



KOTLIN

Ashirbad Swain



Kotlin Programming

Kotlin is a relatively new programming language developed by JetBrains for modern multiplatform applications. Now a days, Kotlin is widely used for Android development instead of Java. It is because Kotlin is safe, concise and fun to read and write.

Kotlin Features:

- Statically typed, object-oriented, Modern programming language
- Properties and extensions for classes
- Created from developer for developers
- Concise, much less boilerplate code than some other languages
- Increased null safety with nullable and non-nullable data types
- Open sourced
- Supports lambdas and higher-order functions
- Fully compatible with Java language, so that you can migrate over time and continue using your favorite libraries
- Officially supported for Android development, and included with IntelliJ and Android studio

Features of Kotlin Programming

- **Open Source**
Kotlin is distributed under Apache License, Version 2.0. The Kompiler (Kotlin Compiler), IntelliJ IDEA plugin, enhancements to basic Java libraries and build tools all are open source.
- **Interoperable**
Kotlin is 100 percent interoperable with Java. This means all your current Java/Android code works seamlessly with Kotlin.
- **Concise and Expressive**
Compared to Java, Kotlin code are much more concise. Also, Kotlin code is much more expressive (easier to understand and write).
- **Tool-friendly**
Kotlin is developed by JetBrains, the company renowned for creating development tools. You can choose any Java IDE to write Kotlin code.
- **Easy to Learn**
Learning Kotlin is easy if you know other programming languages like Java, Scala, Groovy, C#, JavaScript and Gosu.
- **Safe**
Kotlin is a statically typed language. Hence, the type checking occurs at compile-time as opposed to run-time and trivial bugs are caught at an early stage.

Why Learn Kotlin?

- Kotlin is 100 percent interoperable with Java. Hence your Java/Android code works with Kotlin.
- Kotlin allows you to cut off the lines of the code by approximately 40% (compared to Java).
- Learning Kotlin is easy. It is particularly easy if you already know Java.
- Kotlin is total-friendly. You can use any Java IDE or command line to run Kotlin.



Kotlin Introduction

Kotlin Hello World – Your First Kotlin Program

A “Hello, World!” is a simple program that outputs **Hello, World!** on the screen. Since it’s a very simple program, it’s often used to introduce a new programming language.

“Hello, World!” Program

```
// Hello World Program

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

When you run the program, the output will be:

```
Hello, World!
```

How this program works?

1. `// Hello World Program`
Any line starting with `//` is a comment in Kotlin (Similar to Java). Comments are ignored by the compiler. They are intended for person reading the code to better understand the intent and functionality of the program.
2. `fun main (args : Array <String>) { }`
This is the main function, which is mandatory in every Kotlin application. The Kotlin compiler starts executing the code from the `main` function.

The function takes array of strings as a parameter and and return Unit.
Just remember that main function is a mandatory function which is the entry point of every Kotlin program. The signature of main function is:

```
fun main (args : Array<String>) {

    ... ..

}
```

3. `println (“Hello, World!”)`
The `println ()` function prints the given message inside the quotation marks and newline to the standard output stream. In this program, it prints Hello, World! and new line.



Few Important Notes

1. Unlike Java, it is not mandatory to create a `class` in every Kotlin program. It's because the Kotlin compiler creates the class for us.
2. The `println ()` function calls `System.out.println ()` internally.

Kotlin Data Types

A variable is a location in memory (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier).

How to declare a variable in Kotlin?

To declare a variable in Kotlin, either `var` or `val` keyword is used.

Example:

```
var language = "French"
```

```
val score = 95
```

Here, `language` is a variable of type `String`, and `score` is a variable of type `int`. You don't have to specify the type of variables; Kotlin implicitly does that for you.

The compiler knows this by initializer expression ("French" is a string, and 95 is an integer value in the above program). This is called `type inference` in programming.

However, you can explicitly specify the type if you want to:

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

```
val score: Int = 95
```

we have initialized variable during declaration in above examples. However, it's not necessary. You can declare variable and specify its type in one statement, and initialize the variable in another statement later in the program.

```
var language: String    // variable declaration of type String
```

```
language = "French"    // variable initialization
```

```
val score: Int         // variable declaration of type Int
```

```
score = 95             // variable initialization
```

here are few examples that results into error.



```
var language      // Error  
  
language = "French"
```

here, the type of `language` variable is not explicitly specified, nor the variable is initialized during declaration.

```
var language: String  
  
language = 14      // Error
```

here, we are trying to assign 14 (integer value) to variable of different type (string).

Difference Between var and val

val (immutable reference) – The variable declared using `val` keyword cannot be changed once the value is assigned. It is similar to final variable in Java.

var (Mutable reference) – The variable declared using `var` keyword can be changed later in the program. It corresponds to regular Java variable.

Here are few examples:

```
var language = "French"  
  
language = "German"
```

here, `language` variable is assigned to German. Since, the variable is declared using `var`, this code work perfectly.

```
val language = "French"  
  
language = "German"  // Error
```

You cannot reassign `language` variable to `German` in the above example because the variable is declared using `val`.



Kotlin Basic Types

Kotlin is a statically typed language like Java. That is, the type of a variable is known during the compile time. For example,

```
val language: Int
```

```
val marks = 12.3
```

here, the compiler knows that `language` is of type `int`, and `marks` is of type `Double` before the compile time.

The built-in types in Kotlin can be categorized as:

- Numbers
- Characters
- Booleans
- Arrays

Number Type

Numbers in Kotlin are similar to Java. There are 6 built-in types representing numbers

- Byte
- Short
- Int
- Long
- Float
- Double

Byte

The `Byte` data type can have values from `-128 to 127` (8-bit signed two's complement integer).

It is used instead of `int` or other integer data types to save memory if its certain that the value of a variable will be within `[-128 to 127]`.

```
fun main (args : Array<String>) {  
    val range: Byte = 112  
    println("$range")  
  
    // The code below gives error. Why?  
    // val range1: Byte = 200  
}
```

When you run the program, the output will be:

```
112
```



Short

The `Short` data type can have values from `-32768` to `32767` (16-bit signed two's complement integer).

It is used instead of other integer data types to save memory if it's certain that the value of the variable will be within `[-32768 to 32767]`.

Examples:

```
fun main (args : Array<String>) {  
  
    val temperature: Short = -11245  
    println("$temperature")  
}
```

When you run the program, the output will be:

```
-11245
```

Int

The `int` data type can have values from `-231` to `231 - 1` (32-bit signed two's complement integer).

Example:

```
fun main (args : Array<String>) {  
  
    val score: Int = 100000  
    println("$score")  
}
```

When you run the program, the output will be:

```
100000
```



If you assign an integer between -2^{31} to $2^{31} - 1$ to a variable without explicitly specifying its type, the variable will be of `int` type. For example,

```
fun main (args : Array<String>) {  
  
    // score is of type Int  
    val score = 10  
    println("$score")  
}
```

Long

The `Long` data type can have values from -2^{63} to $2^{63} - 1$ (64-bit signed two's complement integer).

Example:

```
fun main (args : Array<String>) {  
  
    val highestScore: Long = 9999  
    println("$highestScore")  
}
```

When you run the program, the output will be:

```
9999
```

If you assign an integer value greater than $2^{63} - 1$ or less than -2^{63} to a variable (without explicitly specifying its type), the variable will be of `Long` type. For example,

```
val distance = 10000000000 // distance variable of type Long
```

similarly, you can use capital letter '`L`' to specify that the variable is of type `Long`. For Example,

```
val distance = 100L // distance value of type Long
```




Double

The `Double` type is a double-precision 64-bit floating point.

Example:

```
fun main (args : Array<String>) {  
  
    // distance is of type Double  
    val distance = 999.5  
    println("$distance")  
}
```

When you run the program, the output will be:

```
999.5
```

Float

The `Float` data type is a single-precision 32-bit floating point.

Example:

```
fun main (args : Array<String>) {  
  
    // distance is of type Float  
  
    val distance = 19.5F  
  
    println("$distance")  
}
```

When you run the program, the output will be:

```
19.5
```

Notice that, we have used `19.5F` instead of `19.5` in the above program. It is because `19.5` is a `Double` literal, and you cannot assign `Double` value to a variable of type `float`.

To tell the compiler to treat `19.5` as `Float`, you need to use `'F'` at the end.



If you are not sure what number value a variable will be assigned in the program, you can specify it as **Number** type. This allows you to assign **both integer and floating-point** value to the variable (one at a time).

For Example:

```
fun main (args : Array<String>) {  
  
    var test: Number = 12.2  
    println("$test")  
  
    test = 12  
    // Int smart cast from Number  
    println("$test")  
  
    test = 120L  
    // Long smart cast from Number  
    println("$test")  
}
```

When you run the program, the output will be:

```
12.2  
12  
120
```

Char

To represent a character in Kotlin, **char** types are used. Unlike Java, **char** types cannot be treated as numbers.

```
fun main (args : Array<String>) {  
  
    val letter: Char  
    letter = 'k'  
    println("$letter")  
}
```

When you run the program, the output will be:

```
k
```



Boolean

The **Boolean** data types has two possible values, either **true** or **false**.

Booleans are used in decision making statements.

Example:

```
fun main (args : Array<String>) {  
  
    val flag = true  
    println("$flag")  
}
```

Kotlin Arrays

An **array** is a container that holds data (values) of **one single type**. For example, you can create an array that can hold 100 values of **Int** type.

In Kotlin, arrays are represented by the **Array** class. The class has **get** and **set** functions, **size** property, and a few other useful member functions.

To create an array, we can use a library function **arrayOf ()** and pass the item values to it, so that **arrayOf (1, 2, 3)** creates an array [1, 2, 3]. Alternatively, the **arrayOfNulls ()** library function can be used to create an array of a given size filled with null elements.

Kotlin Strings

A **string** is nothing but a **sequence of characters**. This **String** class represents character strings.

In Kotlin, strings are represented by the **String** class. The string literals such as **"this is a string"** is implemented as an instance of this class.

Syntax:

```
val myString = "Hey there!"
```



Kotlin Operators

Kotlin has a set of operators to perform arithmetic, assignment, comparison operators and more.

Operators are special symbols (characters) that carry out operations on operands (variables and values). For example, + is an operator that performs addition.

Arithmetic Operators

Here's a list of arithmetic operators in Kotlin:

Kotlin Arithmetic Operators

Operator	Meaning
+	Addition (also used for string concatenation)
-	Subtraction Operator
*	Multiplication Operator
/	Division Operator
%	Modulus Operator

Example: Arithmetic Operators

```
fun main (args: Array<String>) {  
  
    val number1 = 12.5  
    val number2 = 3.5  
    var result: Double  
  
    result = number1 + number2  
    println ("number1 + number2 = $result")  
  
    result = number1 - number2  
    println ("number1 - number2 = $result")  
  
    result = number1 * number2  
    println ("number1 * number2 = $result")  
  
    result = number1 / number2  
    println ("number1 / number2 = $result")  
}
```



```
result = number1 % number2
println ("number1 % number2 = $result")
}
```

When you run the program, the output will be:

```
number1 + number2 = 16.0
number1 - number2 = 9.0
number1 * number2 = 43.75
number1 / number2 = 3.5714285714285716
number1 % number2 = 2.0
```

The **+** operator is also used for the concatenation of String values.

Example: Concatenation of Strings

```
fun main (args: Array<String>) {

    val start = "Talk is cheap. "
    val middle = "Show me the code. "
    val end = "- Linus Torvalds"

    val result = start + middle + end
    println(result)
}
```

When you run the code, the output will be:

```
Talk is cheap. Show me the code. - Linus Torvalds
```



How arithmetic operators actually work?

Suppose, you are using + operator to add two numbers a and b.

Under the hood, the expression `a + b` calls `a.plus(b)` member function. The `plus` operator is overloaded to work with String values and other basic data types (except Char and Boolean).

```
// + operator for basic types

operator fun plus (other: Byte): Int

operator fun plus (other: Short): Int

operator fun plus (other: Int): Int

operator fun plus (other: Long): Long

operator fun plus (other: Float): Float

operator fun plus (other: Double): Double

// for string concatenation

operator fun String?.plus(other: Any?): String
```

You can also use + operator to work with user-defined types (like objects) by overloading `plus ()` function.

Here's a table of arithmetic operators and their corresponding functions:

Expression	Function name	Translates to
<code>a + b</code>	<code>plus</code>	<code>a.plus(b)</code>
<code>a - b</code>	<code>minus</code>	<code>a.minus(b)</code>
<code>a * b</code>	<code>times</code>	<code>a.times(b)</code>
<code>a / b</code>	<code>div</code>	<code>a.div(b)</code>
<code>a % b</code>	<code>mod</code>	<code>a.mod(b)</code>



Assignment Operators

Assignment operators are used to assign value to a variable. We have already used simple assignment operator = before.

```
val age = 5
```

Here, 5 is assigned to variable age using = operator.

Here's a list of all assignment operators and their corresponding functions:

Expression	Equivalent to	Translates to
<code>a += b</code>	<code>a = a + b</code>	<code>a.plusAssign(b)</code>
<code>a -= b</code>	<code>a = a - b</code>	<code>a.minusAssign(b)</code>
<code>a *= b</code>	<code>a = a * b</code>	<code>a.timesAssign(b)</code>
<code>a /= b</code>	<code>a = a / b</code>	<code>a.divAssign(b)</code>
<code>a %= b</code>	<code>a = a % b</code>	<code>a.modAssign(b)</code>

Example: Assignment Operator

```
fun main (args: Array<String>) {  
    var number = 12  
  
    number *= 5 // number = number*5  
    println ("number = $number")  
}
```

```
number = 60
```



Unary prefix and Increment/Decrement Operators

Here's a table of unary operators, their meaning, and corresponding functions:

Operator	Meaning	Expression	Translates to
+	Unary plus	+a	a.unaryPlus()
-	Unary minus (inverts sign)	-a	a.unaryMinus()
!	not (inverts value)	!a	a.not()
++	Increment: increases value by 1	++a	a.inc()
--	Decrement: decreases value by 1	--a	a.dec()

Example: Unary Operators

```
fun main (args: Array<String>) {  
    val a = 1  
    val b = true  
    var c = 1  
    var result: Int  
    var booleanResult: Boolean  
  
    result = -a  
    println ("-a = $result")  
    booleanResult = !b  
    println ("!b = $booleanResult")  
    --c  
    println ("--c = $c")  
}
```

```
-a = -1  
!b = false  
--c = 0
```




Comparison and Equality Operators

Here's a table of equality and comparison operators, their meaning, and corresponding functions:

Operator	Meaning	Expression	Translates to
>	greater than	<code>a > b</code>	<code>a.compareTo(b) > 0</code>
<	less than	<code>a < b</code>	<code>a.compareTo(b) < 0</code>
>=	greater than or equals to	<code>a >= b</code>	<code>a.compareTo(b) >= 0</code>
<=	less than or equals to	<code>a <= b</code>	<code>a.compareTo(b) <= 0</code>
==	is equal to	<code>a == b</code>	<code>a?.equals(b) ?: (b === null)</code>
!=	not equal to	<code>a != b</code>	<code>!(a?.equals(b) ?: (b === null))</code>

Comparison and equality operators are used in control flow such as if expression, when expression and loops.

Example: Comparison & Equality Operators

```
fun main (args: Array<String>) {  
    val a = -12  
    val b = 12  
    // use of greater than operator  
    val max = if (a > b) {  
        println ("a is larger than b.")  
        a  
    } else {  
        println ("b is larger than a.")  
        b  
    }  
    println ("max = $max")  
}
```

When you run the code, the output will be:

```
b is larger than a.  
max = 12
```



Logical Operators

There are two logical operators in Kotlin: `||` and `&&`.

Here's a table of logical operators, their meaning, and corresponding functions.

Operator	Description	Expression	Corresponding Function
<code> </code>	<code>true</code> if either of the Boolean expression is <code>true</code>	<code>(a>b) (a<c)</code>	<code>(a>b)or(a<c)</code>
<code>&&</code>	<code>true</code> if all Boolean expressions are <code>true</code>	<code>(a>b)&&(a<c)</code>	<code>(a>b)and(a<c)</code>

Note that, `or` & `and` are functions that support infix notation.

Logical operators are used in control flow such as `if` expression, `when` expression and loops.

Example: Logical Operators

```
fun main (args: Array<String>) {  
  
    val a = 10  
    val b = 9  
    val c = -1  
    val result: Boolean  
  
    // result is true is a is largest  
    result = (a>b) && (a>c) // result = (a>b) and (a>c)  
    println(result)  
}
```

true



in Operator

The in operator is used to check whether an object belongs to a collection.

Operator	Expression	Translates to
in	a in b	b.contains(a)
!in	a !in b	!b.contains(a)

Example: in Operator

```
fun main (args: Array<String>) {  
  
    val numbers = intArrayOf (1, 4, 42, -3)  
  
    if (4 in numbers) {  
        println ("numbers array contains 4.")  
    }  
}
```

When you run the program, the output will be,

```
numbers array contains 4.
```



Index access Operator

Here are some expressions using index access operator with corresponding functions in Kotlin.

Expression	Translated to
<code>a[i]</code>	<code>a.get(i)</code>
<code>a[i, n]</code>	<code>a.get(i, n)</code>
<code>a[i1, i2, ..., in]</code>	<code>a.get(i1, i2, ..., in)</code>
<code>a[i] = b</code>	<code>a.set(i, b)</code>
<code>a[i, n] = b</code>	<code>a.set(i, n, b)</code>
<code>a[i1, i2, ..., in] = b</code>	<code>a.set(i1, i2, ..., in, b)</code>

Example: Index access Operator

```
fun main (args: Array<String>) {  
  
    val a = intArrayOf (1, 2, 3, 4, - 1)  
    println(a[1])  
    a[1]= 12  
    println(a[1])  
}
```

```
2  
12
```



Invoke Operator

Here are some expressions using invoke operator with corresponding functions in Kotlin.

Expression	Translated to
<code>a()</code>	<code>a.invoke()</code>
<code>a(i)</code>	<code>a.invoke(i)</code>
<code>a(i1, i2, ..., in)</code>	<code>a.invoke(i1, i2, ..., in)</code>
<code>a[i] = b</code>	<code>a.set(i, b)</code>

In Kotlin, parenthesis are translated to call `invoke` member function.

Bitwise Operation

Unlike Java, there are no bitwise and bitshift operators in Kotlin. To perform these task, various functions (supporting infix notation) are used:

- `shl` - Signed shift left
- `shr` - Signed shift right
- `ushr` - Unsigned shift right
- `and` - Bitwise and
- `or` - Bitwise or
- `xor` - Bitwise xor
- `inv` - Bitwise inversion



Kotlin Type Conversion

In Kotlin, a numeric value of one type is not automatically converted to another type even when the other type is larger. This is different from Java handles numeric conversions.

For Example: In Java,

```
int number1 = 55;

long number2 = number1; // Valid code
```

Here, value of `number1` of type `int` is automatically converted to type `long`, and assigned to variable `number2`.

In Kotlin,

```
val number1: Int = 55

val number2: Long = number1 // Error: type mismatch.
```

Though the size of `Long` is larger than `Int`, Kotlin doesn't automatically convert `int` to `Long`.

Instead, you need to use `toLong ()` explicitly (to convert to type `Long`). Kotlin does it for type safety to avoid surprises.

```
val number1: Int = 55

val number2: Long = number1.toLong()
```

Here's a list of functions in Kotlin used for type conversion:

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

Note, there is no conversion for `Boolean` types.



Conversion from Larger to Smaller Type

The functions mentioned above can be used in both directions (conversion from larger to smaller type and conversion from smaller to larger type).

However, conversion from larger to smaller type may truncate the value.

For Example,

```
fun main (args : Array<String>) {  
    val number1: Int = 545344  
    val number2: Byte = number1.toByte()  
    println ("number1 = $number1")  
    println ("number2 = $number2")  
}
```

```
number1 = 545344  
number2 = 64
```



Kotlin Expressions, Statements and Blocks

Kotlin Expressions

Expressions consists of **variables**, **operators** etc that evaluates to a **single value**.

Example:

```
val score: Int  
  
score = 90 + 25
```

Here, **90 + 25** is an expression that returns **int** value.

In Kotlin, **if** is an expression unlike Java (in Java, **if** is a statement). For example:

```
fun main (args: Array<String>) {  
  
    val a = 12  
    val b = 13  
    val max: Int  
  
    max = if (a > b) a else b  
    println("$max")  
}
```

Here, **if (a > b) a else b** is an expression. Then value of the expression is assigned to **max** variable in the above program.

Kotlin Statements

Statements are everything that make up a complete unit of execution.

Example:

```
val score = 90 + 25
```

Here, **90 + 25** is an **expression** that returns 115, and **val score = 90 + 25** is a **statement**.

Expressions are part of **statements**.



Kotlin Blocks

A block is a group of statements (zero or more) that is enclosed in curly braces {}.

For example:

```
fun main (args: Array<String>) { // main function block
    val flag = true

    if (flag == true) { // start of if block
        print ("Hey ")
        print("jude!")
    } // end of if block
} // end of main function block
```

There are two statements `print ("Hey ")` and `print("jude!")` inside `if` branch block.

```
print ("Hey ")
print("jude!")
```

similarly, the `main` function also has a block body.

```
val flag = true

if (flag == true) { // start of block

    print ("Hey ")

    print("jude!")

} // end of block
```



Kotlin Comments

In programming, comments are portion of the program intended for you and your fellow programmers to understand the code. They are completely ignored by the Kotlin compiler (**Kompiler**).

Similar like Java, there are two types of comments in Kotlin,

- `/* */`
- `//`

Traditional Comment `/* ... */`

This is a multiline comment that can span over multiple lines. The Kotlin compiler ignores everything from `/*` to `*/`. For Example:

```
/* This is a multi-line comment.  
 * The problem prints "Hello, World!" to the standard output.  
 */  
fun main (args: Array<String>) {  
  
    println ("Hello, World!")  
}
```

The tradition comment is also used for **documenting Kotlin code (KDoc)** with a little variation. The KDoc comments starts with `/**` and ends with `*/`.

End of Line Comment `//`

The compiler ignores everything from `//` to the end of the line. For example,

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

The program above contains two end of line comments:

```
// Kotlin Hello World Program and //outputs Hello, World! on the screen
```



Kotlin Basic Input/Output

Kotlin Output

You can use `println ()` and `print ()` functions to send output to the standard output (screen).

Example:

```
fun main (args : Array<String>) {  
    println ("Kotlin is interesting.")  
}
```

Kotlin is interesting.

Here, `println ()` outputs the string (inside quotes).

Difference between `println ()` and `print ()`

- `print ()` - prints string inside the quotes.
- `Println ()` – prints string inside the quotes similar like `print ()` function. Then the cursor moves to the beginning of the next line.

When you use `println ()` or `print ()` function, it calls `System.out.println ()` function internally.

Example 1: `println ()` and `print ()`

```
fun main (args : Array<String>) {  
    println ("1. println ");  
    println ("2. println ");  
  
    print ("1. print ");  
    print ("2. print");  
}
```

1. println
2. println
1. print 2. print



Example 2: Print Variables and Literals

```
fun main (args : Array<String>) {  
    val score = 12.3  
  
    println("score")  
    println("$score")  
    println ("score = $score")  
    println ("${score + score}")  
    println (12.3)  
}
```

```
score  
12.3  
score = 12.3  
24.6  
12.3
```

Kotlin Input

To read a line of string in Kotlin, you can use `readLine ()` function.

Example 3: Print String Entered by the User

```
fun main (args: Array<String>) {  
    print ("Enter text: ")  
  
    val stringInput = readLine ()!!  
    println ("You entered: $stringInput")  
}
```

```
Enter text: Hmm, interesting!  
You entered: Hmm, interesting!
```

It's possible to take input as a string using `readLine ()` function, and convert it to values of other data type (like int) explicitly .

If you want input of other data types, you can use `Scanner` object.



For that, you need to import Scanner class from Java standard library using:

```
import java.util.Scanner
```

Then you need to create **Scanner object** from this class.

```
val reader = Scanner (System.`in`)
```

Now, the **reader object** is used to take input from the user.

Example 4: Getting Integer Input from the User

```
import java.util.Scanner

fun main (args: Array<String>) {

    // Creates an instance which takes input from standard input (keyboard)
    val reader = Scanner (System.`in`)
    print ("Enter a number: ")

    // nextInt () reads the next integer from the keyboard
    var integer:Int = reader.nextInt()

    println ("You entered: $integer")
}
```

Enter a number: -12

You entered: -12

Here, **reader** object of **Scanner** class is created. Then, the **nextInt ()** method is called which takes integer input from the user which is stored in variable integer.

To get **Long**, **Float**, **Double** and **Boolean** input from the user, you can use **nextLong ()**, **nextFloat ()**, **nextDouble ()** and **nextBoolean ()** methods respectively.



Kotlin Flow Control

Kotlin if expression

Traditional usage of if...else:

The syntax of if...else is:

```
if (testExpression) {  
    // codes to run if testExpression is true  
}  
  
else {  
    // codes to run if testExpression is false  
}
```

If executes a certain section of code if the `testExpression` is evaluated to `true`. It can have optional else clause. Codes inside `else` clause are executed if the testExpression is `false`.

Example: Traditional usage of if...else

```
fun main (args: Array<String>) {  
    val number = -10  
  
    if (number > 0) {  
        print ("Positive number")  
    } else {  
        print ("Negative number")  
    }  
}
```

Negative number



Unlike Java (and other many programming language), **if** can be used as an expression in Kotlin; it returns a value.

Example: Kotlin if expression

```
fun main (args: Array<String>) {  
  
    val number = -10  
  
    val result = if (number > 0) {  
        "Positive number"  
    } else {  
        "Negative number"  
    }  
  
    println(result)  
}
```

Negative number

The **else** branch is mandatory when using **if** as an expression.

The **curly braces** are optional if the body of if has only one statement. For example:

```
fun main (args: Array<String>) {  
    val number = -10  
    val result = if (number > 0) "Positive number" else "Negative number"  
    println(result)  
}
```

This is similar to **ternary operator** in Java. Hence, there is **no** ternary operator in Kotlin.



Example: if block with Multiple Expressions

If the block of `if` branch contains more than one expression, the last expression is returned as the value of the block.

```
fun main (args: Array<String>) {  
  
    val a = -9  
    val b = -11  
  
    val max = if (a > b) {  
        println ("$a is larger than $b.")  
        println ("max variable holds value of a.")  
        a  
    } else {  
        println ("$b is larger than $a.")  
        println ("max variable holds value of b.")  
        b  
    }  
    println ("max = $max")  
}
```

```
-9 is larger than -11.  
max variable holds value of a.  
max = -9
```

Kotlin if...else...if Ladder

You can return a block of code among many blocks in Kotlin using `if...else...if` ladder.

Example: if...else...if Ladder

```
fun main (args: Array<String>) {  
  
    val number = 0  
  
    val result = if (number > 0)  
        "positive number"  
    else if (number < 0)  
        "negative number"  
    else
```




```
"zero"

println ("number is $result")
}
```

Number is zero

This program checks whether `number` is positive number, negative number or zero.

Kotlin Nested if Expression

An `if` expression can be inside the block of another `if` expression known as nested `if` expression.

Example: Nested if Expression

This program computes the largest number among three numbers.

```
fun main (args: Array<String>) {
    val n1 = 3
    val n2 = 5
    val n3 = -2

    val max = if (n1 > n2) {
        if (n1 > n3)
            n1
        else
            n3
    } else {
        if (n2 > n3)
            n2
        else
            n3
    }
    println ("max = $max")
}
```

max = 5



Kotlin When Expression

Kotlin when Construct

The **when construct** in Kotlin can be thought of as a replacement of **Java switch statement**. It evaluates a section of code among many alternatives.

Example: Simple when Expression

```
fun main (args: Array<String>) {  
  
    val a = 12  
    val b = 5  
  
    println ("Enter operator either +, -, * or /")  
    val operator = readLine ()  
  
    val result = when (operator) {  
        "+" -> a + b  
        "-" -> a - b  
        "*" -> a * b  
        "/" -> a / b  
        else -> "$operator operator is invalid operator."  
    }  
  
    println ("result = $result")  
}
```

```
Enter operator either +, -, * or /  
*  
result = 60
```

The program above takes an input **string** from the user. Suppose, the user entered *****. In this case, the expression **a * b** is evaluated, and the value is assigned to variable **result**.

If none of the branch conditions are satisfied (user entered anything except +, -, * or /) **else** branch is evaluated.



In the above example, we used `when` as an expression. However, it's not mandatory to use `when` as an expression.

For example:

```
fun main (args: Array<String>) {  
  
    val a = 12  
    val b = 5  
  
    println ("Enter operator either +, -, * or /")  
    val operator = readLine ()  
  
    when (operator) {  
        "+" -> println (" $a + $b = ${a + b}")  
        "-" -> println (" $a - $b = ${a - b}")  
        "*" -> println (" $a * $b = ${a * b}")  
        "/" -> println (" $a / $b = ${a / b}")  
        else -> println (" $operator is invalid")  
    }  
}
```

```
Enter operator either +, -, * or /  
-  
12 - 5 = 7
```

Here, `when` is not an expression (return value from `when` is not assigned to anything). In this case, the `else` branch is not mandatory.

NOTE:

The `->` is a **separator**. It is a special symbol used to separate code with different purpose.



Few Possibilities

Example: Combine two or more branch conditions with a comma.

```
fun main (args: Array<String>) {  
  
    val n = -1  
  
    when (n) {  
        1, 2, 3 -> println ("n is a positive integer less than 4.")  
        0 -> println ("n is zero")  
        -1, -2 -> println ("n is a negative integer greater than 3.")  
    }  
}
```

n is a negative integer greater than 3.

Example: Check value in the range

```
fun main (args: Array<String>) {  
  
    val a = 100  
  
    when (a) {  
        in 1..10 -> println("A positive number less than 11.")  
        in 10..100 -> println("A positive number between 10 and 100 (inclusive)")  
    }  
}
```

A positive number between 10 and 100 (inclusive)



Check if a value is of a particular type

To check whether a value is of a particular type in runtime, we can use `is` and `!is` operator.

Example:

```
when (x) {  
    is Int -> print (x + 1)  
    is String -> print (x.length + 1)  
    is IntArray -> print(x.sum())  
}
```

Use expressions as branch condition

Example:

```
fun main (args: Array<String>) {  
  
    val a = 11  
    val n = "11"  
  
    when (n) {  
        "cat" -> println ("Cat? Really?")  
        12.toString () -> println ("Close but not close enough.")  
        a.toString() -> println("Bingo! It's eleven.")  
    }  
}
```

Bingo! It's eleven.



Kotlin while and do-while Loop

Loop is used in programming to repeat a specific block of code until certain condition is met (text expression is false).

Loops are what makes computers interesting machines. Imagine you need to print a sentence 50 times on your screen. Well, you can do it by using print statement 50 times (without using loops). How about you need to print a sentence one million times? You need to use loops.

Kotlin while Loop

The syntax of while loop is:

```
while (testExpression) {  
  
    // codes inside body of while loop  
  
}
```

How while loop works?

The test expression inside the parenthesis is a Boolean expression.

If the test expression is evaluated to true,

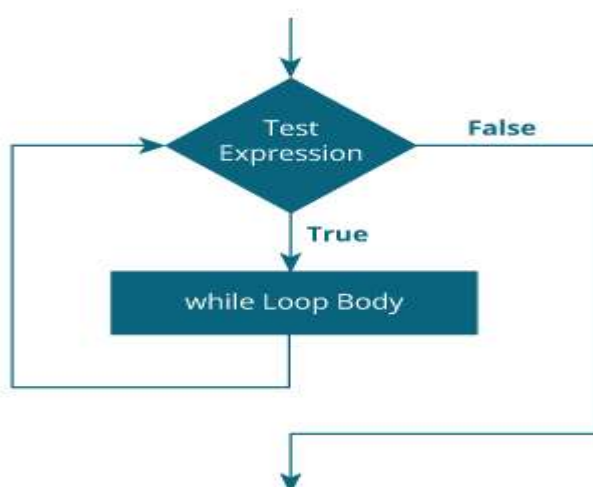
- Statements inside the while loop are executed.
- Then, the test expression is evaluated again.

This process goes on until the test expression is evaluated to false.

If the test expression is evaluated to false,

- While loop is terminated

Flowchart of while loop





Example: Kotlin while Loop

```
// Program to print line 5 times

fun main (args: Array<String>) {

    var i = 1

    while (i <= 5) {
        println ("Line $i")
        ++i
    }
}
```

Line 1
Line 2
Line 3
Line 4
Line 5

Notice, `++i` statement inside the `while` loop. After 5 iterations, variable `i` will be incremented to 6. Then, the test expression `i <= 5` is evaluated to false and the loop terminates.

If the body of loop has only one statement, it's not necessary to use `curly braces {}`.

Example: Compute sum of Natural Numbers

```
// Program to compute the sum of natural numbers from 1 to 100.

fun main (args: Array<String>) {

    var sum = 0
    var i = 100

    while (i != 0) {
        sum += i    // sum = sum + i;
        --i
    }
    println ("sum = $sum")
}
```

When you run the program, the output will be:



```
sum = 5050
```

Here, the variable `sum` is initialized to '0' and `i` is initialized to 100. In each iteration of while loop, variable `sum` is assigned `sum + i`, and the value of `i` is decreased by 1 until `i` is equal to 0.

For better visualization,

```
1st iteration: sum = 0+100 = 100, i = 99
```

```
2nd iteration: sum = 100+99 = 199, i = 98
```

```
3rd iteration: sum = 199+98 = 297, i = 97
```

```
... ..
```

```
... ..
```

```
99th iteration: sum = 5047+2 = 5049, i = 1
```

```
100th iteration: sum = 5049+1 = 5050, i = 0 (then loop terminates)
```




Kotlin do-while Loop

The **do-while** loop is similar to while loop with **one** key difference. The body of do-while loop is executed **once** before the test expression is checked.

Syntax:

```
do {  
    // codes inside body of do while loop  
} while (testExpression);
```

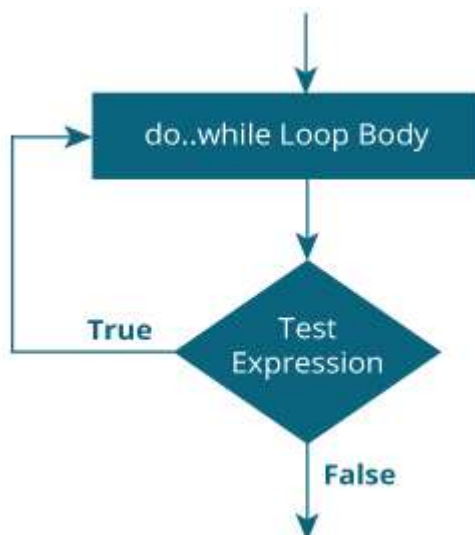
How do-while loop works?

The codes inside the body of **do** construct is executed once (without checking the **testExpression**). Then the **testExpression** is checked.

If the **testExpression** is evaluated to **true**, codes inside the body of the loop are executed, and **testExpression** is evaluated again. This process goes on until the **testExpression** is evaluated to **false**.

When the **testExpression** is evaluated to **false**, **do-while** loop terminates.

Flowchart of do-while Loop





Example: Kotlin do-while Loop

The program below calculates the sum of numbers entered by the user until user enters 0.

To take input from the user, `readLine ()` function is used.

```
fun main (args: Array<String>) {  
  
    var sum: Int = 0  
    var input: String  
  
    do {  
        print ("Enter an integer: ")  
        input = readLine ()!!  
        sum += input.toInt()  
  
    } while (input != "0")  
  
    println ("sum = $sum")  
}
```

```
Enter an integer: 4  
Enter an integer: 3  
Enter an integer: 2  
Enter an integer: -6  
Enter an integer: 0  
sum = 3
```



Kotlin for Loop

The for loop in Kotlin iterates through anything that provides an iterator.

There is no **traditional for loop** in Kotlin unlike Java and other languages.

In Kotlin, **for** loop is used to iterate through **ranges**, **arrays**, **maps** and so on (anything that provides an iterator).

The syntax of for loop in Kotlin is:

```
for (item in collection) {  
  
    // body of loop  
  
}
```

Example: Iterate Through a Range

```
fun main (args: Array<String>) {  
  
    for (i in 1..5) {  
        println(i)  
    }  
}
```

Here, the loop iterates through the range and prints individual item.

```
1  
2  
3  
4  
5
```

If the body of the loop contains only one statement (like above example), it's not necessary to use curly braces {}.

```
fun main (args: Array<String>) {  
    for (i in 1..5) println(i)  
}
```

It's possible to iterate through a range using **for** loop because ranges provide an iterator.



Example: Different ways to Iterate through a Range

```
fun main (args: Array<String>) {

    print ("for (i in 1..5) print(i) = ")
    for (i in 1..5) print(i)

    println ()

    print ("for (i in 5..1) print(i) = ")
    for (i in 5..1) print(i)      // prints nothing

    println ()

    print ("for (i in 5 downTo 1) print(i) = ")
    for (i in 5 downTo 1) print(i)

    println ()

    print ("for (i in 1..5 step 2) print(i) = ")
    for (i in 1..5 step 2) print(i)

    println ()

    print ("for (i in 5 downTo 1 step 2) print(i) = ")
    for (i in 5 downTo 1 step 2) print(i)
}
```

```
for (i in 1..5) print(i) = 12345
for (i in 5..1) print(i) =
for (i in 5 downTo 1) print(i) = 54321
for (i in 1..4 step 2) print(i) = 135
for (i in 4 downTo 1 step 2) print(i) = 531
```



Example: Iterate Through an Array

Here's an example to iterate through a `String` array.

```
fun main (args: Array<String>) {  
  
    var language = arrayOf ("Ruby", "Kotlin", "Python" "Java")  
  
    for (item in language)  
        println(item)  
}
```

Ruby
Koltin
Python
Java

It is possible to iterate through an array with an index. For example:

```
fun main (args: Array<String>) {  
  
    var language = arrayOf ("Ruby", "Koltin", "Python", "Java")  
  
    for (item in language.indices) {  
  
        // printing array elements having even index only  
        if (item%2 == 0)  
            println(language[item])  
    }  
}
```

Ruby
Python



Example: Iterating through a String

```
fun main (args: Array<String>) {  
  
    var text= "Kotlin"  
  
    for (letter in text) {  
        println(letter)  
    }  
}
```

K
o
t
l
i
n

similar like arrays, you can iterate through a String with an index. For example:

```
fun main (args: Array<String>) {  
  
    var text= "Kotlin"  
  
    for (item in text.indices) {  
        println(text[item])  
    }  
}
```

K
o
t
l
i
n



Kotlin break Expression

Suppose you are working with loops. It is sometimes desirable to terminate the loop immediately without checking the testExpression.

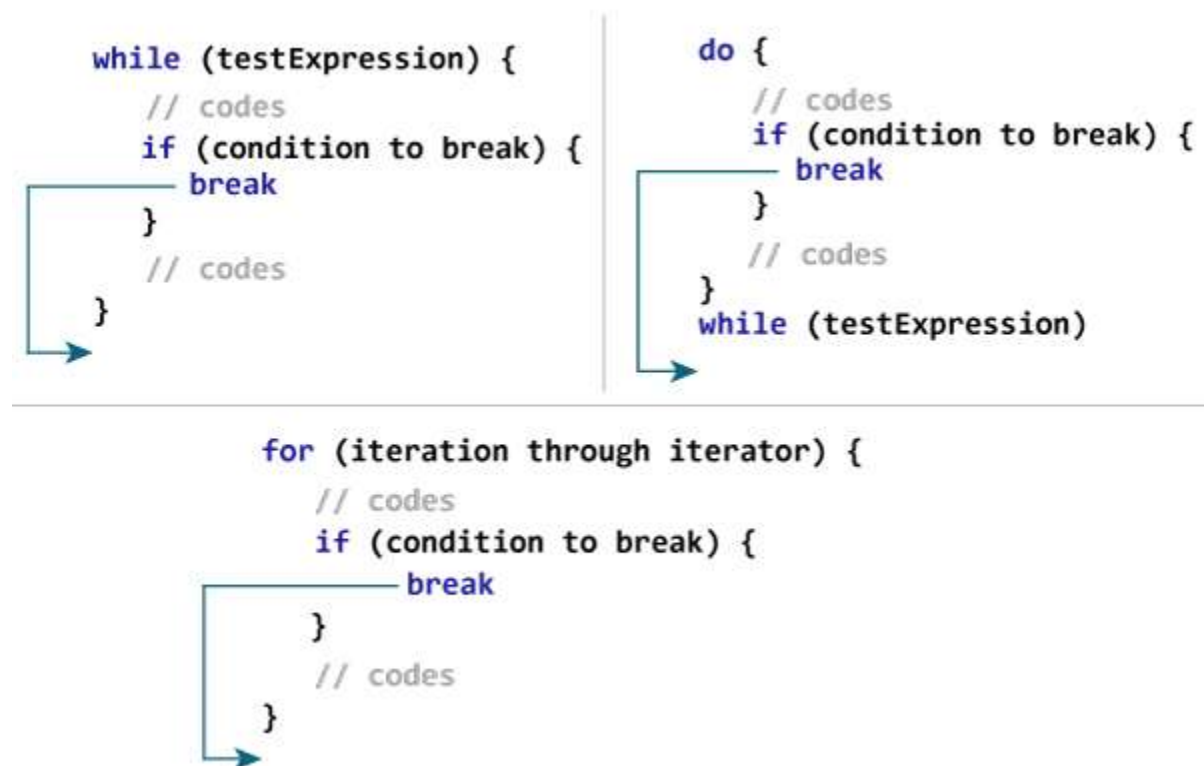
In such case, `break` is used. It terminates the nearest enclosing loop when encountered (without checking the testExpression). This is similar to how break statement works in Java.

How break works?

It is almost always used with `if-else` construct. For example:

```
for (...) {  
    if (testExpression) {  
        break  
    }  
}
```

If `testExpression` is evaluated to `true`, `break` is executed which terminates the `for` loop.





Example: Kotlin break

```
fun main (args: Array<String>) {  
  
    for (i in 1..10) {  
        if (i == 5) {  
            break  
        }  
        println(i)  
    }  
}
```

```
1  
2  
3  
4
```

When the value of `i` is equal to 5, expression `i == 5` inside `if` is evaluated to `true`, and `break` is executed. This terminates the `for` loop.

Example: Calculate Sum until user enters 0

```
fun main (args: Array<String>) {  
    var sum = 0  
    var number: Int  
  
    while (true) {  
        print ("Enter a number: ")  
        number = readLine ()!!.toInt()  
  
        if (number == 0)  
            break  
  
        sum += number  
    }  
  
    print ("sum = $sum")  
}
```

When you run the program, the output will be:



```
Enter a number: 4
Enter a number: 12
Enter a number: 6
Enter a number: -9
Enter a number: 0
sum = 13
```

The program above calculates the sum of numbers by the user until user enters 0.

In this program, the testExpression of the **while** loop is always **true**.

Here, the **while** loop runs until user enters 0. When user inputs 0, **break** is executed which terminates the **while** loop.

Kotlin Labeled break

What you have done till now is unlabeled form of **break**, which terminates the nearest enclosing loop. There is another way **break** can be used (labeled form) to terminate the desired loop (can be outer loop).

How labeled break works?

```
test@ while (testExpression) {
    // codes
    while (testExpression) {
        // codes
        if (condition to break) {
            break@test
        }
        // codes
    }
    // codes
}
```

Label in Kotlin starts with an **identifier** which is followed by **@**.

Here, **test@** is a label marked at the outer **while** loop. Now, by using **break** with a label (**break@test** in this case), you can break the specific loop.



Example:

```
fun main (args: Array<String>) {  
  
    first@ for (i in 1..4) {  
  
        second@ for (j in 1..2) {  
            println ("i = $i; j = $j")  
  
            if (i == 2)  
                break@first  
        }  
    }  
}
```

```
i = 1; j = 1  
i = 1; j = 2  
i = 2; j = 1
```

Here, when `i == 2` expression is evaluated to true, `break@first` is executed which terminates the loop marked with label `first@`.

Here's a little variation of the above program.

In the program below, `break` terminates the loop marked with label `second@`.

```
fun main (args: Array<String>) {  
  
    first@ for (i in 1..4) {  
  
        second@ for (j in 1..2) {  
            println ("i = $i; j = $j")  
  
            if (i == 2)  
                break@second  
        }  
    }  
}
```

When you run the program, the output will be:

Ashirbad Swain



```
i = 1; j = 1  
i = 1; j = 2  
i = 2; j = 1  
i = 3; j = 1  
i = 3; j = 2  
i = 4; j = 1  
i = 4; j = 2
```

NOTE: Since, break is used to terminate the innermost loop in this program, it is not necessary to use labeled break in this case.

There are 3 structural jump expressions in Kotlin: break, continue and return.

- Break
- Continue
- Return



Kotlin continue Expression

Suppose you are working with loops. It is sometimes desirable to skip the **current iteration** of the loop.

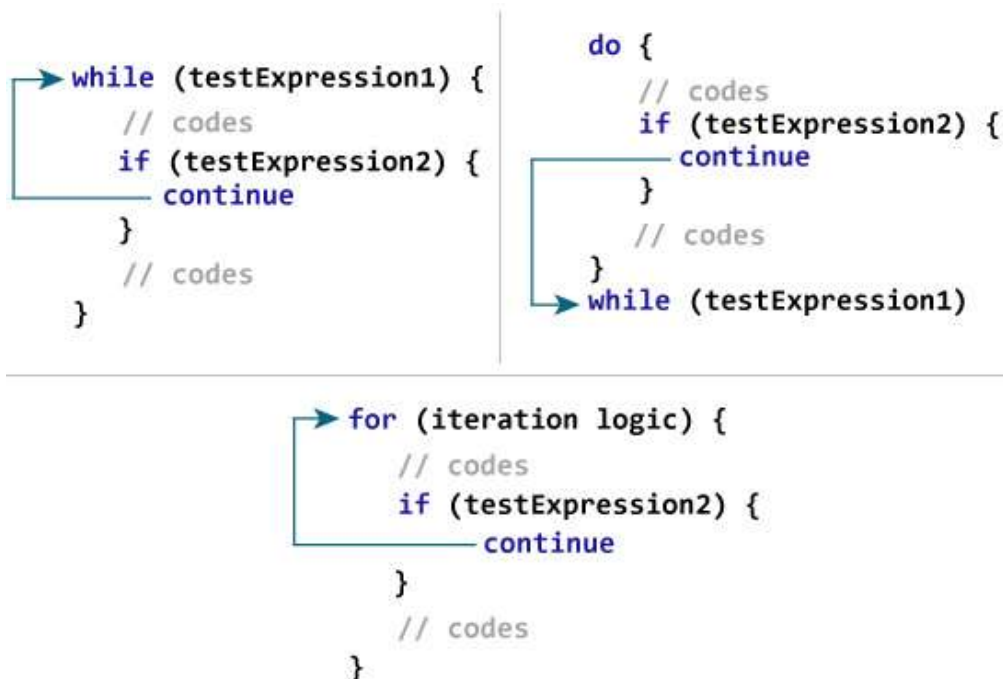
In such case, **continue** is used. The continue construct skips the current iteration of the enclosing loop, and the control of the program jumps to the end of the loop body.

How continue works?

It is almost always used with **if-else** construct. For example:

```
while (testExpression1) {  
  
    // codes1  
  
    if (testExpression2) {  
  
        continue  
  
    }  
  
    // codes2  
  
}
```

If the **testExpression2** is evaluated to **true**, **continue** is executed which skips all the codes inside **while** loop after it for that iteration.





Example: Kotlin Continue

```
fun main (args: Array<String>) {  
  
    for (i in 1..5) {  
        println ("$i Always printed.")  
        if (i > 1 && i < 5) {  
            continue  
        }  
        println ("$i Not always printed.")  
    }  
}
```

```
1 Always printed.  
1 Not always printed.  
2 Always printed.  
3 Always printed.  
4 Always printed.  
5 Always printed.  
5 Not always printed.
```

When the value of `i` is greater than 1 and less than 5, `continue` is executed, which skips the execution of below statement.

```
println ("$i Not always printed.")
```

However, the statement

```
println ("$i Always printed.")
```

is executed in each iteration of the loop because this statement exists before the `continue` construct.



Example: Calculate Sum of Positive Numbers Only

The program below calculates the sum of maximum of 6 positive numbers entered by the user. If the user enters negative number or zero, it is skipped from calculation.

```
fun main (args: Array<String>) {  
  
    var number: Int  
    var sum = 0  
  
    for (i in 1..6) {  
        print ("Enter an integer: ")  
        number = readLine ()!!.toInt()  
  
        if (number <= 0)  
            continue  
  
        sum += number  
    }  
    println ("sum = $sum")  
}
```

```
Enter an integer: 4  
Enter an integer: 5  
Enter an integer: -50  
Enter an integer: 10  
Enter an integer: 0  
Enter an integer: 12  
sum = 31
```




Kotlin Labeled continue

What you have learned till now is unlabeled form of `continue`, which skips current iteration of the nearest enclosing loop. Continue can also be used to skip the iteration of the desired loop (can be outer loop) by using continue labels.

How labeled continue works?

```
outerloop@ while (testExpression) {  
    // codes  
    while (testExpression) {  
        // codes  
        if (condition for continue) {  
            continue@outerloop  
        }  
        // codes  
    }  
    // codes  
}
```



Label in Kotlin starts with an `identifier` which is followed by `@`.

Here, `outerloop@` is a label marked at outer while loop. Now, by using `continue` with the label (`continue@outerloop` in this case), you can skip the execution of codes of the specific loop for that iteration.

Example: labeled continue

```
fun main (args: Array<String>) {  
    here@ for (i in 1..5) {  
        for (j in 1..4) {  
            if (i == 3 || j == 2)  
                continue@here  
            println ("i = $i; j = $j")  
        }  
    }  
}
```

```
i = 1; j = 1  
i = 2; j = 1  
i = 4; j = 1  
i = 5; j = 1
```

The use of labeled `continue` is often discouraged as it makes your code hard to understand. If you are in a situation where you have to use labeled continue, refactor your code and try to solve it in a different way to make it more readable.



Kotlin Functions

In programming, **function** is a group of related statements that perform a specific task.

Functions are used to break a large program into smaller and modular chunks.

For example, you need to create and color a circle based on input from the user. You can create two functions to solve this problem:

- `createCircle ()` function
- `colorCircle ()` function

Dividing a complex program into smaller components makes our program more organized and manageable.

Furthermore, it avoids **repetition** and makes code **reusable**.

Type of Functions

Depending on whether a function is defined by the user, or available in standard library, there are two types of functions:

- Kotlin Standard Library Function
- User-defined Functions

Kotlin Standard Library Function

The standard library functions are built-in functions in Kotlin that are readily available for use. For example,

- `print ()` is a library function that prints message to the standard output stream (monitor).
- `sqrt ()` returns the square root of a number (Double value).

Example:

```
fun main (args: Array<String>) {  
  
    var number = 5.5  
    print ("Result = ${Math.sqrt(number)}")  
}
```

Result = 2.345207879911715



User-defined Functions

As mentioned, you can create functions yourself. Such functions are called user-defined functions.

How to create a user-defined function in Kotlin?

Before you can use (call) a function, you need to define it.

Here's how you can define a function in Kotlin:

```
fun callMe () {  
  
    // function body  
  
}
```

To define a function in Kotlin, **fun** keyword is used. Then comes the name of the function (**identifier**). Here, the name of the function is **callMe**.

In the above program, the **parenthesis ()** is empty. It means, this function doesn't accept any argument.

The codes inside curly braces **{ }** is the body of the function.

How to call a function?

You have to call the function to run codes inside the body of the function. Here's how:

```
callMe ()
```

This statement calls the `callMe ()` function declared earlier.

```
fun callMe() { ←  
    ... ..  
    ... ..  
}  
  
fun main(args: Array<String>) {  
    ... ..  
    callMe() —————  
    ... ..  
}
```



Example: Simple Function Program

```
fun callMe () {  
    println ("Printing from callMe () function.")  
    println ("This is cool (still printing from inside).")  
}  
  
fun main (args: Array<String>) {  
    callMe ()  
    println ("Printing outside from callMe () function.")  
}
```

```
Printing from callMe () function.  
This is cool (still printing from inside).  
Printing outside from callMe () function.
```

The `callMe ()` function in the above code doesn't accept any arguments.

Also, the function doesn't return any value (return type is `unit`).

Example: Add two numbers using function

```
fun addNumbers (n1: Double, n2: Double): Int {  
    val sum = n1 + n2  
    val sumInteger = sum.toInt()  
    return sumInteger  
}  
  
fun main (args: Array<String>) {  
    val number1 = 12.2  
    val number2 = 3.4  
    val result: Int  
  
    result = addNumbers (number1, number2)  
    println ("result = $result")  
}
```

```
result = 15
```



How functions with arguments and return value work?

Here, two arguments `number1` and `number2` of type `Double` are passed to the `addNumbers ()` function during function call. These arguments are called **actual arguments**.

```
result = addNumbers (number1, number2)
```

The parameters `n1` and `n2` accepts the passed arguments (in the function definition). These arguments are called **formal arguments** (or parameters).

```
fun addNumbers(n1: Double, n2: Double): Int {  
    ... ..  
    ... ..  
}  
  
fun main(args: Array<String>) {  
    ... ..  
    result = addNumbers(number1, number2)  
    ... ..  
}
```

In Kotlin, arguments are separated using **commas**. Also, the type of the formal argument must be explicitly typed.

Note that, the data type of actual and formal arguments should match, i.e. the data type of first actual argument should match the type of first formal argument. Similarly, the type of second actual argument must match the type of second formal argument and so on.

Here,

```
return sumInteger
```

is the return statement. This code terminates the `addNumbers()` function, and control of the program jumps to the `main()` function.

In the program, `sumInteger` is returned from `addNumbers()` function. This value is assigned to the variable `result`.



```
fun addNumbers(n1: Double, n2: Double): Int {  
    ... ..  
    return sumInteger  
}  
  
fun main(args: Array<String>) {  
    ... ..  
    result = addNumbers(number1, number2)  
    ... ..  
}
```

Notice,

- both `sumInteger` and `result` are of type `Int`.
- the return type of the function is specified in the function definition.

```
// return type is Int
```

```
fun addNumbers (n1: Double, n2: Double): Int {  
  
    ... ..  
}
```

If the function doesn't return any value, its return type is `Unit`. It is optional to satisfy the return type in the function definition of the return type is `Unit`.



Example: Display Name by using Function

```
fun main (args: Array<String>) {  
    println (getName ("John", "Doe"))  
}  
  
fun getName(firstName: String, lastName: String): String = "$firstName $lastName"
```

John Doe

Here, the function `getName()` takes two `String` arguments, and returns a `String`.

You can omit the curly braces `{ }` of the function body and specify the body after `=` symbol if the function returns a single expression (like above example).

It is optional to explicitly declare the return type in such case because the return type can be inferred by the compiler.

In the above example, you can replace:

```
fun getName(firstName: String, lastName: String): String = "$firstName $lastName"
```

with

```
fun getName(firstName: String, lastName: String) = "$firstName $lastName"
```



Kotlin Infix Function Call

When you use `||` and `&&` operations, the compiler looks up for `or` & `and` functions respectively, and calls them under the hood.

These two functions support infix notation.

```
fun main (args: Array<String>) {  
    val a = true  
    val b = false  
    var result: Boolean  
  
    result = a or b // a.or(b)  
    println ("result = $result")  
  
    result = a and b // a.and(b)  
    println ("result = $result")  
}
```

When you run the program, the output will be:

```
result = true  
result = false
```

In the above program, `a or b` instead of `a.or(b)`, and `a and b` instead of `a.and(b)` is used. It was possible because these two functions support infix notation.

How to create a function with infix notation?

You can make a function call in Kotlin using infix notation if the function,

- is a member function (or an extension function)
- has only one single parameter
- is marked with `infix` keyword



Example: User-defined Function with Infix Notation

```
class Structure () {  
  
    infix fun createPyramid (rows: Int) {  
        var k = 0  
        for (i in 1..rows) {  
            k = 0  
            for (space in 1..rows-i) {  
                print (" ")  
            }  
            while (k != 2*i-1) {  
                print ("* ")  
                ++k  
            }  
            println ()  
        }  
    }  
}  
  
fun main (args: Array<String>) {  
    val p = Structure ()  
    p createPyramid 4    // p.createPyramid(4)  
}
```

```
*  
* * *  
* * * * *  
* * * * * *
```

Here, `createPyramid()` is an infix function that creates a pyramid structure. It is a member function of class `Structure`, takes only one parameter of type `Int`, and starts with keyword `infix`.

The number of rows of the pyramid depends on the argument passed to the function.



Kotlin Default and Named Arguments

Kotlin Default Argument

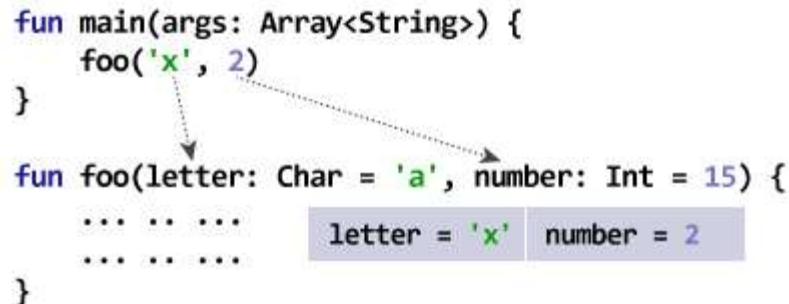
In Kotlin, you can provide default values to parameters in function definition.

If the function is called with arguments passed, those arguments are used as parameters. However, if the function is called without passing arguments, default arguments are used.

How default arguments work?

Case 1: All Arguments passed

```
fun main(args: Array<String>) {  
    foo('x', 2)  
}  
  
fun foo(letter: Char = 'a', number: Int = 15) {  
    ... ..  
    ... ..  
}
```

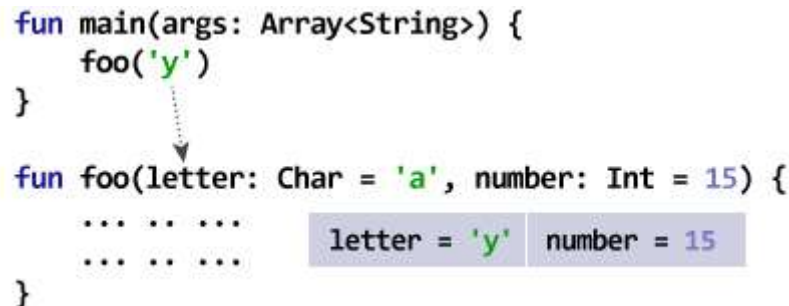


The function `foo()` takes two arguments. The arguments are provided with default values. However, `foo()` is called by passing both arguments in the above program. Hence, the default arguments are not used.

The value of `letter` and `number` will be `x` and `2` respectively inside the `foo()` function.

Case 2: All arguments are not passed

```
fun main(args: Array<String>) {  
    foo('y')  
}  
  
fun foo(letter: Char = 'a', number: Int = 15) {  
    ... ..  
    ... ..  
}
```



Here, only one (first) argument is passed to the `foo()` function. Hence, the first argument uses the value passed to the function. However, second argument `number` will take the default value since the second argument is not passed during function call.

The value of `letter` and `number` will be `y` and `15` respectively inside the `foo()` function.



Case 3: No argument is passed

```
fun main(args: Array<String>) {  
    foo()  
}  
  
fun foo(letter: Char = 'a', number: Int = 15) {  
    ... ..  
    ... ..  
    letter = 'a' number = 15  
}
```

Here, the `foo()` function is called without passing any argument. Hence, both arguments use its default values.

The value of `letter` and `number` will be `a` and `15` respectively inside the `foo()` function.

Example: Kotlin Default Argument

```
fun displayBorder (character: Char = '=', length: Int = 15) {  
    for (i in 1..length) {  
        print(character)  
    }  
}  
  
fun main (args: Array<String>) {  
    println("Output when no argument is passed:")  
    displayBorder()  
  
    println("\n\n'*' is used as a first argument.")  
    println("Output when first argument is passed:")  
    displayBorder('*')  
  
    println("\n\n'*' is used as a first argument.")  
    println("5 is used as a second argument.")  
    println("Output when both arguments are passed:")  
    displayBorder('*', 5)  
}
```



Output when no argument is passed:

=====

'*' is used as a first argument.

Output when first argument is passed:

'*' is used as a first argument.

5 is used as a second argument.

Output when both arguments are passed:

Kotlin named argument

Before talking about named argument, let us consider a little modification of the above code:

```
fun displayBorder(character: Char = '=', length: Int = 15) {  
    for (i in 1..length) {  
        print(character)  
    }  
}  
  
fun main(args: Array<String>) {  
    displayBorder(5)  
}
```

Here, we are trying to pass second argument to the `displayBorder()` function, and use default argument for first argument. However, this code will give use an error. It's because the compiler thinks we are trying to provide 5 (`Int` value) to character (`Char` type).

To solve this situation, named arguments can be used. Here's how



Example: Kotlin named argument

```
fun displayBorder(character: Char = '=', length: Int = 15) {  
    for (i in 1..length) {  
        print(character)  
    }  
}  
  
fun main(args: Array<String>) {  
    displayBorder(length = 5)  
}
```

```
=====
```

In the above program, we are using named argument (`length = 5`) specifying that `length` parameter in the function definition should take this value (doesn't matter the position of the argument).

```
fun displayBorder(character: Char = '=', length: Int = 15) {  
    for (i in 1..length) {  
        print(character)  
    }  
}  
  
fun main(args: Array<String>) {  
    displayBorder(length = 5)  
}
```

The first argument `character` uses the default value `'=`' in the program.



Kotlin Recursion (Recursion Function) and Tail Recursion

A **function** that calls itself is known as **recursive function**. And, this technique is known as **recursion**.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

How does recursion work in programming?

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

Here, the `recurse()` function is called from the body of `recurse()` function itself. Here's how this program works:

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

Normal Function Call

Recursive Call

Here, the recursive call continues forever causing infinite recursion.



To avoid infinite recursion, `if-else` (or similar approach) can be used where one branch makes the recursive call other doesn't.

Example: Find factorial of a Number using Recursion

```
fun main(args: Array<String>) {  
    val number = 4  
    val result: Long  
  
    result = factorial(number)  
    println("Factorial of $number = $result")  
}  
  
fun factorial(n: Int): Long {  
    return if (n == 1) n.toLong() else n*factorial(n-1)  
}
```

Factorial of 4 = 24

How this program works?

The recursive call of the `factorial()` function can be explained in the following figure

```
factorial(4)      // 1st function call. Argument: 4  
  
4*factorial(3)    // 2nd function call. Argument: 3  
  
4*(3*factorial(2)) // 3rd function call. Argument: 2  
  
4*(3*(2*factorial(1))) // 4th function call. Argument: 1  
  
4*(3*(2*1))  
  
24
```



```
fun main(args: Array<String>) {  
    ... ..  
    result = factorial(number) ← 4  
    ... ..  
}  
  
fun factorial(n: Int): Long {  
    return if (n == 1)  
        n.toLong()  
    else  
        n * factorial(n - 1) ← 3  
    } 4  
}  
  
fun factorial(n: Int): Long {  
    return if (n == 1)  
        n.toLong()  
    else  
        n * factorial(n - 1) ← 2  
    } 3  
}  
  
fun factorial(n: Int): Long {  
    return if (n == 1)  
        n.toLong()  
    else  
        n * factorial(n - 1) ← 1  
    } 2  
}  
  
fun factorial(n: Int): Long {  
    return if (n == 1)  
        1 n.toLong()  
    else  
        n * factorial(n - 1)  
    }  
}
```

return 4*3*2*1

return 3*2*1

return 2*1

return 1



Kotlin Tail Recursion

Tail recursion is a generic concept rather than the feature of Kotlin language. Some programming languages including Kotlin use it to optimize recursive calls, whereas other languages do not support them.

What is tail recursion?

In normal recursion, you perform all recursive calls first, and calculate the result from return values at last. Hence, you don't get result until all recursive calls are made.

In **tail recursion**, calculations are performed first, then recursive calls are executed (the recursive call passes the result of your current step to the next recursive call). This makes the recursive call equivalent to looping, and avoids the risk of stack overflow.

Condition for tail recursion

A recursive function is eligible for tail recursion if the function call to itself is the last operation it performs.

Example 1:

Not eligible for tail recursion because the function call to itself $n * \text{factorial}(n - 1)$ is not the last operation.

```
fun factorial(n: Int): Long {  
    if (n == 1) {  
        return n.toLong()  
    } else {  
        return n*factorial(n - 1)  
    }  
}
```

Example 2:

Eligible for tail recursion because function call to itself $n * \text{factorial}(n - 1)$ is the last operation.

```
fun fibonacci(n: Int, a: Long, b: Long): Long {  
    return if (n == 0) b else fibonacci(n-1, a+b, a)  
}
```

To tell the compiler to perform tail recursion in Kotlin, you need to mark the function with **tailrec** modifier.



Example: Tail Recursion

```
import java.math.BigInteger

fun main(args: Array<String>) {
    val n = 100
    val first = BigInteger("0")
    val second = BigInteger("1")

    println(fibonacci(n, first, second))
}

tailrec fun fibonacci(n: Int, a: BigInteger, b: BigInteger): BigInteger {
    return if (n == 0) a else fibonacci(n-1, b, a+b)
}
```

354224848179261915075

This program computes the 100th term of the Fibonacci series. Since, the output can be a very large integer, we have imported `BigInteger` class from Java standard library.

Here, the function `fibonacci()` is marked with `tailrec` modifier and the function is eligible for tail recursive call. Hence, the compiler optimizes the recursion in this case.

If you try to find the 20000th term (or any other big integer) of the Fibonacci series without using tail recursion, the compiler will throw `java.lang.StackOverflowError` exception. It's because we have used tail recursion which uses efficient loop-based version instead of traditional recursion.

Example: Factorial using Tail Recursion

The example to compute factorial of a number in the above example cannot be optimized for tail recursion. Here's a different program to perform the same task.

```
fun main(args: Array<String>) {
    val number = 5
    println("Factorial of $number = ${factorial(number)}")
}

tailrec fun factorial(n: Int, run: Int = 1): Long {
    return if (n == 1) run.toLong() else factorial(n-1, run*n)
}
```




Factorial of 5 = 120

The compiler can optimize the recursion in this program as the recursive function is eligible for tail recursion, and we have used `tailrec` modifier that tells compiler to optimize the recursion.

Kotlin OOP (Object Oriented Programming)

Kotlin supports both `functional` and `object-oriented` programming.

Kotlin supports features such as higher-order functions, functions types and lambdas which makes it great choice for working in functional programming style.

Object-oriented Programming (OOP)

In object-oriented style of programming, you can divide a complex problem into smaller sets by creating objects.

These `objects` share two characteristics:

- state
- behavior

Let's take few examples:

1. `Lamp` is an object
 - It can be in `on` or `off` state.
 - You can `turn on` and `turn off` lamp (behavior).
2. `Bicycle` is an object
 - It has `current gear`, `two wheels`, `number of gears` etc. states.
 - It has `braking`, `accelerating`, `changing gears` etc behavior.

Kotlin Class

Before you create objects in Kotlin, you need to define a class.

A `class` is a blueprint for the object.

We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.

Since, many houses can be made from the same description, we can create many objects from a class.



How to define a class in Kotlin?

To define a class in Kotlin, `class` keyword is used:

```
class ClassName {  
  
    // property  
  
    // member function  
  
    ... ..  
  
}
```

Here's an example:

```
class Lamp {  
  
    // property (data member)  
  
    private var isOn: Boolean = false  
  
  
    // member function  
  
    fun turnOn() {  
  
        isOn = true  
  
    }  
  
    // member function  
  
    fun turnOff() {  
  
        isOn = false  
  
    }  
  
}
```



Here, we defined a class named `Lamp`.

The class has one property `isOn` (defined in same way as variable), and two member functions `turnOn()` and `turnOff()`.

In Kotlin, either the property must be initialized or must be declared `abstract`. In the above example, `isOn` property is initialized to `false`.

Classes, objects, properties, member function etc. can have visibility modifiers. For example, the `isOn` property is private. This means, the `isOn` property can be changed from only inside the `Lamp` class.

Other visibility modifiers are:

- `private` - visible (can be accessed) from inside the class only.
- `public` - visible everywhere.
- `protected` - visible to the class and its subclass.
- `internal` - any client inside the module can access them.

If you do not specify the visibility modifier, it will be `public` by default.

In the above program, `turnOn()` and `turnOff()` member functions are `public` whereas, `isOn` property is private.

Kotlin Objects

When the class is defined, only the specification for the object is defined; no memory or storage is allocated.

To access members defined within the class, you need to create objects. Let's create objects of `Lamp` class.



Kotlin Class and Object

www.programmerway.com



```
class Lamp {  
  
    // property (data member)  
    private var isOn: Boolean = false  
  
    // member function  
    fun turnOn() {  
        isOn = true  
    }  
  
    // member function  
    fun turnOff() {  
        isOn = false  
    }  
}  
  
fun main(args: Array<String>) {  
  
    val l1 = Lamp() // create l1 object of Lamp class  
    val l2 = Lamp() // create l2 object of Lamp class  
}
```

This program creates two objects l1 and l2 of class `Lamp`. The `isOn` property for both lamps l1 and l2 will be `false`.

How to access members?

You can access properties and member functions of a class by using `.` notation. For example,

```
l1.turnOn()
```

this statement calls `turnOn()` function for l1 object.

Let's take another example:

```
l2.isOn = true
```

Here, we tried to assign `true` to `isOn` property of l2 object. Note that, `isOn` property is `private`, and if you try to access `isOn` from outside the class, an exception is thrown.



Example: Kotlin Class and Object

```
class Lamp {  
  
    // property (data member)  
    private var isOn: Boolean = false  
  
    // member function  
    fun turnOn() {  
        isOn = true  
    }  
  
    // member function  
    fun turnOff() {  
        isOn = false  
    }  
  
    fun displayLightStatus(lamp: String) {  
        if (isOn == true)  
            println("$lamp lamp is on.")  
        else  
            println("$lamp lamp is off.")  
    }  
}  
  
fun main(args: Array<String>) {  
  
    val l1 = Lamp() // create l1 object of Lamp class  
    val l2 = Lamp() // create l2 object of Lamp class  
  
    l1.turnOn()  
    l2.turnOff()  
  
    l1.displayLightStatus("l1")  
    l2.displayLightStatus("l2")  
}
```

```
l1 Lamp is on.  
l2 Lamp is off.
```



In the above program,

- `Lamp` class is created.
- The class has a property `isOn` and three member functions `turnOn()`, `turnOff()` and `displayLightStatus()`.
- Two objects `l1` and `l2` of `Lamp` class are created in the `main()` function.
- Here, `turnOn()` function is called using `l1` object: `l1.turnOn()`. This method sets `isOn` instance variable of `l1` objects to `true`.
- And, `turnOff()` function is called using `l2` object: `l2.turnOff()`. This method sets `isOn` instance variable of `l2` objects to `false`.
- Then, `displayLightStatus()` function is called for `l1` and `l2` objects which prints appropriate message depending on whether `isOn` property is `true` or `false`.

Notice that, the `isOn` property is initialized to `false` inside the class. When an object of the class is created, `isOn` property for the object is initialized to `false` automatically. So, it's not necessary for `l2` object to call `turnOff()` to set `isOn` property to `false`.

```
class Lamp {  
    // property (data member)  
    private var isOn: Boolean = false  
    // member function  
    fun turnOn() {  
        isOn = true  
    }  
    // member function  
    fun turnOff() {  
        isOn = false  
    }  
    fun displayLightStatus() {  
        if (isOn == true)  
            println("lamp is on.")  
        else  
            println("lamp is off.")  
    }  
}  
  
fun main(args: Array<String>) {  
  
    val lamp = Lamp()  
    lamp.displayLightStatus()  
}
```

lamp is off.



Kotlin Constructors

A constructor is a concise way to initialize class properties.

It is a special member function that is called when an object is instantiated (created). However, however, how they work in Kotlin is slightly different.

In Kotlin, there are two constructors:

- **Primary constructor** – concise way to initialize a class
- **Secondary constructor** – allows you to put additional initialization logic

Primary Constructor

The primary constructor is part of the class header. Here's an example:

```
class Person(val firstName: String, var age: Int) {  
  
    // class body  
  
}
```

The block of code surrounded by parenthesis is the primary constructor:

`(val firstName: String, var age: Int)`

The constructor declared two properties: `firstName` (read-only property as it's declared using keyword `val`) and `age` (read-write property as it is declared with keyword `var`).

Example: Primary Constructor

```
fun main(args: Array<String>) {  
  
    val person1 = Person("Joe", 25)  
  
    println("First Name = ${person1.firstName}")  
    println("Age = ${person1.age}")  
}  
  
class Person(val firstName: String, var age: Int) {  
  
}
```

```
First Name = Joe  
Age = 25
```



When the object of `Person` class is created, `"Joe"` and `25` values are passed as if `Person` is a function.

This initializes `firstName` and `age` properties of `person1` object to `"Joe"` and `25` respectively.

There are other ways of using primary constructors.

Primary Constructor and Initializer Blocks

The primary constructor has a constrained syntax, and cannot contain any code.

To put the initialization code (not only code to initialize properties), initializer block is used. It is prefixed with `init` keyword. Let's modify the above example with initializer block:

```
fun main(args: Array<String>) {  
    val person1 = Person("joe", 25)  
}  
  
class Person(fName: String, personAge: Int) {  
    val firstName: String  
    var age: Int  
  
    // initializer block  
    init {  
        firstName = fName.capitalize()  
        age = personAge  
  
        println("First Name = $firstName")  
        println("Age = $age")  
    }  
}
```

```
First Name = Joe  
Age = 25
```

Here, parameters `fName` and `personAge` inside the parenthesis accepts values `"Joe"` and `25` respectively when `person1` object is created. However, `fName` and `personAge` are used without using `var` or `val`, and are not properties of the `Person` class.

The `Person` class has two properties, `firstName` and `age` are declared.

When `person1` object is created, code inside initializer block is executed. The initializer block not only initializes its properties but also prints them.

Here is another way to perform the same task:



```
fun main(args: Array<String>) {  
    val person1 = Person("joe", 25)  
}  
  
class Person(fName: String, personAge: Int) {  
    val firstName = fName.capitalize()  
    var age = personAge  
  
    // initializer block  
    init {  
        println("First Name = $firstName")  
        println("Age = $age")  
    }  
}
```

To distinguish the constructor parameter and property, different names are used (`fName` and `firstName`, `personAge` and `age`). It's more common to use `_firstName` and `_age` instead of completely different name for constructor parameters.

For example:

```
class Person(_firstName: String, _age: Int) {  
  
    val firstName = _firstName.capitalize()  
  
    var age = _age  
  
    // initializer block  
  
    init {  
  
        ... ..  
  
    }  
  
}
```



Default Value in Primary Constructor

You can provide default values to constructor parameters (similar to providing default arguments to functions).

For example:

```
fun main(args: Array<String>) {

    println("person1 is instantiated")
    val person1 = Person("joe", 25)

    println("person2 is instantiated")
    val person2 = Person("Jack")

    println("person3 is instantiated")
    val person3 = Person()
}

class Person(_firstName: String = "UNKNOWN", _age: Int = 0) {
    val firstName = _firstName.capitalize()
    var age = _age

    // initializer block
    init {
        println("First Name = $firstName")
        println("Age = $age\n")
    }
}
```

When you run the program, the output will be:

```
First Name = Joe
Age = 25

person2 is instantiated
First Name = Jack
Age = 0

person3 is instantiated
First Name = UNKNOWN
Age = 0
```



Kotlin Secondary Constructor

In Kotlin, a class can also contain one or more secondary constructors. They are created using `constructor` keyword.

Secondary constructors are not that common in Kotlin. The most common use of secondary constructor comes up when you need to extend a class that provides multiple constructors that initialize the class in different ways.

Here's how you can create a secondary constructor in Kotlin:

```
class
  constructor(data:
    //
  }
  constructor
    (data:
      String,
      //some
    )
  }
}
```

Log{
String){
code
Int){
code

Here, the `Log` class has two secondary constructors, but no primary constructor.

You can extend the class as:

```
class Log {

    constructor(data: String) {

        // code

    }

    constructor(data: String, numberOfData: Int) {

        // code

    }

}

class AuthLog: Log {

    constructor(data: String): super(data) {

        // code

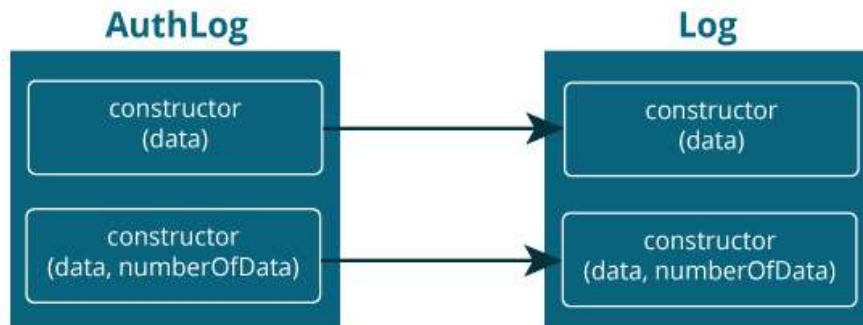
    }

    constructor(data: String, numberOfData: Int): super(data, numberOfData) {
```



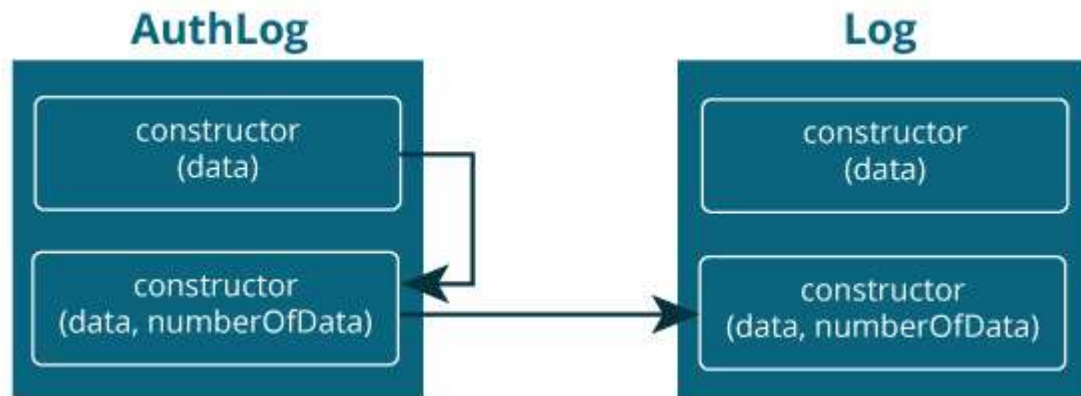
```
// code  
  
}  
  
}
```

Here, constructors of the derived class `AuthLog` calls the corresponding constructor of the base class `Log`. For that, `super()` is used.



In Kotlin, you can also call a constructor from another constructor of the same class (like in Java) using `this()`.

```
class AuthLog: Log {  
  
    constructor(data: String): this(data, 10) {  
  
        // code  
  
    }  
  
    constructor(data: String, numberOfData: Int): super(data, numberOfData) {  
  
        // code  
  
    }  
  
}
```



Example: Kotlin Secondary Constructor

```
fun main(args: Array<String>) {

    val p1 = AuthLog("Bad Password")
}

open class Log {
    var data: String = ""
    var numberOfData = 0
    constructor(_data: String) {

    }
    constructor(_data: String, _numberOfData: Int) {
        data = _data
        numberOfData = _numberOfData
        println("$data: $numberOfData times")
    }
}

class AuthLog: Log {
    constructor(_data: String): this("From AuthLog -> " + _data, 10) {
    }

    constructor(_data: String, _numberOfData: Int): super(_data, _numberOfData) {
    }
}
```

From AuthLog -> Bad Password: 10 times



NOTE:

The secondary constructor must initialize the base class or delegate to another constructor (like in above example) if the class has no primary constructor.

Kotlin Getters and Setters

In programming, getters are used for getting value of the property. Similarly, setters are used for setting value of the property.

In Kotlin, getters and setters are optional and are auto-generated if you do not create them in your program.

How getters and setters work?

The following code in Kotlin,

```
class Person {  
  
    var name: String = "defaultValue"  
  
}
```

is equivalent to

```
class Person {  
  
    var name: String = "defaultValue"  
  
    // getter  
  
    get() = field  
  
    // setter  
  
    set(value) {  
  
        field = value  
  
    }  
  
}
```



When you instantiate object of the `Person` class and initialize the `name` property, it is passed to the setters parameter `value` and sets `field` to `value`.

```
val p = Person()

p.name = "jack"
```

Now, when you access `name` property of the object, you will get `field` because of the code `get() = field`.

```
println("${p.name}")
```

here's a working example:

```
fun main(args: Array<String>) {

    val p = Person()
    p.name = "jack"
    println("${p.name}")
}

class Person {
    var name: String = "defaultValue"

    get() = field

    set(value) {
        field = value
    }
}
```

```
jack
```

This is how getters and setters work by default. However, you can change value of the property (modifier value) using getters and setters.



Example: Changing value of the property

```
fun main(args: Array<String>) {

    val maria = Girl()
    maria.actualAge = 15
    maria.age = 15
    println("Maria: actual age = ${maria.actualAge}")
    println("Maria: pretended age = ${maria.age}")

    val angela = Girl()
    angela.actualAge = 35
    angela.age = 35
    println("Angela: actual age = ${angela.actualAge}")
    println("Angela: pretended age = ${angela.age}")
}

class Girl {
    var age: Int = 0
    get() = field
    set(value) {
        field = if (value < 18)
            18
        else if (value >= 18 && value <= 30)
            value
        else
            value-3
    }

    var actualAge: Int = 0
}
```

```
Maria: actual age = 15
Maria: pretended age = 18
Angela: actual age = 35
Angela: pretended age = 32
```

Here, the `actualAge` property works as expected. However, there is additional logic in setters to modify value of the age property.



Kotlin Inheritance

Inheritance is one of the key features of object-oriented programming. It allows user to create a new class (derived class) from an existing class (base class).

The derived class inherits all the features from the base class and can have additional features of its own.

Why Inheritance?

Suppose, in your application, you want three characters – a **math teacher**, a **footballer** and a **businessman**.

Since, all of the characters are persons, they can walk and talk. However, they also have some special skills. A math teacher can **teach math**, a footballer can play **football** and businessman can **run a business**.

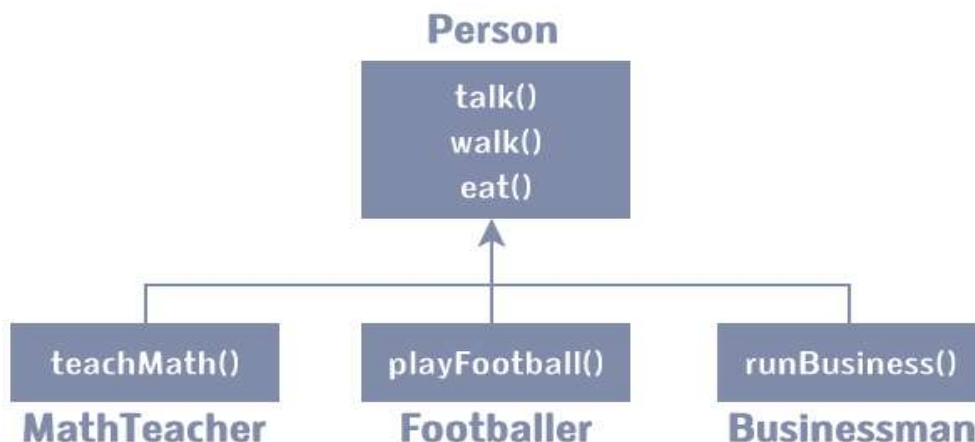
You can individually create three classes who can walk, talk and perform their special skills.



In each of the classes, you would be copying the same code for walk and talk for each character.

If you want to add a new feature – **eat**, you need to implement the same code for each character. This can easily become error prone (when copying) and duplicate codes.

It would be a lot easier if we had a **Person** class with basic features like talk, walk, eat, sleep and add special skills to those features as per our characters. This is done using inheritance.



Using inheritance, now you don't implement the same code for **walk()**, **talk()** and **eat()** for each class. You just need to **inherit** them.

So, for **MathTeacher** (derived class), you inherit all features of a **Person** (base class) and add a new feature **teachMath()**. Likewise, for the **Footballer** class, you inherit all the features of the **Person** class and add a new feature **playFootball()** and so on.



This makes your code cleaner, understandable and extendable.

It is important to remember: When working with inheritance, each derived class should satisfy the condition whether it “is a” base class or not.

In the example above, `MathTeacher` is a `Person`, `Footballer` is a `Person`. You cannot have something like, `Businessman` is a `Business`.

Let's try to implement the above discussion in code:

```
open class Person(age: Int) {  
  
    // code for eating, talking, walking  
  
}  
  
class MathTeacher(age: Int): Person(age) {  
  
    // other features of math teacher  
  
}  
  
class Footballer(age: Int): Person(age) {  
  
    // other features of footballer  
  
}  
  
class Businessman(age: Int): Person(age) {  
  
    // other features of businessman  
  
}
```

Here, `Person` is a base class, and classes `MathTeacher`, `Footballer` and `Businessman` are derived from the `Person` class.

Notice, the keyword `open` before the base class `Person`, it's important.

By default, classes in Kotlin are `final`. If you are familiar with Java, you know that a final class cannot be subclassed. By using the `open` annotation on a class, compiler allows you to derive new classes from it.

```
open class Person(age: Int, name: String) {  
    init {  
        println("My name is $name.")  
        println("My age is $age")  
    }  
}
```



```
class MathTeacher(age: Int, name: String): Person(age, name) {  
  
    fun teachMaths() {  
        println("I teach in primary school.")  
    }  
}  
  
class Footballer(age: Int, name: String): Person(age, name) {  
    fun playFootball() {  
        println("I play for LA Galaxy.")  
    }  
}  
  
fun main(args: Array<String>) {  
    val t1 = MathTeacher(25, "Jack")  
    t1.teachMaths()  
  
    println()  
  
    val f1 = Footballer(29, "Christiano")  
    f1.playFootball()  
}
```

My name is Jack.
My age is 25
I teach in primary school.

My name is Cristiano.
My age is 29
I play for LA Galaxy.

Here, two classes `MathTeacher` and `Footballer` are derived from the `Person` class.

The primary constructor of the `Person` class declared two properties: `age` and `name`, and it has an initializer block. The initializer block (and member functions) of the base class `Person` can be accessed by the objects of derived classes (`MathTeacher` and `Footballer`).

Derived classes `MathTeacher` and `Footballer` have their own member functions `teachMaths()` and `playFootball()` respectively. These functions are accessible only from the objects of their respective class.

When the object `t1` of `MathTeacher` class is created,



```
val t1 = MathTeacher(25, "Jack")
```

The parameters are passed to the primary constructor. In Kotlin, `init` block is called when the object is created. Since, `MathTeacher` is derived from `Person` class, it looks for initializer block in the base class (`Person`) and executes it. If the `MathTeacher` had `init` block, the compiler would have also executed the `init` block of the derived class.

Next, the `teachMaths()` function for the object `t1` is called using statement `t1.teachMaths()`.

The program works similarly when object `f1` of `Footballer` class is created. It executes the `init` block of the base class. Then, the `playFootball()` method of `Footballer` class is called using statement `f1.playFootball()`.

Important Notes: Kotlin Inheritance

- If the class has a primary constructor, the base must be initialized using the parameters of the primary constructor. In the above program, both derived classes have two parameters `age` and `name`, and both these parameters are initialized in primary constructor in the base class.

Here's another example:

```
open class Person(age: Int, name: String) {
    // some code
}

class Footballer(age: Int, name: String, club: String): Person(age, name) {
    init {
        println("Football player $name of age $age and plays for $club.")
    }

    fun playFootball() {
        println("I am playing football.")
    }
}

fun main(args: Array<String>) {
    val f1 = Footballer(29, "Cristiano", "LA Galaxy")
}
```

Here the primary constructor of the derived class has 3 parameters, and the base class has 2 parameters. Note that, both parameters of the base class are initialized.

- In case of no primary constructor, each base class has to initialize the base (using `super` keyword), or delegate to another constructor which does that. For example:



```
fun main(args: Array<String>) {

    val p1 = AuthLog("Bad Password")
}
open class Log {
    var data: String = ""
    var numberOfData = 0
    constructor(_data: String) {

    }

    constructor(_data: String, _numberOfData: Int) {
        data = _data
        numberOfData = _numberOfData
        println("$data: $numberOfData times")
    }
}
class AuthLog: Log {
    constructor(_data: String): this("From AuthLog -> + $_data", 10) {
    }

    constructor(_data: String, _numberOfData: Int): super(_data, _numberOfData) {
    }
}
```

Overriding Member Functions and Properties

If the base class and the derived class contains a member function (or property) with the same name, you can need to override the member function of the derived class using `override` keyword, and use `open` keyword for the member function of the base class.

Example: Overriding Member Function

```
// Empty primary constructor
open class Person() {
    open fun displayAge(age: Int) {
        println("My age is $age.")
    }
}
class Girl: Person() {

    override fun displayAge(age: Int) {
        println("My fake age is ${age - 5}.")
    }
}
```



```
}  
}  
fun main(args: Array<String>) {  
    val girl = Girl()  
    girl.displayAge(31)  
}
```

My fake age is 26.

Here, `girl.displayAge(31)` calls the `displayAge()` method of the derived class `Girl`.

You can override property of the base class in similar way.

```
// Empty primary constructor  
open class Person() {  
    open var age: Int = 0  
    get() = field  
  
    set(value) {  
        field = value  
    }  
}  
  
class Girl: Person() {  
  
    override var age: Int = 0  
    get() = field  
  
    set(value) {  
        field = value - 5  
    }  
}  
  
fun main(args: Array<String>) {  
  
    val girl = Girl()  
    girl.age = 31  
    println("My fake age is ${girl.age}.")  
}
```



My fake age is 26.

As you can see, we have used `override` and `open` keywords for `age` property in derived class and base class respectively.

Calling Members of Base Class from Derived Class

You can call functions (and access properties) of the base class from a derived class using `super` keyword.

```
open class Person() {
    open fun displayAge(age: Int) {
        println("My actual age is $age.")
    }
}

class Girl: Person() {

    override fun displayAge(age: Int) {

        // calling function of base class
        super.displayAge(age)

        println("My fake age is ${age - 5}.")
    }
}

fun main(args: Array<String>) {
    val girl = Girl()
    girl.displayAge(31)
}
```

My age is 31.
My fake age is 26.



Kotlin Visibility Modifiers

Visibility modifiers are keywords that set the **visibility** (accessibility) of classes, objects, interface, constructors, functions, properties and their setters. (You cannot set visibility modifier of **getters** as they always take the same visibility as that of the property.)

Visibility Modifiers Inside Package

A package organizes a set of related functions, properties and classes, objects and interfaces.

Modifier	Description
Public	Declarations are visible everywhere
Private	Visible inside the file containing the declaration
Internal	Visible inside the same module (a set of Kotlin files compiled together)
Protected	Not available for packages (used for subclasses)

Note: If visibility modifier is not specified, it is **Public** by default.

Example:

```
// file name: hello.kt

package test

fun function1() {} // public by default and visible everywhere

private fun function2() {} // visible inside hello.kt

internal fun function3() {} // visible inside the same module

var name = "Foo" // visible everywhere

    get() = field // visible inside hello.kt (same as its property)

    private set(value) { // visible inside hello.kt

        field = value

    }

private class class1 {} // visible inside hello.kt
```




Visibility Modifiers Inside Classes and Interfaces

Here's how visibility modifiers work for members (functions, properties) declared inside a class:

Modifier	Description
Public	Visible to any client who can see the declaring class
Private	Visible inside the class only
Protected	Visible inside the class and its subclasses
Internal	Visible to any client inside the module that can see the declaring class

Note: If you override a **Protected** member in the derived class without specifying its visibility, its visibility will also be **Protected**.

Example:

```
open class Base() {  
  
    var a = 1          // public by default  
  
    private var b = 2   // private to Base class  
  
    protected open val c = 3 // visible to the Base and the Derived class  
  
    internal val d = 4    // visible inside the same module  
  
    protected fun e() {}  // visible to the Base and the Derived class  
}  
  
class Derived: Base() {  
  
    // a, c, d, and e() of the Base class are visible  
  
    // b is not visible  
  
    override val c = 9    // c is protected  
}  
  
fun main(args: Array<String>) {  
  
    val base = Base()  
}
```



```
// base.a and base.d are visible

// base.b, base.c and base.e() are not visible

val derived = Derived()

// derived.c is not visible

}
```

Changing Visibility of a Constructor

By default, the visibility of a constructor is **Public**. However, you can change it. For that, you need to explicitly add **constructor** keyword.

The constructor is **Public** by default in the example below:

```
class Test(val a: Int) {

    // code

}
```

Here's how you can change its visibility.

```
class Test private constructor(val a: Int) {

    // code

}
```

Here the constructor is **Private**.

Note: In Kotlin, local functions, variables and classes cannot have visibility modifiers.



Kotlin Abstract Class

Like Java, `abstract` keyword is used to declare abstract classes in Kotlin. An abstract class cannot be instantiated (you cannot create objects of an abstract class). However, you can inherit subclasses from them.

The members (properties and methods) of an abstract class are `non-abstract` unless you explicitly use `abstract` keyword to make them abstract.

Example:

```
abstract class Person {  
  
    var age: Int = 40  
  
    fun displaySSN(ssn: Int) {  
  
        println("My SSN is $ssn.")  
  
    }  
  
    abstract fun displayJob(description: String)  
  
}
```

Here,

- An abstract class `Person` is created. You cannot create objects of the class.
- The class has a non-abstract property `age` and a non-abstract method `displaySSN()`. If you need to override these members in the subclass, they should be marked with `open` keyword.
- The class has an abstract method `displayJob()`. It doesn't have any implementation and must be overridden in its subclasses.

Note:

Abstract classes are always open. You do not need to explicitly use `open` keyword to inherit subclasses from them.

Example: Kotlin Abstract Class and Method

```
abstract class Person(name: String) {  
  
    init {  
        println("My name is $name.")  
    }  
    fun displaySSN(ssn: Int) {  
        println("My SSN is $ssn.")  
    }  
}
```



```
abstract fun displayJob(description: String)
}
class Teacher(name: String): Person(name) {

    override fun displayJob(description: String) {
        println(description)
    }
}
fun main(args: Array<String>) {
    val jack = Teacher("Jack Smith")
    jack.displayJob("I'm a mathematics teacher.")
    jack.displaySSN(23123)
}
```

```
My name is Jack Smith.
I'm a mathematics teacher.
My SSN is 23123.
```

Here, a class `Teacher` is derived from an abstract class `Person`.

An object `jack` of `Teacher` class is instantiated. We have passed `"Jack Smith"` as a parameter to the primary constructor while creating it. This executes the initializer block of the `Person` class.

Then, `displayJob()` method is called using `jack` object. Note, that the `displayJob()` method is declared abstract in the base class, and overridden in the derived class.

Finally, `displaySSN()` method is called using `jack` object. The method is non-abstract and declared in `Person` class (and not declared in `Teacher` class).

Kotlin `interfaces` are similar to `abstract` classes. However, interfaces `cannot store state` whereas abstract classes can.

Meaning, interface may have property but it needs to be abstract or has to provide accessor implementations. Whereas, it's not mandatory for property of an abstract class to be abstract.



Kotlin Interfaces

Kotlin **interfaces** are similar to interfaces in Java 8. They can contain definitions of abstract methods as well as implementations of non-abstract methods. However, they cannot contain any state.

Meaning, interface may have property but it needs to be abstract or has to provide accessor implementations.

Abstract classes in Kotlin are similar to interface with one important difference. It's not mandatory for properties of an abstract class to be abstract or provide accessor implementations.

How to define an interface?

Keyword **interface** is used to define interfaces in Kotlin. For example:

```
interface MyInterface {  
  
    var test: String // abstract property  
  
    fun foo()        // abstract method  
  
    fun hello() = "Hello there" // method with default implementation  
  
}
```

Here,

- An interface **MyInterface** is created.
- The interface has an abstract property **test** and an abstract method **foo()**.
- The interface also has an non-abstract method **hello()**.

How to implement interface?

Here's how a class or object can implement the interface:

```
interface MyInterface {  
  
    val test: Int // abstract property  
  
    fun foo() : String // abstract method (returns String)  
  
    fun hello() { // method with default implementation  
  
        // body (optional)  
  
    }  
  
}  
  
class InterfacImp : MyInterface {
```



```
override val test: Int = 25

override fun foo() = "Lol"

// other code

}
```

Here, a class `Interfacelmp` implements the interface `MyInterface`.

The class overrides abstract members (`test` property and `foo()` method) of the interface.

[Example: How interface works?](#)

```
interface MyInterface {

    val test: Int

    fun foo() : String

    fun hello() {
        println("Hello there, pal!")
    }
}

class Interfacelmp : MyInterface {

    override val test: Int = 25
    override fun foo() = "Lol"

}

fun main(args: Array<String>) {
    val obj = Interfacelmp()

    println("test = ${obj.test}")
    print("Calling hello(): ")

    obj.hello()

    print("Calling and printing foo(): ")
    println(obj.foo())
}
```



```
test = 25
Calling hello(): Hello there, pal!
Calling and printing foo(): Lol
```

As mentioned above, an interface may also have a property that provide accessor implementation. For example:

```
interface MyInterface {
    // property with implementation
    val prop: Int
    get() = 23
}
class InterfacImp : MyInterface {
    // class body
}
fun main(args: Array<String>) {
    val obj = InterfacImp()

    println(obj.prop)
}
```

When you run the program, the output will be:

```
23
```

Here, `prop` is not abstract. However, it's valid inside the interface because it provides implementation for accessor.

However, you cannot do something like `val prop: Int = 23` inside the interface.

Implementing Two or More Interfaces in a Class

Kotlin does not allow true [multiple inheritance](#). However, it's possible to implement two or more interfaces in a single class. For example:

```
interface A {

    fun callMe() {
        println("From interface A")
    }
}
```



```
interface B {  
    fun callMeToo() {  
        println("From interface B")  
    }  
}  
  
// implements two interfaces A and B  
class Child: A, B  
  
fun main(args: Array<String>) {  
    val obj = Child()  
  
    obj.callMe()  
    obj.callMeToo()  
}
```

From interface A
From interface B

Resolving overriding conflicts (Multiple Interface)

Suppose, two interfaces (A and B) have a non-abstract method with the same name (let's say callMe() method). You implemented these two interfaces in a class (let's say C). Now, if you call the callMe() method using the objects of class C, compiler will through error.

For example:

```
interface A {  
  
    fun callMe() {  
        println("From interface A")  
    }  
}  
  
interface B {  
    fun callMe() {  
        println("From interface B")  
    }  
}
```




```
class Child: A, B

fun main(args: Array<String>) {
    val obj = Child()

    obj.callMe()
}
```

Here's the error:

```
Error:(14, 1) Kotlin: Class 'C' must override public open fun callMe(): Unit defined in A because it inherits multiple interface methods of it
```

To solve this issue, you need to provide your own implementation. Here's how:

```
interface A {
    fun callMe() {
        println("From interface A")
    }
}

interface B {
    fun callMe() {
        println("From interface B")
    }
}

class C: A, B {
    override fun callMe() {
        super<A>.callMe()
        super<B>.callMe()
    }
}

fun main(args: Array<String>) {
    val obj = C()
    obj.callMe()
}
```

```
From interface A
From interface B
```



Here, explicit implementation of `callMe()` method is provided in class `C`.

```
class C: A, B {  
  
    override fun callMe() {  
  
        super<A>.callMe()  
  
        super<B>.callMe()}}  
  
// The statement super<A>.callMe() calls the callMe() method of class A. Similarly, super<B>.callMe() calls  
the callMe() method of class B.
```

Kotlin Nested and Inner Classes

Kotlin Nested Class

Similar like Java, Kotlin allows you to define a class within another class known as `nested` class.

```
class Outer {  
  
    ... ..  
  
    class Nested {  
  
        ... ..  
  
    }  
  
}
```

Since `Nested` class is a member of its enclosing class `Outer`, you can use “.” notation to access `Nested` class and its members.

Example: Kotlin Nested Class

```
class Outer {  
    val a = "Outside Nested class."  
    class Nested {  
        val b = "Inside Nested class."  
        fun callMe() = "Function call from inside Nested class."  
    }  
}  
  
fun main(args: Array<String>) {  
    // accessing member of Nested class
```



```
println(Outer.Nested().b)

// creating object of Nested class
val nested = Outer.Nested()
println(nested.callMe())
}
```

Inside Nested class.
Function call from inside Nested class.

For Java Users

The nested class in Kotlin is similar to static nested class in Java.

In Java, when you declare a class inside another class, it becomes an inner class by default. However, in Kotlin, you need to use `inner` modifier to create an inner class.

Kotlin Inner Class

The nested classes in Kotlin do not have access to the outer class instance. For example:

```
class Outer {
    val foo = "Outside Nested class."

    class Nested {
        // Error! cannot access member of outer class.
        fun callMe() = foo
    }
}

fun main(args: Array<String>) {

    val outer = Outer()
    println(outer.Nested().callMe())
}
```

The above code won't compile because we tried to access `foo` property of `Outer` class from inside `Nested` class.

In order to solve this issue, you need to mark the nested class with `inner` to create an inner class. Inner classes carry a reference to an outer class, and can access outer class members.



Example: Kotlin Inner Class

```
class Outer {  
  
    val a = "Outside Nested class."  
  
    inner class Inner {  
        fun callMe() = a  
    }  
}  
  
fun main(args: Array<String>) {  
  
    val outer = Outer()  
    println("Using outer object: ${outer.Inner().callMe()}")  
  
    val inner = Outer().Inner()  
    println("Using inner object: ${inner.callMe()}")  
}
```

Using outer object: Outside Nested class.
Using inner object: Outside Nested class.



Kotlin Data Class

There may arise a situation where you need to create a class solely to hold data. In such cases, you can mark the class as `data` to create a data class. For example:

```
data class Person(val name: String, var age: Int)
```

For this class, the compiler automatically generates:

- `Copy()` function, `equals()` and `hashCode()` pair, and `toString()` form of the primary constructor
- `componentN()` functions

Kotlin Data Class Requirements

Here are the requirements:

- The primary constructor must have at least one parameter.
- The parameters of the primary constructor must be marked as either `val` (read-only) or `var` (read-write).
- The class cannot be open, abstract, inner or sealed.
- The class may extend other classes or implement interfaces. If you are using Kotlin version before 1.1, the class only implements interfaces.

Example: Kotlin Data Class

```
data class User(val name: String, val age: Int)

fun main(args: Array<String>) {
    val jack = User("jack", 29)
    println("name = ${jack.name}")
    println("age = ${jack.age}")
}
```

```
name = jack
age = 29
```

when you declare a data class, the compiler automatically generates several functions such as `toString()`, `equals()`, `hashCode()` etc behind the scenes. This helps to keep your code concise.



Copying

For a data class, you can create a copy of an object with some of its properties different using `copy()` function.

Example:

```
data class User(val name: String, val age: Int)

fun main(args: Array<String>) {
    val u1 = User("John", 29)

    // using copy function to create an object
    val u2 = u1.copy(name = "Randy")

    println("u1: name = ${u1.name}, name = ${u1.age}")
    println("u2: name = ${u2.name}, name = ${u2.age}")
}
```

```
u1: name = John, name = 29
u2: name = Randy, name = 29
```

toString() Method

The `toString()` function returns a string representation of the object.

```
data class User(val name: String, val age: Int)

fun main(args: Array<String>) {
    val u1 = User("John", 29)
    println(u1.toString())
}
```

```
User(name=John, age=29)
```



hashCode() and equals()

The `hashCode()` method returns hash code for the object. If two objects are equal, `hashCode()` produces the same integer result.

The `equals()` returns `true` if two objects are equal (has same `hashCode()`). If objects are not equal, `equals()` returns `false`.

```
data class User(val name: String, val age: Int)
```

```
fun main(args: Array<String>) {
```

```
    val u1 = User("John", 29)
```

```
    val u2 = u1.copy()
```

```
    val u3 = u1.copy(name = "Amanda")
```

```
    println("u1 hashcode = ${u1.hashCode()}")
```

```
    println("u2 hashcode = ${u2.hashCode()}")
```

```
    println("u3 hashcode = ${u3.hashCode()}")
```

```
    if (u1.equals(u2) == true)
```

```
        println("u1 is equal to u2.")
```

```
    else
```

```
        println("u1 is not equal to u2.")
```

```
    if (u1.equals(u3) == true)
```

```
        println("u1 is equal to u3.")
```

```
    else
```

```
        println("u1 is not equal to u3.")
```

```
}
```

```
u1 hashcode = 71750738
```

```
u2 hashcode = 71750738
```

```
u3 hashcode = 771732263
```

```
u1 is equal to u2.
```

```
u1 is not equal to u3.
```



Destructuring Declarations

You can destructure an object into a number of variables using Destructuring declaration.

Example:

```
data class User(val name: String, val age: Int, val gender: String)

fun main(args: Array<String>) {
    val u1 = User("John", 29, "Male")

    val (name, age, gender) = u1
    println("name = $name")
    println("age = $age")
    println("gender = $gender")
}
```

```
name = John
age = 29
gender = Male
```

This was possible because the compiler generates `componentN()` functions all properties for a data class. For example:

```
data class User(val name: String, val age: Int, val gender: String)

fun main(args: Array<String>) {
    val u1 = User("John", 29, "Male")

    println(u1.component1()) // John
    println(u1.component2()) // 29
    println(u1.component3()) // "Male"
}
```

```
John
29
Male
```




Kotlin Sealed Classes

Sealed classes are used when a value can have only one of the types from a limited set (restricted hierarchies).

Like abstract classes, Sealed classes allow you to represent hierarchies.

Let's explore what problem they solve. Let's example:

```
class Expr
class Const(val value: Int) : Expr
class Sum(val left: Expr, val right: Expr) : Expr

fun eval(e: Expr): Int =
    when (e) {
        is Const -> e.value
        is Sum -> eval(e.right) + eval(e.left)
        else ->
            throw IllegalArgumentException("Unknown expression")
    }
```

In the above program, the base class `Expr` has two derived classes `Const` (represents a number) and `Sum` (represents sum of two expressions). Here, it's mandatory to use `else` branch for default condition in `when` expression.

Now, if you derive a new subclass from `Expr` class, the compiler won't detect anything as `else` branch handles it which can lead to bugs. It would have been better if the compiler issued an error when we added a new subclass.

To solve this problem, you can use sealed class. As mentioned, sealed class restricts the possibility of creating subclasses. And, when you handle all subclasses of a sealed class in an `when` expression, it's not necessary to use `else` branch.

To create a sealed class, sealed modifier is used. For example:

```
sealed class Expr
```



Example: Sealed Class

Here's how you can solve the above problem using sealed class:

```
when class Expr
class Const(val value: Int) : Expr()
class Sum(val left: Expr, val right: Expr) : Expr()
object NotANumber : Expr()

fun eval(e: Expr): Int =
    when (e) {
        is Const -> e.value
        is Sum -> eval(e.right) + eval(e.left)
        NotANumber -> java.lang.Double.NaN
    }
```

As you can see, there is no `else` branch. If you derive a new subclass from `Expr` class, the compiler will complain unless the subclass is handled in the `when` expression.

Notes:

- All subclasses of a sealed class must be declared in the same file where sealed class is declared.
- A sealed class is `abstract` by itself, and you cannot instantiate objects from it.
- You cannot create non-private constructors of a sealed class; their constructors are `private` by default.

Difference Between Enum and Sealed Class

`Enum class` and `sealed class` are pretty similar. The set of values for an enum type is also restricted like a sealed class.

The only difference is that, enum can have just a `single instance`, whereas a subclass of a sealed class can have `multiple instances`.



Kotlin Object

Kotlin Object Declarations and Expression

Object Declarations

Singleton is an object-oriented pattern where a class can have only one instance (object).

For example: you are working an application having SQL database backend. You want to create a connection pool to access the database while reusing the same connection for all clients. For this, you can create the connection through singleton class so that every client get the same connection.

Kotlin provides an easy way to create singletons using the object declaration feature. For that, **object** keyword is used.

```
object SingletonExample {  
  
    ... ..  
  
    // body of class  
  
    ... ..  
  
}
```

The above code combines a class declaration and a declaration of a single instance **SingletonExample** of the class.

An object declaration can contain properties, methods and so on. However, they are not allowed to have **constructors**.

Similar to objects of a normal class, you can call methods and access properties by using the **.** notation.

Example: Object Declaration

```
object Test {  
    private var a: Int = 0  
    var b: Int = 1  
  
    fun makeMe12(): Int {  
        a = 12  
        return a  
    }  
}  
  
fun main(args: Array<String>) {  
    val result: Int
```



```
result = Test.makeMe12()

println("b = ${Test.b}")
println("result = $result")
}
```

```
b = 1
result = 12
```

Object declaration can inherit from classes and interfaces in a similar way like normal classes.

Singletons and Dependency Injection

Object declarations can be useful sometimes. However, they are not ideal in large software systems that interact with many other parts of the system.

Kotlin Object Expressions

The `object` keyword can also be used to create objects of an anonymous class known as **anonymous objects**. They are used if you need to create an object of a slight modification of some class or interface without declaring a subclass for it. For example,

```
window.addMouseListener(object : MouseAdapter() {

    override fun mouseClicked(e: MouseEvent) {

        // ...

    }

    override fun mouseEntered(e: MouseEvent) {

        // ...

    }

})
```

Here, an anonymous object is declared extending `MouseAdapter` class. The program overrides two `MouseAdapter` methods: `mouseClicked()` and `mouseEntered()`.

If necessary, you can assign a name to the anonymous object and store it in a variable. For example:



```
val obj = object : MouseAdapter() {  
    override fun mouseClicked(e: MouseEvent) {  
        // ...  
    }  
    override fun mouseEntered(e: MouseEvent) {  
        // ...  
    }  
}
```

Example: Kotlin Object Expression

```
open class Person() {  
    fun eat() = println("Eating food.")  
  
    fun talk() = println("Talking with people.")  
  
    open fun pray() = println("Praying god.")  
}  
  
fun main(args: Array<String>) {  
    val atheist = object : Person() {  
        override fun pray() = println("I don't pray. I am an atheist.")  
    }  
  
    atheist.eat()  
    atheist.talk()  
    atheist.pray()  
}
```

Eating food.
Talking with people.
I don't pray. I am an atheist.



Here, anonymous object is stored in variable `atheist` which implements `Person` class with `pray()` methods is overridden.

If you are implementing a class that has a constructor to declare an anonymous object, you need to pass appropriate constructor parameters. For example:

```
open class Person(name: String, age: Int) {  
  
    init {  
        println("name: $name, age: $age")  
    }  
  
    fun eat() = println("Eating food.")  
    fun talk() = println("Talking with people.")  
    open fun pray() = println("Praying god.")  
}  
  
fun main(args: Array<String>) {  
    val atheist = object : Person("Jack", 29) {  
        override fun pray() = println("I don't pray. I am an atheist.")  
    }  
  
    atheist.eat()  
    atheist.talk()  
    atheist.pray()  
}
```

```
name: Jack, age: 29  
Eating food.  
Talking with people.  
I don't pray. I am an atheist.
```



Kotlin Companion Objects

Before taking about companion objects, let's take an example to access members of a class.

```
class Person {  
    fun callMe() = println("I'm called.")  
}  
  
fun main(args: Array<String>) {  
    val p1 = Person()  
  
    // calling callMe() method using object p1  
    p1.callMe()  
}
```

Here, we created an object `p1` of the `Person` class to call `callMe()` method. That's how things normally work.

However, in Kotlin, you can also call `callMe()` method by using the class name, i.e., `Person` in this case. For that, you need to create a companion object by marking `object declaration` with `companion` keyword.

Example: Companion Objects

```
class Person {  
    companion object Test {  
        fun callMe() = println("I'm called.")  
    }  
}  
  
fun main(args: Array<String>) {  
    Person.callMe()  
}
```

I'm called.

In the program, `Test` object declaration is marked with keyword `companion` to create a companion object. Hence, it is possible to call `callMe()` method by using the name of the class as:

```
Person.callMe()
```



The name of the companion object is optional and can be omitted.

```
class Person {  
  
    // name of the companion object is omitted  
    companion object {  
        fun callMe() = println("I'm called.")  
    }  
}  
  
fun main(args: Array<String>) {  
    Person.callMe()  
}
```

If you are familiar with Java, you may relate companion objects with `static` methods (even though how they work internally is totally different).

The companion objects can access private members of the class. Hence, they can be used to implement the `factory method patterns`.



Kotlin Extension Function

Suppose, you need to extend a class with new functionality. In most programming languages, you either derive a new class or use some kind of design pattern to do this.

However, in Kotlin, you can also use extension function to extend a class with new functionality. Basically, an extension function is a member function of a class that is defined outside the class.

For example, you need to use a method to the `String` class that returns a new string with first and last character removed; this method is not already available in `String` class. You can extend function to accomplish this task.

Example: Remove First and Last Character of String

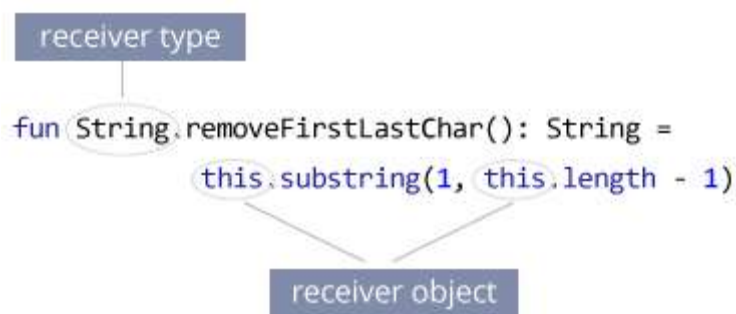
```
fun String.removeFirstLastChar(): String = this.substring(1, this.length - 1)

fun main(args: Array<String>) {
    val myString= "Hello Everyone"
    val result = myString.removeFirstLastChar()
    println("First character is: $result")
}
```

First character is: ello Everyon

Here, an extension function `removeFirstLastChar()` is added to the `String` class.

The class name is the receiver type (`String` class in our example). The `this` keyword inside the extension function refers the receiver object.



If you need to integrate Kotlin on the top of Java project, you do not need to modify the whole code to Kotlin. Just use extension functions to add functionalities.



Kotlin Coroutines

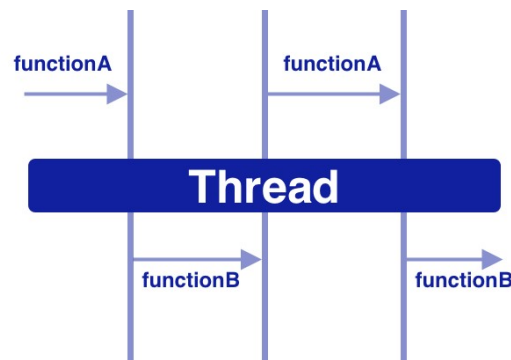
This is a framework which is available to handle multithreading leads to callback hells and blocking states with thread-safe execution.

Also, it is a very efficient and complete framework to manage concurrency (It allows performing multiple tasks or processes simultaneously.) in a more performant and simple way.

Coroutines are a light weight thread, we use a coroutine to perform an operation on other threads, by this our main thread doesn't block and our app doesn't crash.

Coroutines: Coroutines = Co + Routines

Here, **Co** means **cooperation** and **Routines** means **functions**. It meant that when functions cooperate with each other, we call it as Coroutines.



Suppose we have two function as **functionA** and **functionB**.

functionA as below:

```
fun functionA (case: Int) {  
    when (case) {  
        1 -> {  
            taskA1()  
            functionB(1)  
        }  
        2 -> {  
            taskA2()  
            functionB(2)  
        }  
        3 -> {  
            taskA3()  
            functionB(3)  
        }  
        4 -> {  
            taskA4()  
            functionB(4)  
        }  
    }  
}
```

functionB as below:

```
fun functionB (case: Int) {  
    when (case) {  
        1 -> {  
            taskB1()  
            functionA(2)  
        }  
        2 -> {  
            taskB2()  
            functionA(3)  
        }  
        3 -> {  
            taskB3()  
            functionA(4)  
        }  
        4 -> {  
            taskB4()  
        }  
    }  
}
```



Then we call the `functionA` as below:

```
functionA (1)
```

here, `functionA` will do the `taskA1` and give control to the `functionB` to execute the `taskB1`.

Then, `functionB` will do the `taskB1` and give the control back to the `functionA` to execute the `taskA2` and so on.

The important thing is that `functionA` and `functionB` are cooperating with each other.

With Kotlin Coroutines, the above cooperation can be done very easily which is without the use of `when` or `switch case` which I have used in the above example for the sake of understanding.

There are endless possibilities that open up because of the cooperative nature of functions. A few of the possibilities are as follows:

- It can execute a few lines of `functionA` and then execute a few lines of `functionB` and then again, a few lines of `functionA` and so on. [This will be helpful when a thread is sitting idle not doing anything](#), in that case, it can execute a few lines of another function. This way, it can take the full advantage of thread. Ultimately the cooperation helps in multitasking.
- It will enable writing asynchronous code in a synchronous way.

Overall, the Coroutines make the multitasking very easy.

So, we can say that `Coroutines` and the `threads` both are `multitasking`. But the difference is that threads are managed by the OS and coroutines by the users as it can execute a few lines of function by taking advantage of the cooperation.

It's an optimized framework written over the actual threading by taking advantage of the cooperative nature of functions to make it light and yet powerful. So, we can say that Coroutines are `lightweight threads`. A lightweight thread means it doesn't map on the native thread, so it doesn't require context switching on the processor, so they are faster.

What does it mean when I say 'it doesn't map on the native thread'?

Coroutines are available in many languages. Basically, there are two types of Coroutines:

- Stackless
- Stackful

Kotlin implements `Stackless` coroutines – it means that the coroutines don't have their own stack, so they don't map on the native thread.

NOTE:

- Coroutines do not replace threads, it's more like a framework to manage it.
- [Definition of Coroutines](#): A framework to manage concurrency in a more performant and simple way with its lightweight thread which is written on top of the actual threading framework to get the most out of it by taking the advantage of cooperative nature of functions.

Why there is a need for Kotlin Coroutines?

Let's take very standard use-case of an Android Application which is as follows:

- Fetch User from the server
- Show the User in the UI



```
1. fun fetchUser(): User {
2.     // make network call
3.     // return user
4. }
5.
6. fun showUser(user: User) {
7.     // show user
8. }
9.
10. fun fetchAndShowUser() {
11.     val user = fetchUser()
12.     showUser(user)
13. }
```

When we call the `fetchAndShowUser` function, it will throw the `NetworkOnMainThreadException` as the network call is not allowed on the main thread.

There are many ways to solve that. A few of them are as follows:

1. [Using Callback](#): Here, we run the `fetchUser` in the background thread and we pass the result with the callback.

```
1. fun fetchAndShowUser() {
2.     fetchUser { user ->
3.         showUser(user)
4.     }
5. }
```

2. [Using RxJava](#): Reactive world approach. This way we can get rid of the nested callback.

```
1. fetchUser()
2.     .subscribeOn(Schedulers.io())
3.     .observeOn(AndroidSchedulers.mainThread())
4.     .subscribe { user ->
5.         showUser(user)
6.     }
```

3. [Using Coroutines](#): Here the code looks like synchronous, but it is asynchronous.

```
1. suspend fun fetchAndShowUser() {
2.     val user = fetchUser() // fetch on IO thread
3.     showUser(user) // back on UI thread
4. }
```

Here, the above code looks synchronous, but it is asynchronous.

[Implementation of Kotlin Coroutines in Android](#)

Add the Kotlin Coroutines dependencies in the Android project as below:

```
dependencies {
```



```
implementation "org.jetbrains.kotlin:kotlin-coroutines-core:x.x.x"

implementation "org.jetbrains.kotlin:kotlin-coroutines-android:x.x.x"

}
```

Now, our function `fetchUser` will look like below,

```
suspend fun fetchUser(): User {

    return GlobalScope.async(Dispatchers.IO) {

        // make network call

        // return user

    }.await()

}
```

And the `fetchAndShowUser` like below,

```
suspend fun fetchAndShowUser() {

    val user = fetchUser() // fetch on IO thread

    showUser(user) // back on UI thread

}
```

And the `showUser` function as below which is same as it was earlier:

```
fun showUser(user: User) {

    // show user

}
```

We have introduced two things here as follows:

- `Dispatcher`: Dispatchers help coroutines in deciding the thread on which the work has to be done. There are majorly three types of Dispatchers which are as,
 - o `Default`
 - o `Main`
 - o `IO``IO` dispatcher is used to do the network and disk-related work.
 - o `Default``Default` is used to do the CPU intensive work.

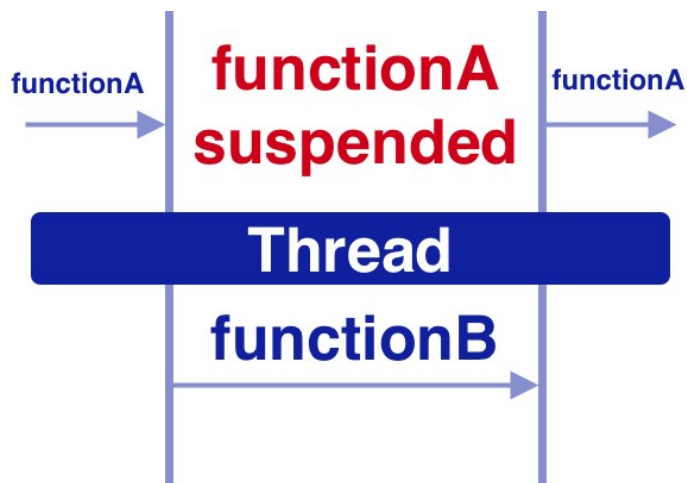


`Main` is the UI thread of Android.

In order to use these, we need to wrap the work under the `async` function. Async function looks like below.

```
suspend fun async() // implementation removed for brevity
```

- `Suspend`: Suspend function is a function that could be started, paused and resume.



Suspend functions are only allowed to be called from a coroutines or another suspend function. You can see that the `async` function which includes the keyword `suspend`. So, in order to use that, we need to make our function `suspend` too.

So, the `fetchAndShowUser` can only be called from another suspend function or a coroutines. We can't make the `onCreate` function of an activity `suspend`, so we need to call it from the coroutines like below:

```
1. override fun onCreate(savedInstanceState: Bundle?) {
2.     super.onCreate(savedInstanceState)
3.
4.     GlobalScope.launch(Dispatchers.Main) {
5.         fetchAndShowUser()
6.     }
7.
8. }
9. }
```

Which is actually,

```
1. override fun onCreate(savedInstanceState: Bundle?) {
2.     super.onCreate(savedInstanceState)
3.
4.     GlobalScope.launch(Dispatchers.Main) {
5.         val user = fetchUser() // fetch on IO thread
6.         showUser(user) // back on UI thread
7.     }
8. }
```



```
8.  
9. }
```

`showUser` will run on UI thread because we have used the `Dispatcher.Main` to launch it.

There are two functions in Kotlin to start the coroutines which are as follows:

- `Launch { }`
- `Async { }`

Launch vs Async in Kotlin Coroutines

The difference is that the `launch { }` does not return anything and the `async { }` returns an instance of `Deferred <T>`, which has an `await ()` function that returns the result of the coroutines like we have future in Java in which we do `future.get ()` to get the result.

In order words:

- `launch`: fire and forget
- `async`: perform a task and return a result

Example:

We have a function `fetchUserAndSaveInDatabase` like below:

```
1. suspend fun fetchUserAndSaveInDatabase() {  
2.     // fetch user from network  
3.     // save user in database  
4.     // and do not return anything  
5. }
```

Now, we can use the launch like below:

```
1. GlobalScope.launch(Dispatchers.Main) {  
2.     fetchUserAndSaveInDatabase() // do on IO thread  
3. }
```

As the `fetchUserAndSaveInDatabase` does not return anything, we can use the `launch` to complete that task and then do something on Main Thread.

But when we need the result back, we need to use the `async`.

We have two functions which return User like below:

```
1. suspend fun fetchFirstUser(): User {  
2.     // make network call  
3.     // return user  
4. }  
5.  
6. suspend fun fetchSecondUser(): User {  
7.     // make network call  
8.     // return user  
9. }
```



Now, we can use the `async` like below:

```
1. GlobalScope.launch(Dispatchers.Main) {
2.     val userOne = async(Dispatchers.IO) { fetchFirstUser() }
3.     val userTwo = async(Dispatchers.IO) { fetchSecondUser() }
4.     showUsers(userOne.await(), userTwo.await()) // back on UI thread
5. }
```

Here, it makes both the network call in parallel, wait for the results, and then calls the `showUsers` function.

So, now that, we have understood the difference between the `launch` function and the `async` function.

There is something called `withContext`.

```
1. suspend fun fetchUser(): User {
2.     return GlobalScope.async(Dispatchers.IO) {
3.         // make network call
4.         // return user
5.     }.await()
6. }
```

`withContext` is nothing but another way of writing the `async` where we do not have to write `await()`.

```
1. suspend fun fetchUser(): User {
2.     return withContext(Dispatchers.IO) {
3.         // make network call
4.         // return user
5.     }
6. }
```

But there are many more things that we should know about the `withContext` and the `await`.

Now, let's use `withContext` in our `async` example of `fetchFirstUser` and `fetchSecondUser` in parallel.

```
1. GlobalScope.launch(Dispatchers.Main) {
2.     val userOne = withContext(Dispatchers.IO) { fetchFirstUser() }
3.     val userTwo = withContext(Dispatchers.IO) { fetchSecondUser() }
4.     showUsers(userOne, userTwo) // back on UI thread
5. }
```

When we use `withContext`, it will run in series instead of parallel. That is a major difference.

The thumb-rules:

- Use `withContext` when you do not need the parallel execution.
- Use `async` only when you need the parallel execution.
- Both `withContext` and `async` can be used to get the result which is not possible with the `launch`.
- Use `withContext` to return the result of a single task.
- Use `async` for results from multiple tasks that run in parallel.



Scopes in Kotlin Coroutines

Scopes in Kotlin Coroutines are very useful because we need to cancel the background tasks as soon as the activity is destroyed.

Assuming that our activity is the scope, the background task should get cancelled as soon as the activity is destroyed.

In activity, we need to implement `CoroutineScope`.

```
1. class MainActivity : AppCompatActivity(), CoroutineScope {
2.
3.     override val coroutineContext: CoroutineContext
4.         get() = Dispatchers.Main + job
5.     private lateinit var job: Job
6. }
```

In the `onCreate` and `onDestroy` function.

```
1. override fun onCreate(savedInstanceState: Bundle?) {
2.     super.onCreate(savedInstanceState)
3.     job = Job() // create the Job
4. }
5. override fun onDestroy() {
6.     job.cancel() // cancel the Job
7.     super.onDestroy()
8. }
```

Now, just use the launch like below:

```
1. launch {
2.     val userOne = async(Dispatchers.IO) { fetchFirstUser() }
3.     val userTwo = async(Dispatchers.IO) { fetchSecondUser() }
4.     showUsers(userOne.await(), userTwo.await())
5. }
```

As soon as the activity is destroyed, the task will get canceled if it is running because we have defined the scope.

When we need the global scope, which is our application scope, not the activity scope, we can use the `GlobalScope` as below:

```
1. GlobalScope.launch(Dispatchers.Main) {
2.     val userOne = async(Dispatchers.IO) { fetchFirstUser() }
3.     val userTwo = async(Dispatchers.IO) { fetchSecondUser() }
4. }
```



Here, even if the activity gets destroyed, the `fetchUser` functions will continue as we have used the `GlobalScope`.

This is how the Scopes in Kotlin Coroutines are very useful.

Exception Handling in Kotlin Coroutines

When using launch

One way is to try-catch block:

```
1. GlobalScope.launch(Dispatchers.Main) {
2.     try {
3.         fetchUserAndSaveInDatabase() // do on IO thread and back to UI Thread
4.     } catch (exception: Exception) {
5.         Log.d(TAG, "$exception handled !")
6.     }
7. }
```

Another way is to use a handler:

For this we need to create an exception handler like below:

```
1. val handler = CoroutineExceptionHandler { _, exception ->
2.     Log.d(TAG, "$exception handled !")
3. }
```

Then, we can attach the handler like below:

```
1. GlobalScope.launch(Dispatchers.Main + handler) {
2.     fetchUserAndSaveInDatabase() // do on IO thread and back to UI Thread
3. }
```

If there is an exception in `fetchUserAndSaveInDatabase`, it will be handled by the handler which we have attached.

When using in the activity scope, we can attach the exception in our `coroutineContext` as below:

```
1. class MainActivity : AppCompatActivity(), CoroutineScope {
2.
3.     override val coroutineContext: CoroutineContext
4.         get() = Dispatchers.Main + job + handler
5.
6.     private lateinit var job: Job
7.
8. }
```

And use like below:

```
1. launch {
2.     fetchUserAndSaveInDatabase()
3. }
```



When using async

When using async, we need to use the try-catch block to handle the exception like below:

```
1. val deferredUser = GlobalScope.async {
2.     fetchUser()
3. }
4. try {
5.     val user = deferredUser.await()
6. } catch (exception: Exception) {
7.     Log.d(TAG, "$exception handled !")
8. }
```

Example:

Suppose, we have two network calls like below:

- `getUser ()`
- `getMoreUsers ()`

And, we are making the network calls in series like below:

```
1. launch {
2.     try {
3.         val users = getUsers()
4.         val moreUsers = getMoreUsers()
5.     } catch (exception: Exception) {
6.         Log.d(TAG, "$exception handled !")
7.     }
8. }
```

If one of the network calls fails, it will directly go to the `catch` block.

But suppose, we want to return an empty list or the network call which has failed and continue with the response from the other network call. We can add the `try-catch` block to the individual network call like below:

```
1. launch {
2.     try {
3.         val users = try {
4.             getUsers()
5.         } catch (e: Exception) {
6.             emptyList<User>()
7.         }
8.         val moreUsers = try {
9.             getMoreUsers()
10.        } catch (e: Exception) {
11.            emptyList<User>()
12.        }
13.    } catch (exception: Exception) {
```



```
14.     Log.d(TAG, "$exception handled !")
15. }
```

This way, if any error comes, it will continue with the empty list.

Now, what if we want to make the network calls in parallel. We can write the code below using `async`.

```
1.  launch {
2.    try {
3.      val usersDeferred = async { getUsers() }
4.      val moreUsersDeferred = async { getMoreUsers() }
5.      val users = usersDeferred.await()
6.      val moreUsers = moreUsersDeferred.await()
7.    } catch (exception: Exception) {
8.      Log.d(TAG, "$exception handled !")
9.    }
10. }
```

Here, we will face one problem, if any network error comes, the application will `crash!`, it will NOT go to `catch` block.

To solve this, we will have to use the `coroutineScope` like below:

```
1.  launch {
2.    try {
3.      coroutineScope {
4.        val usersDeferred = async { getUsers() }
5.        val moreUsersDeferred = async { getMoreUsers() }
6.        val users = usersDeferred.await()
7.        val moreUsers = moreUsersDeferred.await()
8.      }
9.    } catch (exception: Exception) {
10.      Log.d(TAG, "$exception handled !")
11.    }
12. }
```

Now, if any network error comes, it will go to the `catch` block.

But suppose again, we want to return empty list for the network call which has failed and continue with the response from the other network call. We will have to use the `supervisorScope` and the `try-catch` block to the individual network call like below:



```
1. launch {
2.     try {
3.         supervisorScope {
4.             val usersDeferred = async { getUsers() }
5.             val moreUsersDeferred = async { getMoreUsers() }
6.             val users = try {
7.                 usersDeferred.await()
8.             } catch (e: Exception) {
9.                 emptyList<User>()
10.            }
11.            val moreUsers = try {
12.                moreUsersDeferred.await()
13.            } catch (e: Exception) {
14.                emptyList<User>()
15.            }
16.        }
17.    } catch (exception: Exception) {
18.        Log.d(TAG, "$exception handled !")
19.    }
20. }
```

Again, this way, if any error comes, it will continue with the empty list.

This is how `supervisorScope` helps.

NOTE:

- While NOT using `async`, we can go ahead with the `try-catch` or the `CoroutineExceptionHandler` and achieve anything based on our use-cases.
- While using `async`, in addition to `try-catch`, we have two options: `coroutineScope` and `supervisorScope`.
- With `async`, use `supervisorScope` with the individual `try-catch` for each task in addition to help top-level `try-catch`, when you want to continue with other tasks if one or some of them have failed.
- With `async`, use `coroutineScope` with the top-level `try-catch`, when you do NOT want to continue with other tasks if any of them have failed.
- The major difference is that a `coroutineScope` will cancel whenever any of its children fail. If we want to continue with the other tasks even when one fails, we go with the `supervisorScope`. A `supervisorScope` won't cancel other children when one of them fails.

This is how the exception handling can be done in the Kotlin Coroutines.



Kotlin Interview Questions

First open the build.gradle file. Then go to:

1. Tools
2. Kotlin
3. Configure Kotlin in project
4. Choose all modules and click ok

Then, go to all file

1. Select code
2. Convert Java file to Kotlin file

Then we need to install another Kotlin plugin in build.gradle file,

```
apply plugin: 'kotlin-android-extensions'
```

To make the Button color same as the Action bar

The code will be:

```
style="@style/Widget.AppCompat.Button.Colored"
```

What does the arrow ('->') operator do in Kotlin?

The -> is a **separator**. It is a special symbol used to separate code with different purposes.

It can be used to:

- Separate the parameters and body of a lambda expression

```
val sum = { x: Int, y: Int -> x + y }
```

- Separate the parameters and return type declaration in a function type

```
(R, T) -> R
```

- Separate the condition and body of a when expression branch

```
when (x) {  
    0, 1 -> print("x == 0 or x == 1")  
    else -> print("otherwise")  
}
```



How and when do you use Nullable in Kotlin?

Kotlin apps distinguish between elements that can be null and those that can't. This special feature in Kotlin helps you avoid errors from null references. These errors are common in other languages!

Here's an example of a variable that is not nullable:

```
var myVariable: String = "variable is not nullable"

myVariable = null //Compiler error. You can't do this!

var myVariable: String? = "This is my nullable variable"

myVariable = null
```

Notice the '?' symbol. That's how you identify an item as nullable.

How do you do Type Checking in Kotlin?

Kotlin uses the operator `is`, and its opposite, `!is`, for type checking. These operators return a boolean value. You run a type check to validate the type of the incoming value before performing an operation on it.

Example:

```
if (name is String) {

    // do my operation if name is a String

}

if (name !is String) {

    // do my operation if name is not a String

}
```

This prevents errors that can occur when you attempt to perform an operation on a value with an unexpected type.

How do you declare switch statement in Kotlin?

In Kotlin, you use the `when` keyword where you would use `switch` in some other languages.

Example:

```
when (lotteryNumber) {

    329013 -> print ("You won!!!")

}
```



```
else -> print ("Sorry you didn't win the lottery, try again!")  
}
```

What is Lambda Expression?

A lambda is a function that is not declared, but is passed immediately as an expression.

Lambdas expressions are anonymous functions that can be treated as values i.e., we can pass the lambdas expressions as arguments to a function return them, or do any other thing we could do with a normal object.

```
val amountOfCandies = {  
  
    candiesStored: Int, candiesBought: Int -> candiesStored + candiesBought  
  
}
```

It is simple, concise and no return statement needed. Best of all, it can be passed around as a parameter to other functions.

When is the keyword 'it' used in Lambda Expression?

When you have a lambda expression with a single parameter, you can use the keyword 'it' to refer to that parameter.

```
val isPositiveNumber = {  
  
    it > 0  
  
}
```

What is a Class Extension?

Extensions exist in many programming languages. As their name states, they “**extend**” the functionality of a class.

Example:

Take a look at this example of an extension on [String](#). This one validates an email address:

```
import java.util.regex.Pattern //1  
  
//2 fun String.isEmailAddressValid(): Boolean {  
  
    // 3 val expression = "^([\\w.-]+@[\\w\\-]+\\.\\w+)[A-Z]{2,8}$"  
  
    val pattern = Pattern.compile(expression, Pattern.CASE_INSENSITIVE)  
  
    val matcher = pattern.matcher(this)  
  
    return matcher.matches() //4  
}
```




```
}
```

Here's what you're doing in this code:

1. You import the Java library for regex patterns.
2. You declare the function expression. Notice that the name of the class it's extending, `String`, precedes the name of the function `isEmailAddressValid()`.
3. Here you create three constants:
 - The first constant contains a regular expression for emails.
 - The second constant creates a pattern matcher. Notice that it is not case sensitive.
 - The third constant makes it a pattern matcher.
4. Finally, you run `matches` on it. This function returns a boolean value that tells you whether the data entered is an email or not.

What are Ranges in Kotlin?

The term range is very similar to its equivalent in math. It represents a group of numbers that goes from a lower to a higher bound (or the other way around).

Example:

```
i in 1..10 // equivalent of 1 <= i && i <= 10
```

Notice the range has two dots between the numbers. To iterate down instead up, use the keyword `downTo` instead of the two dots:

```
i in 10 downTo 1
```

What is Polymorphism?

Polymorphism means having multiple forms. In OOP, the most general definition of polymorphism is the ability to define one interface or class with many implementations. In Android, there are two types of polymorphism, dynamic and static.

In **Dynamic Polymorphism**, the type of object on which method is being invoked is not known as compile time but will be decided at run time. For example, a parent class `Animal` might have two subclasses, `Dog` and `Cat`, both inheriting from `Animal`.

Static Polymorphism refers to constructors. You can have multiple constructors with the same name, but they can receive different parameters. The intended constructor will be executed depending on the arguments passed.

For example, a school system uses an app to maintain student information. You write a constructor to create a student object:

```
fun createStudent(name: String, nickname: String) {  
  
    addStudentToDatabase(name: String, nickname: String)  
  
}
```



Some students, but not all of them, play sports. You write another constructor to create an object for a student who plays sports:

```
fun createStudent(name: String, nickname: String, sport: Sport) {  
  
    addStudentToDatabase(name: String, nickname: String)  
  
    assignSportToStudent(sport: Sport)  
  
}
```

This second constructor will also create a student object, just as the first one does. But afterward, it will also assign the desired sport. The compiler determines which constructor to execute based on the parameters it receives.

What are Data Classes?

A **Data class** in Kotlin is a special type of class that exists only to contain data. You declare a data class by providing the name of the class and its attributes, as in this example:

```
data class Cat (val name: String, val age: Int, val furColor: Color)
```

This special construct in Kotlin provides standard functionality with minimal coding. Behind the scenes, Kotlin generates getters and setters and an initializer for you. Then you can get on with creating and modifying instances of the object!

What is Encapsulation?

Encapsulation is a primary concept of Object-Oriented Programming. It refers to the bundling of an object with its attributes, state and operations. This simplifies code and protects data integrity.

What are Visibility Modifiers?

Data Encapsulation allows you to control how objects and their behaviors are exposed outside the object. This is called visibility. In Kotlin, there are four levels of visibility, called visibility modifiers:

public

internal

protected

private

The default is public.

How does Kotlin work on Android?

Just like Java, the Kotlin code is also compiled into the Java bytecode and is executed at runtime by the Java Virtual Machine that is JVM. When a Kotlin file named `Main.kt` is compiled then it will eventually turn into a class and then the bytecode of the class will be generated. The name of the bytecode file will be `MainKt.class` and this file will be executed by the JVM.

Why should we use Kotlin?



- Kotlin is concise
- Kotlin is null-safe
- Kotlin is interoperable

What is the difference between the variable declaration with var and val?

If you want to declare some mutable (changeable) variable, then you can use `var`. For the immutable variable, use `val` i.e., `val` variables can't be changed once assigned.

What is the difference between the variable declaration with val and const?

Both the variables that are declared with `val` and `const` are immutable in nature. But the value of the `const` variable must be known at the compile-time whereas the value of the `val` variable can be assigned at runtime also.

How to ensure null safety in Kotlin?

One of the major advantages of using Kotlin is null safety. In Java, if you access some null variable then you will get a `NullPointerException`. So, the following code in Kotlin will produce a compile-time error:

```
var name: String = "MindOrks"

name = null //error
```

So, to assign null values to a variable, you need to declare the `name` variable as nullable string and then during the access of this variable, you need to use a safe call operator i.e., `?.`

```
var name: String? = "MindOrks"

print(name?.length) // ok

name = null // ok
```

What is the difference between safe calls (?.) and null check (!)?

Safe call operator i.e., `?.` is used to check if the value of the variable is null or not. If it is null then null will be returned otherwise it will return the desired value.

```
var name: String? = "MindOrks"

println(name?.length) // 8

name = null

println(name?.length) // null
```

If you want to throw `NullPointerException` when the value of the variable is null, then you can use the null check or `!!` operator.

```
var name: String? = "MindOrks"
```



```
println(name?.length) // 8  
  
name = null  
  
println(name!!.length) // KotlinNullPointerException
```

Do we have a ternary operator in Kotlin just like Java?

No, we don't have a ternary operator in Kotlin but you can use the functionality of ternary operator by using if-else or Elvis operator.

What is Elvis operator in Kotlin?

In Kotlin, you can assign null values to a variable by using the null safety property. To check if a value is having null value then you can use if-else or can use the Elvis operator i.e., `?:`.

For example:

```
var name:String? = "Mindorks"  
  
val nameLength = name?.length ?: -1  
  
println(nameLength)
```

The Elvis operator (`?:`) used above will return the length of name if the value is not null otherwise if the value is null, then it will return `-1`.

How to convert a Kotlin source file to a Java source file?

Steps to convert your Kotlin source file to Java source file:

- Open your Kotlin project in the IntelliJ IDEA / Android Studio.
- Then navigate to Tools > Kotlin > Show Kotlin Bytecode.
- Now click on the Decompile button to get your Java code from the bytecode.

What is the use of @JvmStatic, @JvmOverloads, and @JvmField in Kotlin?

`@JvmStatic`:

This annotation is used to tell the compiler that the method is a static method and can be used in Java code.

`@JvmOverloads`:

To use the default values passed as an argument in Kotlin code from the Java code, we need to use the `@JvmOverloads` annotation.

`@JvmField`:

To access the fields of a Kotlin class from Java code without using getters and setters, we need to use the `@JvmField` in the Kotlin code.



What is a data class in Kotlin?

Data classes are those classes which are made just to store some data. In Kotlin, it is marked as data. The following is an example of the same:

When we mark a class as a data class, you don't have to implement or create the following functions like we do in Java: `hashCode ()`, `equals ()`, `toString ()`, `copy ()`. The compiler automatically creates these internally, so it also leads to clean code. Although, there are few other requirements that data class need to fulfill.

Can we use primitive types such as int, double, float in Kotlin?

In Kotlin, we can't use primitive types directly. We can use classes like `Int`, `Double` etc. as an object wrapper for primitives. But the compiled byte code has these primitive types.

What is String Interpolation in Kotlin?

If you want to use some variable or perform some operation inside a string then String Interpolation can be used. You can use the `$` sign to use some variable in the string or can perform some operation in between `{ }` sign.

```
var name = "MindOrks"

print("Hello! I am learning from $name")
```

What do you mean by Destructuring in Kotlin?

Destructuring is a convenient way of extracting multiple values from data stored in (possible nested) objects and Arrays. It can be used in locations that receive data (such as the left-hand side of an assignment). Sometimes it is convenient to destructure an object into a number of variables.

For example:

```
val (name, age) = developer
```

Now, we can use name and age independently like below:

```
println(name)

println(age)
```

When to use the `lateinit` keyword in Kotlin?

`lateinit` is late initialization.

Normally, properties declared as having a non-null type must be initialized in the constructor. However, fairly often this is not convenient.



For example, properties can be initialized through dependency injection, or in the setup method of a unit test. In this case, you cannot supply a non-null initializer in the constructor, but you still want to avoid null checks when referencing the property inside the body of a class. To handle this case, you can mark the property with the `lateinit` modifier.

How to check if a `lateinit` variable has been initialized or not?

You can check if the `lateinit` variable has been initialized or not before using it with the help of `isInitialized` method. This method will return true if the `lateinit` property has been initialized otherwise it will return false.

For example:

```
class Person {  
  
    lateinit var name: String  
  
    fun initializeName() {  
  
        println(this::name.isInitialized)  
  
        name = "MindOrks" // initializing name  
  
        println(this::name.isInitialized)  
  
    }  
}  
  
fun main(args: Array<String>) {  
  
    Person().initializeName()  
  
}
```

What is the difference between `lateinit` and `lazy` in Kotlin?

Lazy can only be used for `val` properties, whereas **lateinit** can only be applied to `var` because it can't be compiled to a final field, thus no immutability can be guaranteed.

If you want your property to be initialized from outside in a way probably unknown beforehand, use `lateinit`.

Is there any difference between `==` operator and `===` operator?

Yes, the `==` operator is used to compare the values stored in variables and the `===` operator is used to check if the reference of the variables are equal or not. But in the case of primitive types, the `===` operator also checks for the value and not reference.

```
// primitive example
```



```
val int1 = 10

val int2 = 10

println(int1 == int2) // true

println(int1 === int2) // true

// wrapper example

val num1 = Integer(10)

val num2 = Integer(10)

println(num1 == num2) // true

println(num1 === num2) //false
```

What is the forEach in Kotlin?

In Kotlin, to use the functionality of a for-each loop just like in Java, we use a `forEach` function. The following is an example of the same:

```
var listOfMindOrks = listOf("mindorks.com", "blog.mindorks.com", "afteracademy.com")

listOfMindOrks.forEach {

    Log.d(TAG,it)

}
```

What are companion objects in Kotlin?

In Kotlin, if you want to write a function or any member of the class that can be called without having the instance of the class then you can write the same as a member of a companion object inside the class.

To create a **companion** object, you need to add the companion keyword in front of the object declaration.

The following is an example of a companion object in Kotlin:

```
class ToBeCalled {

    companion object Test {

        fun callMe() = println("You are calling me :)")

    }

}

fun main(args: Array<String>) {
```



```
ToBeCalled.callMe()  
}
```

What is the equivalent of Java static methods in Kotlin?

To achieve the functionality similar to Java static methods in Kotlin, we can use:

- Companion object
- Package-level function
- Object

What is the difference between FlatMap and Map in Kotlin?

- FlatMap is used to combine all the items of lists into one list.
- Map is used to transform a list based on certain conditions.

What is the difference between List and Array types in Kotlin?

If you have a list of data that is having a fixed size, then you can use an Array. But if the size of the list can vary, then we have to use a mutable list.

Can we use the new keyword to instantiate a class object in Kotlin?

No. in Kotlin we don't have to use the `new` keyword to instantiate a class object. To instantiate a class object, simply we use:

```
var varName = ClassName()
```

What are visibility modifiers in Kotlin?

A visibility modifier or access specifier or access modifier is a concept that is used to define the scope of something in a programming language. In Kotlin, we have four visibility modifiers:

Private: Visible inside that particular class or file containing the declaration.

Protected: Visible inside that particular class or file and also in the subclass of that particular class where it is declared.

Internal: Visible everywhere in that particular module.

Public: Visible to everyone.

Note: By default, the visibility modifier in Kotlin is **Public**.

How to create a Singleton class in Kotlin?

A **Singleton** class is a class that is defined in such a way that only one instance of the class can be created and is used where we need only one instance of the class like logging, database connections etc.

To create a Singleton class in Kotlin, you need to use the object keyword.

```
object AnySingletonClassName
```

What are init blocks in Kotlin?



`init` blocks are initializer blocks that are executed just after execution of the primary constructor. A class file can have one or more `init` blocks that will be executed in series. If you want to perform some operation in the primary constructor, then it is possible in Kotlin, for that, you need to use the `init` block.

What are the types of constructors in Kotlin?

Primary Constructor:

These constructors are defined in the class header and you can't perform some operation in it, unlike Java's constructor.

Secondary Constructor:

These constructors are declared inside the class body by using the constructor keyword. You must call the primary constructor explicitly. Also, the property of the class can't be declared inside the secondary constructor.

There can be more than one secondary constructor in Kotlin.

Is there any relationship between primary and secondary constructors?

Yes, when using a secondary constructor, you need to call the primary constructor explicitly.

What is the default type of argument used in a constructor?

By default, the type of arguments of a constructor is `val`. But you can change it to `var` explicitly.

What are Coroutines in Kotlin?

A framework to manage concurrency in a more performant and simple way with its lightweight thread which is written on top of the actual threading framework to get the most out of it by taking the advantage of cooperative nature of functions.

What is suspend function in Kotlin Coroutines?

Suspend function is the building block of the Coroutines in Kotlin. Suspend function is a function that could be started, paused and resume. To use a suspend function, we need to use the `suspend` keyword in our normal function definition.

What is the difference between Launch and Async in Kotlin Coroutines?

The difference is that the `launch { }` does not return anything and the `async { }` returns an instance of `Deferred <T>`, which has an `await ()` function that returns the result of the coroutines like we have future in Java in which we do `future.get ()` to get the result.

In other words:

- `Launch`: fire and forget
- `Async`: perform a task and return a result

Example: Suppose we have a function like this,

```
fun fetchUserAndSaveInDatabase () {  
  
    // fetch user from network
```



```
// save user in database

// and do not return anything

}
```

Now, we can use the `launch` like below:

```
GlobalScope.launch(Dispatchers.Main) {

    fetchUserAndSaveInDatabase () // do on IO thread

}
```

As the function doesn't return anything, we can use `launch` to complete that task and then do something on main thread.

Now, suppose we need the result back, we need to use the `async`.

For `example`: we have two functions like below which return user, the first one is `fetch first user` and we have another function is `fetch second user`. So, both will be making the network call and returning the user.

Here I have not made the function suspend, we can make it based on the requirement.

```
fun fetchFirstUser (): User {

    // make network call

    // return user}

fun fetchSecondUser (): User {

    // make network call

    // return user}
```

Now, we can use the `async` like below:

```
GlobalScope.launch(Dispatchers.Main) {

    val userOne = async(Dispatchers.IO){fetchFirstUser()}

    val userTwo = async(Dispatchers.IO){fetchSeconduser()}

    showUsers (userOne.await(), userTwo.await()) // back on UI thread

}
```



We can call both the function and user 1 and user 2 are here the deferred one so it means, it will have the function of await, we can call the await on those. So, we can call like this and then we can show into the UI back on the UI thread. So, here it will make both the network call in parallel and waits for the results and then we will call the show user function.

How to choose between a switch and when in Kotlin?

Whenever we want to handle many if-else conditions, then we generally use switch-case statements. But Kotlin provides a more concise option i.e., in Kotlin, we can use when in place of the switch. And, when can be used as:

- Expression
- Arbitrary condition expression
- Without argument
- With two or more choices

For example:

```
when(number) {  
  
    1 -> println("One")  
  
    2, 3 -> println("Two or Three")  
  
    4 -> println("Four")  
  
    else -> println("Number is not between 1 and 4")  
  
}
```

What is the open keyword in Kotlin used for?

By default, the classes and functions are final in Kotlin. So, you can't inherit the class or override the functions. To do so, you need to use open keyword before the class and function.

What are Higher-Order functions in Kotlin?

A higher-order function is a function that takes functions as parameters or returns a function. For example, A function can take functions as parameters.

```
fun passMeFunction(abc: () -> Unit) {  
  
    // I can take function  
  
    // do something here  
  
    // execute the function  
  
    abc()  
}
```



```
}
```

For example, A function can return another function.

```
fun add (a: Int, b: Int): Int {  
  
    return a + b  
  
}
```

And, we have a function `returnMeAddFunction` which takes zero parameters and returns a function of the type `((Int, Int) -> Int)`.

```
fun returnMeAddFunction(): ((Int, Int) -> Int) {  
  
    // can do something and return function as well  
  
    // returning function  
  
    return ::add  
  
}
```

And to call the above function, we can do:

```
val add = returnMeAddFunction()  
  
val result = add(2, 2)
```

What are extension functions in Kotlin?

Extension functions are like extensive properties attached to any class in Kotlin. By using extension functions, you can add some methods or functionalities to an existing class even without inheriting the class. For example: Let's say, we have views where we need to play with the visibility of the views. So, we can create an extension function for views like,

```
fun View.show() {  
  
    this.visibility = View.VISIBLE  
  
}  
  
fun View.hide() {
```



```
this.visibility = View.GONE  
}
```

And to use it we use, like,

```
toolbar.hide()
```

What is an infix function in Kotlin?

An infix function is used to call the function without using any bracket or parenthesis. You need to use the infix keyword to use the infix function.

```
class Operations {  
  
    var x = 10;  
  
    infix fun minus(num: Int) {  
  
        this.x = this.x - num  
  
    }  
}  
  
fun main() {  
  
    val opr = Operations()  
  
    opr minus 8  
  
    print(opr.x)  
}
```

What is an inline function in Kotlin?

Inline function instruct compiler to insert complete body of the function wherever that function got used in the code. To use an Inline function, all you need to do is just add an inline keyword at the beginning of the function declaration.

What is noinline in Kotlin?

While using an inline function and want to pass some lambda function and not all lambda function as inline, then you can explicitly tell the compiler which lambda it shouldn't inline.

```
inline fun doSomethingElse(abc: () -> Unit, noinline xyz: () -> Unit) {
```



```
abc()

xyz()

}
```

What are Reified types in Kotlin?

When you are using the concept of Generics to pass some class as a parameter to some function and you need to access the type of that class, then you need to use the reified keyword in Kotlin.

```
inline fun <reified T> genericsExample(value: T) {

    println(value)

    println("Type of T: ${T::class.java}")

}

fun main() {

    genericsExample<String>("Learning Generics!")

    genericsExample<Int>(100)

}
```

What is the operator overloading in Kotlin?

In Kotlin, we can use the same operator to perform various tasks and this is known as **operator overloading**. To do so, we need to provide a member function or an extension function with a fixed name and operator keyword before the function name because normally also, when we are using some operator then under the hood some functions gets called. For example, if you are writing `num1 + num2`, then it gets converted to `num1.plus(num2)`.

For example:

```
1. fun main() {
2.   val bluePen = Pen(inkColor = "Blue")
3.   bluePen.showInkColor()
4.
5.   val blackPen = Pen(inkColor = "Black")
6.   blackPen.showInkColor()
7.
8.   val blueBlackPen = bluePen + blackPen
9.   blueBlackPen.showInkColor()
10. }
11.
12. operator fun Pen.plus(otherPen: Pen): Pen {
13.   val ink = "$inkColor, ${otherPen.inkColor}"
```



```
14. return Pen(inkColor = ink)
15. }
16.
17. data class Pen(val inkColor:String){
18.     fun showInkColor(){ println(inkColor)}
19. }
```

What are pair and triple in Kotlin?

Pair and Triples are used to return two and three values respectively from a function and the returned values can be of the same data type or different.

```
val pair = Pair("My Age: ", 25)

print(pair.first + pair.second)
```

What are labels in Kotlin?

Any expression written in Kotlin is called a **label**. For example, if we are having a for-loop in our Kotlin code then we can name that for-loop expression as a label and will use the label name for the for-loop.

We can create a label by using an identifier followed by the @ sign. For example, `name@`, `loop@`, `xyz@` etc.

The following is an example of a label:

```
loop@ for (i in 1..10) {

    // some code goes here

}
```

The name of the above for-loop is `loop`.

What are the benefits of using a Sealed class over Enum?

Sealed classes give us the flexibility of having **different types of subclasses and also containing the state**. The important point to be noted here is the subclasses that are extending the Sealed classes should be either nested classes of the Sealed class or should be declared in the same file as that of the Sealed class.

What are scopes in Kotlin Coroutines?

Scopes in Kotlin Coroutines are very useful because we need to cancel the background task as soon as the activity is destroyed.

Assuming that our activity is the scope, the background task should get cancelled as soon as the activity is destroyed.

In the activity, we need to implement Coroutines Scope.

```
1. class MainActivity : AppCompatActivity(), CoroutineScope {
2.     override val coroutineContext: CoroutineContext
```



```
3.     get() = Dispatchers.Main + job
4.     private lateinit var job: Job
5. }
```

In the `onCreate` and `onDestroy` function.

```
1. override fun onCreate(savedInstanceState: Bundle?) {
2.     super.onCreate(savedInstanceState)
3.     job = Job() // create the Job
4. }
5.
6. override fun onDestroy() {
7.     job.cancel() // cancel the Job
8.     super.onDestroy()
9. }
10.
```

Now, just use the launch like below:

```
1. launch {
2.     val userOne = async(Dispatchers.IO) { fetchFirstUser() }
3.     val userTwo = async(Dispatchers.IO) { fetchSecondUser() }
4.     showUsers(userOne.await(), userTwo.await())
5. }
6.
```

As soon as the activity is destroyed, the task will get canceled if it running because we have defined the scope.

When we need the global scope, which is our application scope, not the activity scope, we can use the `GlobalScope` as below:

```
1. GlobalScope.launch(Dispatchers.Main) {
2.     val userOne = async(Dispatchers.IO) { fetchFirstUser() }
3.     val userTwo = async(Dispatchers.IO) { fetchSecondUser() }
4. }
5.
```

Here, even if the activity gets destroyed, the `fetchUser` functions will continue running as we have used the `GlobalScope`.

How Exception Handling is done in Kotlin Coroutines?

When using launch

One way is to use try-catch block:



```
1. GlobalScope.launch(Dispatchers.Main) {
2.     try {
3.         fetchUserAndSaveInDatabase() // do on IO thread and back to UI Thread
4.     } catch (exception: Exception) {
5.         Log.d(TAG, "$exception handled !")
6.     }
7. }
```

Another way is to use a handler:

For this we need to create an exception handler like below:

```
1. val handler = CoroutineExceptionHandler { _, exception ->
2.     Log.d(TAG, "$exception handled !")
3. }
```

Then, we can attach the handler like below:

```
1. GlobalScope.launch(Dispatchers.Main + handler) {
2.     fetchUserAndSaveInDatabase() // do on IO thread and back to UI Thread
3. }
```

If there is an exception in `fetchUserAndSaveInDatabase`, it will be handled by the handler which we have attached.

When using in the activity scope, we can attach the exception in our `coroutineContext` as below:

```
1. class MainActivity : AppCompatActivity(), CoroutineScope {
2.
3.     override val coroutineContext: CoroutineContext
4.         get() = Dispatchers.Main + job + handler
5.
6.     private lateinit var job: Job
7.
8. }
```

And use like below:

```
1. launch {
2.     fetchUserAndSaveInDatabase()
3. }
```

When using async

When using `async`, we need to use the try-catch block to handle the exception like below.

```
1. val deferredUser = GlobalScope.async {
2.     fetchUser()
3. }
```



```
4. try {  
5.     val user = deferredUser.await()  
6. } catch (exception: Exception) {  
7.     Log.d(TAG, "$exception handled !")  
8. }  
9.
```

Suppose, we have two network calls like below:

- `getUser ()`
- `getMoreUsers ()`

And, we are making the network calls in series like below:

```
1. launch {  
2.     try {  
3.         val users = getUsers()  
4.         val moreUsers = getMoreUsers()  
5.     } catch (exception: Exception) {  
6.         Log.d(TAG, "$exception handled !")  
7.     }  
8. }  
9.
```

If one of the network call fail, it will directly go to the `catch` block.

But suppose, we want to return an empty list for the network call which has failed and continue with the response from the other network call. We can add the `try-catch` block to the individual network call like below:

```
1. launch {  
2.     try {  
3.         val users = try {  
4.             getUsers()  
5.         } catch (e: Exception) {  
6.             emptyList<User>()  
7.         }  
8.         val moreUsers = try {  
9.             getMoreUsers()  
10.        } catch (e: Exception) {  
11.            emptyList<User>()  
12.        }  
13.    } catch (exception: Exception) {  
14.        Log.d(TAG, "$exception handled !")  
15.    }  
16. }
```

This way, if any error comes, it will continue with the empty list.

Now, what if we want to make the network calls in parallel. We can write the code like below using `async`.



```
1. launch {
2.     try {
3.         val usersDeferred = async { getUsers() }
4.         val moreUsersDeferred = async { getMoreUsers() }
5.         val users = usersDeferred.await()
6.         val moreUsers = moreUsersDeferred.await()
7.     } catch (exception: Exception) {
8.         Log.d(TAG, "$exception handled !")
9.     }
10. }
```

Here, we will face one problem, if any network error comes, the application will crash!, it will NOT go to the `catch` block.

To solve this, we have to use the `coroutineScope` like below:

```
1. launch {
2.     try {
3.         coroutineScope {
4.             val usersDeferred = async { getUsers() }
5.             val moreUsersDeferred = async { getMoreUsers() }
6.             val users = usersDeferred.await()
7.             val moreUsers = moreUsersDeferred.await()
8.         }
9.     } catch (exception: Exception) {
10.        Log.d(TAG, "$exception handled !")
11.    }
12. }
```

Now, if any network error comes, it will go to the `catch` block.

But suppose again, we want to return an empty list for the network call which has failed and continue with the response from the other network call. We will have to use the `supervisorScope` and add the `try-catch` block to the individual network call like below:

```
1. launch {
2.     try {
3.         supervisorScope {
4.             val usersDeferred = async { getUsers() }
5.             val moreUsersDeferred = async { getMoreUsers() }
6.             val users = try {
7.                 usersDeferred.await()
8.             } catch (e: Exception) {
9.                 emptyList<User>()
10.            }
11.            val moreUsers = try {
12.                moreUsersDeferred.await()
13.            } catch (e: Exception) {
14.                emptyList<User>()
15.            }
16.        }
17.    } catch (exception: Exception) {
18.        Log.d(TAG, "$exception handled !")
19.    }
```



```
19. }  
20. }
```

Again, this way, if any error comes, it will continue with the empty list.

This is how `supervisorScope` helps.

Kotlin Higher-Order Functions & Lambdas

Lambdas Expressions

Lambdas Expressions are essentially anonymous functions that we treat as values – we can, for example, pass them as arguments to functions, return them, or do any other thing we could do with a normal object.

Lambdas Expressions look like below:

```
val square : (Int) -> Int = { value -> value * value }  
  
val nine = square(3)
```

Example 1:

```
val doNothing : (Int) -> Int = { value -> value }
```

This is a lambda expression that does nothing.

Here the `value -> value` is a complete function in itself. It takes an `int` as a parameter and returns a value as an `int`.

In `(Int) -> Int`, `(Int)` represents input `int` as a parameter. `Int` represents return type as an `int`.

So, the `doNothing` is a function in itself that takes a value as an `int` and returns the same name value as an `int`.

Example 2:

```
val add : (Int, Int) -> Int = { a, b -> a + b }
```

This is also a lambda expression that takes two `int` as the parameters, adds them, and returns as an `int`.

`{a, b -> a + b}` is a function in itself that takes two `int` as the parameters, adds them, and returns as an `int`.

In `(Int, Int) -> Int`, `(Int, Int)` represents two `int` as the input parameters. `Int` represents return type as an `int`.

So, the `add` is a function in itself that takes two `int` as the parameters, adds them, and returns as an `int`.

We can call it very similar as we do with the function like below:



```
val result = add(2,3)
```

Now, that we have understood the lambda expressions. Let's move to the Higher-order functions.

Higher-Order Functions

A higher-order function is a function that takes functions as parameters or returns a function.

Generally, it's a function which can take do two things:

- Can take functions as parameters
- Can return a function

Example 1: A function can take functions as parameters.

```
1. fun passMeFunction(abc: () -> Unit) {  
2.     // I can take function  
3.     // do something here  
4.     // execute the function  
5.     abc()  
6. }
```

This takes a function `abc: () -> Unit`.

`abc` is just the name for the parameter. It can be anything. We just need to use this when we execute the function.

`() -> Unit`, this is important. `()` represents that the function takes no parameters. `Unit` represents that the function does not return anything.

So, the `passMeFunction` can take a function that takes zero parameters and does not return anything.

Let try passing that type of function to the `passMeFunction`.

```
1. passMeFunction(  
2.     {  
3.         val user = User()  
4.         user.name = "ABC"  
5.         println("Lambda is awesome")  
6.     }  
7. )
```

Here, `{ }` is a function in itself.

```
1. passMeFunction(  
2.     {  
3.         val user = User()  
4.         user.name = "ABC"
```



```
5.     println("Lambda is awesome")
6.     }
7. )
```

Which takes zero parameters and does not return anything. It just creates a user, set the name, and print something. This means it does not take any parameters and does not return anything.

Let's add `fun abc ()` just for the sake of understanding it. Actually, we do not need to write `fun abc ()` in our code.

```
1. passMeFunction(
2. fun abc(){
3.     val user = User()
4.     user.name = "ABC"
5.     println("Lambda is awesome")
6. }
7. )
```

Now, we can clearly see that we are passing a function to the `passMeFunction`.

As Kotlin provides us a concise way for writing code, it can be changed to the following by removing the `()`.

```
1. passMeFunction {
2.     val user = User()
3.     user.name = "ABC"
4.     println("Lambda is awesome")
5. }
```

Now, that we have understood how to pass a function as a parameter to a function.

Example 2: A function can return a function

Suppose we have a function `add` which takes two parameters and returns a value as in int.

```
1. fun add(a: Int, b: Int): Int {
2.     return a + b
3. }
```

And, we have a function `returnMeAddFunction` which takes zero parameters and returns a function of the type `((Int, Int) -> Int)`.

In `((Int, Int) -> Int)`, `(Int, Int)` means that the function should take two parameters both as the `int`. `Int` means that the function should return value as an `int`.

Now, we can call the `returnMeAddFunction`, get the add function, and call it like below:



1. `val add = returnMeAddFunction()`
- 2.
3. `val result = add(2, 2)`

This is what higher-order functions are.