Kotlin

Kotlin is a **modern**, **statically-typed programming language** that runs on the **Java Virtual Machine (JVM)** and can also be compiled to **JavaScript** or **native code**.

- Developed by **JetBrains** (the makers of IntelliJ IDEA)
- Officially supported for **Android app development** by **Google** 2017
- Designed to be **concise**, **safe** and **interoperable with Java**

Key Features

- Concise syntax
- Null Safety built-in
- Fully interoperable with Java
- Coroutines for lightweight concurrency
- Multi-platform support (JVM, Android, Web, Native)

Variables

var : *Mutable* reference. The value can change during runtime. Its value can be reassigned after declaration.

val : *Immutable* reference. You cannot reassign a value once assigned. Read-only variables.

</> Immutable variables (val) - values cannot change

```
val drinkC ount: Int // ✓ Must specify Type since no initializer
drinkCount = 12  // ✓ Assigned later before usage
printl n(d rin kCount) // Output: 12
drinkCount = 15 // ✕ Error: Val cannot be reassigned
val popcor nBoxes = 5 // ✓ Kotlin infers Int type
val hotdog Count: Int = 7 // ✓ Explicitly declared as Int
```

</> Mutable variable (var) - value can change

```
var x = 5 // Mutable variable
x += 1 // ✓ Increments x by 1
println(x) // Output: 6.
```

lateinit

- Allows initializing a non-null variable later
- Used when you cannot initialize a variable at the time of declaration, often for classes or Android views.
- Must be a var (NOT val)
- Must be a non-primitive type (no Int, Double, etc.)
- Must be initialized before use

</> Basic Syntax

```
lateinit var variab leName: Type
</> lateinit var count: Int // ✕ Compile-time error
</> lateinit var count: String // ✓ Valid
```

Nullable types

- A nullable type means a variable can hold a null value.
- Enhances null safety, reducing runtime crashes due to NullPointerException (NPE) from Java.
- Must explicitly allow a variable to be null by adding **?** to its type.

```
</> var studen tName: String? = " Sar ah"
studen tName = null // ✓ Allowed because of the "?" in String?
</> var teache rName: String = " Joh n"
teache rName = null // ✕ Error: Null can not be a value of a nonnull type String.
```

Safe Call

- **The Safe Call Operator ?.** allows you to **safely access** properties or methods **only if the variable is NOT null**
- If the variable is **null**, the call **returns null** instead of throwing a crash.

</> Basic Syntax

```
 nullab leV ari abl e?.m et hod OrP roperty
</> var name: String? = " Sar ah"
printl n(n ame ?.l ength) // Output: 5
name = null
printl n(n ame ?.l ength) // Output: null (no crash!)
</> studen t?.s ch ool ?.a ddr ess ?.city // Chain safe calls. Each ?. checks at every level safely
```

Elvis operator

The Elvis Operator ?: is used to **provide a default value** when an expression on the left is **null**.

If the left side is **not null**, it returns the left side.

If the left side is **null**, it returns the right side (the default).

</> Basic Syntax

```
val result = nullab leV ariable ?: defaul tValue
</> var name: String? = " Sar ah"
val finalName = name ?: " Unk now n"
printl n(f ina lName) // Output: Sarah
</> var name: String? = null
val finalName = name ?: " Unk now n"
printl n(f ina lName) // Output: Unknown
```

Classes

A **class** is a **blueprint** for creating objects (instances) with **properties** and **functions** (methods).

</> Basic Syntax

```
class User (
var firstName: String,
var lastName: String,
var address: String? = null
)
</> class Car(val brand: String, var speed: Int) {
fun drive() {
 printl n("D riving $brand at $speed km/h")
 }
}
</> val myCar = Car("To yot a", 120)
myCar.d rive() // Output: Driving Toyota at 120 km/h
```

Data classes

A **Data Class** is a special class designed to **hold data**.

To declare a data class, use the keyword **data**.

Kotlin **automatically generates** useful **methods** for you: toString() equals(), hashCode(), copy()...

</> Basic Syntax

```
data class ClassN ame(val prop1: Type, val prop2: Typ
)
</> data class User(val name: String, val age: Int)
val u = User("S ara h", 25)
println(u) // Output: User(name=Sarah, age=25)
val u2 = u.copy(age = 26)
printl n(u2)
// Output: User(name=Sarah, age=26)
```

Must have at least one property inside primary constructor.

Properties should be val or var Otherwise not allowed.

Collections

Groups of related elements you can store, manage, and manipulate

List: Ordered collection, allows duplicates. **listOf(1, 2, 2, 3)**

Set: Unordered, unique elements. **setOf(1, 2, 3)**

Map: Key-value pairs. **mapOf("name" to "Sarah")**

☐ **Mutable Collections:** Collections that allow modifications (add/remov after creation. Useful when the data structure needs to change dynamic
✓ **mutableListOf, mutableMapOf, mutableSetOf()**

</> Mutable list with explicit type declaration

```
val shapes: Mutabl eLi st< Str ing> = mutabl eLi stO
, " squ ")
shapes.ad d("c ir")
printl n(s hapes) // [tri, squ, cir]
```

☐ **Immutable Collections:** Immutable collections that cannot be modifie creation.

Collections (cont)

They provide safety by ensuring your data remains unchanged.
☐ **listOf, mapOf, emptyListOf, emptyMapOf**

</> **Read-only list**

```
val readOn lyS hapes = listOf ("tr ian gle ", " squ
 are ")
printl n(r ead Onl ySh apes) // [triangle, square]
printl n("First item is: ${read Onl ySh ape s[0 ]}")
// triangle
```

.

Common Collection Operations

Add	fruits.add("Mango")
Remove	fruits.remove("Banana")
Loop List	for (item in list) {}
Loop Map	for ((k,v) in map) {}
Contains	list.contains(2)
Size	list.size

Conditional expressions

In Kotlin, many control structures like if, when are **expressions**, **they return a value** (not just perform actions like in Java or C++).
You can assign the result of an if or when directly to a variable!
if :
</> `val max = if (a > b) a else b` // if (a > b) returns a, otherwise b, and assigns the result to max. No need to create extra variables manually.
There is **no ternary operator** condition ? then : else in Kotlin.
when :

```
val result = when (score) {
 in 90..100 -> " Exc ell ent "
 in 75..89 -> " Goo d"
 in 60..74 -> " Pas s"
 else -> " Fai l"
```

Conditional expressions (cont)

} // All branch conditions are checked sequentially until one of them is satisfied. So only the first suitable branch is executed.

Loops

for : Iterate over a range of values,array, list, etc and perform an action.
</> **Iterating a list**

```
val fruits = listOf ("Ap ple ", " Ban ana ", " Che
 rry ")
for (fruit in fruits) {
    printl n(f ruit) }
```

</> **Iterating a range**

```
for (i in 1..5) {
println(i) // Prints 1 2 3 4 5}
```

While : To execute a code block while a conditional expression is true.

```
var x = 5
while (x > 0) {
println(x)
 x-- } // Repeats while x > 0.
```

do...while : To execute the code block first and then check the conditional expression.

```
var y = 0
do {
println(y)
y++
} while (y < 3) // Runs at least once, even if condition is false after first execution.
```

Functions

- A function groups reusable code.
- Can be defined in a class (method) or directly in a package.
- You can declare your own functions in Kotlin using **fun** keyword.

</> Basic Syntax

```
fun functi onN ame (pa ram1: Type, param2: Type): Return
Type
{
// function body
}
```

```
</> fun add(a: Int, b: Int): Int {
return a + b
} // add(2, 3) returns 5.
</> fun multip ly(a: Int, b: Int) = a * b
// No {} or return needed if it's one expression.
```

Lambda expressions

- A **Lambda** is an **anonymous function**, a “function literal”.
- A function that is not declared, but passed immediately as an expression.
- **Lambda expressions** can be assigned to variables, passed as arguments, or returned from functions.

</> **Basic Syntax**

```
{ parame ter1: Type, parame ter2: Type -> body }
// {} : Lambda block
// -> : Separates parameters from body
```

</> **Simple Lambda**

```
val greet = { printl n("H ello, World! ") }
greet() // Prints "Hello, World!"
```

</> **Lambda with parameters**

Lambda expressions (cont)

```
val add = { a: Int, b: Int -> a + b }
printl n(a dd(3, 5)) // Output: 8
```

</> **Lambda passed to a function**

```
fun calc(a: Int, b: Int, op: (Int, Int) -> Int): Int
return op(a, b)
}
val res = calc(2, 3) { x, y -> x * y }; printl n(res)
// 6
```

it keyword : When there's only one parameter, it is used automatically.

```
</> val square: (Int) -> Int = { it * it }
printl n(s qua re(4)) // Output: 16. No need to name the parameter explicitly if there's only one.
```