Kotlin Cheatsheet

A concise reference for Kotlin syntax, keywords, and common patterns, designed to accelerate development and provide quick solutions.

Kotlin Basics

Variables

`val` (Immutable)	Read-only variable, value assigned once. `val name: String = "Kotlin"`

`var` (Mutable)	Mutable variable, value can be changed.
	`var age: Int = 30`
	`age = 31`

Type Inference	Kotlin can often infer variable types.
	`val message = "Hello"`
	`// message is inferred as String`

Nullable Types	Variables can be declared as nullable using `?`. `var nullableString: String? = null`

Lateinit	For non-null vars that are initialized later.
	`lateinit var subject: String`
	
	`@BeforeEach`
	`fun setup() {`
	`subject = "Kotlin"`
	`}`

Constants	Compile-time constants must be declared at the top level or as members of `object` declarations or companion objects. `const val API_KEY = "your_api_key"`

Functions

Basic Function	`fun add(a: Int, b: Int): Int {`
	`return a + b`
	`}`

Single-Expression Function	`fun multiply(a: Int, b: Int): Int = a * b`

Default Arguments	`fun greet(name: String = "World") {`
	`println("Hello, $name")`
	`}`

Named Arguments	`greet(name = "Kotlin")`

Varargs	`fun printAll(vararg messages: String) {`
	`for (m in messages) println(m)`
	`}`
	
	`printAll("Hello", "Kotlin", "World")`

Control Flow

`if` statement:

```
val a = 10
val b = 20
val max = if (a > b) a else b
```

`when` expression:

```
val x = 1
when (x) {
    1 -> println("x == 1")
    2 -> println("x == 2")
    else -> println("x is different from 1 and 2")
}
```

`for` loop:

```
val numbers = listOf(1, 2, 3)
for (number in numbers) {
    println(number)
}
```

`while` loop:

```
var i = 0
while (i < 5) {
    println(i)
    i++
}
```

Classes and Objects

Classes

Basic Class	`class Person(val name: String, var age: Int)`

Constructors	Primary and secondary constructors.
	`class Rectangle(val width: Int, val height: Int) {`
	`constructor(side: Int) : this(side, side)`
	`}`

Data Classes	Automatically generates `equals()`, `hashCode()`, `toString()`, `copy()`. `data class User(val name: String, val id: Int)`

Inheritance	Classes are `final` by default; use `open` to allow inheritance.
	`open class Shape`
	`class Circle : Shape()`

Abstract Classes	`abstract class AbstractClass {`
	`abstract fun doSomething()`
	`}`

Objects

Object Declaration (Singleton)	`object Database {`
	`fun connect() { ...`
	`}`
	`}`

Companion Objects	Like static members in Java.
	`class MyClass {`
	`companion object {`
	`fun create(): MyClass = MyClass()`
	`}`
	`}`

Interfaces

```
interface Clickable {
    fun click()
    fun showOff() = println("I'm clickable!")
// Default implementation
}

class Button : Clickable {
    override fun click() {
        println("Button was clicked")
    }
}
```

Collections and Functional Programming

Collections

`List` (Immutable)	`val list = listOf(1, 2, 3)`
`MutableList`	`val mutableList = mutableListOf(1, 2, 3)` `mutableList.add(4)`
`Set` (Immutable)	`val set = setOf(1, 2, 3)`
`MutableSet`	`val mutableSet = mutableSetOf(1, 2, 3)` `mutableSet.add(4)`
`Map` (Immutable)	`val map = mapOf(1 to "one", 2 to "two")`
`MutableMap`	`val mutableMap = mutableMapOf(1 to "one", 2 to "two")` `mutableMap[3] = "three"`

Functional Operations

`filter`:

```
val numbers = listOf(1, 2, 3, 4, 5)
val evenNumbers = numbers.filter { it % 2 == 0 }
// evenNumbers is [2, 4]
```

`map`:

```
val numbers = listOf(1, 2, 3)
val squaredNumbers = numbers.map { it * it }
// squaredNumbers is [1, 4, 9]
```

`forEach`:

```
val numbers = listOf(1, 2, 3)
numbers.forEach { println(it) }
```

`reduce`:

```
val numbers = listOf(1, 2, 3)
val sum = numbers.reduce { acc, i -> acc + i }
// sum is 6
```

`fold`:

```
val numbers = listOf(1, 2, 3)
val product = numbers.fold(1) { acc, i -> acc * i }
// product is 6
```

`any` and `all`:

```
val numbers = listOf(1, 2, 3, 4, 5)
val hasEven = numbers.any { it % 2 == 0 } // true
val allEven = numbers.all { it % 2 == 0 } // false
```

Coroutines

Basic Usage

`launch` Fire and forget coroutine.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
    delay(2000L) // keeps the JVM
alive
}
```

`async` Coroutine that returns a result using `await()`.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val deferred = async {
        delay(1000L)
        "World!"
    }
    println("Hello,")
    println(deferred.await())
}
```

`runBlocking` Bridge between non-coroutine world and coroutine world.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    println("Hello, Coroutines!")
}
```

Coroutine Scope

`CoroutineScope`:

```
import kotlinx.coroutines.*

class MyClass: CoroutineScope by MainScope() {

    fun doSomething() {
        launch {
            delay(1000)
            println("Done")
        }
    }

    fun destroy() {
        cancel() // Cancels all coroutines
launched in this scope
    }
}
```

Suspending Functions

Marked with `suspend` keyword.

```
suspend fun fetchData(): Data {
    delay(1000) // Simulate network request
    return Data()
}
```