Kotlin for Java Developers

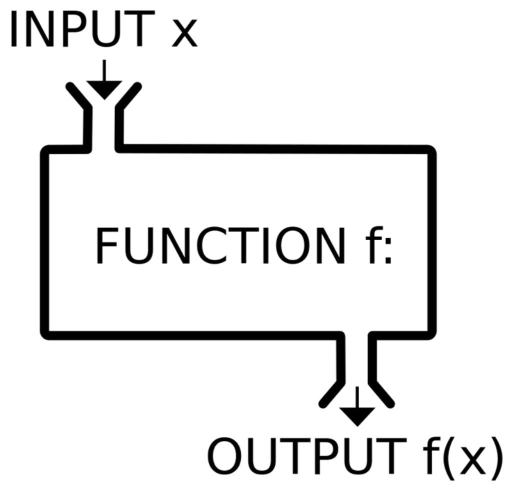
Jeriel Ng (@jerielng)
Mobile Software Developer at NCR
Auburn University 2019

Overview of Kotlin

- Functional programming language
- Developed by JetBrains (see also: **Android Studio**)
- Runs on JVM
- Popular for Android development
    - 2017: Supported for Android
    - 2019: Primary language for Android
- Not just for Android
    - Libraries
    - Data science & machine learning
    - Backend web

What is Functional Programming?

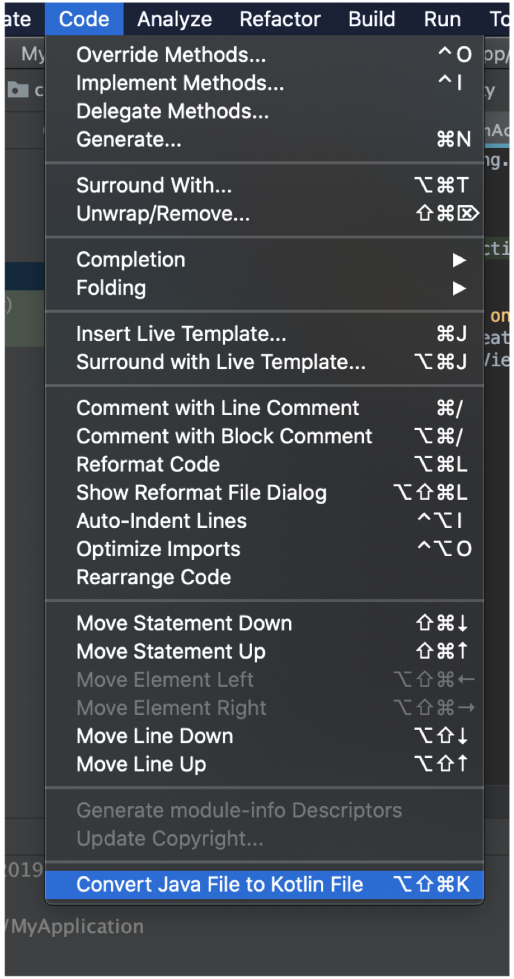
- Paradigm shift from object-oriented programming
- Similar to mathematical functions
  - f(x): input x produces output y
- Strive to achieve immutability
- Strive to produce no side effects



Interoperability with Java

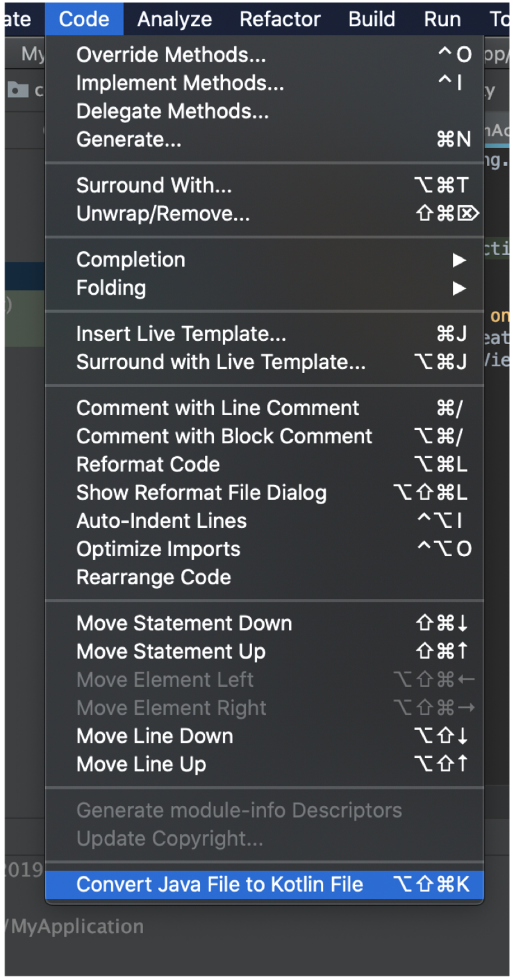
- Runs on JVM
- Easy conversion from Java to Kotlin
    - Android Studio
    - IntelliJ

$$\begin{tikzcd} \text{Java} \arrow[r, shift left, "\text{calls}"] & \text{Kotlin} \arrow[l, shift left, "\text{calls}"] \end{tikzcd}$$



Interoperability with Java

- Auto-conversion pitfalls:
    - Class constructors
    - Data classes
    - Static modifiers
    - Nullable types



Type Inference

- Property type inference -> `val`, `var`
- Functions use the keyword -> `fun`
  - Must specify return type
  - Must specify argument types

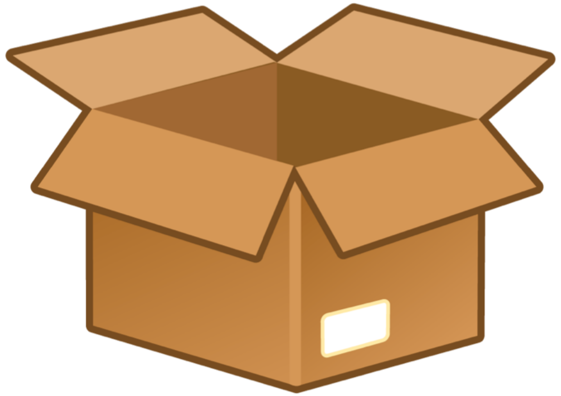
```
fun veryExcitingFunction() {
    var mutableProp = “Property”
    val immutableProp = “Also property”
    var explicitInteger: Int = 5
}

fun anotherExcitingFunction(): Boolean {
    return if (isItTrue) true else false
}
```

Null Safety: Nullable Types

Fun Fact: In Swift, we call these "optionals"

```
var helloString: String? = null
```



Acts as a container for the specified type

Could contain a `String`
Could contain nothing
Would never contain an `Int`, `Boolean`, etc.

Null Safety: Operators

Safe Call Operator

```
object?.performFunction()
```

"Does it exist? If so, perform this function from it."

Not-Null Assertion Operator

```
object!!.performFunction()
```

"I know for sure it exists. If I'm wrong, you can crash my app."

Elvis Operator

```
assignment = myString ?: “”
```

"I'd like to perform this if this object exists. If not, try this instead."

Type Checking & Casting

Type Checking -> `is`

```
val excitingDouble: Double = 5.0
if (excitingDouble is String) {
    print("This will not execute.")
else {
    print("Double failed the type check as a String.")
}
```

Type Casting -> `as`, `as?`

```
var myCoolString = "Type Casting Exercise"
var crashFailure: Int = myCoolString as Int // This will crash
var safeFailure: Int? = myCoolString as? Int // This will set safeFailure to null
```

Scope Functions

```
.let
.run
.apply
.with
.also
```

- Executes a block pertaining to a specific object
- Differentiating between each one:
  - How is the object being referred to (`this` or `it`)
  - What is being returned
- Example:

```
object.doThis()
object.doThat()
object.done()
```

Normal Flow

```
object.run {
      doThis()
      doThat()
      done()
}
```

Using a Scope Function

For a deeper look:
[Check out this Medium article by Elye]

Useful Combos

```
int a = someClass.retrieveA();
if (a != null) {
    a.setStatus("Retrieved");
} else {
    a = new ConstructorForA();
    a.setStatus("Created");
}

// Do some stuff with `a` later on
```

Java

```
someClass.retrieveA()?.let { a ->
    a.status = "Retrieved"
} ?: run {
    var a = ConstructorForA()
    a.status = "Created"
}

// Do some stuff with `a` later on
```

Kotlin

Lambdas and Higher-Order Functions

- Higher-Order functions
  - Can take functions as arguments
  - Can return functions
- Lambda Expressions
  - Undeclared functions that can be passed like variables

```
val multiplyLambda: (Int) -> Int = { input ->
    input * 2
}

fun higherOrder(lambda: (Int) -> Int) {
    val someValue = 5
    val result = lambda(someValue)
    // Some other stuff
}
```

The `static` Keyword: Companion Objects

- `static` keyword doesn't exist -> use `companion` objects instead

```
class MyCoolCar {
    static void  myCoolFunction() {
        ...
    }
}
```

Java

```
class MyCoolCar {
    companion object MyCompanionClass {
        fun myCoolFunction() {
            ...
        }
    }
}
```

Kotlin

Calling a companion object property/function:
Inside Kotlin: `MyCoolCar.myCoolFunction()`
Inside Java: `MyCoolCar.Companion.myCoolFunction()`

The `static` Keyword: `@JvmStatic`

- Alternative Hack: annotate with `@JvmStatic`
  - For interactions with Java classes
  - Compiler generates both a static and instance version of the method/variable

```
class MyCoolCar {
    @JvmStatic
    fun myCoolFunction() {
        ...
    }
}
```

→

Inside Kotlin: `MyCoolCar.myCoolFunction()`
Inside Java: `MyCoolCar.myCoolFunction()`

Syntactic Sugar: Lifting assignments

```
if (condition) {
    return 1;
} else {
    return 0;
}
```

Java

```
return if (condition) {
    1
} else {
    0
}
```

Kotlin

Syntactic Sugar: Lifting assignments & `when`

```
String a;
switch (x) {
    case 0:
        a = “0”;
        break;
    case 1:
        a = “1”;
        break;
    default:
        a = “Other”;
        break;
}
```

Java

```
val a = when(x) {
    0 -> “0”
    1 -> “1”
    else -> “Other”
}
```

Kotlin

Additional Thoughts

- Is it better? Not necessarily
    - Popular opinion: Kotlin is easier to read
    - Android is “Kotlin-first”
    - Kotlin is open source
- Converting a Java codebase into Kotlin
    - It’s okay to have a mix of both
    - Take one file at a time
    - Start any new files in Kotlin
- **If you use Kotlin, write like a Kotlin programmer, not like a Java programmer**

Further Resources

- Kotlin Documentation
- Kotlin vs Java Syntax
- Fragmented Podcast
  - Episode 83
  - Episode 120
  - Episode 121
- Talking Kotlin Podcast (If you're interested in other use cases of Kotlin)
- KotlinConf (Annual Kotlin conference from JetBrains)

Questions