What is Kotlin

Kotlin is a typed programming language, whose datatype of every experession is known at compile time. Kotlin developed by JetBrains. It is interoperable with Java, which means all Java libraries and frameworks could be used in Kotlin, and vice versa. The major usage is for Android application development.

Variables and Data Types

Mutable variables: use **var** keyword

Mutable: variable can be changed to a different value after initialized.

```
var num = 8
println("number = $number")
```

Read-Only Variables: use **val** keyword

Read-only: means variable cann't be reassigned once initialized.

```
val num = 9

num = 10 // throws an exception
```

Basic Data Types : can not be null

Integers: Byte, Short, Int, Long

```
// the type reference are optional, when declare the variables.
val byte: Byte = 127
val short: Short = 32767
val int: Int = 2147483647
val long: Long = 9223372036854775807
```

Floating Point Numbers: Float, Double

```
val float: Float = 3.4028235e38f //with f
val double: Double = 1.7976931348623157e308
```

Text: Char, String

```
val character: Char = '#'
val text: String = "Learning about Kotlin's data types"
```

Booleans: Boolean

```
val yes: Boolean = true
val no: Boolean = false
```

Nullable Types: denoted as Type?

```
val input: String? = null

val output = input.toUpperCase()  // Compile-time error: unsafe call on nullable
object

val output = input?.toUpperCase()  // Works and returns null
```

Elvis Operator: **?:** or else

```
val name: String? = null

val chatName = name ?: "Anonymous"  // Elvis operator
val displayName = chatName.toUpperCase()
```

Unsafe Call Operator: **!!**

```
val input: String? = null

val output = input!!.toUpperCase()
```

Conditions: if-else if-else; when like switch

if-else if-else same as Java

== : Structural euqality != : Structural inequality ===: Referential equality !==: Referential inequality

condition using when

```
when (planet) {
  "Jupiter" -> println("Radius of Jupiter is 69,911km")
```

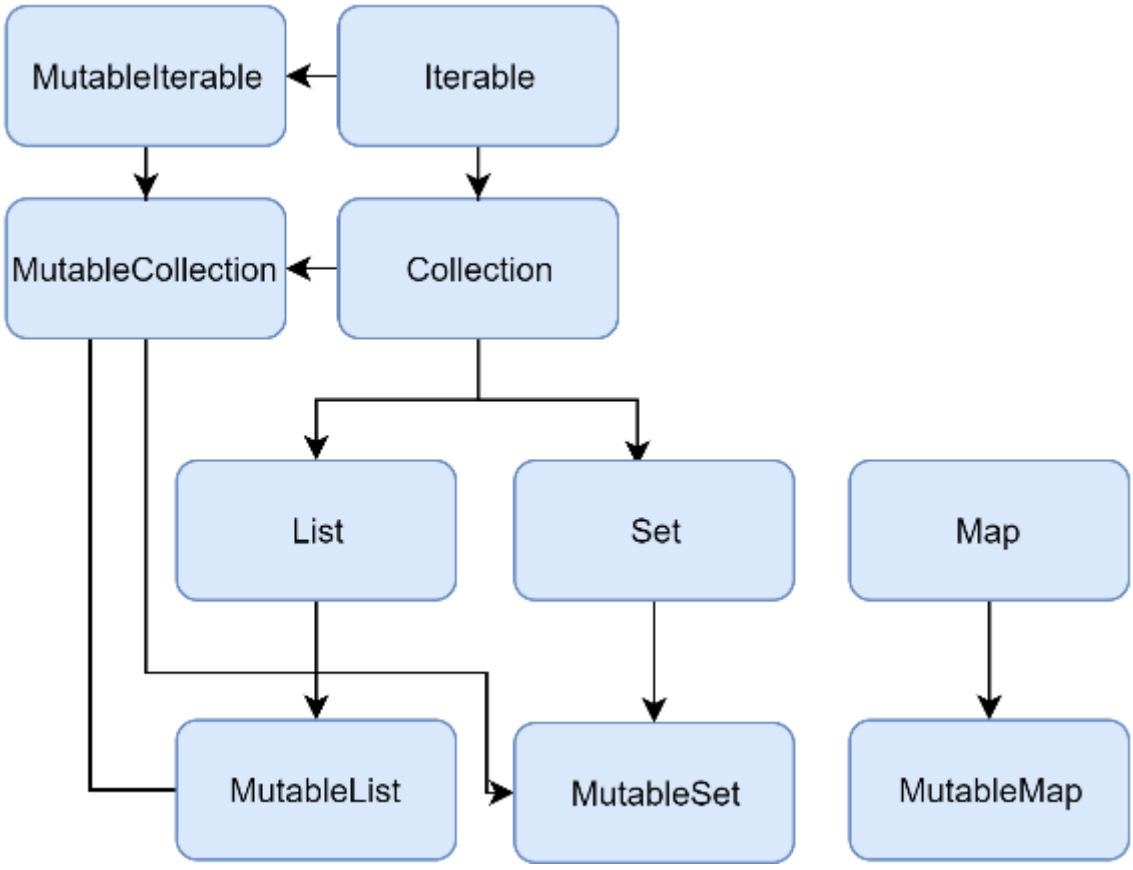
```
    "Saturn"  -> println("Radius of Saturn is 58,232km")
    else      -> println("No data for planet $planet")
}
```

Collections Types: Lists, Sets, Arrays, Maps

Read-Only VS Mutable Lists (Immutability)

	val	**var**
Read-only collection	Canot reassign variable or change values stored in variable("immutable")	Can reassign the variable(to a whole different collection) but can't overwrite the values in the original collection
Mutable collection	Cannot reassign variable but can freely change the values stored in the variable	Can both reassign the variable and change values in the collection it stores

In Kotlin, every collection type has a mutable and read-only variant, as portaryed in the following diagram.



Lists: listOf(), mutableListOf(), arrayListOf()

```
val siUnits = listOf("s", "m", "kg", "A", "K", "mol", "cd")     // Creates
readonly list List<String>
val quarks = mutableListOf("up", "down", "charm", "strange", "top", "bottom") //
Creates mutable list  MutableList<String>
val physicists = arrayListOf("Albert Einstein", "Isaac Newton")      // Creates
```

```
mutable list ArrayList<String>

//Accessing the elements
val siUnit = siUnit[2]            // Gets 3rd element of siUnits
val quark = quarks[0]             // Gets 1st element of quarks
// val physicist = physicists[3]  // Would cause an error because there is no 4th
element in the physicists

//Manipulating Elements
// siUnit[1] = "meter"          // Would cause error because siUnits is readonly
quarks[2] = "CHARM"             // Overwrites 3rd element of quarks
physicists[1] = "Marie Curie"   // Overwrites 2nd element of physicists
```

Sets: no sequences, cannot contain duplicates

```
//create a set
val awards = setOf("World's Best Programmer 2019", "Best Programming Language
2020")
val members = mutableSetOf("Susan", "Jake", "Jenny")
// other APIs
    <!-- hashSetOf(...)
    linkedSetOf(...)
    sortedSetOf(...) -->

//Accessing elements in a set
val members = mutableSetOf("Susan", "Jake", "Jenny")
println(members[0])  // Causes compile-time error

members.add("Greg")
members.remove("Jake")
members.clear()
members.addAll(setOf("Adam", "Eve"))
```

Array: arrayOf()

```
val priorities = arrayOf("HIGH", "MEDIUM", "LOW")
println(priorities[1])
priorities[1] = "NORMAL"
println(priorities[1])
println(priorities.size)
```

Maps

```
val grades = mapOf(
  "Kate" to 3.9,
  "Jake" to 3.4,
  "Susan" to 3.5
)

val grades = mapOf(
  Pair("Kate", 3.9),
  Pair("Jake", 3.4),
  Pair("Susan", 3.5)
)

//creating a mutable map
val enrollments = mutableMapOf(
  "Kate" to listOf("Maths", "Engineering"),
  "Jake" to listOf("CS", "Bioengineering", "Psychology"),
  "Susan" to listOf("Engineering", "Psychology")
)

val marcusClasses = enrollments.getOrDefault("Marcus", emptyList())
println(marcusClasses)   // [], not null

val simonsClasses = enrollments.getOrElse("Simon", {
    /* compute... */
    emptyList()
})
println(simonsClasses)   // [], not null

enrollments.put("Marcus", listOf("Maths", "CS"))

// Can be written more idiomatically as:
enrollments["Marcus"] = listOf("Maths", "CS")

enrollments.remove("Jake")

enrollments["Susan"] = listOf("CS", "Psychology")
```

Loops: while, do{}while(), for loops

```
for (number in 1..5) println(number)        // 1, 2, 3, 4, 5

for (number in 1 until 5) println(number)   // 1, 2, 3, 4

for (number in 1..5 step 2) println(number) // 1, 3, 5

for (number in 5 downTo 1) println(number)  // 5, 4, 3, 2, 1

for (number in 5 downTo 1 step 2) println(number)  // 5, 3, 1
```

```
for (char in 'a'..'c') println(char)        // 'a', 'b', 'c'

for (planet in planets) println(planet)     // "Jupiter", "Saturn", ...

for (char in "Venus") println(char)         // 'V', 'e', 'n', 'u', 's'
```

Functions

Basic Functions

```
//function signatures
fun fibonacci(index: Int): Long

//declaring a function
fun fibonacci(index: Int): Long {
  return if (index < 2) {
    1
  } else {
    fibonacci(index-2) + fibonacci(index-1)  // Calls `fibonacci` recursively
  }
}

//call a function
val output = fibonacci(6)
println(output);
```

The main() function

```
fun main() {
  // Your code here...
}

fun main(args: Array<String>) {
  // Your code here...
}
```

Shorthand Functions

```
fun isValidUsername(username: String): Boolean {
  return username.length >= 3
```

```
}

fun isValidUsername(username: String): Boolean = username.length >= 3
```

Default Values & Named Parameters

```
//Default Parameter Values
fun join(strings: Collection<String>, delimiter: String = ", ") =
strings.joinToString(delimiter)

//Named Parameters
fun join(strings: Collection<String>, delimiter: String = ", ") =
strings.joinToString(delimiter)

fun main() {
  val planets = listOf("Saturn", "Jupiter", "Earth", "Uranus")

  val joined1 = join(planets, delimiter = " - ")
  val joined2 = join(strings = planets)
  val joined3 = join(strings = planets, delimiter = " - ")
  val joined4 = join(delimiter = ", ", strings = planets)

  println(joined1)
  println(joined2)
  println(joined3)
  println(joined4)
}
```

Extension Functions

```
fun Number.print() = println("$this is a Number")
fun Int.print() = println("$this is an Int")

val num: Number = 17
num.print()  // Prints "17 is a Number"
```

Infix Functions:an infix function must have exactly two parameters

```
val str = "Ho "
str.repeat(3)

infix fun Int.times(str: String) = str.repeat(this)
```

```
val greeting = 3 times "Ho "
println(greeting)

<!--
Infix functions must…

    be either member functions or extension functions.
    have exactly one additional parameter in the function signature (in addition
to the extended class, which acts as the first parameter's type).
    not have varargs or default values. -->
```

Operator Functions: define the meaning of well-known symbols such as + or - as other functions.

```
operator fun Int.times(str: String) = str.repeat(this)

val greeting = 3 * "Ho "
```

Reference: Education.io