

# Lesson 2: Control Flow Statements

- Introduction** ..... 2-1
- If Statement** ..... 2-1
- If – Else Statement**..... 2-4
- If Else and Logical Operators**..... 2-6
- When Statement and Expression** ..... 2-8
- For Loops**..... 2-9
- While Loops**..... 2-12
- Do-while Loops** ..... 2-13
- Jump Expressions** ..... 2-15
  - Break Statement..... 2-15
  - Continue Statement ..... 2-16
  - Return Statement ..... 2-16
- Functions**..... 2-17
  - Function Structure..... 2-17
  - Creating a Function ..... 2-18
  - Functions and Variable Scope ..... 2-22

## Introduction

The statements inside your source files are generally executed from top to bottom, in the order that they appear. **Control flow statements**; however, break up the flow of execution by employing decision making, looping, branching, and enabling your program to conditionally execute particular blocks of code. This section describes the decision-making statements (if-then, if-then-else), the looping statements (for, while, do-while), and the branching statements (break, continue, return) supported by the Kotlin programming language.

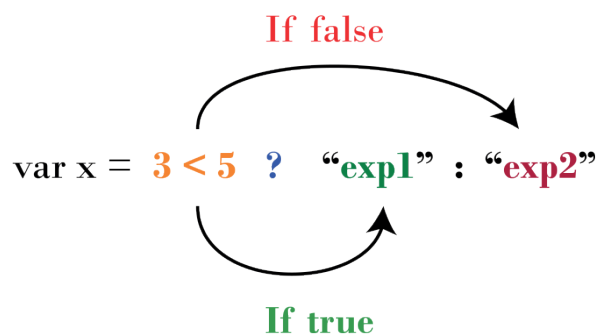
This lesson explains how these control flow statements can control the work flow of Kotlin programs.

The following table includes some conditional operators that help in the control flow of the Kotlin program:

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

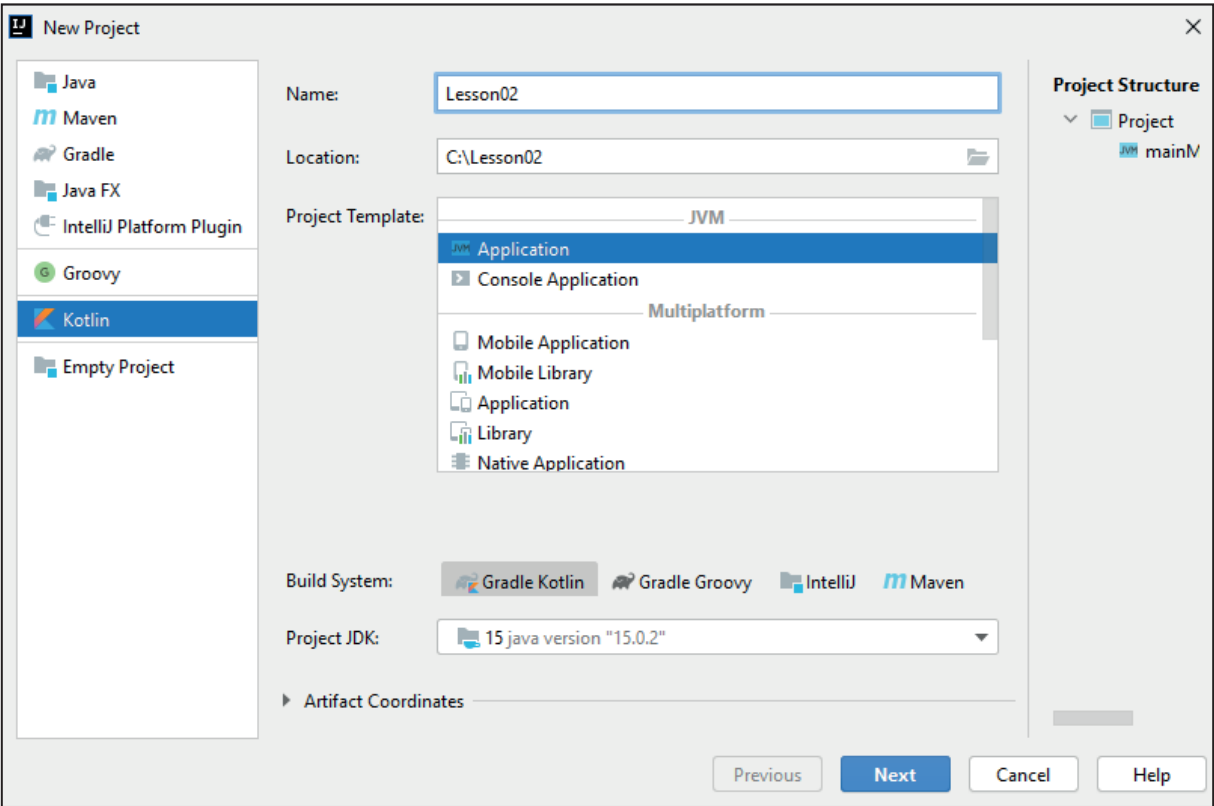
## If Statement

The If statement is a programming conditional statement that, if proved true, executes the second part of the statement. Otherwise, if proved false, the program will skip the execution of the second part and do something else as illustrated in the figure below.

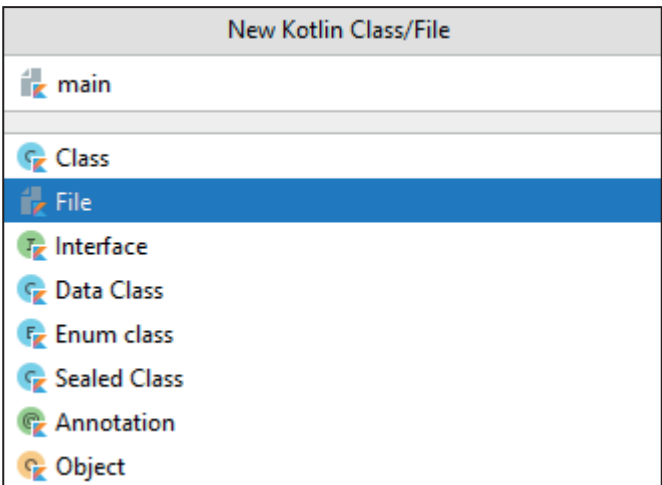


**Example:**

- 1- Open **IntelliJ IDEA**
- 2- Click **File** → **New Project**
- 3- Type **Lesson02** for the project name, and select the project SDK which you have installed in lesson 01 as illustrated in the figure below. Click **Next**, then click **Finish**.



- 4- Go to the Kotlin folder which exists in the following path : Lesson02 → src → main → Kotlin
- Right click the Kotlin folder, select **New** → **Kotlin Class/File**
- Then, as illustrated in the figure below, type: **main** for the Kotlin file name and select **File** from the option below. Press **Enter** (or Return) key.



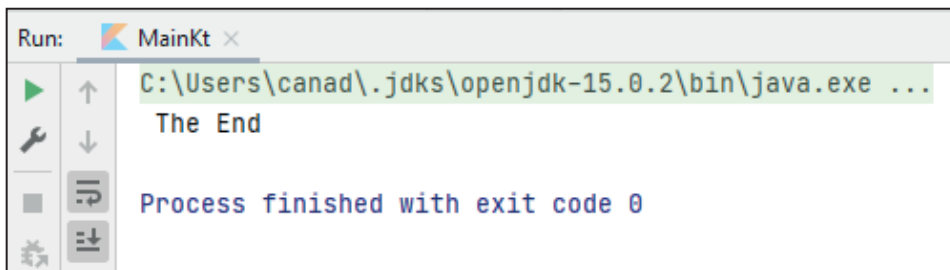
5- In this example, the Kotlin compiler will consider the **if** condition and the statement highlighted in gray color as one block. If **x=10 (true)** the program will print a hello message and if it is **Not (false)** the program will continue to the next action.

```
/**
 * Created by Android ATC.
 */
fun main() {

    var x=10
    if (x>30)
        println(" Hello, I am If statement running now .....")
    println(" The End ")

}
```

When you run the Kotlin program, you will get the following result:

The screenshot shows the 'Run' window of an IDE. At the top, it says 'Run: MainKt x'. Below that, the command 'C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...' is shown. The output of the program is displayed in the center: 'The End'. At the bottom, it says 'Process finished with exit code 0'. On the left side of the window, there are several icons for debugging and running the program.

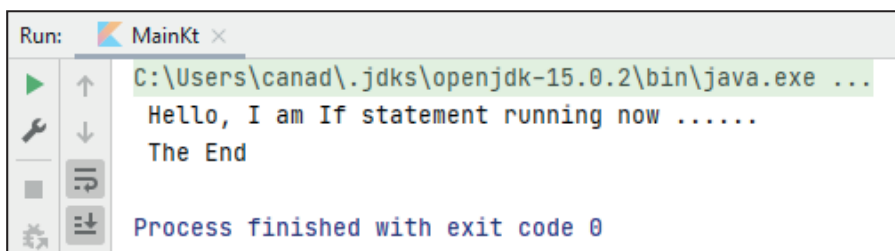
If you change the **x** value to be 50 in this program as follows:

```
/**
 * Created by Android ATC
 */
fun main() {

    var x=50
    if (x>30)
        println(" Hello, I am If statement running now .....")
    println(" The End ")

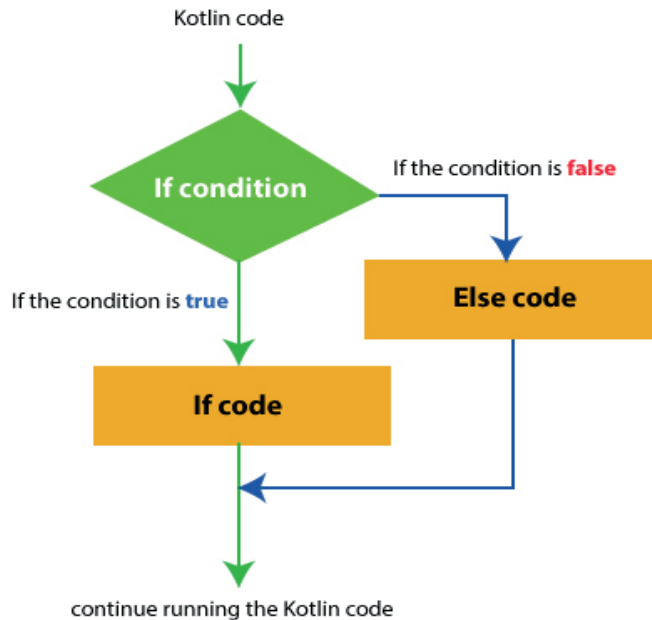
}
```

When you run this Kotlin code, you will get the following result:

The screenshot shows the 'Run' window of an IDE. At the top, it says 'Run: MainKt x'. Below that, the command 'C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...' is shown. The output of the program is displayed in the center: 'Hello, I am If statement running now .....', followed by 'The End'. At the bottom, it says 'Process finished with exit code 0'. On the left side of the window, there are several icons for debugging and running the program.

## If – Else Statement

The **if- else** statement is the basic of all the control flow statements. It tells your program to execute a specific section of a code only if a particular test proves to be true.



The following example gives an idea of how IF-Else statement works in an easy way:

### Example:

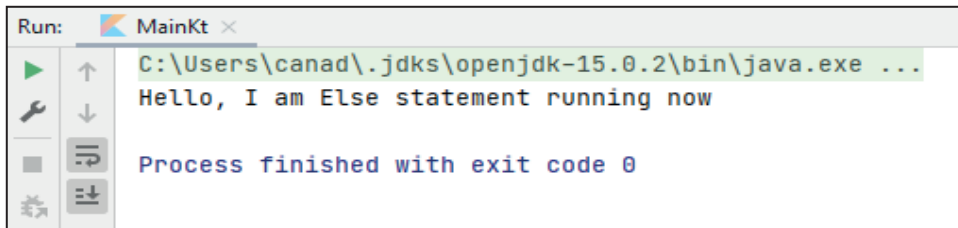
In the example below, when the Kotlin program starts, the value of the **x** variable =10. In the next line, the **If statement** will work using the “greater than” condition. After the Kotlin compiler checks whether this condition is true or not, IntelliJ (Compiler) will directly run the next line of codes. However, if the condition is proved not true, IntelliJ compiler will continue to run the codes under the **Else statement**.

```
/**
 * Created by Android ATC
 */
fun main() {

    var x=10
    if (x>30)
        println("Hello, I am If statement running now")
    else
        println("Hello, I am Else statement running now")

}
```

When you run the Kotlin program, you will get the following result:



## If...Else and Else...If... Statement

This statement is the most basic of all control flow statements. It tells your program to execute a certain section of code only if a particular test evaluates to be true.

An If-statement can be followed by an optional else-if-else statement. This is very useful when testing various conditions using a single if-else-if statement.

### Example:

The following code displays how if-else and else-if statements work:

```
/*
   Created by Android ATC.
*/

fun main() {

    var score: Int = 80
    var grade: String?

    if(score >= 90) grade="Grade A"
    else if (score >= 80 ) grade="Grade B"
    else if (score >= 70 ) grade= "Grade C"
    else if (score >= 50 ) grade= "Grade D"
    else grade="Grade F"

    println(grade)

}
```

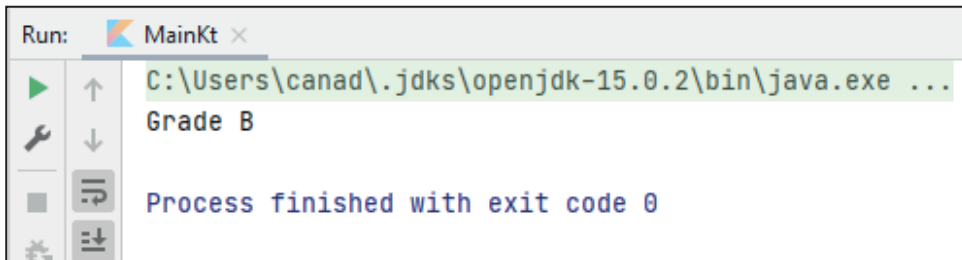
### Explain:

The code above includes four IF statements. IntelliJ IDEA follows the codes line by line from top to bottom. If any of these **IF** statements is true, it will run the action that belongs to this true **IF** statement and then move directly to run the code which comes after the last **else** statement. However, if any **IF** statement proves not to be true, the program moves to check the next **IF** statement.

In this example, the score value is 80 which means that the first IF statement **if (score >= 90)** is **false**; therefore, IntelliJ will move to the next **else if** statement without performing any action. In the second **else if** statement, if the condition is true, the program will execute the action related to this else-if statement, and then IntelliJ compiler will move directly after the last **else** line in this if else block.

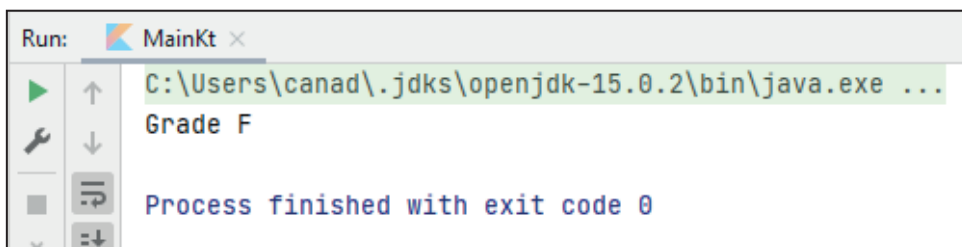
You may add any number of else-if lines depending on the purpose of the program. However, you must close this “else-if” block with an **else** statement at the end.

When you run the Kotlin program, the following result is shown:



The screenshot shows the IntelliJ Run console for a program named 'MainKt'. The command executed is 'C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...'. The output is 'Grade B'. The console also shows 'Process finished with exit code 0'.

If you change the score value to 40, the following result is shown:



The screenshot shows the IntelliJ Run console for a program named 'MainKt'. The command executed is 'C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...'. The output is 'Grade F'. The console also shows 'Process finished with exit code 0'.

## If Else and Logical Operators

Logical operators like AND “&&” and OR “||” may be utilized within If- Else statements. If you have two conditions (i.e. multiple Boolean expressions) which must be true or one of them is enough to be true respectively, logical operators IF statement may be used to test more than one condition at the same time.

Logical operators like AND “&&” and OR “||” may be utilized within If -Else statements. When you have two conditions (i.e. multiple Boolean expressions) where one condition should at least be satisfied, logical operators (i.e. IF statements) are used to test more than one condition at the same time.

The following table displays the || (OR) operator results (true or false) of the two Boolean expressions. These results will be considered by the IF statement:

Boolean Expressions (Condition): A	Boolean Expressions (Condition): B	A    B
True	True	True
True	False	True
False	True	True
False	False	False

The following table displays the use of “&&” (AND) logical operator to guarantee that the two conditions are true:

Boolean Expressions (Condition): A	Boolean Expressions (Condition): B	A && B
True	True	True
True	False	False
False	True	False
False	False	False

**Example:** The following example includes two conditions. It is enough to prove one of the conditions to consider the **if** condition is true because you used the || operator.

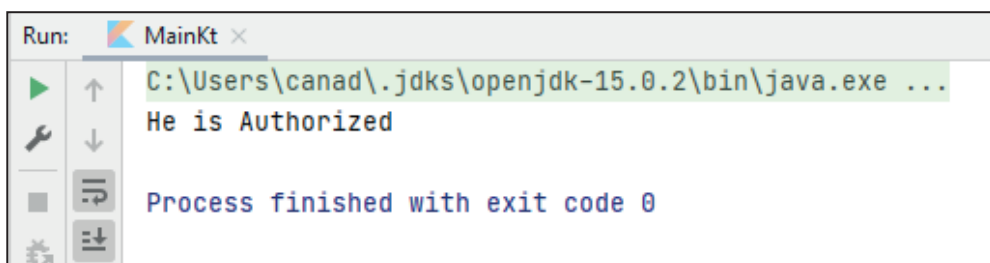
```

/*
   Created by Android ATC.
*/
fun main() {

    var Age = 16
    var DOB = 1998
    if (Age >= 18 || DOB >= 1998) println("He is Authorized")
    else println ("He is Not Authorized")
}

```

When you run the Kotlin program, you will get the following result:



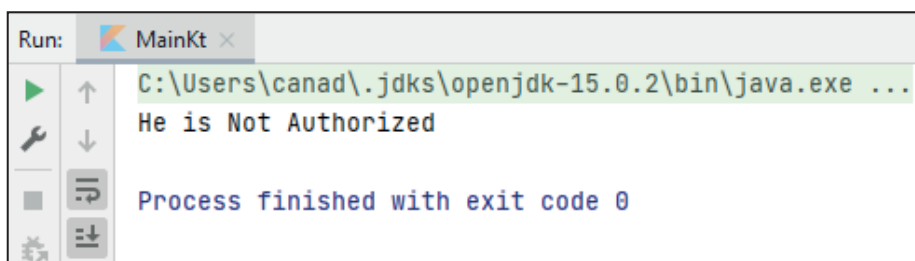
If you replaced the || operator with && operator as illustrated in the following example:



```
/*
    Created by Android ATC.
*/
fun main() {

    var Age = 16
    var DOB = 1998
    if (Age >= 18 && DOB >= 1998) println("He is Authorized")
    else println ("He is Not Authorized")
}
```

You will get the following run result, because the **if** condition will be considered true only if the two conditions are true. In this case, if the condition is false, **else** statement will work as shown below:



## When Statement and Expression

*When statement* is similar to the *switch statement* in other programming languages such as Java & C++. The *when statement* can have a number of possible execution paths.

When using **When** statements, we can put three options, and if none of them is true, IntelliJ IDEA compiler will perform the action related to the **else** statement. Thus, using when statement enables you to use many conditions for the same variable.

### Example:

The following example displays how *When statement* works:

```
/*
    Created by Android ATC.
*/

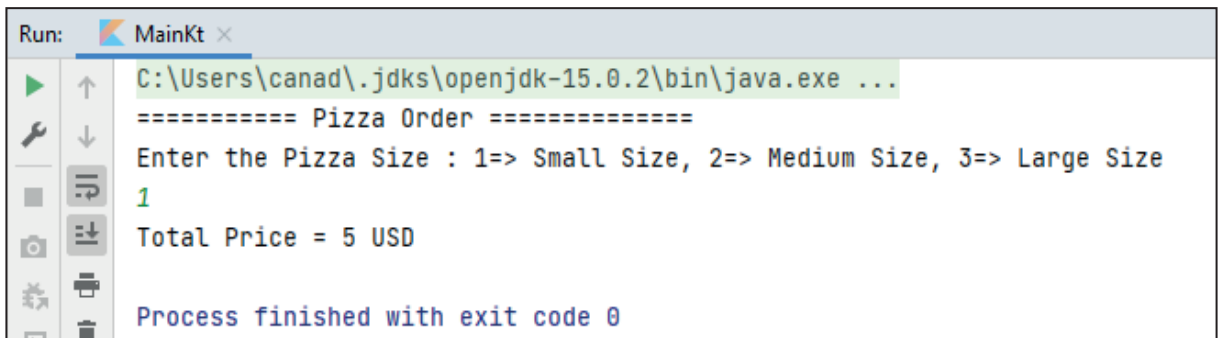
fun main() {
    println("===== Pizza Order =====")
    println("Enter the Pizza Size : 1=> Small Size, 2=> Medium Size, 3=> Large Size")
    var size= readLine()!!.toInt()
    var price: Int? =null
}
```

```

when (size) {
    1-> price=5
    2-> price=7
    3-> price=10
    else->println("You did not enter the correct size")
}
println("Total Price = $price" + " USD")
}

```

When you run this Kotlin program and type 1 for the pizza size, you will get the following result:



```

Run: MainKt x
C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...
===== Pizza Order =====
Enter the Pizza Size : 1=> Small Size, 2=> Medium Size, 3=> Large Size
1
Total Price = 5 USD
Process finished with exit code 0

```

What does the code below, which was part of the previous Kotlin program, mean?

```
var price: Int? =null
```

It means that the **price** is an integer variable which will return **null** value in case it did not get a value when the Kotlin program has run.

## For Loops

A **for** statement provides a compact way to iterate over a range of values. Programmers often refer to it as the “for-loop” because of how it repeatedly loops until a particular condition is satisfied.

The following is an example of how the for-loop works:

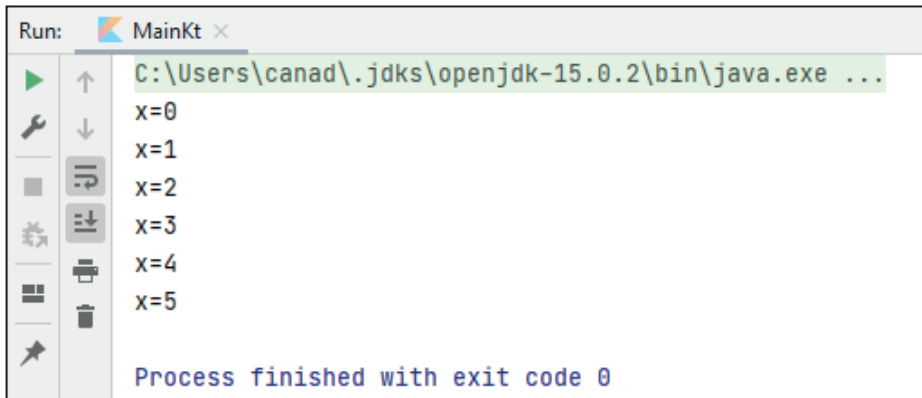
```

fun main() {
    for (x in 0..5) {
        println("x=$x")
    }
}

```

When the program starts, the **x** variable has a value = 0. However, when the Kotlin compiler runs the codes below within the for-loop braces and finishes its loop, the **x** variable takes the second value which is 1, and so on until it takes the last value which is 5 here. When the **x** variable takes its final value, the Kotlin compiler will resume its work outside the for-loop braces.

When you run the Kotlin program, you will get the following run result:

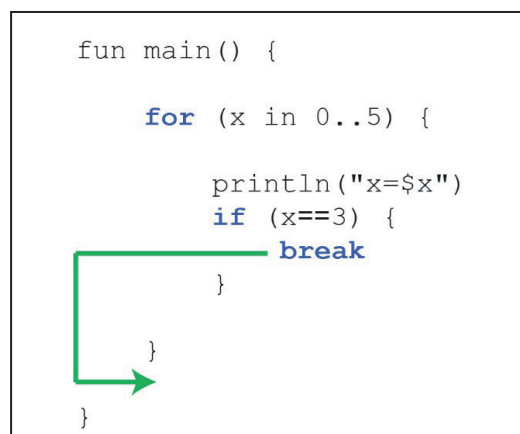


```
Run: MainKt x
C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...
x=0
x=1
x=2
x=3
x=4
x=5
Process finished with exit code 0
```

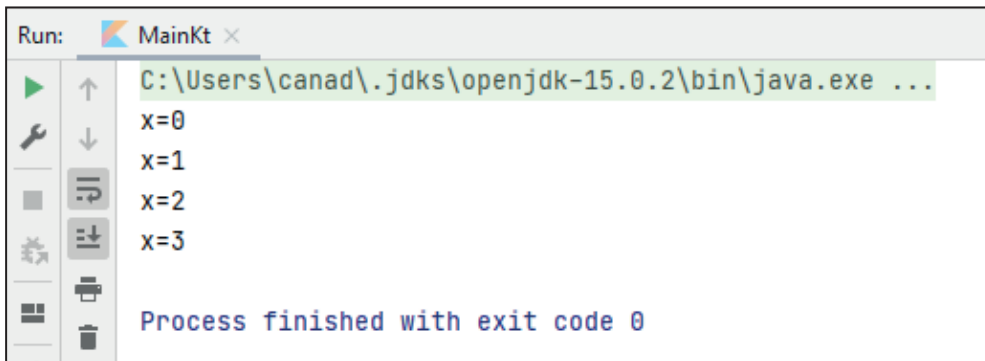
You may also use an if or if-else statement inside the loop if the program requires comparing a value more than once as illustrated in the following code.

```
fun main() {
    for (x in 0..5) {
        println("x=$x")
        if (x==3) {
            break
        }
    }
}
```

Here, when x is equal 3, the **break** statement will stop the **for** loop, and the program will continue running the code which is outside the for loop as illustrated in the following figure:



The run result of this code is as follows:



```
Run: MainKt x
C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...
x=0
x=1
x=2
x=3
Process finished with exit code 0
```

### Example:

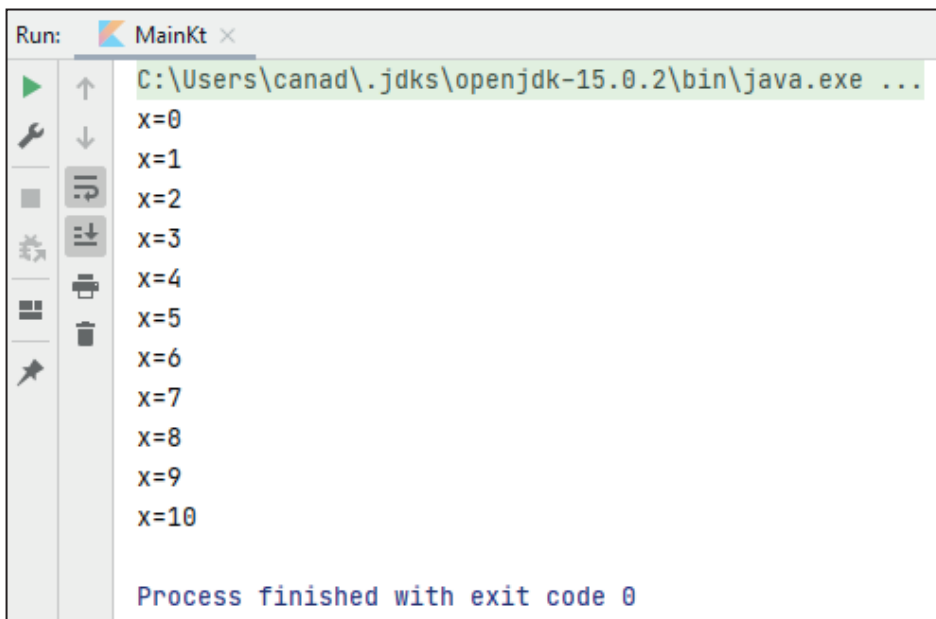
You can replace the number 5 in the previous example with an integer variable. This will determine the number of times you want the loop to work. The following example displays these changes:

```
fun main() {
    var y=10
    for (x in 0..y) {

        println("x=$x")

    }
}
```

When you run this Kotlin program, you will get the following result:



```
Run: MainKt x
C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...
x=0
x=1
x=2
x=3
x=4
x=5
x=6
x=7
x=8
x=9
x=10
Process finished with exit code 0
```

**Note:** You can also enter the value of the variable “y” using `readLine()!!.toInt()` statement.

**Example:**

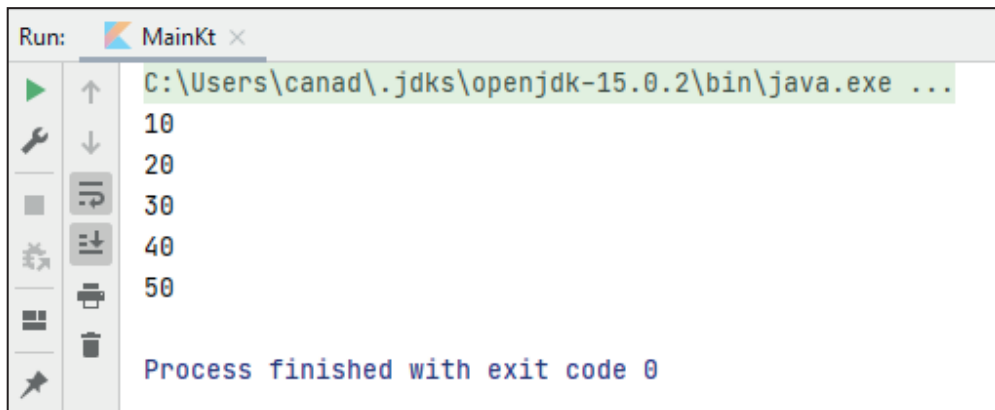
This example shows how you can use for-loop with array:

```
var x = arrayOf(10, 20, 30, 40, 50)

fun main () {

    for (index in 0..4) {
        println(x[index])
    }
}
```

The run result is as follows:



## While Loops

A while-statement depends on a Boolean expression and its counter. If the expression proves to be true, the *while-statement* executes the statement(s) in the while block. The while-statement continues testing the expression and executing its loop until the expression is proved false.

The following example explains how the while-loops work:

```
/*Created by Android ATC */

fun main() {

    var x=1
    while (x<=5)
    {
```

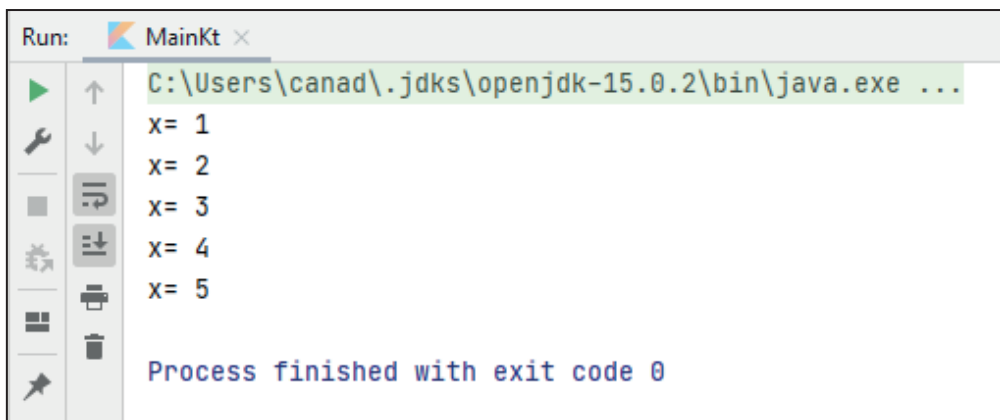
```
println("x= $x")
    x++ // x++ means: x=x+1 i.e. increment x by 1 each time

}
}
```

The following steps illustrate the flow of execution of the previous code:

- 1) The initial value of integer variable named x is 1.
- 2) The compiler checks the condition (**x <= 5**), which proves to be true, so it moves into the code block of the *while-statement* and prints "x=1".
- 3) The counter variable is incremented to value 1 and the compiler loops to the top.
- 4) The compiler checks again the *while-loop* condition ( $x \leq 5$ ), which also proves to be true, so the compiler prints "x=2" and increments the counter variable to value 2.
- 5) The program will continue working until it reaches x=6. After that, the compiler will check again the *while-loop* condition ( $x \leq 5$ ), but this time it will prove to be false. Therefore, the compiler will skip the code block and move forward to execute the code of the statements that follow the *while-statement*.

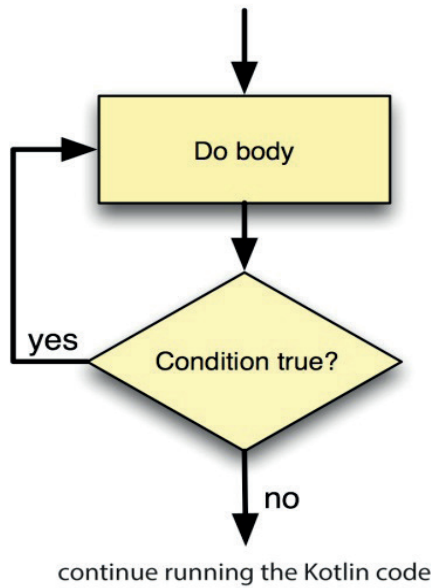
When you run the Kotlin program, you will get the following result:



## Do-while Loops

Contrary to the *while-loop*, the **do-while** loop starts evaluating its expressions from the bottom and not from the top. Thus, the statements within the *do* block are executed at least once. Do-while loops are exit controlled loops, i.e. the compiler will first execute the code block associated with the *do-while* loop, and then check the associated Boolean expression. If the Boolean expression proves to be true, only then the compiler will loop and execute the *do-while* code block again, or else, the compiler would skip the *do-while* code block and continue to execute the statements that follow the *do-while* loop.

The following figure displays the work flow of the *Do-while* loop:



```
/*Created by Android ATC*/

fun main() {

    var x = 1
    do {

        println("x= $x")
        x++
        // x++ means: x=x+1 i.e. increment x by 1 each time.
    }

    while (x <= 5)

}
```

When you run the Kotlin program, you will get the following result:

Run: MainKt x

```
C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...
x= 1
x= 2
x= 3
x= 4
x= 5

Process finished with exit code 0
```

## Jump Expressions

Kotlin has three structural jump expressions:

- 1) Break Statement
- 2) Continue Statement
- 3) Return Statement

### 1) Break Statement

The break statement allows you to exit a loop at any point, and overpass its normal termination expression. When the break statement is encountered inside a loop, the loop is immediately terminated, and resumes its work starting from the statement that follows the loop. The break statement can be used in any type of loop such as the While, do-While, or for-loop.

The following example displays the same do-while loop example code used previously in the break statement. In this example, when the count value of the variable **x** becomes equal to 3, the break statement will take action by immediately terminating the while-loop. Then the program control will resume its work starting with the statement and following the while loop:

```
/*Created by Android ATC*/

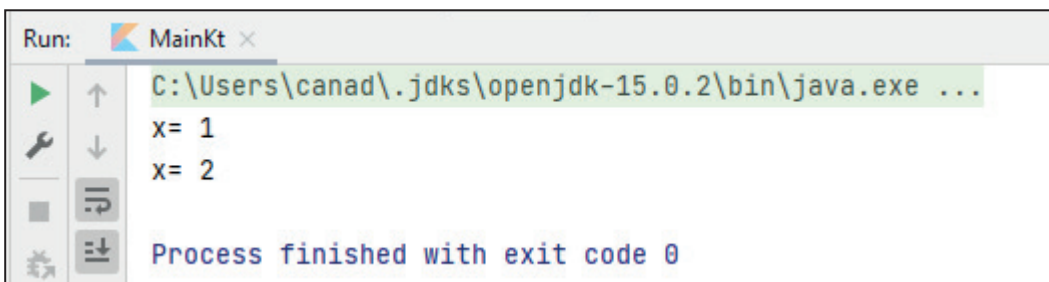
fun main() {

    var x=1
    do {

        println("x= $x")
        x++
        if(x==3)
            break
    }

    while (x<=5)
}
```

When you run the Kotlin program, you will get the following result:



```
Run: MainKt x
C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...
x= 1
x= 2
Process finished with exit code 0
```



## 2)Continue Statement

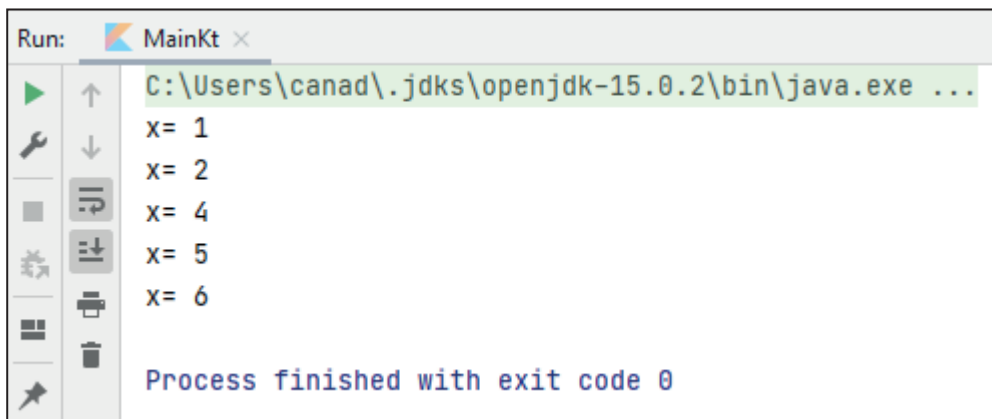
The continue statement lets control go directly to the test condition and then continue the looping process. In the case of the for-loop, the increment part of the loop continues. One good use of a continue-statement is when you want to restart or skip a statement sequence when an error occurs.

### Example:

In the following example, the result of the Kotlin program will print the count value from 1 to 6 except 3 where the continue statement will make the loop start again from the beginning of the do-while loop (or any other loop type):

```
/**Created by Android ATC */  
fun main() {  
  
    var x=0  
    do {  
  
        x++  
        if(x==3) {continue}  
        println("x= $x")  
    }  
  
    while (x<=5)  
}
```

When you run the Kotlin program, you will get the following result:



```
Run: MainKt x  
C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...  
x= 1  
x= 2  
x= 4  
x= 5  
x= 6  
  
Process finished with exit code 0
```

## 3)Return Statement

The return statement will be discussed later in this course with other related topics.

## Functions

A function is any close identity which includes a certain code or a collection of statements grouped together to perform an operation. Each function has a unique name which is used to call this function inside the Kotlin program several times without the need to duplicate statements in multiple source code files.

### Function Structure

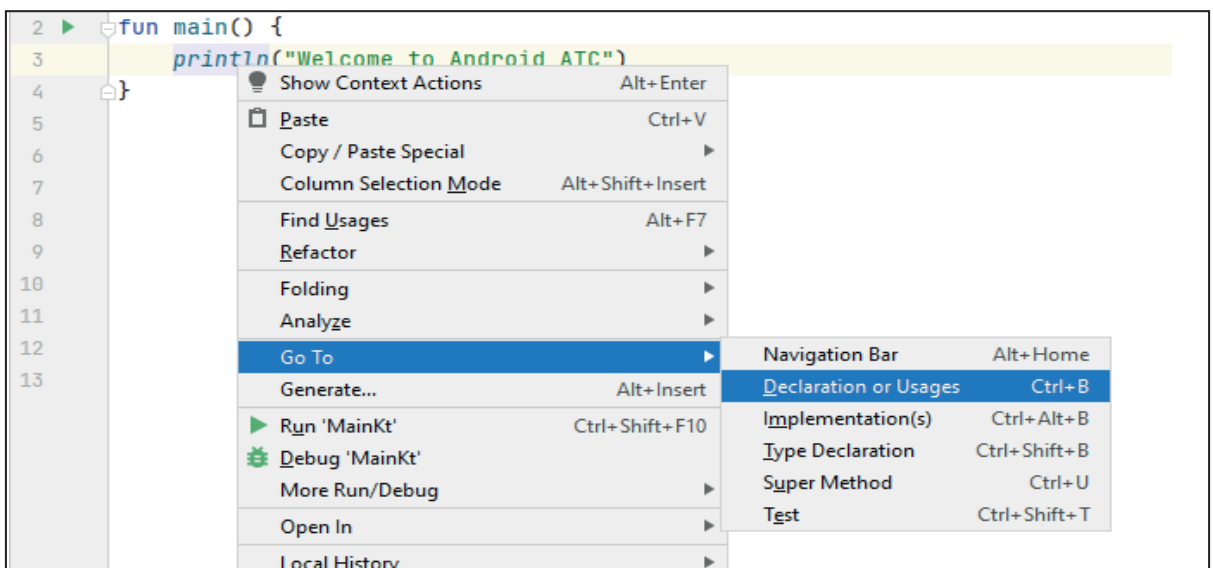
The following image displays the main components of the main function:

```
fun main() {  
    println("Welcome to Android ATC")  
}
```

For example, the following Kotlin code `main()` is a function. Also, `println()` which is part of the main function, is a function too and its role is to print what is within two parentheses `()`:

```
fun main() {  
    println("Welcome to Android ATC")  
}
```

This `println()` function consists of a hidden code, so when you use this function, you call this function hidden code to work for a specific purpose many times from different locations in the program. To see the hidden code which built `println()` function, right click this function, select **Go To**, then select **Declaration or Usages** as illustrated in the figure below:



You will get the following code which runs in back to show the role of `println()` function:

A screenshot of an IDE window with two tabs: 'main.kt' and 'Console.kt'. The 'Console.kt' tab is active and shows Kotlin code. The code defines three overloaded versions of the `println()` function. Each function is marked with `@kotlin.internal.InlineOnly` and has a comment: 'Prints the given message and the line separator to the standard output stream.' The first function is `public inline fun print(message: CharArray) { System.out.print(message) }`. The second is `public actual inline fun println(message: Any?) { System.out.println(message) }`. The third is `public inline fun println(message: Int) { System.out.println(message) }`. The code is partially visible, showing lines 72 through 91. The `println()` text is visible in the bottom status bar.

```
72  @kotlin.internal.InlineOnly
73  public inline fun print(message: CharArray) {
74      System.out.print(message)
75  }
76
77      Prints the given message and the line separator to the standard output stream.
78
79  @kotlin.internal.InlineOnly
80  public actual inline fun println(message: Any?) {
81      System.out.println(message)
82  }
83
84      Prints the given message and the line separator to the standard output stream.
85
86  @kotlin.internal.InlineOnly
87  public inline fun println(message: Int) {
88      System.out.println(message)
89  }
89
90      Prints the given message and the line separator to the standard output stream.
91  @kotlin.internal.InlineOnly
92  public inline fun println(message: Long) {
93      System.out.println(message)
94  }
```

## Creating a Function

If you have a number of code that needs to be used more than once within your program, you can gather them inside a function which has a specific name and then call this function within the program as many times as needed.

Doing so will repeat the work without the need to write the codes again. Creating functions makes programming easier to read and write.

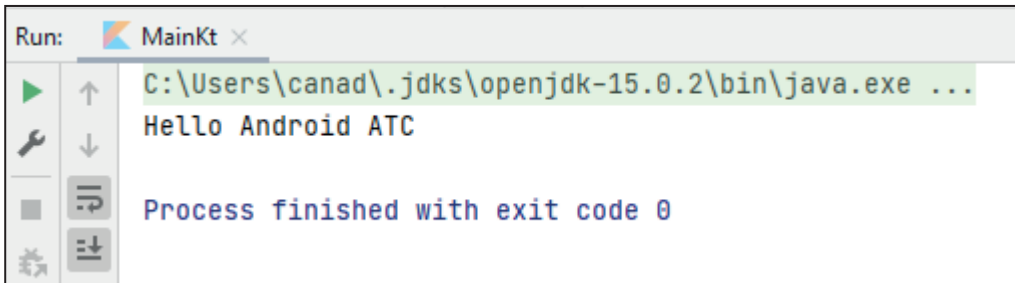
### Example:

In this example, and as illustrated in the figure below, the `sum` function takes two integer variables `x` and `y` and prints out their sum value.

```
1
2 ▶ fun main() {
3
4     println("Hello Android ATC")
5 }
6
7
8 fun sum (x:Int, y:Int)
9 {
10     var z=x+y
11     println("Sum Result =$z")
12 }
13
14
```

sum function

When you run the Kotlin code in this example, only the code inside the **main** function will be executed. The result is shown in the following snapshot:



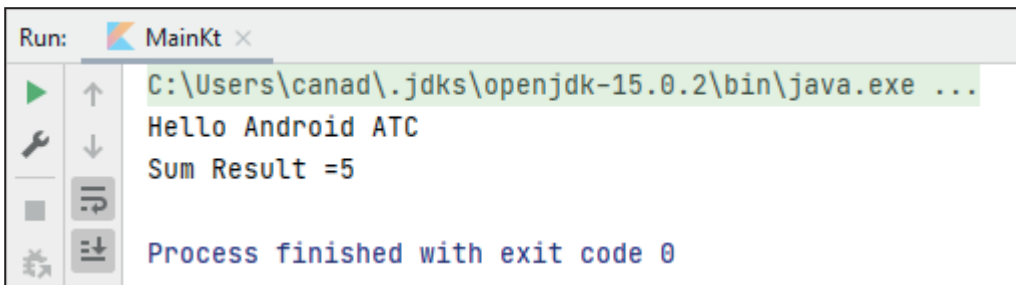
You need to call the **sum** function which you have created inside the **main** function. The following example includes the same previous code as well as the way used to call the sum function:

```
fun main() {

    println("Hello Android ATC")
    sum (2,3)
}

fun sum (x:Int, y:Int)
{
    var z=x+y
    println("Sum Result =$z")
}
```

The run result is shown below:



You called the function by one-line: `sum(2, 3)`. This function or any other function may include a block of code lines. As you can see, functions make writing programming codes easier.

You can also write your functions using the **return** command; this method makes the function easier to write and understand. In the following code, you will find the same previous code and added to it a **return** command which will return the function operation result to the location where the function had been called from.

```
fun main() {  
  
    println("Hello Android ATC")  
    var result=sum (2,3)  
    println("Result = $result")  
}  
  
fun sum (x:Int, y:Int) :Int  
{  
    var z=x+y  
    return z  
}
```

### Explanation:

The previous code is explained step by step below:

- 1) The **sum** function includes two parameters (**x** and **y**) - each has been declared previously as an integer variable. The variable **z** is the result of adding **x** and **y**. It is important here to add the **return** command at the end of the function. This would return the result of the addition operation, variable **z**, to the location from which the function was called. You must declare the data type of the return value of **z**, so you add **Int** (highlighted in gray color) to the function below.

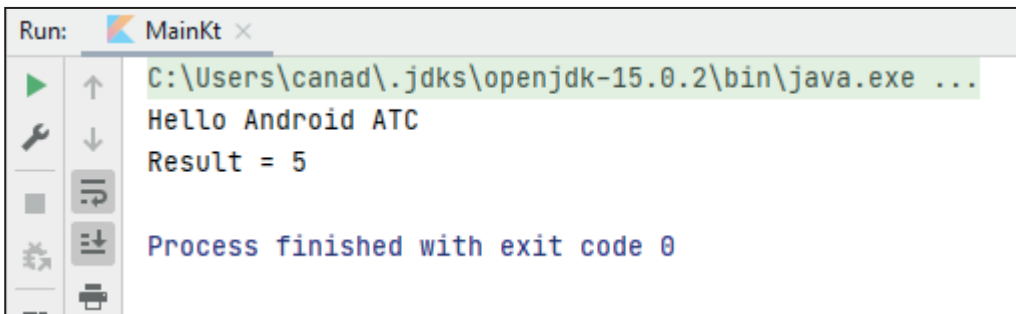
```
fun sum (x:Int, y:Int) :Int  
{  
    var z=x+y  
    return z  
}
```

- 2) The code below in the main function (highlighted in gray) is part of what's called the sum function.

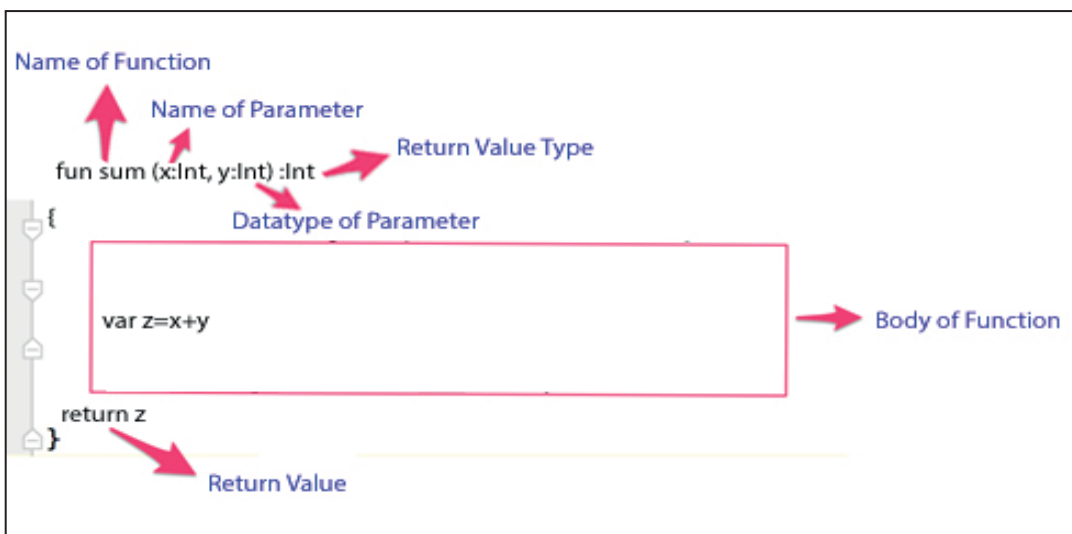
```
fun main() {  
  
    println("Hello Android ATC")  
    var result=sum (2,3)  
    println("Result = $result")  
}
```

The variable **result** will call the **sum** function by assigning the values **2** and **3** to the variables **x** and **y** respectively. The sum function will then work and return the **z** value to replace the `sum (2, 3)`.

When you run the code, you will get the following result:



Now, you know how to use functions in a Kotlin program and know how important they are. You can understand now the role of each component in a function. The following image displays the function structure:

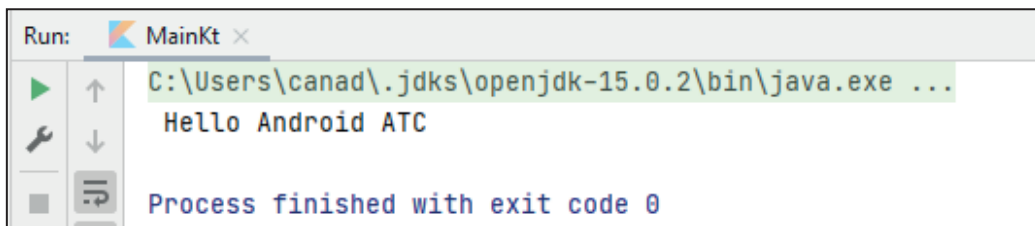


**Example:**

Some functions may work without any parameters or return value, like the following code:

```
fun main() {  
    Hello()  
}  
  
fun Hello ()  
{  
    println(" Hello Android ATC")  
}
```

When you run the code, you will get the following result:



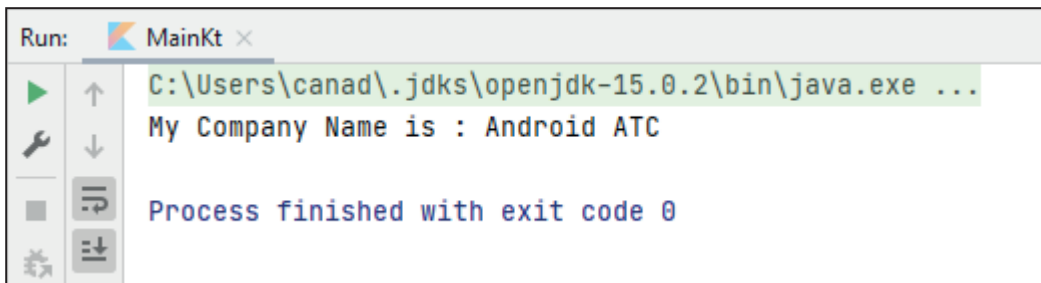
## Functions and Variable Scope

The location of variables - whether inside or outside the main function - has an important effect on the function's output. The following code displays how the program workflow will be affected if we change the location of variables inside or outside the functions:

The following Kotlin code includes a function called **name** which is used to print the name value:

```
var myCompany="Tesla"  
  
fun main() {  
    name("Android ATC")  
}  
  
fun name (myCompany:String) {  
    println("My Company Name is : $myCompany")  
}
```

The run result of this code is the following:

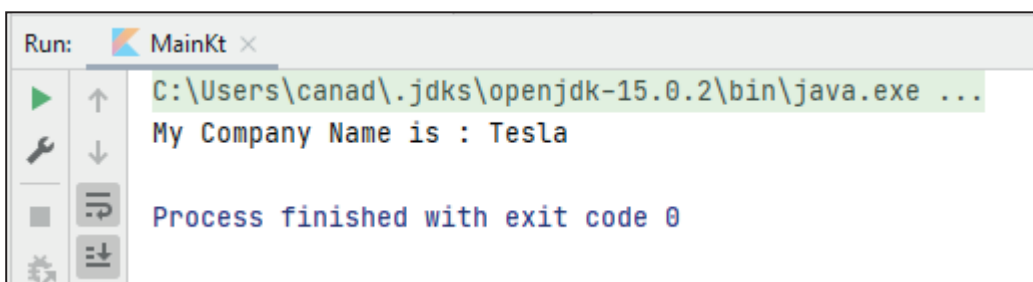


Here, the `var myCompany="Tesla"` is outside any function body. This means that `"myCompany"` variable is considered as a global variable and can be used within any other function.

However, if this variable was written inside any function's body (local variable) as it illustrated in the code below, it will be dominant over any other configuration that may come from outside this function. As shown in the code below, you are going to use the same previous example and add `var myCompany="Tesla"` inside the body of the `name ()` function.

```
fun main() {  
    name("Android ATC")  
}  
  
fun name (myCompany:String) {  
    var myCompany="Tesla"  
    println("My Company Name is : $myCompany")  
}
```

The run result will be the following:



**Note:** For sure in the last code you should write the code in a better way; but, this is just an example to explain how you may use a variable inside a function.