Learning Kotlin.

Shafiqur Rhaman

November 20, 2022

Build command for `kotlin` code compile and Execute.

- Install `OpenJdk` from apt
- Download Kotlin compiler from `github`
- Then add bin folder path to `.bashrc`
- compile command

```
kotlinc basic-operation.kt -include-runtime -d basi-opration.jar
java -jar basi-opration.jar
```

Five Basic Concepts are

- Variables and Types
  - A variable is a location in memory (storage).
  - To indicate the storage area, each variable should be given a unique name (Identifier).
- Control Flows
  - Do something conditionally
  - To repeatedly run code conditionally.
- Functions
  - Functions enable us to separate code.
  - Run code blocks when needed.

- Collections
  - Collections enable us to store multiple elements in one place.
  - Iterate through multiple elements (with the help of control flows).
- Classes and Objects (including inheritance)
  - Create our own data types
  - Keep the data members and methods together in one place.
  - Write more readable and maintainable code.
  - Work better in a team.

Variables and Data Types

- There is two types of variables in `kotlin`
  - Mutable

    ```
    var aquarium = 1
    aquarium = 50
    ```

  - Immutable

    ```
    val fish = "Nemo"
    ```

- Immutable list can not reassign new value but can be manipulate list.

  ```
  val myList = mutalbleListOf("tuna", "salmon", "shark")
  myList.remove("shark")
  ```

- Byte

  ```
  val myByte: Byte = 13
  ```

- Short

  ```
  val myShort: Short = 125
  ```

- Int

  ```
  val myInt: Int = 12345678
  ```

- Long

  ```
  val myLong: Long = 12_123_123_123_1234
  ```

- Float

  ```
  val myFloar: Float = 13.43F
  ```

- Double

  ```
  val myDouble: Double = 3.1233445535445
  ```

- Boolean

  ```
  val myBoolean: Boolean = true
  ```

- Char

  ```
  val myChar: Char = 'A'
  ```

- String

  ```
  val myString: String = "Shafiqur Rahman"
  ```

Arithmetic Operators

- (+, -, /, *, %)
- int divide by int returns int
- float divide by float returns float

  ```
  1/2 // Will return 0
  1.0/20. // Will return 0.5
  ```

- Arithmetic Methods

  ```
  val fish = 2
  fish.times(6)
  fish.div(10)
  fish.plus(3)
  fish.minus(3)
  ```

- Boxing

  ```
  1.toLong()

  val boxed: Number = 1
  boxed.toLong()
  ```

Comparison Operators

- `(==, !=, <, >, <=, >=)`

Null Safety

- Add `?` to indicate variable can contain null value

  ```
  var marble: Int? = null
  var lotsOfFish: List<String?> = listOf(null, null)
  var evenMoreFish: List<String>? = null
  var definitelyFish: List<String?>? = null
  ```

- Force a null able type in `kotlin`

  ```
  goldfish!!.eat()
  ```

- Null check operator `?:` Elvis Operator

  ```
  return fishFoodTreats?.dec() ?: 0
  ```

- Safe Call Operator `?.let`

  ```
  var nullableName: String? = "Pallab"

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

- Chain Null check

  ```
  val age: String? = user?.wife?.age ?: 0
  ```

Switch with When

```
var welcomeMessage = "Hello and welcome to Kotlin"
when (welcomeMessage.length) {
   0, 1 -> println("Nothing to say?")
   in 2..50 -> println("Perfect")
   else -> println("Too long!")
}
```

Array (Collections)

- Typed Array

  ```
  // IntArray
  val numbers = intArrayOf(1, 2, 3)
  // BooleanArray
  // DoubleArray
  val doubles = doublesArrayOf(3.0, 4.0, 5.0)
  // ByteArray
  // LongArray
  // ShortArray
  // FloatArray
  ```

- Non Typed Array

  ```
  // arrayOf<String>
  val school = arrayOf("tuna", "salmon", "shark")
  println(Arrays.toString(school))
  // arrayOf<Fruit>
  val colects = array(1, 2, "Jhon", "Doe", 0.5, Fruit())
  ```

- Arrays of arrays

  ```
  var fish = 12
  var plants = 5

  val swam = listOf(fish, plants)

  val bigSwarm = arrayOf(swam, arrayOf("Dolphin", "Whale", "orka"))
  ```

```
println(Arrays.toString(bigSwarm))
```

- Array comprehension

  ```
  val array = Array(5) { it * 2 }
  println(array.asList())
  ```

- Array of data class

  ```
  data class Fruit(val name: String, val price: Double)

  val fruits = arrayOf(Fruit("Apple", 2.5), Fruit("Grape", 3.5))

  for (fruit in fruits){
      println("${fruit.name}")
  }

  for(index in fruits.indices){
      println("${fruits[index].name} is in index $index")
  }
  ```

Array List

- Array List are used to create a dynamic array. Which means the size of an array can be increased or decreased according to requirement.
- The Array List class provide both read and write functionality
- The Array List follows the sequence of insertion order.
- An array is non synchronized and it may contain duplicate element.
- `ArrayList<E>():` Is used to create an empty Array List.
- `ArrayList(capacity: Int):` Is used to create an Array List of specified capacity.
- `ArrayList(elements: Collection<E>):` Is used to create an Array List filled with the elements of a collection.

- `open fun add(element: E): Boolean` -> used to add the specific element into the collection.
- `open fun clear()` -> used to remove all elements from the collection.
- `open fun get(index: Int): E` -> used to return the element at specific index in the list.
- `open fun remove(element: E): Boolean` -> used to remove a single instance of the specific element from current collection, if it is available.
- Empty Array List

  ```
  fun main() {
      val arrayList = ArrayList<String>()

      arrayList.add("One")
      arrayList.add("Two")

      for(i in arrayList){
  println(i)
      }
  }
  ```

- Array List using collection

  ```
  fun main() {
      val arrayList: ArrayList<String> = ArrayList<String>(5)
      var list: MutableList<String> = mutableListOf<String>()

      list.add("One")
      list.add("Two")

      arrayList.addAll(list)

      val itr = arrayList.iterator()

      while (itr.hasNext()) {
  println(itr.next())
      }
  ```

```
        println("Size of array list = ${arrayList.size}")

    }
```

List (Collections)

```
// List of Strings
val stringList: List<String> = listOf(
    "Denish", "Frank", "Michael", "Greater"
)
// List of Mixed Type
val mixedTypeList: List<Any> = listOf(
    "Denish", 31, 5, "Bday", 70.5, "KG"
)

val months = listOf("January", "February", "March")


val additionalMonths = months.toMutableList()
val newMonths = arrayOf("April", "May", "June")
additionalMonths.addAll(newMonths)
additionalMonths.add("July")

print(additionalMonths)


val days = mutableListOf<String>("Saturday", "Sunday", "Monday")

print(day)
```

Map (Collections)

```
val fruits = setOf("Orange", "Apple", "Mango", "Apple", "Grape", "Orange")
print(fruits.size)

// mapOf key, value

val daysOfTheWeek = mapOf(1 to "Monday", 2 to "Tuesday", 3 to "Wednesday")
```

```
for(key in daysOfTheWeek.keys){
    println("$key is to ${daysOfTheWeek[key]}")
}

data class Fruit(val name: String, val price: Double)

val fruitsMap = mapOf(
    "Favorite" to Fruit("Mango", 2.5),
    "Okay" to Fruit("Apple", 1.0)
)
```

For Loop

- looping without index

  ```
  for (element in swarm) println(element)
  ```

- looping with index

  ```
  for ((index, element) in swarm.withIndex()){
      println("Fish at $index is $element")
  }
  ```

- Ranges print

  ```
  for (i in 'b'..'g') println(i)

  for (i in 1..120) println(i)

  for (i in 5 downTo 1) println(i)

  for (i in 5 downTo 1 step 2) println(i)

  for (i in 3..6 step 2) println(i)

  for (i in 1 until 10) println(i)
  ```

For Each

While Loop

```
var x = 1
while(x <= 10) {
    println("$x")
    x++
}
println("While loop is done.")
```

Do While Loop

```
x = 15

do {
    print("$x")
    x++
} while(x <= 10)
```

Repeat Loop

Filter

- Eager Filter (Create a new list)

  ```
  val decorations = listOf(
      "rock", "pagoda", "plastic plant", "alligator", "flowerpot"
  )
  val eager = decorations.filter { it[0] == 'p'}
  println(eager)
  ```

- Lazy Filter

  ```
  val decorations = listOf(
      "rock", "pagoda", "plastic plant", "alligator", "flowerpot"
  )
  val filtered = decorations.asSequence().filter() { it[0] == 'p' }
  println(filtered)
  println(filtered.toList())
  ```

`lambda`

- A value assigned at compile time, and the value never changes when the variable is accessed.
- a lambda assigned at compile time, and the lambda is executed every time the variable is referenced, returning a different value.

Classes

- Simple way to create a class

```
class Person constructor(_firstName: String, _lastName: String) {
    // Member Variable (Properties) of the class
    var firstName: String
    var lastName: String

    // Initializer Blocks
    init {
this.firstName = _firstName
this.lastName = _lastName
println("First Name: $firstName")
println("Last Name: $lastName")
    }
}

fun main() {
    val pallab = Person("Pallab", "pal")
    println(pallab)
}
```

- More Simple way to create a class

```
class Person(_firstName: String, _lastName: String){
     // Member Variables (Properties) of the class
    var firstName: String = _firstName
    var lastName: String = _lastName

    // Initializer Block
    init {
```

```
println("FirstName = $firstName and LastName = $lastName")
    }
}

fun main() {
    val pallab = Person("Pallab", "pal")
    println(pallab)
}
```

- Even more simple way to create a class

```
class Person(var firstName: String = "Jhon", var lastName: String = "Doe"){
    // Initializer Blocks
    init {
println("First Name: $firstName")
println("Last Name: $lastName")
    }
}

fun main() {
    val pallab = Person("Pallab", "pal")
    println(pallab)
}
```

- With Secondary Constructor

```
class Person (var firstName: String = "Shafiqur", var lastName: String = "Rahman")
    var hobby: String = "Fishing"
    // This property is import for Secondary Constructor
    var age: Int? = null
    // This property is import for Secondary Overload Constructor
    var eyeColor: String? = null

    // Secondary Constructor
    constructor(firstName: String, lastName: String, age: Int): this(firstName, la
this.age = if (age > 0) age else throw IllegalArgumentException("Age must be great
    }

    // Secondary Constructor Overloaded
    constructor(firstName: String, lastName: String, age: Int, eyeColor: String):
    this(firstName, lastName, age)  {
this.eyeColor = eyeColor
    }

    // Method
    fun sayHobby(){
println("$firstName\'s Hobby is $hobby.")
    }
}


fun main() {
    var shafiq = Person()
    shafiq.sayHobby()
    var pallab = Person("Shafiq", "Pallab")
    pallab.sayHobby()
    var dia = Person("Habiba", "Akter", 20)
    dia.hobby = "Planting"
    dia.sayHobby()
}
```

Class Example

```
class Aquarium (
    var width: Int = 20,
    var height: Int = 40,
    var length: Int = 100
) {
    var volume: Int
get() = width * height * length / 1000
set(value) { height = (value * 1000) / (width  * length) }

    var water = volume * 0.9

    // Member Secondary Constructor
    constructor(numberOfFish: Int): this() {
val water: Int = numberOfFish * 2000
val tank: Double = water + (water * 0.1)
height = (tank / (length * width)).toInt()
```

```
    }

    init {
println("Length: $length")
println("Width: $width")
println("Height: $height")
    }
}


fun main() {
    val smallAquarium = Aquarium(numberOfFish = 9)
    println("Volume: ${smallAquarium.volume}")
}
```

SETTERS AND GETTERS

- Kotlin internally generates a default getter and setter for mutable properties,
- `Getter` (only) for read-only properties.
- Example

  ```
  class Car(_brand: String, _model: String, _maxSpeed: Int){
      val _brand: String = _brand
  get() = field

      var _model: String = _model
  get() = field
  set(value) {field = value}

      var _maxSpeed: Int = _maxSpeed
  get() = field
  set(value) {field = value}
  }
  ```

- Backing Field (field)

  ```
  class Car {
  ```

```
    lateinit var owner: String
    val myBrand: String = "BMW"
    // Custom Getter
get() { return field.lowercase() }

    var myModel: String = "M5"
    // Default Setter and Getter
private set


    var myMaxSpeed: Int = 40
get() = field
    // Custom Setter
set(value) {
    field = if(value > 0) value
    else throw IllegalArgumentException("_maxSpeed must be greater than zero")
}

    init {
this.owner = "Shafiq"
    }
}

fun main() {
    val myCar = Car()
    println(myCar.myBrand)
    println(myCar.myModel)
    myCar.myMaxSpeed = 100
    println(myCar.myMaxSpeed)
}
```

Package Visibility

- public - Default Everywhere.
- private - File
- internal - Module

Class Visibility

- public - Default. Class and public member
- private - Inside class. Sub classes `can't` see.
- protected - Inside class. Sub classes can see.
- internal - Module

Inheritance

- We have to add `open` to class to make sub class from it.
- We have to add `override` to sub class to override properties or methods.

```
import kotlin.math.PI

open class BaseAquarium (
    var lenght: Int = 100,
    var width: Int = 20,
    var height: Int = 40
){
    open var volume: Int
get() = (width * height * lenght) / 1000
set(value) { height = (value * 1000) / (width * lenght) }

    open var water = volume * 0.9

    constructor(numberOfFish: Int): this(){
val water: Int = numberOfFish * 2000
val tank: Double = water + (water * 0.1)
height = (tank / (lenght * width)).toInt()
    }
}


class TowerTank(): BaseAquarium() {
    override var water = volume * 0.8
```

```
        override var volume: Int
    get() = ((width * height * lenght) / 1000 * PI).toInt()
    set(value) { height = (value * 1000) / (width * lenght) }
    }


    fun main() {
        var myTowerTank = TowerTank()

        println("Volume of new tower tank aquarium ${myTowerTank.volume}")

    }
```

Abstract Class, Interface, Singleton Object

```
fun main() {
    delegate()
}

fun delegate(){
    val pleco = Plecostomus()
    println("Fish has color ${pleco.color}")
    pleco.eat()
}

interface FishAction{
    fun eat()
}

interface FishColor {
    val color: String
}

class Plecostomus(fishColor: FishColor = GoldColor):
    FishAction by PrintingFishAction("a lot of Food"),
    FishColor by fishColor
```

```
object GoldColor: FishColor {
    override val color = "gold"
}


object RedColor: FishColor {
    override val color = "red"
}


class PrintingFishAction(val food: String): FishAction {
    override fun eat() {
println(food)
    }
}
```