Kotlin Quick Reference

Null safety, coroutines, data classes, functional programming essentials

Basics

Hello World

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

Variables

```
val name = "Kotlin"        // immutable (prefer)
var count = 0               // mutable
val pi: Double = 3.14159   // explicit type
const val MAX = 100         // compile-time constant
```

Basic Types

`Int, Long`	32-bit / 64-bit signed integers
`Double, Float`	64-bit / 32-bit floating point
`Boolean`	**`true`** / **`false`**
`Char`	Single Unicode character
`String`	Immutable text, supports templates
`Unit`	Equivalent to **`void`** (single value)
`Nothing`	Function never returns (e.g., throws)

String Templates

```
val name = "World"
println("Hello, $name!")
println("Length: ${name.length}")
val raw = """line 1
    |line 2""".trimMargin()
```

Functions

Function Declaration

```
fun add(a: Int, b: Int): Int {
    return a + b
}
fun add(a: Int, b: Int) = a + b  // single expression
```

Default & Named Arguments

```
fun greet(name: String, greeting: String = "Hello") {
    println("$greeting, $name!")
}
greet("Alice")                  // Hello, Alice!
greet("Bob", greeting = "Hi")   // Hi, Bob!
```

Higher-Order Functions

```
fun operate(a: Int, b: Int, op: (Int, Int) -> Int): Int {
    return op(a, b)
}
val sum = operate(3, 4) { a, b -> a + b }
```

Varargs

```
fun sum(vararg nums: Int): Int = nums.sum()
sum(1, 2, 3)
val arr = intArrayOf(1, 2, 3)
sum(*arr)                       // spread operator
```

Classes

Class Definition

```
class Person(val name: String, var age: Int) {
    fun greet() = "Hi, I'm $name"
}
val p = Person("Alice", 30)
println(p.name)
```

Inheritance

```
open class Shape(val sides: Int) { open fun area(): Double = 0.0 }
class Circle(val r: Double) : Shape(0) {
    override fun area() = Math.PI * r * r
}
```

Visibility Modifiers

`public`	Visible everywhere (default)
`private`	Visible within the class / file
`protected`	Class and subclasses
`internal`	Same module only

Abstract & Interfaces

```
interface Drawable { fun draw() }
abstract class Widget : Drawable { abstract val label: String }
class Button(override val label: String) : Widget() {
    override fun draw() = println("Drawing $label")
}
```

Null Safety

Nullable Types

```
var name: String? = null     // nullable
val len = name?.length       // safe call: null
val len2 = name?.length ?: 0 // Elvis operator: 0
val len3 = name!!.length     // assert non-null (throws)
```

Safe Operations

`?.`	Safe call — returns null if receiver is null
`?:`	Elvis — default value when null
`!!`	Non-null assertion (throws if null)
`?.let { }`	Execute block only if non-null
`as?`	Safe cast — returns null on failure

Smart Casts

```
if (obj is String) println(obj.length)  // auto-cast
when (obj) {
    is Int    -> println(obj + 1)
    is String -> println(obj.uppercase())
}
```

Collections

Creating Collections

```
val list = listOf(1, 2, 3)         // immutable
val mList = mutableListOf(1, 2, 3) // mutable
val map = mapOf("a" to 1, "b" to 2)
val set = setOf("x", "y", "z")
```

Collection Operations

```
val nums = listOf(1, 2, 3, 4, 5)
nums.filter { it > 2 }             // [3, 4, 5]
nums.map { it * 2 }                // [2, 4, 6, 8, 10]
nums.firstOrNull { it > 3 }        // 4
nums.sumOf { it }                  // 15
```

Common Operations

`.filter { }`	Keep elements matching predicate
`.map { }`	Transform each element
`.flatMap { }`	Map and flatten
`.groupBy { }`	Group by key into Map
`.sortedBy { }`	Sort by selector
`.associate { }`	Transform to Map (key-value pairs)
`.any { } / .all { }`	Check if any/all match predicate
`.fold(init) { }`	Reduce with initial accumulator

Coroutines

Basic Coroutine

```
import kotlinx.coroutines.*
fun main() = runBlocking {
    launch { delay(1000); println("World") }
    println("Hello")
}
```

Async / Await

```
val deferred = async { fetchData() }
val result = deferred.await()
// parallel: launch multiple async, await all
val (a, b) = awaitAll(async { fetchA() }, async { fetchB() })
```

Coroutine Builders

`launch { }`	Fire-and-forget coroutine (returns Job)
`async { }`	Returns Deferred<T> with result
`runBlocking { }`	Bridges blocking and suspending code
`withContext(dispatcher)`	Switch coroutine context
`coroutineScope { }`	Structured concurrency scope

Dispatchers

`Dispatchers.Default`	CPU-intensive work (thread pool)
`Dispatchers.IO`	Blocking I/O operations
`Dispatchers.Main`	Main/UI thread (Android, Swing)
`Dispatchers.Unconfined`	Starts in caller thread, resumes in any

Extensions

Extension Functions

```
fun String.isPalindrome(): Boolean {
    return this == this.reversed()
}
println("racecar".isPalindrome())  // true
```

Extension Properties

```
val String.wordCount: Int
    get() = this.split("\\s+".toRegex()).size
println("hello world".wordCount)   // 2
```

Operator Overloading

```
data class Vec(val x: Double, val y: Double) {
    operator fun plus(other: Vec) = Vec(x + other.x, y + other.y)
}
val v = Vec(1.0, 2.0) + Vec(3.0, 4.0)  // Vec(4.0, 6.0)
```

Data Classes

Data Class

```
data class User(val name: String, val age: Int)
val u1 = User("Alice", 30)
val u2 = u1.copy(age = 31)        // non-destructive copy
val (name, age) = u1               // destructuring
```

Auto-Generated Members

`equals()`	Structural equality based on properties
`hashCode()`	Consistent with **`equals()`**
`toString()`	**`User(name=Alice, age=30)`**
`copy()`	Create modified copy
`componentN()`	Destructuring support

Kotlin Quick Reference

Enum Classes

```
enum class Direction { NORTH, SOUTH, EAST, WEST }
val dir = Direction.NORTH
when (dir) { Direction.NORTH -> "up"; else -> "other" }
```

Sealed Classes

Sealed Class Hierarchy

```
sealed class Result<out T> {
    data class Success<T>(val data: T) : Result<T>()
    data class Error(val message: String) : Result<Nothing>()
    data object Loading : Result<Nothing>()
}
```

Exhaustive When

```
fun handle(result: Result<String>): String = when (result) {
    is Result.Success -> result.data
    is Result.Error   -> "Error: ${result.message}"
    is Result.Loading -> "Loading..."
}   // no else needed — compiler checks exhaustiveness
```

Sealed vs Enum

`Sealed class`	Subclasses can hold different state
`Sealed interface`	Allows multiple inheritance
`Enum class`	Fixed set of singleton instances
`data object`	Singleton with **`toString()`** override

Scope Functions

Scope Function Comparison

`let`	Context as **`it`**, returns lambda result
`run`	Context as **`this`**, returns lambda result
`with(obj)`	Context as **`this`**, returns lambda result
`apply`	Context as **`this`**, returns context object
`also`	Context as **`it`**, returns context object

let & apply

```
val name: String? = "Alice"
name?.let { println("Name is $it") }
val person = Person("Bob", 25).apply {
    age = 26                    // configure object
}
```

run & with

```
val result = "Hello".run { uppercase() + " WORLD" }
val info = with(person) { "$name is $age years old" }
```

also

```
val numbers = mutableListOf(1, 2, 3)
    .also { println("Original: $it") }
    .also { it.add(4) }
// also is useful for side effects (logging, validation)
```