Mobile Application Development

Week2 The Fundamentals of Kotlin

Yi SUN

Kobe Institute of Computing

The Fundamentals of Kotlin

Basic Kotlin Syntax and Structure

Today, we dive into the essentials of Kotlin, an intuitive language for Android development. From variables and data types to control flows and operators, this guide will solidify your foundation in Kotlin, whether you're a seasoned developer or a beginner in app development.

Kotlin Reference

- Kotlin Documentation
  - https://kotlinlang.org/docs/home.html
- Kotlin Tutorial
  - https://www.tutorialspoint.com/kotlin/index.htm
- Kotlin by Example
  - https://play.kotlinlang.org/byExample/overview

Creating a New Android Project

1. Open Android Studio
2. Select `File > New > New Project`
3. Choose `No Activity` as your template
4. Configure your project:
    - Name: Enter project name
    - Package name: Set package name
    - Language: Select **Kotlin**
    - Minimum SDK: Choose minimum Android version
5. Click `Finish`

Adding Your First kotlin file

open the project directory

- `app/kotlin+java/<package name>/` : Contains Kotlin source files
- Right-click on the `app/kotlin+java/<package name>/` directory
- Select `New > File`
- Enter the file name, e.g., `helloworld.kt`
- Click `OK`

Run the kotlin program in the online platform

- Kotlin Playground
  https://play.kotlinlang.org/

Writing your first Kotlin program

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

Click `Run` button to run the program

What are Variables?

Variables are crucial in Kotlin, allowing you to:

- Store, modify, and manage data
- Hold different data types like numbers, characters, strings, and objects

Creating Variables in Kotlin

Kotlin defines variables with `var` or `val`, the variable name, data type (optional), and value assignment.

```
var variableName: DataType = value // Mutable variable
val constantName: DataType = value // Immutable variable
```

More examples:

```
var age: Int = 30 // Mutable integer variable
val pi: Double = 3.14 // Immutable double variable
var name = "Yi Sun" // Type inferred as String, Mutable
val isAdult = true // Type inferred as Boolean, Immutable
```

`val` :

`val` : Stands for “**value**” and it’s **immutable**, which means once you assign a value to a **val** variable, you cannot change or reassign it.

Preferred when you have a variable whose value shouldn’t change once initialized, like **constants** or **properties** that should remain unchanged.

```
val pi = 3.14 // An immutable variable
// pi = 3.14159 // This would cause a compilation error
```

`var` :

`var` : Is mutable, meaning after you assign an initial value, you can change or reassign that variable to a new value as many times as you want.

Used when you anticipate the value of **a variable will change**, like counters in a loop or **a value being updated** based on user input.

```
var counter = 0 // A mutable variable
counter = 1 // Modifying the value of the variable
```

Datatypes

In programming, you work with various types of data, such as numbers, text, or true/false values. In Kotlin, data types act as labels that inform the computer about the kind of data you're handling, helping it process the data correctly. When you create a variable, you specify its data type (like number, text, or true/false), and once set, this type remains fixed.

Integers (`Int` and `Long`)

Description: Integer types can hold **whole numbers**, both positive and negative. The most commonly used integer type is `Int` .For larger integer values, `Long` can be used.

Syntax and Examples:

```
val age: Int = 25
val largeNumber: Long = 10000000000L
```

Floats and Doubles (Float and Double)

Description: Floats and Doubles are used to represent decimal numbers. Double has higher precision and is generally used as the default for decimal numbers.

Syntax and Examples:

```
val pi: Double = 3.14
val floatNumber: Float = 2.73F
```

Characters (Char)

Description: Characters represent a single character.

```
val letter: Char = 'A'
```

Special characters start from an escaping backslash `¥` .

`¥t` : Tab	`¥r` : Carriage return
`¥b` : Backspace	`¥"` : Double quote
`¥n` : Newline	`¥'` : Single quote
`¥$` : dollar sign	`¥¥` : Backslash

Strings (String)

Description: Strings represent text data. In Kotlin, strings can be created using double quotes.

```
val name: String = "John Smith"
val multiLineText: String = """
    This is a multi-line
    string in Kotlin
    It preserves formatting
"""
```

String templates:

```
val name = "John"
println("Hello, $name") // Using $variableName
val age = 25
println("Next year, ${name} will be ${age + 1}") // Using ${expression}
```

String Formatting

To format a string to your specific requirements, use the String.format() function.

```
// Formats an integer, adding leading zeroes to reach a length of seven characters
val integerNumber = String.format("%07d", 31416)
println(integerNumber)
// 0031416

// Formats a floating-point number to display with a + sign and four decimal places
val floatNumber = String.format("%+.4f", 3.141592)
println(floatNumber)
// +3.1416

// Formats two strings to uppercase, each taking one placeholder
val helloString = String.format("%S %S", "hello", "world")
println(helloString)
// HELLO WORLD
```

Boolean (Boolean)

Description: Boolean type represents true/false values.

```
val isStudent: Boolean = true
val isEmployed: Boolean = false
val result = 5 > 3 // evaluates to true
println(result)
```

Logical Operators

- `||` (Logical OR): Returns true if at least one condition is true.
- `&&` (Logical AND): Returns true only if both conditions are true.
- `!` (Logical NOT): Negates the value; turns true into false and vice versa.

```
val a = true
val b = false
println(a || b) // true
println(a && b) // false
println(!a) // false
```

Arrays

An array is a data structure that holds a fixed number of values of the same type or its subtypes. The most common type of array in Kotlin is the object-type array, represented by the Array class.

Create Arrays

To create arrays in Kotlin, you can use:

- functions, such as `arrayOf()` , `arrayOfNulls()`

```
val simpleArray = arrayOf(1, 2, 3)
val nullArray: Array<Int?> = arrayOfNulls(3)  //Nullable Integer Array
```

- Use the `Array` constructor to create an array of a specific size.

```
// Creates an Array<Int> that initializes with zeros [0, 0, 0]
val initArray = Array<Int>(3) { 0 }
println(initArray.joinToString())
// 0, 0, 0

// Creates an Array<String> with values ["0", "1", "4", "9", "16"]
val asc = Array(5) { i -> (i * i).toString() }
asc.forEach { print(it) }
// 014916
```

Nested arrays

```
// Creates a two-dimensional array
val twoDArray = Array(2) { Array<Int>(2) { 0 } }
println(twoDArray.contentDeepToString())
// [[0, 0], [0, 0]]

// Creates a three-dimensional array
val threeDArray = Array(3) { Array(3) { Array<Int>(3) { 0 } } }
println(threeDArray.contentDeepToString())
// [[[0, 0, 0], [0, 0, 0], [0, 0, 0]], [[0, 0, 0], [0, 0, 0], [0, 0, 0]], [[0, 0, 0], [0, 0, 0], [0, 0, 0]]]
```

Access and modify elements

```
val simpleArray = arrayOf(1, 2, 3)
val twoDArray = Array(2) { Array<Int>(2) { 0 } }

// Accesses the element and modifies it
simpleArray[0] = 10
twoDArray[0][0] = 2

// Prints the modified element
println(simpleArray[0].toString()) // 10
println(twoDArray[0][0].toString()) // 2
```

Common Array Operations:

```
// Find element
val hasTwo = numbers.contains(2)
val indexOfThree = numbers.indexOf(3)

// Transform array
val doubled = numbers.map { it * 2 }.toTypedArray()
val filtered = numbers.filter { it > 3 }.toTypedArray()

// Sort array
numbers.sort() // In-place sorting
val sorted = numbers.sorted() // Returns new sorted list
val descendingSorted = numbers.sortedDescending()
```

Note: Some operations like `map` and `filter` return a List by default. Use `toTypedArray()` to convert back to an Array if needed.

Control Flow: if Statement

```
val score = 85
if (score >= 90) {
    println("You got an A.")
}
val score = 75
if (score >= 90) {
    println("You got an A.")
} else {
    println("You got a B.")
}
val score = 65
if (score >= 90) {
    println("You got an A.")
} else if (score >= 80) {
    println("You got a B.")
} else {
    println("You got a C.")
}
```

If Expression

In Kotlin, if is an expression and can return a value.

```
// Basic usage
val max = if (a > b) {
    println("Choosing a")
    a  // return value
} else {
    println("Choosing b")
    b  // return value
}

// Used as expression
val min = if (a < b) a else b
```

When Expression

Description: Similar to switch in other languages, but more powerful.

```
val score = 85
val grade = when {
    score >= 90 -> "A"
    score >= 80 -> "B"
    score >= 70 -> "C"
    score >= 60 -> "D"
    else -> "F"
}
```

When with parameter

```
when (x) {
    1 -> println("x is 1")
    2, 3 -> println("x is 2 or 3")
    in 4..10 -> println("x is between 4 and 10")
    else -> println("x is neither 1 nor 2")
}
```

For Loop

```
// Range iteration
for (i in 1..5) {
    println(i) // prints 1 to 5
}

// Collection iteration
val fruits = listOf("apple", "banana", "orange")
for (fruit in fruits) {
    println(fruit)
}

// Iteration with index
for ((index, value) in fruits.withIndex()) {
    println("$index: $value")
}
```

While and Do-While Loops

```
// while loop
val items = listOf("apple", "banana", "kiwifruit")
var index = 0
while (index < items.size) {
    println("item at $index is ${items[index]}")
    index++
}

// do-while loop
do {
    println("Executed at least once")
} while (false)
```

Functions

Description: Functions are blocks of reusable code.

```
// Basic function declaration
fun sayHello(name: String): String {
    return "Hello, $name!"
}

// Single-expression function
fun double(x: Int) = x * 2

// Default parameters
fun greet(name: String = "Guest") = "Hello, $name!"

// Named parameters
fun createUser(name: String, age: Int, isStudent: Boolean) {
    // function body
}
// Calling with named parameters
createUser(name = "Alice", age = 20, isStudent = true)
```

Null Safety

Kotlin's type system distinguishes between nullable and non-nullable types.
In Kotlin, the type system distinguishes between types that can hold null (nullable types) and those that cannot (non-nullable types). For example, a regular variable of type String cannot hold null:

```
// Assigns a non-null string to a variable
var a: String = "abc"
// Attempts to re-assign null to the non-nullable variable
a = null
print(a)
// Null can not be a value of a non-null type String
```

Null Safety: Declaring Nullable Variables

To allow null values, declare a variable with a `?` sign right after the variable type. For example, you can declare a nullable string by writing `String?`. This expression makes String a type that can accept null:

```
// Assigns a nullable string to a variable
var b: String? = "abc"
// Successfully re-assigns null to the nullable variable
b = null
print(b)
// null
```

Collections: List, Set, Map

The following collection types are relevant for Kotlin:

- **List** is an ordered collection with access to elements by indices – integer numbers that reflect their position. Elements can occur more than once in a list.
- **Set** is a collection of unique elements. It reflects the mathematical abstraction of set: a group of objects without repetitions. Generally, the order of set elements has no significance.
- **Map** (or **dictionary**) is a set of key-value pairs. Keys are unique, and each of them maps to exactly one value. The values can be duplicates. Maps are useful for storing logical connections between objects.

List

`List<T>` stores elements in a specified order and provides indexed access to them. Indices start from zero – the index of the first element – and go to `lastIndex` which is the `(list.size - 1)`.

```
val numbers = listOf("one", "two", "three", "four")
println("Number of elements: ${numbers.size}")
println("Third element: ${numbers.get(2)}")
println("Fourth element: ${numbers[3]}")
println("Index of element ¥"two¥" ${numbers.indexOf("two")}")
```

Set

`Set<T>` stores unique elements; their order is generally undefined. `null` elements are unique as well: a `Set` can contain only one `null`. Two sets are equal if they have the same size, and for each element of a set there is an equal element in the other set.

```
val numbers = setOf(1, 2, 3, 4)
println("Number of elements: ${numbers.size}")
if (numbers.contains(1)) println("1 is in the set")

val numbersBackwards = setOf(4, 3, 2, 1)
println("The sets are equal: ${numbers == numbersBackwards}")
```

Map

`Map<K, V>` is not an inheritor of the `Collection` interface; however, it's a Kotlin collection type as well. A `Map` stores **key-value** pairs (or **entries**); keys are unique, but different keys can be paired with equal values. The Map interface provides specific functions, such as access to value by key, searching keys and values, and so on.

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)

println("All keys: ${numbersMap.keys}")
println("All values: ${numbersMap.values}")
if ("key2" in numbersMap) println("Value by key ¥"key2¥": ${numbersMap["key2"]}")
if (1 in numbersMap.values) println("The value 1 is in the map")
if (numbersMap.containsValue(1)) println("The value 1 is in the map")
```

Collections: types

Kotlin provides both immutable(read-only) and mutable collections.

```
// List
val readOnlyList = listOf(1, 2, 3)
val mutableList = mutableListOf(1, 2, 3)

// Set
val readOnlySet = setOf(1, 2, 3)
val mutableSet = mutableSetOf(1, 2, 3)

// Map
val readOnlyMap = mapOf("a" to 1, "b" to 2)
val mutableMap = mutableMapOf("a" to 1, "b" to 2)
```

MutableList<T>

`MutableList<T>` is a `List` with list-specific write operations, for example, to add or remove an element at a specific position.

```
val numbers = mutableListOf(1, 2, 3, 4)
numbers.add(5)
numbers.removeAt(1)
numbers[0] = 0
numbers.shuffle()
println(numbers)
```

MutableSet<T>

`MutableSet<T>` is a `Set` with set-specific write operations, for example, to add or remove an element.

```
val numbers = mutableSetOf(1, 2, 3, 4)
numbers.add(5)
numbers.remove(1)
println(numbers)
```

MutableMap<K, V>

`MutableMap<K, V>` is a `Map` with map-specific write operations, for example, to add or remove a key-value pair.

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)
numbersMap.put("three", 3)
numbersMap["one"] = 100
println(numbersMap)
```

Difference between Array and List in Kotlin

Feature	Array	List
Size	Fixed	Variable
Access	Direct access, efficient	Can be accessed directly, but efficiency might be lower than arrays
Operations	Basic operations	Rich operations, supports various functional operations
Use Cases	For fixed-size data and frequent element access	For data with variable size and various operations

When to Use Array or List?

- **Array:**
  - When storing fixed-size data.
  - When frequent element access is required.
  - When performance is a high priority.
- **List:**
  - When storing data with a variable size.
  - When dynamic add, remove, and update operations are needed.
  - When code readability and conciseness are preferred.

User Input

Getting input from the user is essential in programming **when you want your application to interact with the user** by receiving data that the user provides. In Kotlin, you can receive user input from the console using the standard library functions, making your programs interactive and dynamic.

If you want to ask the user for input and display it, you can use the combination of `println()` to show a message and `readln()` to get the user's response.

```
println("Please enter your name:")
val name = readln()
println("Hello, $name!")
```

Practice program: Rock Paper Scissors Game

Task Description

Create a command-line Rock Paper Scissors game where a player competes against the computer.

Requirements

1. Player can input their choice (1 for Rock, 2 for Scissors, 3 for Paper)
2. Computer randomly generates its choice
3. Game displays both choices and determines the winner
4. Player can exit the game by entering 0
5. Invalid inputs should be handled appropriately

Game Rules

- Rock beats Scissors
- Scissors beats Paper
- Paper beats Rock
- Same choices result in a tie

Technical Requirements

- Language: Kotlin
- Interface: Command-line
- Input: Standard input (keyboard)
- Output: Text display of choices and results

Expected Output Example

```
Please enter your choice (1 for Rock, 2 for Scissors, 3 for Paper, 0 to exit):
1
You chose Rock
The computer chose Paper
You lose!
```