PUBLISHED BY MKPROJECTS

MKPROJ.COM

Kotlin is a statically typed, general purpose programming language developed by JetBrains. Kotlin is designed to work alongside Java, and due to it's type inference, it's code can be a lot more concise and easier to read.

1.1 Installation

You can install the Kotlin Compiler manually from it's Github Releases
`https://github.com/JetBrains/kotlin/releases/tag/v1.4.32`

MacOS Homebrew

To install on Mac OS, you may also choose to install the Kotlin compiler via Homebrew:

```
$ brew update
$ brew install kotlin
```

Linux Snap

To install on a Linux Distribution, you may install via Snap:

```
$ sudo snap install --classic kotlin
```

1.2 Compile

To Compile a Kotlin `/kt` program, you must first compile it using the Kotlin Compiler to produce a Java !.jar! file:

```
# We will be compile an example test.kt program
$ kotlinc hello.kt -include-runtime -d hello.jar
# -include-runtime makes the jar file self-contained and runnable
# -d is the output directory option followed by the output file
# Run kotlinc -help for more options
```

To run the program use java:

```
$ java -jar hello.jar
```

Now that you know how to compile and run the program, let's proceed to the next section!

To explain this section, we will be going line by line of the different parts of our program:

```
// examples/01-hello.kt
fun main(){
    // Our program is very simple and is aimed to say hello world
    /*
    We will run each word of "Hello World Kotlin!" in seperate println
    so we can show the order of execution.
    */
    println("Hello")
    println("World")
    println("Kotlin!")
}
```

2.1 Main Function

You can see at the beginning of our program is `fun main(){}`, the main function is the starting point of which the Kotlin Program executes the code. For an executable program, a main function is required for the Kotlin Program.

2.2 Comments

Comments are used for the programmer and not for the actual execution of the program. It is typically used to document and jot ideas down on the program, and is ignored by the compiler. There is two different type of comments, there is `//` which is a single line comment and `\* */` which is multi-line comments that contains comments in between `\*` and `*/`.

2.3 Print Statements

A print statement is a way to output values onto the console, and there is two different print statements that exist.

- `println()` : This will create a new line after outputting
- `print()` : This will only output **(not create a new line)**

Now let's Compile and Run:

```
# If using this repo, cd examples
$ kotlinc 01-hello.kt -include-runtime -d 01-hello.jar
$ ls
01-hello.jar  01-hello.kt
$ java -jar 01-hello.jar
Hello
World
Kotlin!
```

We will begin this section off by disecting this example program that contains most topics that will be discussed:

```
fun main(){
    /*
     Our goal is to be able to have variables that
     calculate various geometric areas
     */

    // An immutable variable for the value of pi
    val pi = 3.14159

    var radius = 10.0
    var area = pi * radius // Circle

    //Since we will be using the area statement a lot
    //it's good to make it a variables

    var stm = "The area of this "
    var shape = "circle"

    println(stm + shape + " is $area") // String Concetation + Template

    // Now let's say we want to find the area and volume of a square

    var length = 56.21
    area = Math.pow(length, 2.00) // Will square the length
    var volume = Math.pow(length, 3.00) // Will cube the length
```

```
        // Now to prepare this statement
        // However I want each beginning letter of each word capatalized

        shape = "square"

        stm = stm + shape + " is $area" +
        "\nThe volume of this " + shape + " is $volume"

        println(stm.capitalize())

    }
```

3.1 Mutable and Immutable Variables

Variables are used in a program to store data, and in Kotlin there is two different ways to declare a variable, one being `var` which is for mutable variables, and the other `val` being for immutable. Now what is the difference between mutable and immutable, well the difference is between being able to be reassigned. A mutable variable is a variable that can be reassigned any amount of times in a program, while a immutable cannot be reassigned a value once it is given one.

You can see this in the program in the use case of `val pi = 3.14159` where we don't want `pi` to ever change. You can see a few of our varaibles were reassigned, such as `shape`, `area`, etc.

3.1.1 Type Inference

If you're coming from Java to Kotlin, you must've been being so confused by the no semi colons, or the fact that when you declare a variable, you don't need to provide a type with it. This is what Kotlin means with type inference, where the Kotlin compiler at compiling time can infer what data type each varaible is.

A nice way to look at Kotlin's type inference is to compare Kotlin and Java:

Java	Kotlin
`double num_double = 67.89;`	`var num_double = 67.89`

In Java a data type must be declared explicitly while in Kotlin, the type is implicitly inferred.

3.2 Strings

Strings are essentially an array of characters, and in Java is not considered to be a primitive type, however, Strings do have useful builtin properties that are good to take advantage of.

String Concentation

String Concetation is to combine Strings together with the + operator, and as you can see, it was useful for formatting our statement with the various variables we had with us.

Ex.

```
stm = stm + shape + " is $area" +
"\nThe volume of this " + shape + " is $volume"
```

String Templates

String Templates, which is also used in our example is useful to have variables in a String with `$variable` inside the String to cleanly write the strings without needing to concetate a lot.

String Builtin Properties

In the ending of the program, we use a builtin String function called `.capatilize()` which served our purpose to capatalize each word in a String. There is also `.length()` which returns the number of characters in a String. It is a good idea to explore the various String properties that Kotlin offers.

Character Escape Sequences

Character escape sequences consist of a backslash and character and are used to format text.

- `\n` Inserts a new line
- `\t` Inserts a tab
- `\r` Inserts a carriage return
- `\'` Inserts a single quote
- `\"` Inserts a double quote
- `\\` Inserts a backslash
- `\$` Inserts a dollar symbol

3.3 Math

Arithmetic Operators

Kotlin includes the following arithmetic operators:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division
- `%` modulus

Order of Operations

The order of operations for compound arithmetic expressions is as follows:

1. Parantheses
2. Multiplication
3. Division
4. Modulus
5. Addition
6. Subtraction

When an expression contains operations such as multiplication and division or addition and subtraction side by side, the compiler will evaluate the expression in a left to right order.

Augmented Assignment Operators

An augmented assignment operator includes a single arithmetic and assignment operator used to calculate and reassign a value in one step.

```
var a = 10
var b = 7

a += b // 10 + 7
a -= b // 17 - 7
a *= b // 10 * 7
a /= b // 70 / 7
a %= b // 10 % 7
```

Increment and Decrement Operators

Increment and decrement operators provide a shorthand syntax for adding or subtracting 1 from a value. An increment operator consists of two consecutive plus symbols, ++, meanwhile a decrement operator consists of two consecutive minus symbols, --.

```
var num = 78

num++ // 79
num-- // 78
```

The Increment Operator is commonly used in loops (Chapter 6)

The Math Library

The Math library, inherited from Java, contains various mathematical functions that can be used within a Kotlin program.

```
Math.pow(2.0, 3.0)  // 8.0
Math.min(6, 9)      // 6
Math.max(10, 12)    // 12
Math.round(13.7)    // 14
```

4.1 If Statements

An `if` statement is used to execute a section of code if an expression results to `true`. A great way to look at this is an example:

```
var a = 5

if (a == 5) {
    println("True")
}
```

Since `a` is indeed equal to 5, then the program will execute `println("True")` however, let's say we had...

```
var a = 5

if (a == 6) {
    println("True")
}
```

Since `a` doesn't equal to 6, the expression results to `false`, so the program doesn't print anything.

4.2 Else Statements

An `else` statement is used to execute a section of code if an expression results to `false`. Let's fix our previous example a bit to include an `else` statement:

```
var a = 5

if (a == 6) {
    println("True")
} else {
    println("False")
}
```

Since a doesn't equal to 6, it follows the `else` statement, and executes `println("False")`.

4.3 Else If Statements

An `else if` expression allows for more conditions to be evaluated within an `if/else` expression.

Note: You can use multiple else if expressions as long as they appear after the if expression and before the else expression.

```
var num = 8

if (num > 8){
    println(num)
} else if (num == 8){
    println(num++)
} else {
    println(num--)
}
// Prints 9
```

4.4 Nested Conditionals

A nested conditional is a conditional statement within another conditional statement. Essentially an `if` statement inside an `if` statement.

```
var foo = true
var bar = true

if (foo){
    println("FOO!")
    if (bar){
        println("BAR!")
    }
} else {
    println("No Foo or Bar for you")
}
```

```
/*
Prints:
FOO!
BAR!
*/
```

4.5 When Statements

After a while you may get sick of a huge tree of `else if` statements, so any better option? The `when` statement is useful to execute code depending on the value of the expression.

```
var grade = "F"

when (grade){
    "A" -> println("Excellent")
    "B" -> println("Okay")
    "C" -> println("Bad")
    "D" -> println("Terrible")
    else -> println("FAIL")
}
// Prints FAIL
```

4.6 Comparison Operators

Comparison operators are symbols that are used to compare two values in order to return a result of `true` or `false`.

The comparison operators include the following:

- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to

```
var Khloe = 19
var Zane = 18
var Nimu = 20

Khloe > Zane //true
Zane < Nimu // true
Khloe >= Nimu // false
Nimu <= Zane // false
```

4.7 Logic Operators

Logical operators are symbols used to evaluate the relationship between two or more Boolean expressions in order to return a `true` or `false` value.

The logic operators include the following:

- ! NOT
- && AND
- || OR

"!" NOT Operator

The NOT operator evaluates a boolean expression and return it's opposite value `(true -> false)`.

```
var exp = true
println(!exp)
// Prints false
```

"&&" AND Operator

The AND operator only evaluates to `true` if both expressions it's evaluating results to `true`.

```
if (4 < 5 && 6 > 3){ //Both expressions are true
    println("True")
} else {
    println("False")
}
// Prints true
```

"||" OR Operator

The OR operator only evaluates to `true` if one of the expressions result to `true`.

```
if (1.2 < 9.8 || 5 > 9){ // One results to true
    println("True")
} else {
    println("False")
}
//Prints True
```

Order of Evaluation

The order of evaluation when using multiple logical operators in a single Boolean expression is:

1. Expressions placed in parantheses
2. NOT(!) Operator
3. AND(&&) Operator
4. OR(||) Operator

Ex.

```
    !true || (true && false) // false
```

4.8 The Range Operator

The range operator (. .) is used to create a succession of number or character values.

```
var grades = 90

when (grades){
    90..101 -> println("A+")
    80..90 ->  println("A")
    .
    .
    .
    else -> println("F")
}
// Prints A+
```

In Kotlin, there is various type of collections that exist for different case usages. In this section, we will be covering `Lists`, `Sets` and `Maps`.

5.1 Lists

5.1.1 Mutable and Immutable Lists

- An immutable list represents a group of elements with read-only operations.
  - It can be declared with the term `listOf`, followed by a pair of parentheses containing elements that are separated by commas.
  - Ex. `var fruits = listOf("Apples", "Bananas", "Oranges")`
- A mutable list represents a group of ordered elements with read and write operations.
  - It can be declared with the term, `mutableListOf` followed by a pair of parentheses containing elements that are separated by commas.
  - Ex. `var companies = mutableListOf("Google", "Tesla", "Apple")`

Accessing Lists Elements

In order to retrieve an element from a list, we can reference its numerical position or index using square bracket notation `[ ]`.

Note: The beginning index of a list starts at `0`.

The Size Property

The `size` property returns the size of a collection of the number of elements that exists.

```
var grades = listOf("A", "B", "C", "D", "F")
println(grades.size)
// Prints 5
```

5.1.2 List Operations

The list collection supports various operations in the form of built-in functions that can be performed on its elements.

Some functions perform read and write operations, whereas others perform read-only operations.

The functions that perform read and write operations can only be used on `mutable` lists while read-only operations can be performed on both `mutable` and `immutable` lists.

```
var programming_languages = mutableListOf("Kotlin", "Java", "C++", "Rust")

if (programming_languages.contains("Python") /*read only*/){
  println("Python is pretty cool")
} else {
  programming_languages.add("Python") //write
}
```

5.2 Sets

5.2.1 Immutable and Mutable Sets

- An immutable set represents a collection of unique elements in an unordered format whose elements cannot be changed throughout a program.
  - It is declared with the term, `setOf`, followed by a pair of parentheses, `( )` holding unique values.
  - `var origin_factions = setOf("Mythicals", "Sorcerers", "Kindgom")`
- A mutable set represents a collection of ordered elements that possess both read and write functionalities.
  - It is declared with the term, `mutableSetOf`, followed by a pair of parantheses, `( )` holding unique values.
  - `var mkproj = mutableSetOf("Phaktionz", "Books", "UniConv", "Moka")`

Accessing Set Elements

Elements in a set can be accessed using the `elementAt()` or `elementAtOrNull()` functions.

- The `elementAt()` function gets appended onto a set name and returns the element at the specified position within the parentheses.
- The `elementAtOrNull()` function is a safer variation of the `elementAt()` function
  - Returns `null` if the position is out of bounds as opposed to throwing an error.

```
var example = setOf("Foo", "Bar", "Baz")

println(example.elementAt(1)) // Prints Bar
println(example.elementAt(5)) // Returns an error
println(example.elementAtOrNull(5)) // Prints null
```

5.3 Maps

5.3.1 Immutable and Mutable Maps

- An immutable Map represents a collection of entries that cannot be altered throughout a program.
  - It is declared with the term, `mapOf`, followed by a pair of parentheses.
    - Within the parentheses, each key should be linked to its corresponding value with the `to` keyword, and each entry should be separated by a comma.
    - `var student = mapOf("Johnny" to 95, "Billy  to 65, "Kimmy" to 85)`
- A mutable map represents a collection of entries that possess read and write functionalities. Entries can be added, removed, or updated in a mutable map.
  - A mutable map can be declared with the term, `mutableMapOf`, followed by a pair of parentheses holding key-value pairs.
  - `var prices = mutableMapOf("Gum" to 1.50, "Kitkat" to 0.88, "Spinach" to 3.97)`

Retrieving Map Keys and Values

- Keys and values within a map can be retrieved using the `.keys` and `.values` properties.
- The `.keys` property returns a list of key elements, whereas the `.values` property returns a list of value elements.
- To retrieve a single value associated with a key, the shorthand, `[key]`, syntax can be used.

```
var albums = mapOf("2001" to "Dr Dre", "4:44" to "Jay Z", "Relapse" to "Eminem")

println(albums.keys)
// Prints ["2001", "4:44", "Relapse"]
println(albums.values)
// Prints ["Dr Dre", "Jay Z", "Eminem" ]
println(albums["4:44"])
// Prints Jay Z
```

Adding and Removing Map Entries

An entry can be added to a mutable map using the `put()` function. Oppositely, an entry can be removed from a mutable map using the `remove()` function.

- The `put()` function accepts a key and a value separated by a comma.
- The `remove()` function accepts a key and removes the entry associated with that key

```
var albums = mutableMapOf("2001" to "Dr Dre", "4:44" to "Jay Z", "Relapse" to "Eminem")

albums.put("All Eyez On Me", "2Pac")

println(albums)
// Prints:
// {"2001"="Dr Dre", "4:44"="Jay Z", "Relapse"="Eminem", "All Eyez On Me"="2Pac"}

albums.remove("2001")
println(albums)
// Prints:
// {"4:44"="Jay Z", "Relapse"="Eminem", "All Eyez On Me"="2Pac"}
```

6.1 for Loops

A `for` loop is used when we know how many times we want a section of code repeated.

Let's examine the following code:

```
var list = listOf["Hello", "Example", "Test"]

for (i in 0..list){
    println(list[i])
}
/* Prints:
Hello
Example
Test
*/
```

- `for` is a keyword used to declare a for loop.
- We define `i` as the loop variable. This variable holds the current iteration value and can be used within the loop body.
- The `in` keyword is between the variable definition and the iterator.
- The range `0..list` is the for loop iterator.

An `iterator` is an object that allows us to step through and access every individual element in a collection of values.

Note: It is important to note that the loop variable only exists within the loop's code block. Trying to access the loop variable outside the `for` loop will result in an error.

6.1.1 Controlling Iteration

Sometimes we want to count backwards, or count by 5s, or maybe both! Using certain functions alongside or instead of the normal range operator `(..)` can enhance the iterative abilities of our `for` loops. The functions `downTo`, `until` and `step` give us more control of a range and therefore more control of our loops.

- The `downTo` function simply creates a reverse order group of values, where the starting boundary is greater than the ending boundary. To accomplish this, replace the range operator `(..)` with `downTo`:

```
for (i in 4 downTo 1) {
  println("i = $i")
}
/*
Output:
i = 4
i = 3
i = 2
i = 1
*/
```

We can see in the output that the first number in `i` is 4 and the last is 1

- The `until` function creates an ascending range, just like the `(..)` operator, but excludes the upper boundary:

```
for (i in 1 until 4) {
  println("i = $i")
}
/*
Output:
i = 1
i = 2
i = 3
*/
```

The upper boundary, 4, is not included in the output

- Up until now, each of these functions, including the range operator (..), have counted up or down by one. The `step` function specifies the amount these functions count by:

```
for (i in 1..8 step 2) {
  println("i = $i")
}
/*
Output:
i = 1
i = 3
i = 5
i = 7
*/
```

The loop variable `i` now increases by 2 for every iteration. The last number output is 7, since 2 steps up from that is 9 which is outside the defined range, `1..8`

A function is a named, reusable block of code that can be called and executed throughout a program.

A function is declared with the `fun` keyword, a function name, parentheses containing (optional) parameters, as well as an (optional) return type.

To call/invoke a function, write the name of the function followed by parentheses.

```
fun print_five() {
  println("5")
}

fun main() {
  // Function call
  print_five() // Prints: 5
}
```

7.1 Function Parameters

In Kotlin, a parameter is a piece of data we can pass into a function when invoking it.

To pass data into a function, the function's header must include parameters that describe the name and data type of the incoming data. If a function is declared with parameters, then data must be passed when the function is invoked. We can include as many parameters as needed.

```
fun birthday(name: String, age: Int) {
   println("Happy birthday $name! You turn $age today!")
}

fun main() {
  birthday("Oscar", 26) // Prints: Happy birthday Oscar! You turn 25 today!
  birthday("Amarah", 30) // Prints: Happy birthday Amarah! You turn 30 today!
}
```

7.2 Default Parameters

We can give parameters a default value which provides a parameter an automatic value if no value is passed into the function when it's invoked.

```
fun favoriteLanguage(name, language = "Kotlin") {
  println("Hello, $name. Your favorite programming language is $language")
}


fun main() {
  favoriteLanguage("Manon")
  // Prints: Hello, Manon. Your favorite programming language is Kotlin

  favoriteLanguage("Lee", "Java")
  // Prints: Hello, Lee. Your favorite programming language is Java
}
```

7.3 Named Parameters

We can name our parameters when invoking a function to provide additional readability.

To name a parameter, write the parameter name followed by the assignment operator (=) and the parameter value. The parameter's name must have the same name as the parameter in the function being called.

By naming our parameters, we can place parameters in any order when the function is being invoked.

```
fun findMyAge(currentYear: Int, birthYear: Int) {
   var myAge = currentYear - birthYear
   println("I am $myAge years old.")
}

fun main() {
  findMyAge(currentYear = 2020, birthYear = 1995)
  // Prints: I am 25 years old.
  findMyAge(birthYear = 1920, currentYear = 2020)
  // Prints: I am 100 years old.
}
```

7.4 Return Statement

In Kotlin, in order to return a value from a function, we must add a return statement to our function using the `return` keyword. This value is then passed to where the function was invoked.

If we plan to return a value from a function, we must declare the return type in the function header.

```
// Return type is declared outside the parentheses
fun getArea(length: Int, width: Int): Int {
  var area = length * width

  // return statement
  return area
}

fun main() {
  var myArea = getArea(10, 8)
  println("The area is $myArea.") // Prints: The area is 80.
}
```

7.5 Single Expression Functions

If a function contains only a single expression, we can use a shorthand syntax to create our function.

Instead of placing curly brackets after the function header to contain the function's code block, we can use an assignment operator = followed by the expression being returned.

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

fun main() {
  println(fullName("Ariana", "Ortega")) // Prints: Ariana Ortega
  println(fullName("Kai", "Gittens")) // Prints: Kai Gittens
}
```

7.6 Function Literals

Function literals are unnamed functions that can be treated as expressions: we can assign them to variables, call them, pass them as parameters, and return them from a function as we could with any other value.

Two types of function literals are `anonymous functions` and `lambda expressions`.

```
fun main() {
  // Anonymous Function:
  var getProduct = fun(num1: Int, num2: Int): Int {
     return num1 * num2
  }
  println(getProduct(8, 3)) // Prints: 24

  // Lambda Expression
  var getDifference = { num1: Int, num2: Int -> num1 - num2 }
  println(getDifference(10, 3)) // Prints: 7
}
```

A class is an object-oriented concept which resembles a blueprint for individual objects. A class can contain properties and functions and is defined using the `class` keyword followed by a name and optional body.

If the class does not have a body, its curly braces can be omitted.

```
// A class with properties that contain default values
class Student {
  var name = "Lucia"
  var semester = "Fall"
  var gpa = 3.95
}

// Shorthand syntax with no class body
class Student
```

8.1 Class Instances

A new instance of a class is created by calling the class name followed by a pair of parentheses `()` and any necessary arguments.

When creating an instance of a class, we must declare a variable in which we intend to store our instance and assign it equal to the class call.Once the instance has been created, we can use dot syntax to access and retrieve the value of each property.

```
// Class
class Student {
  var name = "Lucia"
  var semester = "Fall"
  var gpa = 3.95
}
```

```
fun main() {
  var student = Student()   // Instance
  println(student.name)     // Prints: Lucia
  println(student.semester) // Prints: Fall
  println(student.gpa)      // Prints: 3.95
```

8.2 Primary Constructor

A primary constructor defines each property value within a class header and allows us to then set unique values when the object is instantiated.

To create a primary constructor using the shorthand syntax, we can follow the name of the class with a pair of parentheses `()` inside of which each property is defined and assigned a data type.

```
class Student(val name: String, val gpa: Double, val semester: String,
val estimatedGraduationYear: Int)

fun main() {
    var student = Student("Lucia", 3.95, "Fall", 2022)
    println(student.name)     // Prints: Lucia
    println(student.gpa)      // Prints: 3.95
    println(student.semester) // Prints: Fall
    println(student.estimatedGraduationYear) // Prints: 2022
}
```

8.3 Init Blocks

The init block gets invoked with every instance that's created and is used to add logic to the class. The `init` keyword precedes any member functions and is followed by a pair of curly braces.

```
class Student(val name: String, val gpa: Double, val semester: String,
val estimatedGraduationYear: Int) {

  init {
    println("$name has ${estimatedGraduationYear - 2020} years left in college.")
  }
}

fun main() {
  var student = Student("Lucia", 3.95, "Fall", 2022)
  // Prints: Lucia has 2 years left in college.
}
```

8.4 Member Functions

A function declared within a class is known as a member function of that class. In order to invoke a member function, we must call the function on an instance of the class.

```
class Student(val name: String, val gpa: Double, val semester: String,
val estimatedGraduationYear: Int) {

  init {
    println("$name has ${estimatedGraduationYear - 2020} years left in college.")
  }

  // Member Function
  fun calculateLetterGrade(): String {
    return when {
      gpa >= 3.0 -> "A"
      gpa >= 2.7 -> "B"
      gpa >= 1.7 -> "C"
      gpa >= 1.0 -> "D"
      else -> "E"
    }
  }
}

// When an instance is created and the function is called,
//the when expression will execute and return a letter grade

fun main() {
  var student = Student("Lucia", 3.95, "Fall", 2022)
  // Prints: Lucia has 2 years left in college.
  println("${student.name}'s letter grade is ${student.calculateLetterGrade()}.")
  // Prints: Lucia's letter grade is A.
}
```