

CS 398: Application Development

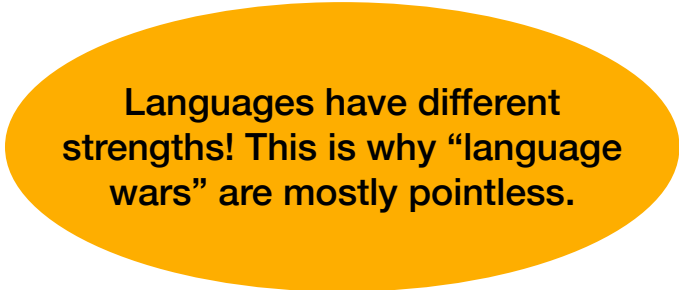
Week 02 Video: Kotlin 1

Why Kotlin?

There are literally [hundreds of programming languages](#) to choose from. How do you pick a language?

There are some non-trivial considerations when picking a language:

- Does it offer the features and capabilities that you require?
- It is easy to work with? How productive can you be with it?
- How mature is the ecosystem around it? Does it have deep libraries, tooling?
- **Is it designed to solve your type of problem?**
- **Does it make appropriate tradeoffs for the type of software that you're building?**



Languages have different strengths! This is why “language wars” are mostly pointless.

Kotlin Features & Strengths

Kotlin is designed for building applications.

- Class-based, object-oriented, general-purpose language.
- Imperative, object-oriented, functional programming styles.
- Automatic memory management and GC; Iterable collections; Generics; Broad framework support (graphics, UI).
- Modern features: named arguments, default arguments; NULL handling.
- 100% interoperable with Java source and libraries.
- Multi-platform: Windows, Linux, Mac (JVM or native); Mobile: Android and iOS.

Installation

You need the Kotlin compiler and runtime. We'll run on the Java JVM.

1. Install Java 16 from adoptopenjdk.net (or another site of your choice).

2. Install Kotlin from kotlinlang.org

3. Check installation from shell:

```
jaffe@Bishop » java --version
```

```
openjdk 16.0.1 2021-04-20
```

```
OpenJDK Runtime Environment (build 16.0.1+9-24)
```

```
OpenJDK 64-Bit Server VM (build 16.0.1+9-24, mixed mode, sharing)
```

```
jaffe@Bishop » kotlinc -version
```

```
info: kotlinc-jvm 1.6.10 (JRE 16.0.1+9-24)
```

IDE Installation



We highly recommend installing and using IntelliJ IDEA in this course.

IDEs offer advanced features: debugging, profiling, code-completion, refactoring.

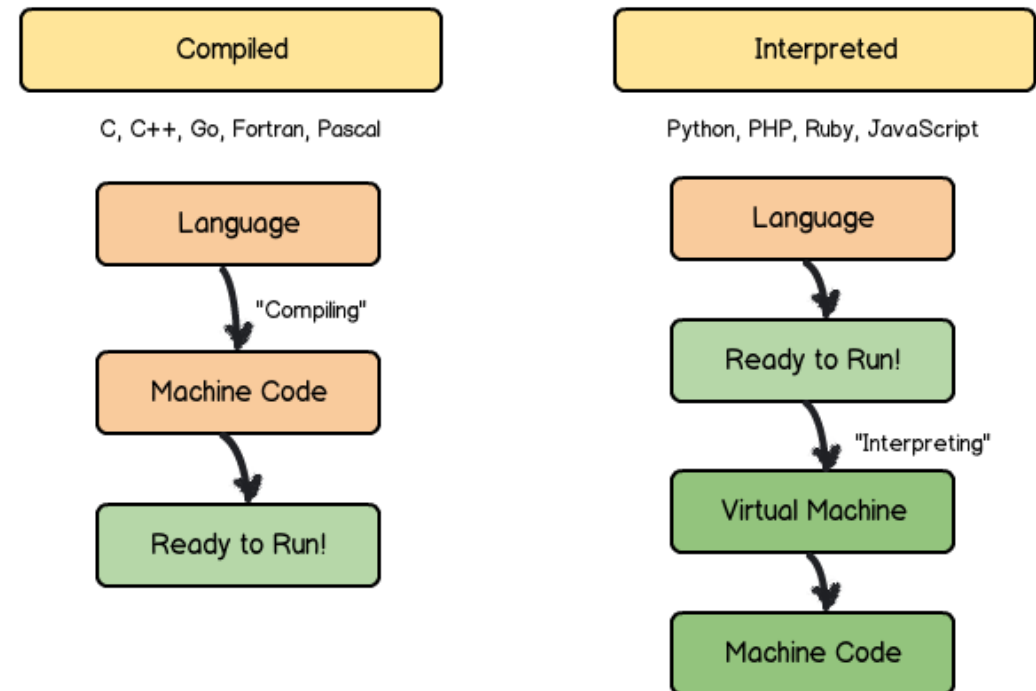
It can be installed from <https://www.jetbrains.com/idea/>

- Community Edition is fine (you can get a free Student license for Ultimate Edition as well as their other products)
- Runs on macOS (Intel or Apple), Windows, Linux.
- Includes Kotlin and Java plugins.
- Requires you to setup projects (which we will demonstrate/discuss soon).

Compilation

Compiled languages require an explicit step to compile code and generate native executables to a specific architecture. e.g. C++.

Interpreted languages interpret the source code (or some intermediate code) at runtime, for the target platform. e.g. Python.



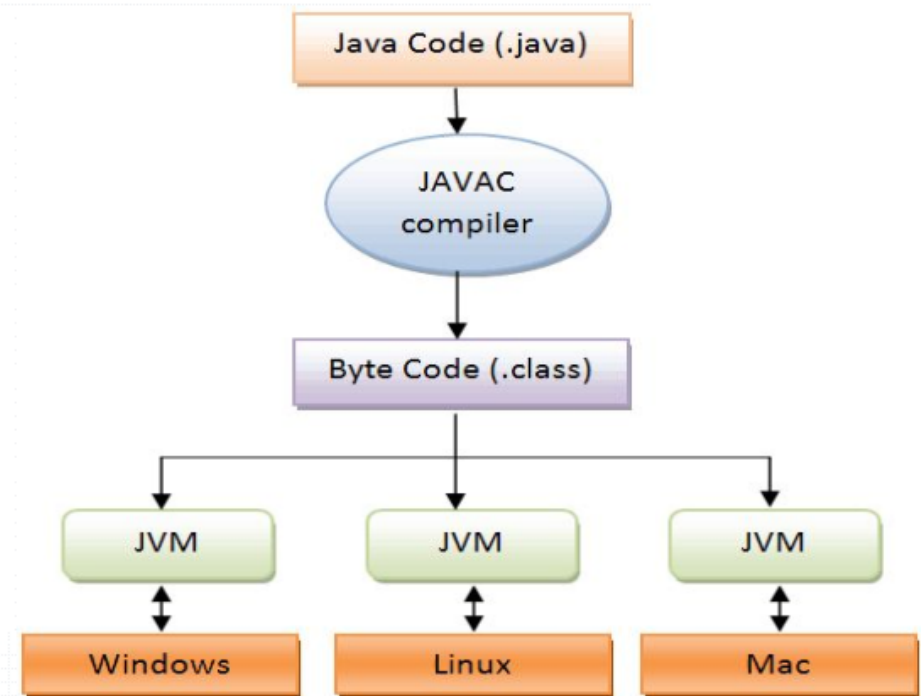
Compiling optimizes startup and execution time at the cost of compilation time.

Interpreting gives us the ability to optimize code at runtime, and provides platform independence.

Java compiles to an intermediate format (bytecode, or .class files), which can be interpreted by a Java Virtual Machine (JVM).

The JVM can also include a Just-in-Time (JIT) compiler which compiles and caches the results at runtime, to optimize for a specific target platform.

We have JVMs for every conceivable platform (incl. Raspberry Pi, refrigerators)



<https://www.fatalerrors.org/a/tomcat-deployment-of-java-application->

How is this relevant to Kotlin?

Other languages can compile to the JVM as well e.g. Scala, Clojure, Kotlin.

See https://en.wikipedia.org/wiki/List_of_JVM_languages

Kotlin can be compiled to native code, or to bytecode (intermediate representation) which is interpreted at runtime.

- **Kotlin/JVM** compiles Kotlin code to JVM bytecode, which can run on any Java virtual machine.
- **Kotlin/Android** compiles Kotlin code to native Android binaries, which leverage native versions of the Java Library and Kotlin standard libraries.
- **Kotlin/Native** compiles Kotlin code to native binaries, which can run without a virtual machine. It is an LLVM based backend for the Kotlin compiler and native implementation of the Kotlin standard library.
- **Kotlin/JS** transpiles (converts) Kotlin to JavaScript. The current implementation targets ECMAScript 5.1 (with plans to eventually target ECMAScript 2015).

We'll be using Kotlin/JVM, and later Kotlin/Android for this course.

Kotlin Ecosystem

Backend

Android

iOS

Web

Desktop

Data Science

Libraries
Frameworks

Tools

Compiler Frontend

Kotlin/JVM

Kotlin/JS

Kotlin/Wasm

Kotlin/Native

Compiling & Executing Code

Hello World!

It's tradition to write "Hello World" when learning a new programming language.

Here's the Kotlin version!

```
fun main() {  
    println("Hello World")  
}
```

Yes, that's all of it.

So how do we run it?

Running Kotlin Code

There are three primary ways of executing Kotlin code:

1. **Read-Evaluate-Print-Loop (REPL):** We can interact directly with the Kotlin runtime, one line at-a-time.
2. **KotlinScript:** We can use Kotlin as a scripting language, by placing our code in a script and executing directly from our shell.
3. **Application:** We can write standalone applications. This is what you'll do most of the time.

1. Read-Evaluate-Print-Loop (REPL)

REPL is a paradigm where you type and submit expressions to the compiler one line-at-a-time. It's commonly used with dynamic languages for debugging, or checking short expressions.

It's not intended as a means of writing full applications!

```
$ kotlinc
```

```
Welcome to Kotlin version 1.6.0 (JRE 16.0.1+9-24)
```

```
Type :help for help, :quit for quit
```

```
>>> val message = "Hello Kotlin"
```

```
>>> println(message)
```

```
Hello Kotlin!
```

2. KotlinScript

KotlinScript is Kotlin code in a script file that we can execute from our shell. Kotlin compiles in the background before executing it.

```
kotlin.kts
```

```
#!/usr/bin/env kotlinc -script  
val message = "Hello Kotlin"  
println(message)
```

```
$ chmod +x kotlin.kts  
$ ./kotlin.kts  
Hello Kotlin
```

This is useful, but *not* a straight-up replacement for shell scripts. Why?

3. Applications

Kotlin applications are fully-functional, and can be compiled to native code, or to the JVM. Kotlin application code looks a little like C, or Java. Here's the world's simplest Kotlin program, consisting of a single main method.

Hello.kt

```
/*  
 * Comment-blocks supported  
 */  
  
fun main(args: Array<String>) {  
    println("Hello Kotlin") // no semi-colon!  
}
```

The argument to main is optional.

No semi-colons!

No top-level class required. We can have a top-level function.

Compiling and executing this is fairly simple:

```
$ kotlinc Hello.kt
```

```
$ ls Hello*  
Hello.kt HelloKt.class
```

```
$ kotlin HelloKt ?  
Hello Kotlin
```

```
$ javap HelloKt  
Compiled from "Hello.kt"  
public final class HelloKt {  
    public static final void main();  
    public static void main(java.lang.String[]);  
}
```

By default, Kotlin targets the JVM, which expects every file to contain a top-level class. Kotlin creates a “wrapper class” for our code.

The **kotlinc** compiler will compile each source file (.kt) into one or more class files (.class). However, this can get messy if you have a large number of classes.

The best-practice is to put all of your output files into a JAR file (basically a ZIP file with some extra data included). This allows you to distribute your application to users as a single file instead of a series of .class files.

This example compiles “Hello.kt” into Hello.jar. The **-include-runtime** flag will also include the Kotlin runtime classes.

```
$ kotlinc Hello.kt -include-runtime -d Hello.jar
```

We can then run from the JAR file directly.

```
$ java -jar Hello.jar  
Hello Kotlin!
```

The JVM includes all of the Java libraries but not standard Kotlin libraries. **-include-runtime** is needed so that your application has the Kotlin libs too.

JAR structure

JAR files are created to distribute your application. Typical contents:

- HelloKt.class – a class wrapper generated by the compiler
- META-INF/MANIFEST.MF – a file containing metadata.
- kotlin/* – Kotlin runtime classes not included in the JDK.

Hello.jar

```
.
├── HelloKt.class
├── META-INF
│   ├── MANIFEST.MF
│   └── main.kotlin_module
└── kotlin
    ├── ArrayIntrinsicsKt.class
    ├── BuilderInference.class
    ├── Deprecated.class
    ├── DeprecationLevel.class
    ├── DslMarker.class
    ├── ExceptionsKt.class
    ├── ExceptionsKt__ExceptionsKt.class
    ├── Experimental$Level.class
    └── Experimental.class
    ...
```

Manifest File

The MANIFEST.MF file is autogenerated by the compiler, and included in the JAR file. It tells the runtime 'main' method to execute. e.g. 'HelloKt.main()'.

```
$ cat MANIFEST.MF
Manifest-Version: 1.0
Created-By: JetBrains Kotlin
Main-Class: HelloKt
```

```
$ java -jar Hello.jar
Hello Kotlin!
> Kotlin 1.5.20
> Java 15.0.2
```

Running from a JAR file

The downside of this approach, of course, is that we need to specify 'java -jar filename.jar' to run our programs. It would be preferable to have a single executable that we can run.

We can create a script to launch our application, with the same effect:

```
$ cat hello
#!/bin/bash
java -jar hello.jar
```

```
$ chmod +x hello
$ ./hello
Hello Kotlin!
> Kotlin 1.3.72
> Java 11.0.7
```

Types

Type System

Programming languages can take different approaches to handling types:

- **Dynamic typing:** type is inferred at runtime. e.g. Python.
- ★ • **Static typing:** variable types need to be declared before use. e.g. C++, Java, Kotlin. This eliminates runtime type errors.

Type systems are often referred to as strong or weakly typed.

- ★ • **Strong typed:** stricter typing rules at compile-time, and less coercion of types, which leads to errors being caught at compile-time. e.g. Java, C++, Kotlin.
- **Weak typed:** looser typing rules, and may allow automatic coercing of variables to different types. Errors deferred to runtime. e.g. JavaScript.

Standard Types

Category	Type	Range	Conversion (& Example)	
Integer	Short	-32768 to 32767	128.toShort()	128
	Int	-2^{31} to $2^{31}-1$	2.78.toInt()	2
	Long	-2^{63} to $2^{63}-1$	B'.toLong()	66
Floating point	Float	24 bits, 6-7 dec.	5.toFloat()	5.0
	Double	53 bits, 15-16 dec.	1.15.toDouble()	1.15
Other	Byte	-128 to 127	'a'.toByte()	97
	Char	ASCII text	97.toChar()	a
	Boolean	true false	"true".toBoolean()	true

Standard Kotlin Types

<https://kotlinlang.org/docs/reference/basic-types.html>

Kotlin
primitives are heap-
allocated objects.

Operators

<code>+, -, *, /, %</code>	Mathematical operators
<code>=</code>	Assignment operator
<code>&&, , !</code>	Logical operators
<code>==, !=</code>	Structural equality
<code>===, !==</code>	Referential equality
<code>[]</code>	Index operators (call get, set)

Operators generally work as you would expect. Note the structural and referential equality.

Variables

Variable declaration includes keywords to indicate *mutability*.

- **Mutable:** variable can be changed (var)
- **Immutable:** variable cannot be changed after initialization (val)

Kotlin emphasizes the use of immutable variables and data structures. This follows best-practices in other languages (e.g. 'final' in Java, 'const' in C++).

<code>var a = 0</code>	// Type inference e.g. <code>auto a = 0</code>
<code>a = 5</code>	
<code>val b = 1</code>	
<code>b = 2</code>	// ERROR: val cannot be reassigned
<code>var c:Int = 10</code>	
<code>val d:String</code>	// ERROR val needs to be initialized
<code>val e:String = "A"</code>	

Strings

Strings are represented by the **String** type, and are immutable.

<https://devdocs.io/kotlin~1.6/api/latest/jvm/stdlib/kotlin/-string/index>

Strings are iterable, so string can be iterated over with a for-loop:

```
for (c in str) {  
    println(c)  
}
```

Strings have properties and methods. e.g.

`str.length`, `str.capitalize`, `str.drop(1)`, `str.dropLast(5)`.

You can concatenate strings using the `+` operator.

```
val s = "abc" + 1  
println(s + "def")
```

String Templates

Kotlin supports the use of string templates, so we can perform variable substitution directly in strings. It's a minor feature that is very commonly used!

```
val version = "1.6"  
println("Kotlin $version")  
> Kotlin 1.6
```

We can even evaluate expressions as part of a string.

```
val str = "abc"  
println("$str.length is ${str.length}")  
  
var n = 5  
println("${if(n > 0) "+ve" else "-ve"}")
```

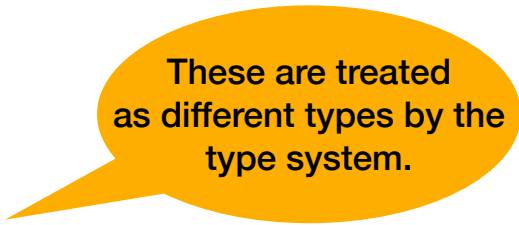
NULL data

What is NULL data? NULL is a special value that indicates that there is *no data*. Kotlin has special semantics for dealing with nulls that avoids the need to explicitly check for them.

By default, a variable cannot be assigned a NULL value.

A ? suffix on the type indicates that NULL-able.

```
val length: Int = null    // ERROR
var name: String? = null  // OK
```



These are treated as different types by the type system.

If you have a nullable variable, then all calls to that variable must handle nulls.

```
if (name != null) println(name)
```

NULL syntax

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

`?.` is the “safe call operator”. A method call will only be invoked if the object is not null.

```
var name: String? = null
val len = name?.length
```

`?:` is a ternary operator for NULL data (also called the Elvis operator)

```
val len = name?.length ?: name?.length : 0
```

if exists return length else return zero

Functions

Functions

Functions are preceded with the 'fun' keyword. Function arguments require types, and are immutable.

```
fun hello() {  
    println("Hello World")  
}
```

// Arguments require type annotations!

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

// Return type can be inferred

```
fun sum(a: Int, b: Int) = a + b
```

Default Arguments

We can supply default values for parameters. A parameter with a default value is optional, since the caller can always assume that the default will be used.

// Second argument has a default value, so it's optional

```
fun mult(a:Int, b:Int = 1): Int {  
    return a * b  
}
```

```
mult(1) // 1
```

```
mult(5,2) // 10
```

```
mult() // error
```

// if is an expression and returns a value

```
fun minOf(a: Int, b: Int) = if (a < b) a else b
```

```
minOf(1,2) // 1
```

```
minOf(5,4) // 4
```


Named Arguments

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

```
#!/usr/bin/env kotlinc-jvm -script
```

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

```
println(repeat()) // *  
println(repeat(n=3)) // ***  
println(repeat(n=5,s="#")) // #####
```

Variable-Length Argument Lists

Finally, we can have a list of undefined length (i.e. evaluated at runtime).

```
// Variable number of arguments can be passed!  
// Arguments in the list need to have the same type
```

```
fun sum(vararg numbers: Int): Int {  
    var sum: Int = 0  
    for(number in numbers) {  
        sum += number  
    }  
    return sum  
}
```

```
sum(1) // 1
```

```
sum(1,2,3) // 6
```

```
sum(1,2,3,4,5,6,7,8,9,10) // 55
```

Examples

Example: mean.kts

Here's a Kotlin snippet to calculate the mean of a series of numbers.

- 'args' is an array of command-line arguments, automatically passed
- loop over 'args' and sum the values to calculate the mean

mean.kts

```
var sum = 0f // Inferred as Float
for (arg in args) {
    sum += arg.toFloat()
}
println(sum/args.size)
```

See the public repo for some samples.
<https://git.uwaterloo.ca/j2avery/cs346-public>

Simple examples: /scripts