# Mobile Application Development

Produced by

David Drohan (ddrohan@wit.ie)
Dr. Siobhan Drohan (sdrohan@wit.ie)

Department of Computing & Mathematics
Waterford Institute of Technology
http://www.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

# Introducing Kotlin Syntax - Part 2.3

# Agenda for Part 2

- ❑ Writing Classes (properties and fields)
- ❑ Data Classes (just for data)
- ❑ Collections: Arrays and Collections
- ❑ Collections: *in* operator and lambdas
- ❑ Arguments (default and named)

# Agenda for Part 2

- ❑ Writing Classes (properties and fields)
- ❑ Data Classes (just for data)
- ❑ Collections: Arrays and Collections
- ❑ Collections: *in* operator and lambdas
- ❑ Arguments (default and named)

# Collections

The **in** operator and using lambdas

# Collections – iterating using the in operator

```kotlin
fun main() {
    val items = listOf("apple", "banana", "kiwifruit")
    for (item in items) {
        println(item)
    }
}
```

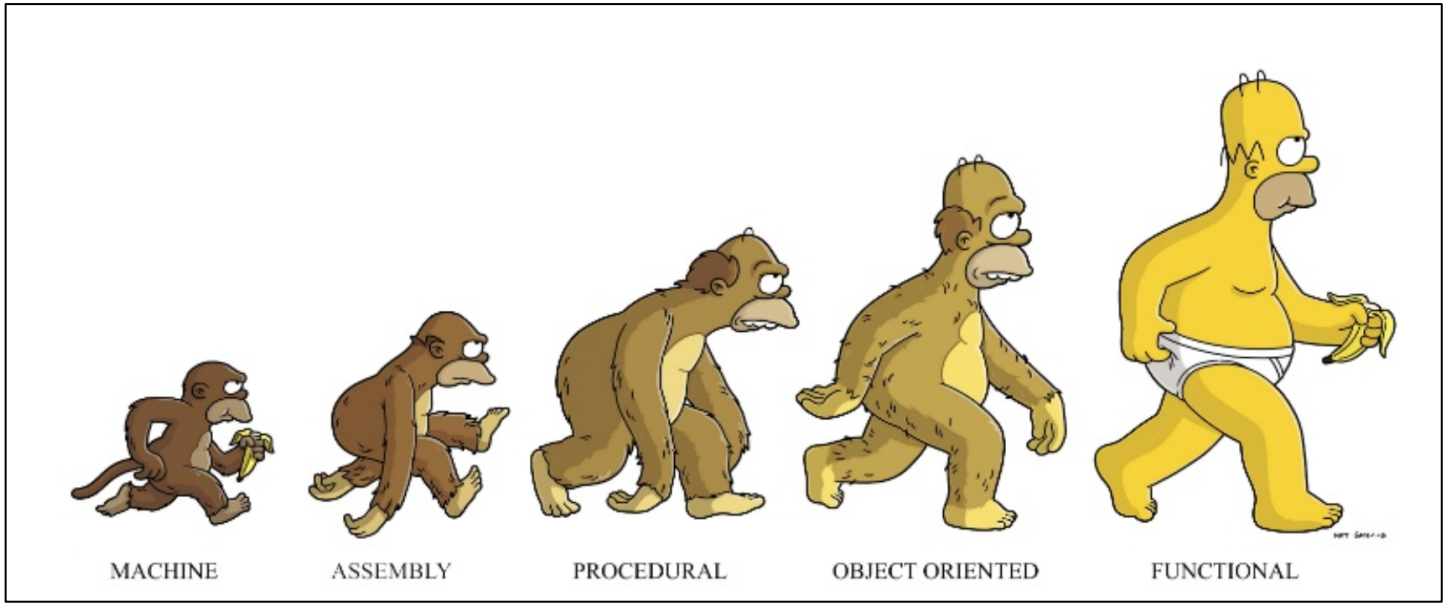
```
apple
banana
kiwifruit
```

# Collections – checking if collection contains an object

```
fun main() {
    val items = setOf("apple", "banana", "kiwifruit")
    when {
        "orange" in items -> println("juicy")
        "apple" in items -> println("apple is fine too")
    }
}
```

```
apple is fine too
```

# Kotlin…. functional programming is prevalent!



# Quick Overview – Functional Programming

- In a nutshell, its a style of programming where you focus on transforming data through the use of small expressions that ideally don't contain side effects.

- In other words, when you call "myfun(a, b)", it will always return the same result. This is achieved by immutable data typical of a functional language.

- With the functional approach, we are expressing what we want done, not how to do it.

# Quick Overview – Functional Programming

- The main characteristics of functional programming languages are :
  - Designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.
  - Supports **higher-order functions** and lazy evaluation features.
  - Doesn't support flow Controls like loop statements and conditional statements like If-Else and Switch/When Statements. They directly use the functions and functional calls.
  - Like OOP, functional programming languages support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism.

# Quick Overview – lambdas

- Lambdas aren't unique to Kotlin and have been around for many years in many other languages (very similar to Java)

- Lambda expressions (or lambda functions) are essentially blocks of code (anonymous functions) that can be assigned to variables, passed as an argument to methods, or returned from a function call, in languages that support **higher-order functions.**

# Quick Overview – lambdas

- To define a Lambda (expression) we say something like

```
val lambdaName : Type = { arguementList -> codeBody }
```

- Note : The only part of a lambda which **isn't optional** is the *codeBody.*
- The *arguementList* can be skipped (omitted) when defining **at most** one **argument** and the *Type* can often be inferred.
- We don't always need a variable either - lambda can be passed directly as a **function argument**.
- The *type* of the last command within a lambda block is the return type.

# Quick Overview – lambdas

- Here, a lambda expression is assigned to variable **`greeting`**. The expression doesn't accept any parameters and doesn't return any value in this program.

```
fun main(args: Array<String>) {
    val greeting = { println("Hello!")}
    // invoking function
    greeting()
}
```

- When you run the program, the output will be:

```
Hello!
```

# Quick Overview – lambdas

- Here, we have a lambda expression that accepts two integers as parameters, and returns the product of those two integers.

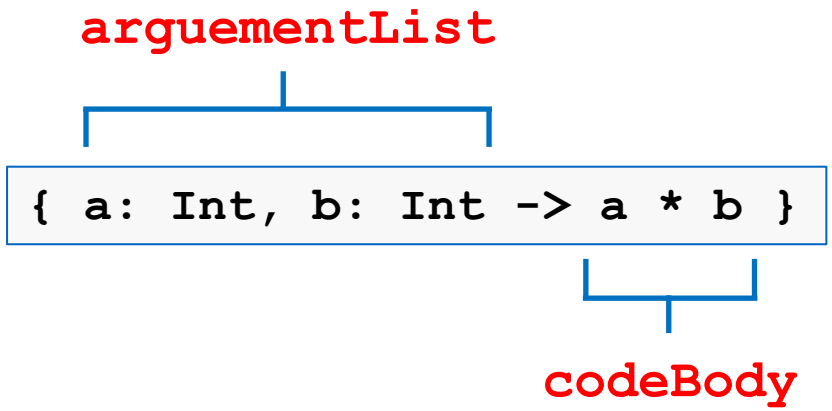
```
fun main(args: Array<String>) {
    val product = { a: Int, b: Int -> a * b }
    val result = product(9, 3)
    println(result)
}
```

- When you run the program, the output will be:

```
27
```

# Quick Overview – lambdas

- In the previous example, the lambda expression is:




- Note again, a lambda expression is enclosed inside curly braces.

# Collections – lambdas

- Lambdas are frequently used while working with **collections**.
- There are several built-in functions available (next few slides) in standard-library that take lambdas to make tasks easier.
- If the lambda expression accepts only one parameter/argument (a list of objects perhaps? (next slide)) you can refer to the argument by using the keyword "it".
- It is an implicit variable and can be omitted when it refers to a particular object in a single list argument.

# Collections – lambdas

- Using lambda expressions to filter and map collections

```
fun main(args: Array<String>) {

  val fruits = listOf ("Banana", "Avocado", "Apple", "Kiwi")
  fruits.forEach  { it -> println(it) }
}
```

it -> is optional here

No need for function brackets ( . . . )

```
Console
<terminated> Config - Main.kt [Java Application] C:\Program Files\J
Banana
Avocado
Apple
Kiwi
```

# Collections – lambdas

- Using lambda expressions to filter and map collections

```
fun main(args: Array<String>) {

  val fruits = listOf ("Banana", "Avocado", "Apple", "Kiwi")
  fruits.filter   { it.startsWith("A") }
        .forEach  { println(it) }
}
```

Console

```
<terminated> Config - Main.kt [Java Application] C:\Program Files\Java\jdk1.8.0_1
Avocado
Apple
```

# Collections – lambdas

- Using lambda expressions to filter and map collections

```
fun main(args: Array<String>) {

  val fruits = listOf ("Banana", "Avocado", "Apple", "Kiwi")
  fruits.filter   { it.startsWith("A") }
        .sortedBy { it }
        .forEach  { println(it) }
}
```

Console

```
<terminated> Config - Main.kt [Java Application] C:\Program Fi
Apple
Avocado
```

# Collections – lambdas

- Using lambda expressions to filter and map collections

```
fun main(args: Array<String>) {

  val fruits = listOf ("Banana", "Avocado", "Apple", "Kiwi")
  fruits.filter   { it.startsWith("A") }
        .sortedBy { it }
        .map      { it.toUpperCase() }
        .forEach  { println(it) }
}
```

```
Console
<terminated> Config - Main.kt [Java Application] C:\Progr
APPLE
AVOCADO
```

# Collections – sample functions

```
fun main(args: Array<String>) {

  val numbers = listOf(-42, 17, 13, -9, 12)
  println(numbers)

  println("First element:         " + numbers.first())
  println("Last element:          " + numbers.last())
  println("Smallest element:      " + numbers.min())
  println("Sum of elements:       " + numbers.foldRight(0, { a, b -> a + b }))
  println("First two elements:    " + numbers.take(2))
  println("All except first two: " + numbers.drop(2))
  println(numbers)
}
```

Console  Gradle Tasks  Gradle Execut

```
<terminated> Config - Main.kt [Java Application] C:\
[-42, 17, 13, -9, 12]
First element:         -42
Last element:          12
Smallest element:      -42
Sum of elements:       -9
First two elements:    [-42, 17]
All except first two: [13, -9, 12]
[-42, 17, 13, -9, 12]
```

# Collections – sample functions

```
fun main(args: Array<String>) {

  val numbers = listOf(-42, 17, 13, -9, 12)
  println(numbers)

  // New list only containing non-negative numbers
  val nonNegative = numbers.filter { it >= 0 }
  println(nonNegative)

  // Double each element
  numbers.forEach { print("${it * 2} ") }
  println();

  // Output Even elements only
  numbers.filter {it % 2 == 0}
         .forEach {print ("$it " )}
  println();

}
```

Console

```
<terminated> Config - Main.kt [Java Applicat
[-42, 17, 13, -9, 12]
[17, 13, 12]
-84 34 26 -18 24
-42 12
```

# Sets and Lambdas

```kotlin
fun main(args: Array<String>) {

  val numbers = setOf(-42, 17, 13, -9, 12)
  println(numbers)

  // New list only containing non-negative numbers
  val nonNegative = numbers.filter { it >= 0 }
  println(nonNegative)

  // Double each element
  numbers.forEach { print("${it * 2} ") }
  println();

  // Output Even elements only
  numbers.filter {it % 2 == 0}
         .forEach {print ("$it " )}
  println();

}
```

Console

```
<terminated> Config - Main.kt [Java Applicat
[-42, 17, 13, -9, 12]
[17, 13, 12]
-84 34 26 -18 24
-42 12
```

# Maps and Lambdas

```
fun main(args: Array<String>) {

    val counties = mapOf(
              Pair("W","Waterford"),
              Pair("C","Cork"),
              Pair ("D","Dublin") )

     println("All items:");
     counties.forEach {print(it); print (", ")}

     println("\n\nSorted:");
     counties.toSortedMap()
              .forEach {print(it); print (", ")}

     println("\n\nFilter, max 6 chars:");
     counties.filter {it.value.length <= 6 }
             .forEach {print(it); print (", ")}

     println("\n\nFilter, sorted and between 5 & 9 chars:");
     counties.filterValues {it.length >= 5 && it.length <=9}
              .toSortedMap()
              .forEach {print(it); print (", ")}
}
```

Console

```
<terminated> Config - Main.kt [Java Application] C:\Progran
All items:
W=Waterford, C=Cork, D=Dublin,

Sorted:
C=Cork, D=Dublin, W=Waterford,

Filter, max 6 chars:
C=Cork, D=Dublin,

Filter, sorted and between 5 & 9 chars:
D=Dublin, W=Waterford,
```

# Arguments

default and named

# Default Arguments (optional)

- In Java, you often have to duplicate code in order to define different variants of a method or constructor (i.e. overloading).

- Kotlin simplifies this by using **default values** for arguments (i.e. makes them optional arguments).

# Default Arguments (optional)

Primary Constructor

```
class NutritionFacts( val foodName: String,
                      val calories: Int,
                      val protein: Int = 0,
                      val carbohydrates: Int = 0,
                      val fat: Int = 0,
                      val description: String = "" )
{
}
```

Optional Parameters

Some possible constructor calls

```
val pizza  = NutritionFacts("Pizza",  442, 12, 27, 24,
                                        "Deep Pan Pizza")
val pasta = NutritionFacts("Pasta", 371, 14, 25, 11)
val soup  = NutritionFacts("Soup",  210)
```

# Named Arguments

```
class NutritionFacts( val foodName: String,
                      val calories: Int,
                      val protein: Int = 0,
                      val carbohydrates: Int = 0,
                      val fat: Int = 0,
                      val description: String = "" )
{
}
```

Naming arguments make your code more readable

Some possible constructor calls

```
val pasta   = NutritionFacts("Pasta", 371, 14, 25, 11)
val burger  = NutritionFacts("Hamburger", calories = 541, fat = 33,
                                                        protein = 14)
val rice      = NutritionFacts("Rice", 312, carbohydrates = 23,
                                                  description = “Grains")
```

# Some additional sources for exploration:

| | |
|---|---|
| Inheritance | https://www.programiz.com/kotlin-programming/inheritance |
| Interfaces | https://www.programiz.com/kotlin-programming/interfaces |
| Collections | https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index.html |
| Try examples online | https://try.kotlinlang.org/#/Examples/Hello,%20world!/Simplest%20version/Simplest%20version.kt |
| Encapsulation & Polymorphism | https://medium.com/@napperley/kotlin-tutorial-12-encapsulation-and-polymorphism-6e5a150f25e1 |
| Spek (testing) | https://objectpartners.com/2016/02/23/an-introduction-to-kotlin/<br>https://github.com/mike-plummer/KotlinCalendar |


# References

Sources:

http://kotlinlang.org/docs/reference/basic-syntax.html
http://petersommerhoff.com/dev/kotlin/kotlin-for-java-devs/
https://www.programiz.com/kotlin-programming
https://www.baeldung.com/kotlin-lambda-expressions
https://www.programiz.com/kotlin-programming/lambdas
https://medium.com/@napperley/kotlin-tutorial-5-basic-collections-3f114996692b


Thanks.