

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 drinkCount: Int // ✓ Must specify Type since no initializer
drinkCount = 12 // ✓ Assigned later before usage
```

```
println(drinkCount) // Output: 12
```

```
drinkCount = 15 // ✗ Error: Val cannot be reassigned
```

```
val popcornBoxes = 5 // ✓ Kotlin infers Int type
```

```
val hotdogCount: 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 variableName: 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 studentName: String? = "Sarah"
```

```
studentName = null // ✓ Allowed because of the "?" in String?
```

```
</> var teacherName: String = "John"
```

```
teacherName = 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

```
nullableVariable?.methodName() Or Property
```

```
</> var name: String? = "Sarah"
```

```
println(name?.length) // Output: 5
```

```
name = null
```

```
println(name?.length) // Output: null (no crash!)
```

```
</> student?.school?.address?.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 ?: default tValue
</> var name: String? = " Sarah"
val finalName = name ?: " Unknown"
println(f ina lName) // Output: Sarah
</> var name: String? = null
val finalName = name ?: " Unknown"
println(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() {
        println("Driving $brand at $speed km/h")
    }
}
</> val myCar = Car("Toyota", 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: Type)
</> data class User(val name: String, val age: Int)
val u = User("Sarah", 25)
println(u) // Output: User(name=Sarah, age=25)
val u2 = u.copy(age = 26)
println(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/remove) after creation. Useful when the data structure needs to change dynamically.
 - ✓ **mutableListOf, mutableMapOf, mutableSetOf()**

</> Mutable list with explicit type declaration

```
val shapes: MutableList<String> = mutableListOf("square", "square")
shapes.add("circle")
println(shapes) // [tri, square, circle]
```

- ☐ **Immutable Collections**: Immutable collections that cannot be modified after 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 ")
println (r ead Onl ySh apes) // [triangle, square]
println ("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) {
    println (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.
```



By **Bochrak**

cheatography.com/bochrak/

Not published yet.

Last updated 27th April, 2025.

Page 3 of 4.

Sponsored by **CrosswordCheats.com**

Learn to solve cryptic crosswords!

<http://crosswordcheats.com>

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 functionName (parameter1: Type, parameter2: Type): ReturnType {
    // function body
}

</> fun add(a: Int, b: Int): Int {
    return a + b
} // add(2, 3) returns 5.

</> fun multiply(a: Int, b: Int) = a * b
// No {} or return needed if it's one expression.
```

Lambda expressions (cont)

```
val add = { a: Int, b: Int -> a + b }
println(add(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 }; println(res)
// 6
```

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

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

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

```
{ parameter1: Type, parameter2: Type -> body }
// {} : Lambda block
// -> : Separates parameters from body
```

</> Simple Lambda

```
val greet = { println("Hello, World! ") }
greet() // Prints "Hello, World!"
```

</> Lambda with parameters

