i = 0
while (i < 5) {
  println(i)
  i++
}
```

Note: Do not forget to increase the variable used in the condition, otherwise the loop will never end.

The Do..While Loop

The `do..while` loop is a variant of the `while` loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax
```
do {
  // code block to be executed
}
while (condition);
```

The example below uses a `do/while` loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

Example
```
var i = 0
do {
  println(i)
  i++
  }
while (i < 5)
```

Do not forget to increase the variable used in the condition, otherwise the loop will never end!

Kotlin Break and Continue

Kotlin Break

The `break` statement is used to jump out of a **loop**.

This example jumps out of the loop when i is equal to 4:

Example
```
var i = 0
while (i < 10) {
  println(i)
  i++
  if (i == 4) {
    break
  }
}
```

Kotlin Continue

The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 4:

Example
```
var i = 0
while (i < 10) {
  if (i == 4) {
    i++
    continue
  }
  println(i)
  i++
}
```

Kotlin Arrays

Arrays are used to store multiple values in a single variable, instead of creating separate variables for each value.

To create an array, use the `arrayOf()` function, and place the values in a comma-separated list inside it:

```
val cars = arrayOf("Volvo", "BMW", "Ford", "Mazda")
```

Access the Elements of an Array

You can access an array element by referring to the **index number**, inside **square brackets**.

In this example, we access the value of the first element in cars:

Example
```
val cars = arrayOf("Volvo", "BMW", "Ford", "Mazda")
println(cars[0])
// Outputs Volvo
```

Note: Just like with Strings, Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Array Element

To change the value of a specific element, refer to the index number:

Example
```
cars[0] = "Opel"
```

Example
```
val cars = arrayOf("Volvo", "BMW", "Ford", "Mazda")
cars[0] = "Opel"
```

```
println(cars[0])
// Now outputs Opel instead of Volvo
```

Array Length / Size

To find out how many elements an array have, use the `size` property:

Example

```
val cars = arrayOf("Volvo", "BMW", "Ford", "Mazda")
println(cars.size)
// Outputs 4
```

Check if an Element Exists

You can use the `in` operator to check if an element exists in an array:

Example

```
val cars = arrayOf("Volvo", "BMW", "Ford", "Mazda")
if ("Volvo" in cars) {
  println("It exists!")
} else {
  println("It does not exist.")
}
```

Loop Through an Array

Often when you work with arrays, you need to loop through all of the elements.

You can loop through the array elements with the `for` loop, which you will learn even more about in the next chapter.

The following example outputs all elements in the **cars** array:

Example

```
val cars = arrayOf("Volvo", "BMW", "Ford", "Mazda")
for (x in cars) {
  println(x)
}
```

Kotlin For Loop

Often when you work with arrays, you need to loop through all of the elements.

To loop through array elements, use the `for` loop together with the `in` operator:

Example

Output all elements in the cars array:

```
val cars = arrayOf("Volvo", "BMW", "Ford", "Mazda")
for (x in cars) {
  println(x)
}
```

You can loop through all kinds of arrays. In the example above, we used an array of strings.

In the example below, we loop through an array of integers:

Example

```
val nums = arrayOf(1, 5, 10, 15, 20)
for (x in nums) {
  println(x)
}
```

Traditional For Loop

Unlike Java and other programming languages, there is no traditional `for` loop in Kotlin.

In Kotlin, the `for` loop is used to loop through arrays, ranges, and other things that contains a countable number of values.

Kotlin Ranges

With the `for` loop, you can also create **ranges** of values with "`..`":

Example

Print the whole alphabet:

```
for (chars in 'a'..'x') {
  println(chars)
}
```

You can also create ranges of numbers:

Example

```
for (nums in 5..15) {
  println(nums)
}
```

Note: The first and last value is included in the range.

Check if a Value Exists

You can also use the `in` operator to check if a value exists in a range:

Example

```
val nums = arrayOf(2, 4, 6, 8)
if (2 in nums) {
  println("It exists!")
} else {
  println("It does not exist.")
}
```

Example

```
val cars = arrayOf("Volvo", "BMW", "Ford", "Mazda")
if ("Volvo" in cars) {
  println("It exists!")
} else {
  println("It does not exist.")
}
```

Break or Continue a Range

You can also use the `break` and `continue` keywords in a range/`for` loop:

Example

Stop the loop when `nums` is equal to `10`:

```
for (nums in 5..15) {
  if (nums == 10) {
    break
  }
  println(nums)
}
```

Example

Skip the value of 10 in the loop, and continue with the next iteration:

```
for (nums in 5..15) {
  if (nums == 10) {
    continue
  }
  println(nums)
}
```

Kotlin Functions

A **function** is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

Functions are used to perform certain actions, and they are also known as **methods**.

Predefined Functions

So it turns out you already know what a function is. You have been using it the whole time through this tutorial!

For example, `println()` is a function. It is used to output/print text to the screen:

Example

```
fun main() {
  println("Hello World")
}
```

Create Your Own Functions

To create your own function, use the `fun` keyword, and write the name of the function, followed by parantheses **()**:

Example

Create a function named "myFunction" that should output some text:

```
fun myFunction() {
  println("I just got executed!")
}
```

Call a Function

Now that you have created a function, you can execute it by **calling** it.

To call a function in Kotlin, write the name of the function followed by two parantheses **()**.

In the following example, `myFunction()` will print some text (the action), when it is called:

Example
```
fun main() {
  myFunction() // Call myFunction
}

// Outputs "I just got executed!"
```

A function can be called multiple times, if you want:

Example
```
fun main() {
  myFunction()
  myFunction()
  myFunction()
}

// I just got executed!
// I just got executed!
// I just got executed!
```

Function Parameters

Information can be passed to functions as parameter.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma. Just note that you must specify the type of each parameter (Int, String, etc).

The following example has a function that takes a `String` called **fname** as parameter. When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
fun myFunction(fname: String) {
  println(fname + " Doe")
}

fun main() {
  myFunction("John")
  myFunction("Jane")
  myFunction("George")
}

// John Doe
// Jane Doe
// George Doe
```

When a **parameter** is passed to the function, it is called an **argument**. So, from the example above: `fname` is a **parameter**, while `John`, `Jane` and `George` are **arguments**.

Multiple Parameters

You can have as many parameters as you like:

Example

```
fun myFunction(fname: String, age: Int) {
  println(fname + " is " + age)
}

fun main() {
  myFunction("John", 35)
  myFunction("Jane", 32)
  myFunction("George", 15)
}

// John is 35
// Jane is 32
// George is 15
```

Note: When working with multiple parameters, the function call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

Return Values

In the examples above, we used functions to output a value. In the following example, we will use a function to **return** a value and assign it to a variable.

To return a value, use the `return` keyword, and specify the **return type** after the function's parantheses (`Int` in this example):

Example

A function with one `Int` parameter, and `Int` return type:

```
fun myFunction(x: Int): Int {
  return (x + 5)
}
```

```
fun main() {
  var result = myFunction(3)
  println(result)
}

// 8 (3 + 5)
```

Using two parameters:

Example

A function with two `Int` parameters, and `Int` return type:

```
fun myFunction(x: Int, y: Int): Int {
  return (x + y)
}

fun main() {
  var result = myFunction(3, 5)
  println(result)
}

// 8 (3 + 5)
```

Shorter Syntax for Return Values

There is also a shorter syntax for returning values. You can use the `=` operator instead of `return` without specifying the return type. Kotlin is smart enough to automatically find out what it is:

Example

```
fun myFunction(x: Int, y: Int) = x + y

fun main() {
  var result = myFunction(3, 5)
  println(result)
}

// 8 (3 + 5)
```