kotlin programming

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

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

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

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

In Kotlin, code statements do not have to end with a semicolon (;)

difference between “println” and print:

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

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

Comments works same as c++.

single line

multi line /* ….text…..*/

VARIABLES

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

`var` *`variableName`* `=` *`value`*

`val` *`variableName`* `=` *`value`*

```
var name = "John"
val birthyear = 1975

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

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

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

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

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

output :- John Doe

```
val x = 5
val y = 6
println(x + y)
```

output :- 11

some rules to keep in mind➖

- Names should start with a lowercase letter and it cannot contain whitespace
- Names are case sensitive ("myVar" and "myvar" are different variables)
- Reserved words (like Kotlin keywords, such as `var` or `String`) cannot be used as names
- Names can contain letters, digits, underscores, and dollar signs

In kotlin you may or may not specify the data types.

IMPORTANT:-

Use `Float` or `Double`?

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

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

Difference between speicfying the types:

NOT SPECIFYING THE TYPE

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

WHEN THE TYPES ARE SPECIFIED

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

Type Conversion

To convert a numeric data type to another type, you must use one of the following functions: `toByte()`

, `toShort()`

, `toInt()`

, `toLong()`

, `toFloat()`

, `toDouble()`

or `toChar()`

e.g.

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

Operators

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

STRINGS

the spicification of the strings are discussed already.

Access a string

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

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

Length of a string: String is an object in kotlin. by writing a dot character (`.`) after the specific string variable. For example, the length of a string can be found with the `length` property:

Kotlin Booleans

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

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

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

Example:

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

Kotlin If ... Else

Kotlin if

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

Kotlin else

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

Kotlin else if

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

```
}
// Outputs "Good evening."
```

Kotlin If..Else Expressions

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

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

Kotlin When

```
val day = 4

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

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

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

Kotlin While Loop

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

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

Kotlin Break

```
var i = 0
while (i < 10) {
  println(i)
  i++
  if (i == 4) {
    break
  }
}
// output - 0 1 2 3
```

Kotlin Continue

```
var i = 0
while (i < 10) {
  if (i == 4) {
    i++
    continue
  }
  println(i)
  i++
}
// output- 0 1 2 3 4 5 6 7 8 9
```

Kotlin Array

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

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

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

Change an Array Element

```
val cars = arrayOf("Volvo", "BMW", "Ford", "Mazda")
cars[0] = "Opel"
println(cars[0])
// Now outputs Opel instead of Volvo
println(cars.size)
// Outputs 4
```

Loop through an array

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

Traditional For Loop

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

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

You will learn more about **ranges** in the next chapter - which will create a range of values.

Kotlin Ranges

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

```
for (chars in 'a'..'x') {
  println(chars)
}
// print the whole alphabet.
```

To check if a value exsisted

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

Kotlin Functions

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

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

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

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

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

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


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

// output "i just got executed"
```

example

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

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

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

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

Return values

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

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

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

// 8 (3 + 5)
```

using two parameters

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

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

// 8 (3 + 5)
```

important:

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

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

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

// 8 (3 + 5)
```

OOPS

OOP stands for **Object-Oriented Programming**.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

to create a class

```
class Car {
  var brand = ""
  var model = ""
  var year = 0
}
```

creating an object

```
// Create a c1 object of the Car class
val c1 = Car()

// Access the properties and add some values to it
c1.brand = "Ford"
c1.model = "Mustang"
c1.year = 1969

println(c1.brand)   // Outputs Ford
println(c1.model)   // Outputs Mustang
println(c1.year)    // Outputs 1969
```

multiple objects

```
val c1 = Car()
c1.brand = "Ford"
c1.model = "Mustang"
c1.year = 1969

val c2 = Car()
c2.brand = "BMW"
c2.model = "X5"
c2.year = 1999

println(c1.brand)  // Ford
println(c2.brand)  // BMW
```

kotlin constructor

```
class Car {
  var brand = ""
  var model = ""
  var year = 0
}

fun main() {
  val c1 = Car()
  c1.brand = "Ford"
  c1.model = "Mustang"
  c1.year = 1969
}
```

difference

```
class Car(var brand: String, var model: String, var year: Int)

fun main() {
  val c1 = Car("Ford", "Mustang", 1969)
}
```

```
class Car(var brand: String, var model: String, var year: Int)

fun main() {
  val c1 = Car("Ford", "Mustang", 1969)
  val c2 = Car("BMW", "X5", 1999)
  val c3 = Car("Tesla", "Model S", 2020)
}
```

Class Function Parameters

```
class Car(var brand: String, var model: String, var year: Int) {
  // Class function
  fun drive() {
    println("Wrooom!")
  }

  // Class function with parameters
  fun speed(maxSpeed: Int) {
    println("Max speed is: " + maxSpeed)
  }
}

fun main() {
  val c1 = Car("Ford", "Mustang", 1969)

  // Call the functions
  c1.drive()
  c1.speed(200)
}
```

inheritance

In Kotlin, it is possible to inherit class properties and functions from one class to another. We group the "inheritance concept" into two categories:

- **subclass** (child) - the class that inherits from another class
- **superclass** (parent) - the class being inherited from

In the example below, `MyChildClass` (subclass) inherits the properties from the `MyParentClass` class (superclass):

```
// Superclass
open class MyParentClass {
  val x = 5
}

// Subclass
class MyChildClass: MyParentClass() {
  fun myFunction() {
    println(x) // x is now inherited from the superclass
  }
}

// Create an object of MyChildClass and call myFunction
fun main() {
  val myObj = MyChildClass()
  myObj.myFunction()
}
```

Why And When To Use "Inheritance"?

- It is useful for code reusability: reuse properties and functions of an existing class when you create a new class.