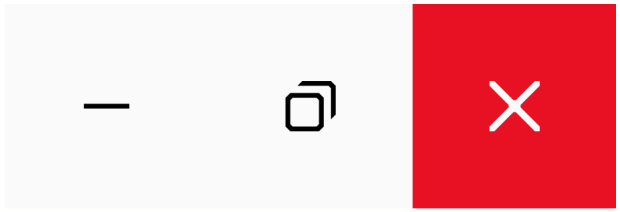


IntelliJ & Kotlin

Software Stack
Kotlin Crash Course

May 10

Software Stack

Software

Git, a distributed version control software.

Kotlin, a cross-platform, statically typed, general-purpose programming language. It's 100% interoperable with Java, and can be used as a replacement for that language. Can be used to build desktop, mobile applications and services.

IntelliJ IDEA, an integrated development environment (IDE) for developing computer software in Java, Kotlin, Groovy, and other languages.

Gradle, a build automation tool, similar to 'make'.

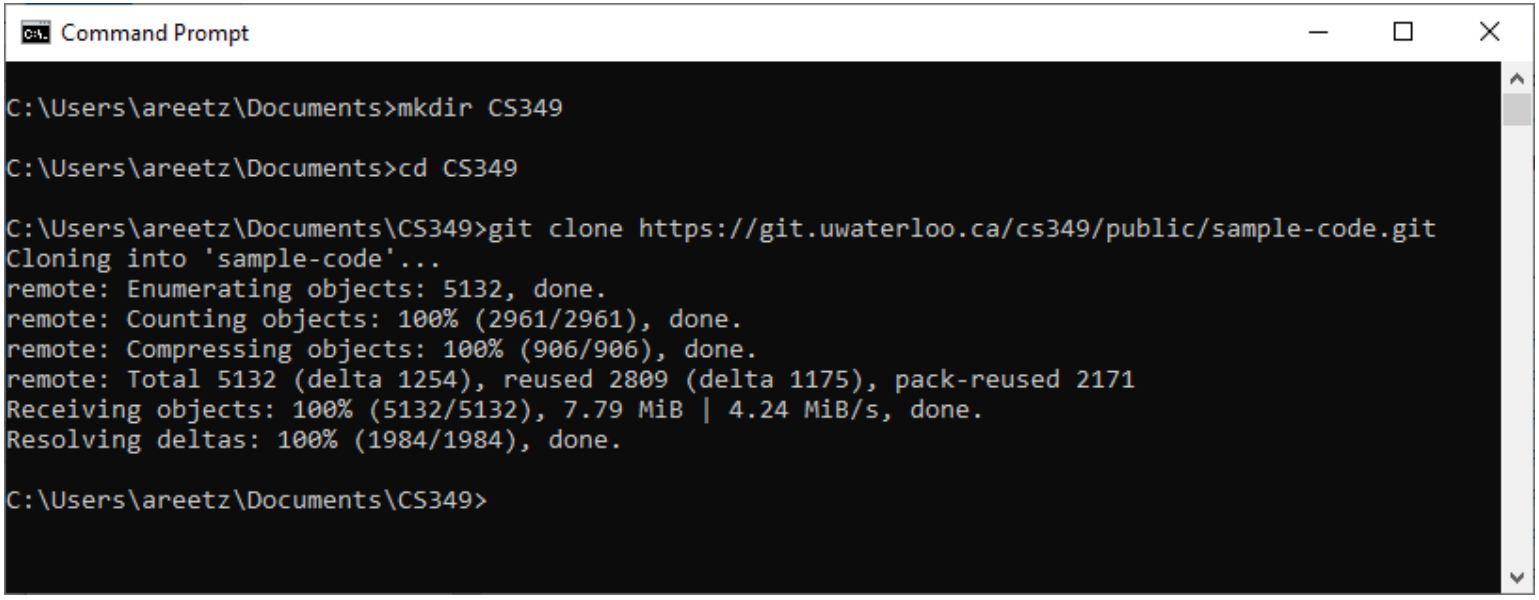
JavaFX, libraries for creating and delivering desktop applications. Not included in Kotlin, so we will discuss this next class!

Git

Install Git from https://git-scm.com/download, latest version is 2.39.0.
Install with default options, except preferred text editor.

Create a local copy of `cs349-public`:

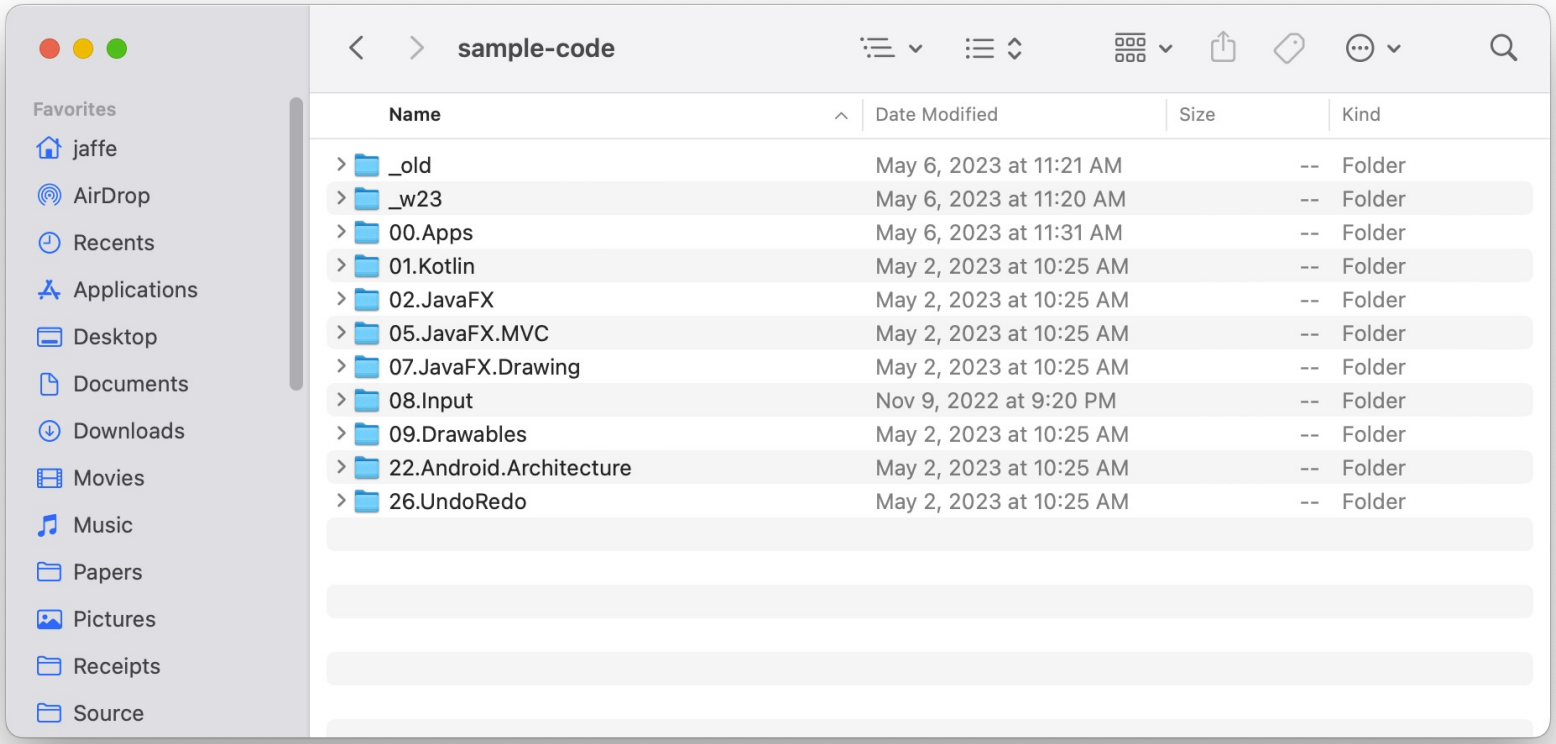
```
git clone https://git.uwaterloo.ca/cs349/public/sample-code.git
```



The public repository contains the code shown in class.

Git

Your local copy directory should look like this:



IntelliJ

Install IntelliJ IDEA Community Edition from https://www.jetbrains.com/idea/download/, latest version is 2022.3.1.

Install with default options.

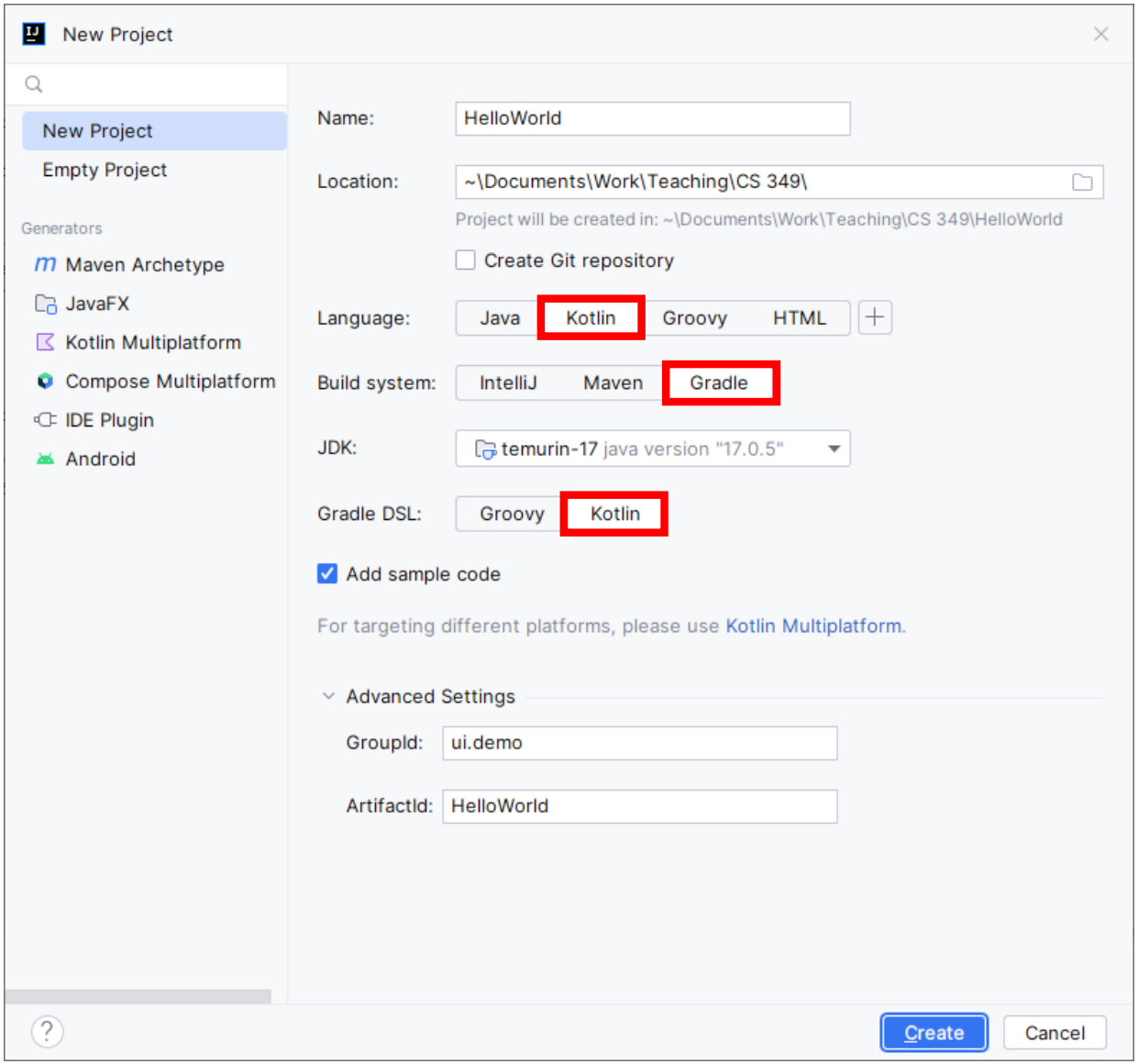
Once installed, IntelliJ will install Gradle, Kotlin, and the Java SDK for you, either when you create a new project (preferred, as you can choose the JDK version) or when you open an existing project (not preferred, installs JDK version of the project).

- **Java SDK**: use version 17.0.5.
  - SDK 17 is marked as LTS, so `cs349-public` is using this version.
- **Gradle**: use version 7.4.2+ (automatically installed by IntelliJ).
- **Kotlin**: use version 1.7.21 (automatically installed by IntelliJ).

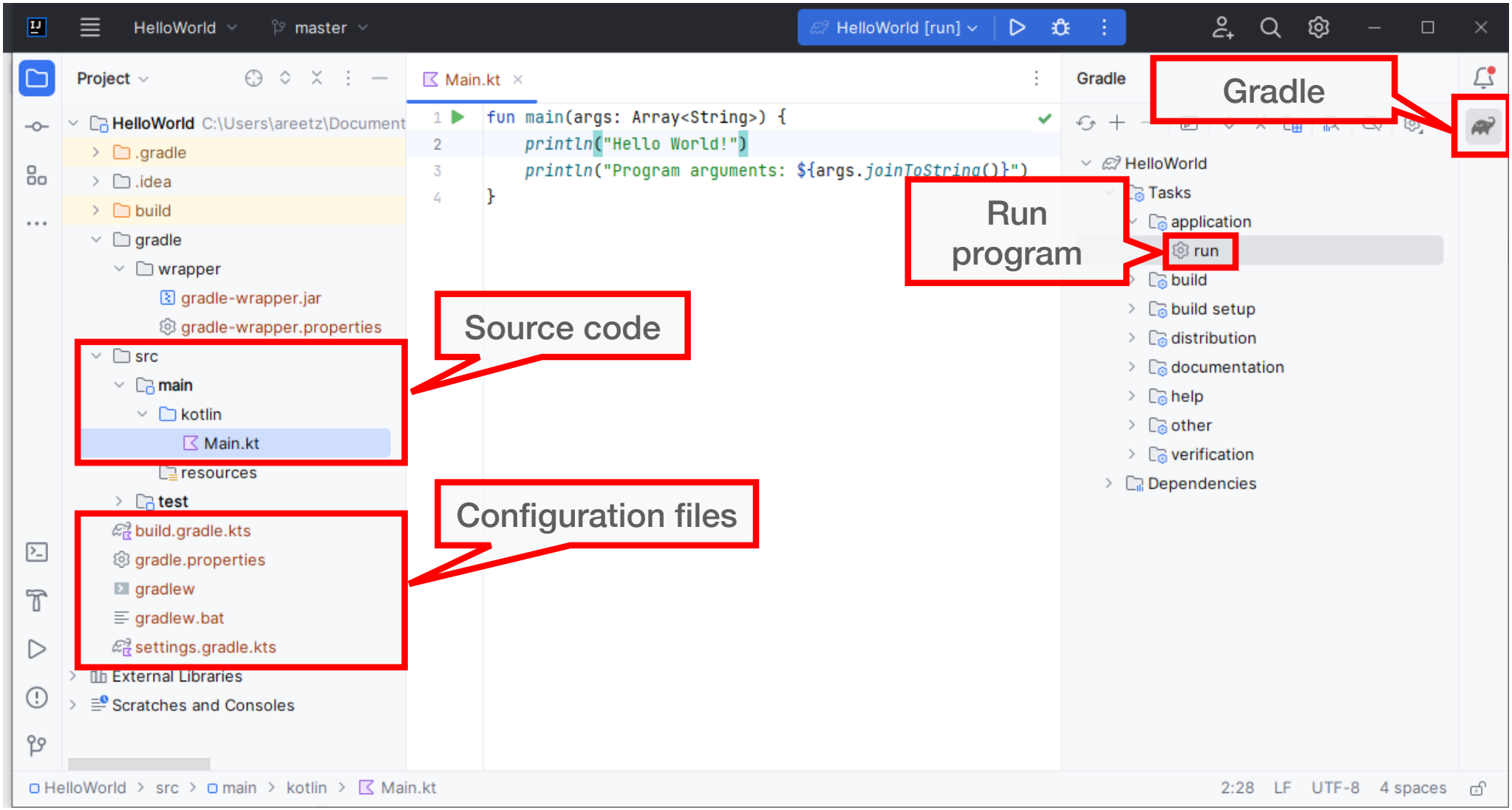
CAUTION: For the Java SDK, make sure you install the correct version for your system architecture. Apple M1/M2 machines need an ARM build!

IntelliJ & Gradle – New Project

Creating a new project:



IntelliJ & Gradle – Project Structure



Command Line & Gradle Run Application

We can run Gradle from the command line to build our projects.

Hint: You might want to test if your assignment project runs from the command line before submitting.

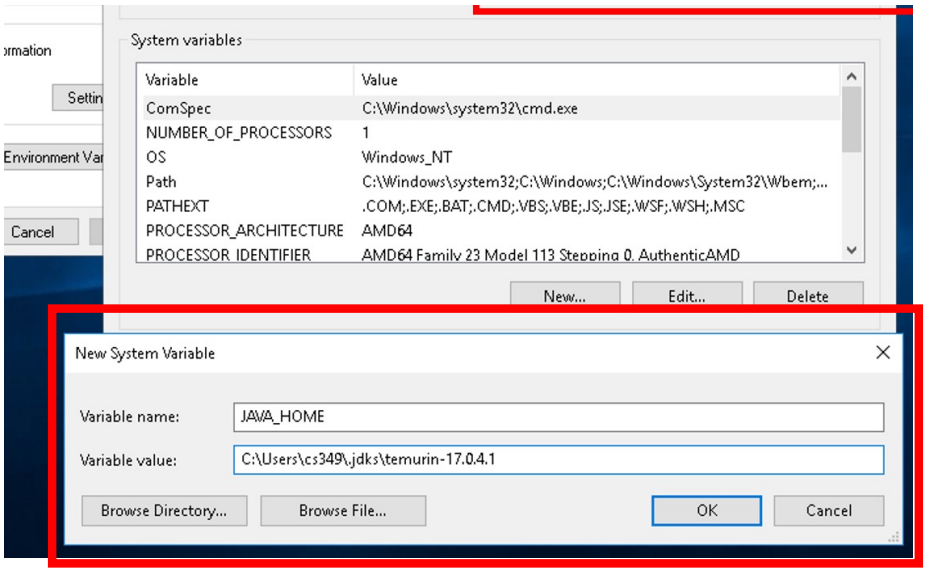
For macOS/Linux: set JAVA_HOME in your init scripts.

```
# JDK
export JAVA_HOME='/Users/jaffe/Library/Java/JavaVirtualMachines/azul-18/Contents/Home'
export PATH=$PATH:$JAVA_HOME/bin
```

For Windows:

set `JAVA_HOME` to

`%USERPROFILE%\.jdks\...`



Gradle Command-Line

The top-level of your IntelliJ project will contain the Gradle configuration files.

It will also contain a script: `./gradlew`

Use this script to run Gradle commands:

```
./gradlew clean          // remove temp files
./gradlew build          // build the project
./gradlew run            // run the main method
```

IMPORTANT:

- Replace `./gradlew` with `gradlew.bat` on Windows.
- You can also run these commands in IntelliJ (Gradle menu).

Demo

Creating a new Kotlin project:

- https://student.cs.uwaterloo.ca/~cs349/1235/getting-started/3-kotlin-project/

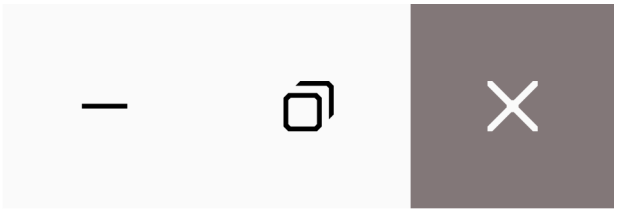
Running sample code:

- https://student.cs.uwaterloo.ca/~cs349/1235/getting-started/5-run-samples/

Finding your repository

- https://git.uwaterloo.ca

Kotlin Crash Course



CS 349

Kotlin – Why

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:

- It is easy to work with? How productive can you be with it?
- How mature is the ecosystem around it?
- Does it have rich libraries? Tool support?

Does it offer the features and capabilities that you require?

Does it let you compile and build to your target?

- JS/TS for web
- Swift/Kotlin for mobile
- (pick one of many) for desktop

Kotlin – Features & Strengths

Kotlin is designed for building applications.

- Class-based, object-oriented, general-purpose language.
- Supports imperative, object-oriented, and functional programming styles.
- Automatic memory management and GC; iterable collections; generics; Broad framework support (graphics, UI).
- Modern features: named arguments, default arguments; NULL handling.

It was designed to be an appealing replacement for Java.

- 100% interoperable with Java source and libraries.
- Multi-platform: Windows, Linux, Mac (JVM or native)
- Mobile: Android and iOS.

Kotlin – Other 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.
- **Kotlin/JS** transpiles (converts) Kotlin to JavaScript.

We will be using Kotlin/JVM and later Kotlin/Android for this course.

Learning Kotlin – Resources

You *might* want to read more than just these slides!

Online (free)

- **Kotlin**. https://kotlinlang.org/
- **Kotlin Playground**. https://play.kotlinlang.org/
- **Dave Leeds on Kotlin**. https://typealias.com/start/
- **Kotlin Basics course**. JetBrains Academy. https://www.jetbrains.com/academy

Books (not free)

- Elizarov, Isakova, Aigner, Jemerov. **Kotlin in Action**. 2nd ed. Pre-publication. https://www.manning.com/books/kotlin-in-action-second-edition
- Bailey, Greenhalgh, Skeen. **Kotlin Programming: The Big Nerd Ranch Guide**. 2021. Pearson. ISBN 978-0136870487.

Program Entry

Program Entry Point

It’s tradition to write ”Hello World” when learning a new programming language.

Here’s the Kotlin version!

```
// one form with no arguments
fun main() {
    print("Hello ")  // print prints parameter as string
    println("World") // println prints parameter as string + newline
}

// another form with arguments
fun main(args: Array<String>) {
    println(args.contentToString())
}
```

Program Entry Point

It’s tradition to write ”Hello World” when learning a new programming language.

Here’s the Kotlin version!

```
fun main() {                            // argument(s):
    print("Hello ")
    println("World")                    // prints: Hello World
}

fun main(args: Array<String>) {         // argument(s): This is CS349
    println(args.contentToString()) // prints: [This, is, CS349]
}
```

Data

Data – Typing

Kotlin is *strongly statically* typed.

- Static: type is verified at compile time (not run time)
- Strong: type is verified from syntax (not derived from underlying data)

```
var i: Int = 349 // declaration & assignment:
var i = 349      // type optional (inferred at compile time)
var i: Int       // declaration only: type is required
```

Types

- Integer: Byte, Short, Int, Long
- Floating Point: Float, Double
- Other: Boolean, Character, String

Data - Mutability

There are two flavours of data:

Mutable data

- *Changeable*
- *Var for variable*

```
var i: Int = 349
i = 350  // ok
```

Immutable data

- *Unchangeable*
- *Val for value*

```
val i: Int = 349
i = 350  // error
```

1. Mutability qualifier†: `var`
2. Identifier: `i`
3. `:` Type‡: `Int`
4. Assignment operator†: `=`
5. Value: `349`

†: see below

‡: optional

Immutable Data – Values

Immutable data (*value*) cannot be reassigned after initialization. We use the `val` keyword to denote immutable data (`val` for value).

```
val i: Int = 349 // declaration & assignment

i = 449          // compile error: Val cannot be reassigned

val j: Int       // declaration only, type is required
j = 349          // deferred assignment

j = 449          // compile error: Val cannot be reassigned
j = "449"        // compile error: Val cannot be reassigned &
                 //                          type mismatch
```

Data – Variables vs Values

Data should be immutable unless they absolutely need to change!

This follows best-practices in other languages, e.g., `final` in Java, `const` in C++.

Nullable Data

Nullable Data

A reference must be explicitly marked as nullable when `null` value is possible. Nullable type names have a `?` at the end:

```
var myString: String = "CS349"
println(myString.length)              // prints: 5
var maybeNullString: String? = null
```

`?.` is the safe call operator. It only continues with de-referencing if the data is non-`null`:

```
var maybeNullString: String? = null
println(maybeNullString?.length) // prints: null

maybeNullString = myString
println(maybeNullString?.length) // prints: 5
```

Nullable Data

`?:` is a ternary operator for `null` (also called the “Elvis operator”). It will return the LHS if non-`null`, otherwise it returns the RHS of the expression.

```
var maybeNullString : String? = null
println(maybeNullString?.length ?: 0)  // prints: 0
```

`!!` converts any nullable value to a non-`null` type and throws an exception if the value is `null`. Only use this operator if you are certain that a value is not `null`!

```
var maybeNullString: String? = "abc"
var nonNullString: String = maybeNullString!! // fail if the
var nonNullStringLength = maybeNullString!!.length // str null
```

Nullable Data

By using the `?:` operator, we can “remove” nullable data; and by using the `is` operator, we can “weaken” the strict typing of Kotlin.

```
var myList = listOf(1, 2, 3, null) // myList is of type List<Int?>
myList.forEach {                   // prints: 1, 2, 3, null
    println(it)                    // it is of type Int?
}

println(myList.fold(0) { acc, elem -> acc + (elem ?: 0) }) // prints: 6
// type of acc is Int, type of elem is Int?, type of (elem ?: 0) is Int


var myList = listOf(1, 2, 3, null, "Four")
println(myList.fold(0) { acc, elem ->                  // prints: null 9
    acc + when (elem) { // is of type Int; when(elem) yields an Int
        is Int -> elem
        is String -> elem.length
        null -> { println("null")
                  -1 }
        else -> 0
    }})
```

Strings

Strings

Strings are represented by the String class and are immutable. A string value is a sequence of characters in double quotes (""):

```
var str = "CS349"
println(str.lowercase()) // prints: cs349
println(str)             // prints: CS349
```

Strings can be accessed via the indexing operator and iterated over:

```
println(str[0])          // prints: C

for (c: Char in str)     // prints: C_S_3_4_9_
    print("$c_")

str.forEach { print(it.lowercaseChar()) } // prints: cs349
```

Strings

Strings are represented by the String class and are immutable. A string value is a sequence of characters in double quotes (""):

```
val myCourseCode = "CS"
val myCourseNumber = "349"
```

Strings can be concatenated using the +-operator:

```
println(myCourseCode + myCourseNumber)   // prints: CS349
println("$myCourseCode $myCourseNumber") // prints: CS 349
```

String Templates

String literals may contain template expressions – pieces of code that are evaluated and whose results are concatenated into the string.

A template expression starts with a dollar sign ($) and consists of either a name or an expression in curly braces ({}).

```
fun sum(a: Int, b: Int): Unit {
    println("$a + $b = ${a + b}")
}
sum(349, 42)                                    // prints: 349 + 42 = 391


val str = "CS349"
println("$str.length is ${str.length}") // prints: CS349.length is 5
```

Collections

Collections

Kotlin distinguished between normal (immutable) and mutable collections:

Immutable:

- List
- Set
- Map

Mutable:

- MutableList
- MutableSet
- MutableMap

Collections – Lists

`List<T>` stores elements in a specified order and provides indexed access to them:

```
var fruits = listOf("apple", "banana", "cherry")
fruits.forEach { println(it) }              // prints: apple, banana, cherry

fruits = listOf("apricot", "blueberry", "coconut")
println(fruits[2])                          // prints: coconut
println(fruits.get(0))                      // prints: apricot
println(fruits.indexOf("blueberry"))        // prints: 1
println(fruits.size)                        // prints: 3

val animals = mutableListOf("ape", "beaver", "camel")
animals.forEach { println(it) }     // prints: ape, beaver, camel
animals.removeAt(0)
animals.remove("camel")
println(animals)                    // prints: [beaver]
animals.add("comoran")
animals.add(0, "ant")
println(animals)                    // prints: [ant, beaver, comoran]
```

Collections – Sets

`Set<T>` stores unique elements; their order is undefined. `null` elements are unique as well: a Set can contain only one `null`. Two sets are equal if they have the same size, and for each element of a set there is an equal element in the other set.

```
val set1 = setOf(1, 1, 2, null)
val set2 = setOf(1, 2, null, null, null)
val set3 = mutableSetOf(1, 1, 2, 3)

println(set1 == set2)          // prints: true
println(set1 == set3)          // prints: false
println(set1.union(set3))      // prints: [1, 2, null, 3]
println(set1.intersect(set3)) // prints: [1, 2]
println(set1.minus(set2))      // prints: []
```

Collections – Maps

`Map<K, V>` stores key-value pairs (or entries)

Keys are unique, but different keys can be paired with equal values.

```
val days = mapOf(1 to "Monday", 4 to "Thursday", 6 to "Saturday")
println(days[3])      // prints: null

val langs = mutableMapOf("135" to "Racket",
                         "136" to "C",
                         "349" to "Kotlin")

println(langs["349"]) // prints: Kotlin

langs["246"] = "C++"
langs["135"] = "Python"
println(langs)        // prints: {135=Python, 136=C, 349=Kotlin, 246=C++}
```

Collections – Higher Order Functions

```
var list= listOf(1, 2, 3, 4, 5, 6, 7, 8)   // could also express as (1..8)
list.filter { value -> value % 2 == 0 }    // yields: [2, 4, 6, 8]

list.filter { it % 2 == 0 }                // same idea, concise syntax
list.map { it * 2 }                        // yields: [2, 4, 6, .. 16]
list.fold(0) { acc, elem -> acc + elem }   // yields: 36
list.reduce { acc, elem -> acc + elem }    // yields: 36
list.forEach { print(it) }                 // prints: 12345678
list.onEach { print(it) }                  // prints: 12345678

val (even, odd) = list.partition { it % 2 == 0 }
// yields: [2, 4, 6, 8], [1, 3, 5, 7]

var factors = list.groupBy { it % 3 }
// contains: {0=[3, 6, 9], 1=[1, 4, 7], 2=[2, 5, 8]}
```

These work on all collections!
Also see *take*, *first*, *last*, *slice*, …

Ranges and Progressions

Ranges and Progressions

A range defines a closed interval in the mathematical sense: it is defined by its two endpoint values which are both included in the range.

```
var range = 5..10           // contains: 5, 6, 7, 8, 9, 10
range = 5 until 10          // contains: 5, 6, 7, 8, 9

for (value in range)        // prints: 5, 6, 7, 8, 9
    println(value)

val existing = 8 in range  // true
val missing = 42 !in range // true
```

Ranges can be defined for any comparable data type.

Ranges and Progressions

A progression is defined by its start and end points, as well as a non-zero step.

```
var prog = 10 downTo 5     // contains: 10, 9, 8, 7, 6, 5
prog = 10 downTo 5 step 2 // contains: 10, 8, 6

for (value in range)       // prints: 5, 6, 7, 8, 9
    println(value)

val i = 7
println(i in prog)
// false


(5..10).forEach { println(it) }
```

Loops

Loops – `for` and *`forEach`*

`for` loops through any data that provides an iterator.

```
for (c in "CS349") { // prints: CS349
    print(c)
}

for (i in 0..9) {    // prints: 0123456789
    print(i)
}
```

`forEach` iterates through any data that provides an iterator.

```
val printInt: (Int) -> Unit = { i: Int -> print(i) }
(0..9).forEach(printInt)     // prints: 0123456789
```

A more concise (i.e., common, Kotlin-like, “sophisticate”) way to express the above is using an anonymous function:

```
(0..9).forEach { print(it) } // prints: 0123456789
```

Loops – `for` and *`forEach`*

`for` loops through any data that provides an iterator.

```
val fruits = listOf("Apple", "Banana", "Cherry")

for (i in 0 until fruits.size) { // i runs from 0 to fruits.size - 1
    println("Element at $i is \"${fruits[i]}\"")
}
// prints: Element at 0 is "Apple"
//         Element at 1 is "Banana"
//         Element at 2 is "Cherry"
```

While the code above works, there are more elegant way to traverse a list in Kotlin:

```
for ((index, value) in fruits.withIndex()) {
    println("Element at $index is \"$value\"")
}
```

Generally, avoid indices if not strictly necessary.

Loops – `for` and *`forEach`*

`for` loops through any data that provides an iterator.

```
val fruits = mapOf("Apple" to 2.99, "Banana" to 0.69, "Cherry" to 4.99)

for ((key, value) in fruits) {
    println("Price for $key is \"$value\"")
}
```

`forEach` iterates through any data that provides an iterator.

```
fruits.forEach {
    (key, value) -> println("Price for $key is \"$value\"")
}
```

Loops – `while` and `do…while`

while checks the condition and, if it's satisfied, executes the body and then returns to the condition check.

```
var x=10
while (x > 0) {
      x--
}
```

do-while executes the body and then checks the condition. If it's satisfied, the loop repeats. So, the body of do-while executes at least once regardless of the condition.

```
do {
      val y = retrieveData()
}
while (y != null)
```

Conditionals

Conditionals – `if`

`if`-conditionals yield a value:

```
fun parity(i: Int) {
    if (i % 2 == 0) {
        println("$i is even")
    } else {
        println("$i is odd")
    }
}
```

```
fun sign(i: Int): Int {
    return if (i > 0)
               1
           else if (i < 0)
               -1
           else
               0
}
```

Conditionals – when

when defines a conditional expression with multiple branches, it may yield a value:

```
fun selector(course: String): Unit {
    when (course.take(2)) {
        "CS" -> println("Computer Science course")
        "BU" -> println("Business course")
        else -> println("unknown course")
    }
}

fun selector(course: String): String {
    return when (course.take(2)) {
        "CS" -> "Computer Science course"
        "BU" -> "Business course"
        else -> "unknown course"
    }
}
```

Functions

Functions

```
fun meaning(): Int {
    return 42
}

fun sum(a: Int, b: Int) {
    println("$a + $b = ${a + b}")
}
```

1. Function keyword: `fun`
2. Identifier: `meaning`
3. (Parameter list):
4. : Return type‡: `Int`
5. {Function body}

‡: optional

Functions – Return Type

The `Unit` object is a type with only one value. This type corresponds to `void` in many other programing languages.

```
fun sum(a: Int, b: Int): Unit { ... }

fun sum(a: Int, b: Int) { ... } // No return type defaults to Unit
```

Kotlin does not (usually) infer return types:

```
fun add1(i: Int) {
    return i + 1   // Type mismatch. Required: Unit Found: Int
}
```

Functions – Calling, Default Arguments

```
fun addn(i: Int, n: Int = 1): Int { // n has default value 1
    print("add$n($i) ")             // string template
    return i + n
}

println(addn(5, 3))          // prints: add3(5) 8

println(addn(5))             // prints: add1(5) 6: uses default value for n

println(addn(n = 3, i = 5)) // prints: add3(5) 8: naming parameters
```

Functions – Variable-length Argument Lists

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

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

println(sum(1))     // prints: 1
println(sum(1,2,3)) // prints: 6
```

Functions – Lambda

Let's start with a typical function:

```
fun addn(i:Int, n:Int): Int {
    print("add$n($i) ")
    return i + n
}
```

In Kotlin, functions are *first-class language elements*. This means that we can treat functions as *data*. Let's re-express this function:

```
var addn: (Int, Int) -> Int = {
    i: Int, n: Int ->
    print("add$n($i) ")
    i + n
}
```

Why is this different? The variable is just a reference to a free-floating function, which is defined within the braces. We can, for example, pass this variable into other functions (i.e., it's just a variable!)

Functions – Lambda

Since we can express a function as data, we can do anything we would do with data – like *passing functions as parameters into other functions*.

```
var addn = {                // type (function signature) inferred from
    i: Int, n: Int ->       //   parameter list ...
    print("add$n($i) ")
    i + n                   //  ... and last statement in body (return type)
}

fun apply3(i: Int, func: (Int, Int) -> Int) : Int {
    return func(i, 3)
}

println(apply3(5, addn)) // prints: add3(5) 8
```

Functions – Lambda

Functions can be passed as parameter from data:

```
var addn = {                // type (function signature) inferred from
    i: Int, n: Int ->       //   parameter list ...
    print("add$n($i) ")
    i + n                   //  ... and last statement in body (return type)
}

fun apply3(i: Int, func: (Int, Int) -> Int) : Int {
    return func(i, 3)
}

println(apply3(5, addn)) // prints: add3(5) 8
```

Functions – Lambda

Functions can be passed as parameter from an inline definition as long as the function is the last parameter in the parameter list of the higher-order function:

```
fun apply3(i: Int, func: (Int, Int) -> Int) : Int {
    return func(i, 3)
}

println(apply3(5) {                 // prints: 5 mod 3 is 2
    i: Int, m: Int ->
    print("$i mod $m is ")
    i % m
})
```

Functions – Lambda

If a function only has one parameter, it does not need to be declared, and `->` can be omitted. The parameter will be implicitly declared under the name `it`:

```
fun printif(n: Int, pred: (Int) -> Boolean) {
    if (pred(n)) {
        println(n)
    }
}

printif(349) { n: Int -> n % 2 == 1 } // prints: 349

printif(349) { it % 2 == 1 }          // prints: 349
```

Functions – Lambda

Functions can be passed as parameter from inline definition:

```
fun printif(n: Int, pred: (Int) -> Boolean) {
    if (pred(n)) {
        println(n)
    }
}

(0..6).forEach { printif(it) { it % 2 == 1 } } // prints: 1, 3, 5
```

Operators

Precedence	Title	Symbols	Example
Highest	Postfix	++, --, ., ?., ?, !!	i++ obj.func() obj?.field obj!!
	Prefix	-, +, ++, --, !, label	-i --i !b
	Type RHS	:, as, as?	var i: Int objA as TypeB
	Multiplicative	*, /, %	i * j
	Additive	+, -	i + j
	Range	..	for (i in 1 .. 10)
	Infix function	simpleIdentifier	val b1 = b2 or b3 var lst = 1 until 10
	Elvis	?:	val i = obj?.field ?: -1
	Named checks	in, !in, is, !is	5 in lst objA is TypeA
	Comparison	<, >, <=, >=	a < b a >= b
	Equality	==, !=, ===, !==	a != b objA === objB
	Conjunction	&&	b1 && b2
	Disjunction	||	b1 || b2
	Spread operator	*	
Lowest	Assignment	=, +=, -=, *=, /=, %=	var i = 349 i += j

Standard Types

Category	Type	Range	Examples & Conversions
Numbers	`Byte`	-128 to 127	`127` `128.toByte() // yields: -128`
	`UByte`	0 to 255	
	`Short`	-32768 to 32767	`349.99.toInt().toShort() // yields: 349`
	`UShort`	0 to 65535	
Floating point	`Int`	-2^{31} to $2^{31}-1$	`1000003` `0x349A // yields 13466`
	`UInt`	0 to $2^{32}-1$	
Text	`Long`	-2^{63} to $2^{63}-1$	`349L` `349.toLong()`
	`ULong`	0 to $2^{64}-1$	
Other	`Float`	23b sign, 8b exp	`349.99f` `349.99.toFloat()`

	`Double`	52b sign, 11b exp	`349.99`
			`1E6 // yields: 1000000.0`
			`3.49E-5 // yields: 0.0000349`

	`Char`		`'a'  '\t'  '\n'  '\''  '\"'  '\\'  '\?'` `'\u03BB' // yields 'λ'`

	`String`		`"Hello \"World\"\?\n"`
			`"""|Hello 'World'?`
			`|""".trimMargin() // yields "Hello \"World\"\?\n"`

	`Boolean`	`false`, `true`	`"true".toBoolean()`

Classes

Classes

Classes in Kotlin are declared using the keyword `class`.

```
class Pos                   // minimal class definition
```

```
fun main() {
    var myPos = Pos()  // create an instance (no new keyword)
}
```

Classes – Primary Constructor

The primary constructor is a part of the class header.

```
class Pos(x: Double, y: Double)  // one constructor only
```

```
fun main() {
    var myPos = Pos(3.0, 4.0)
}
```

Classes – Initializer

The primary constructor cannot contain any code. Initialization code can be placed in initializer blocks prefixed with the `init` keyword.

```
class Pos(x: Double, y: Double) {
    init {
        println("x: $x; y: $y")
    }
}
```

```
fun main() {
    var myPos = Pos(3.0, 4.0) // prints: x: 3.0; y: 4.0
}
```

Classes – Properties

Properties of a class can be listed in its declaration by adding the `val` or `var` keyword, or in its body.

```
class Pos(val x: Double, val y: Double) {
    val eucl = Math.sqrt(x * x + y * y)
}
```

```
fun main() {
    var myPos = Pos(3.0, 4.0)
    println("x: ${myPos.x}; y: ${myPos.y}; eucl: ${myPos.eucl}")
    // prints: x: 3.0; y: 4.0; eucl: 5.0
}
```

Classes – Properties & Initializer block(s)

Properties and initializers of a class are evaluated top to bottom.

```
class Pos(val x: Double, val y: Double) {
    val eucl = Math.sqrt(x * x + y * y)

    init {
        println("x: $x; y: $y; eucl: $eucl")
    }
}

fun main() {
    var myPos = Pos(3.0, 4.0) // prints: x 3.0; y: 4.0; eucl: 5.0
}
```

Classes – Secondary Constructor(s)

A class can also declare secondary constructors, which are prefixed with `constructor`. Each secondary constructor needs to delegate, directly or indirectly, to the primary constructor.

```
class Pos(val x: Double, val y: Double) {
    val eucl = Math.sqrt(x * x + y * y)

    constructor(pos: Pos) : this(pos.x, pos.y) {
        println("Copy constructor")
    }
}

fun main() {
    var myPos = Pos(3.0, 4.0)
    var myCopy = Pos(myPos)   // prints: Copy constructor
}
```

Code in initializer blocks can be considered part of the primary constructor.

Classes – Anonymous Classes

Anonymous classes are created using the `object` keyword.

```
fun main() {

    val myPos = object { // prints: myPos created
        var x = 3.0
        var y = 4.0
        init {
            println("Anonymous class instantiated...")
        }
        fun getLength() : Double {
            return sqrt(x * x + y * y)
        }
    }

    println(myPos.getLength()) // prints: 5.0
}
```

Like anonymous functions, these classes are useful for one-time use.

Classes – Scope Function *apply*

Scope functions execute code within the context of an object.

```
class Course(val name: String) {
    var grade = -1
    var avg = -1

    fun reset() {
       grade = -1
       avg = -1
    }
}

fun main() {
    val cs246 = Course("CS246").apply {
       grade = 92
       avg = 91
    }
}
```

All code within *apply* is executed on the object of class `Course`; within *apply*, the object has the identifier `this`; *apply* yields the object.

Classes – Scope Function *apply* and *also*

Some additional examples:

```
fun main() {
    val cs246 = Course("CS246").apply {
        grade = 92
    }
    cs246.apply {
        grade = 95
    }.print() // prints: Course: CS246, 95
}
```

The scope function *also* works as *apply*, but the object is labelled **it**:

```
fun main() {
    val cs246 = Course("CS246").apply {
        grade = 92
    }
    cs246.also {
        println("Resetting mark of ${it.name}...")
    }.reset()
}
```

Classes – Inheritance, properties

Parent classes and overwritable properties must be declared `open`.
Overwriting properties must be modified with `override`.

```
open class Pos(val x: Double, val y: Double) {
    open val eucl = Math.sqrt(x * x + y * y)
}

class Pos3D(x: Double, y: Double, val z: Double, w: Double = 1.0):
Pos(x, y) {
    override val eucl = Math.sqrt((x * x + y * y + z * z) / w)
}

fun main() {
    val myPos = Pos3D(3.0, 4.0, 12.0)
    println("eucl: ${myPos.eucl}")    // prints: eucl: 13.0
}
```

Classes – Inheritance, functions

Overwritable functions must be declared `open`. Overwriting functions must be modified with `override`; access to the parent class via the keyword `super`.

```
open class Pos(val x: Double, val y: Double) {
    open fun calcLen() : Double {
        return Math.sqrt(x * x + y * y)
    }
}

class Pos3D(x: Double, y: Double, val z: Double, w: Double = 1.0):
Pos(x, y) {
    override fun calcLen() : Double {
        return Math.sqrt(super.calcLen() * super.calcLen() + z * z)
    }
}

fun main() {
    val myPos = Pos3D(3.0, 4.0, 12.0)
    println(myPos.calcLen())  // prints: 13.0
}
```

Classes – Abstract classes and interfaces

```
interface Comparable {
    fun isEqual(other: Any): Boolean
}

abstract class Pos(var x: Double, var y: Double) {
    abstract fun printFun()
}

class Pos3D(x: Double, y: Double, var z: Double): Pos(x, y), Comparable {
    override fun printFun() {
        println("CS349!")
    }
    override fun isEqual(other: Any): Boolean {
        return when (other) {
            is Pos -> (x == other.x) and (y == other.y) and (z == other.z)
            else -> false
        }
    }
}
```

Classes – Anonymous Classes

Anonymous classes can sub-class existing `open` or `abstract` classes, or interfaces:

```
abstract class Pos(var x: Double, var y: Double) {
    abstract fun printFun()
}

val myPos = object: Pos(3.0, 4.0) {
    fun getLength() : Double {
        return sqrt(super.x.pow(2) + super.y.pow(2))
    }
    override fun printFun() {
        println("CS349!")
    }

 println(myPos.getLength()) // prints: 5.0
 println(myPos.printFun())  // prints: CS349!
```

Classes – Data classes

A class whose main purpose is to hold data. Kotlin automatically provides a `copy` and an enhanced `toString` function.

```
data class Pos(val x: Double, val y: Double)

fun main() {
    var myPos = Pos(3.0, 4.0)
    println("(${pos.x},${pos.y})") // prints: (3.0,4.0)

    val anotherPos = myPos.copy(y=5.0)
    println(anotherPos)           // prints: Pos(x=3.0, y=5.0)
}
```

Classes – Enum classes

```
enum class Direction {
    NORTH, EAST, SOUTH, WEST
}
```

Enumeration entries behave like sub-classes of the Enum class:

```
enum class Direction {
    NORTH { override fun turnRight(): Direction { return EAST } },
    EAST { override fun turnRight() = SOUTH }, // shortened from above
    SOUTH { override fun turnRight() = WEST },
    WEST { override fun turnRight() = NORTH };

    abstract fun turnRight(): Direction
}

fun main(args: Array<String>) {
    val direction = Direction.SOUTH
    println(direction.turnRight())  // prints: WEST
```

Scope

Annotations or visibility modifiers go before the constructor or function name.

Kotlin defaults to “public” scope if you omit the modifier (which we will often do in examples).

Modifier	Scope
`public`	Everywhere
`protected`	Class and subclasses
`private`	Class
`internal`	Module

End of the Chapter

- Get you tool-chain up and running!
- While Kotlin works with many programming styles, it is best a adopt a “functional” mindset with your implementation.
- Higher-order functions and anonymous functions (and classes) rule!
- Try to remember “all the small things” that can make your implementation shorter / more legible / “slicker“.

Any further questions?