Introduction to Kotlin

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

Contents

What is Kotlin? 2
- History 3
- Features 4

Getting Started 6
- Toolchain 7
- Hello, World! 8
- Variables 9
- Strings 10
- Collections 11
- Null Safety 14
- Functions 20
- Expressions 22
- Loops 25
- Ranges 27

Bibliography 28

What is Kotlin?

History

Kotlin was created by JetBrains in 2011.

- They have a lot of experience with Java language/ecosystem and wanted to contribute something substantial.
- Kotlin was originally created as a drop-in replacement for Java/JVM [1].

At the same time, Google wanted to update Android development. Their platform was Java-based, and increasingly complex and fragile.

- Their internal developers started using Kotlin.
- Adopted as the “official” language for Android in 2019.

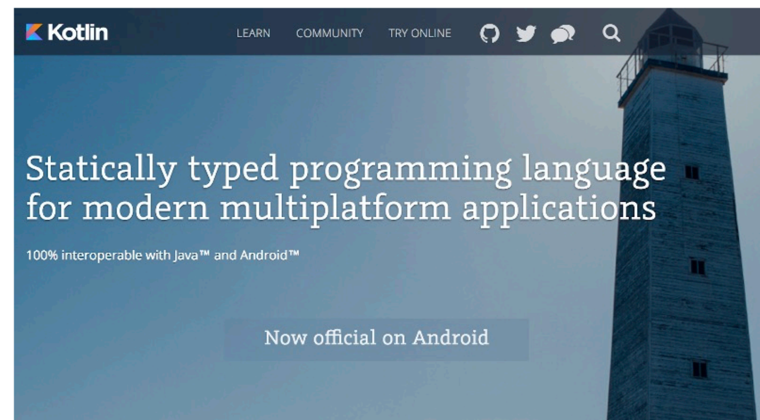

Figure 1: Kotlin is named after an island in the Gulf of Finland.

Features

It's a modern, general-purpose language.

- Compares favourably to Swift, Dart, Scala i.e. other high-level languages.
- Imperative, object-oriented, functional styles (hybrid language).
- Statically-typed; Strong-types; type inference; NULL safety.
- Automatic memory management & garbage-collection.
- Strong standard library, extensions for concurrency, serialization.

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 more.

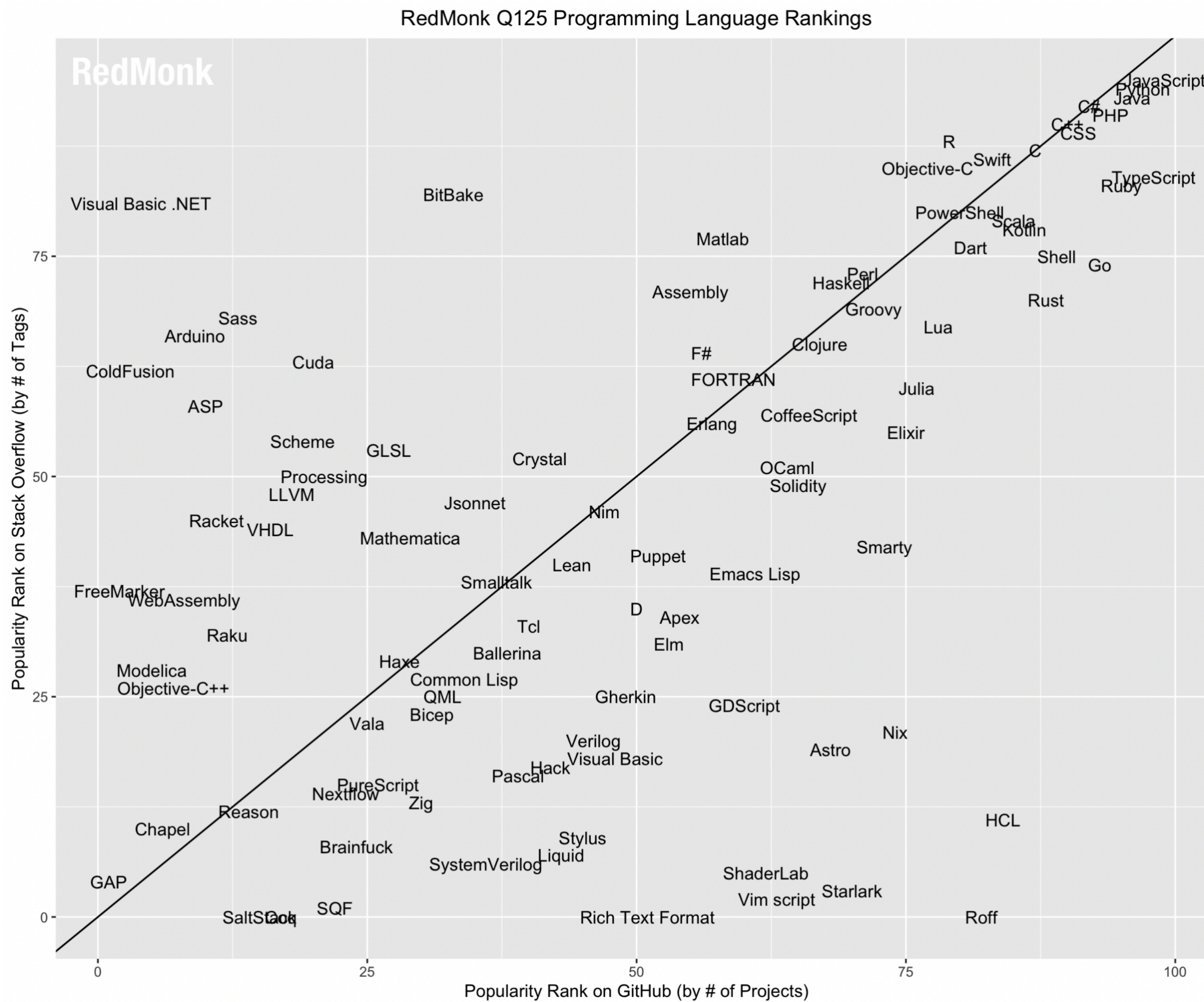

Figure 2: Taken from [RedMonk Q1 2025 data](). Kotlin is popular, and has been adopted for major projects by Google, Netflix, Meta and others.

Getting Started

Toolchain

How do you run Kotlin code? You have options:

1. Command-line compiler
   - Install the compiler from https://kotlinlang.org
   - Use an editor of your choice e.g., VS Code with Kotlin extension.

2. Kotlin Play
   - Online REPL provided by JetBrains
   - Run code directly from https://play.kotlinlang.org

3. Integrated Development Environment (IDE)
   - Install the toolchain locally. Recommended!
   - `IntelliJ IDEA` from https://www.jetbrains.com/idea
   - `Android Studio` from https://developer.android.com

Hello, World!

The entry point of a Kotlin application is a top-level `main` function

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

- all functions start with the `fun` keyword.
- `args` is an optional array of command-line parameters.

Where are the semi-colons?!

source

Variables

Keywords

```
// `val` is an immutable variable
val a: Int = 1 // cannot reassign after initialization
val b: String = "Jeff"

// `var` is a mutable variable
var c: String = "Julie"
c = "Alice"    // can be reassigned
```

Type can be inferred most of the time

```
var d = 2     // 'Int' type is inferred from initial value
```

Initialization can be deferred (aka “late initialization”)

```
val e: Int    // Type required when no initializer is provided
e = 3         // Deferred assignment, can be late initialized
e = 4         // Error: Val cannot be reassigned
```

Strings

Strings are a class in Kotlin, with their own properties and methods. We can use string templates to expand variables or expressions within a string.

```
val i = 10
val s = "kotlin"

println("i = $i") // i = 10
println("${s.capitalize()} is ${s.length} chars") // Kotlin is 6 chars
```

Strings are immutable once created. You *can* append to a string, but you are actually creating a new instance in the background.

Use StringBuilder for better performance.

```
val sb = StringBuilder()
sb.append("Hello")
sb.append(", world!")
println(sb.toString())
```

source

Collections

Kotlin has support for collection types. The main collections are:

- List: an ordered collection of elements that can be accessed by index. This is similar to an Array, but it's dynamic and can grow as items are added to it.
- Set: a collection of unique elements; order isn't guaranteed to stay constant as elements are added or removed.
- HashMap: a collection of key-value pairs. Keys must be unique, but values can be repeated across elements.

All collections support two interfaces [2]:

- A `mutable` interface that extends the corresponding read-only interface with write operations: adding, removing, and updating its elements.
- A `read-only` interface: the collection is read-only and cannot be modified once created.

Collections

List

A List can be created using helper functions: `mutableListOf` and `listOf`.

```
// a mutable list
val numbers1 = mutableListOf("one", "two", "three", "four") // ok
numbers1.add("five") // ok

// note that mutable here refers to the list, not the reference to it
numbers1 = mutableListOf("six", "seven") // error

// immutable list
val numbers2 = listOf("one", "two", "three", "four")  // ok
numbers2.add("five")   // error
```

source

Collections

HashMap

A HashMap can be created using a helper functions. Note that all HashMaps are mutable i.e. they can have elements added or removed.

Note that order is not guaranteed to be preserved, so you should not rely on index when iterating through a HashMap.

Elements consists of a key, mapped to a value.

```
val userMap1 = hashMapOf("id" to 1, "name" to "Alice", "auth" to true)
println(userMap1["name"]) // Alice
println(userMap1["auth"]) // true
```

> We’ll revisit collections when we discuss looping and iterations.

Null Safety

> `Null` represents the absence of a value. We can use NULL to represent an invalid memory address (null pointer), or a string terminator (null character), or an uninitialied value.

What makes `null` values challenging to work with?

- The only valid operation aganist a `null` value is checking if it's `null`. Attempting anything else will throw an exception e.g., NPE.
- This means that you always need explicitly check values to see that they are not `null` before proceeding with any other operation.
- When a `null` is detected, you often need to have a separate execution path (exception) to handle that condition.

```
// status COULD be null, so we need to check before using the value
val status = loadData()
if (status is null) { /* error */ } else { /* do_something */ }
```

Null Safety

Nullable Types

Kotlin introduces the idea of `nullable types`.

- Regular types are not nullable, meaning they can never be assigned a value of null. This is checked by the compiler e.g., `Int`, `String`.
- A type with a `?` suffix is a nullable type, which can include any of the values of the regular type OR null. The compiler can also type-check assignments at compile time e.g., `Int?`, `String`?

```
// Int is not nullable so no need to check the return value
val status: Int = loadData()
println("Status code is ${status}")

// Int? is nullable, meaning that any Int value OR null is valid
// Because it could be null, we have to reintroduce null checks
val libStatus: Int? = library.loadData()
if (libStatus is not null) { println("Status code is ${libStatus}") }
```

Kotlin has features that will make this easier as well.

Null Safety

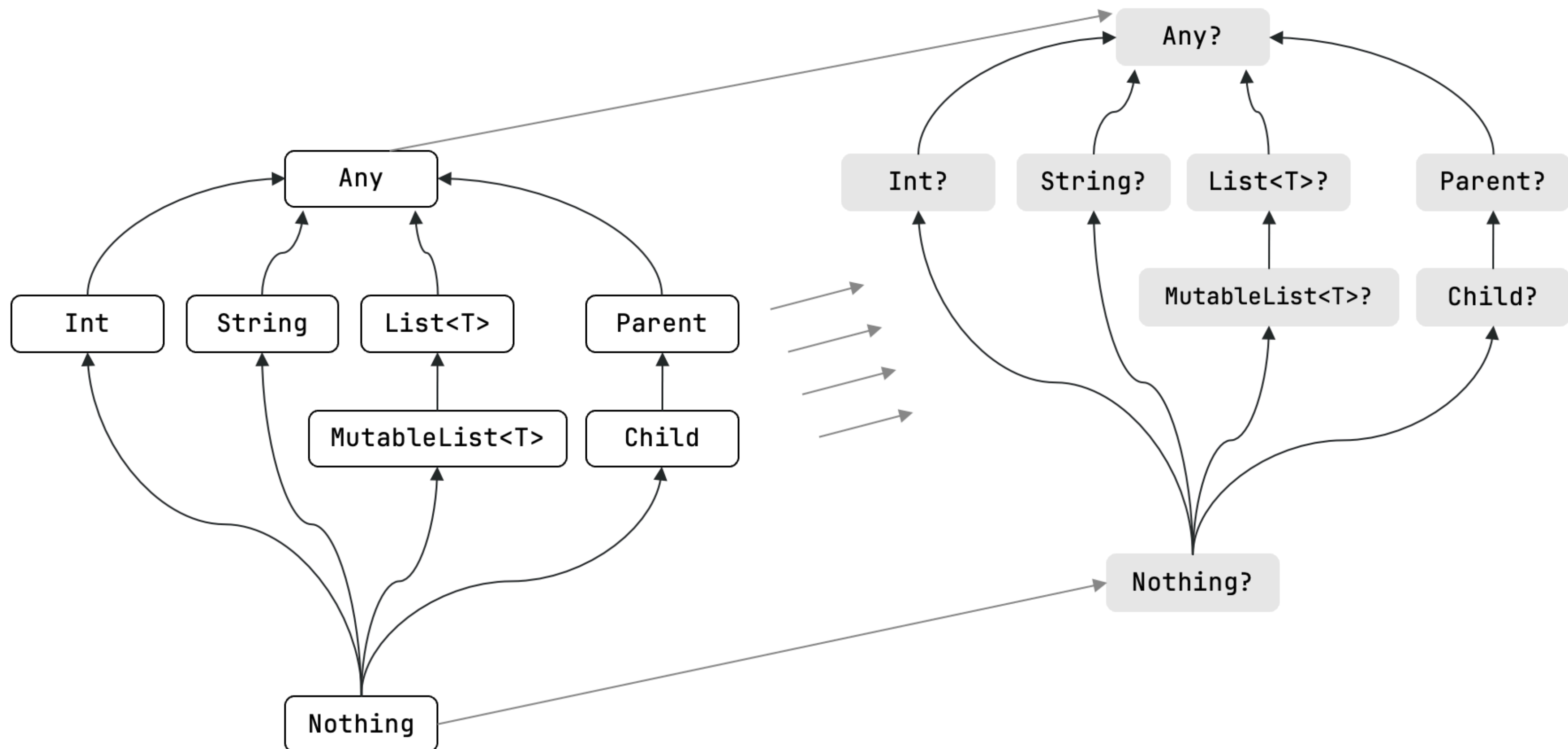

Figure 3: The Kotlin Type Hierarchy showing nullable and non-nullable types.

Null Safety

Safe Calls

Imagine we have two classes with nullable fields. In this example, the employee could have a null department, or a department could have a null name.

```
class Department(var id: Int, var name: String?)
class Employee(var name: String, var department: Department?)

fun main() {
  var employee = Employee("Jeff", Department(1001, "Computer Science"))
    println("${employee.name} works in ${employee.department.name}")
}
```

This doesn't compile because we haven't checked managed the case where department or department.name are null.

`Safe call` syntax (`?.`) accesses a variable ONLY when it's not null. This works:

```
    println("${employee.name} works in ${employee?.department?.name}")
```

Null Safety

Unsafe Calls

The `not-null assertion operator` (!!) force-converts any value to a non-null type and throws an NPE exception if the value is null. This actually removes the error from our previous code.

```
class Department(var id: Int, var name: String?)
class Employee(var name: String, var department: Department?)

fun main() {
  var employee = Employee("Jeff", Department(1001, "Computer Science"))
    println("${employee.name} works in ${employee!!.department!!.name}")
}
```

However, this ONLY works because we *happen* to not be assigning null values anywhere in this code. This is extremely unsafe, and the compiler cannot guarantee that a runtime exception will not be thrown.

Null Safety

Elvis Operator (?:)

A common pattern is to branch based on whether a value is null.

```
val count: Int?
count = processRecords(); // could return null

if (count != null) {
  println("${count} records processed")
} else {
  println("0 records processed")
}
```

We can use the Elvis operator to express this more succinctly.

```
println("${count ?: 0} records processed")
```

If the expression to the LHS of `?:` is not null, then it is returned; if it's null then the RHS is returned.

source

Functions

Kotlin supports top-level functions. You still need a top-level `main` function.

```
fun main() {
  printSum(5, 10) // 15
}

fun printSum(a: Int = 1, b: Int) {
  println(sum(a, b))
}

fun sum(a: Int, b: Int): Int {
  return a + b
}

// could also be written this way
// fun sum(a: Int, b: Int) = a + b
```

source

Functions

Named arguments

- You can specify arguments by name as well as positionally.
- A parameter can be also be given a default value in the function signature

```
fun greeting(userId: Int = 0, message: String) {
  println("${message} from ${userId}")
}

fun main() {
    // positional works, as expected
    greeting(1, "First!")

    // we can reorder parameters if we identify them
    greeting(message = "Second!", userId = 2);

    // uses the default value for 'userId'
    greeting(message = "Third!")
}
```

source

Expressions

If Expression

`if` can be an expression i.e., it can return a value directly.

```
fun maxOf(a: Int, b: Int) = if (a > b) a else b

fun maxOf(a: Int, b: Int): Int {
  if (a > b) {
    return a
  } else {
    return b
  }
}

fun maxOf(a: Int, b: Int) =
if (a > b) {
  a
} else {
  b
}
```

source

Expressions

When Expression

`when` returns, the same way that `if` does. You can evaluate a condition both outside and inside of the branches.

```
when (x) {
  1 -> print("x == 1")
  2 -> print("x == 2")
  else -> {
    print("x is neither 1 nor 2")
  }
}

when {
  x < 0 -> print("x < 0")
  x > 0 -> print("x > 0")
  else -> {
    print("x == 0")
  }
}
```

source

When Statement

`when` can accept several options in one branch.

`else` branch can be omitted if `when` block is used as a statement.

```
fun serveTeaTo(customer: Customer) {
  val teaSack = takeRandomTeaSack()

  when (teaSack) {
    is OolongSack -> error("We don't serve $teaSack!")
    in trialTeaSacks, staleTea -> error("Uncertified tea!")
  }

  teaPackage.brew().serveTo(customer)
}
```

Loops

In this this example, `items` is an iterable collection.

```
val items = listOf("apple", "banana", "kiwifruit")

for (item in items) {
    println(item)
}
// apple
// banana
// kiwifruit

for (index in items.indices) {
    println("item at $index is ${items[index]}")
}

for ((index, item) in items.withIndex()) {
    println("item at $index is $item")
}
```

source

Loops

More traditional looping works as well.

```
val items = listOf("apple", "banana", "kiwifruit")

var index = 0
while (index < items.size) {
    println("item at $index is ${items[index]}")
    index++
}

var toComplete: Boolean
do {
    ...
    toComplete = ...
} while(toComplete)

// The variable can be initialized inside to the do…while loop.
```

Ranges

```
val x = 10
if (x in 1..10) {
    println("fits in range")
}
// fits in range

for (x in 1..5) {
    println(x)
}
// 12345

for (x in 9 downTo 0 step 3) {
    println(x)
}
// 9630
```

source

Bibliography

[1] S. Aigner, R. Elizarov, S. Isakova, and D. Jemerov, *Kotlin in Action*, Second. New York, USA: Manning Publications, 2024. [Online]. Available: https://www.manning.com/books/kotlin-in-action-second-edition

[2] JetBrains Inc., “Kotlin Documentation.” [Online]. Available: https://kotlinlang.org/