Kotlin Part 1: Introduction

CS 346 Application Development

Introduction

What is Kotlin? Why do we use it?

Why Kotlin?

There are literally hundreds of programming languages to choose from.

How do you pick a programming language?

- Does it offer features and capabilities that you require?
- How mature is the ecosystem around the language — do you have useful libraries and tools support to use it effectively?
- How productive can you be with it - are there books, tutorials, videos etc.?
- Is it suitable for the type of software + environment in which you are working?

"Why can't we just build everything in C++?"
— every C++ developer, ever

Low-level languages are suitable when you are concerned with the performance of your software, and when you need to control resources.

- They are often used for **systems programming** i.e., delivering code that runs as fast as possible and uses as little memory as possible.
- Examples of systems languages include C, C++, and Rust.
- Appropriate domains include device drivers, and game engines.

High-level languages are suitable when you are concerned with the speed of development, extensibility, and the robustness of your solution.

- **Applications programming** leans heavily on high-level languages, trading some performance for desirable programming language features.
- Examples of application languages include Swift, Kotlin, Go, and Dart.
- Appropriate domains include web, mobile/desktop applications, servers.

Kotlin

kotlinlang.org

Kotlin is a modern, general-purpose language.

- Compares favourably to Swift, Dart, Scala i.e. other high-level languages.

Developed for this exact purpose.

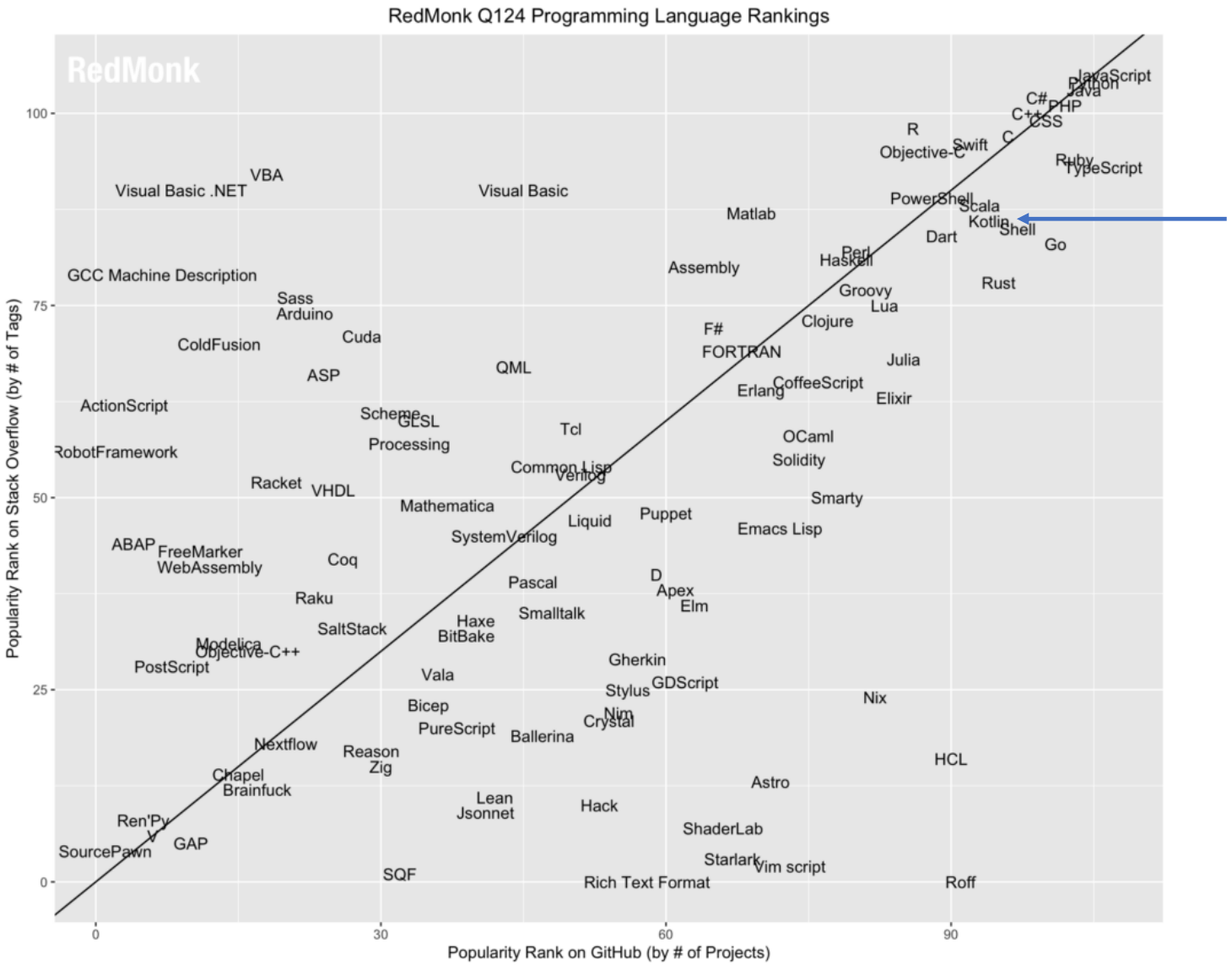
- Imperative, object-oriented, functional styles (*hybrid language*).
- Statically-typed; type inference; NULL safety; automatic memory management/GC; concurrency support.

It's multi-platform.

- Android, iOS, Desktop (Windows, Linux, Mac)
- Future: JS (beta), WASM/web (alpha), and more.

It has broad library support for graphics, networking, databases...

- Compose (UI), Ktor (Networking), Exposed (DB) and others.


RedMonk Q124 Programming Language Rankings
RedMonk
Popularity Rank on Stack Overflow (by # of Tags)
Popularity Rank on GitHub (by # of Projects)
0
25
50
75
100
0
30
60
90
JavaScript
Python
Java
C#
PHP
C++
CSS
R
Objective-C
Swift
C
Ruby
TypeScript
PowerShell
Scala
Kotlin
Shell
Dart
Go
Matlab
Perl
Haskell
Assembly
Rust
Groovy
Lua
Clojure
F#
FORTRAN
Julia
CoffeeScript
Erlang
Elixir
OCaml
Solidity
Smarty
Emacs Lisp
Puppet
Liquid
Mathematica
Common Lisp
Verilog
Tcl
Scheme
GESL
Processing
Racket
VHDL
SystemVerilog
Pascal
D
Apex
Elm
Smalltalk
Haxe
BitBake
SaltStack
Raku
Coq
ABAP
FreeMarker
WebAssembly
Modelica
Objective-C++
PostScript
Gherkin
Vala
Stylus
GDScript
Nix
Bicep
Nim
Crystal
PureScript
Ballerina
Nextflow
Reason
Zig
HCL
Chapel
Brainfuck
Astro
Lean
Jsonnet
Hack
Ren'Py
ShaderLab
GAP
SourcePawn
Starlark
Vim script
SQF
Rich Text Format
Roff
Sass
Arduino
Cuda
ColdFusion
ASP
QML
ActionScript
RobotFramework
VBA
Visual Basic .NET
Visual Basic
GCC Machine Description

Installation

Command-line/editor (*not* recommended)

- Install Java 21 *or later* from Azul or a similar site.
- Install Kotlin from the Kotlin landing page
- Install VS Code + Kotlin extensions
  - Install LSP from GitHub (alpha!)

Only really useful for snippets.

IDE (recommended!)

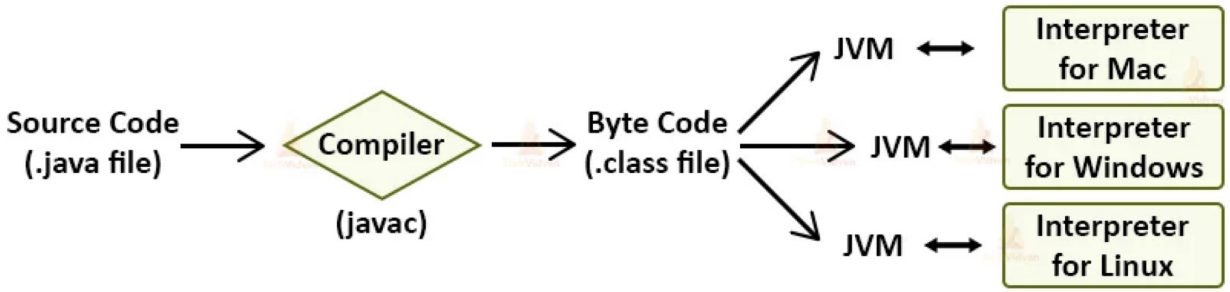
- IntelliJ IDEA: install from jetbrains.com/idea
  - Community edition is free for non-commercial use.
  - As a student, you can get a free ultimate license.
- Android Studio: install from developer.android.com
  - For Android development, you need the.

Better for complex/mobile projects.

Compiling Code (JVM)

The `kotlinc` compiler consists of multiple backend compilers:

- Kotlin/Native compiles Kotlin code to native binaries. It includes an LLVM based backend and a native implementation of the Kotlin `stdlib`. Works for Windows, macOS, Linux, Android, (iOS, JS, wasm).
- Kotlin/Android compiles Kotlin code to native Android binaries. ★
- Kotlin/JVM compiles Kotlin code to bytecode, usually for desktop. ★



Kotlin/JVM compiler generates bytecode that executes in a native JVM.

How to learn Kotlin

Attend lectures, tinker with the sample code.

Read some of the excellent online resources!

- Online resources (free)
  - Kotlin Documentation: official documentation.
  - Dave Leeds on Kotlin: online version of the book below. ★
  - Kotlin Youtube channel: official videos.
- Books (not free)
  - Elizarov, Isakova, Aigner, Jemerov. Kotlin in Action. 2nd ed. Manning. ★
  - Dave Leeds. 2025. Kotlin Illustrated. Typealias. ★

Getting Started

What does a Kotlin program actually look like?

Getting started

Kotlin source files must end with a .kt extension. A source file can contain:

- Top-level functions (like an imperative language)
- Class definitions (like an object-oriented language)
- Global variable declarations

Programs require a single main method to serve as the entry point:

```
fun main(args: Array<String>) {
  println("Hello World")
}
```

NOTE: `args` is optional. Arrays have methods, so you can check `args.size()`.

Tiny but valid program

```
// All of this is in a file named `hello.kt`
// No header files, no real restrictions.

fun main() {
  println(hello(“Jeff”)
}

fun hello(name: String): String {
  return “Hello, $name!”
}

// To compile and run it

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

Hello, Jeff!
```

```
// fun main() is the entry point
// println() prints to the console

// fun hello() defines a function
// return just returns a value

// kotlinc compiles the program
// java runs the program
```

Type System

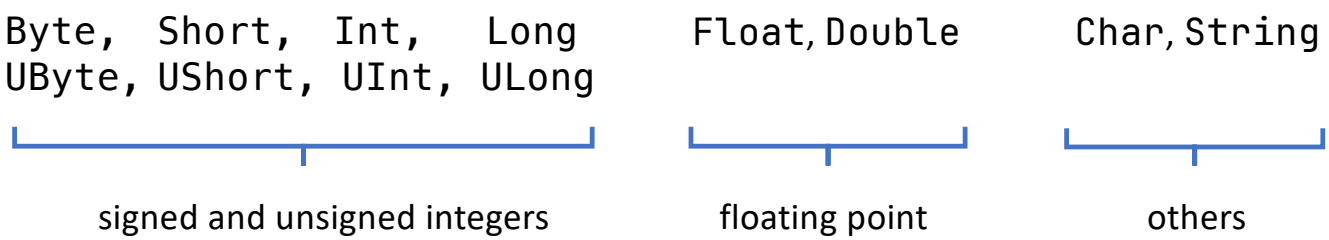
It’s a good a place to start as any…

Type systems

Kotlin is a statically typed language.

- **Static typing**: variables need to be declared before use. e.g. C++, Java, Kotlin, TS. Types are verified at compile time! This eliminates runtime type errors.
- **Strong typed**: stricter typing rules enforced at compile-time. e.g. Java, C++, Kotlin.

Basic-types:



Type declaration

The `var` keyword is used to declare variables.

- The type follows the name.
- Variables are mutable (i.e., they can be reassigned after definition).

```
var pi: Float = 3.14 // initial definition
pi = 3.1415926       // reassignment is allowed
```

Kotlin supports type inference at compile-time, if it's unambiguous.

```
var pi = 3.14       // Double (not Float)
var name = "Jeff"   // String
```

Mutability

Declaration keywords indicate *mutability*.

- **var**: the value of the variable can be changed (it's *mutable*).
- **val**: the variable cannot be changed (it's *immutable*)

```
var pi: Float = 3.14            // ok
pi = 3.1415926                  // ok

val e: Float = 2.718            // ok
pi = 2.71828                    // compilation error
```

> Best practice: use `val` as much as you can!

NULL Safety

NULL is the absence of a value.

- You cannot operate on a NULL as you would a regular type.
- If a type system allows NULL as a type value, then you need to be very careful to ensure that you are not accidentally operating on a NULL.
- Checks are usually done at *runtime*
  - Explicitly checking return values for NULL
  - Catching exceptions.

NULL values can cause runtime exceptions and crash your application.

- *Tony Hoare: "NULL was my billion-dollar mistake".*

NULL Safety

Kotlin has semantics for dealing with NULL & checks at compile-time.

By default, types cannot be NULL.

? suffix indicates a NULLABLE type.

```
var length1: Int = null            // Not nullable so can't assign null value.
var length2: Int? = null           // Nullable so can be assigned null value.
```

If a type is nullable, evaluations *must* handle the null case.

```
var name: String? = "Jeff"         // name could be null
if (name != null) println(name)    // handles null case
```

NULL Syntax

We have special syntax to make dealing with NULL values a little easier.

?. is the “safe call operator”. Can only be invoked if not null.

```
var name: String? = null
val len = name?.length                              // len == null
val len = if (name != null) name.length else 0      // len == 0
```

?: is a ternary operator for NULL (“elvis operator”)

```
val len = name?.length ?: 0                         // nicer syntax!
```

https://pl.kotl.in/Fd8L89CPb

Functions

Function syntax and arguments.

Functions

```
// 1. No parameters
fun hello() {
    println("Hello World")
}
```

```
// 2. Parameter list, return type
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

Function body

```
// 3. Parameter list, return type
fun sum(a: Int, b: Int): Int = a + b
```

```
// 4. Parameter list, type inferred
fun sum(a: Int, b: Int) = a + b
```

Function expression (tidy!)

Default Arguments

We can supply default values for parameters.
A parameter with a default value is optional for the caller.

```
fun mult(a: Int, b: Int = 2): Int {
    return a * b
}

fun main() {
    println(mult(1))        // a=1, b defaults to 2; 1x2=2
    println(mult(5,2))      // a=5, b=2; 5x2=10
}
```

https://pl.kotl.in/GW424Hz-q

Named Arguments

You can (optionally) provide argument names when you call a function.
If you do this, you can change the calling order!

```
fun repeat(s: String = "*", n: Int = 1):String {
    return s.repeat(n)
}

fun main() {
    println(repeat())            // prints '*' using both defaults
    println(repeat("#"))         // prints '#' using default n==1
    println(repeat("=", 3))      // prints '===' positionally
    println(repeat(n=5, s="#")) // prints '#####' using named arguments
}
```

https://pl.kotl.in/vBjdzjDmf

Control Flow

Range of control flow options…

Standard Control Flow

Traditional control flow is supported

- `if... then.. else`
- `while, do... while`
- `break, continue`

New!

- `when // replaces switch`
- `for (s in collection) // iteration`
- `for (a in 1.. 5) // iteration up through range`
- `for (a in 5 downTo 1) // iteration down through range`

if… then

if… then has both statement and expression forms.

```
// statement
if (a > b) {
    println(a)
} else {
    println(b)
}

// expression
val max = if (a > b) a else b
```

for…

A `for` loop iterates through anything that provides an iterator (e.g., the built-in collection classes).

```
val items1 = listOf("apple", "banana", "kiwifruit")
for (item in items1) {
  println(item)
}

val items2 = listOf("apple", "banana", "kiwifruit")
for (index in items2.indices) {
  println("item at $index is ${items2[index]}")
}
```

https://pl.kotl.in/D8boDJXak

Ranges

```
for(i in 15..18) {
  println(i) // 15 16 17 18
}

for (i in 5 downTo 1 step 2) {
  println(i) // 5 3 1
}

val low=1
val high=10
if (num in low..high) {
  println("The number ${num} is between ${low} and ${high}")
}
```

https://pl.kotl.in/vyJZoPZAV

when

`when` is an improved switch statement.

```
val x = 13
val validNumbers = listOf(11,13,17,19)

when (x) {
  0, 1 -> print("x == 0 or x == 1")
  in 2..10 -> print("x is in the range")
  in validNumbers -> print("x is valid")
  !in 10..20 -> print("x is outside the range")
  else -> print("none of the above")
}
```

https://pl.kotl.in/CzlL_j4zz

when

`when` as an expression

```
val x = 13
val response = when {
    x < 0 -> "negative"
    x >= 0 && x <= 9 -> "small"
    x >=10 -> "large"
    else -> "how do we get here?"
}
println(response)
```

https://pl.kotl.in/IEf9RSRB0

Collections

Built-in collection classes.

Collections

- A collection is a group of some variable number of items (possibly zero) of the same type. Objects in a collection are called elements.
- Kotlin provides mutable and immutable interfaces to these collections.

List	An ordered collection of objects.
Pair	A tuple of two values.
Triple	A tuple of three values.
Set	An unordered collection of objects.
Map	An associative dictionary of keys and values.
Array	Indexed, fixed-size collection of object or primaries - rarely used

List

A list is an ordered collection of objects.

```
// immutable (due to listOf)
var fruits = listOf( "advocado","banana","cantaloupe")
println(fruits.get(0)) // advocado
println(fruits[1]) // banana
// fruits.add("dragon fruit") // unresolved, since immutable

// mutable (due to mutableListOf)
var mutableFruits = mutableListOf("advocado","banana")
mutableFruits.add("cantaloupe") // this works!
println(mutableFruits.last())
```

https://pl.kotl.in/DcUwgxGWx

Pair

A pair is a tuple of two values.

```
val ns = Pair("Halifax Airport", "YHZ")
println(ns) // (Halifax Airport, YHZ)
```

The contents of Pair are NOT mutable, since this is a data class whose contents aren't expected to change. `copy` to duplicate with a modified value.

```
// characters.second = "Jennifer" // error!!
val characters2 = characters.copy(second = "Jennifer")
println(characters2) // (Tom, Jennifer)
```

https://pl.kotl.in/1uWdBModj

Map

A map is an associative dictionary of key and value pairs (i.e. it maps one value to another).

```
// immutable (initialize with pairs)
val imap = mapOf(1 to "x", 2 to "y", 3 to "z")
println(imap) // {1=x, 2=y, 3=z}
// imap.put(4, "q") // immutable, so unresolved reference

// mutable
val mmap = mutableMapOf(5 to "x", 6 to "y")
mmap.put(7,"z") // ok
println(mmap) // {5=x, 6=y, 7=z}
```

https://pl.kotl.in/FcT0DJrsP

Accessors

Kotlin has special properties that can be used to access data elements in collections.

```
val list = listOf("one", “two", "three", "four")
list.contains("four")) // true

// slice - extract into a new collection
list.slice(1..2) // [two, three]
list.slice(0..2 step 2) // [one, three]

// take - extract n elements
list.take(3) // [one, two, three]
list.takeLast(2) // [three, four]
```

https://pl.kotl.in/TQL-o3RYI

Accessors

```
// extract using iterators
list.first { it.length > 3 } // [three]
list.last { it.startsWith("o") } // [one]

// iterate over map
for ((k, v) in imap) {
  println("$k = $v")
}
// alternate syntax
imap.forEach { k, v -> println("$k = $v") }
```

Reference

- Dave Leeds. 2025. Dave Leeds on Kotlin. Online.
- Dave Leeds. 2025. **Kotlin: An Illustrated Guide**. TypeAlias Studios LLC. ISBN 979-8992796605.
- JetBrains. 2025. Kotlin Documentation. Online.
- Roman Elizarov, et al. 2024. **Kotlin in Action**. 2nd edition. Manning Publications. ISBN 9781617299605.