Introduction to Kotlin

What are we going to cover?

- The nuts and bolts of the language.
- Kotlin as an object oriented language
- Kotlin as a functional language
- Kotlin and Java interoperability

Copyright © 2021 The Third Lane LLC

Nuts and Bolts

- Semi colons are optional!!!
- Top level functions.
- Multiple public classes allowed in one file.
- Type safe language, but specifying type is often optional.

Copyright © 2021 The Third Lane LLC

Val and Var

- Declare variables as **val** or **var**
  - **val** – read only **val**ues
    - NOT necessarily constant – more on this later.
    - MUST be initialized.
  - **var –** read/write **var**iables
- Type specified after variable name
  - val count: Int
- Type is not required if it can be inferred by the compiler
  - val count = 0
  - might be a good idea to be explicit anyway.
  - IDE can help
- Examples : 1Intro/2Variables

Copyright © 2021 The Third Lane LLC

Functions

- Can be declared at the top level, outside of a class.
- Top level functions are called directly
  - Converted into static method calls in the byte code.
- Return type is specified after the function name and arguments
  - fun doSomething(age: Int, name: String) : String {…}
- No return type implies **Unit**
  - fun foo() {} is the same as
  - fun foo() : Unit {}

Copyright © 2021 The Third Lane LLC

Functions

- Functions can have default values for arguments.
  - fun doSomething(age: Int = 10, name: String) ...
- Can be called using named arguments
  - doSomething(name = “Joe”)
- Can be overloaded as in Java
  - But default values and named arguments reduce the need for overloading.

Copyright © 2021 The Third Lane LLC

Single Expression Functions

- Functions with a single expression for the body can be expressed in a simpler way with an assignment.
- e.g. this function
  - fun doSomething(input: String) : String {

    return input + input.length

    }
- can be written as
  - **fun doSomething(input: String) = input + input.length**

Copyright © 2021 The Third Lane LLC

Nullability

- All types have a Nullable version, which is a distinct type
  - val str: String
  - val nullableStr: String? = null
- Can only assign nulls to nullable types
- Null safe operators to enforce null safety
  - ?.
  - ?: (elvis operator)
- Safer and more convenient (usually) than explicit null checks
- Examples: 1Intro/3NullNess

Copyright © 2021 The Third Lane LLC

Initialization

- **val**s *must* be initialized when created.
  - sometimes in a constructor or **init** method
- **var**s too, though with some loopholes.
  - Nullable vars can be initialized to null and then reset later
  - If not nullable, then **lateinit** could possibly be your friend.
- Examples: 1Intro/4Lateinit

Copyright © 2021 The Third Lane LLC

Control Flow

- **if/when**
  - Both are *expressions* in Kotlin – both can return values
- val x = if(…) 0 else 1 is used in place of the “ternary” operator.
- Prefer **when** for more complex cases.
- **when** is more flexible than a switch in Java
  - arbitrary condition expressions, not just equality
  - smart casting of arguments
  - can be used without an argument, working like a generalized if statement.
- Examples: 1Intro/5IfAndWhen

Copyright © 2021 The Third Lane LLC

Loops And Ranges

- No *for(int I = 0; I < 10; i++)* loop
- Many other variations available
- Ranges
  - are also a convenient and useful option for fixed length iterations
  - Can be used to do boundary checks
    - j in 1..10
  - Can create ranges for any class implementing Comparable.
  - Can create custom ranges to iterate over arbitrary classes
    - Custom **rangeTo** implementation that returns an Iterable
- Examples: 1Intro/6Loops, 7Ranges

Copyright © 2021 The Third Lane LLC

Exceptions

- All Exceptions in Kotlin are unchecked
  - No need to try and catch.
  - But requires extra vigilance from you to make sure you are doing the right thing.
  - try/catch construct is also an expression
    - val x = try { doSomething() } catch (e:Exception) { -1 }
- Can use annotations for Exception declarations. Useful for Java – no effect in Kotlin
  - @Throws(NoSuchFileException::class)

    fun doSomethingWithFile(fileName: String) {

    val f = FileInputStream(fileName)

    …

    }
- Examples: 1Intro/8Exceptions

Copyright © 2021 The Third Lane LLC

Classes

- Usually declare a **Primary constructor.**
- Can also have other constructors.
  - Must call primary constructor from other constructors.
- Can do initialization in **init** blocks.
- Instances are created by calling a Constructor directly – no new keyword.
- Default argument values and named arguments often reduce the need for multiple constructors.
- Visibility levels
  - **public –** the default, as opposed to package in java
  - **protected –** This class and it's super classes
  - **internal –** All classes in the same *module*.
    - A module in kotlin is a set of Kotlin files compiled together. e.g maven or gradle project, an IntelliJ IDEA module, a set of files compiled together with kotlinc, etc.
- Default visibility of classes, properties and methods in Kotlin is **public.**
  - In Java it is package

- Examples ClassesEtc/Classes, PrimaryConstructor, OtherConstructors

Copyright © 2021 The Third Lane LLC

Class Properties

- Implemented using val and var
  - implemented by compiler as getters and setters
- Can be declared private
- Can have custom getters and setters (var)
- Examples: ClassesEtc/Properties

Copyright © 2021 The Third Lane LLC

Class Inheritance

- Classes are **final** by default
  - Need to **open** them to use as super class
- Methods are **final** by default
  - Need to **open** to be able to override
- **override** key word is *required* for a method that is overriding a super class method
  - unlike @Override annotation in Java
- Examples in ClassesEtc

Copyright © 2021 The Third Lane LLC

Class Inheritance

- Interfaces
  - can have default implementations
  - can specify properties which have to be created by implementing classes
    - like declaring get/set methods in an interface
- Abstract classes
  - can have state
- Examples: ClassesEtc/Interfaces, AbstractClasses

Copyright © 2021 The Third Lane LLC

Data Classes

- Data classes
  - Have to declare all relevant properties in Primary constructor
  - data class Person(var name: String, var rank: Int)
- Compiler gives you some methods for free
  - toString
  - equals and hashCode
  - copy – to make selective copies of an existing instance
    - val i2 = i1.copy(name = “othername”)
  - componentN methods – Useful for destructuring
    - val(name, rank) = i2
    - requires component1 and component2 methods in class which will return the name and the rank
- Examples: ClassesEtc/DataClasses

Copyright © 2021 The Third Lane LLC

Objects

- Singletons with help from the compiler
- An **object** represents both a type and the only instance of that type that will ever exist.
- object methods are called like static functions from Kotlin code, using the *type* of the object.
  - To allow this from Java you need to use @**JvmStatic** on the method
- **Companion** objects are special.
  - Declared in a class
    - Closest thing to the Java notion of ‘static’ that you can get in Kotlin
  - Their methods and properties can be invoked directly through the enclosing class
  - Again, @**JvmStatic** allow ‘static like’ calls from Java too.
- Examples in ClassesEtc/Objects

Copyright © 2021 The Third Lane LLC

Extension Functions

- A surprising amount of Kotlin functionality is implemented using the feature of **Extension Functions.**
- It is a way of creating a function *as if* it is a member of some class X, *without changing the code of that class.*
  - You can call it on objects of type X.
  - You can only access the public parts of the object in the function.
  - The Kotlin compiler fakes it by creating a static function.
- Can similarly create **Extension Properties** for a class
- Examples in 4FunctionalFun/BasicExtensions and ExtensionProperties

Copyright © 2021 The Third Lane LLC

Functional Types and Lambdas

- Functional programming often requires passing functions as arguments, or returning them as results.
  - Such functions are referred to as **high order functions.**
  - e.g. many of the extension methods in the Kotlin collection library take a function as an argument
    - filter(predicate: (String) → Boolean)
- So functions are often *first class citizens* in such languages.
  - You can declare variables of function type
  - You can declare the return type of functions to be a function type. i.e. this function returns a function
  - Java fakes this with the use of *Functional Interfaces*

Copyright © 2021 The Third Lane LLC

Functional Types and Lambdas

- Functions are a first class type in Kotlin
  - fun doSomething(String) : Int { return 10 }
  - **val funPtr: (String) → Int = ::doSomething**
  - **::doSomething** is a *method reference*
  - **funPtr** is a read only variable that should point at a function that takes a String and returns an Int
- Syntax for funtional types is
  - (arg1, arg2, …) → return type
  - e.g.
    - (String, Int) → Unit //takes a String and an Int and returns nothing
    - () → String //takes no arguments and returns a String

Copyright © 2021 The Third Lane LLC

Functional Types and Lambdas

- Lambdas give you a syntax for creating functions “on the fly”
  - val doubleLength: (String) → Int = **{ arg → arg.length * 2 }**
  - or val doubleLength = **{ arg: String → arg.length * 2 }**
- Syntax is
  - {arg1, arg2, … → lambda_body}
  - the last expression in the body becomes the return from non Unit returning lambdas.
- If only one argument, you can omit it and refer to it in the body as **it**
  - val doubleLength: (String) → Int = **{ it.length * 2 }**

Copyright © 2021 The Third Lane LLC

Functional Types and Lambdas

- If the last argument of a method is a function, you can pass in the function as a lambda *outside the parentheses. e.g*
  - fun checkInput(input: String, predicate: (String) → Boolean) { … }
  - can call with two arguments in the parentheses
    - checkInput(“A message”, { str → str.length > 10 })
  - But better Kotlin form to call with the lambda *outside the parentheses*
    - checkInput(“A message”) **{ str → str.length > 10 }**
  - Can also use the implicit **it** argument for 1 argument lambdas
    - checkInput(“A message”) { **it**.length > 10 }

Copyright © 2021 The Third Lane LLC

Lambda with Receiver

- What extensions functions are to regular functions, **lambdas with receiver** are to lambdas
  - Extension Function
    - fun **String.lengthSquared()** {
      return this.length * this.length
      }
    - called as “hello”.lengthSquared()
  - Ordinary lambda
    - val lengthSquaredLambda: **(String)** → **Int** = { str → str.length * str.length }
    - called as lengthSquaredLambda(“hello”)
  - **Lambda With Receiver**
    - val lengthSquaredLambdaWithReceiver: **String.()** → **Int** = { this.length * this.length }
    - called as “hello”.lengthSquaredLambdeWithReceiver()
- Very useful for creating DSL’s

Copyright © 2021 The Third Lane LLC

Collection Operations

- filter(pred: (T) → R)
- map(fn: (T) → R)
- flatMap(fn: (T) → Iterable<R>)
- group/groupingBy – create groupings (maps) of collections.
- Operators +=, and -= can be used to add and remove elements from collections.
  - list += 10 – add 10 to the list
  - list += otherList – append one list to another
  - list -= 10 – remove 10 from the list
  - list -= otherList – remove all elements in otherList from list.
  - similar operations for maps
  - Examples in 6Conventions/Conventions.kt

Copyright © 2021 The Third Lane LLC

Sequences

- Kotlin collection operations are **eager.** This means that each operation completes before the next begins.
  - Intermediate collection created at each step
  - For long pipelines and/or large collections this may be expensive.
- You can use **sequenc**es instead. Sequence operations are **lazy**.
  - Each element goes from the beginning to the end of the collection, or gets thrown away.
  - No intermediate collection created.
  - Similar to how the *Streams* library works in Java.
  - Can use Streams too, if you want.
- Examples in 5CollectionOps/5SequenceOps
- You can create sequences from scratch or from existing collections
  - list.asSequence().filter(…).map(…).toList()

Copyright © 2021 The Third Lane LLC

Scope Functions

- Kotlin library provides 5 functions which allow you to execute code within the *context* of some object
  - You pass a lambda into the functions
  - the **context** object is the one you call the functions on
  - The functions vary in the way you access the context object in the lambda, and in what they return
    - context object is available either as **this** or as the first argument, **it** by default
    - return type is either the **context object** or the **return of the lambda** passed in to the function.
  - Examples in 4FunctionalFun/ScopeFunctions.kt

Copyright © 2021 The Third Lane LLC

Scope Functions - apply

- Context object available as **this**
- Returns the **context object**
- Useful for initialization

```
public inline fun <T> T.apply(block: T.() -> Unit): T {
    block()
    return this
}

val ap = Processor().apply {
    id = 10                    //context object available as ‘this’
    name = "Apply"             //this.name = “Apply”
    headers.add("Someheader=SomeValue")
    headers.add("OtherHeader=OtherValue")
    address = "tcp://@xray"
    //return of lambda is ignored.  ‘apply’ returns ‘this’
}
```

Copyright © 2021 The Third Lane LLC

Scope Functions - let

- Context object available as the first argument
- Returns the **result of the lambda**
- Useful for dealing with possibly null objects.
- And/or for using one object as an argument to perform some work and returning another object.

```
public inline fun <T, R> T.let(block: (T) -> R): R {
    return block(this)
}

val str4: String? = getAPossiblyNullStringFromSomewhere()
val size = str4?.let { it ->
    val ucase = it.toUpperCase()
    processUpperCaseString(ucase)

    it.length          //return of lambda becomes return of ‘let’
} ?: -1  //Need this as an "else".
```

Copyright © 2021 The Third Lane LLC

Scope Functions - also

- Context object available as the first argument, **it**
- Returns the **context object**
- Useful for performing side effects

```
public inline fun <T> T.also(block: (T) -> Unit): T {
    block(this)
    return this
}
```

```
val connection1 = getProcessor()?.let { proc ->
    makeConnection(proc).also {
        println("made connection1 to ${it}")
    }
}
```

Copyright © 2021 The Third Lane LLC

Scope Functions - run

- Context object available as **this**
- Returns the **result of the lambda**
- Useful for performing actions on the context to object to create a different result object

```
public inline fun <T, R> T.run(block: T.() -> R): R {
    return block()
}

val processedValue2 = Processor(10).run {
    name = "Toaster"            //context object available as ‘this’
    proxy = "stu.com"           //this.proxy = “stu.com”
    val xa = process1()         //this.process1()
    val xb = process2()
    val xc = process3()
    //String returned from lambda becomes the return of ‘run’
    "${id} ${name}  $xa  $xb $xc"
}
```

Copyright © 2021 The Third Lane LLC

Scope Functions - with

- No context object here. The object to work with is supplied as an **argument**
- Returns the **lambda result**
- Useful for working with an object when you don’t care about the results
  - “ with this object, do the following.”

```
public inline fun <T, R> with(receiver: T, block: T.() -> R): R {
    return receiver.block()
  }

with(Processor()) {
    val x = process2()     //context object available as ‘this’
    val y = process3()     //this.process3()
    //do some other work with x and y
}
```

Copyright © 2021 The Third Lane LLC

The End