Kotlin JVM language and compiler

START HERE

`kotlinc <file>.kt -include-runtime -d <out>.jar`
Compile to a jar

`java -jar <out>.jar`
Run the jar

`kotlinc -script <file>.kts`
Run a script

`val <name> = <value>`
Read-only variable

`var <name>: <Type>`
Mutable variable

`fun <name>(<p>: <T>): <R>`
Declare a function

`data class <Name>(val <p>: <T>)`
Value holder

`<x>?.<member> ?: <default>`
Null-safe fallback

hello.kt source → `kotlinc` → **hello.jar** bytecode → `-include-runtime` → **JVM** java -jar → **Output** stdout

CONCEPTS

Types are non-null by default: String cannot hold null, String? can.

Most Kotlin projects build through Gradle; kotlinc is for single files and scripts.

COMPILER AND RUNNER

`kotlinc <file>.kt -d <dir>`
Compile to class files

`kotlinc <file>.kt -include-runtime -d <out>.jar`
Self-contained jar
Without -include-runtime the jar needs the Kotlin stdlib on the classpath

`java -jar <out>.jar`
Run a self-contained jar

`kotlin -classpath <out>.jar <Name>Kt`
Run a main class
A file hello.kt compiles to the class HelloKt

`kotlinc -script <file>.kts`
Run a Kotlin script

`kotlinc -script <file>.kts -- <args>`
Pass arguments to a script

COMPILER AND RUNNER (cont.)

`kotlinc -Xrepl`
Interactive shell

`kotlinc -cp <path> <file>.kt`
Set the classpath

`kotlinc -version`
Print compiler version

VARIABLES AND TYPES

`val <n>: <T> = <v>`
Read-only value

`var <n> = <v>`
Mutable variable, inferred type

`const val <N> = <v>`
Compile-time constant

`"Hi $<name> ${<expr>}"`
String template

`"""<text>"""`
Raw multiline string

`<x> as <T>` / `<x> as? <T>`
Cast / safe cast

`<x> is <T>`
Type check with smart cast

`<a>..<b>` / `<a> until <b>`
Inclusive / exclusive range

`val <n> by lazy { <init> }`
Compute on first use

NULL SAFETY

`<T>?`
Nullable type

`<x>?.<member>`
Safe call

`<x> ?: <default>`
Elvis: fallback if null

`<x>!!`
Assert non-null, may throw
Prefer ?. or ?: ; !! throws NullPointerException

`<x>?.let { <block> }`
Run only if not null

`<x> ?: return`
Early exit on null

FUNCTIONS AND LAMBDAS

`fun <n>(<p>: <T> = <d>): <R> { }`
Function with default value

`fun <n>(<p>: <T>) = <expr>`
Expression body

`<f>(<name> = <v>)`
Named argument

`{ <x> -> <expr> }`
Lambda

`it`
Implicit single lambda parameter

`fun <T>.<n>(): <R>`
Extension function

FUNCTIONS AND LAMBDAS (cont.)

`(<A>) -> <B>`
Function type

CLASSES AND OBJECTS

`class <N>(val <p>: <T>)`
Class with primary constructor

`data class <N>(val <p>: <T>)`
equals, copy, toString for free

`<obj>.copy(<p> = <v>)`
Copy with one field changed

`val (<a>, <b>) = <obj>`
Destructuring

`object <N> { }`
Singleton

`companion object { }`
Class-level members

`interface <N> { }`
Interface

`open class <N>`
Allow subclassing
Classes are final by default

`sealed class <N>` / `sealed interface <N>`
Closed type hierarchy

`enum class <N> { <A>, <B> }`
Enumeration

CONTROL FLOW

`if (<c>) <a> else <b>`
Conditional expression

`when (<x>) { <v> -> <r>; else -> <d> }`
Pattern-style switch

`for (<i> in <range>) { }`
Loop over range or iterable

COLLECTIONS

`listOf(<a>, <b>)`
Read-only list

`mutableListOf(<a>)`
Mutable list

`mapOf(<k> to <v>)`
Read-only map

`<c>.map { <expr> }`
Transform each element

`<c>.filter { <cond> }`
Keep matching elements

`<c>.first { <cond> }` / `<c>.firstOrNull { <cond> }`
Find element

SCOPE FUNCTIONS

`<x>.let { <it> -> <r> }`
Transform, returns lambda result

`<x>.apply { <this> }`
Configure, returns object

COROUTINES

`suspend fun <n>()`
Function that can suspend

`runBlocking { <block> }`
Bridge into blocking code

`launch { <block> }`
Start a coroutine, no result

`async { <expr> }.await()`
Start and await a result

RECIPES

Hello world as a jar

```
# hello.kt
fun main() = println("Hello")

kotlinc hello.kt -include-
runtime -d hello.jar
java -jar hello.jar
```

Quick script

```
# list.kts
java.io.File(".").listFiles()?.
forEach { println(it.name) }

kotlinc -script list.kts
```

Sealed result with when

```
sealed interface Result
data class Ok(val v: Int) :
Result
data object Fail : Result

fun show(r: Result) = when (r)
{
  is Ok -> "ok ${r.v}"
  Fail -> "failed"
}
```