Kotlin

Core syntax, null safety, and collections.

01 Run a Kotlin program

```
fun main() {
    val name = "Maya"
    println("Hello, $name!")
    println("Kotlin on the JVM")
}
```

Save as Main.kt. Compile: kotlinc Main.kt -include-runtime -d app.jar; run java -jar app.jar. Later snippets belong inside main unless a complete program is shown. Each stands alone.

02 Variables & types

```
val name: String = "Maya"
var count: Int = 3
count += 1
val price: Double = 19.95
val ready: Boolean = true
println("$name $count $price $ready")
```

Types can be inferred. val prevents reassignment; it does not make an object immutable. var permits reassignment. Long uses an L suffix; Float uses f. Char holds one UTF-16 code unit.

03 Operators & conversions

```
val whole = 7 / 2
val ratio = 7.toDouble() / 2
val rest = 7 % 2
val ok = whole > 0 && rest == 1
println("$whole $ratio $rest $ok")
```

Integer division truncates. Use explicit conversions such as toLong(). == compares equality; === compares references. &&, ||, and ! work with Boolean. + - * / % are arithmetic operators.

04 Strings & templates

```
val name = " Maya ".trim()
val greeting = "Hello, $name"
println("${greeting.uppercase()}!")
println(name.contains("ay"))
println("a,b,c".split(","))
println(name.length)
```

Strings are immutable. $name and ${expression} insert values. Triple quotes create raw strings. length counts UTF-16 code units, not displayed characters.

05 Null safety

```
val input: String? = "42"
val length = input?.length ?: 0
val number = input?.toIntOrNull()
if (number != null) println(number + 1)
println(length)
```

? permits null; ?. safely accesses; ?: defaults. A checked stable value is smart-cast. toIntOrNull handles invalid input; !! throws on null.

06 if & when expressions

```
val score = 82
val pass = if (score >= 60) "Yes" else "No"
val grade = when {
    score >= 90 -> "A"
    score >= 60 -> "Pass"
    else -> "Try again"
}
```

if and when return values. when expressions must be exhaustive; else handles the rest. when (value) tests one subject. Conditions need Boolean.

07 Loops & ranges

```
for (n in 1..3) { println(n) }
for (n in 0 until 3) { println(n) }
for (n in 6 downTo 2 step 2) { println(n) }
var left = 2
while (left > 0) { left-- }
println(left)
```

.. includes the end; until excludes it. downTo descends; step sets the increment. break exits, continue skips ahead. do/while runs at least once.

08 Functions & named args

```
fun greet(name: String = "Guest"): String {
    return "Hello, $name"
}
fun add(a: Int, b: Int) = a + b
println(greet(name = "Maya"))
println(add(2, 3))
```

Parameters need types; return types follow :. Expression bodies use = and can infer the return type. Arguments can be named or defaulted. Unit means no useful result.

09 Arrays & lists

```
val fixed = arrayOf(1, 2)
val names = listOf("Maya", "Leo")
val nums = mutableListOf(10, 20)
nums.add(30)
println("${fixed[0]} ${names.getOrNull(8)}")
println(nums)
```

Arrays have fixed size. List is read-only; MutableList permits changes. Read-only is not deep immutability. getOrNull checks bounds; invalid indexing throws.

Kotlin

Data models and everyday standard APIs.

10 Sets & maps

```
val tags = setOf("web", "web", "api")
val ages = mutableMapOf("Maya" to 28)
ages["Leo"] = 31
println("web" in tags)
println(ages["Maya"])
println(ages.getOrDefault("Sam", 0))
```

Sets keep unique values. to builds a pair; maps associate keys and values. Missing keys return null. mutableSetOf/mutableMapOf permit changes.

11 Lambdas & transformations

```
val nums = listOf(1, 2, 3, 4)
val twice = { n: Int -> n * 2 }
val result = nums.filter { it % 2 == 0 }
    .map(twice)
println(result)
println(nums.any { it > 3 })
```

Lambdas use { parameters -> body }; it names one implicit parameter. filter selects, map transforms, any tests. These operations are eager and leave nums unchanged.

12 Classes & properties

```
class Counter(var value: Int = 0) {
    fun add() { value++ }
}
val counter = Counter()
counter.add()
println(counter.value)
```

Constructor val/var parameters become properties. init initializes. Classes/methods are final by default; open permits inheritance/overriding.

13 Data classes

```
data class User(val name: String,
                val age: Int)
val user = User("Maya", 28)
val older = user.copy(age = 29)
val (name, age) = older
println("$name $age")
```

Primary-constructor properties supply equals, hashCode, toString, copy, and destructuring. copy is shallow: referenced mutable objects remain shared.

14 Interfaces

```
interface Named { fun name(): String }
class User : Named {
    override fun name() = "Maya"
}
fun main() {
    println(User().name())
}
```

Complete program. : lists supertypes; override implements a member. Interfaces can provide method bodies, but cannot store backing-field state.

15 Extension functions

```
fun String.shout() = uppercase()
val name = "Maya"
println(name.shout())
println("hello".shout())
println(name)
```

An extension adds call syntax without changing a class. The receiver is available as this. Resolution uses the declared receiver type; extensions cannot access private members.

16 Scope functions

```
val names = mutableListOf<String>().apply {
    add("Maya")
    add("Leo")
}
println(names.let { it.size })
```

apply uses this and returns the receiver. let uses it and returns the lambda result. value?.let { ... } skips null. Avoid confusing nesting.

17 Checks & exceptions

```
val age = -1
try {
    require(age >= 0) { "Invalid age" }
} catch (e: IllegalArgumentException) {
    println(e.message)
}
```

require checks arguments; check checks state. Failure throws. try can return a value; finally handles cleanup. No checked-exception declarations are required.

18 Read a text file

```
val text = try {
    java.io.File("notes.txt").readText()
} catch (e: java.io.IOException) {
    println("Read failed: ${e.message}")
    return
}
println(text)
```

JVM: provide notes.txt or handle failure. readText loads UTF-8 and closes the file. useLines closes after processing; writeText replaces contents.