

Introduction to Kotlin Programming

By Coding-Bootcamps.com

Course Outline

1. [Kotlin Overview, Installation, and Setup](#)

- Why Kotlin?
- Install and Setup Kotlin
 - Installing the Standalone Compiler- Command Line
 - Setting up Kotlin in IntelliJ IDEA
 - Setting up Kotlin in Eclipse

2. [Writing your first Kotlin program](#)

3. [Kotlin Variables and Data Types](#)

- Variables
- Type inference
- Data Types
- Arrays
- Type Conversions

4. [Kotlin Operators with Examples](#)

- Operations on Numeric Types
- Bitwise Operators
- Operations on Boolean Types
- Operations on Strings

5. [Kotlin Control Flow: if and when expressions, for and while loops](#)

- If Statement
- If-Else Statement
- Using If as an Expression
- If-Else-If Chain
- When Expression
- While Loop
- do-while loop
- For Loop
- Break and Continue

6. [Nullable Types and Null Safety in Kotlin](#)

- Nullability and Nullable Types in Kotlin
- Working with Nullable Types
- Null Safety and Java Interoperability
- Nullability and Collections

7. [Kotlin Functions, Default and Named Arguments, Varargs and Function Scopes](#)

- Defining and Calling Functions
- Function Default Arguments
- Function Named Arguments
- Variable Number of Arguments (Varargs)
- Function Scope
 - 1. Top Level Functions
 - 2. Member Functions
 - 3. Local/Nested Functions

8. [Kotlin Infix Notation - Make function calls more intuitive](#)

Session 1- Kotlin Overview, Installation, and Setup

[Kotlin](#) is a programming language developed by JetBrains, the same company that has built world-class IDEs like IntelliJ IDEA, PhpStorm, PyCharm, ReSharper etc.

It runs on the Java Virtual Machine (JVM), and can also be compiled to JavaScript and Machine Code.

In this session, we will give you a brief overview of Kotlin and its features. We will also help you set up Kotlin in your system and prepare you for future sessions.

Why Kotlin?

In today's world where we have a dozen programming language for every possible task, following are few reasons to choose Kotlin as the primary language for your next exciting project -

1. Statically Typed

Kotlin is a Statically typed programming language. This means that the type of every variable and expression is known at compile time.

The advantage with static typing is that the compiler can validate the methods calls and property access on the objects at compile time itself and prevent lots of trivial bugs that would otherwise crop up at runtime.

Although Kotlin is a statically typed language, it doesn't require you to explicitly specify the type of every variable you declare. Most of the time, Kotlin can infer the type of a variable from the initializer expression or the surrounding context. This is called *Type Inference*. You'll learn more about Type inference in the Variables and Data Types session.

2. Concise

Kotlin is concise. It drastically reduces the amount of boilerplate code that you have been writing all the time in other OOP languages like Java.

It provides rich idioms for performing common tasks. For example, You can create a POJO class with getters, setters, equals(), hashCode() and toString() methods in a single line -

```
data class User(val name: String, val email: String, val country: String)
```

3. Safe

Kotlin is safe. It avoids the most dreaded and annoying NullPointerExceptions by supporting nullability as part of its type system.

It works like this - Every variable in Kotlin is non-null by default:

```
String str = "Hello, World"    // Non-null type (can't hold null value)
str = null // Compiler Error
```

To allow a variable to hold null value, you need to explicitly declare it as nullable:

```
String nullableStr? = null    // Nullable type (can be null)
```

Since Kotlin knows which variables are nullable and which are not, It can detect and disallow unsafe calls at compile time itself that would otherwise result in a NullPointerException at runtime -

```
println(nullableStr.length()) // Compiler Error
```

Kotlin doesn't allow the method call length() on the nullableStr variable because the call is not safe and may lead to NullPointerException.

However, if you add a null check then the method call is allowed -

```
if(nullableStr != null) {
    println(nullableStr.length())
}
```

```
}
```

Notice how Kotlin is enforcing developers to write safe code by distinguishing between nullable and non-null types.

4. Explicit

Kotlin is Explicit. It will do/allow things only if you tell it to do so. Explicitness is considered a good thing. Being explicit means being specific about your design choices and not hiding anything from the readers or consumers of your code.

Following are few examples of Explicitness in Kotlin -

- Kotlin doesn't allow implicit type conversions, for example, `int` to `long`, or `float` to `double`. It provides methods like `toLong()` and `toDouble()` to do so explicitly.
- All the classes in Kotlin are `final` (non-inheritable) by default. You need to explicitly mark a class as `open` to allow other classes to inherit from it. Similarly, all the properties and member functions of a class are `final` by default. You need to explicitly mark a function or property as `open` to allow child classes to override it.
- If you're overriding a parent class function or property, then you need to explicitly annotate it with the `override` modifier.

5. Easy to learn.

Kotlin has a very low learning curve. The [basic syntax](#) looks a lot like Java. If you have a little experience in Java or any other OOP language then you'll be able to pick up Kotlin in a matter of hours.

6. Functional and Object Oriented Capabilities

Kotlin has both functional and object-oriented capabilities. It has a rich set of features to support functional programming which includes functional types, lambda expressions, data classes and much more.

7. Completely interoperable with Java

Kotlin is 100% interoperable with Java. You can easily access Java code from Kotlin and vice versa. You can use Kotlin and Java in the same project without any problem. This enables easy adoption of Kotlin into your existing Java projects.



Kotlin Java Interoperability

8. Excellent Tooling

Kotlin has excellent tooling support. You can choose any Java IDE - IntelliJ IDEA, Eclipse, Android Studio. All of them support Kotlin.

Moreover, you can also download Kotlin's standalone compiler and run Kotlin code from the command line.



9. Build Applications for Server Side, Android, Browser, and Desktop

You can use Kotlin to build applications for a wide range of platforms including Server side, Android, Browser, and Desktop.

- Android has official support for Kotlin.
- On the server side, you can use Kotlin with the [Spring framework](#) which has added full support for Kotlin in Spring version 5.
- Kotlin can be compiled to [JavaScript](#) and [Machine code](#) as well.

10. Free and Open Source

Kotlin programming language, including the compiler, libraries and all the tooling is completely free and open source. It is available under Apache 2 license and the complete project is hosted on Github - <https://github.com/JetBrains/kotlin>

Setup Kotlin

You can set up and run Kotlin programs in several ways. You can either install Kotlin's compiler and run Kotlin programs from the command line or install and setup Kotlin in an IDE like IntelliJ or Eclipse -

- Install Kotlin's Standalone Compiler
- Setup Kotlin in IntelliJ IDEA
- Setup Kotlin in Eclipse

Installing the Standalone Compiler

Follow the steps below to install Kotlin's compiler -

1. Go to [Kotlin releases](#) page on Github
2. Download Kotlin's compiler in the form of a zip file from the Assets section on the Github releases page. The latest version of Kotlin compiler at the time of writing this page is 1.2.10
3. Unzip the downloaded `kotlin-compiler-x.x.x.zip` file and store the unzipped folder in a location where you have write access.
4. Add `path-to-unzipped-folder/bin` to your PATH variable.
5. Verify the installation by typing `kotlinc` in the command line -

```
$ kotlinc

Welcome to Kotlin version 1.2.10 (JRE 1.8.0_112-b16)

Type :help for help, :quit for quit

>>>
```

Run your first Kotlin program from the command line

Open your favorite editor and create a new file called `hello.kt` with the following contents -

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

Save the file and type the following commands to compile and run the program

```
$ kotlinc hello.kt
```



```
$ kotlin HelloKt  
Hello, World
```

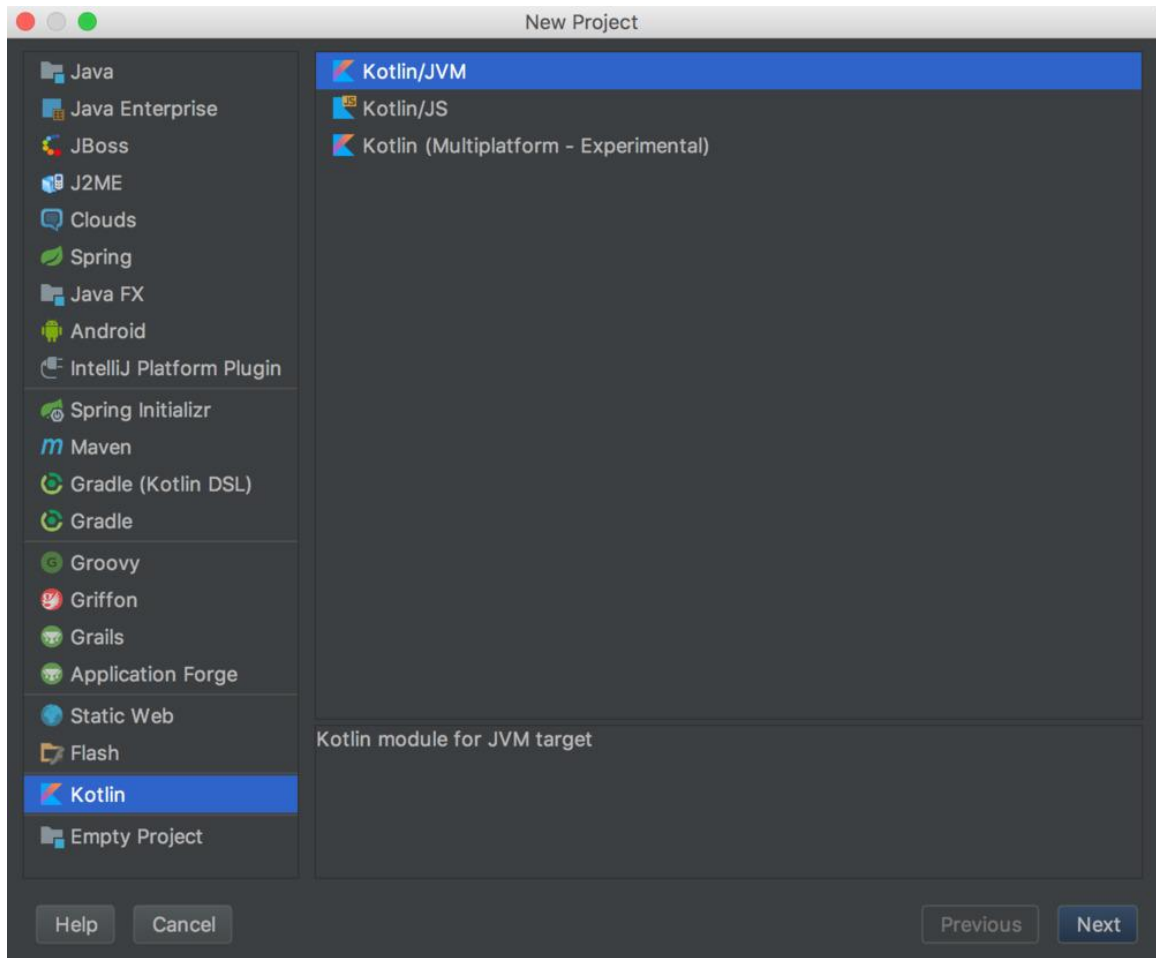
Setting up Kotlin in IntelliJ IDEA

Install the latest version of [IntelliJ IDEA](#). Kotlin comes bundled with the recent versions of IntelliJ. You won't need to install any plug-in separately to run Kotlin programs.

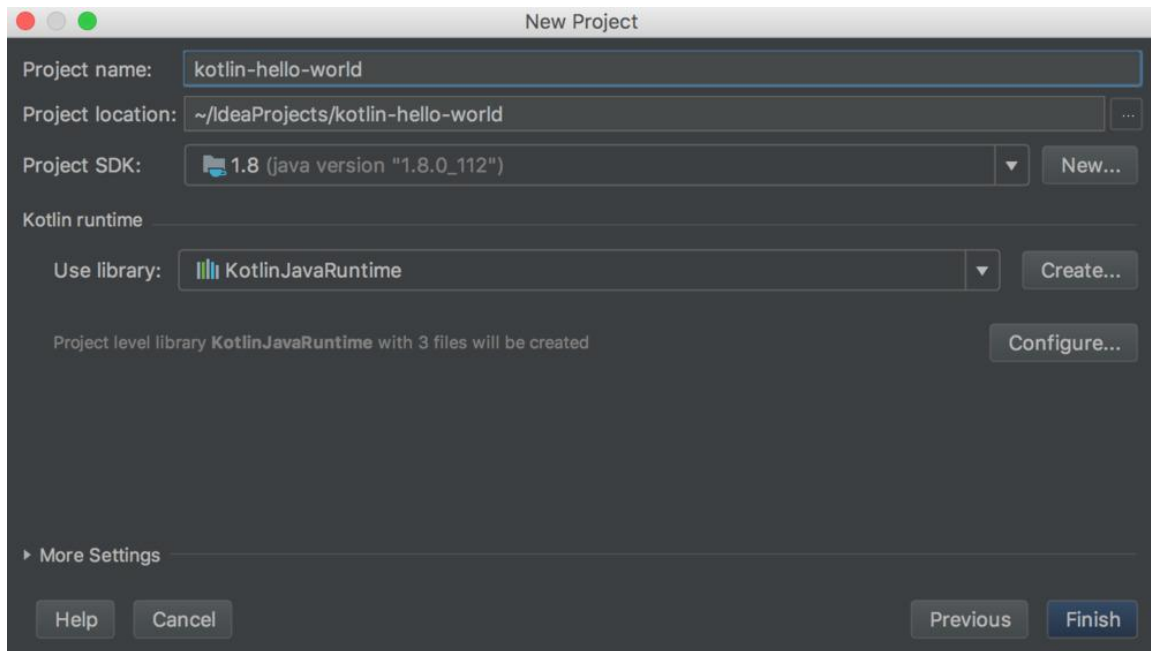
Follow these steps to create and run a new Kotlin project in IntelliJ

1. Create a new project by selecting "Create New Project" on the welcome screen or go to `File → New → Project`.

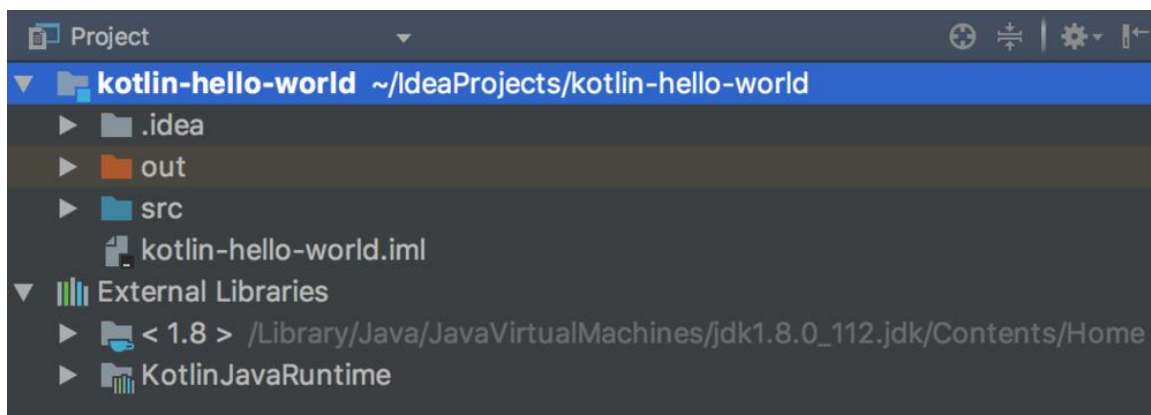
Select `Kotlin` on the left side menu and `Kotlin/JVM` from the options on the right side -



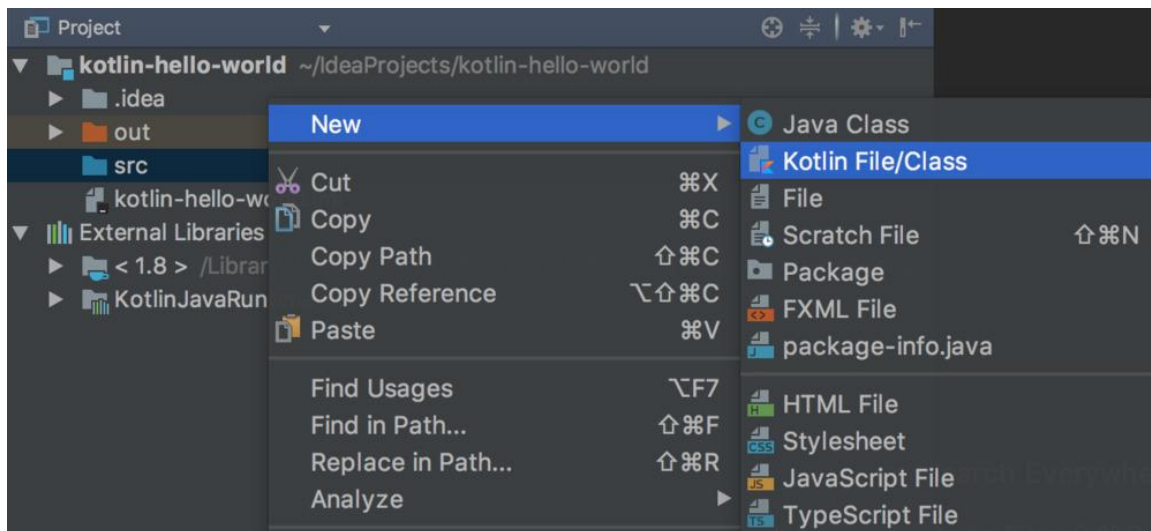
2. Specify the project's name and location, and select a Java version (1.6+) in the Project SDK. Once all the details are entered, click `Finish` to create the project -



The generated project will look like this -

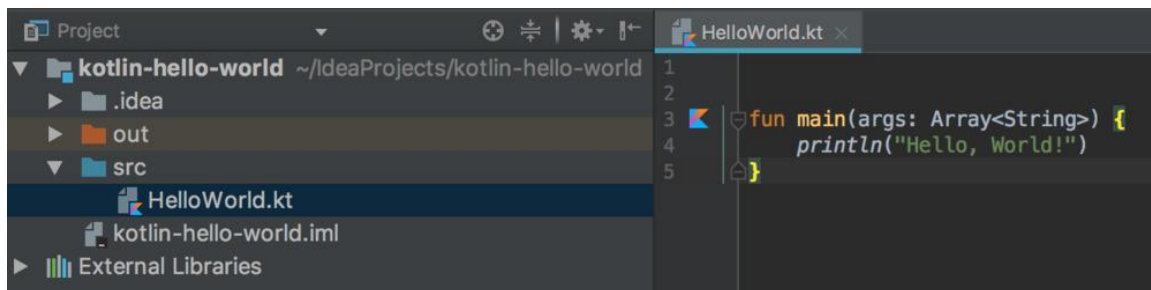


3. Let's now create a new Kotlin file. Right click on src folder → New → Kotlin File/Class.

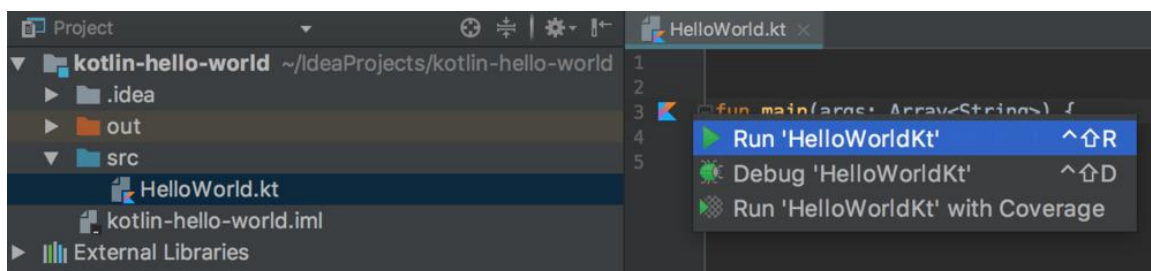


A prompt will appear where you'll need to provide a name for the file. Let's name it `HelloWorld`.

- Now let's write a simple hello world program in the new file that we have created. Add the following `main()` function to the `HelloWorld.kt` file -



- Finally, You can run the program by clicking the Kotlin icon that appears beside the `main()` method -



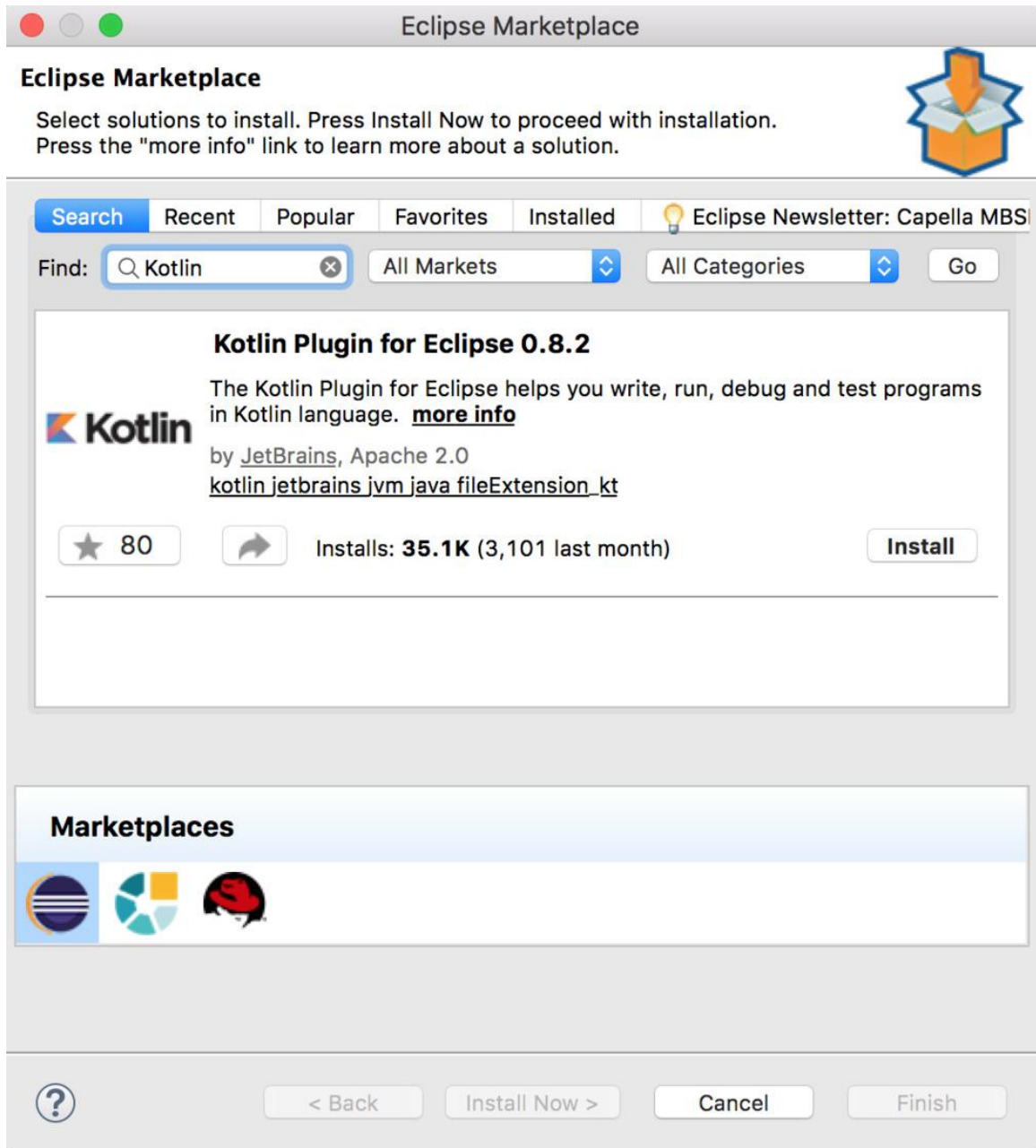
You can also run the program by Right Clicking the `HelloWorld.kt` file and selecting `Run 'HelloWorldKt'`.

Setting up Kotlin in Eclipse

I assume that you have Eclipse installed on your system. If not, download the eclipse installer from [Eclipse Downloads](#) page, and install “Eclipse IDE for Java Developers”.

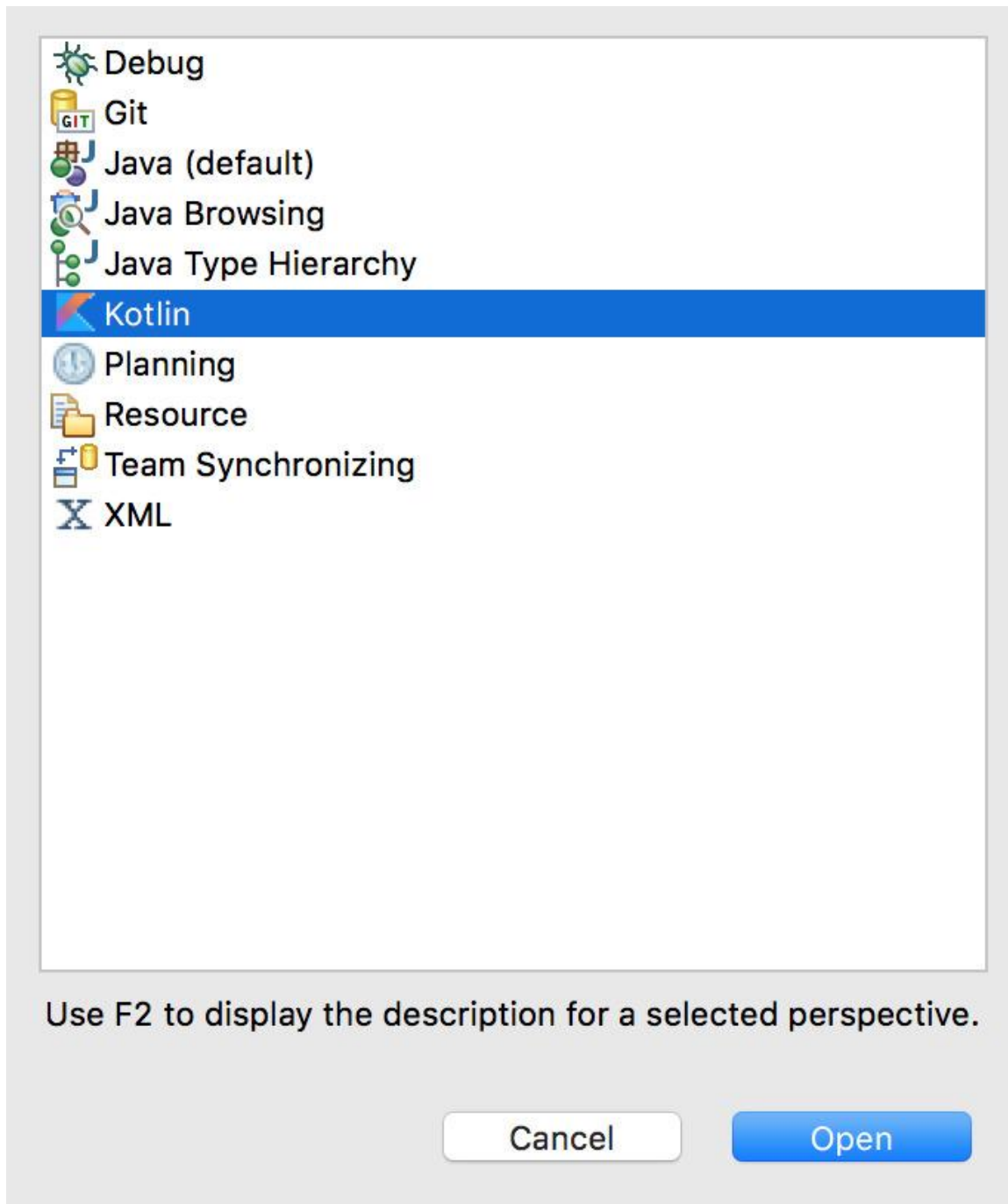
Once Eclipse is installed, follow the steps below to setup and run Kotlin in Eclipse -

1. Install Kotlin Plugin from Eclipse Marketplace: Go to `Help → Eclipse Marketplace`, and search for Kotlin.

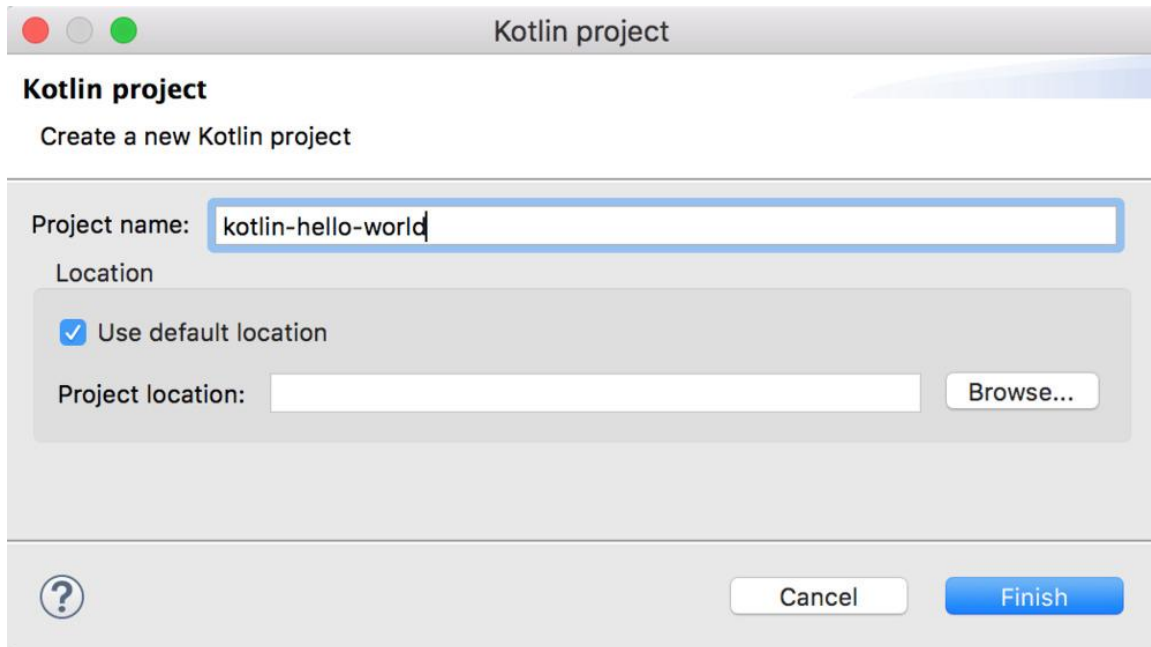


Click install to install the plugin.

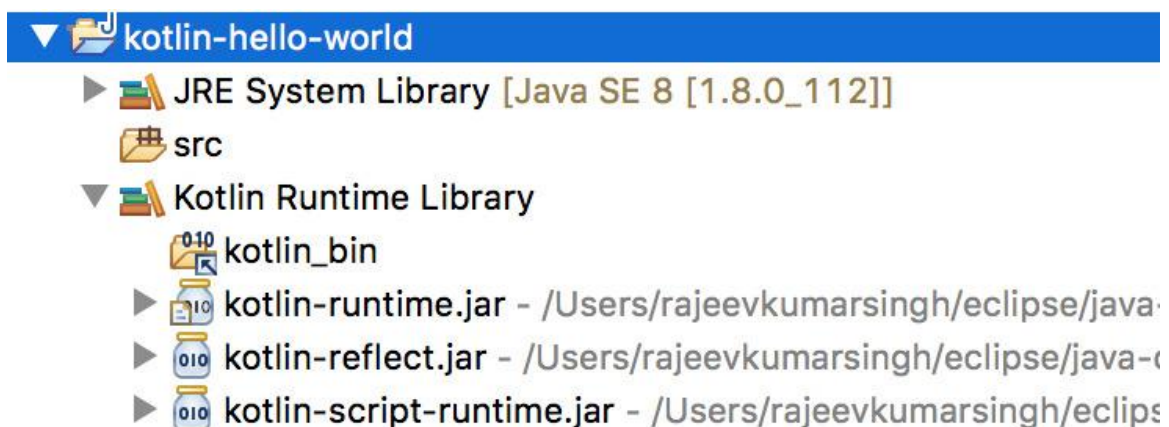
2. You will need to restart eclipse once the installation is finished.
3. Let's verify the plugin's installation switching to Kotlin perspective in eclipse. Go to `Window → Perspective → Open Perspective → Other`. A window will open which will show `Kotlin` as a new perspective. Select Kotlin and click `Open` to open Kotlin perspective -



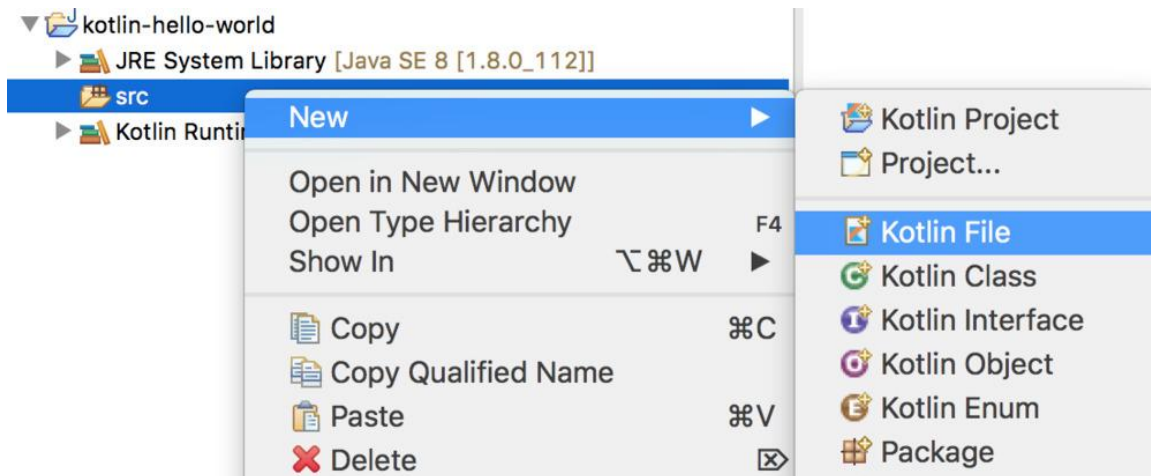
4. Let's now create a new project. Select `File → New → Kotlin Project`. Enter the project's name and click finish -



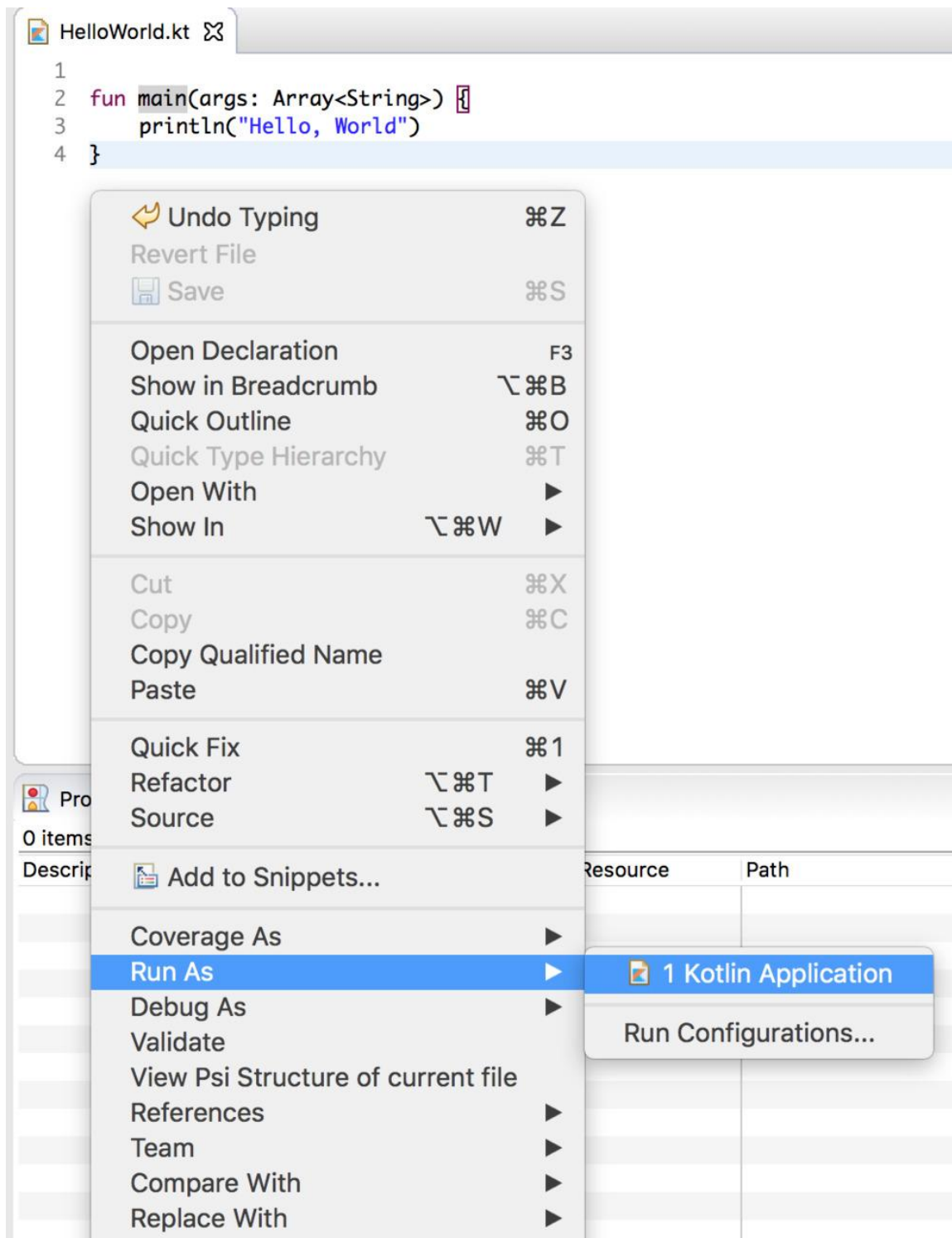
5. A new project will be created which will look like this -



6. Let's now create a new Kotlin file under the `src` folder. Right click `src` folder
→ New → Kotlin File -



7. First, add the following code in the `HelloWorld.kt` file, and then right-click anywhere on the source file and click `Run As → Kotlin Application` to run the application -



Session 2- Writing your first Kotlin program

The first program that we typically write in any programming language is the “Hello, World” program. Let’s write the “Hello, World” program in Kotlin and understand its internals.

The “Hello, World!” program in Kotlin

Open your favorite editor or an IDE and create a file named `hello.kt` with the following code -

```
// Kotlin Hello World Program
fun main(args: Array<String>) {
    println("Hello, World!")
}
```

You can compile and run the program using Kotlin’s compiler like so -

```
$ kotlinc hello.kt
$ kotlin HelloKt
Hello, World!
```

You can also run the program in an IDE like Eclipse or IntelliJ. Learn how to Setup and run Kotlin programs in your system.

Internals of the “Hello, World!” program

- **Line 1.** The first line is a comment. Comments are ignored by the compiler. They are used for your own sake (and for others who read your code).

Kotlin supports two different styles of comments similar to other languages like C, C++, and Java -

1. Single Line Comment:

```
2. // This is a Single line comment
```

3. Multi-line Comment:

```
4. /*  
5.     This is an example of a multi-line comment.  
6.     It can span over multiple lines.  
7. */
```

- **Line 2.** The second line defines a function called `main`.

```
• fun main(args: Array<String>) {  
•     // ...  
• }
```

Functions are the building blocks of a Kotlin program. All functions in Kotlin start with the keyword `fun` followed by a name of the function (`main` in this case), a list of zero or more comma-separated parameters, an optional return type, and a body.

The `main()` function takes one argument - an `Array` of strings, and returns `Unit`. The `Unit` type corresponds to `void` in Java. It is used to indicate that the function doesn't return any meaningful value.

The `Unit` type is optional. i.e. you don't need to declare it explicitly in the function declaration -

```
fun main(args: Array<String>): Unit {    // `Unit` is optional  
  
}
```

If you don't specify a return type, `Unit` is assumed.

The `main()` function is not just any ordinary function. It is the entry point of your program. It is the first thing that gets called when you run a Kotlin program.

- **Line 3.** The third line is a statement. It prints a String “Hello, World!” to standard output.

```
• println("Hello, World!")    // No Semicolons needed :)
```

Note that we can directly use `println()` to print to standard output, unlike Java where we need to use `System.out.println()`.

Kotlin provides several wrappers over standard Java library functions, `println` is one of them.

Also, Semicolons are optional in Kotlin, just like many other modern languages.

Session 3- Kotlin Variables and Data Types

In this session, you’ll learn how to declare variables in Kotlin, how Kotlin infers the type of variables, and what are the basic data types supported by Kotlin for creating variables.

You’ll also learn how to work with various data types and how to convert one type to another.

Variables

A variable refers to a memory location that stores some data. It has a name and an associated type. The type of a variable defines the range of values that the variable can hold, and the operations that can be done on those values.

You can declare a variable in Kotlin using `var` and `val` keywords.

A variable declared using `val` keyword is immutable (read-only). It cannot be reassigned after it is initialized -

```
val name = "Bill Gates"

name = "Satoshi Nakamoto"    // Error: Val cannot be reassigned
```

For defining a mutable variable, i.e. a variable whose value can be changed, use the `var` keyword -

```
var country = "USA"

country = "India"    // Works
```

Type inference

Did you notice one thing about the variable declarations in the previous section? We didn't specify the type of variables.

Although Kotlin is a statically typed language, It doesn't require you to explicitly specify the type of every variable you declare. It can infer the type of a variable from the initializer expression -

```
val greeting = "Hello, World"    // type inferred as `String`

val year = 2018                  // type inferred as `Int`
```

If you want to explicitly specify the type of a variable, you can do that like this -

```
// Explicitly defining the type of variables

val greeting: String = "Hello, World"

val year: Int = 2018
```

Note that the type declaration becomes mandatory if you're not initializing the variable at the time of declaration -

```
var language    // Error: The variable must either have a Type annotation or
be initialized
```

```
language = "French"
```

The above variable declaration fails because Kotlin has no way to infer the type of the variable without an initializer expression. In this case, you must explicitly specify the type of the variable -

```
var language: String // Works  
language = "French"
```

Data Types

Data Types are used to categorize a set of related values and define the operations that can be done on them.

Just like other languages, Kotlin has predefined types like `Int`, `Double`, `Boolean`, `Char` etc.

In Kotlin, everything (even the basic types like `Int` and `Boolean`) is an object. More specifically, everything behaves like an Object.

Kotlin may represent some of the basic types like numbers, characters and booleans as primitive values at runtime to improve performance, but for the end users, all of them are objects.

This is contrary to languages like Java that has separate primitive types like `int`, `double` etc, and their corresponding wrapper types like `Integer`, `Double` etc.

Let's now look at all the basic data types used in Kotlin one by one -

Numbers

Numeric types in Kotlin are similar to Java. They can be categorized into integer and floating point types.

Integers

- `Byte` - 8 bit
- `Short` - 16 bit
- `Int` - 32 bit
- `Long` - 64 bit

Floating Point Numbers

- `Float` - 32 bit single-precision floating point value.
- `Double` - 64 bit double-precision floating point value.

Following are few examples of numeric types -

```
// Kotlin Numeric Types Examples

val myByte: Byte = 10
val myShort: Short = 125

val myInt = 1000
val myLong = 1000L    // The suffix 'L' is used to specify a long value

val myFloat = 126.78f  // The suffix 'f' or 'F' represents a Float
val myDouble = 325.49
```

You can also use underscore in numeric values to make them more readable -

```
val hundredThousand = 100_000
val oneMillion = 1_000_000
```

You can declare hexadecimal and binary values like this -

```
val myHexa = 0x0A0F  // Hexadecimal values are prefixed with '0x' or '0X'
```



```
val myBinary = 0b1010 // Binary values are prefixed with '0b' or '0B'
```

Note that Kotlin doesn't have any representation for octal values.

Booleans

The type `Boolean` is used to represent logical values. It can have two possible values `true` and `false`.

```
val myBoolean = true
val anotherBoolean = false
```

Characters

Characters are represented using the type `Char`. Unlike Java, `Char` types cannot be treated as numbers. They are declared using single quotes like this -

```
val letterChar = 'A'
val digitChar = '9'
```

Just like other languages, special characters in Kotlin are escaped using a backslash. Some examples of escaped characters are -

`\n` (newline), `\t` (tab), `\r` (carriage return), `\b` (backspace) etc.

Strings

Strings are represented using the `String` class. They are immutable, that means you cannot modify a String by changing some of its elements.

You can declare a String like this -

```
var myStr = "Hello, Kotlin"
```

You can access the character at a particular index in a String using `str[index]`. The index starts from zero -

```
var name = "John"

var firstCharInName = name[0] // 'J'

var lastCharInName = name[name.length - 1] // 'n'
```

The `length` property is used to get the length of a String.

Escaped String and Raw String

Strings declared in double quotes can have escaped characters like `'\n'` (new line), `'\t'` (tab) etc -

```
var myEscapedString = "Hello Reader,\nWelcome to my Blog"
```

In Kotlin, you also have an option to declare raw strings. These Strings have no escaping and can span multiple lines -

```
var myMultilineRawString = """

    The Quick Brown Fox

    Jumped Over a Lazy Dog.

    """
```

Arrays

Arrays in Kotlin are represented using the `Array` class. You can create an array in Kotlin either using the library function `arrayOf()` or using the `Array()` constructor.

Creating Arrays using the `arrayOf` library function

You can pass a bunch of values to the `arrayOf` function to create an array like this -

```
var numbers = arrayOf(1, 2, 3, 4, 5)
var animals = arrayOf("Cat", "Dog", "Lion", "Tiger")
```

Note that you can also pass values of mixed types to the `arrayOf()` function, and it will still work (but don't do that) -

```
var mixedArray = arrayOf(1, true, 3, "Hello", 'A')    // Works and creates an
array of Objects
```

You can also enforce a particular type while creating the array like this -

```
var numArray = arrayOf<Int>(1, 2, 3, 4, "Hello")    // Compiler Error
```

Accessing the elements of an array by their index

You can access the element at a particular index in an array using `array[index]`. The index starts from zero -

```
val myDoubleArray = arrayOf(4.0, 6.9, 1.7, 12.3, 5.4)
val firstElement = myDoubleArray[0]
val lastElement = myDoubleArray[myDoubleArray.size - 1]
```

Every array has a `size` property that you can use to get the size of the array.

You can also modify the array element at an index like this -

```
val a = arrayOf(4, 5, 7)    // [4, 5, 7]
a[1] = 10                    // [4, 10, 7]
```

Primitive Arrays

As we learned earlier, everything in Kotlin is an object. But to improve performance it represents some of the basic types like numbers, characters and booleans as primitive types at runtime.

The `arrayOf()` function creates arrays of boxed/wrapper types. That is, `arrayOf(1, 2, 3)` corresponds to Java's `Integer[]` array.

But, Kotlin provides a way to create arrays of primitive types as well. It contains specialized classes for representing array of primitive types. Those classes are - `IntArray`, `DoubleArray`, `CharArray` etc. You can create an array of primitive types using the corresponding library functions -

`intArrayOf()`, `doubleArrayOf()`, `charArrayOf()` etc. -

```
val myCharArray = charArrayOf('K', 'O', 'T') // CharArray (corresponds to
Java 'char[]')

val myIntArray = intArrayOf(1, 3, 5, 7) // IntArray (corresponds
to Java 'int[]')
```

Creating Arrays using the `Array()` constructor

The `Array()` constructor takes two arguments -

1. the size of the array, and
2. a function that takes the array index as an argument and returns the element to be inserted at that index.

```
var mySquareArray = Array(5, {i -> i * i}) // [0, 1, 4, 9, 16]
```

The second argument to the `Array()` constructor is a lambda expression. Lambda expressions are anonymous functions that are declared and passed around as expressions. We'll learn more about lambda expressions in a future session.

The above lambda expression takes the index of an array element and returns the value that should be inserted at that index, which is the square of the index in this case.

Type Conversions

Unlike Java, Kotlin doesn't support implicit conversion from smaller types to larger types. For example, `Int` cannot be assigned to `Long` or `Double`.

```
var myInt = 100  
  
var myLong: Long = myInt // Compiler Error
```

However, Every number type contains helper functions that can be used to explicitly convert one type to another.

Following helper functions are supported for type conversion between numeric types -

- `toByte()`
- `toShort()`
- `toInt()`
- `toLong()`
- `toFloat()`
- `toDouble()`
- `toChar()`

Examples of explicit type conversions

Here is how you can convert an `Int` to `Long` -

```
val myInt = 100  
  
val myLong = myInt.toLong() // Explicitly converting 'Int' to 'Long'
```

You can also convert larger types to smaller types -

```
val doubleValue = 176.80  
  
val intValue = doubleValue.toInt() // 176
```

Every type in Kotlin, not just numeric type, supports a helper function called `toString()` to convert it to `String`.

```
val myInt = 1000  
myInt.toString() // "1000"
```

You can also convert a String to a numeric type like so -

```
val str = "1000"  
val intValue = str.toInt()
```

If the String-to-Number conversion is not possible then a `NumberFormatException` is thrown -

```
val str = "1000ABC"  
str.toInt() // Throws java.lang.NumberFormatException
```

To summarize, in this session, you learned how to declare mutable and immutable variables. How type inference works in Kotlin. What are the basic data types supported in Kotlin. How to work with data types like `Int`, `Long`, `Double`, `Char`, `Boolean`, `String` and `Array`, and how to convert one type to another.

Session 4- Kotlin Operators with Examples

In the previous session, you learned how to create variables and what are various basic data types available in Kotlin for creating variables.

In this session, you'll learn what are various operators provided by Kotlin to perform operations on basic data types.

Operations on Numeric Types

Just like other languages, Kotlin provides various operators to perform computations on numbers -

- Arithmetic operators (+, -, *, /, %)

- Comparison operators (==, !=, <, >, <=, >=)
- Assignment operators (+=, -=, *=, /=, %=)
- Increment & Decrement operators (++ , --)

Following are few examples that demonstrate the usage of above operators -

```
var a = 10
var b = 20

var c = ((a + b) * (a + b))/2    // 450

var isALessThanB = a < b    // true

a++    // a now becomes 11
b += 5    // b equals to 25 now
```

Understanding how operators work in Kotlin

Everything in Kotlin is an object, even the basic data types like `Int`, `Char`, `Double`, `Boolean` etc. Kotlin doesn't have separate primitive types and their corresponding boxed types like Java.

Note that Kotlin may represent basic types like `Int`, `Char`, `Boolean` etc. as primitive values at runtime to improve performance, but for the end users, all of them are objects.

Since all the data types are objects, the operations on these types are internally represented as function calls.

For example, the addition operation `a + b` between two numbers `a` and `b` is represented as a function call `a.plus(b)` -

```
var a = 4
var b = 5

println(a + b)
```

```
// equivalent to  
println(a.plus(b))
```

All the operators that we looked at in the previous section have a symbolic name which is used to translate any expression containing those operators into the corresponding function calls -

Expression	Translates to
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)
a++	a.inc()
a--	a.dec()
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a += b	a.plusAssign(b)
...	...

You can check out other expressions and their corresponding function calls on [Kotlin's reference page](#).

The concept of translating such expressions to function calls enable operator overloading in Kotlin. For example, you can provide implementation for

the `plus` function in a class defined by you, and then you'll be able to add the objects of that class using `+` operator like this - `object1 + object2`.

Kotlin will automatically convert the addition operation `object1 + object2` into the corresponding function call `object1.plus(object2)` (Think of a `ComplexNumber` class with the `+` operator overloaded).

You'll learn more about operator overloading in a future session.

Note that the operations on basic types like `Int`, `Char`, `Double`, `Boolean` etc. are optimized and do not include the overhead of function calls.

Bitwise Operators

Unlike C, C++ and Java, Kotlin doesn't have bitwise operators like `|` (bitwise-or), `&` (bitwise-and), `^` (bitwise-xor), `<<` (signed left shift), `>>` (signed right shift) etc.

For performing bitwise operations, Kotlin provides following methods that work for `Int` and `Long` types -

- `shl` - signed shift left (equivalent of `<<` operator)
- `shr` - signed shift right (equivalent of `>>` operator)
- `ushr` - unsigned shift right (equivalent of `>>>` operator)
- `and` - bitwise and (equivalent of `&` operator)
- `or` - bitwise or (equivalent of `|` operator)
- `xor` - bitwise xor (equivalent of `^` operator)
- `inv` - bitwise complement (equivalent of `~` operator)

Here are few examples demonstrating how to use above functions -

```
1 shl 2    // Equivalent to 1.shl(2), Result = 4
16 shr 2   // Result = 4
2 and 4    // Result = 0
2 or 3     // Result = 3
4 xor 5    // Result = 1
4.inv()    // Result = -5
```

All the bitwise functions, except `inv()`, can be called using infix notation. The infix notation of `2.and(4)` is `2 and 4`. Infix notation allows you to write function calls in a more intuitive way. We will cover infix notation later on.

Operations on Boolean Types

Kotlin supports following logical operators for performing operations on boolean types -

- `||` - Logical OR
- `&&` - Logical AND
- `!` - Logical NOT

Here are few examples of logical operators -

```
2 == 2 && 4 != 5 // true
4 > 5 && 2 < 7   // false
!(7 > 12 || 14 < 18) // false
```

Logical operators are generally used in [control flow statements](#) like `if`, `if-else`, `while` etc., to test the validity of a condition.

Operations on Strings

String Concatenation

The `+` operator is overloaded for String types. It performs String concatenation -

```
var firstName = "Raj"
var lastName = "johnson"
var fullName = firstName + " " + lastName // "Raj johnson"
```

String Interpolation

Kotlin has an amazing feature called String Interpolation. This feature allows you to directly insert a *template expression* inside a String. Template expressions are tiny pieces of code that are evaluated and their results are concatenated with the original String.

A template expression is prefixed with \$ symbol. Following is an example of String interpolation -

```
var a = 12
var b = 18

println("Avg of $a and $b is equal to ${ (a + b)/2 }")

// Prints - Avg of 12 and 18 is equal to 15
```

If the template expression is a simple variable, you can write it like \$variableName. If it is an expression then you need to insert it inside a \${} block.

Session 5- Kotlin Control Flow: if and when expressions, for and while loops

In this session, you'll learn how to use Kotlin's control flow expressions and statements which includes conditional expressions like if, if-else, when and looping statements like for, while, and do-while.

If Statement

The `If` statement allows you to specify a section of code that is executed only if a given condition is true-

```
var n = 34

if(n % 2 == 0) {

    println("$n is even")
}
```

```
}
```

```
// Displays - "34 is even"
```

The curly braces are optional if the body of `if` statement contains a single line -

```
if(n % 2 == 0) println("$n is even")
```

If-Else Statement

The `if-else` statement executes one section of code if the condition is true and the other if the condition is false -

```
var a = 32
var b = 55

if(a > b) {
    println("max($a, $b) = $a")
} else {
    println("max($a, $b) = $b")
}

// Displays - "max(32, 55) = 55"
```

Using If as an Expression

In Kotlin, You can use `if` as an expression instead of a statement. For example, you can assign the result of an `if-else` expression to a variable.

Let's rewrite the `if-else` example of finding the maximum of two numbers that we saw in the previous section as an expression -

```
var a = 32

var b = 55

var max = if(a > b) a else b

println("max($a, $b) = $max")

// Displays - "max(32, 55) = 55"
```

Note that when you're using `if` as an expression, it is required to have an `else` branch, otherwise, the compiler will throw an error.

The `if-else` branches can also have block bodies. In case of block bodies, the last expression is the value of the block -

```
var a = 32

var b = 55

var max = if(a > b) {

    println("$a is greater than $b")

    a

} else {

    println("$a is less than or equal to $b")

    b

}

println("max($a, $b) = $max")

# Output

32 is less than or equal to 55

max(32, 55) = 55
```

Unlike Java, Kotlin doesn't have a ternary operator because we can easily achieve what ternary operator does, using an `if-else` expression.

If-Else-If Chain

You can chain multiple `if-else-if` blocks like this -

```
var age = 17

if (age < 12) {
    println("Child")
} else if (age in 12..17) {
    println("Teen")
} else if (age in 18..21) {
    println("Young Adult")
} else if (age in 22..30) {
    println("Adult")
} else if (age in 30..50) {
    println("Middle Aged")
} else {
    println("Old")
}

// Displays - "Teen"
```

In the next section, we'll learn how to represent `if-else-if` chain using a `when` expression to make it more concise.

When Expression

Kotlin's `when` expression is the replacement of `switch` statement from other languages like C, C++, and Java. It is concise and more powerful than `switch` statements.

Here is how a `when` expression looks like -

```
var dayOfWeek = 4
when(dayOfWeek) {
    1 -> println("Monday")
    2 -> println("Tuesday")
    3 -> println("Wednesday")
    4 -> println("Thursday")
    5 -> println("Friday")
    6 -> println("Saturday")
    7 -> println("Sunday")
    else -> println("Invalid Day")
}

// Displays - "Thursday"
```

`when` expression matches the supplied argument with all the branches one by one until a match is found. Once a match is found, it executes the matched branch. If none of the branches match, the `else` branch is executed.

In the above example, all the branches contain a single statement. But they can also contain multiple statements enclosed in a block -

```
var dayOfWeek = 1
when(dayOfWeek) {
    1 -> {
        // Block
        println("Monday")
        println("First day of the week")
    }
    7 -> println("Sunday")
    else -> println("Other days")
}
```

```
}
```

Using `when` as an expression

Just like `if`, `when` can be used as an expression and we can assign its result to a variable like so -

```
var dayOfWeek = 4

var dayOfWeekInString = when(dayOfWeek) {
    1 -> "Monday"
    2 -> "Tuesday"
    3 -> "Wednesday"
    4 -> "Thursday"
    5 -> "Friday"
    6 -> "Saturday"
    7 -> "Sunday"
    else -> "Invalid Day"
}

println("Today is $dayOfWeekInString") // Today is Thursday
```

Combining multiple `when` branches into one using comma

You can combine multiple branches into one using comma. This is helpful when you need to run a common logic for multiple cases -

```
var dayOfWeek = 6
```



```

when(dayOfWeek) {
    1, 2, 3, 4, 5 -> println("Weekday")
    6, 7 -> println("Weekend")
    else -> println("Invalid Day")
}

// Displays - Weekend

```

Checking whether a given value is in a range or not using `in` operator

A range is created using the `..` operator. For example, you can create a range from 1 to 10 using `1..10`. You'll learn more about `range` in a future session.

The `in` operator allows you to check if a value belongs to a range/collection -

```

var dayOfMonth = 5
when(dayOfMonth) {
    in 1..7 -> println("We're in the first Week of the Month")
    !in 15..21 -> println("We're not in the third week of the Month")
    else -> println("none of the above")
}

// Displays - We're in the first Week of the Month

```

Checking whether a given variable is of certain type or not using `is` operator

```

var x : Any = 6.86
when(x) {

```

```
is Int -> println("$x is an Int")

is String -> println("$x is a String")

!is Double -> println("$x is not Double")

else -> println("none of the above")

}

// Displays - none of the above
```

Using when as a replacement for an if-else-if chain

```
var number = 20

when {

    number < 0 -> println("$number is less than zero")

    number % 2 == 0 -> println("$number is even")

    number > 100 -> println("$number is greater than 100")

    else -> println("None of the above")

}

// Displays - 20 is even
```

While Loop

While loop executes a block of code repeatedly as long as a given condition is true -

```
while(condition) {

    // code to be executed

}
```

Here is an example -

```

var x = 1
while(x <= 5) {
    println("$x ")
    x++
}

// Displays - 1 2 3 4 5

```

In the above example, we increment the value of x by 1 in each iteration. When x reaches 6, the condition evaluates to false and the loop terminates.

do-while loop

The do-while loop is similar to while loop except that it tests the condition at the end of the loop.

```

var x = 1
do {
    print("$x ")
    x++
} while(x <= 5)

// Displays - 1 2 3 4 5

```

Since do-while loop tests the condition at the end of the loop. It is executed at least once -

```

var x = 6
do {
    print("$x ")
    x++
} while(x <= 5)

```

```
} while(x <= 5)

// Displays - 6
```

For Loop

A for-loop is used to iterate through ranges, arrays, collections, or anything that provides an iterator (You'll learn about iterator in a future session).

Iterating through a range

```
for(value in 1..10) {
    print("$value ")
}

// Displays - 1 2 3 4 5 6 7 8 9 10
```

Iterating through an array

```
var primeNumbers = intArrayOf(2, 3, 5, 7, 11)

for(number in primeNumbers) {
    print("$number ")
}

// Displays - 2, 3, 5, 7, 11
```

Iterating through an array using its indices

Every array in Kotlin has a property called `indices` which returns a range of valid indices of that array.

You can iterate over the indices of the array and retrieve each array element using its index like so -

```
var primeNumbers = intArrayOf(2, 3, 5, 7, 11)

for(index in primeNumbers.indices) {
    println("PrimeNumber(${index+1}): ${primeNumbers[index]}")
}

# Output

PrimeNumber(1): 2
PrimeNumber(2): 3
PrimeNumber(3): 5
PrimeNumber(4): 7
PrimeNumber(5): 11
```

Iterating through an array using withIndex()

You can use the `withIndex()` function on arrays to obtain an iterable of `IndexedValue` type. This allows you to access both the index and the corresponding array element in each iteration -

```
var primeNumbers = intArrayOf(2, 3, 5, 7, 11)

for((index, number) in primeNumbers.withIndex()) {
    println("PrimeNumber(${index+1}): $number")
}
```

The output of this snippet is same as the previous snippet.

Break and Continue

Break out of a loop using the `break` keyword

```
for (num in 1..100) {  
    if (num%3 == 0 && num%5 == 0) {  
        println("First positive no divisible by both 3 and 5: ${num}")  
        break  
    }  
}  
  
# Output  
First positive no divisible by both 3 and 5: 15
```

Skip to the next iteration of a loop using the `continue` keyword

```
for (num in 1..10) {  
    if (num%2 == 0) {  
        continue;  
    }  
    print("${num} ")  
}  
  
# Output  
1 3 5 7 9
```

Session 6- Nullable Types and Null Safety in Kotlin

If you have been programming in Java or any other language that has the concept of null reference then you must have heard about or experienced `NullPointerException` in your programs.

`NullPointerException`s are Runtime Exceptions which are thrown by the program at runtime causing application failure and system crashes.

Wouldn't it be nice if we could detect possible `NullPointerException` exception errors at compile time itself and guard against them?

Well, Enter Kotlin!

Nullability and Nullable Types in Kotlin

Kotlin supports nullability as part of its type System. That means, you have the ability to declare whether a variable can hold a null value or not.

By supporting nullability in the type system, the compiler can detect possible `NullPointerException` errors at compile time and reduce the possibility of having them thrown at runtime.

Let's understand how it works!

All variables in Kotlin are non-nullable by default. So If you try to assign a null value to a regular variable, the compiler will throw an error -

```
var greeting: String = "Hello, World"
greeting = null // Compilation Error
```

To allow null values, you have to declare a variable as nullable by appending a question mark in its type declaration -

```
var nullableGreeting: String? = "Hello, World"
nullableGreeting = null // Works
```

We know that `NullPointerException` occurs when we try to call a method or access a property on a variable which is `null`. Kotlin disallows method calls and property access on nullable variables and thereby prevents many possible `NullPointerExceptions`.

For example, The following method access works because Kotlin knows that the variable `greeting` can never be null -

```
val len = greeting.length
val upper = greeting.toUpperCase()
```

But the same method call won't work with `nullableGreeting` variable -

```
val len = nullableGreeting.length // Compilation Error
val upper = nullableGreeting.toUpperCase() // Compilation Error
```

Since Kotlin knows beforehand which variable can be null and which cannot, It can detect and disallow calls which could result in `NullPointerException` at compile-time itself.

Working with Nullable Types

All right, it's nice that Kotlin disallows method calls and property access on nullable variables to guard against `NullPointerException` errors. But we still need to do that right?

Well, there are several ways of **safely** doing that in Kotlin.

1. Adding a null Check

The most trivial way to work with nullable variables is to perform a null check before accessing a property or calling a method on them -

```
val nullableName: String? = "John"

if(nullableName != null) {
    println("Hello, ${nullableName.toUpperCase()}")
    println("Your name is ${nullableName.length} characters long.")
} else {
    println("Hello, Guest")
}
```



```
}
```

Once you perform a null comparison, the compiler remembers that and allows calls to `toUpperCase()` and `length` inside the `if` branch.

2. Safe call operator: ?.

Null Comparisons are simple but too verbose. Kotlin provides a Safe call operator, `?.` that reduces this verbosity. It allows you to combine a null-check and a method call in a single expression.

For example, The following expression -

```
nullableName?.toUpperCase()
```

is same as -

```
if(nullableName != null)
    nullableName.toUpperCase()
else
    null
```

Wow! That saves a lot of keystrokes, right? :-)

So if you were to print the name in uppercase and its length **safely**, you could do the following -

```
val nullableName: String? = null

println(nullableName?.toUpperCase())
println(nullableName?.length)

// Prints
null
```

```
null
```

That printed `null` since the variable `nullableName` is null, otherwise, it would have printed the name in uppercase and its length.

But what if you don't want to print anything if the variable is `null`?

Well, **To perform an operation only if the variable is not null, you can use the safe call operator with `let`** -

```
val nullableName: String? = null

nullableName?.let { println(it.toUpperCase()) }
nullableName?.let { println(it.length) }

// Prints nothing
```

The lambda expression inside `let` is executed only if the variable `nullableName` is not null.

That's great but that's not all. Safe call operator is even more powerful than you think. For example, **You can chain multiple safe calls** like this -

```
val currentCity: String? = user?.address?.city
```

The variable `currentCity` will be null if any of `user`, `address` or `city` is null. *(Imagine doing that using null-checks.)*

3. Elvis operator: `?:`

The Elvis operator is used to provide a default value when the original variable is `null` -

```
val name = nullableName ?: "Guest"
```

The above expression is same as -

```
val name = if(nullableName != null) nullableName else "Guest"
```

In other words, The Elvis operator takes two values and returns the first value if it is not null, otherwise, it returns the second value.

The Elvis operator is often used with Safe call operator to provide a default value other than `null` when the variable on which a method or property is called is `null` -

```
val len = nullableName?.length ?: -1
```

You can have more complex expressions on the left side of Elvis operator -

```
val currentCity = user?.address?.city ?: "Unknown"
```

Moreover, you can use `throw` and `return` expressions on the right side of Elvis operator. This is very useful while checking preconditions in a function. So instead of providing a default value in the right side of Elvis operator, you can throw an exception like this -

```
val name = nullableName ?: throw IllegalArgumentException("Name can not be null")
```

4. Not null assertion : !! Operator

The !! operator converts a nullable type to a non-null type, and throws a `NullPointerException` if the nullable type holds a null value.

So it's a way of asking for `NullPointerException` explicitly. Please don't use this operator.

```
val nullableName: String? = null

nullableName!!.toUpperCase() // Results in NullPointerException
```

Null Safety and Java Interoperability

Kotlin is fully interoperable with Java but Java doesn't support nullability in its type system. So what happens when you call Java code from Kotlin?

Well, Java types are treated specially in Kotlin. They are called **Platform types**. Since Kotlin doesn't have any information about the nullability of a type declared in Java, It relaxes compile-time null checks for these types.

So you don't get any null safety guarantee for types declared in Java, and you have full responsibility for operations you perform on these types. The compiler will allow all operations. If you know that the Java variable can be null, you should compare it with null before use, otherwise, just like Java, you'll get a `NullPointerException` at runtime if the value is null.

Consider the following User class declared in Java -

```
public class User {

    private final String name;

    public User(String name) {

        this.name = name;

    }

    public String getName() {

        return name;

    }

}
```

Since Kotlin doesn't know about the nullability of the member variable `name`, It allows all operations on this variable. You can treat it as nullable or non-nullable, but the compiler won't enforce anything.

In the following example, we simply treat the variable `name` as non-nullable and call methods and properties on it -

```
val javaUser = User(null)

println(javaUser.name.toUpperCase()) // Allowed (Throws NullPointerException)
println(javaUser.name.length) // Allowed (Throws NullPointerException)
```

The other option is to treat the member variable `name` as nullable and use the safe operator for calling methods or accessing properties -

```
val javaUser = User(null)

println(javaUser.name?.toUpperCase()) // Allowed (Prints null)
println(javaUser.name?.length) // Allowed (Prints null)
```

Nullability Annotations

Although Java doesn't support nullability in its type system, You can use annotations like `@Nullable` and `@NotNull` provided by external packages like `javax.validation.constraints`, `org.jetbrains.annotations` etc to mark a variable as Nullable or Not-null.

Java compiler doesn't use these annotations, but these annotations are used by IDEs, ORM libraries and other external tools to provide assistance while working with null values.

Kotlin also respects these annotations when they are present in Java code. Java types which have these nullability annotations are represented as actual nullable or non-null Kotlin types instead of platform types.

Nullability and Collections

Kotlin's collection API is built on top of Java's collection API but it fully supports nullability on Collections.

Just as regular variables are non-null by default, a normal collection also can't hold null values -

```
val regularList: List<Int> = listOf(1, 2, null, 3) // Compiler Error
```

1. Collection of Nullable Types

Here is how you can declare a Collection of Nullable Types in Kotlin -

```
val listOfNullableTypes: List<Int?> = listOf(1, 2, null, 3) // Works
```

To filter non-null values from a list of nullable types, you can use the `filterNotNull()` function -

```
val notNullList: List<Int> = listOfNullableTypes.filterNotNull()
```

2. Nullable Collection

Note that there is a difference between a collection of nullable types and a nullable collection.

A collection of nullable types can hold null values but the collection itself cannot be null -

```
var listOfNullableTypes: List<Int?> = listOf(1, 2, null, 3) // Works  
listOfNullableTypes = null // Compilation Error
```

You can declare a nullable collection like this -

```
var nullableList: List<Int>? = listOf(1, 2, 3)
```

```
nullableList = null // Works
```

3. Nullable Collection of Nullable Types

Finally, you can declare a nullable collection of nullable types like this -

```
var nullableListOfNullableTypes: List<Int?>? = listOf(1, 2, null, 3) // Works  
nullableListOfNullableTypes = null // Works
```

Session 7- Kotlin Functions, Default and Named Arguments, Varargs and Function Scopes

Functions are the basic building block of any program. In this session, you'll learn how to declare and call functions in Kotlin. You'll also learn about Function scopes, Default arguments, Named Arguments, and Varargs.

Defining and Calling Functions

You can declare a function in Kotlin using the `fun` keyword. Following is a simple function that calculates the average of two numbers -

```
fun avg(a: Double, b: Double): Double {  
    return (a + b)/2  
}
```

Calling a function is simple. You just need to pass the required number of parameters in the function name like this -

```
avg(4.6, 9.0) // 6.8
```

Following is the general syntax of declaring a function in Kotlin.

```
fun functionName(param1: Type1, param2: Type2, ..., paramN: TypeN): Type {  
    // Method Body  
}
```

Every function declaration has a function name, a list of comma-separated parameters, an optional return type, and a method body. The function parameters must be explicitly typed.

Single Expression Functions

You can omit the return type and the curly braces if the function returns a single expression. The return type is inferred by the compiler from the expression -

```
fun avg(a: Double, b: Double) = (a + b)/2  
avg(10.0, 20.0) // 15.0
```

Note that, unlike other statically typed languages like Scala, Kotlin does not infer return types for functions with block bodies. Therefore, functions with block body must always specify return types explicitly.

Unit returning Functions

Functions which don't return anything has a return type of `Unit`. The `Unit` type corresponds to `void` in Java.

```
fun printAverage(a: Double, b: Double): Unit {  
    println("Avg of ($a, $b) = ${(a + b)/2}")  
}  
  
printAverage(10.0, 30.0) // Avg of (10.0, 30.0) = 20.0
```

Note that, the `Unit` type declaration is completely optional. So you can also write the above function declaration like this -


```
fun printAverage(a: Double, b: Double) {  
    println("Avg of ($a, $b) = ${ (a + b)/2}")  
}
```

Function Default Arguments

Kotlin supports default arguments in function declarations. You can specify a default value for a function parameter. The default value is used when the corresponding argument is omitted from the function call.

Consider the following function for example -

```
fun displayGreeting(message: String, name: String = "Guest") {  
    println("Hello $name, $message")  
}
```

If you call the above function with two arguments, it works just like any other function and uses the values passed in the arguments -

```
displayGreeting("Welcome to the CalliCoder Blog", "John") // Hello John,  
Welcome to the CalliCoder Blog
```

However, If you omit the argument that has a default value from the function call, then the default value is used in the function body -

```
displayGreeting("Welcome to the Coding Bootcamps School") // Hello Guest,  
Welcome to the Coding Bootcamps School
```

If the function declaration has a default parameter preceding a non-default parameter, then the default value cannot be used while calling the function with position-based arguments.

Consider the following function -

```
fun arithmeticSeriesSum(a: Int = 1, n: Int, d: Int = 1): Int {  
    return n/2 * (2*a + (n-1)*d)  
}
```

While calling the above function, you can not omit the argument `a` from the function call and selectively pass a value for the non-default parameter `n` -

```
arithmeticSeriesSum(10) // error: no value passed for parameter n
```

When you call a function with position-based arguments, the first argument corresponds to the first parameter, the second argument corresponds to the second parameter, and so on.

So for passing a value for the 2nd parameter, you need to specify a value for the first parameter as well -

```
arithmeticSeriesSum(1, 10) // Result = 55
```

However, the above use-case of selectively passing a value for a parameter is solved by another feature of Kotlin called *Named Arguments*.

Function Named Arguments

Kotlin allows you to specify the names of arguments that you're passing to the function. This makes the function calls more readable. It also allows you to pass the value of a parameter selectively if other parameters have default values.

Consider the following `arithmeticSeriesSum()` function that we defined in the previous section -

```
fun arithmeticSeriesSum(a: Int = 1, n: Int, d: Int = 1): Int {  
    return n/2 * (2*a + (n-1)*d)  
}
```

You can specify the names of arguments while calling the function like this -

```
arithmeticSeriesSum(n=10) // Result = 55
```

The above function call will use the default values for parameters `a` and `d`.

Similarly, you can call the function with all the parameters like this -

```
arithmeticSeriesSum(a=3, n=10, d=2) // Result = 120
```

You can also reorder the arguments if you're specifying the names -

```
arithmeticSeriesSum(n=10, d=2, a=3) // Result = 120
```

You can use a mix of named arguments and position-based arguments as long as all the position-based arguments are placed before the named arguments -

```
arithmeticSeriesSum(3, n=10) // Result = 75
```

The following function call is not allowed since it contains position-based arguments after named arguments -

```
arithmeticSeriesSum(n=10, 2) // error: mixing named and positioned arguments  
is not allowed
```

Variable Number of Arguments (Varargs)

You can pass a variable number of arguments to a function by declaring the function with a `vararg` parameter.

Consider the following `sumOfNumbers()` function which accepts a `vararg` of numbers -

```
fun sumOfNumbers(vararg numbers: Double): Double {  
    var sum: Double = 0.0  
    for(number in numbers) {  
        sum += number  
    }  
    return sum  
}
```

You can call the above function with any number of arguments -

```
sumOfNumbers(1.5, 2.0) // Result = 3.5
```

```
sumOfNumbers(1.5, 2.0, 3.5, 4.0, 5.8, 6.2) // Result = 23.0
```

```
sumOfNumbers(1.5, 2.0, 3.5, 4.0, 5.8, 6.2, 8.1, 12.4, 16.5) // Result = 60.0
```

In Kotlin, a `vararg` parameter of type `T` is internally represented as an array of type `T` (`Array<T>`) inside the function body.

A function may have only one `vararg` parameter. If there are other parameters following the `vararg` parameter, then the values for those parameters can be passed using the named argument syntax -

```
fun sumOfNumbers(vararg numbers: Double, initialSum: Double): Double {  
    var sum = initialSum  
    for(number in numbers) {  
        sum += number  
    }  
}
```

```
    }  
    return sum  
}  
  
sumOfNumbers(1.5, 2.5, initialSum=100.0) // Result = 104.0
```

Spread Operator

Usually, we pass the arguments to a `vararg` function one-by-one. But if you already have an array and want to pass the elements of the array to the `vararg` function, then you can use the **spread** operator like this -

```
val a = doubleArrayOf(1.5, 2.6, 5.4)  
  
sumOfNumbers(*a) // Result = 9.5
```

Function Scope

Kotlin supports functional programming. Functions are first-class citizens in the language.

Unlike Java where every function needs to be encapsulated inside a class, Kotlin functions can be defined at the top level in a source file.

In addition to top-level functions, you also have the ability to define member functions, local functions, and extension functions.

1. Top Level Functions

Top level functions in Kotlin are defined in a source file outside of any class. They are also called package level functions because they are a member of the package in which they are defined.

The `main()` method itself is a top-level function in Kotlin since it is defined outside of any class.

Let's now see an example of a top-level function. Check out the following `findNthFibonacciNo()` function which is defined inside a package named `maths` -

```
package maths

fun findNthFibonacciNo(n: Int): Int {
    var a = 0
    var b = 1
    var c: Int

    if(n == 0) {
        return a
    }

    for(i in 2..n) {
        c = a+b
        a = b
        b = c
    }
    return b
}
```

You can access the above function directly inside the `maths` package -

```
package maths
```

```
fun main(args: Array<String>) {
    println("10th fibonacci number is - ${findNthFibonacciNo(10)}")
}

//Outputs - 10th fibonacci number is - 55
```

However, If you want to call the `findNthFibonacciNo()` function from other packages, then you need to import it as in the following example -

```
package test

import maths.findNthFibonacciNo

fun main(args: Array<String>) {
    println("10th fibonacci number is - ${findNthFibonacciNo(10)}")
}
```

2. Member Functions

Member functions are functions which are defined inside a class or an object.

```
class User(val firstName: String, val lastName: String) {

    // Member function

    fun getFullName(): String {
        return firstName + " " + lastName
    }
}
```

Member functions are called on the objects of the class using the `dot(.)` notation -

```
val user = User("Bill", "Gates") // Create an object of the class
println("Display Name : ${user.getFullName()}") // Call the member function
```

3. Local/Nested Functions

Kotlin allows you to nest function definitions. These nested functions are called Local functions. Local functions bring more encapsulation and readability to your program -

```
fun findBodyMassIndex(weightInKg: Double, heightInCm: Double): Double {
    // Validate the arguments
    if(weightInKg <= 0) {
        throw IllegalArgumentException("Weight must be greater than zero")
    }
    if(heightInCm <= 0) {
        throw IllegalArgumentException("Height must be greater than zero")
    }

    fun calculateBMI(weightInKg: Double, heightInCm: Double): Double {
        val heightInMeter = heightInCm / 100
        return weightInKg / (heightInMeter * heightInMeter)
    }

    // Calculate BMI using the nested function
    return calculateBMI(weightInKg, heightInCm)
}
```

Local functions can access local variables of the outer function. So the above function is equivalent to the following -


```

fun findBodyMassIndex(weightInKg: Double, heightInCm: Double): Double {
    if (weightInKg <= 0) {
        throw IllegalArgumentException("Weight must be greater than zero")
    }

    if (heightInCm <= 0) {
        throw IllegalArgumentException("Height must be greater than zero")
    }

    // Nested function has access to the local variables of the outer
    function

    fun calculateBMI(): Double {
        val heightInMeter = heightInCm / 100

        return weightInKg / (heightInMeter * heightInMeter)
    }

    return calculateBMI()
}

```

Session 8- Kotlin Infix Notation - Make function calls more intuitive

Kotlin supports method calls of a special kind, called *infix calls*.

You can mark any member function or extension function with the *infix* modifier to allow it to be called using infix notation. The only requirement is that the function should have only one required parameter.

Infix notations are used extensively in Kotlin. If you've been programming in Kotlin, chances are that you've already used infix notations.

Following are few common examples of infix notations in Kotlin -

1. Infix Notation Example - Creating a Map

```
val map = mapOf(1 to "one", 2 to "two", 3 to "three")
```

In the above example, the expressions `1 to "one"`, `2 to "two"` etc, are infix notations of the function calls `1.to("one")` and `2.to("two")` etc.

`to()` is an infix function that creates a `Pair<A, B>` from two values.

2. Infix Notation Example - Range Operators (until, downTo, step)

Kotlin provides various range operators that are usually called using infix notation -

```
for(i in 1 until 10) { // Same as - for(i in 1.until(10))
    print("$i ")
}

for(i in 10 downTo 1) { // Same as - for(i in 10.downTo(1))
    print("$i ")
}

for(i in 1 until 10 step 2) { // Same as - for(i in 1.until(10).step(2))
    print("$i ")
}
```

3. Infix Notation Example - String.matches()

The `String.matches()` function in Kotlin which matches a `String` with a `Regex` is an infix function -

```
val regex = Regex("[tT]rue|[yY]es")
val str = "yes"
```

```
str.matches(regex)

// Infix notation of the above function call -
str matches regex
```

Creating an Infix Function

You can make a single argument member function or extension function, an infix function by marking it with the `infix` keyword.

Check out the following example where I have created an infix member function called `add()` for adding two Complex numbers -

```
data class ComplexNumber(val realPart: Double, val imaginaryPart: Double) {
    // Infix function for adding two complex numbers
    infix fun add(c: ComplexNumber): ComplexNumber {
        return ComplexNumber(realPart + c.realPart, imaginaryPart +
c.imaginaryPart)
    }
}
```

You can now call the `add()` method using infix notation -

```
val c1 = ComplexNumber(3.0, 5.0)
val c2 = ComplexNumber(4.0, 7.0)

// Usual call
c1.add(c2) // produces - ComplexNumber(realPart=7.0, imaginaryPart=12.0)
```

```
// Infix call  
c1 add c2 // produces - ComplexNumber(realPart=7.0, imaginaryPart=12.0)
```