1

2

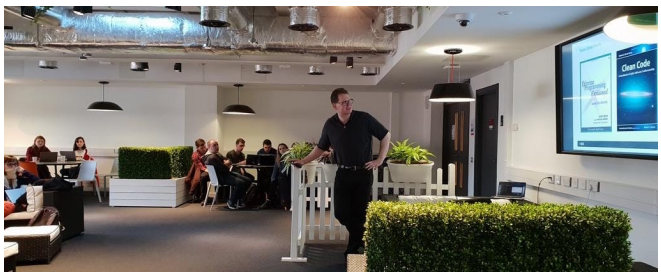
Welcome!

- Let's get acquainted...

INSTIL

3

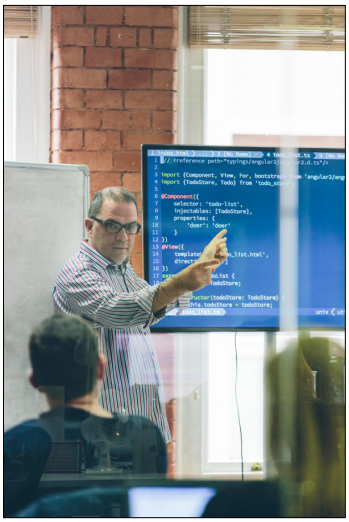
Garth (not Gareth)



INSTIL

4

Gareth (not Garth)



INSTIL

5

We Are JetBrains Partners

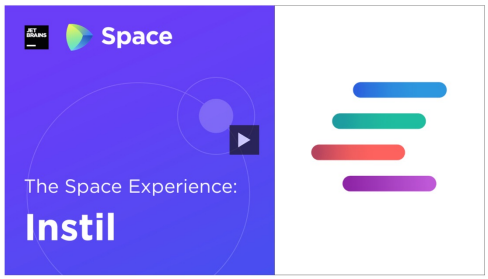
Accredited Kotlin training provider

Accredited JetBrains Space partner

Garth is a Kotlin Google Dev Expert

«Since starting to offer virtual deliveries in April, we've effectively relaunched the training side of the business and are now busier than ever. Space has been a huge part of that success. It's an umbrella that provides and integrates everything we need to deliver our products in a distributed manner.»

— Garth Gilmour, Head of Learning, Instil



INSTIL

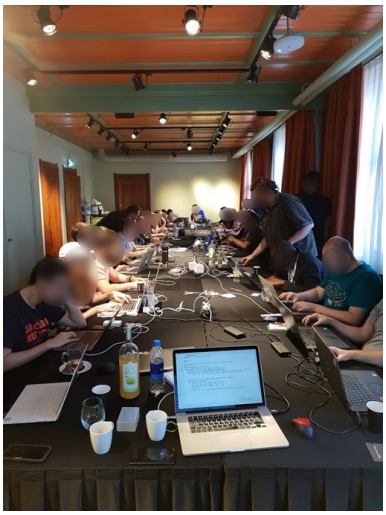
6

Kotlin Conf Booth / Workshop 2018

INSTIL

7

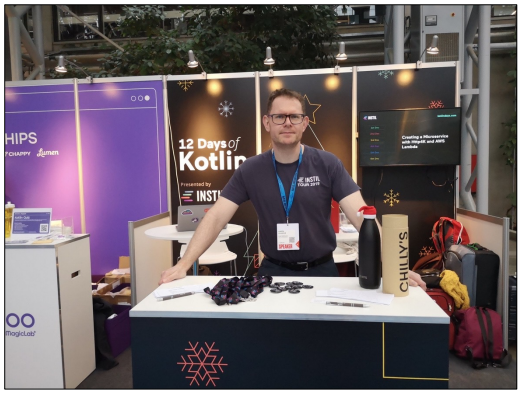
Kotlin Conf Booth / Workshop 2018



INSTIL

8

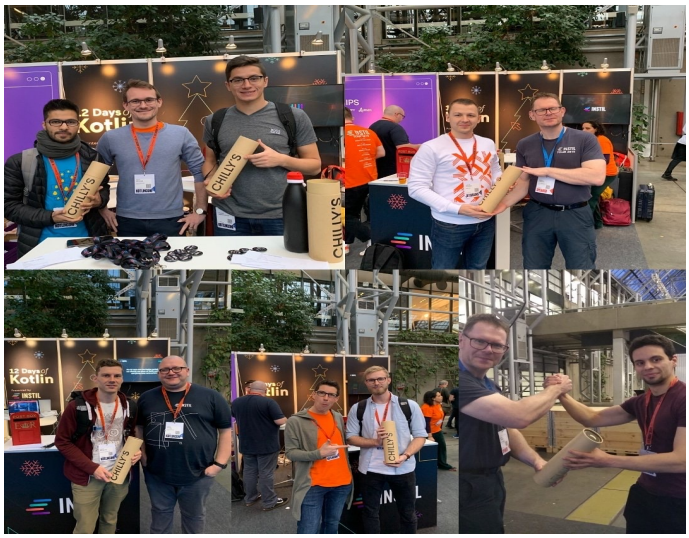
Kotlin Conf Booth / Workshop 2019



INSTIL

9

Kotlin Conf Booth / Workshop 2019



INSTIL

10

Kotlin Conf 2019

11

Why Listen To Us?

We don't get royalties from JetBrains or Google

We're placing a bet on Kotlin being the future

Over the years we've done a lot of Android projects

12

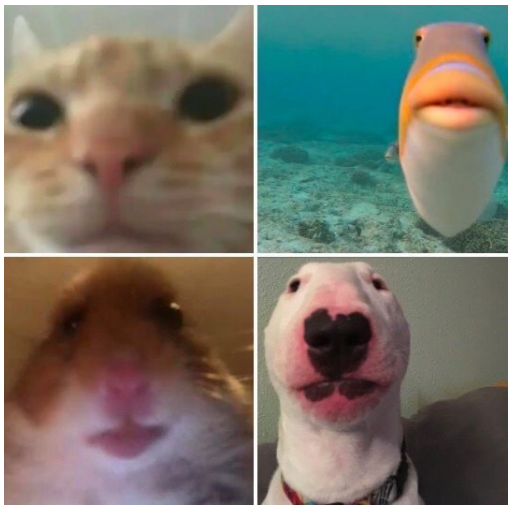
Please Ask Questions

INSTIL

13

Facilities and Logistics

- Prefer video on – it's better if we see each other
- Default to being on mute – but do contribute to discussion
- Zoom has a gallery mode so you can see everyone while also viewing the screen



INSTIL

14

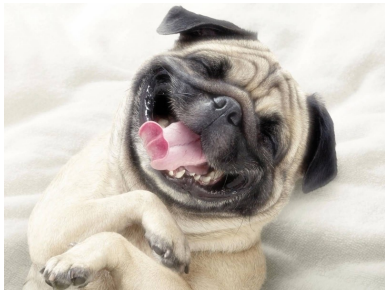
Distributed not remote

There is no back seat in online education
- Every student is in the front row

You're learning programming
- The best way to learn is to do

More time doing and talking than listening
- Shout at me if I talk for more than 15 minutes
- Without giving opportunities for contribution

Have fun!
- Play is the most efficient way to learn
- Just ask any puppy or kitten ☺



INSTIL

15

Quiz

- Experience

INSTIL

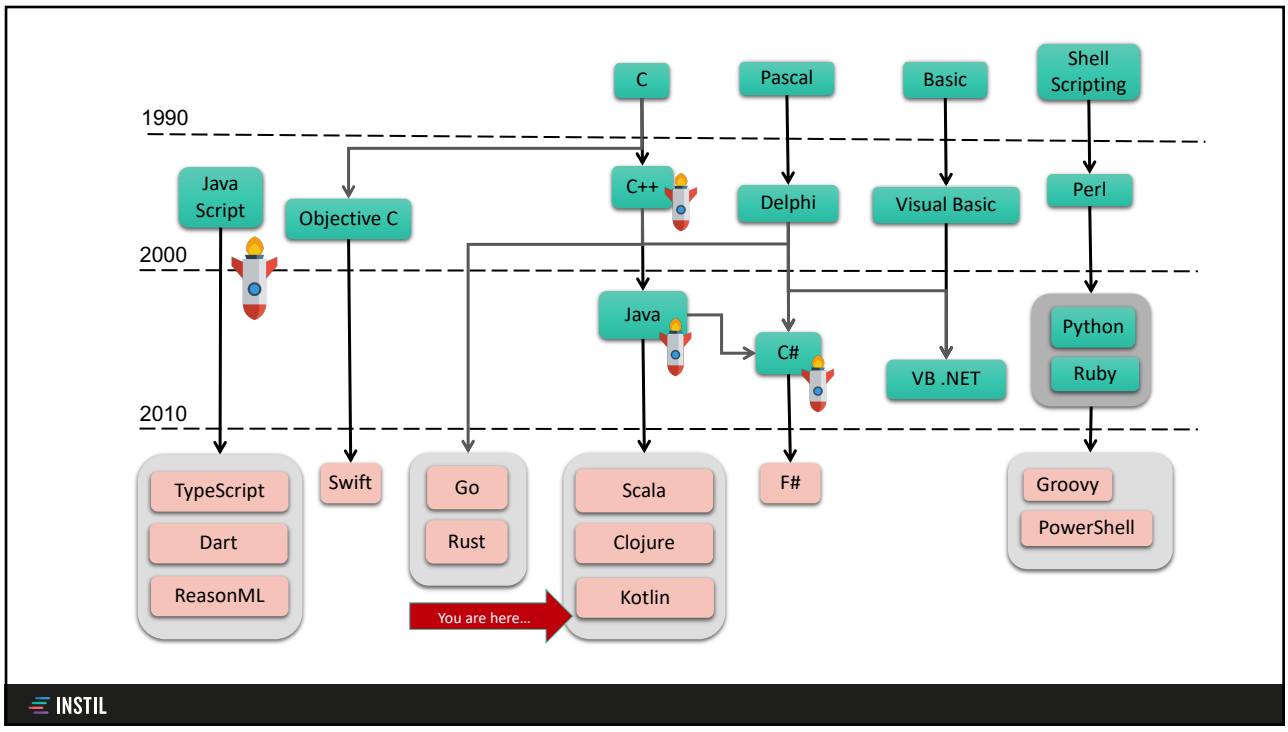
16

Introduction

- The evolution of Kotlin

INSTIL

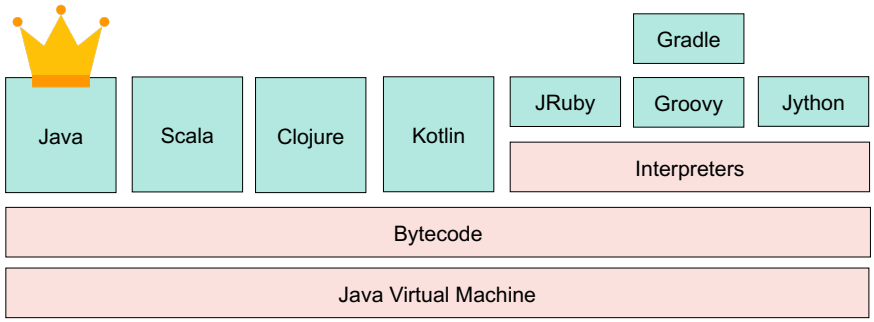
17



INSTIL

18

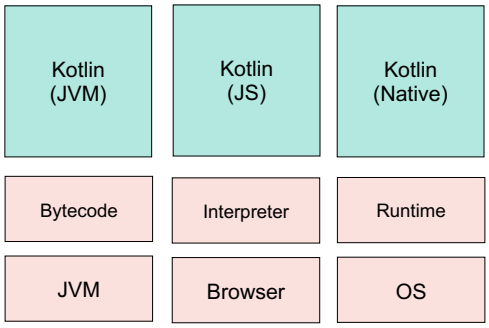
Languages on the JVM



INSTIL

19

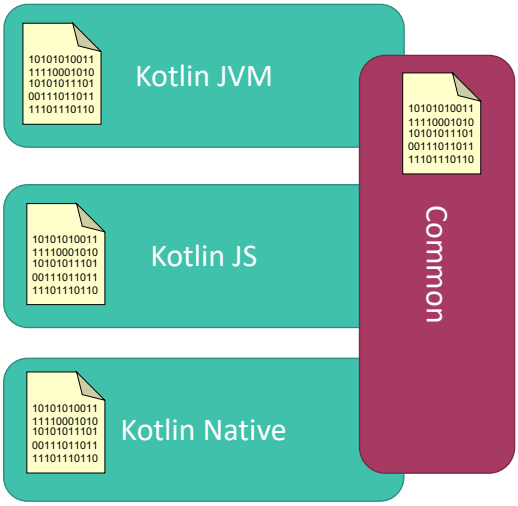
Kotlin On Other Platforms



INSTIL

20

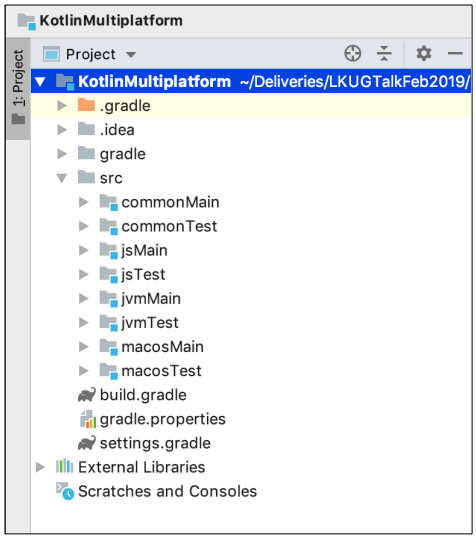
Multiplatform Libraries



INSTIL

21

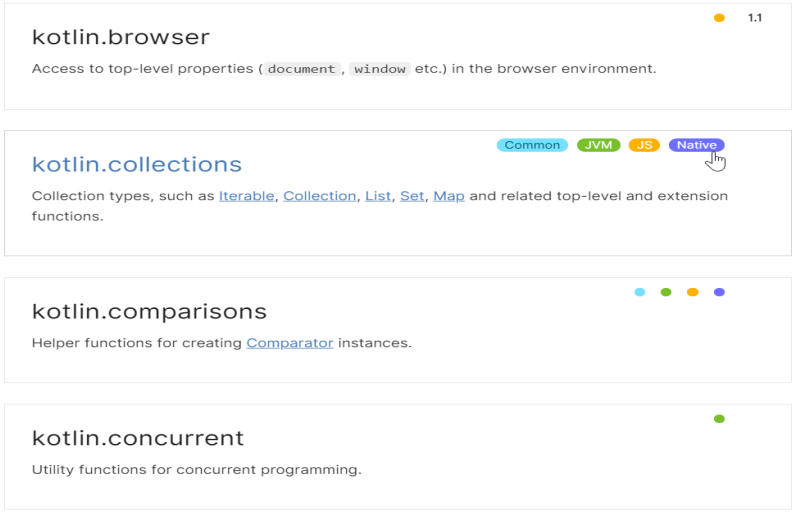
Multiplatform Libraries



INSTIL

22

Multiplatform Libraries - Compatibility



INSTIL

23

The Evolution of Java

```
Vector vec = new Vector();
vec.add("foo");
vec.add("bar");
vec.add("zed");
for(int i=0;i<vec.size();i++) {
    System.out.println(vec.get(i));
}

List list = new ArrayList();
list.add("foo");
list.add("bar");
list.add("zed");
for(int i=0;i<list.size();i++) {
    System.out.println(list.get(i));
}

List<String> strList = new ArrayList<>();
strList.add("foo");
strList.add("bar");
strList.add("zed");
for(String item : strList) {
    System.out.println(item);
}

strList.stream()
       .forEach(System.out::println);
```

Java 1

Java 2 - 4

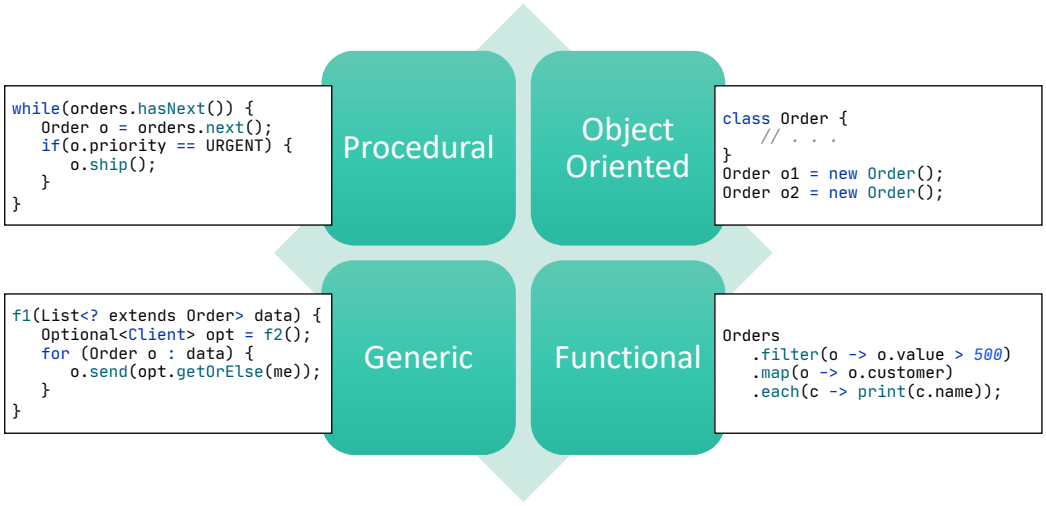
Java 5 - 7

Java 8

INSTIL

24

Styles of Programming



25

A Sample Java Program

```
public class Program {
    public static void main(String[] args) {
        @SuppressWarnings("resource")
        Scanner scanner = new Scanner(System.in);
        List<Integer> numbers = new ArrayList<>();

        System.out.println("Enter some numbers or three 'X' to finish");
        Pattern endOfInput = Pattern.compile("X{3}");
        while(scanner.hasNextLine()) {
            if (scanner.hasNext(endOfInput)) {
                break;
            } else if (scanner.hasNextInt()) {
                numbers.add(scanner.nextInt());
            } else {
                String mysteryText = scanner.nextLine();
                System.out.printf("Ignoring %s\n", mysteryText);
            }
        }
        int total = numbers.stream().reduce((a,b) -> a + b).orElse(0);
        System.out.printf("Total of numbers is: %s\n",total);
    }
}
```

hello.Program.java

26

```
Enter some numbers or three 'X' to finish
10
20
30
40
50
XXX
Total of numbers is: 150
```

```
Enter some numbers or three 'X' to finish
wibble
Ignoring wibble
12
13
14
XXX
Total of numbers is: 39
```

```
Enter some numbers or three 'X' to finish
XXX
Total of numbers is: 0
```


The Sample Re-Written in Kotlin

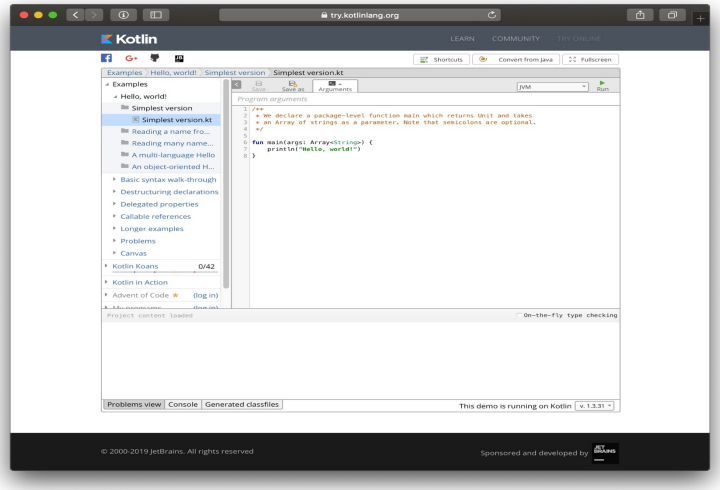
```
fun main(args: Array<String>) {
    val numbers = mutableListOf<Int>()
    val scanner = Scanner(System.`in`)
    val endOfInput = Regex("X{3}")
    println("Enter some numbers or three 'X' to finish")

    while (scanner.hasNextLine()) {
        if (scanner.hasNext(endOfInput.toPattern())) {
            break
        } else if (scanner.hasNextInt()) {
            numbers += scanner.nextInt()
        } else {
            val mysteryText = scanner.nextLine()
            println("Ignoring $mysteryText")
        }
    }
    //Would be better to use 'numbers.sum()'
    val total = numbers.fold(0, Int::plus)
    println("Total of numbers is: $total")
}
```

Points to note:

- ✓ No redundant class
- ✓ No semi-colons
- ✓ Type inference
- ✓ Both 'val' and 'var'
- ✓ Helper functions
- ✓ String interpolation
- ✓ Simplified collections
- ✓ Interop with Java types
- ✓ Simpler use of FP

hello.Program.kt

You Can Try This Online ☺



Running the Kotlin Compiler as a REPL

The Kotlin compiler can run as a REPL

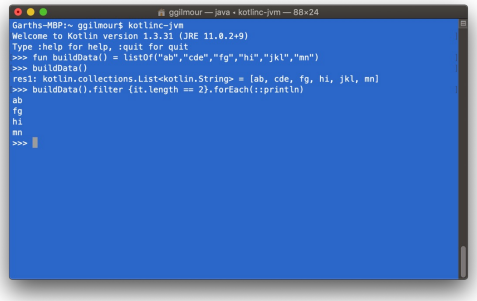
- Read / Evaluate / Print / Loop
- Like the console in a browser

This lets you explore the language

- In the same way you would learn a scripting language like Python

You can launch the REPL:

- From the Command Line / Shell
- From inside the IntelliJ IDE
  - Tools → Kotlin → Kotlin REPL



Kotlin vs. Scala Comparison

Feature	Java 9	Kotlin	Scala
Type Inference	No	Yes	Yes
Unified Type System	No	Yes	Yes
Protection from Null Results	No	Yes	Yes
String Interpolation	No	Yes	Yes
Extension Functions	No	Yes	Partially
Inline Functions	No	Yes	Yes
Primary Constructors	No	Yes	Yes
Case / Data Classes	No	Yes	Yes
Function Literals	Yes	Yes	Yes
Local Functions	No	Yes	Yes
Enclosure of local variables	Partially	Yes	Yes
Single Expression Functions	No	Yes	Yes
Tuples	No	Partially	Yes
Monads	No	Partially	Yes

31

A Kotlin Timeline

2010: Work begins on Kotlin within JetBrains

2011: First public announcement on Kotlin

2012: Open sourced under Apache 2 license

2016: Version one released

2017: First class support on Android

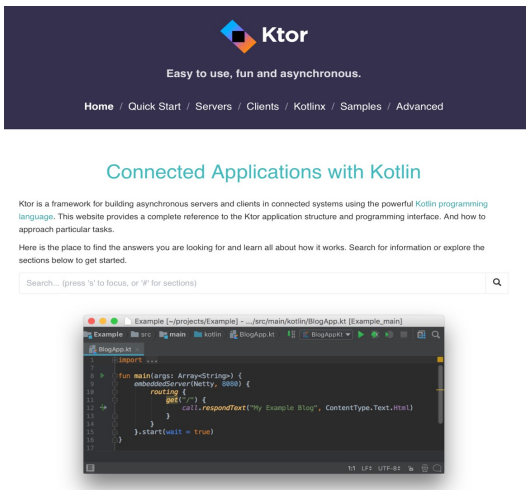
2018: Version 1.3 released

2020: Version 1.4 released

2021: Version 1.5 available

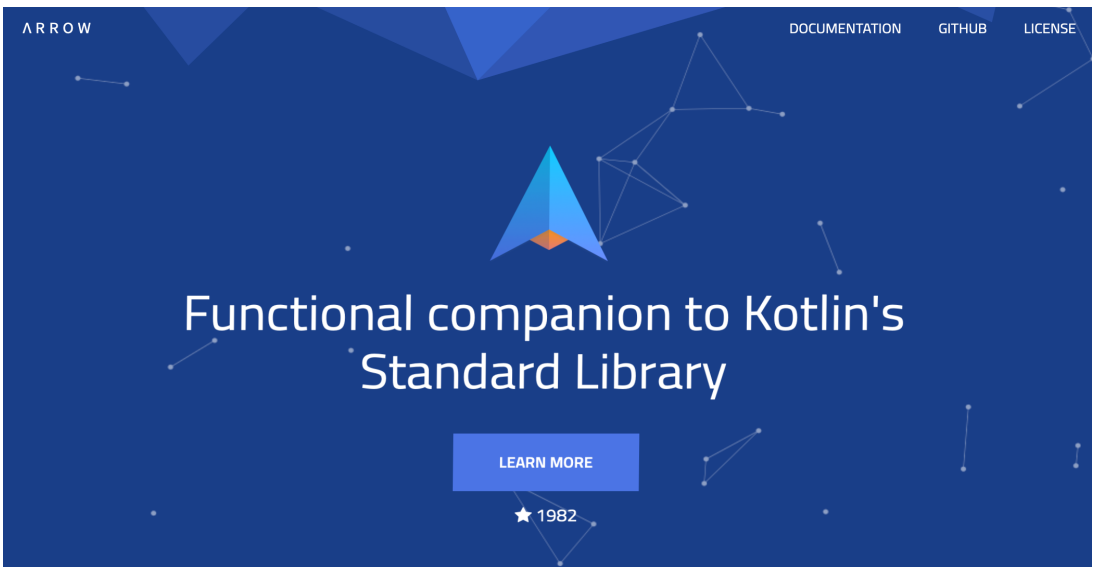
32

Making Waves at the Moment



33

Making Waves at the Moment



34

So Why Kotlin?

Excellent interoperability
- Call legacy Java code easily
- Make calls into Kotlin from Java

Really clear, yet concise syntax
- Reduces your codebase by 1/3+

Beloved by frameworks
- First class language on Android
- Mainstream language for Spring
- New language for the Gradle DSL
- Kotlin specific frameworks emerging
  - Arrow, Ktor, Koin, Http4K etc…

Null Safety
String Templates
Default parameters
Extensions
Free Functions
Coroutines
Single Expression Functions
Reified generics
Data classes and Properties
Type Inference
Smart Casts
Operator overloading

INSTIL

35

Kotlin Fundamentals

- Basic Syntax and Type System

INSTIL

36

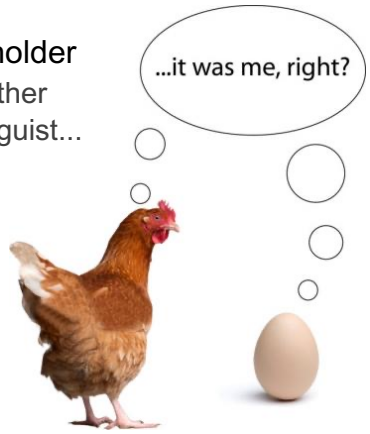
Bootstrapping the Kotlin Language

Learning a new language is always going to be an iterative process

- A few concepts will only need to be discussed once
- Most need to be revisited repeatedly in greater depth

Also note that simplicity resides in the eye of the beholder

- What's simple to one person is often complex to another
- You might be a mechanical engineer, I might be a linguist...



37

Starting to Code in Kotlin

Kotlin shares many coding conventions with Java

- Package names are all in lower-case
- Method and property names use camel-case
- Type names always use pascal-case
- Four space indentation is preferred

Semi-colons are no longer required

- Except to put multiple statements on the same line

38

Starting to Code in Kotlin

There is no 'new' operator

- As in Python objects are created solely via the type name
- `val p = Person("Jane")`

Braces are not required on empty classes

- Which can occur due to primary constructors (see later)
- `class Person(val name: String)`

Type Inference in Kotlin

The Kotlin compiler infers the types of a variable

- Based on the type of the initial definition
- `var x = 123`
- `var y = "abc"`

You still need an explicit type when:

- The variable is a function or constructor parameter
- You want to postpone the definition (e.g. conditional initialization)
- You want the type of the variable to be a subtype of the value
  - `var emp: Employee = Manager("Susan")`

Type Inference in Kotlin

Variables are introduced via the '`var`' and '`val`' keywords

- If you use '`val`' then the reference (not the value) is immutable
- This is encouraged, for both simplicity and thread safety

Neither '`var`' nor '`val`' is required when:

- You are declaring the counter in a '`for`' loop
- You are declaring a parameter in a function

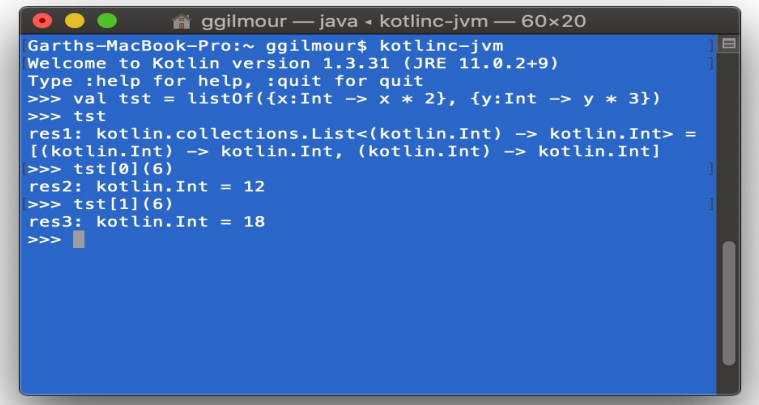
Type Inference In Kotlin

```
class Person(val name: String)

fun main(args: Array<String>) {
    val a = 123
    val b = "ABC"
    val c = "[A-Z]{3}".toRegex()
    val d = 1..10
    val e = Person("Jason")

    val f = arrayOf(a, b, c, d, e)
    f.forEach { println(it.javaClass.name) }
}
```

```
java.lang.Integer
java.lang.String
kotlin.text.Regex
kotlin.ranges.IntRange
basics.type.inference.Person
```

basics.type.inference.Program

Type Inference In Kotlin



43

The Kotlin Type System

Kotlin provides a fully object oriented type system

Values are stored as JVM primitive types where possible

- But all the (un)boxing is abstracted by the compiler for you

The 'Any' type is the universal base class

- This is compiled to '`java.lang.Object`'

This is very convenient but has a few drawbacks

- For example types are not automatically widened
- So you have to perform explicit conversions like this:
  - `val myLong: Long = myInt.toLong()`

44

Type Casting

The Kotlin compiler can automatically performs casts
- As long as it is certain the cast is safe

This typically occurs within conditionals
- E.g. `if (x is String) { x.toUpperCase() }`

The 'as' operator is used for unsafe casts
- E.g. `val x: Foo = y as Foo`
- It will throw an exception if the cast cannot succeed

There is also a nullable 'as' alternative
- `val x: Foo = y as? Foo`
- This will set 'x' to null if 'y' cannot be cast to 'Foo'

Type Conversions

```
class Person(val name: String)

fun foo(input: Any) {
    val result = when(input) {
        is String -> input.toUpperCase()
        is LocalTime -> input.hour
        is Person -> input.name
        else -> input.toString()
    }
    println(result)
}

fun main(args: Array<String>) {
    foo("wibble")
    foo(LocalTime.NOON)
    foo(Person("Dave"))
    foo(123)
}
```

```
WIBBLE
12
Dave
123
```

basics.casting.Program

Default Imports in Kotlin

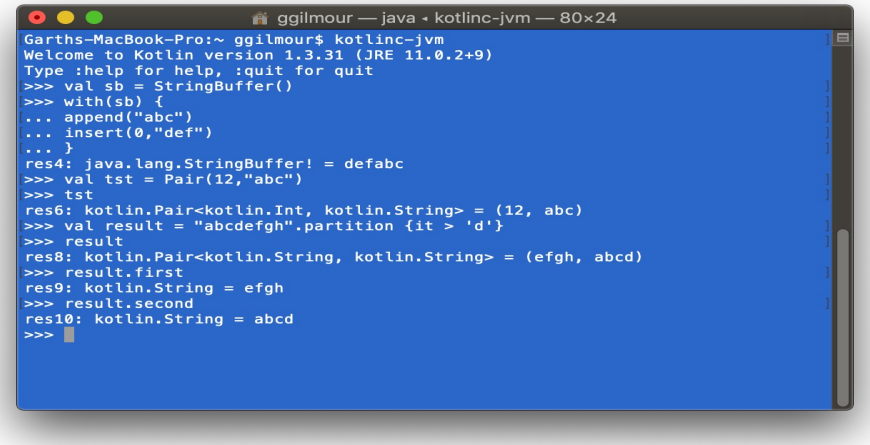
Some packages are automatically imported into Kotlin files
- A complete (at present) list is provided below

These contain types and functions helpful to most programs
- Such as the '`println`' method we will use extensively

Useful examples from '`kotlin`' include:
- The '`Pair`' and '`Triple`' types for tuples
  - E.g. `val p: Pair<Int, String>`
- The '`with`' function for simplifying OO

```
kotlin
kotlin.annotation
kotlin.collections
kotlin.comparisons
kotlin.io
kotlin.ranges
kotlin.sequences
kotlin.text
```

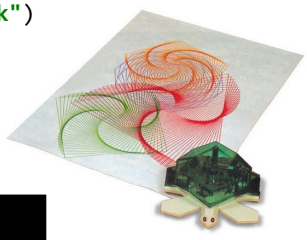

Default Imports in Kotlin



The With Statement

```
class Turtle {
    fun turnRight(degrees: Int=90)  = println("Turtle turning right")
    fun turnLeft(degrees: Int=90)   = println("Turtle turning left")
    fun forward(distance: Int=10)   = println("Turtle moving forward")
    fun back(distance: Int=10)      = println("Turtle moving back")
    fun penUp()                     = println("Turtle pen up")
    fun penDown()                   = println("Turtle pen down")
}
fun main(args: Array<String>) {
    val turtle = Turtle()

    with(turtle) {
        forward(100)
        turnRight()
        penDown()
        forward(50)
        penUp()
    }
}
```

```
Turtle moving forward
Turtle turning right
Turtle pen down
Turtle moving forward
Turtle pen up
```



basics.with

INSTIL

49

Managing Null Pointers in Kotlin

A design goal for Kotlin was to minimise null pointer errors

- They cant be eliminated without affecting interoperability

The Kotlin compiler checks for nulls automatically

It is not possible for a regular variable to be set to null

But every type can be made nullable by adding a question mark

- E.g. `String?`, Person**?** etc…

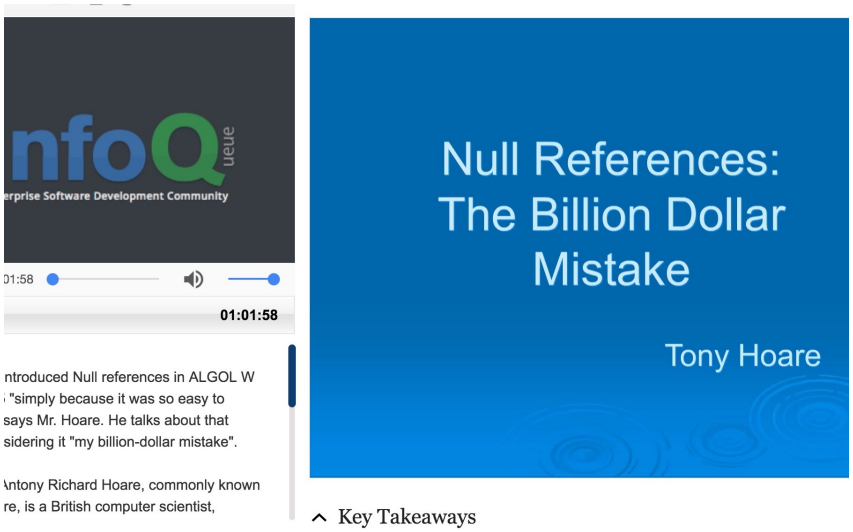
If the null can be tested for you can safely cast to non-nullable again

- Making the code safer

INSTIL

50

Null Pointer Errors



Operators for Nullable Types

The safe call operator, '`?.`', only makes a call if the reference is not null

- E.g. `foo?.bar()`
- This only calls 'bar' if 'foo' is not null

Calls can be chained as required

- E.g. `foo?.bar().zed()`

The Elvis operator, '`?:`', is used to specify a backup value

- E.g. `val x = foo ?: bar`
- This assigns x foo if foo is not null, otherwise bar
- Note that the right hand side is only evaluated if required

Operators for Nullable Types

The Not Null Assertion operator, '`!!.`', throws an exception if null

This can be useful when replacing a Java library with Kotlin

- While retaining the exception behaviour

So '`foo!!.bar()`' throws an exception if '`foo`' is null

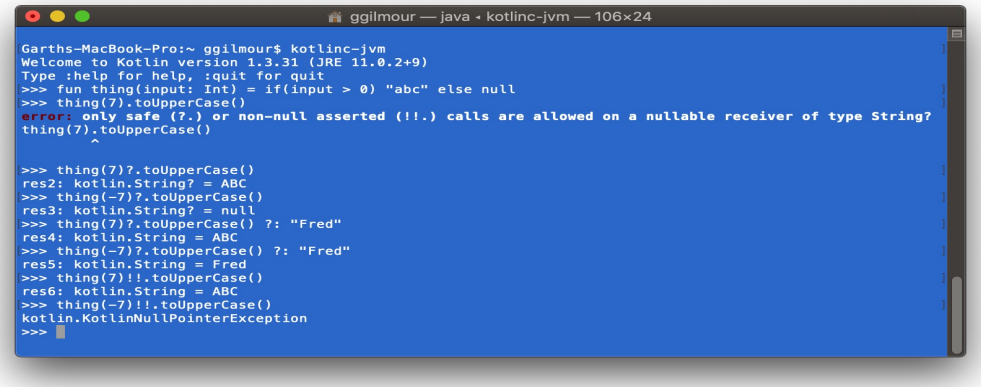
Managing Null Pointers in Kotlin

```
fun foo(input: Int) = if (input > 0) "abc" else "def" //return type is String
fun bar(input: Int) = if (input > 0) "abc" else null  //return type is String?

fun main(args: Array<String>) {
    val str1 = foo(12)                              //inferred type is String
    val str2 = bar(34)                              //inferred type is String?

    val str3: String = foo(12)                      //declared type is String
    val str4: String? = bar(34)                     //declared type must be String?

    println(bar(12)?.toUpperCase())                 //safe call operator (success)
    println(bar(-12)?.toUpperCase())                //safe call operator (failure)

    println(bar(12) ?: "wibble")                    //elvis operator (success)
    println(bar(-12) ?: "wibble")                   //elvis operator (failure)

    println(bar(12)!!.toUpperCase())                //NPE operator (success)
    try {
        println(bar(-12)!!.toUpperCase())           //NPE operator (failure)
    } catch(ex: Throwable) {
        println("Finally a NPE :-)")
    }
}
```

```
ABC
null
abc
wibble
ABC
Finally a NPE :-)
```

basics.safety.Program

Managing Null Pointers in Kotlin



55

Functions

- Declaring, Defining and Invoking

56

Function Declarations

Kotlin functions are both succinct and powerful
- But there are a number of subtleties to become familiar with

The function body can be declared in two ways

Single expression bodies use an '=' and no parenthesis
- The functions return value is that of the expression
- The return type can be inferred
- **Single line, expression functions are very useful**

Multi-line functions require braces and an explicit return
- Unlike Ruby or Scala the return type is never inferred

Function Declarations

The default return type for a function is 'Unit'
- '`Unit`' is a built in type which only ever has a single instance

This is a very common feature of functional languages
- In type theory a 'unit type' can only have one value

It is a great improvement on 'void'
- Which is an ugly kludge within the 'C' language family

For example you can use '`Unit`' as a type parameter
- Avoiding issues like '`Func<String, void> f = someFunc`' in C#

Function Parameters

You can specify a default parameter value
- Via the syntax ‘`foo = 123`’

Use ‘`vararg`’ to declare a variable arguments parameter
- The caller can pass in any number of values
- The compiler will group them into a collection

When invoking a function parameters can be set by name
- In which case the ordering no longer matters
- So the following could all be equivalent
  - `foo(12,34)`
  - `foo(a=12, b=34)`
  - `foo(b=34, a=12)`

59

Function Declarations

```
fun demo1(input: Int) {
    println("Demo1 called with $input")
}

fun demo2(input: Int): Unit { // same as above
    println("Demo2 called with $input")
}

fun demo3(input: Int): String = "Demo3 called with $input"

fun demo4(input: Int): String {
    return "Demo4 called with $input"
}
```

basics.functions.Program.kt

60

Function Declarations

```
fun demo5(input: Int = 456): String {
    return "Demo3 called with $input"
}

fun demo6(vararg input: Int) = input.reduce { a,b -> a + b}

fun demo7(port: Int, timeout: Int, retries: Int): String {
    return """
           Trying to connect on port $port
           with $retries retries and a timeout of $timeout
           """.trimIndent();
}
```

basics.functions.Program.kt

61

Function Declarations

```
fun main(args: Array<String>) {
    demo1(123)
    demo2(123)
    println(demo3(123))
    println(demo4(123))
    println(demo5(123))
    println(demo5())
    println("Adding numbers gives: ${demo6(10,20,30,40,50)}")
    println(demo7(timeout = 60, port = 8080, retries = 10))
}
```

```
Demo1 called with 123
Demo2 called with 123
Demo3 called with 123
Demo3 called with 123
Demo3 called with 123
Demo3 called with 456
Adding numbers gives: 150
Trying to connect on port 8080
  with 10 retries and a timeout of 60
```

basics.functions.Program.kt

62

Special Kinds Of Function

A Kotlin function can also be declared as infix

- This allows it to be used as if it were a keyword
- E.g. '`12 downTo 1`' is the same as '`12.downTo(1)`'

An infix function must follow some simple rules:

- It must take a single parameter
- It must be a member of a type i.e. a method
  - Or an extension to an existing type (see later)
- It must be declared using the '`infix`' keyword

Special Function Declarations

```
class Number(val value: Int) {
    infix fun plus(input: Int) = Number(value + input)

    infix fun minus(input: Int) = Number(value - input)

    fun doTimes(action: (Int) -> Unit) {
        for(x in 1..value) {
            action(x)
        }
    }
}
```

basics.functions.infix

Special Function Declarations

```
fun main(args: Array<String>) {
    val num1 = Number(5)
    val num2 = num1 plus 7
    val num3 = num2 minus 3

    num1.doTimes { print("$it ") }
    println()

    num2.doTimes { print("$it ") }
    println()

    num3.doTimes { print("$it ") }
}
```

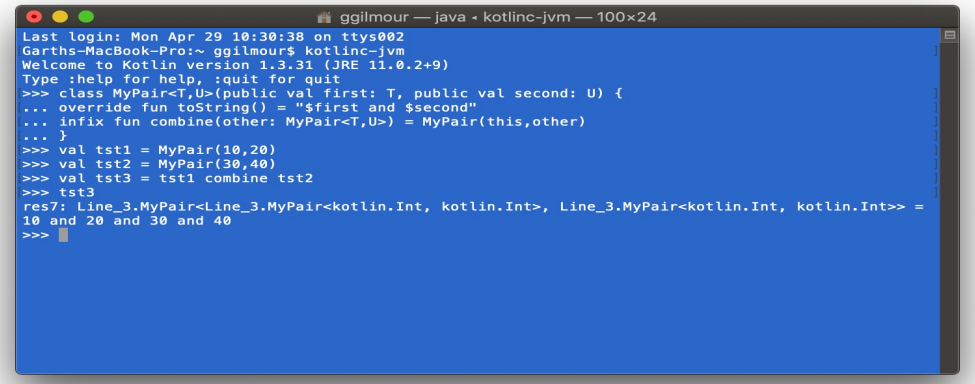
```
1 2 3 4 5
1 2 3 4 5 6 7 8 9 10 11 12
1 2 3 4 5 6 7 8 9
```

basics.functions.infix

INSTIL

65

Special Function Declarations



INSTIL

66

Special Kinds of Function

It is possible to pass references to functions as parameters

- This will be covered in depth during the chapter on FP

We saw an example in the introductory program

```
val total1 = numbers.fold(0, Int::plus)
println("Total of numbers is: $total1")
```

Another simple (but useful) example is '`repeat`'

- This executes a code block the specified number of times

The Repeat Utility Function

```
fun sample(x: Int) = println("Hello $x")

fun main(args: Array<String>) {
    repeat(4, ::sample)
    repeat(5) { x -> println("Bye $x") }
}
```

```
Hello 0
Hello 1
Hello 2
Hello 3
Bye 0
Bye 1
Bye 2
Bye 3
Bye 4
```

basics.repeat.Program

Ranges

Kotlin provides a built in syntax for ranges

- This is extremely useful as it avoids the need to have both a *for-loop-with-counter* and a *for-loop-with-element*

The syntax 'a`..`b' creates an inclusive Range

- You can specify the step via the `step` keyword

There are many other options for creating Ranges

- 'a `until` b' creates an half open range which omits 'b'
- 'a `downTo` b' creates a descending inclusive range
  - Once again you can specify the step via `step`

Working With Ranges

```
fun main(args: Array<String>) {
    val range1 = 1..10
    val range2 = 1..10 step 2
    val range3 = 10 downTo 1
    val range4 = 10 downTo 1 step 2
    val range5 = 1 until 10
    val range6 = 1 until 10 step 2

    val ranges = listOf(range1,range2,range3,
                        range4,range5,range6)
    println("The values of the ranges:")
    ranges.forEach(::useRange)

    val reversedRanges = ranges.map { it.reversed() }
    println("And now in reverse:")
    reversedRanges.forEach(::useRange)
}
```

```
fun useRange(input: IntProgression) {
    print('\t')
    for(x in input) {
        print("$x ")
    }
    println()
}
```

```
The values of the ranges:
    1 2 3 4 5 6 7 8 9 10
    1 3 5 7 9
    10 9 8 7 6 5 4 3 2 1
    10 8 6 4 2
    1 2 3 4 5 6 7 8 9
    1 3 5 7 9
And now in reverse:
    10 9 8 7 6 5 4 3 2 1
    9 7 5 3 1
    1 2 3 4 5 6 7 8 9 10
    2 4 6 8 10
    9 8 7 6 5 4 3 2 1
    9 7 5 3 1
```

basics.ranges.Program.kt

Control Flow

- Selection, Iteration and Exceptions

INSTIL

71

Selection

Kotlin provides '`if`' and '`when`' keywords
- Which are the equivalents of 'if' and 'switch' in Java

They both count as **expressions** so evaluate to (or return) a value
- So you can set a variable directly to their result

The '`if`' expression can be written on a single line
- Removing the need for a 'C' style ternary conditional

The '`when`' keyword is extremely versatile
- You can specify a single value, a selection, a range, a collection etc…
- Multiple expressions on the RHS must be enclosed in braces

INSTIL

72

Making Choices in Kotlin

```kotlin
val input = Scanner(System.`in`)

println("Enter a number")
val num = Integer.parseInt(input.nextLine())

val message1 = if (num % 2 == 0) "even" else "odd"
println("The number is $message1")

var message2: String
if (num % 2 == 0) {
    message2 = "even"
} else {
    message2 = "odd"
}
println("The number is $message2")
```

basics.selection.Program

INSTIL

73

Making Choices in Kotlin

```kotlin
val tmpList = listOf(60,70,80)

val message3 = when (num) {
    10 ->          "The number is 10"
    20, 30, 40 -> "The number is 20, 30 or 40"
    in 41..50 ->  "The number is between 41 and 50"
    in tmpList -> "The number is 60,70 or 80"
    else ->       "This number perplexes me..."
}
println(message3)

var message4: String
when (num) {
    10 ->          message4 = "The number is 10"
    20,30,40 ->    message4 = "The number is 20, 30 or 40"
    in 41..50 ->   message4 = "The number is between 41 and 50"
    in tmpList -> message4 = "The number is 60,70 or 80"
    else ->       message4 = "This number perplexes me..."
}
println(message4)
```

IntelliJ can automatically refactor this to the above style

basics.selection.Program

INSTIL

74

Iteration

Kotlin supports the standard '`for`', '`while`' and '`do`' loops

As noted previously the '`for`' loop has only a single form

- Thanks to the built-in support for ranges
- Also no keyword is needed when declaring the variable

An alternative to *external* iteration is the functional style or *internal* iteration

- Where we pass code blocks that represent tasks
- E.g. '`(1..10).forEach { println(it) }`'
- We will examine this is great detail later on
  - Each style has its strengths and weaknesses

Iteration in Kotlin

```
fun pyramidViaFor(height: Int) {
    for (rowNum in 1..height) {
        val spaces = height - rowNum
        val hashes = (rowNum * 2) - 1

        for (x in 1..spaces) print(' ')
        for (x in 1..hashes) print('#')
        println()
    }
}
```

```
Enter the height of the pyramid
6
     #
    ###
   #####
  #######
 #########
###########
```

basics.loops.Program.kt

Iteration in Kotlin

```
fun pyramidViaWhile(height: Int) {
    fun printSpaces(spaces: Int) {
        var x = 0
        while (x < spaces) {
            print(' ')
            x++
        }
    }
    fun printHashes(hashes: Int) {
        var x = 0
        while (x < hashes) {
            print('#')
            x++
        }
    }
    var rowNum = 1
    while (rowNum <= height) {
        printSpaces(height - rowNum)
        printHashes((rowNum * 2) - 1)
        rowNum++;
        println()
    }
}
```

```
Enter the height of the pyramid
6
     #
    ###
   #####
  #######
 #########
###########
```

basics.loops.Program.kt

INSTIL

77

Iteration in Kotlin

```
fun pyramidViaDoWhile(height: Int) {
    fun printSpaces(spaces: Int) {
        var x = 0
        do {
            print(' ')
            x++
        } while (x <= spaces)
    }
    fun printHashes(hashes: Int) {
        var x = 0
        do {
            print('#')
            x++
        } while (x < hashes)
    }
    var rowNum = 1
    do {
        printSpaces(height - rowNum)
        printHashes((rowNum * 2) - 1)
        rowNum++;
        println()
    } while (rowNum <= height)
}
```

```
Enter the height of the pyramid
6
      #
     ###
    #####
   #######
  #########
 ###########
```

Because we chose the wrong tool for the job our pyramid needs to be padded one space out from the lhs

basics.loops.Program.kt

INSTIL

78

Iteration in Kotlin

```
fun pyramidViaFP(height: Int) {
    (1..height).forEach { rowNum ->
        val spaces = height - rowNum
        val hashes = (rowNum * 2) - 1

        (1..spaces).forEach { print(' ') }
        (1..hashes).forEach { print('#') }
        println()
    }
}
```

```
Enter the height of the pyramid
6
     #
    ###
   #####
  #######
 #########
###########
```

It is also possible to iterate using the functional programming style. We show it here for completeness and will return to it later…

`basics.loops.Program.kt`

Exception Handling In Kotlin

Exception handling in Kotlin is similar to Java

- Both the syntax and the runtime remain very much the same
- However there are a couple of important differences to note

Kotlin does not support compiler checked exceptions

- A good idea as these have fallen out of favour in general
- The '@Throws' annotation can be used to ask the compiler to add a 'throws' clause to a method for backwards compatibility

The '`try … catch`' block is an **expression** in Kotlin

- You can assign a variable to the result of the block

A method may be declared with the '`Nothing`' return type

- This tells the compiler it will always terminate abnormally

Exception Handling In Kotlin

```
fun possibleDeath(input: Int) {
    if (input < 0) throw IllegalStateException("Boom!")
}

fun foo(input: Int) {
    bar(input)
}

fun bar(input: Int) {
    zed(input)
}

fun zed(input: Int) {
    if (input < 0) {
        throw IllegalStateException("Having a bad day...")
    }
}
```

basics.exceptions.Program

INSTIL

81

Exception Handling in Kotlin

```
fun commonSyntax() {
    try {
        foo(-1)
        println("Success!")
    } catch (ex: IllegalStateException) {
        println(ex.message)
    }

    try {
        foo(1)
        println("Success!")
    } catch (ex: IllegalStateException) {
        println(ex.message)
    }
}
```

basics.exceptions.Program

INSTIL

82

Exception Handling in Kotlin

```
fun tryAsExpression() {
    val message1 = try {
        foo(-1)
        "Success!"
    } catch(ex: Exception) {
        ex.message
    }

    println(message1)
    val message2 = try {
        foo(1)
        "Success!"
    } catch(ex: Exception) {
        ex.message
    }
    println(message2)
}
```

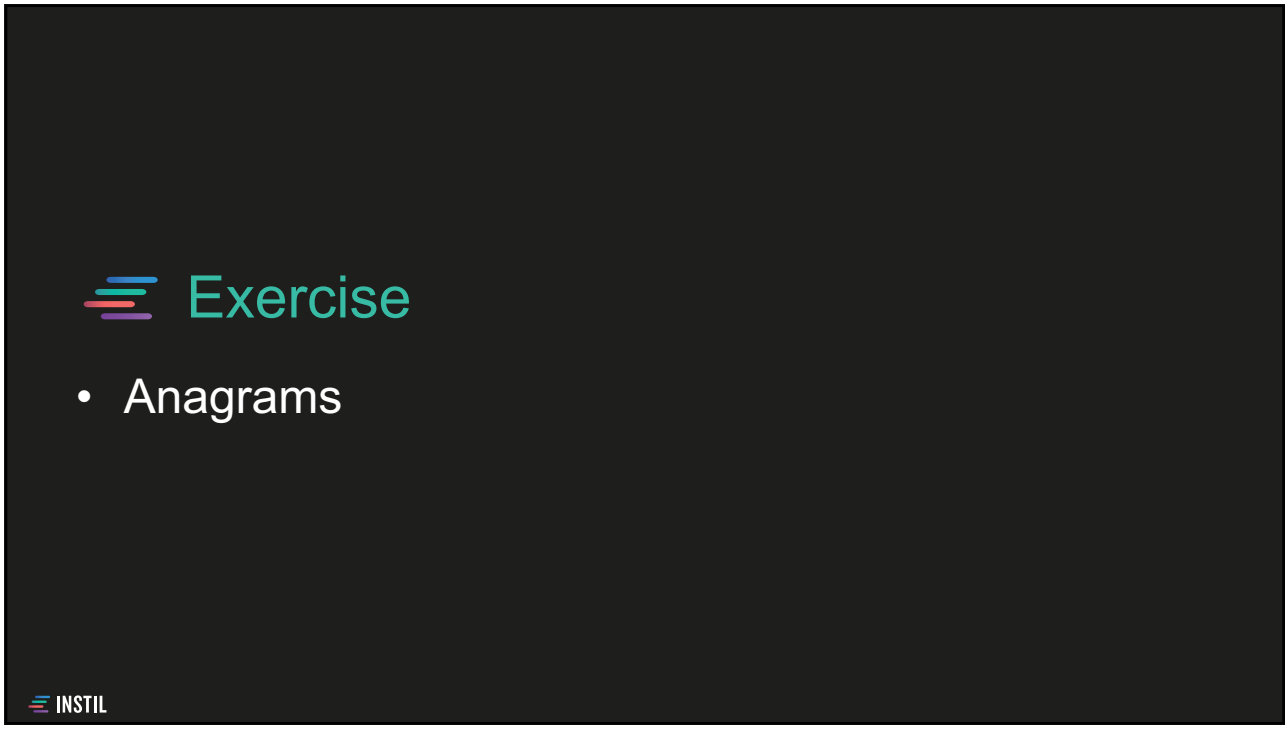
basics.exceptions.Program

Exception Handling in Kotlin

```
fun certainDeath(): Nothing = throw IllegalStateException("Boom!")

fun theNothingType() {
    val result1 = possibleDeath(1)
    println(result1)
    // result2 is unreachable code
    val result2 = certainDeath()
    // will not compile
    // println(result2)
}

fun main(args: Array<String>) {
    commonSyntax()
    tryAsExpression()
    try {
        theNothingType()
    } catch(ex: Exception) {
        ex.printStackTrace(System.out)
    }
}
```

```
Having a bad day...
Success!
Having a bad day...
Success!
kotlin.Unit
java.lang.IllegalStateException: Boom
  at ...certainDeath(Program.kt:52)
  at ...theNothingType(Program.kt:48)
  at ...main(Program.kt:7)
```

basics.exceptions.Program


Exercise
• Anagrams
INSTIL

85


Basic Object Orientation
• Classes, Constructors and Members
INSTIL
© Instil Software 2022

86

Revision: What is an Object?

In all OO languages an object is a collection of slots

- A slot that contains data is referred to as a field
- A slot that contains code is referred to as a method
- In C++ / Java / C# the correct term is 'member'

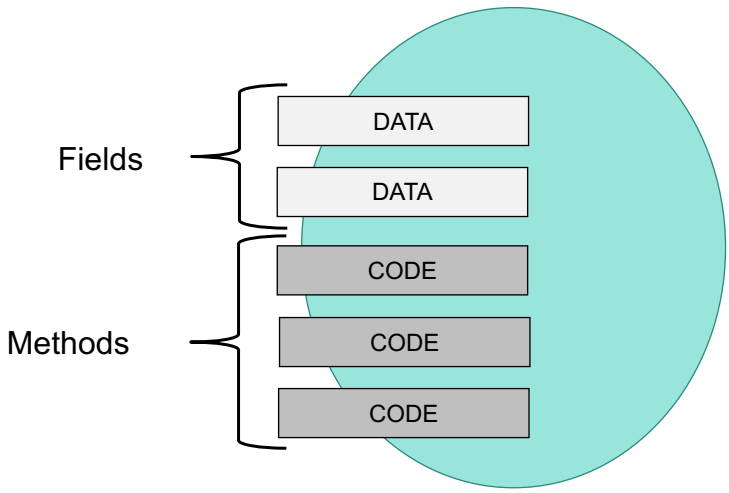
Objects contain state and offer behaviour

- The state is the combination of the values in the fields
- The behaviour is the services provided by the methods

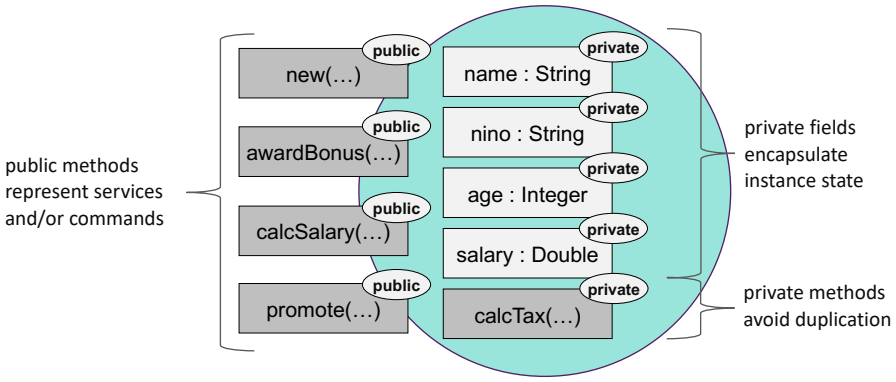
Clients should be able to remain ignorant of the state

- E.g. you can call 'calcSalary' on an 'Employee' object without needing to be aware how this is stored and/or calculated

Revision: What is an Object?



Revision: What is an Object?



89

Classes and Properties in Kotlin

When coding in Kotlin we think in terms of properties

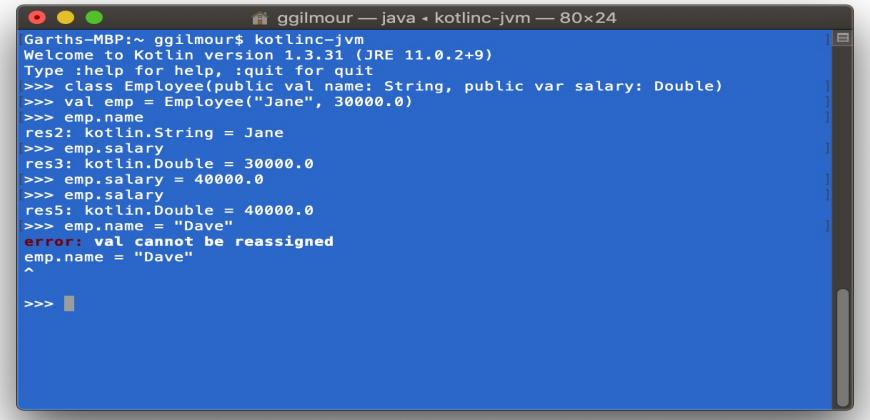
- In Kotlin, it isn't possible to manually add fields to a class
- Instead you add properties, which are public by default

When you add a property to a class Kotlin will:

- Declare a 'backing field' of the appropriate type
- Declare a standard Java getter method
  - Unless the property has been declared private
- Declare a standard Java setter method
  - Assuming the property has been declared with 'var'

90

Classes and Properties in Kotlin



91

Classes and Properties in Kotlin

Accessor methods are automatically invoked for you

```
val wage = emp.salary      ⟶   val wage = emp.getSalary()

emp.salary = n             ⟶   emp.setSalary(n)
```


92

Classes and Properties in Kotlin

```
class Person {
    var name: String = "Dave“

    fun sayHello() = println("Hello $name")
}

fun main(args: Array<String>) {
    val p = Person()
    println(p.name)
    p.sayHello()
    p.name = "Jane"
    println(p.name)
    p.sayHello()
}
```

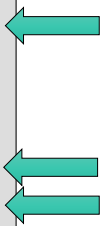
```
javap -p oo.properties.Person

Compiled from "Person.kt"
public final class oo.properties.Person {
  private java.lang.String name;
  public final java.lang.String getName();
  public final void setName(java.lang.String);
  public final void sayHello();
  public oo.properties.regular.Person();
}
```

```
Dave
Hello Dave
Jane
Hello Jane
```

`oo.properties`

Classes and Properties in Kotlin

```
$ javap -p -c oo.properties.regular.PersonKt
Compiled from "Person.kt"
public final class oo.properties.regular.PersonKt {
  public static final void main(java.lang.String[]);
    Code:
       0: aload_0
       1: ldc           #9          // String args
       3: invokestatic  #15         // Method …/Intrinsics.checkParameterIsNotNull:(…)V
       6: new           #17         // class oo/properties/regular/Person
       9: dup
      10: invokespecial #21         // Method oo/properties/regular/Person."<init>":()V
      13: astore_1
      14: aload_1
      15: invokevirtual #25         // Method oo/properties/regular/Person.getName:()Ljava/lang/String;
      18: astore_2
      19: getstatic     #31         // Field java/lang/System.out:Ljava/io/PrintStream;
      22: aload_2
      23: invokevirtual #37         // Method java/io/PrintStream.println:(Ljava/lang/Object;)V
      26: aload_1
      27: invokevirtual #40         // Method oo/properties/regular/Person.sayHello:()V
      30: aload_1
      31: ldc           #42         // String Jane
      33: invokevirtual #46         // Method oo/properties/regular/Person.setName:(Ljava/lang/String;)V
      36: aload_1
      37: invokevirtual #25         // Method oo/properties/regular/Person.getName:()Ljava/lang/String;
      40: astore_2
      41: getstatic     #31         // Field java/lang/System.out:Ljava/io/PrintStream;
      44: aload_2
      45: invokevirtual #37         // Method java/io/PrintStream.println:(Ljava/lang/Object;)V
      48: aload_1
      49: invokevirtual #40         // Method oo/properties/regular/Person.sayHello:()V
      52: return
}
```



Customising Properties

The default accessor methods can be overridden

- By providing ‘get’ and/or ‘set’ blocks after the declaration
- The backing field can be accessed via the keyword ‘field’
  - Note this is only allowed within the ‘get/set’ blocks

This is useful in two scenarios:

- When you want a computed property derived from other data
- When you want to add to intercept access to add more behaviour
  - E.g. validation, logging etc

95

Customising Properties

```
fun main(args: Array<String>) {
    val p = Person()
    p.sayHello()
    p.name = "Jane"
    p.sayHello()
}

class Person {
    var name: String = "Dave"
        get() = "[$field]"
        set(value) { field = "$value Jones" }

    fun sayHello() = println("Hello $name")
}
```

```
Hello [Dave]
Hello [Jane Jones]
```

oo.properties.customized

96

Kotlin and Primary Constructors

Kotlin provides a feature called ‘primary constructors’

The class, constructor and field declarations are merged
- This removes a lot of boilerplate code

The way the parameters are declared is crucial
- If declared with var / val then fields and properties are added
  - When ‘val’ is used only a getter method is created
- If declared without ‘var’ or ‘val’ they are normal inputs
  - No extra members will be added to the class

The properties will be public by default
- You can alter this by adding an access modifier

Understanding Primary Constructors

```
package oo.basics

class Person(name: String, age: Int)
```

Here the parameters lack ‘val’ or ‘var’ so no fields or properties are created.

```
$ javap -p  oo.basics.Person

Compiled from "Person.kt"
public final class oo.basics.Person {
  public oo.basics.Person(java.lang.String, int);
}
```

oo.basics

Understanding Primary Constructors

```
package oo.basics

class Person(val name: String, val age: Int)
```

Adding 'val' to the parameters means that a backing field will be created, initialised and accessible via a 'getter' method

```
$ javap -p  oo.basics.Person

Compiled from "Person.kt"
public final class oo.basics.Person {
  private final java.lang.String name;
  private final int age;
  public final java.lang.String getName();
  public final int getAge();
  public oo.basics.Person(java.lang.String, int);
}
```

oo.basics

INSTIL

99

Understanding Primary Constructors

```
package oo.basics

class Person(var name: String, var age: Int)
```

Using 'var' instead of 'val' means that the compiler also generates 'setter' methods

```
Compiled from "Person.kt"

public final class oo.basics.Person {
  private java.lang.String name;
  private int age;
  public final java.lang.String getName();
  public final void setName(java.lang.String);
  public final int getAge();
  public final void setAge(int);
  public oo.basics.Person(java.lang.String, int);
}
```

oo.basics

INSTIL

100

Understanding Primary Constructors

```
package oo.basics

class Person(val name: String, var age: Int)
```

There is no issue with using both 'val' and 'var' on different parameters. Note the default access is public.

```
$ javap -p  oo.basics.Person

Compiled from "Person.kt"
public final class oo.basics.Person {
  private final java.lang.String name;
  private int age;
  public final java.lang.String getName();
  public final int getAge();
  public final void setAge(int);
  public oo.basics.Person(java.lang.String, int);
}
```

oo.basics

INSTIL

101

Understanding Primary Constructors

```
package oo.basics

class Person(public val name: String,
             protected var age: Int)
```

Adding 'public' to the 'name' field will have no effect as that is the default. Adding 'protected' to 'age' will change the access level of the methods but we would need to declare the class 'open' for this to be useful (see discussion of inheritance later)

```
$ javap -p  oo.basics.Person

Compiled from "Person.kt"
public final class oo.basics.Person {
  private final java.lang.String name;
  private int age;
  public final java.lang.String getName();
  protected final int getAge();
  protected final void setAge(int);
  public oo.basics.Person(java.lang.String, int);
}
```

oo.basics

INSTIL

102

Understanding Primary Constructors

```
package oo.basics

class Person(public val name: String,
             private var age: Int)
```

Declaring a parameter as private causes the backing field to be created but not the accessor methods. IntelliJ will suggest making parameters private where possible.

```
javap -p  oo.basics.Person

Compiled from "Person.kt"
public final class oo.basics.Person {
  private final java.lang.String name;
  private int age;
  public final java.lang.String getName();
  public oo.basics.Person(java.lang.String, int);
}
```

oo.basics

INSTIL

103

Understanding Primary Constructors

```
package oo.basics

class Person(private val name: String,
             private var age: Int)
```

There is no issue with all the parameters being private

```
$ javap -p  oo.basics.Person

Compiled from "Person.kt"
public final class oo.basics.Person {
  private final java.lang.String name;
  private int age;
  public oo.basics.Person(java.lang.String, int);
}
```

oo.basics

INSTIL

104

Extra Features of Primary Constructors

You can still declare your own properties within the class

- If the property is declared private then only a field is created
- Otherwise get / set methods are created as appropriate

Arbitrary code can be supplied for the primary constructor

- Any code within a 'init' block is added to the constructor
- NB the Scala syntax is to put the code within the class body
  - Which is a frequent source of confusion for beginners

105

Extra Features of Primary Constructors

A class must always have a constructor

- Since one will be created for you if absent

However it can be a private no-args constructor

- The syntax is 'class Foo private constructor () { ... }'

This is used in Design Patterns such as 'Singleton'

106

Understanding Primary Constructors

```
package oo.basics

class Person(val name: String, val age: Int) {
    private val address = "10 Arcatia Road“

    init {
        println("Person object created")
    }
}
```

```
javap -p -c  oo.basics.Person

Compiled from "Person.kt"
public final class oo.basics.Person {
  private final java.lang.String address;
  private final java.lang.String name;
  private final int age;
  public final java.lang.String getName();
  public final int getAge();
  public oo.basics.Person(java.lang.String, int);
      --------- Earlier code omitted ---------
      21: ldc           #8    // String 10 Arcatia Road
      23: putfield      #36   // Field address:Ljava/lang/String;
      26: ldc           #38   // String Person object created
      28: astore_3
      29: getstatic     #44   // Field java/lang/System.out:Ljava/io/PrintStream;
      32: aload_3
      33: invokevirtual #50   // Method java/io/PrintStream.println:(Ljava/lang/Object;)V
      36: return
}
```


Adding Secondary Constructors

A class should have a single ‘happy path’ constructor

- Even in languages that don’t support primary constructors it is good design
- Alternative constructors should invoke that standard constructor

Kotlin tries to enforce this best practise

A class can have any number of secondary constructors

- They must all call (directly or indirectly) into the primary constructor

A secondary constructor is any method called ‘constructor’

- The call to the primary goes after the list of parameters, preceded by a colon
- It is legal for a class to only have secondary constructors

Adding Secondary Constructors

```
package oo.basics

class Person(val name: String, val age: Int) {
    private var address = "10 Arcatia Road“

    init {
        println("Person object created")
    }

    constructor(name: String, age: Int, address: String) : this(name, age) {
        println("Secondary constructor called")
        this.address = address;
    }
}
```

```
$ javap -p  oo.basics.Person

Compiled from "Person.kt"
public final class oo.basics.Person {
  private java.lang.String address;
  private final java.lang.String name;
  private final int age;
  public final java.lang.String getName();
  public final int getAge();
  public oo.basics.Person(java.lang.String, int);
  public oo.basics.Person(java.lang.String, int, java.lang.String);
}
```


Declaring Classes as Open

All the classes we have seen so far are implicitly final

- **By default it is not possible to inherit from a Kotlin class**

This fits in with best practices in modern OO design

- Developers tend to see inheritance as a ‘magic hammer’

Composition is normally preferable to inheritance

- Because it offers better encapsulation
- Inheritance should be reserved for true ‘IS A’ situations

To make a class non-final use the ‘open’ keyword

- Note that it is legal but pointless to declare protected properties within a class which is not open

Understanding Primary Constructors

```
package oo.basics

open class Person(val name: String, val age: Int)
```

```
$ javap -p  oo.basics.Person

Compiled from "Person.kt"
public class oo.basics.Person {
  private final java.lang.String name;
  private final int age;
  public final java.lang.String getName();
  public final int getAge();
  public oo.basics.Person(java.lang.String, int);
}
```


Declaring Data Classes

Classes are typically split according to the MVC pattern

- View classes interact with the outside world
- Controller classes implement flows of events
- Model classes represent entities from the domain
  - Which may be corporeal or conceptual

Model classes typically require extra infrastructure

- E.g. compare, copy, store them in tables (via hashing)

Declaring Data Classes

Kotlin supports this via the 'data' keyword

When you declare a class as a data class the compiler adds extra methods

- The familiar 'equals' and 'hashCode' methods
- A straightforward implementation of 'toString'
  - Which returns output of type 'Foo(bar=123, zed="abc")'
- A 'copy' method which returns a clone of the object
- Component methods for destructuring (see later)

The 'copy' will copy properties by default but allows you to override values

- E.g. `val person2 = person1.copy(age=28)`

Declaring Data Classes

```
package oo.basics

data class Person(val name: String, val age: Int)
```

```
$ javap -p  oo.basics.Person

Compiled from "Person.kt"
public final class oo.basics.Person {
  private final java.lang.String name;
  private final int age;
  public final java.lang.String getName();
  public final int getAge();
  public oo.basics.Person(java.lang.String, int);
  public final java.lang.String component1();
  public final int component2();
  public final oo.basics.Person copy(java.lang.String, int);
  public static oo.basics.Person copy$default(oo.basics.Person, java.lang.String,
                                     int, int, java.lang.Object);
  public java.lang.String toString();
  public int hashCode();
  public boolean equals(java.lang.Object);
}
```

oo.entities

Declaring Data Classes

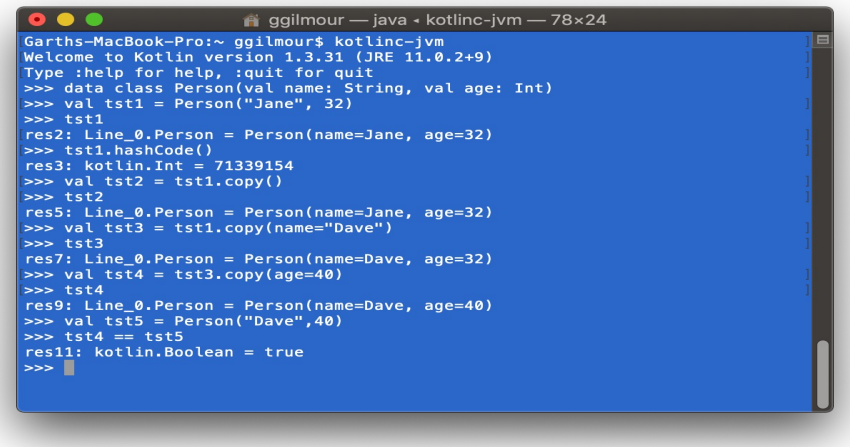
```
fun main(args: Array<String>) {
    val emp1 = Employee("Dave Jones", "10 Arcatia Road", 30000.0)
    val emp2 = Employee("Dave Jones", "10 Arcatia Road", 30000.0)

    println(emp1)
    println(emp2)
    println(emp1 === emp2)
    println(emp1 == emp2)
    println(emp1.equals(emp2))
    println(emp1.hashCode())
    println(emp2.hashCode())

    val emp3 = emp2.copy(salary = 40000.0)

    println(emp3)
}

data class Employee(val name: String, val address: String, val salary: Double)
```

```
Employee(name=Dave Jones, address=10 Arcatia Road, salary=30000.0)
Employee(name=Dave Jones, address=10 Arcatia Road, salary=30000.0)
false
true
true
1296289951
1296289951
Employee(name=Dave Jones, address=10 Arcatia Road, salary=40000.0)
```

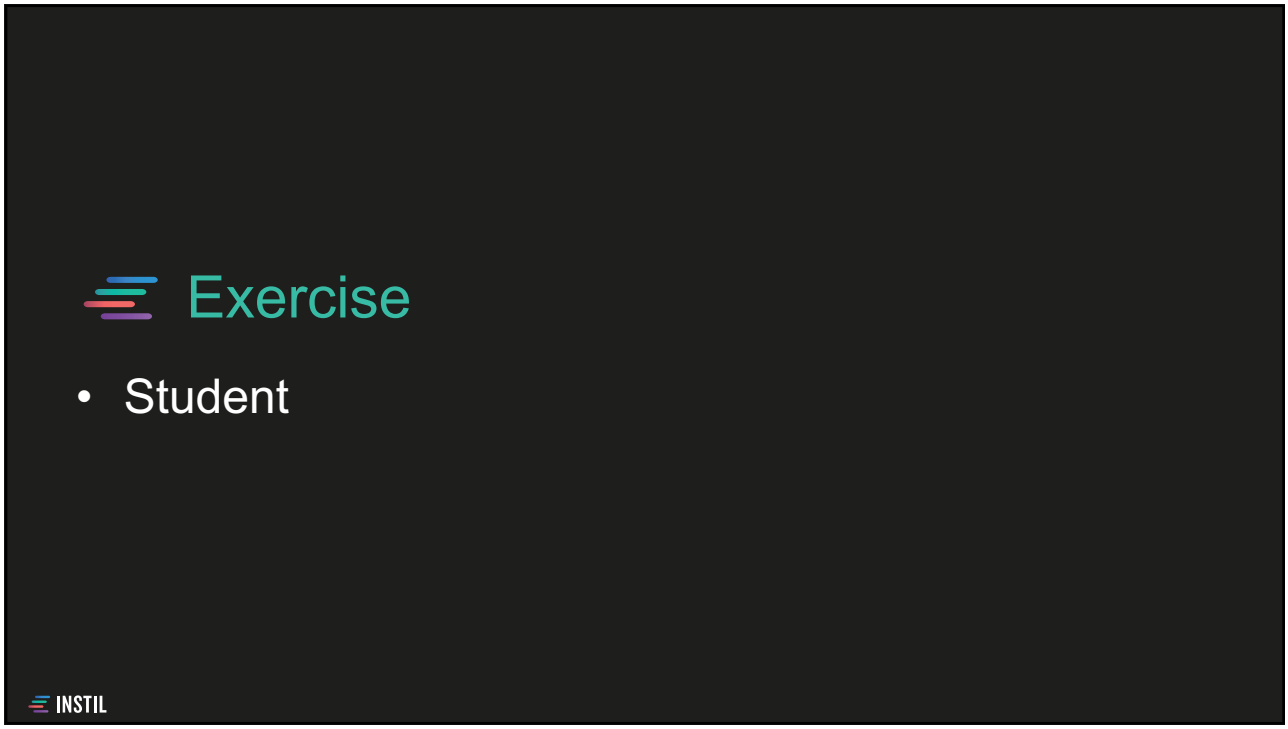
oo.entities

115

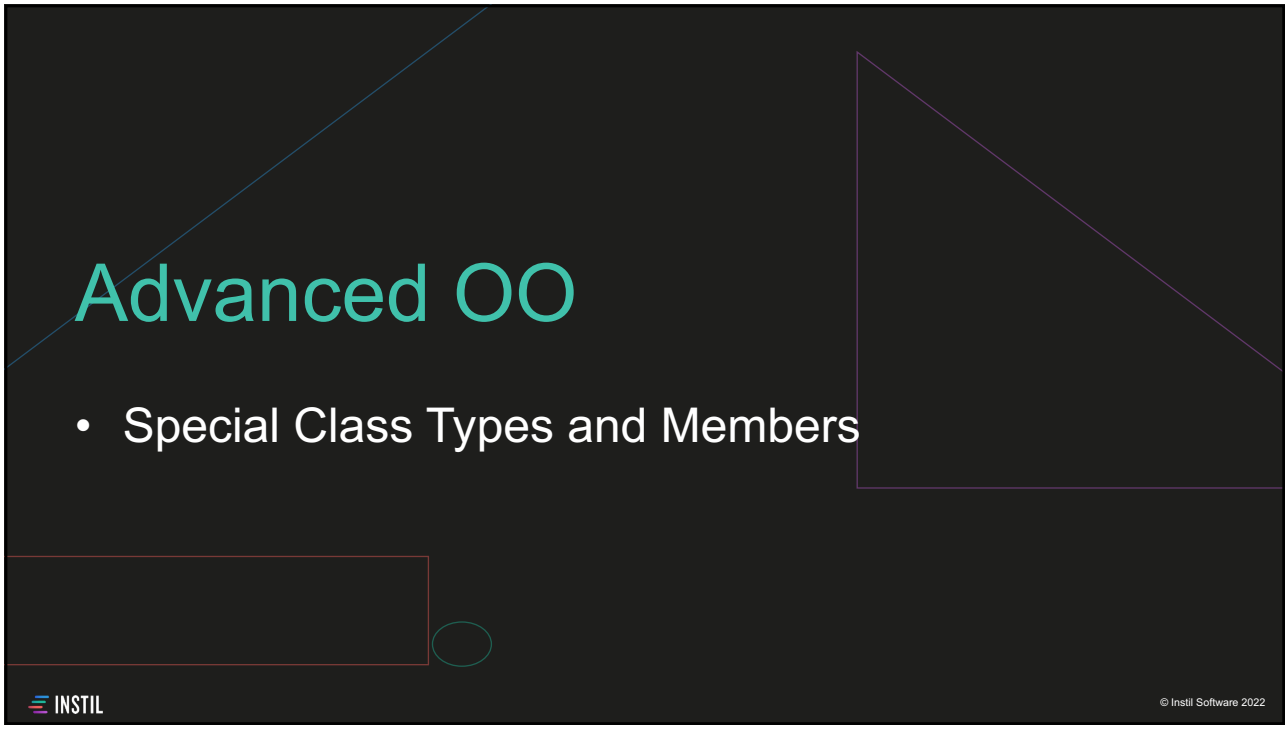
Declaring Data Classes



116


Exercise
• Student
INSTIL

117


Advanced OO
• Special Class Types and Members
INSTIL
© Instil Software 2022

118

Object Expressions

Kotlin allows you to manufacture new types ‘on the fly’

- Much like anonymous inner classes in Java

The syntax is simply ‘`object { … }`’

- Or alternatively ‘object : Foo, Bar { … }’ if you want to extend from a base class or implement one or more interfaces

This is useful when working with Listener interfaces

- For single method listeners we use a lambda or method ref
- But listeners like ‘MouseListener’ and ‘MouseMotionListener’ define a number of methods, so we require a new type
  - Since this new type will only ever be used in one place within the codebase it does not require a name

119

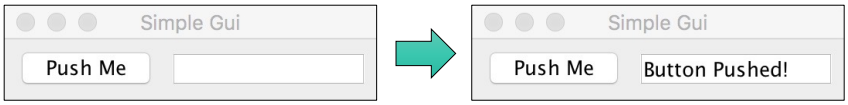
Object Expressions

```
class MyGui : JFrame("Simple Gui") {
    private val button: JButton = JButton("Push Me")
    private val textField: JTextField = JTextField(10)

    …

    private fun setEventHandling() {
        button.addActionListener({ textField.text = "Button Pushed!" })

        button.addMouseMotionListener(object : MouseMotionListener {
            override fun mouseMoved(e: MouseEvent?) {
                println("Mouse moved to ${e?.x}, ${e?.y}")
            }
            override fun mouseDragged(e: MouseEvent?) {
                println("Mouse dragged to ${e?.x}, ${e?.y}")
            }
        })
    }

    …
```

oo.expressions

120

Object Expressions




```
Mouse moved to 47, 25
Mouse moved to 56, 19
Mouse moved to 58, 16
Mouse moved to 60, 14
Mouse moved to 62, 13
Mouse moved to 63, 11
Mouse moved to 64, 10
Mouse moved to 65, 9
Mouse moved to 66, 8
Mouse moved to 67, 7
Mouse moved to 67, 7
```


Object Declarations and Companion Objects

Kotlin provides built-in support for Singletons via the object keyword

- Declaring a type with 'object' rather than class creates a single instance
- Methods of this type can be invoked via the class name
- Provides a better alternative to the Java idiom of creating classes which only hold static methods and fields

These 'object declarations' can be used in another way

- When nested within a regular class and marked with 'companion' then they become 'companion objects'
- The methods of the companion object can be invoked from the outside by using the name of the containing type
  - As with singletons this is a better alternative to a Java idiom - in this case adding static methods to normal types

Object Declarations and Companion Objects

```
object KotlinMath {
    fun add(no1: Int, no2: Int) = no1 + no2
    fun subtract(no1: Int, no2: Int) = no1 - no2
}

data class Employee(val name: String, val dept: String) {
    companion object ExtraStuff {
        fun buildForHR(name: String) = Employee(name, "HR")
        fun buildForIT(name: String) = Employee(name, "IT")
    }
}

fun main(args: Array<String>) {
    println(KotlinMath.add(12,34))
    println(KotlinMath.subtract(56,78))

    val emp1 = Employee("Jane","Sales")
    val emp2 = Employee.buildForHR("Mary")
    val emp3 = Employee.buildForIT("Lucy")

    println(emp1)
    println(emp2)
    println(emp3)
}
```

```
46
-22
Employee(name=Jane, dept=Sales)
Employee(name=Mary, dept=HR)
Employee(name=Lucy, dept=IT)
```

object.declarations

INSTIL

123

Extending Existing Types

Kotlin provides extension methods and properties

- It is possible to add extra methods and properties to existing types
- This includes built-in types
- Within these methods 'this' refers to the **receiver** object

Extensions provide a useful way to:

- Add optional functionality to your types
- Extend the functionality of types you do not control (built-in, library etc)

Note, you are not adding to the class itself

- You are just extending the set of symbols valid after the dot
- These are determined statically by the compiler
- It is determined by type of the reference rather than the object itself

INSTIL

124

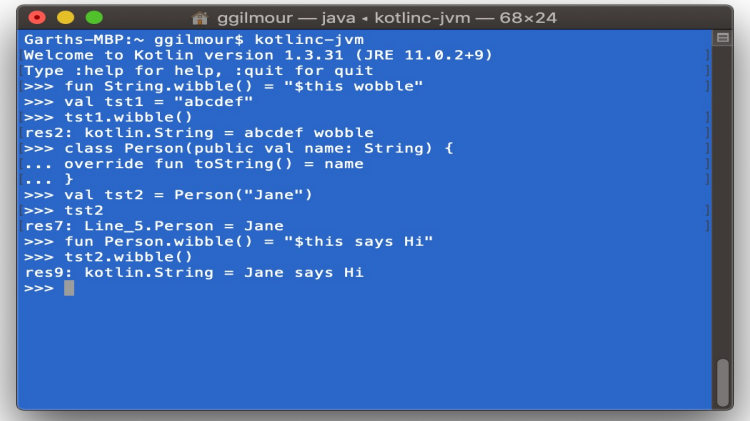
Extending Existing Types

```
data class Employee(val name: String,
                    var yearlySalary: Double) {
    fun awardBonus(bonus: Double) {
        yearlySalary += bonus
    }
}

fun Employee.calcSalary() = this.yearlySalary / 12
fun String.square() = this + this

fun main(args: Array<String>) {
    val str1 = "abc"
    val str2 = str1.square()

    val emp = Employee("Peter", 60000.0)
    emp.awardBonus(240.0)

    println(str1)
    println(str2)
    println(emp)
    println(emp.calcSalary())
}
```

```
abc
abcabc
Employee(name=Peter,
         yearlySalary=60240.0)
5020.0
```

oo.extensions

Extending Existing Types



Operator Overloading

Kotlin provides support for operator overloading

- Defining what the built in operators mean for your types
- E.g. the arithmetic operators

Not every operator can be overloaded

- As in C# you can only overload a subset of the syntax
- E.g. you can overload ‘foo == bar’ but not ‘foo === bar’

127

Operator Overloading

The syntax for operator overloading is simple

- Declare a method with one of the special names
- Ensure that the method is marked with ‘operator’
- Ensure that the parameters and return value match the contract
  - E.g. addition takes a *thing* and returns a new *thing*

128

Operator Overloading

```
class Point(var x: Int, var y: Int) {
    operator fun plus(other: Point) = Point(x + other.x, y + other.y)

    operator fun minus(other: Point) = Point(x - other.x, y - other.y)

    operator fun times(other: Point) = Point(x * other.x, y * other.y)

    operator fun div(other: Point) = Point(x / other.x, y / other.y)

    operator fun get(index: Int) = when(index) {
        0 -> x
        1 -> y
        else -> throw IllegalArgumentException()
    }

    override fun toString() = "Point at $x:$y"
}
```

functions.operator.overloading

INSTIL

129

Operator Overloading

```
fun main(args: Array<String>) {
    val p1 = Point(12,14)
    val p2 = Point(23,25)
    val p3 = p1 + p2
    val p4 = p1 - p2
    val p5 = p1 * p2
    val p6 = p1 / p2

    println(p1)
    println(p2)
    println(p3)
    println(p4)
    println(p5)
    println(p6)

    println("Indexing gives $p1[0] and $p1[1]")
}
```

```
Point at 12:14
Point at 23:25
Point at 35:39
Point at -11:-11
Point at 276:350
Point at 0:0
Indexing gives 12 and 14
```

functions.operator.overloading

INSTIL

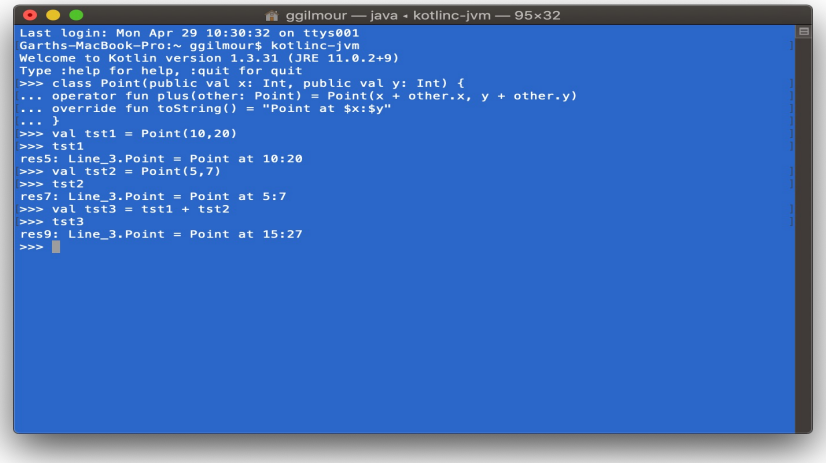
130

Operator Overloading in Kotlin

Type of Operator	Example
Unary Operators	+foo, -foo, !foo, foo++, bar--
Arithmetic Operators	foo + bar, foo - bar
Comparison Operators	foo > bar, foo <= bar
The 'in' Operator	foo in bar, foo !in bar
Indexing Operator	foo[bar], foo[0] = bar
Invocation Operator	foo(bar), foo(123, bar)
Augmented Assignment	foo += bar, foo *= bar
Equality Operators	foo == bar, foo != bar

131

Operator Overloading



132

Inheritance in Kotlin

Inheritance in Kotlin is straightforward
- We put the name of the base type after a colon in the class declaration
- Along with arguments to the base constructor

Parameters passed to the base should not be declared with the 'val' or 'var'
- Because we do not want the compiler to declare a second set of properties

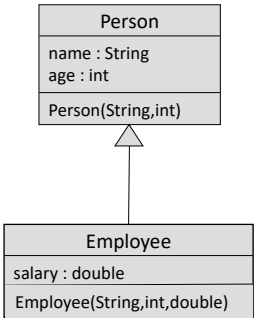
Secondary constructors must still call the primary one
- Which in turn must always call a base class constructor
- This is to ensure initialization works from the top down

If there is no primary constructor then each secondary constructor can call an arbitrary base class constructor

133

Inheritance in Kotlin

```
open class Person(val name: String,
                  val age: Int) {
}

class Employee(name: String,
               age: Int,
               val salary: Double) : Person(name, age) {
}
```



134

Inheritance in Kotlin

```
open class Person(val name: String, val age: Int) {
    constructor(name: String) : this(name, 27)
}

class Employee(name: String, age: Int, val salary: Double) : Person(name, age) {
    override fun toString(): String = "$name of age $age earning $salary"
}

class Customer : Person {
    private var address: String = ""

    constructor(name: String, age: Int, address: String) : super(name, age) {
        this.address = address
    }
    constructor(name: String, address: String) : super(name) {
        this.address = address
    }
    override fun toString(): String = "$name of age $age working at $address"
}
```

oo.inheritance

INSTIL

135

Inheritance in Kotlin

```
fun main(args: Array<String>) {
    val emp1 = Employee("John Smith", 27, 30000.0)
    val cust1 = Customer("Jane Smith", 32, "12 Arcatia Road")
    val cust2 = Customer("Mary Wilson", 34, "14 Arcatia Road")

    println(emp1)
    println(cust1)
    println(cust2)
}
```

```
John Smith of age 27 earning 30000.0
Jane Smith of age 32 working at 12 Arcatia Road
Mary Wilson of age 34 working at 14 Arcatia Road
```

INSTIL

136

Overriding Methods

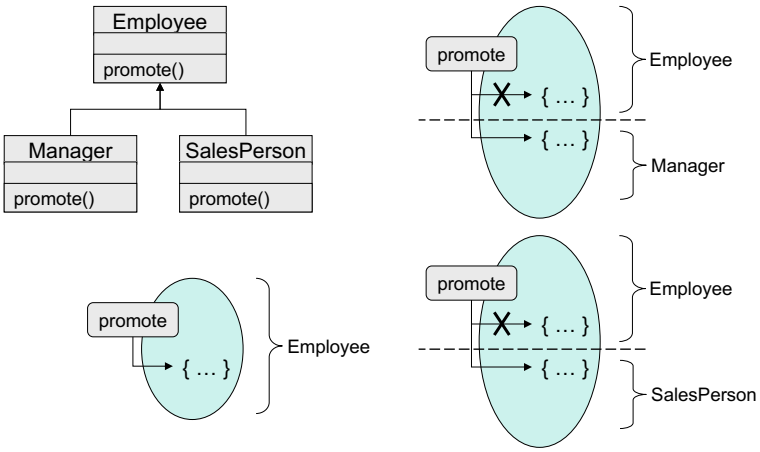
When inheriting you can choose to override methods

- Think of a method as a member of an object which contains the address of a block of byte code (the implementation)
- When we override a method in a derived class we retain the member but 'rewire it' into a different implementation

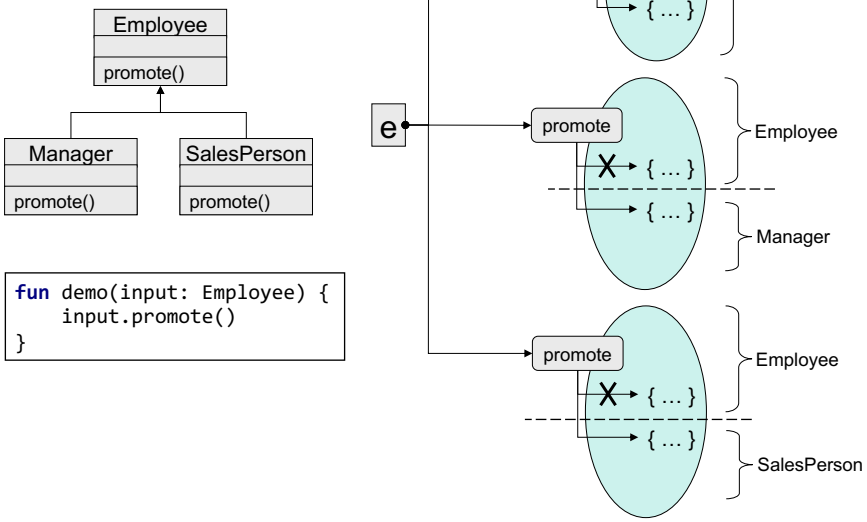
Unlike Java, **methods cannot automatically be overridden**

- In Java terms, methods, like classes, are final by default
- **Use the 'open' keyword to enable polymorphism**

Overriding Methods



Overriding Methods



139

Overriding Methods

```
open class Employee(val name: String) {
    open fun promote() = println("Employee $name has been promoted")
}

class Developer(name: String) : Employee(name) {
    override fun promote() = println("Developer $name has been promoted")
}

class Manager(name: String) : Employee(name) {
    override fun promote() = println("Manager $name has been promoted")
}

class SalesPerson(name: String) : Employee(name) {
    override fun promote() = println("Sales person $name has been promoted")
}
```

oo.overriding

140

Overriding Methods

```
fun main(args: Array<String>) {
    fun demo(input: Employee) {
        input.promote()
    }
    val emp1 = Developer("Jane")
    val emp2 = Manager("Mary")
    val emp3 = SalesPerson("Lucy")

    demo(emp1)
    demo(emp2)
    demo(emp3)
}
```

```
Developer Jane has been promoted
Manager Mary has been promoted
Sales person Lucy has been promoted
```

oo.overriding

Overriding and Collections

Overriding is most powerful when combined with collections

- We can define a collection of the base type
- Then populate it with instances of derived types

When we iterate over the collection and call a polymorphic method:

- Each instance will respond in the correct way
- The client code does need to care about the types

Overriding and Collections

This idiom is very common in GUI libraries:

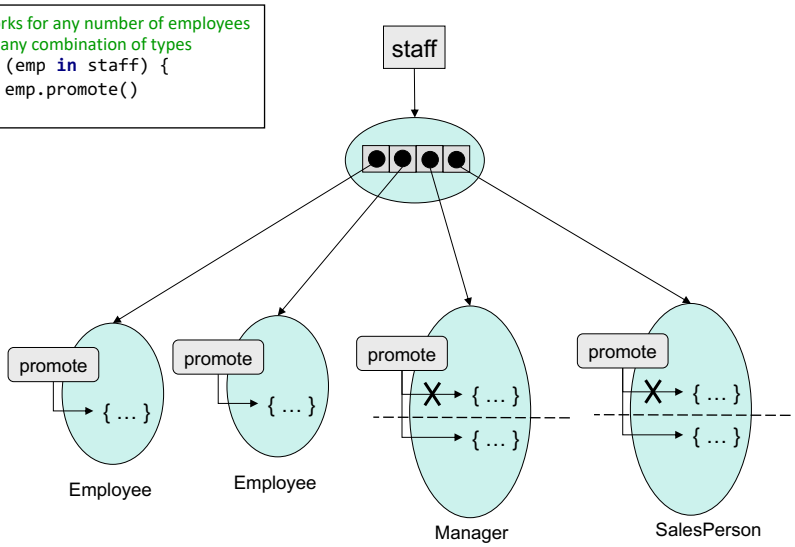
- The 'Window' type has a collection of references of a 'Component' type
- Widgets like 'Button', inherit from 'Component' and override 'render'

When the window is asked to display it renders itself

- ... and then calls 'render' on all the components in the collection

143

```
//works for any number of employees
// in any combination of types
for (emp in staff) {
    emp.promote()
}
```



144

Overriding and Collections

```
open class Employee(val name: String) {
    open fun promote() = println("Employee $name has been promoted")
}
class Developer(name: String) : Employee(name) {
    override fun promote() = println("Developer $name has been promoted")
}
class Manager(name: String) : Employee(name) {
    override fun promote() = println("Manager $name has been promoted")
}
class SalesPerson(name: String) : Employee(name) {
    override fun promote() = println("Sales person $name has been promoted")
}
```

oo.overriding.collections

INSTIL

145

Overriding and Collections

```
fun main(args: Array<String>) {
    val staff = arrayListOf(Developer("Jane"),
                            Manager("Pete"),
                            SalesPerson("Lucy"),
                            Developer("Fred"),
                            Manager("Mary"))

    for (emp in staff) {
        emp.promote()
    }
}
```

oo.overriding.collections

INSTIL

146

Abstract Classes in Kotlin

An open class can still be instantiated

- Which in many cases may not be desirable

Consider a hierarchy of employee types

- Where every person is some kind of employee and there are procedures for promoting developers, managers etc…
- If there is no procedure for promoting a regular employee then we do not want to have to stub out the method
- Instead we declare the class and method as ‘abstract’

147

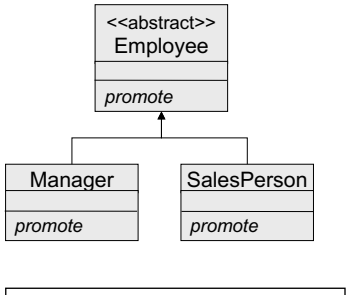
Abstract Classes in Kotlin

```
abstract class Employee(val name: String) {
    abstract fun promote()
}

class Developer(name: String) : Employee(name) {
    override fun promote() = println("Developer $name promoted")
}

class Manager(name: String) : Employee(name) {
    override fun promote() = println("Manager $name promoted")
}

class SalesPerson(name: String) : Employee(name) {
    override fun promote() = println("Sales person $name promoted")
}
```

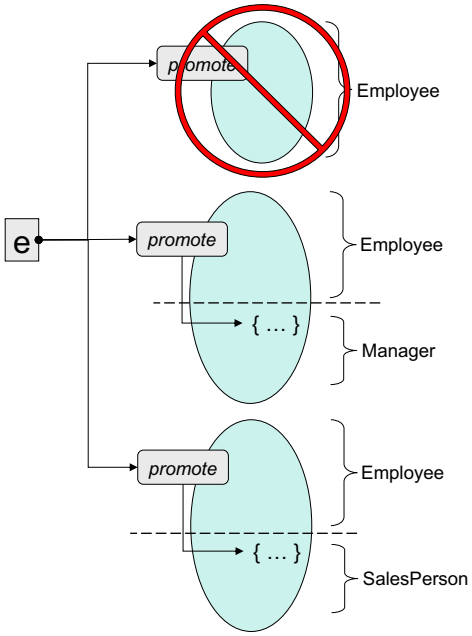
oo.classes.abstract

148

Abstract Classes in Kotlin




```
fun demo(input: Employee) {
    input.promote()
}
```



149

Interfaces in Kotlin

Interfaces in Kotlin are similar to those in Java

- An interface represents a contract to be implemented
  - Typically once per vendor, platform or product
  - A good example is transactions in databases

A Kotlin interface may contain methods and properties

- Both can optionally be given a default implementation

Interfaces cannot contain state

- Backing fields and not created for properties
- Hence the ‘field’ reference cannot be used

150

Interfaces in Kotlin

```kotlin
interface Transaction {
    val timeout: Int
    var retries: Int

    fun start()
    fun commit()
    fun rollback()
}
```


Interfaces in Kotlin

```kotlin
class MySQLTransaction(override var retries: Int) : Transaction {
    override val timeout: Int
        get() = 3000

    override fun start() = println("MySQL transaction started")
    override fun commit() = println("MySQL transaction committed")
    override fun rollback() = println("MySQL transaction rolled back")
}

class PostgreSQLTransaction(override var retries: Int) : Transaction {
    override val timeout: Int
        get() = 4000

    override fun start() = println("PostgreSQL transaction started")
    override fun commit() = println("PostgreSQL transaction committed")
    override fun rollback() = println("PostgreSQL transaction rolled back")
}
```

Interfaces in Kotlin

```
fun main(args: Array<String>) {
    fun useTransaction(t: Transaction) {
        println("Timeout is ${t.timeout / 1000} seconds")
        println("Number of retries is ${t.retries}")
        t.start()
        t.commit()
    }

    val t1 = MySQLTransaction(7)
    val t2 = PostgreSQLTransaction(9)
    useTransaction(t1)
    useTransaction(t2)
}
```

```
Timeout is 3 seconds
Number of retries is 7
MySQL transaction started
MySQL transaction committed
Timeout is 4 seconds
Number of retries is 9
PostgreSQL transaction started
PostgreSQL transaction committed
```

oo.interfaces

153

Resolving Interface Conflicts

A class can implement many interfaces

- Even when they have methods with the same signature
- …and even if those methods have default implementations

The class can provide its own version

- In which case this is the only implementation visible

But it can also selectively reuse one of the defaults

- In the following example class ‘Thing’ implements ‘Shop’ and ‘Connection’ - each of which has ‘open’ and ‘close’ methods
- We choose to implement ‘open’ using the implementation in ‘Shop’ and ‘close’ using the implementation in ‘Connection’

154

Resolving Interface Conflicts

```
interface Shop {
    fun open() = println("Shop open")
    fun close() = println("Shop closed")
}

interface Connection {
    fun open() = println("Connection open")
    fun close() = println("Connection closed")
}

class Thing : Shop, Connection {
    override fun open() {
        super<Shop>.open()
    }
    override fun close() {
        super<Connection>.close()
    }
}
```

oo.interfaces.conflicts

155

Resolving Interface Conflicts

```
fun main(args: Array<String>) {
    val thing = Thing()
    thing.open()
    thing.close()
}
```

```
Shop open
Connection closed
```

oo.interfaces.conflicts

156

Enumerations in Kotlin

All programming languages require enumerations
- To represent sets of values, such as points of the compass

In most languages enums are integers in disguise
- So an enum containing RED, BLUE and GREEN might have the values encoded as RED = 0, BLUE = 1 and GREEN = 2

In both Kotlin and Java enumerations are objects
- The enumeration is a class and each member is an instance

This has a number of benefits:
- You can add properties to the enum type
- You can extend the enum type with methods
- Utility methods are provided for you

157

Enumerations in Kotlin

```
enum class CartoonCharacter(val quote: String) {
    BART("Don't have a cow, man!"),
    LISA("Our lives have taken an odd turn"),
    HOMER("Hmmmmmmmmm forbidden donut"),
    MARGE("Oh honey, you're not a monster")
}
```

basics.enums.CartoonCharacter

158

Enumerations in Kotlin

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

    fun print() = when (this) {
        NORTH -> {
            println("/\\")
            println("/\\")
            println("/\\")
        }
        SOUTH -> {
            println("\\/")
            println("\\/")
            println("\\/")
        }
        EAST -> println(">>>")
        WEST -> println("<<<")
    }
}
```

basics.snums.Direction

159

Enumerations in Kotlin

```
fun main(args: Array<String>) {
    for (person in CartoonCharacter.values()) {
        println("$person:\t${person.quote}")
    }
    for (direction in Direction.values()) {
        println("---------")
        direction.print()
    }
}
```

```
BART:      Don't have a cow, man!
LISA:      Our lives have taken an odd turn
HOMER:     Hmmmmmmmmm forbidden donut
MARGE:     Oh honey, you're not a monster
---------
/\
/\
/\
---------
\/
\/
\/
---------
>>>
---------
<<<
```

basics.enums.Program

160

Collections in Kotlin

- Data Structures in Depth

INSTIL

161

Collections in Kotlin

Kotlin provides data structures via '`kotlin.collections`'

- As in Java, arrays are not part of the collections library
- Support for arrays is provided in the core 'kotlin' package

Collections are typically created indirectly

- Via methods such as '`emptyList`' and '`listOf`'

Both mutable and immutable collections are provided

- '`listOf`' provides a list where adding returns a new collection
- '`mutableListOf`' provides a list whose state can be changed

At present everything is implemented via JSE collections

- So you are given a read-only view of a mutable collection

INSTIL

162

Introducing Kotlin Collections

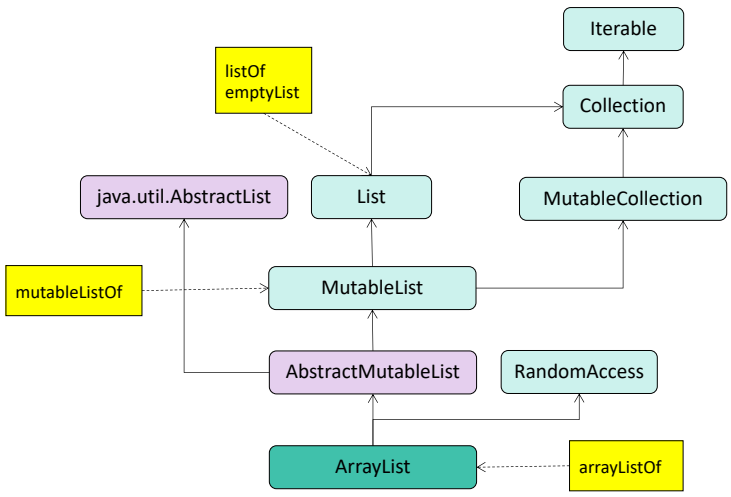
```
fun main() {
    val list1: List<Int> = emptyList()
    val list2 = emptyList<Int>()
    val list3 = listOf(12)
    val list4 = listOf(12, 34, 56)
    val list5 = mutableListOf(12, 34, 56)

    printList(list1)
    printList(list2)
    printList(list3)
    printList(list4)
    printList(list5)

    val list6 = list4.plus(78)
    list5.add(78)

    println("-------------")
    printList(list4)
    printList(list5)
    printList(list6)
}
```

collections.basic.Program

INSTIL

163

Example Types from the Collections Hierarchy



INSTIL

164

Accessing Values in Collections

There are two ways of working with collections
- So far we have used the imperative / procedural style
- In the next chapter we will see the functional style

The API provides a number of useful utility methods
- '`elementAtOrElse`' returns a value or runs a code block
- '`joinTo`' concatenates all the items into a buffer

Remember the '`for`' loop works with all iterable objects
- And that this includes ranges as well as regular collections
- A common idiom with maps is '`for ((key, value) in map) { ... }`'
  - This is an example of destructuring (see later)

Accessing Values in Collections

```
fun main(args: Array<String>) {
    val myList = listOf("ab","cd","ef")

    println(myList.elementAtOrElse(0, { "Nothing at $it“ }))
    println(myList.elementAtOrElse(3, { "Nothing at $it“ }))

    val sb = StringBuilder()
    myList.joinTo(buffer = sb, separator = " ", prefix=">>", postfix="<<")
    println(sb)

    val myList2 = myList.reversed()
    println(myList2)
}
```

```
ab
Nothing at 3
>>ab cd ef<<
[ef, cd, ab]
```

collections.functions.Program

Destructuring in Collections

Kotlin provides support for destructuring

- I.e. retrieving multiple values from a collection at once
- This is often found in scripting languages but rare elsewhere

The syntax is '`val (a, b, c) = myThing`'

- If '`myThing`' is a list this will be compiled as calls to 'get'
- Otherwise the compiler expects and calls methods of the form 'componentN'

```
val a = myThing.component1()
val b = myThing.component2()
val c = myThing.component3()
```


Destructuring in Collections

```
fun main(args: Array<String>) {
    val myList = listOf(12,34,56)
    val myPair = Pair(12,"ab")
    val (a, b, c) = myList
    val (d, e) = myPair

    println("List values are ${a}, ${b} and ${c}")
    println("Pair values are ${d} and ${e}")
}
```

```
List values are 12, 34 and 56
Pair values are 12 and ab
```

`collections.destructuring.Program`

Destructing In Collections

```
javap -p -c collections.destructuring.ProgramKt
Compiled from "Program.kt"
public final class collections.destructuring.ProgramKt {
  public static final void main(java.lang.String[]);
    Code:
      63: invokeinterface #41,  2     // InterfaceMethod java/util/List.get:(I)Ljava/lang/Object;
      68: checkcast     #43           // class java/lang/Number
      71: invokevirtual #47           // Method java/lang/Number.intValue:()I
      82: invokeinterface #41,  2     // InterfaceMethod java/util/List.get:(I)Ljava/lang/Object;
      87: checkcast     #43           // class java/lang/Number
      90: invokevirtual #47         // Method java/lang/Number.intValue:()I
     102: invokeinterface #41,  2     // InterfaceMethod java/util/List.get:(I)Ljava/lang/Object;
     107: checkcast     #43           // class java/lang/Number
     110: invokevirtual #47           // Method java/lang/Number.intValue:()I
     123: invokevirtual #51           // Method kotlin/Pair.component1:()Ljava/lang/Object;
     126: checkcast     #43           // class java/lang/Number
     129: invokevirtual #47           // Method java/lang/Number.intValue:()I
     136: invokevirtual #54           // Method kotlin/Pair.component2:()Ljava/lang/Object;
     139: checkcast     #56           // class java/lang/String
```

Calls to ‘componentN’ methods

Calls to ‘get’ method

INSTIL

169

Functional Coding

- Applying the FP style in Kotlin

INSTIL

170

Functional Features in Kotlin

You can program using **higher order functions**
- One function can be an input to another
- One function can be returned from another

You can nest functions inside one another

Function literals (aka lambdas) are available

Local functions and lambdas can act as closures
- I.e. they capture variables from enclosing scopes

Collections support the 'functional toolkit'
- Operations such as filter, map and partition

Functional Programming in Java 8 - Strings

```
ggilmour — java ◂ jshell — 70×29
Garths-MBP:~ ggilmour$ jshell
|  Welcome to JShell -- Version 11.0.2
|  For an introduction type: /help intro

jshell> var data = List.of("ab","cd","ef","gh");
data ==> [ab, cd, ef, gh]

jshell> data.
   ...> stream().
   ...> flatMapToInt(String::chars).
   ...> mapToObj(num -> (char)num).
   ...> forEach(c -> System.out.printf("%s ", c));
a b c d e f g h
jshell>
```

The Same Example in Kotlin

```
ggilmour — java ◂ kotlinc-jvm — 80×24
Garths-MBP:~ ggilmour$ kotlinc-jvm
Welcome to Kotlin version 1.3.31 (JRE 11.0.2+9)
Type :help for help, :quit for quit
>>> val data = listOf("ab","cd","ef","gh")
>>> data.flatMap {it.toList()}.forEach {print("$it ")}
>>> println()
a b c d e f g h
>>>
```

Using Functions as Inputs

In the following example we show internal iteration

The function '`invokeNTimes`' takes as arguments:

- The number of times to iterate (as an integer)
- A reference to a function that acts as the task

'`invokeNTimes`' will invoke the function as many times as required

- Passing in the current value of the counter

Functional inputs types are specified via lambda notation

- E.g. `x: (String, Int) -> Date`
- 'x' is a reference to a function which takes a String and Int and returns a Date

Passing Functions As Parameters

```
fun invokeNTimes(times: Int, func: (Int) -> Unit) {
    for (i in 1..times) {
        func(i)
    }
}

fun foo(input: Int) = println("\t foo called with $input")

fun bar(input: Int) = println("\t bar called with $input”)

fun printLine() = println("---------”)
```

functions.taking.functions

INSTIL

175

Passing Functions As Parameters

```
fun main(args: Array<String>) {
    invokeNTimes(2, ::foo)
    printLine()

    invokeNTimes(3, ::bar)
    printLine()

    invokeNTimes(4, {
        println("\t first lambda called with $it")
    })
    printLine()

    invokeNTimes(5) {
        println("\t second lambda called with $it")
    }
}
```

```
foo called with 1
foo called with 2
---------
bar called with 1
bar called with 2
bar called with 3
---------
first lambda called with 1
first lambda called with 2
first lambda called with 3
first lambda called with 4
---------
second lambda called with 1
second lambda called with 2
second lambda called with 3
second lambda called with 4
second lambda called with 5
```

functions.taking.functions

INSTIL

176

Using Functions as Inputs

A function's address is obtained via the double colon

- So '::foo' evaluates to the address of the function 'foo'
  - Where 'foo' is a function in the current scope
- 'obj::foo' evaluates to the method 'foo' on 'obj'

Function literals use the code block syntax

- Familiar to Ruby, Groovy and PowerShell developers
- So '{ it.foo() }' is the same as 'x => x.foo()' in C# or Scala
- Parameters can be named e.g. '{a, b -> a + b}'

Code blocks can be passed in two ways

- Either as regular parameters or, for the last parameter, after the parenthesis
- E.g. 'foo(123, {it.bar()})' or 'foo(123) { it.bar() }'

The 'Functional Toolkit'

Internal iteration is used to implement the 'functional toolkit'

- There are only so many common patterns of iteration
- Each of these can be modelled as a method on collections

The '`map`' method is used to perform a transformation

- The input is a list of T and the output a list of U
- We provide a function that converts a **single** T to a U

The '`filter`' method returns a smaller list

- We provide a function that acts as a **predicate**

The '`partition`' method splits the collection

- As with 'filter' we provide a predicate to act as a test
- But we get back a pair of lists instead of only one

The FP Toolkit in Java 8 - Partition

```
ggilmour — java ◂ jshell — 93×32
Last login: Sun Apr 28 21:38:59 on ttys005
Garths-MBP:~ ggilmour$ jshell
|  Welcome to JShell -- Version 11.0.2
|  For an introduction type: /help intro

jshell> import static java.util.stream.Collectors.partitioningBy;

jshell> var data = List.of("aBc","DeFg","HiJ","kLmN","oPq","RsTu");
data ==> [aBc, DeFg, HiJ, kLmN, oPq, RsTu]

jshell> var results = data.stream().collect(partitioningBy(s -> s.length() == 3));
results ==> {false=[DeFg, kLmN, RsTu], true=[aBc, HiJ, oPq]}

jshell> results.get(true).stream().map(s -> s.toUpperCase()).forEach(System.out::println);
ABC
HIJ
OPQ

jshell> results.get(false).stream().map(s -> s.toLowerCase()).forEach(System.out::println);
defg
klmn
rstu

jshell>
```

179

The Same Example in Kotlin

```
ggilmour — java ◂ kotlinc-jvm — 71×24
Garths-MBP:~ ggilmour$ kotlinc-jvm
Welcome to Kotlin version 1.3.31 (JRE 11.0.2+9)
Type :help for help, :quit for quit
>>> val data = listOf("aBc","DeFg","HiJ","kLmN","oPq","RsTu")
>>> val(short,long) = data.partition {it.length == 3}
>>> short.map {it.toUpperCase()}.forEach(::println)
ABC
HIJ
OPQ
>>> long.map {it.toLowerCase()}.forEach(::println)
defg
klmn
rstu
>>>
```

180

The Built In Functional Toolkit

Kotlin collections offer the standard 'functional toolkit'

- A summary of these operations is provided below
- Remember you can add your own via extensions

In the following example we:

- Create a list of course objects with associated instructors
- Query the data in increasingly complex ways

Operation Name	Description
all, any, name	Perform a test using a predicate
first, filter, find, last	Locate elements based on a predicate
map, flatMap	Transform the elements in a collection
fold, foldRight, reduce, reduceRight	Build a single value based on the elements
groupBy, associateBy	Group the elements based on a key selector

Using the Functional Toolkit

```
data class Course(
    val id: String,
    val title: String,
    val courseType: CourseType,
    val duration: Int,
    val instructors: List<Trainer>)

data class Trainer(
    val name: String,
    val rate: Double,
    val skills : List<String>)

enum class CourseType {
    BEGINNER,
    INTERMEDIATE,
    ADVANCED
}
```

`functional.using.the.toolkit`

Using the Functional Toolkit

```
fun buildData(): List<Course> {
    val dave = Trainer("Dave Jones", 500.0, arrayListOf("SQL","Perl","PHP"))
    val jane = Trainer("Jane Smith", 750.0, arrayListOf("SQL","Java","JEE"))
    val pete = Trainer("Pete Hughes", 1000.0, arrayListOf("Java", "JEE", "C#", "Scala"))
    val mary = Trainer("Mary Donaghy", 1250.0, arrayListOf("Java", "JEE", "C#", "C++"))

    return arrayListOf(
       Course("AB12","Intro to Scala",BEGINNER,4,arrayListOf(pete)),
       Course("CD34","JEE Development",INTERMEDIATE,3,arrayListOf(pete, mary, jane)),
       Course("EF56","Meta-Programming in Ruby",ADVANCED,2,arrayListOf()),
       Course("GH78","OO Design with UML",BEGINNER,3,arrayListOf(jane,pete,mary)),
       Course("IJ90","Database Access with JPA",INTERMEDIATE,3, arrayListOf(jane)),
       Course("KL12","Design Patterns in C#",ADVANCED,2, arrayListOf(pete,mary)),
       Course("MN34","Relational Database Design",BEGINNER,4, arrayListOf(jane, dave)),
       Course("OP56",”MySql Stored Procedures",INTERMEDIATE,1, arrayListOf(jane, dave)),
       Course("QR78",”Parallel Programming",ADVANCED,2, arrayListOf(pete,mary)),
       Course("ST90","C++ Programming for Beginners",BEGINNER,5,arrayListOf(mary)),
       Course("UV12","UNIX Threading with PThreads",INTERMEDIATE,2,arrayListOf()),
       Course("WX34","Writing Linux Device Drivers",ADVANCED,3,arrayListOf(mary)))
}
```

functional.using.the.toolkit

INSTIL

183

Using the Functional Toolkit

```
fun main(args: Array<String>) {
    val data = buildData()
    titlesOfCourses(data)
    titlesOfCoursesWithoutATrainer(data)
    namesAndRatesOfTrainers(data)
    theNumberOfAdvancedCourses(data)
    totalDurationsOfBeginnerAndNonBeginnerCourses(data)
    everyPairOfTrainersThatCouldDeliverJava(data)
    possibleCostsOfJeeWebDevelopment(data)
    coursesIdsAndTitlesGroupedByType(data)
}
```

functional.using.the.toolkit

INSTIL

184

Using the Functional Toolkit

```
fun printTabbed(input: Any) = println("\t$input")
fun printTitle(title: String) = println(title)
fun printLine() = println("--------")

//An extension function to return pairs of items without duplicates
// note this is the simplest possible implementation
fun List<String>.combinations(): List<Pair<String,String>> {
    val results = ArrayList<Pair<String,String>>()
    for (str1 in this) {
        for (str2 in this) {
            if (!str1.equals(str2)) {
                if (!results.contains(Pair(str2,str1))) {
                    results.add(Pair(str1, str2))
                }
            }
        }
    }
    return results
}
```

functional.using.the.toolkit

185

Using the Functional Toolkit

```
fun titlesOfCourses(courses: List<Course>) {
    printTitle("Titles of courses:")
    courses.map { it.title }
           .forEach(::printTabbed)
    printLine()
}

fun titlesOfCoursesWithoutATrainer(courses: List<Course>) {
    printTitle("Titles of courses without a trainer:")
    courses.filter { it.instructors.isEmpty() }
           .map { it.title }
           .forEach(::printTabbed)
    printLine()
}

fun namesAndRatesOfTrainers(courses: List<Course>) {
    printTitle("Names and rates of trainers:")
    courses.flatMap { it.instructors }
           .toSet()
           .map { Pair(it.name,it.rate)}
           .forEach(::printTabbed)
    printLine()
}
```

functional.using.the.toolkit

186

Using the Functional Toolkit

```
fun theNumberOfAdvancedCourses(courses: List<Course>) {
    printTitle("The number of advanced courses:")
    printTabbed(courses.count {it.courseType == ADVANCED})
    printLine()
}

fun totalDurationsOfBeginnerAndNonBeginnerCourses(courses: List<Course>) {
    printTitle("Total days for both beginner and non-beginner courses")
    val splitCourses = courses.partition {it.courseType == BEGINNER}
    val beginnerDuration = splitCourses.first.map{it.duration}.sum()
    val nonBeginnerDuration = splitCourses.second.map{it.duration}.sum()
    printTabbed(Pair(beginnerDuration,nonBeginnerDuration))
    printLine()
}

fun everyPairOfTrainersThatCouldDeliverJava(courses: List<Course>) {
    printTitle("Pairs of trainers that could deliver Java")
    courses.flatMap {it.instructors}
           .distinct()
           .filter {it.skills.contains("Java")}
           .map {it.name}
           .combinations()
           .forEach {printTabbed("${it.first} and ${it.second}")}
    printLine()
}
```

functional.using.the.toolkit

187

Using the Functional Toolkit

```
fun possibleCostsOfJeeWebDevelopment(courses: List<Course>) {
    printTitle("Possible costs of 'JEE Web Development'")
    val course = courses.find {it.title.equals("JEE Web Development")}
    val duration = course?.duration ?: 0
    val namesAndCosts = course
                        ?.instructors
                        ?.map { Pair(it.name, it.rate * duration)}
    namesAndCosts?.forEach(::printTabbed)
    printLine()
}
```

functional.using.the.toolkit

188

Using the Functional Toolkit

```
fun coursesIdsAndTitlesGroupedByType(courses: List<Course>) {
    fun process(entry: Entry<CourseType, List<Course>>) {
        printTabbed(entry.key)
        entry.value.forEach { println("\t\t${it.id} ${it.title}")}
    }
    printTitle("Course ID's and titles grouped by type")
    courses.groupBy {it.courseType}
           .forEach(::process)
}
```

functional.using.the.toolkit

189

Using the Functional Toolkit

```
Titles of courses:
        Intro to Scala
        JEE Web Development
        Meta-Programming in Ruby
        OO Design with UML
        Database Access with JPA
        Design Patterns in C#
        Relational Database Design
        MySql Stored Procedures
        Parallel Programming
        C++ Programming for Beginners
        UNIX Threading with PThreads
        Writing Linux Device Drivers
--------
Titles of courses without a trainer:
        Meta-Programming in Ruby
        UNIX Threading with Pthreads
--------
```


190

Using the Functional Toolkit

```
Names and rates of trainers:
        (Pete Hughes, 1000.0)
        (Mary Donaghy, 1250.0)
        (Jane Smith, 750.0)
        (Dave Jones, 500.0)
--------
The number of advanced courses:
        4
--------
Total days for both beginner and non-beginner courses
        (16, 18)
--------
Pairs of trainers that could deliver Java
        Pete Hughes and Mary Donaghy
        Pete Hughes and Jane Smith
        Mary Donaghy and Jane Smith
--------
```


Using the Functional Toolkit

```
Possible costs of 'JEE Web Development'
        (Pete Hughes, 3000.0)
        (Mary Donaghy, 3750.0)
        (Jane Smith, 2250.0)
--------
Course ID's and titles grouped by type
        BEGINNER
                AB12 Intro to Scala
                GH78 OO Design with UML
                MN34 Relational Database Design
                ST90 C++ Programming for Beginners
        INTERMEDIATE
                CD34 JEE Web Development
                IJ90 Database Access with JPA
                OP56 MySql Stored Procedures
                UV12 UNIX Threading with PThreads
        ADVANCED
                EF56 Meta-Programming in Ruby
                KL12 Design Patterns in C#
                QR78 Parallel Programming
                WX34 Writing Linux Device Drivers
```

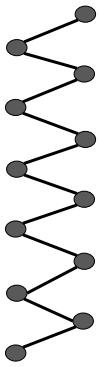
The Power of Reduce

Reduction is one of the most powerful concepts in FP

- It is an article of faith in FP that a reduction can do anything
- This is because it is a general purpose iterator

The process is illustrated on the following slides

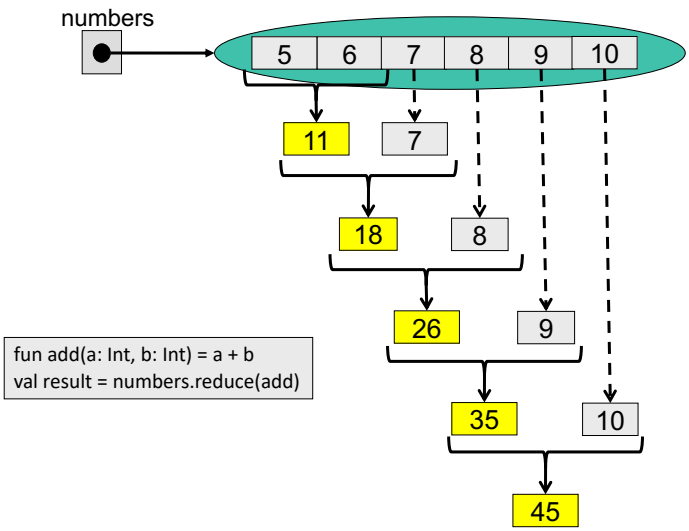
- We apply the function to the first two items
  - Or to the initial value and the first value from the list
- The result is fed into another call to the function
  - Along with the next value from the list
- The process continues till the end of the list
  - We have folded value together to reach a result
  - NB this is a left fold, you can also fold from the right



INSTIL

193

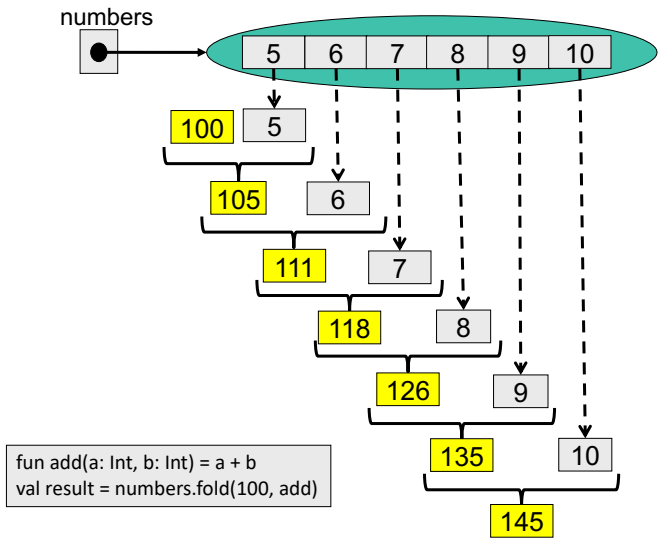
Reduce



INSTIL

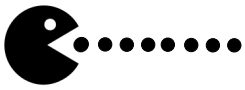
194

Reduce with Initial Value



195

Performing Reductions in Kotlin



Kotlin has excellent support for reductions

- You can reduce a collection from either the left or the right
- It follows the Scala convention of referring to a reduce with an initial value as a 'fold' (again from either the left or right)

Playing with reductions is good for learning FP

- It helps train your mind to solve problems in functional ways
  - Think of the reducer like 'pac-man' consuming items
- Its also a commonly asked question on interviews ☺

In the following demo we use reduce to solve some puzzles

- Note that many of these have simpler solutions but we are solving them via 'reduce' for the educational benefit…

196

Performing Reductions in Kotlin

```
fun main(args: Array<String>) {
    val data = arrayListOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    sumViaReduce(data)
    productViaReduce(data)
    countViaReduce(data)
    averageViaReduce(data)
    lastViaReduce(data)
    penultimateViaReduce(data)
    reverseViaReduce(data)
}

fun sumViaReduce(data: Iterable<Int>) {
    val result = data.reduce(Int::plus)
    println("Sum of elements is " + result)
}

fun productViaReduce(data: Iterable<Int>) {
    val result = data.reduce(Int::times)
    println("Product of elements is " + result)
}
```

functions.and.reductions

INSTIL

197

Performing Reductions in Kotlin

```
fun sumViaReduce(data: Iterable<Int>) {
    val result = data.reduce(Int::plus)
    println("Sum of elements is " + result)
}

fun productViaReduce(data: Iterable<Int>) {
    val result = data.reduce(Int::times)
    println("Product of elements is " + result)
}
```

functions.and.reductions

INSTIL

198

Performing Reductions in Kotlin

```
fun countViaReduce(data: Iterable<Int>) {
    val result = data.fold(0, { sum, item -> sum + 1 })
    println("Count of elements is " + result)
}

fun averageViaReduce(data: Iterable<Int>) {
    fun operator(oldPair: Pair<Double, Double>,
                 item: Int): Pair<Double, Double> {
        return Pair(oldPair.first + item, oldPair.second + 1)
    };
    val output = data.fold(Pair(0.0, 0.0), ::operator)
    val average = output.first / output.second;
    println("Average of elements is " + average);
}

fun lastViaReduce(data: Iterable<Int>) {
    val result = data.reduce { a, b -> b }
    println("The last element is " + result);
}
```

functions.and.reductions

INSTIL

199

Performing Reductions in Kotlin

```
fun penultimateViaReduce(data: Iterable<Int>) {
    val lastTwoItems = data.fold(Pair(0, 0))
                                { a, b -> Pair(a.second, b) }
    val penultimateItem = lastTwoItems.first;
    println("The penultimate element is " + penultimateItem);
}

fun reverseViaReduce(data: Iterable<Int>) {
    val results = data.fold(emptyList<Int>())
                            { items, item -> listOf(item) + items }
    println("The list in reverse is:");
    for (value in results) {
        print("\t$value ");
    }
}
```

functions.and.reductions

INSTIL

200

Performing Reductions in Kotlin

```
Sum of elements is 55
Product of elements is 3628800
Count of elements is 10
Average of elements is 5.5
The last element is 10
The penultimate element is 9
The list in reverse is:
      10 9 8 7 6 5 4 3 2 1
```


Exercise

- Cinema

Advanced FP

- Higher Order Functions in Depth

INSTIL

203

Using Functions as Outputs

Higher order functions can also return new functions

- Functional coders never implement a ‘family’ of functions
- Instead they create a function that returns the one they need

A simple example is wrapping text in HTML tags

INSTIL

204

Using Functions as Outputs

We could define one wrapper function per tag
- This would be tedious and require periodic updates

We could write a function that took the tag name
- But this would be 'stringly typed code' and prone to typos

We could write a function to build the functions we need
- It would take a tag name and return a function
- The returned function would take arbitrary text and wraps it in the tag

The last technique removes a lot of redundancy in our code

Using Functions as Outputs

```
fun wrapInH1(input: String) = "<h1>$input</h1>"
fun wrapInH2(input: String) = "<h2>$input</h2>"

fun wrapInTag(tagName: String, input: String): String {
    val openingTag = "<$tagName>"
    val closingTag = "</$tagName>"
    return "$openingTag$input$closingTag"
}

fun buildWrapper(tagName: String): (String) -> String {
    val openingTag = "<$tagName>"
    val closingTag = "</$tagName>"

    return { "$openingTag$it$closingTag" }
}
```

functions.returning.functions

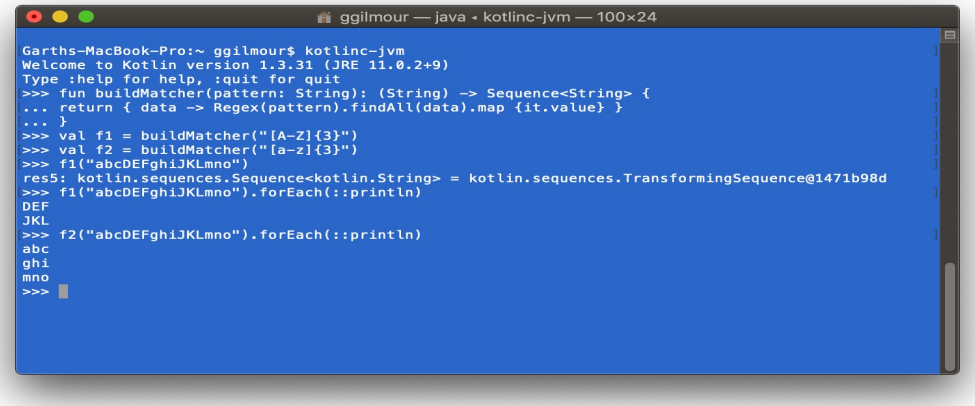
Using Functions as Outputs

```
fun main(args: Array<String>) {
    println(wrapInH1("Marge"))
    println(wrapInH2("Homer"))
    println(wrapInTag("h3","Liza"))
    println(wrapInTag("h4","Bart"))

    val wrapInBold = buildWrapper("b")
    val wrapInItalics = buildWrapper("i")
    val wrapInMark = buildWrapper("mark")

    println(wrapInBold("Maggie"))
    println(wrapInItalics("Snowball III"))
    println(wrapInMark("Santas Little Helper"))
}
```

```
<h1>Marge</h1>
<h2>Homer</h2>
<h3>Liza</h3>
<h4>Bart</h4>
<b>Maggie</b>
<i>Snowball III</i>
<mark>Santas Little Helper</mark>
```

`functions.returning.functions`

Functions as Outputs



Returning Local Functions

In the previous example we returned a function literal

- This worked well but in more complex cases we will be returning functions that contain larger quantities of code

Instead we can return a reference to a local function

- Functional languages like Kotlin allow functions to be nested within one another and these local functions to be returned

In the following example we create 'buildMatcher' which:

- Takes a regular expression in string format as its input
- Builds and returns a function that takes some text and returns all the matches for the regular expression as a list of strings

Returning Local Functions

```
fun buildMatcher(pattern: String): (String) -> Sequence<String> {
    fun matcher(text: String): Sequence<String> {
        val regex = Regex(pattern)
        return regex.findAll(text)
                    .map { it.value }
    }

    return ::matcher
}
```

returning.functions

Returning Local Functions

```
import kotlin.text.Regex

fun main(args: Array<String>) {
    val data = "abcDEFghiJKLmnpPQRstuVWXyz"
    val matcher1 = buildMatcher("[A-Z]{3}")
    val matcher2 = buildMatcher("[a-z]{2,3}")

    println("First matches are:")
    for(str in matcher1(data)) {
        println("\t$str")
    }
    println("Second matches are:")
    for(str in matcher2(data)) {
        println("\t$str")
    }
}
```

```
First matches are:
        DEF
        JKL
        PQR
        VWX
Second matches are:
        abc
        ghi
        mnp
        stu
        yz
```

returning.functions

INSTIL

211

Using Local Functions

Local functions are very helpful

- Even when you are not writing higher-order functions

They offer greater encapsulation than private methods

- If you found `duplication` across multiple public methods then you would refactor it into a private helper method
- But if you found the duplication solely within a single method then (in Java) you would again have to use a private method
- A local function better represents your intent

In the following example we generate Fibonacci numbers

- The main function is concerned with generating the number sequence
- IO tasks are delegated to local functions

INSTIL

212

Using Local Functions

```
fun main(args: Array<String>) {
    fibonacci(10000)
}

fun fibonacci(endPoint: Int) {
    fun startList() = println("<ul>")
    fun endList() = println("</ul>")
    fun addItem(item: Int) = println("\t<li>$item</li>")

    var firstNum = 0
    var secondNum = 1

    startList()
    addItem(firstNum)
    addItem(secondNum)
    while(secondNum < endPoint) {
        val result = firstNum + secondNum
        addItem(result)
        firstNum = secondNum
        secondNum = result
    }
    endList()
}
```

```
<ul>
    <li>0</li>
    <li>1</li>
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>5</li>
    <li>8</li>
    <li>13</li>
    <li>21</li>
    <li>34</li>
    <li>55</li>
    <li>89</li>
    <li>144</li>
    <li>233</li>
    <li>377</li>
    <li>610</li>
    <li>987</li>
    <li>1597</li>
    <li>2584</li>
    <li>4181</li>
    <li>6765</li>
    <li>10946</li>
</ul>
```

functions.local

INSTIL

213

Working With Closures

There is an important part of FP that we have yet to mention

- It has been assumed in some of the previous examples

Local functions and literals can act as closures

- When function ‘A’ builds and returns ‘B’ then code within ‘B’ can reference and use the local variables declared in ‘A’
- We say that ‘B’ can enclose the local variables of ‘A’

INSTIL

214

Working With Closures

Many coders incorrectly assume enclosure means copying

- When a variable is enclosed we are not making a copy
- Instead the nested function takes over ownership
  - Note that multiple local functions can share ownership
  - Due to JVM constraints this is implemented via references

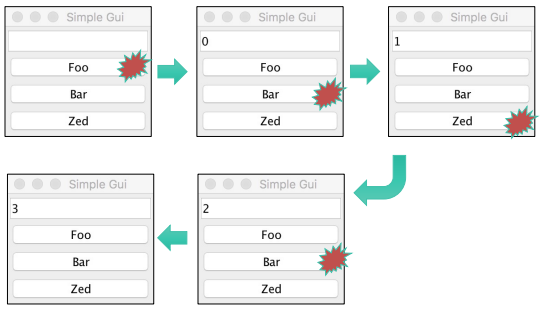
The following demo emphasises this point

- We create a simple UI with three event handlers
- All enclose (and hence increment) the same local variable

Working With Closures

```
class MyGui : JFrame("Simple Gui") {
    private val button1: JButton = JButton("Foo")
    private val button2: JButton = JButton("Bar")
    private val button3: JButton = JButton("Zed")
    private val textField: JTextField = JTextField(10)
    init {
        setLayout()
        setEventHandling()
    }
    private fun setEventHandling() {
        var x = 0
        button1.addActionListener { setText(x++) }
        button2.addActionListener { setText(x++) }
        button3.addActionListener { setText(x++) }
    }
    private fun setText(text: Any) {
        textField.text = text.toString()
    }
    private fun setLayout() {
        defaultCloseOperation = EXIT_ON_CLOSE
        layout = GridLayout(4,1)
        add(textField)
        add(button1)
        add(button2)
        add(button3)
        pack()
    }
}
```

```
fun main(args: Array<String>) {
    val gui = MyGui()
    gui.isVisible = true
}
```



functions.and.closures

Generics in Kotlin

- Programming with Type Parameters

INSTIL

217

Introducing Generics in Kotlin

As with most languages Kotlin supports Type Parameters

- These go within '< … >' on type and function declarations

The convention is to use single uppercase letters

- 'T' for 'type' and 'E' for 'element' are the most common

```
val listOfString = LinkedList<String>()
val listOfPerson = LinkedList<Person>()
```

INSTIL

218

Creating a Generic Linked List

```
class LinkedList<T> {
    private var first : Node<T>? = null

    var empty = true
        get() = size == 0

    var size = 0
        private set()

    fun add(item: T) {
        fun walkToLastNode() : Node<T>? {
            var current = first
            while(current?.next != null) {
                current = current.next
            }
            return current
        }
        if(empty) {
            first = Node<T>(null,null,item)
        } else {
            val last = walkToLastNode()
            last?.next = Node<T>(null,last,item)
        }
        size++
    }
    ...
```

generics.list.Program

INSTIL

219

Creating a Generic Linked List

```
class Person(val name: String, val age: Int) {
    override fun toString(): String {
        return "$name of age $age"
    }
}

class Node<T>(var next: Node<T>?, var previous: Node<T>?, val payload: T)

class InvalidIndexException(msg: String) : Throwable(msg)

class ListEmptyException(msg: String) : Throwable(msg)
```

generics.list.Program

INSTIL

220

Creating a Generic Linked List

```
fun main(args: Array<String>) {
    val list1 = LinkedList<String>()
    val list2 = LinkedList<Person>()

    list1.add("abc")
    list1.add("def")
    list1.add("ghi")
    list1.add("jkl")

    list2.add(Person("dave",20))
    list2.add(Person("jane",21))
    list2.add(Person("pete",22))
    list2.add(Person("mary",23))

    println("Here are the strings")
    for(x in 0 until list1.size) {
        println(list1.get(x))
    }
    println("Here are the people")
    for(x in 0 until list2.size) {
        println(list2.get(x))
    }
}
```

```
Here are the strings
abc
def
ghi
jkl
Here are the people
dave of age 20
jane of age 21
pete of age 22
mary of age 23
```

generics.list.Program

Coping With Type Variance

Sadly generics has its drawbacks

- Generics and OO are distinct kinds of programming
- Intuitions from one don't always transfer to the other

In particular generics and inheritance don't mix well

Coping With Type Variance

```
fun thoughtExperiment() {
    val myList = LinkedList<Manager>()
    foobar(myList) // should this be allowed?
}

fun foobar(input: LinkedList<Employee>)
    input.add(Employee("Dave"))
}
```

Coping With Type Variance

Most coders expect the call '`foobar(myList)`' to succeed

Since Manager derives from Employee

But this does not mean `List<Manager>` derives from `List<Employee>`

Understanding Declaration Site Variance

We can simplify our problem by restricting the generic type

- To allow only the insertion or removal of elements

A read-only list would solve the current problem

- It would be fine to pass in a list of any subtype of 'Employee'
- Since the methods of the list could not accept instances of 'T' there would be no danger of adding the wrong type

225

Understanding Declaration Site Variance

A write-only list might be useful as well

Consider a function '`fun foobar(input: WriteOnlyList<Manager>)`'

- It could be called with a write-only list of 'Employee' or 'Person'
- Since 'input.add(Manager("Jane"))' works in all cases
- Assuming of course that Manager → Employee → Person

Kotlin allows you to restrict generic types in this way

- Saying 'out T' means that 'T' objects can only be returned
- Saying 'in T' means that 'T' object can only be accepted

This is referred to as 'Declaration Site Type Variance'

226

Understanding Declaration Site Variance

The following example illustrates site variance as follows

- We create a hierarchy of Director → Manager → Employee
- We declare read-only, write-only and read-write list types
  - These use the 'in' and 'out' keywords on the generic types
  - All methods are stubbed as we only care about how the compiler will interpret the types used in the declarations
- We declare three test methods, each with a different input
  - In each case the generic parameter will be 'Manager' but the list may be read-only, write-only or read-write
- We attempt to invoke the test methods with different inputs and see which combinations will cause compiler errors

Generics in Kotlin

```
open class Employee
open class Manager : Employee()
open class Director : Manager()

class ReadWriteList<T> {
    fun get(index: Int) : T? = null
    fun findFirst(p: (T) -> Boolean) : T? = null
    fun findAll(p: (T) -> Boolean) : ReadWriteList<T>? = null
    fun add(item: T) {}
    fun add(index: Int, item: T) {}
    fun addAll(items: ReadWriteList<T>) {}
}

class ReadOnlyList<out T> {
    fun get(index: Int) : T? = null
    fun findFirst(p: (T) -> Boolean) : T? = null
    fun findAll(p: (T) -> Boolean) : ReadOnlyList<T>? = null
}

class WriteOnlyList<in T> {
    fun add(item: T) {}
    fun add(index: Int, item: T) {}
    fun addAll(vararg items: T) {}
}
```

generics.declaration.site.variance.Program.kt

Generics in Kotlin

```
fun demo1(input: ReadWriteList<Manager>) {
    input.add(Manager())    //this is not controversial
    input.add(Director())   //this surprises most people
}
fun demo2(input: ReadOnlyList<Manager>) {
    //can only access items here
}
fun demo3(input: WriteOnlyList<Manager>) {
    //can only insert items here
}
fun main(args: Array<String>) {
    // demo1(ReadWriteList<Employee>()) //will not compile
    demo1(ReadWriteList<Manager>())
    // demo1(ReadWriteList<Director>()) //will not compile

    // demo2(ReadOnlyList<Employee>()) //will not compile
    demo2(ReadOnlyList<Manager>())
    demo2(ReadOnlyList<Director>())

    demo3(WriteOnlyList<Employee>())
    demo3(WriteOnlyList<Manager>())
    // demo3(WriteOnlyList<Director>()) //will not compile
}
```

generics.declaration.site.variance.Program.kt

INSTIL

229

Understanding Declaration Site Variance

When we say 'class Foo<**out** T>' then:

- A function which takes a 'Foo<Manager>' will also accept a 'Foo<Director>' or any other subtype (such as 'Foo<CTO>')
  - Hence the possible set of valid values is infinite
- This is because 'Foo' is acting as a producer and a reference of type Manager can refer to any instance of any subtype

We say that in this case 'T' is **covariant**

class ReadOnlyList<**out** T>

INSTIL

230

Understanding Declaration Site Variance

When we say 'class Foo<**in** T>' then:

- A function which takes a 'Foo<Manager>' will also accept either a 'Foo<Employee>' or a 'Foo<Any>'
  - This time the set of valid values is fixed
- This is because 'Foo' acts as a consumer and an input of 'Manager' can be stored via a ref of 'Employee' or 'Any'

We say that in this case 'T' is **contravariant**

```
class WriteOnlyList<in T>
```


Understanding Use Site Variance

Declaration site variance is not always sufficient

Consider a function which takes two lists of type 'Employee' and where we want to transfer items between the lists

The compiler needs to be able to check, on a call by call basis, whether the operations we intend to perform are safe

- E.g. it could be called in one place with lists of 'Employee' and 'Manager' and in another with 'CTO' and 'Director'

Understanding Use Site Variance

We simplify the compilers job by using the 'in' and 'out' modifiers directly on the parameters to the function

- E.g. **fun** demo(p1: MyList<**out** Thing>, p2: MyList<**in** Thing>)

This allows the compiler to validate the code in the function and determine whether a given call should be permitted

- It is known as 'use-site variance' or 'type projection'

```
fun copyFirstTen(input: ReadWriteList<out Employee>,
                 output: ReadWriteList<in Employee>)
```


Generics in Kotlin

```
fun copyFirstTen(input: ReadWriteList<out Employee>,
                 output: ReadWriteList<in Employee>) {
    //try to copy first 10 items
    for(x in 0..9) {
        val item = input.get(x)
        if(item != null) {
            output.add(item)
        }
    }
}
fun main(args: Array<String>) {
    val tst1 = ReadWriteList<Employee>()
    val tst2 = ReadWriteList<Manager>()
    val tst3 = ReadWriteList<Director>()

    copyFirstTen(tst1,tst1)     //fine
    copyFirstTen(tst2,tst1)     //fine
    copyFirstTen(tst3,tst1)     //fine
    //copyFirstTen(tst1,tst2)   //will not compile
    //copyFirstTen(tst1,tst3)   //will not compile
}
```

generics.use.site.variance.Program.kt

Reification and Generics in Kotlin

Generics in Java are non-reified
- Only the compiler is aware of type parameters
- The extra information they provide is erased

In Kotlin there is partial support for reification
- The compiler will produce multiple versions of the code
- A new version of the code is generated for each value of 'T'

This can only be done for inline functions
- So that the reified code can be inserted at the call site
- This severely limits the opportunities for using reification

Reification and Generics in Kotlin

```
fun <T : Any> filterByTypeV1(items: List<Any>): List<Pair<T, T>> {
    val tmp = items.filter { it is T }
    return tmp.map { Pair(it as T, T::class.createInstance()) }
}

inline fun <reified T : Any> filterByTypeV2(items: List<Any>): List<Pair<T, T>> {
    val tmp = items.filter { it is T }
    return tmp.map { Pair(it as T, T::class.createInstance()) }
}

class Customer(private val name: String = "Dave") {
    override fun toString() = "Customer called $name"
}

class Item(private val id: Int = 100) {
    override fun toString() = "Item with id $id"
}
```

Neither of these lines will compile

Program.kt

Reification and Generics in Kotlin

```
fun main() {
    val inputs = listOf(
        Customer("Jane"),
        Item(123),
        Customer("Lucy"),
        Item(456),
        Customer("Mary"),
        Item(789)
    )
    val results1 = filterByTypeV2<Customer>(inputs)
    val results2 = filterByTypeV2<Item>(inputs)

    println("Results for T == Customer")
    results1.forEach(::println)
    println("Results for T == Item")
    results2.forEach(::println)
}
```

```
Results for T == Customer
(Customer called Jane, Customer called Dave)
(Customer called Lucy, Customer called Dave)
(Customer called Mary, Customer called Dave)
Results for T == Item
(Item with id 123, Item with id 100)
(Item with id 456, Item with id 100)
(Item with id 789, Item with id 100)
```

Program.kt

Generic Constraints in Kotlin

So far we have assumed our code can be fully generified

- We can rewrite our 'LinkedList' of string as 'LinkedList<T>' and it will then automatically work for all possible values of 'T'

This is (mostly) true of collections but false in other cases

- Often we find code which can only be partially generified
- For example it might only work for any type which has a 'compareTo' method, or which implements 'Iterable'

Generic Constraints allow us to state this in code

- For example the function 'fun <T: Iterable> foobar(p: T)' can be invoked with any type which implements 'Iterable'
- As in Java this is referred to as an 'upper bound'

Generic Constraints

```
open class Employee {
    fun work() {}
}
open class Manager : Employee() {
    fun manage() {}
}
open class Director : Manager() {
    fun direct() {}
}
fun <T> demo1(input: T) {
    println(input.toString())
}
fun <T : Employee> demo2(input: T) {
    input.work()
}
fun <T : Manager> demo3(input: T) {
    input.work()
    input.manage()
}
fun <T : Director> demo4(input: T) {
    input.work()
    input.manage()
    input.direct()
}
```

generics.constraints.Program

INSTIL

239

Generic Constraints

```
fun main(args: Array<String>) {
    demo1(Employee())
    demo1(Manager())
    demo1(Director())

    demo2(Employee())
    demo2(Manager())
    demo2(Director())

    //demo3(Employee()) // Will not compile
    demo3(Manager())
    demo3(Director())

    //demo4(Employee()) // Will not compile
    //demo4(Manager())  // Will not compile
    demo4(Director())
}
```

generics.constraints.Program

INSTIL

240

Generic Constraints in Kotlin

It is possible to specify multiple generic constraints

- Useful when a type needs to implement multiple interfaces
- E.g. a type might need to be both serializable and cloneable

Multiple constraints use a different syntax

- The function declaration ends in a 'where' clause, whose value is a series of constraints on the type parameter(s)

Generic Constraints in Kotlin

```
interface Edible {
    fun eat()
}

interface Drinkable {
    fun drink()
}
```

demos.constraints.generics.multiple.Program

Generic Constraints in Kotlin

```
class Pie : Edible {
    override fun eat() {}
}

class Ale : Drinkable {
    override fun drink() {}
}

class Ramen : Edible, Drinkable {
    override fun eat() {}
    override fun drink() {}
}
```

demos.constraints.generics.multiple.Program

INSTIL

243

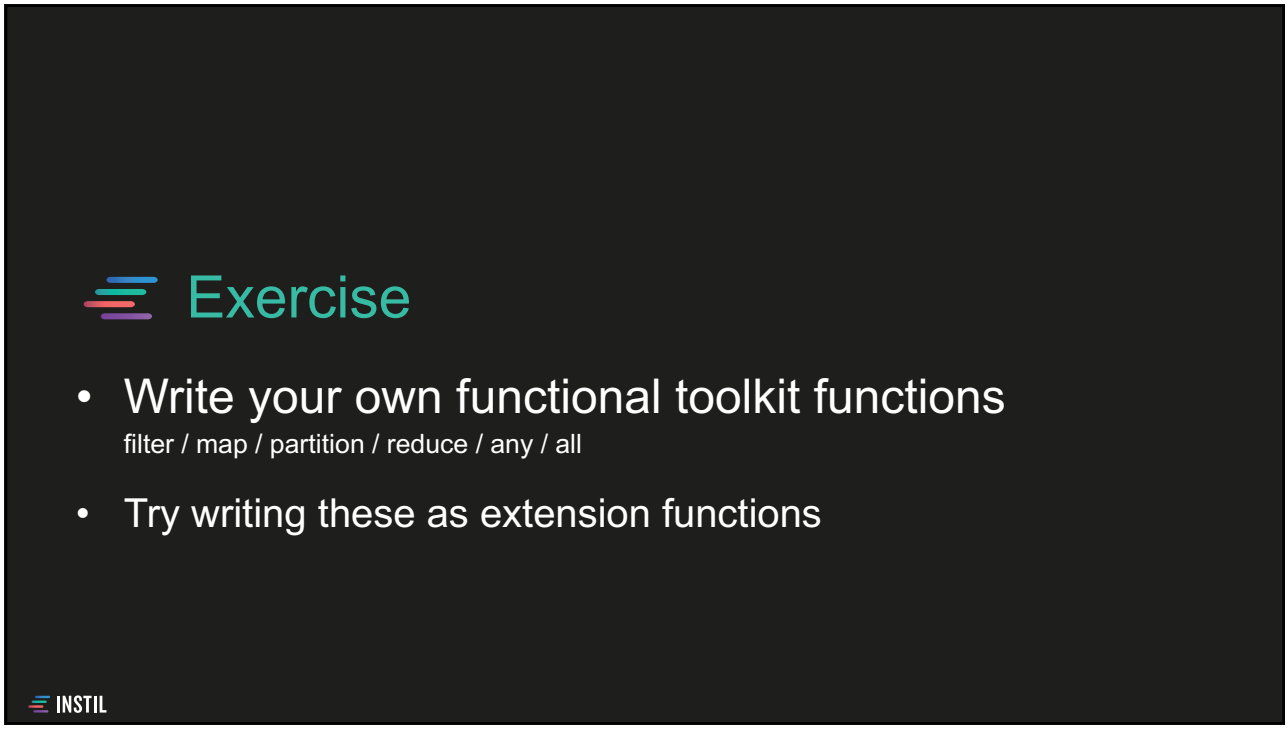
Generic Constraints in Kotlin

```
fun <T> demo(input: T) where T: Edible,
                             T: Drinkable {
    input.eat()
    input.drink()
}

fun main(args: Array<String>) {
    //demo(Pie())  // Will not compile
    //demo(Ale())  // Will not compile
    demo(Ramen())
}
```

demos.constraints.generics.multiple.Program

INSTIL

244


Exercise
• Write your own functional toolkit functions
filter / map / partition / reduce / any / all
• Try writing these as extension functions
INSTIL

245


Sequences
• Lazy Evaluation of Collections
INSTIL
© Instil Software 2022

246

Introducing Sequences

The operations we have seen so far are performed eagerly

- When you chain operations such as ‘map’ and ‘filter’ together each operator creates and returns an intermediate list

This is not as inefficient as you would expect

- Since the functional operators are very simple they are usually inlined, thereby reducing the number of intermediate lists

However it is still unwise for very large collections

- We don’t want to process millions of items in an eager fashion
- Instead we want to process the items lazily i.e. only on demand

Introducing Sequences

Kotlin provides ‘Sequences’ for lazy evaluation

- To obtain one invoke ‘**asSequence**’ on a regular collection
- Similarly ‘toList’ etc. convert sequences to collections

Sequences define their methods into two groups

- Intermediate operations represent a stage in processing
- Terminal operations return the final result to the caller

The intermediate operations do not perform work directly

- Instead the work to be done is represented as a list of lambdas
- We only execute this work queue on the terminal operation

Demonstrating Collections vs. Sequences

```
fun demoIterable() {
    println("Working with iterables")
    val input = listOf("abc","de","fgh","ij","klm")
    println("\tCreated list")

    val results = input.filter {
        println("\t\tFiltering $it")
         it.length == 3
    }.map {
        println("\t\tMapping $it")
        it.toUpperCase()
    }

    println("\tOperations called")
    println("\tResults are:")
    results.forEach { println("\t\t$it") }
}
```

```
Working with iterables
    Created list
        Filtering abc
        Filtering de
        Filtering fgh
        Filtering ij
        Filtering klm
        Mapping abc
        Mapping fgh
        Mapping klm
    Operations called
    Results are:
        ABC
        FGH
        KLM
```

collections.sequences

INSTIL

249

Demonstrating Collections vs. Sequences

```
fun demoSequence() {
    println("Working with sequences")
    val input = listOf("abc","de","fgh","ij","klm").asSequence()
    println("\tCreated sequence")

    val results = input.filter {
        println("\t\tFiltering $it")
        it.length == 3
    }.map {
        println("\t\tMapping $it")
        it.toUpperCase()
    }

    println("\tOperations called")
    println("\tResults are:")
    results.forEach { println("\t\t$it") }
}
```

```
Working with sequences
    Created sequence
    Operations called
    Results are:
        Filtering abc
        Mapping abc
        ABC
        Filtering de
        Filtering fgh
        Mapping fgh
        FGH
        Filtering ij
        Filtering klm
        Mapping klm
        KLM
```

collections.sequences

INSTIL

250

Comparing Collections to Java 8 Streams

Kotlin Sequences are very similar to Java 8 Streams

- Both build a plan which is executed on the terminal operation

Some reasons to opt for Sequences are:

- The Sequences API is much simpler than the Streams one
  - E.g. you don't have to learn all the variations of 'collect'
- A Stream can easily be transformed into a Sequence
  - By invoking the 'asSequence' extension method
- Sequences are always available to a Kotlin developer
  - Whereas Java 8 Streams are not available on Android

Java Streams have one advantage

- Operations on a Stream can be run in parallel over many cores
- Calling 'parallel' associates the Stream with a fork / join pool

INSTIL

251

Reactive Coding

- Via Project Reactor

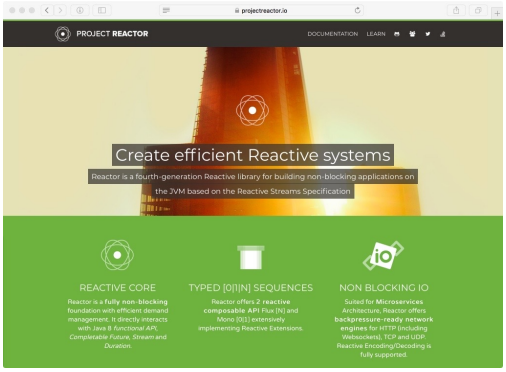
INSTIL

252

Introducing Project Reactor

Project Reactor is a library produced by Pivotal

- It is based on the Reactive Streams specification
- As of Spring 5 it is integrated into the String ecosystem



Introducing Project Reactor

Reactor is based around two key types

- A '`Flux<T>`' is a asynchronous sequence of 0..N items
- A '`Mono<T>`' is an asynchronous sequence of 0..1 items
  - It supports a more limited set of operations

The '`Mono`' type is confusing at first

- Think of it as a pipe down which only one ball will roll

Hello Project Reactor

Lets use Reactor to build a frequency table

- Holding the number of occurrences of letters in sample text

Our strategy will be to:

1. Create a '`Flux<String>`' made up of the words in the text
2. Convert each word to lowercase via the 'map' operation
3. Convert each word to a '`Flux<Char>`' and join the streams
   - This can be achieved via the '`flatMap`' operation
   - NB we need to go from a String to an Array to a Flux
4. Use 'filter' to keep only those chars that are letters
5. Group the characters via the '`collectMultimap`' operation
   - The keys will be chars and the values a list of instances
6. Convert the values in the table to a simple count of instances
7. Print out the resulting table

Hello Project Reactor

```
fun main(args: Array<String>) {
    val loremIpsum = """Lorem ipsum dolor sit amet, consectetur adipiscing
                        elit, sed do eiusmod tempor incididunt ut labore et
                        dolore magna aliqua. Ut enim ad minim veniam, quis
                        nostrud exercitation ullamco laboris nisi ut aliquip
                        ex ea commodo consequat. Duis aute irure dolor in
                        reprehenderit in voluptate velit esse cillum dolore
                        eu fugiat nulla pariatur. Excepteur sint occaecat
                        cupidatat non proident, sunt in culpa qui officia
                        deserunt mollit anim id est laborum."""

    val flux = Flux.fromIterable(loremIpsum.split(" "))
    flux.map { it.toLowerCase() }
        .flatMap { word -> Flux.fromArray(word.toCharArray()
                                          .toTypedArray()) }
        .filter { theChar -> Character.isLetter(theChar) }
        .collectMultimap { it }
        .map { table -> table.mapValues { entry -> entry.value.size } }
        .subscribe(::printTable)
}
```

Program.kt

Hello Project Reactor

```
fun printTable(input: Map<Char, Int>) {
    println("The frequency count of letters in 'lorem ipsum' is:")
    input.forEach { entry ->
        val msgWord = "instance" + if (entry.value > 1) "s" else ""
        println("${entry.value} $msgWord of ${entry.key}")
    }
}
```

```
The frequency count of letters in 'lorem ipsum' is:
29 instances of a
3 instances of b
16 instances of c
19 instances of d
38 instances of e
3 instances of f
3 instances of g
1 instance of h
42 instances of i
22 instances of l
17 instances of m
24 instances of n
29 instances of o
11 instances of p
...
```


Introducing Project Reactor

Nothing happens till you invoke '`subscribe`'

- This ties your chain of operations to the publisher

Prior to that you are in 'assembly time'

- You are assembling a chain of operations to process data
- Behind the scenes this creates a description of your intent

The subscribe method takes up to three callbacks

- The first is for when an item reaches the end of the pipe
- The second is to handle exceptions when they occur
- The final callback signals the end of processing

NB in WebFlux you do not call subscribe manually

- The framework opens and closes event streams for you

Way to Build Reactive Streams

The methods for building streams can be divided into:

- Methods for building streams from existing data
  - Such as '`just`', '`fromArray`' and '`fromIterable`'
- Methods for building streams programmatically
  - Such as '`generate`' and '`create`'

Some Different Ways to Build Reactive Streams

Method Name	Description
`empty`	Creates a stream that completes without emitting any items
`just`	Creates a stream that emits one or more items NB If you pass in an array then it is emitted as the single item
`fromIterable`	Creates a stream that emits the items from a 'java.lang.Iterable'
`fromArray`	Creates a stream that emits the items from a Java array NB You may need to convert a Kotlin / Scala Array to Java
`generate`	Creates a stream via a call-back function and some state
`create`	Creates a stream via a call-back function
`repeat`	Creates a new stream by repeatedly subscribing to an existing one
`mergeWith`	Interleaves the current stream with another one
`concatWith`	Concatenates the output of another stream with the current one

Building a Reactive Stream via 'empty'

```
fun showEmptyFlux() {
    println("--- Empty Flux ---")
    Flux.empty<String>()
          .subscribe(
            { str -> println("\tReceived $str") },
            { error -> println("\tError occurred ($error.message)") },
            { println("\tAll done") })
```

```
--- Empty Flux ---
   All done
```

Program.kt

INSTIL

261

Building a Reactive Stream via 'just'

```
fun showFluxViaJust() {
    println("--- Flux Built via 'Just' ---")
    Flux.just("abc", "def", "ghi", "jkl", "mno")
          .subscribe(
            { str -> println("\tReceived $str") },
            { error -> println("\tError occurred ($error.message)") },
            { println("\tAll done") })
}
```

```
--- Flux Built via 'Just' ---
   Received abc
   Received def
   Received ghi
   Received jkl
   Received mno
   All done
```

Program.kt

INSTIL

262

Building a Reactive Stream via ‘fromIterable’

```
fun showFluxViaIterable() {
    val data = listOf("abc", "def", "ghi", "jkl", "mno")
    println("--- Flux Built via 'Iterable' ---")
    Flux.fromIterable(data)
          .subscribe(
            { str -> println("\tReceived $str") },
            { error -> println("\tError occurred ($error.message)") },
            { println("\tAll done") })
}
```

```
--- Flux Built via 'Iterable' ---
    Received abc
    Received def
    Received ghi
    Received jkl
    Received mno
    All done
```

Program.kt

INSTIL

263

Building a Reactive Stream via ‘fromArray’

```
fun showFluxViaArray() {
    val data = arrayOf("abc", "def", "ghi", "jkl", "mno")
    println("--- Flux Built via 'Array' ---")
    Flux.fromArray(data)
          .subscribe(
            { str -> println("\tReceived $str") },
            { error -> println("\tError occurred ($error.message)") },
            { println("\tAll done") })
}
```

```
--- Flux Built via 'Array' ---
    Received abc
    Received def
    Received ghi
    Received jkl
    Received mno
    All done
```

Program.kt

INSTIL

264

Building a Reactive Stream via ‘mergeWith’

```
fun showFluxViaMerge() {
    val data1 = listOf("abc", "def", "ghi", "jkl", "mno")
    val data2 = arrayOf("pqr", "stu", "vwx", "yza")

    println("--- Flux Built via Merging ---")
    Flux.fromIterable(data1)
           .mergeWith(Flux.fromArray(data2))
          .subscribe(
             { str -> println("\tReceived $str") },
             { error -> println("\tError occurred ($error.message)") },
             { println("\tAll done") })
}
```

```
--- Flux Built via Merging ---
    Received abc
    Received def
    Received ghi
    ...
    Received stu
    Received vwx
    Received yza
    All done
```

Program.kt

Building a Reactive Stream via ‘generate’

```
fun showFluxViaGenerate() {
    val data = listOf("abc", "def", "ghi", "jkl", "mno")
    println("--- Flux Built via 'Generate' ---")
    Flux.generate<String, Int>(
           { 0 },
           { state, sink ->
               if(state < data.size) {
                  sink.next(data[state])
               } else {
                  sink.complete()
              }
              state + 1
           })
          .subscribe(
             { str -> println("\tReceived $str") },
             { error -> println("\tError occurred ($error.message)") },
             { println("\tAll done") })
}
```

```
--- Flux Built via 'Generate' ---
    Received abc
    Received def
    Received ghi
    Received jkl
    Received mno
    All done
```

Program.kt

Building a Reactive Stream via 'sink'

```
fun showFluxViaCreate() {
    val data = listOf("abc", "def", "ghi", "jkl", "mno")
    println("--- Flux Built via 'Create' ---")
    Flux.create<String>(
           { sink ->
               for(str in data) {
                  sink.next(str)
               }
               sink.complete()
           })
          .subscribe(
             { str -> println("\tReceived $str") },
             { error -> println("\tError occurred ($error.message)") },
             { println("\tAll done") })
}
```

```
--- Flux Built via 'Create' ---
    Received abc
    Received def
    Received ghi
    Received jkl
    Received mno
    All done
```

Program.kt

INSTIL

267

Understanding Schedulers

Reactor is concurrency agnostic

- It does not force you to use a particular threading model
- Both producers and consumers can be single or multi threaded

A '`Scheduler`' controls the threading models in use

- Via the '`publishOn`' and '`subscribeOn`' methods

Scheduler is just an abstraction

- Typically it is implemented using an 'ExecutorService'
- But this can be switched, for example to Coroutines

INSTIL

268

The PublishOn and SubscribeOn Operations

Calling ‘publishOn’ sets the scheduler
- For all the operations that follow on from this call

Every operation that follows uses the new threading context
- Unless another call to ‘publishOn’ is encountered

The ‘subscribeOn’ operation works differently
- Calls to ‘subscribe’ create a chain of subscription objects
- Each chain stretches backwards to the first publisher
- ‘subscribeOn’ sets the context for this process

There are two counter intuitive features
- It does not matter where the call to ‘subscribeOn’ is placed
- Each backward chain runs in a single thread from the context

Using An Asynchronous Publisher

```
fun showAsyncPublisher(pool: ExecutorService) {
    println("--- Flux With Async Publisher ---")
    Mono.fromCallable(::genValue)
            .repeat(100)
            .doOnNext { printTabbed("$it generated on ${threadId()}") }
            .publishOn(Schedulers.fromExecutor(pool))
            .doOnNext { printTabbed("$it moved to ${threadId()}") }
            .subscribe { printTabbed("$it received on ${threadId()}") }

}
```

```
--- Flux With Async Publisher ---
    39.22 generated on 1
    24.71 generated on 1
    39.22 moved to 13
    35.48 generated on 1
    39.22 received on 13
    24.71 moved to 13
    24.71 received on 13
    57.05 generated on 1
    35.48 moved to 13
    35.48 received on 13
```

Program.kt

Using An Asynchronous Subscriber

```
fun showAsyncSubscriber(pool: ExecutorService) {
    println("--- Flux With Async Subscriber ---")
    val flux = Mono.fromCallable(::genValue)
            .repeat(10)
            .doOnNext { printTabbed("$it generated on ${threadId()}") }
            .subscribeOn(Schedulers.fromExecutor(pool))
            .doOnNext { printTabbed("$it still on ${threadId()}") }

    flux.subscribe { printTabbed("$it received by S1 on ${threadId()}") }
    flux.subscribe { printTabbed("$it received by S2 on ${threadId()}") }
}
```

```
--- Flux With Async Subscriber ---
    29.26 generated on 13
    29.26 still on 13
    29.26 received by S1 on 13
    85.24 generated on 13
    85.24 still on 13
    85.24 received by S1 on 13
    40.83 generated on 13
    40.83 still on 13
```

Program.kt

INSTIL

271

Combining Async Publisher and Subscriber

```
fun showAsyncPublisherAndSubscriber(pool1: ExecutorService,
                                    pool2: ExecutorService) {
    println("--- Flux With Async Publisher And Subscriber ---")
    val flux = Mono.fromCallable(::genValue)
            .repeat(10)
            .doOnNext { printTabbed("$it generated on ${threadId()}") }
            .subscribeOn(Schedulers.fromExecutor(pool1))
            .publishOn(Schedulers.fromExecutor(pool2))
            .doOnNext { printTabbed("$it now on ${threadId()}") }

    flux.subscribe { printTabbed("$it received by S1 on ${threadId()}") }
    flux.subscribe { printTabbed("$it received by S2 on ${threadId()}") }
}
```

```
--- Flux With Async Publisher And Subscriber ---
    71.88 generated on 13
    76.69 generated on 13
    71.88 now on 15
    85.81 generated on 13
    71.88 received by S1 on 15
    76.69 now on 15
```

Program.kt

INSTIL

272

Other Threading Operators

It is possible to create a 'ParallelFlux' from a 'Flux'
- Via a call to 'parallel' followed by 'runOn'

This publishes to multiple 'rails' (aka. 'groups')
- So calls on a backward chain will be run on multiple threads
- A call to 'sequential' merges results back to a regular Flux

The 'flatMap' operation is also threaded
- It combines the results from multiple Flux / Mono
- Which themselves may have a separate context

Using Parallel Fluxes

```
fun showParallelFlux(pool: ExecutorService) {
    println("--- Parallel Flux ---")
    Mono.fromCallable(::genValue)
            .repeat(10)
            .parallel()
            .runOn(Schedulers.fromExecutor(pool))
            .subscribe {
                printTabbed("$it received by subscriber on ${threadId()}")
             }
}
```

```
--- Parallel Flux ---
    14.78 received by subscriber on 13
    02.91 received by subscriber on 12
    05.45 received by subscriber on 12
    91.06 received by subscriber on 13
    84.60 received by subscriber on 12
    05.69 received by subscriber on 13
    85.57 received by subscriber on 12
```

Program.kt

The FlatMap Operation and Threading

```
fun showAsyncFlatMap(pool: ExecutorService) {
    Flux.fromArray(arrayOf("A", "B"))
            .flatMap { str ->
                Flux.range(1, 10)
                        .map { "$str$it" }
                        .subscribeOn(Schedulers.fromExecutor(pool))
            }
            .subscribe { printTabbed("$it received on ${threadId()}") }
}
```

```
--- Async with FlatMap ---
    A1 received on 13
    B1 received on 12
    A2 received on 12
    A3 received on 12
    A4 received on 12
    A5 received on 12
    A6 received on 12
    A7 received on 12
    A8 received on 12
```

Program.kt

Using Reactor To Process GUI Events

Streams can be created in a variety of ways

- The simplest method is via '`Flux.just(item1, item2, …)`'

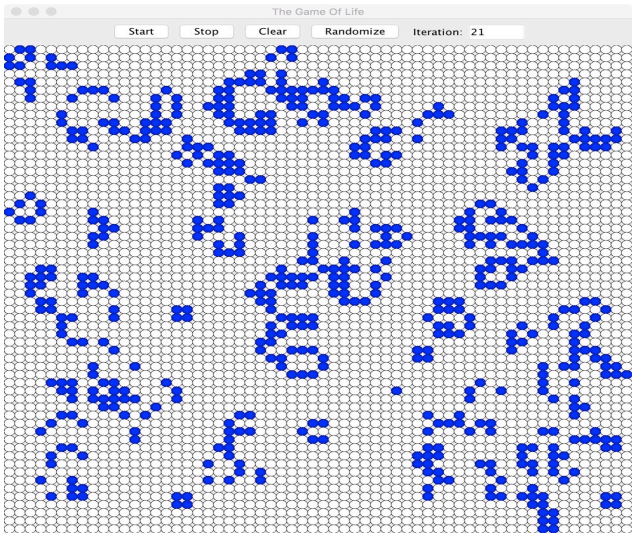
The 'create' method lets us emit events on the fly

- We supply an object or function that takes a '`FluxSink`' as input
- A new event is emitted by calling '`next`' on the sink
- Call '`complete`' when all events have been sent

This is normally combined with the Listener Pattern

- The event publisher (e.g. a Game Of Life GUI) will have one or more events with which listener objects can be associated
- When the event fires all the listeners are notified
- One of the listeners invokes 'next' on the sink

Using Reactor To Process GUI Events



Using Reactor To Process GUI Events

```
class Board(private val cellSize: Int, randomize: Boolean) : JPanel() {
    private var spacing: Int = 0
    private val cells = Array(cellSize, { Array(cellSize, { Cell() }) })
    private val listeners = mutableListOf<CellEventListener>()

    val flux: Flux<CellEvent> = Flux.create({ sink ->
        listeners.add(object : CellEventListener {
            override fun onCellStateChange(x: Int, y: Int, state: Boolean) {
                sink.next(CellEvent(x, y, state))
            }
            override fun onGameStop() {
                sink.complete()
            }
        })
    })
    fun doStateChange() {
        for (x in 0 until cellSize) {
            for (y in 0 until cellSize) {
                cells[x][y].doStateChange()
                for(listener in listeners) {
                    listener.onCellStateChange(x,y, cells[x][y].isAlive)
                }
            }
        }
    }
```

Board.kt

Using Reactor To Process GUI Events

```
fun main(args: Array<String>) {
    val gui = GameOfLife()
    gui.isVisible = true

    gui.flux
        .filter({ event -> event.alive })
        .map({ event -> "Cell at ${event.x}, ${event.y}"} )
        .subscribe(::println)
}
```

```
Cell at 20, 13
Cell at 20, 14
Cell at 20, 15
Cell at 20, 17
Cell at 20, 18
Cell at 20, 19
Cell at 20, 21
Cell at 20, 22
Cell at 20, 24
Cell at 20, 25
Cell at 20, 49
```


Project Reactor and WebFlux

Reactor is used extensively in Spring WebFlux

- WebFlux Controllers return either a ‘Mono’ or a ‘Flux’
- We will see this in detail in the WebFlux topic

Reactor is also used to receive server updates

- Such as currency fluctuations or Bids / Asks on Derivatives
- The ‘`ReactorNettyWebSocketClient`’ allows a client to process WebSocket messages as a conventional event stream
- Typically we use a Jackson ‘`ObjectMapper`’ to convert the payload of the messages from JSON to some model type

Project Reactor and WebFlux

```
fun main(args: Array<String>) {
    val uri = URI("ws://localhost:8080/courses")
    val client = ReactorNettyWebSocketClient()
    val mapper = ObjectMapper()

    client.execute(uri) { session ->
        session.receive()
                .map(WebSocketMessage::getPayloadAsText)
                .map({ mapper.readValue(it,Course::class.java) })
                .doOnNext(::println)
                .then()
    }.block()
}
```

```
AB12 called Intro to Scala duration 4 days
CD34 called JEE Web Development duration 3 days
EF56 called Meta-Programming in Ruby duration 2 days
GH78 called OO Design with UML duration 3 days
IJ90 called Database Access with JPA duration 3 days
KL12 called Design Patterns in C# duration 2 days
...
```


The Verdict…

Reactive is the right choice

- When processing huge volumes of information
- When you have to manage 1000s of connections

But is requires you have

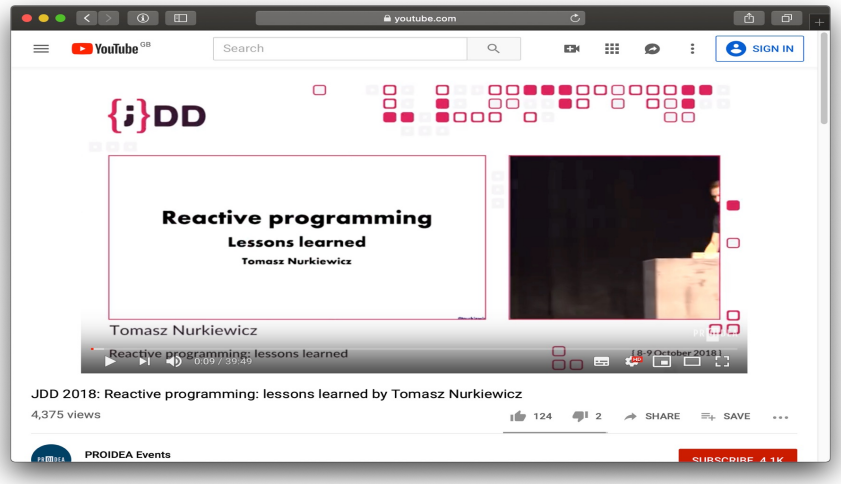
- Enough technical expertise
- A genuine business need

The dangers are

- Needlessly complex code
- Fragmented business logic
- Obscured problem domain

"Do I need it? Does it spark joy?"

A Great Overview of Issues



283

Standard Extensions

- Commonly Used Utility Functions

284

Introducing the Standard Extensions

Kotlin provides helper methods for use with objects

- They encapsulate common patterns of use

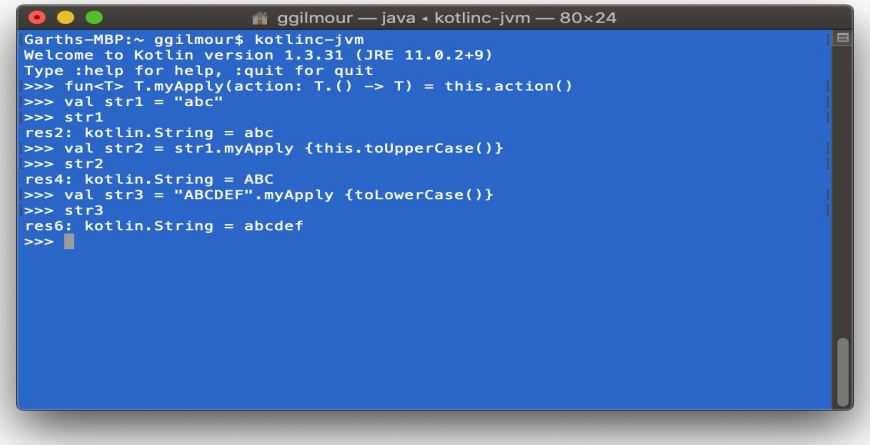
Some of these methods rely on a 'Lambda With Receiver'

- This is a lambda which has an object injected into the code block
- When calls are made within the lambda it is assumed that you wish them made against this object (the receiver)

We will examine this in detail later

- But note that '`(Int) -> String`' denotes a normal Lambda
- Whereas '`Date.(Int) -> String`' denotes a Lambda With Receiver (where the receiver object must be of type 'Date')

Introducing the Standard Extensions



Introducing the Standard Extensions

Let us take the original Java Calendar API as an example
- The flaws in its design are typical of issues in real libraries

Imagine that we need to:
- Create a Calendar object
- Invoke 'set' three times
  - To fix the date, month and year
- Optionally access the time property
  - Sometimes we will only need the Calendar once
  - Other times we will need to hang onto it for later

Each of the standard extensions could be applicable
- Depending on precisely what we want to do
- We will walk through the different options…

The Standard Extensions

```
fun main(args: Array<String>) {
    demoRun()
    demoApply()
    demoAlso()
    demoLet()
    demoTake()
}

fun printDate(msg: String, input: Date?) {
    print(msg)
    input?.let {
        val dateFormat = SimpleDateFormat("dd/MM/yyyy")
        print(dateFormat.format(input))
    }
    println()
}

fun calendar() = Calendar.getInstance()
```

```
Date via run 25/12/2020
Date via apply 25/12/2020
Date via also 25/12/2020
Date via let 25/12/2020
Date via takeIf
Date via takeUnless 25/12/2020
```

The ‘run’ Standard Extension

```
fun demoRun() {
    val result: Date = calendar().run {
        set(DAY_OF_MONTH, 25)
        set(MONTH, 11)
        set(YEAR, 2020)
        time
    }
    printDate("Date via run ", result)
}
```

The ‘run’ function takes the calendar as the receiver and returns the result of executing the block. This helps when you want to configure an object and use it once e.g. Builder objects.

`functions.standard.extensions.Program`

The ‘apply’ Standard Extension

```
fun demoApply() {
    val result = calendar().apply {
        set(DAY_OF_MONTH, 25)
        set(MONTH, 11)
        set(YEAR, 2020)
    }
    printDate("Date via apply ", result.time)
}
```

The ‘apply’ function takes the calendar as the receiver and also uses it as the return value. This helps when you need to perform additional calls on an object before first use.

`functions.standard.extensions.Program`

The ‘also’ Standard Extension

```
fun demoAlso() {
    var date: Date = Date()
    val calendar = calendar().apply {
        set(DAY_OF_MONTH, 25)
        set(MONTH, 11)
        set(YEAR, 2020)
    }.also { c -> date = c.time }

    printDate("Date via also ", date)
}
```

The ‘also’ function takes the calendar as the input to the block and also returns it. However there is no receiver. This enables you to access objects in the surrounding context via ‘this’.

`functions.standard.extensions.Program`

The ‘let’ Standard Extension

```
fun demoLet() {
    val date = calendar().let {
        it.set(DAY_OF_MONTH, 25)
        it.set(MONTH, 11)
        it.set(YEAR, 2020)
        it.time
    }
    printDate("Date via let ", date)
}
```

The ‘let’ function takes the calendar as the input to the block and returns the last line executed. Used to create a new scope and for conditional initialization.

`functions.standard.extensions.Program`

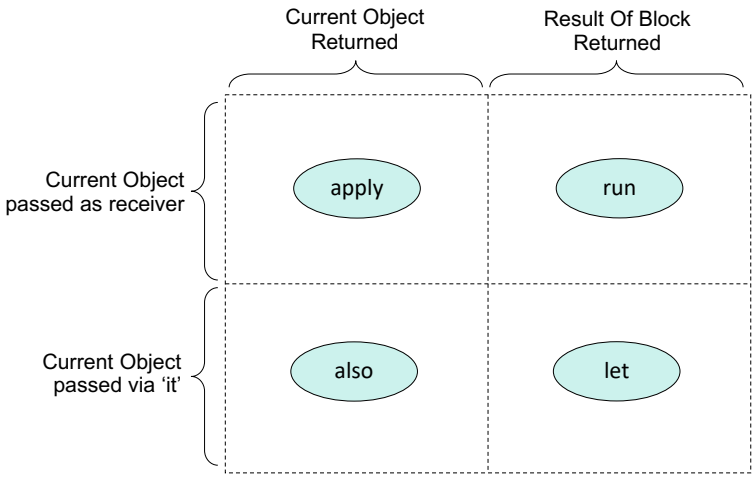
The ‘takeIf’ and ‘takeUnless’ Standard Extensions

```
fun demoTake() {
    val calendar1 = calendar().apply {
        set(DAY_OF_MONTH, 25)
        set(MONTH, 11)
        set(YEAR, 2020)
    }
    val calendar2 = calendar1
                    .takeIf { c -> c.after(Date()) }
    val calendar3 = calendar1
                    .takeUnless { c -> c.before(Date()) }

    printDate("Date via takeIf  ", calendar2?.time)
    printDate("Date via takeUnless ", calendar3?.time)
}
```

The ‘takeIf’ and ‘takeUnless’ functions act as filters, but against a single object rather than a list of items

`functions.standard.extensions.Program`

Summarizing the Standard Extensions



Summarizing the Standard Extensions

Name	Object Passed	Value Returned	Example of Use
run	As receiver	The result of the block	To use and then disregard a factory or builder object
apply	As receiver	The current object	To perform additional configuration on an object
also	As block input (‘it’)	The current object	To perform additional configuration using symbols from the enclosing scope
let	As block input (‘it’)	The result of the block	To create a new scope, perhaps as part of conditional execution

295

Creating DSL’s

- Using the Builder Pattern in Kotlin

296

Domain Specific Languages in Kotlin

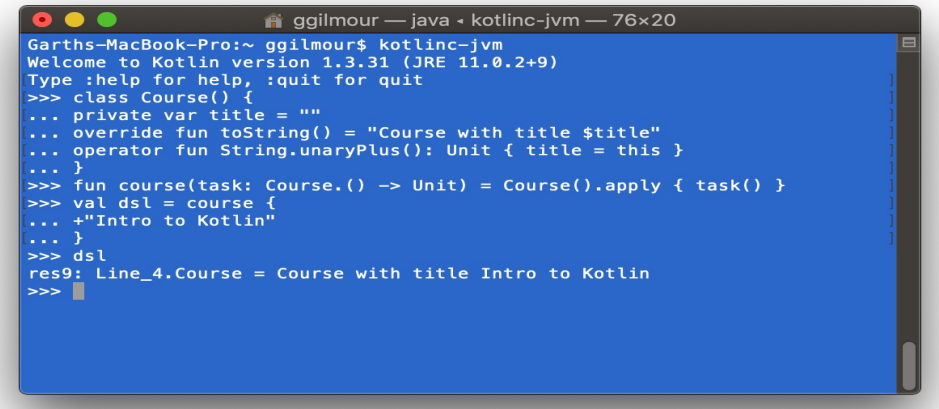
A DSL is a 'language within a language'
- Often used with configuration and builder patterns

Groovy has typically been used for DSL's running on the JVM
- Gradle, Grails and Spring can all be configured in this way
- However dynamic typing can raise errors later than desired

Kotlin is flexible enough to create DSL's since:
- We can omit the parenthesis on a method call
- Arbitrary code can be passed in a code block
- Lambdas can be given a 'receiver object'

297

Domain Specific Languages in Kotlin



298

A DSL for Describing Course Outlines

The following slide shows a DSL for describing course outlines

- We want a simple type-safe syntax that can be saved as XML

299

A DSL for Describing Course Outlines

```
val course = course("AB12") {
    description(duration = 4) {
        !"Kotlin Programming"
    }
    modules {
        module {
            !"Object Orientation"
            +"Classes"
            +"Interfaces"
            +"Methods"
        }
        module {
            !"Functional Programming"
            +"Lambdas"
            +"Reductions”
            +"Currying"
        }
        module {
            !"Java Interoperability"
            +"Kotlin to Java"
            +"Java to Kotlin"
        }
    }
}
```

```
<course id="AB12">
  <description type="Beginner"
               duration="4">
    <title>Kotlin Programming</title>
  </description>
  <moduleList>
    <module>
      <title>Object Orientation</title>
      <item>Classes</item>
      <item>Interfaces</item>
      <item>Methods</item>
    </module>
    <module>
      <title>Functional Programming</title>
      <item>Lambdas</item>
      <item>Reductions</item>
      <item>Currying</item>
    </module>
    <module>
      <title>Java Interoperability</title>
      <item>Kotlin to Java</item>
      <item>Java to Kotlin</item>
    </module>
  </moduleList>
</course>
```

domain.specific.languages

300

A DSL for Describing Course Outlines

We create a hierarchy of 'Node' types with 'render' methods

We also create a series of builder methods (e.g. 'course')

The builder methods each take a Lambda With Receiver

- E.g. `fun course(id: String, inputFunc: Course.() -> Unit)`

Within the builder functions we:

- Create a new node of the specified type (e.g. 'Course')
- Execute the supplied lambda with that node as receiver
- Add the node to the document under construction

This process recursively build the document

- Since the lambda will itself contain calls to builder functions

A DSL for Describing Course Outlines

```
enum class CourseType {
    Beginner, Intermediate, Advanced
}

interface Node {
    fun render(indent: String = ""): String
}

class Attribute(val name: String, val value: Any) : Node {
    override fun render(indent: String) = "$name=\"$value\""
    override fun toString() = render()
}
```

A base interface to represent nodes

A derived type for Attributes

`domain.specific.languages`

A DSL for Describing Course Outlines

```
abstract class Element(private val tagName: String) : Node {
    val attributes = mutableListOf<Attribute>()

    protected val openTag: String
        get() {
            fun spacer() = if (attributes.isEmpty()) "" else " "
            return attributes.joinToString(
                        prefix = "<$tagName${spacer()}",
                        separator = " ",
                        postfix = ">")
        }

    protected val closeTag: String
        get() = "</$tagName>"

    override fun toString() = render()
}
```

An abstract base class for elements that correctly renders the opening and closing tags

domain.specific.languages

INSTIL

303

A DSL for Describing Course Outlines

```
open class ParentElement(tagName: String) : Element(tagName) {
    private val content = mutableListOf<Node>()

    operator fun String.not() = content.add(Title(this))
    operator fun String.unaryPlus() = content.add(Item(this))

    open fun<T : Node> builder(obj: T,
                               inputFunc: T.() -> Unit,
                               attrs: List<Attribute> = emptyList()) {
        content.add(obj)
        attributes.addAll(0, attrs)
        obj.inputFunc()
    }

    override fun render(indent: String): String {
        val builder = StringBuilder()
        builder.append("$indent$openTag\n")
        content.forEach { builder.append(it.render(indent.plus("  "))) }
        builder.append("$indent$closeTag\n")
        return builder.toString()
    }
}
```

Overloading the ! and + operators for the string type, in order to add titles and items to the document.

A generic implementation of the Builder pattern via a Lambda With Receiver

domain.specific.languages

INSTIL

304

A DSL for Describing Course Outlines

```
class Description : ParentElement("description")

class Module : ParentElement("module")

open class LeafElement(tagName: String,
                       private val value: String)
                       : Element(tagName) {

    override fun render(indent: String): String {
        return "$indent$openTag$value$closeTag\n"
    }
}

class Title(value: String) : LeafElement("title", value)

class Item(value: String) : LeafElement("item", value)
```

Most of the types in the document have simple definitions.

domain.specific.languages

A DSL for Describing Course Outlines

```
class Course : ParentElement("course") {
    fun description(type: CourseType = CourseType.Beginner,
                    duration: Int = 3,
                    func: Description.() -> Unit) {
        val description = Description()
        with(description.attributes) {
            add(Attribute("type", type))
            add(Attribute("duration", duration))
        }
        builder(description, func)
    }

    fun modules(func: ModuleList.() -> Unit)
            = builder(ModuleList(), func)
}
```

The Course type needs to contain builder methods for the description and modules sections.

domain.specific.languages

A DSL for Describing Course Outlines

```
class ModuleList : ParentElement("moduleList") {
    fun module(func: Module.() -> Unit) = builder(Module(), func)
}

fun course(id: String,
           inputFunc: Course.() -> Unit) = Course().apply {
    attributes.add(Attribute("id", id))
    inputFunc()
}
```

'course' is the initial builder function. It builds and returns a new Course object, having first added an id attribute and run a Lambda With Receiver. This can all be done very conveniently via the 'apply' standard extension.

`domain.specific.languages`

INSTIL

307

Exercise

- Create a DSL for building HTML
  Add some tags - body, h1, div, p
  Add some attributes – hidden, style, class (is there any issue with this last one?)

INSTIL

308

Interoperability

- Part 1 - Kotlin and Java

INSTIL

309

Interoperability with Java

Interoperability between JVM languages is straightforward

- At least when compared to calling outside of the JVM (such as via JNI) or then combining non-VM based languages

Kotlin was designed to be fully interoperable with Java

- You can make calls in either direction without much fuss
- Calling into Kotlin is inevitably the more complex route
  - Since Kotlin must generate code for features (like properties) which have no built in JVM support

Mixed coding is a great way to experiment

- You can incrementally introduce Kotlin into your process
- E.g. you could write test suites and utilities in Kotlin and move on to production code once your had gained confidence

INSTIL

310

Calling Java from Kotlin

Java properties can be used directly from Kotlin

- Instead of 'emp.setAge(27)' we can say 'emp.age = 27'
- The Java properties must follow the JavaBean conventions
- Setters without getters are not exposed as properties
  - Because Kotlin doesn't support write-only properties

Kotlin keywords may occur in the Java code you're using

- Such as the 'in' member of the 'System' class
- These can be replaced using backtick quotes
  - E.g. 'val scanner = Scanner(System.`in`)'

311

Calling Java from Kotlin

Null checks cannot be performed on Java code

- When a Java method returns a reference the Kotlin compiler has no way to check whether or not it could be null

Types returned by Java methods are called 'platform types'

- The normal rules for null checking are not applied to these types, meaning you have no more safety than regular Java

There is no syntax for declaring these types in Kotlin

- But there are a set of conventions for denoting them in docs
- The syntax 'MyType!' denotes a type called 'MyType' which may or may not be nullable (both 'MyType' and 'MyType?')
- '(Mutable)Collection<MyType>!' demotes a collection of 'MyType' which might be mutable and/or nullable

312

Calling Java From Kotlin

```
public class AwkwardJava {
    private String suggestion;
    private boolean awkward;

    public String getSuggestion() {
        return suggestion;
    }
    public void setSuggestion(String suggestion) {
        this.suggestion = suggestion;
    }
    public boolean isAwkward() {
        return awkward;
    }
    public void setAwkward(boolean awkward) {
        this.awkward = awkward;
    }
    public void object() {
        if(awkward) {
            System.out.printf("Your idea to '%s' is refused!\n", suggestion);
        } else {
            System.out.printf("Sure, you can '%s' if you like\n", suggestion);
        }
    }
}
```

Whoops! - **suggestion** might be null

Whoops! - **object** is a Kotlin keyword

AwkwardJava

313

Calling Java From Kotlin

```
fun main(args: Array<String>) {
    val obj = AwkwardJava()

    val defaultIdea = obj.suggestion
    try {
        defaultIdea.toUpperCase()
    } catch(ex: TypeCastException) {
        println("Whoops...");
    }

    obj.suggestion = "go surfing"
    obj.`object`()

    obj.isAwkward = true
    obj.`object`()
}
```

```
Whoops...
Sure, you can 'go surfing' if you like
Your idea to 'go surfing' is refused!
```

Program

314

Calling Java From Kotlin

Checked exceptions are ignored in Kotlin

- If the compiler detects you are calling a Java method with a 'throws' clause it will not enforce any error handling behaviour

It is easy to obtain class objects

- You can obtain the instance of 'java.lang.Class' for a type via the syntax 'MyType::class.java' or 'myInstance.javaClass'
- This is important when combining popular Java frameworks like JUnit, Mockito and Spring with a Kotlin codebase

Void functions are considered as returning the 'Unit' type

- If you assign a variable to the return value of a void function the Kotlin compiler will set the variable to refer to 'Unit'

315

Calling Java From Kotlin

Kotlin and Java treat arrays differently

- Arrays are invariant in Kotlin but not in Java

Consider the code 'Object [] foo = new String[10]'

- This will compile in Java and might cause runtime errors
- The equivalent Kotlin would not simply not compile

Kotlin does not require invariance when calling Java

- So it is possible to pass an 'Array<MyType>' to a Java method which has a parameter of type 'MyType []'
- The platform type for a a Java array is 'Array<(out) T>!'
  - So 'Array<(out) MyType>!' means a Java array of type 'MyType', or subtype of 'MyType' which may be nullable

316

Calling Java from Kotlin

```
fun main(args: Array<String>) {
    val javaUser = JavaArrayUser()
    val kotlinUser = KotlinArrayUser()
    val testInput = arrayOf(Employee(), Employee())

    javaUser.useArray(testInput)
    kotlinUser.useArray(testInput) //this will not compile
}
public class JavaArrayUser {
    public void useArray(Object[] input) {
    }
}
class KotlinArrayUser {
    fun useArray(input: Array<Any> ) {}
}
public class Employee {
}
```

Program

Calling Kotlin From Java

Hopefully this will not be a regular occurrence

- As you will be moving forwards in your technology stack ☺

A special case is using JEE frameworks

- However today there are mostly annotation driven
- See our discussion of Spring in the next topic

Calling Kotlin from Java requires knowledge of:

- The conventions the Kotlin compiler uses for code generation
- Annotations that can be used to make your Kotlin code more accessible to clients written and compiled in Java

Calling Kotlin From Java

As we saw in the OO topic, a Kotlin property consists of:

- An accessor method, which is given the prefix ‘get’
- For ‘var’ properties an accessor method with the prefix ‘set’
- A private backing field as required, named after the property

Functions in a file are placed within a generated class

- So all functions within ‘Utils.kt’ in package ‘foo.bar’ would end up a regular Java methods of class ‘foo.bar.UtilsKt’
- ‘@JvmName’ can be used to change the file name
  - When applied with ‘@JvmMultifileClass’ this also allows functions from multiple files to be combined into one type

Calling Kotlin From Java

Static members will be created for:

- Properties in named and companion objects
- Properties in normal types marked with ‘const’

Default parameter values are handled via a single method

- If a function takes three parameters with default values then the Kotlin compiler will insert values as needed at the call site
- By default it does not overload the function three times
- You can alter this behaviour via ‘@JvmOverloads’

Calling Kotlin From Java

```kotlin
@file:JvmName("MyStuff")
package interop.tokotlin

fun awkward() {
    println("Function 'awkward' called")
}
class AwkwardKotlin {
    var suggestion: String = "eat fish"

    companion object Extras {
        fun makeSuggestion(k: AwkwardKotlin) {
            println("We recommend you " + k.suggestion)
        }
    }
    fun foo(a: String = "Fred", b:String = "Wilma") {
        println("Fun foo called with $a and $b")
    }
    @JvmOverloads
    fun bar(a: String = "Homer", b:String = "Marge") {
        println("Fun bar called with $a and $b")
    }
}
```

How will we access functions?

How will we access properties?

What does this become?

How many versions of 'foo' will be created?

How many versions of 'bar' will be created?

AwkwardFile.kt

Calling Kotlin from Java

```java
public class Program {
    public static void main(String[] args) {
        MyStuff.awkward();

        AwkwardKotlin ak1 = new AwkwardKotlin();
        AwkwardKotlin ak2 = new AwkwardKotlin();
        ak2.setSuggestion("drink milk");
        AwkwardKotlin.Extras.makeSuggestion(ak1);
        AwkwardKotlin.Extras.makeSuggestion(ak2);

        ak1.foo("Peter", "Lois");
        ak1.bar();
        ak1.bar("Bart");
        ak1.bar("Bart", "Liza");
    }
}
```

```
Function 'awkward' called
We recommend you eat fish
We recommend you drink milk
Fun foo called with Peter and Lois
Fun bar called with Homer and Marge
Fun bar called with Bart and Marge
Fun bar called with Bart and Liza
```

Program.java

Additional Compile Time Annotations

Kotlin provides many more compile time annotations

- Which can be used to shape the generated bytecode
- A summary table is included on the next slide

Most projects will not need this level of control

- Since typically we are developing (or rewriting) a set of microservices or an Aandroid client app in its entirety

However they would be essential if you were

- Incrementally rewriting a Java application one file at a time
- Replacing a Java library with complete fidelity to the original

323

Additional Compile Time Annotations

Annotation	Description
@JvmOverloads	Generate multiple versions of a method so that default parameter values can be used by Java clients
@Throws	Add a throws clause to a Kotlin function so that try … catch blocks in the Java client code will still compile
JvmName @file:JvmName @get:JvmName	Rename a method, a file or a getter in the output so that the final symbol follows Java coding conventions
@JvmMultifileClass	Used with above to generate multiple outputs from single file
@JvmField	Makes the backing field accessible in the output
@JvmWildcard @JvmSuppressWildcards	Adds or removes wildcards in Generic code (e.g. void foobar(List<? extends Person> zed))

324

Interoperability

- Part 2 - Reflection and Code Generation

INSTIL

325

Reflection in Kotlin

Standard Java Reflection works fine with Kotlin

- We can acquire Class objects via the 'javaClass' property
- Operations to retrieve fields and methods work as usual
  - But note the naming conventions discussed previously

Kotlin also has its own reflection library

- On the JVM this ships as a separate library (*kotlin-reflect.jar*)
- To decrease the size of the runtime for those that don't need it
  - E.g. Android projects that do DI at compile time via Dagger

The syntax 'Foobar::class' returns a 'KClass' instance

- This has methods for querying Kotlin specific functionality
- E.g. is the current type a Data class or a Companion object?

INSTIL

326

Reflection in Kotlin

```
fun main() {
    val emp = Employee("Dave", 32, 45000.0);
    showJavaReflection(emp.javaClass)
    showKotlinReflection(emp::class)
}

fun printTabbed(input: Any) = println("\t$input")
```

Program.kt

Reflection in Kotlin

```
fun showJavaReflection(javaClass: Class<*>) {
    println("\n--- Java Reflection ---")
    println("Name is '${javaClass.simpleName}'")
    println("Base type is '${javaClass.superclass.simpleName}'")
    println("Methods are:")
    javaClass.declaredMethods
        .map { it.name }
        .forEach(::printTabbed)
    println("Fields are:")
    javaClass.declaredFields
        .map { it.name }
        .forEach(::printTabbed)
}
```

```
--- Java Reflection ---
Name is 'Employee'
Base type is 'Person'
Methods are:
    toString
    getSalary
Fields are:
    salary
```

Program.kt

Reflection in Kotlin

```
fun showKotlinReflection(kotlinClass: KClass<*>) {
    fun isDataClass() = if(kotlinClass.isData) "is" else "is not"
    fun baseName() = kotlinClass.allSuperclasses.first().simpleName

    println("\n--- Kotlin Reflection ---")
    println("Name is '${kotlinClass.simpleName}'")
    println("Base type is '${baseName()}'")
    println("This ${isDataClass()} a data class")
    println("Properties are:")
    kotlinClass.members
        .map { it.name }
        .forEach(::printTabbed)
}
```

```
--- Kotlin Reflection ---
Name is 'Employee'
Base type is 'Person'
This is not a data class
Properties are:
    salary
    toString
    age
    name
    equals
    hashCode
```

Program.kt

INSTIL

329

Annotation Processing with Kapt

Annotation Processors add or modify your compiler output

- Usually based on the annotations decorating your source

They automate the generation of boilerplate code

- Constructors and properties in Java (e.g. Project Lombok)
- Support for Optics in Functional Coding (e.g. Arrow in Kotlin)

INSTIL

330

Annotation Processing with Kapt

'Kapt' is the Kotlin compiler plugin that supports JSR 269

- It will run regular Java Annotation Processors (such as in Dagger) but also supports Kotlin specific processors

Typically Kapt is executed via Gradle

- You can configure Gradle only to run it when necessary
- In order to speed up the time taken to compile your code

Code Generation with KotlinPoet

KotlinPoet is a library for generating Kotlin course code

- It is inspired by (and very similar to) the JavaPoet library

Useful features of KotlinPoet are:

- Import declarations are automatically added as required
- Statements can be directly specified as multi-line strings
- Lots of assistance is given for generating template strings
- Enums, Anonymous Inner Classes etc. are all supported

Creating an Employee Class with KotlinPoet

```
fun main() {
    FileSpec.builder("", "KotlinPoetDemo")
        .addType(employeeType())
        .build()
        .writeTo(System.out)
}

private fun employeeType(): TypeSpec {
    return TypeSpec.classBuilder("Employee")
        .addProperty(employeeProperty("name", String::class))
        .addProperty(employeeProperty("age", Int::class))
        .addProperty(employeeProperty("salary", Double::class, true))
        .primaryConstructor(employeeConstructor())
        .addFunction(awardBonusFun())
        .addFunction(toStringFun())
        .build()
}
```

Program.kt

INSTIL

333

Creating an Employee Class with KotlinPoet

```
private fun employeeProperty(name: String,
                             type: KClass<*>,
                             isMutable: Boolean = false): PropertySpec {
    val spec = PropertySpec.builder(name, type).initializer(name)
    if(isMutable) {
        spec.mutable()
    }
    return spec.build()
}

private fun employeeConstructor(): FunSpec {
    val msg = "Created \$name";
    return FunSpec.constructorBuilder()
        .addParameter("name", String::class)
        .addParameter("age", Int::class)
        .addParameter("salary", Double::class)
        .addStatement("println(%P)", msg)
        .build()
}
```

Program.kt

INSTIL

334

Creating an Employee Class with KotlinPoet

```
private fun awardBonusFun(): FunSpec {
    return FunSpec.builder("awardBonus")
        .addCode(
            """
                    |if (age > 40) {
                    |    salary += 5000.0
                    |} else {
                    |    salary += 3000.0
                    |}
                    |""".trimMargin())
        .build()
}
```

Program.kt

INSTIL

335

Creating an Employee Class with KotlinPoet

```
private fun toStringFun(): FunSpec {
    val msg = "\$name of age \$age earning \$salary"
    return FunSpec.builder("toString")
        .addModifiers(KModifier.OVERRIDE)
        .addStatement("return %P", msg)
        .returns(String::class)
        .build()
}
```

Program.kt

INSTIL

336

Creating an Employee Class with KotlinPoet

```
class Employee(
  val name: String,
  val age: Int,
  var salary: Double
) {
  init {
    println("""Created $name""")
  }
  fun awardBonus() {
    if (age > 40) {
       salary += 5000.0
    } else {
       salary += 3000.0
    }
  }

  override fun toString(): String = """$name of age $age earning $salary"""
}
```

INSTIL

337

Coroutines

- Understanding suspending functions

INSTIL

338

What is a Coroutine?

So what is a coroutine?

I know coroutines can be used for asynchronous programming

So why do we even want to do asynchronous programming?

How do coroutines help?

339

Synchronous is Easy

```
fun doWork() {
    disableUI()
    val exchangeRates = readExchangeRate()
    val report = buildReport(exchangeRates)
    saveReport(report)
    enableUI()
} return
```


340

Why do we need asynchronous?

```
fun doWork() {
1    disableUI()
2    val exchangeRates = readExchangeRate()
3    val report = buildReport(exchangeRates)
4    saveReport(report)
5    enableUI()
return }
```

What if the highlighted lines take a long?

The thread is waiting a long time for I/O

- Essentially idle

341

Why do we need asynchronous?

```
fun doWork() {
1    disableUI()
2    val exchangeRates = readExchangeRate()
3    val report = buildReport(exchangeRates)
4    saveReport(report)
5    enableUI()
return }
```

Typically we can't block due to limited resources

So we can't escape doing asynchronous programming

But we want ways to hide the complexity of async

342

Solution 1 – Callback Hell

```
fun doWork() {
    disableUI()
return readExchangeRate { exchangeRate ->
        buildReport(exchangeRate) { report ->
            saveReport(report) {
                enableUI()
            }
        }
    }
}
```


Solution 2 – Abstraction

```
fun doWork() {
    disableUI()
return readExchangeRate()
        .thenCompose(::buildReport)
        .thenCompose(::saveReport)
        .thenApply { enableUI() }
}
```

- Many platforms have APIs that abstract the callbacks
  - e.g. Futures, Promises, Tasks etc.
- This helps, but the APIs can be large, confusing and introduce noise

Solution 3 - Coroutines

```
suspend fun doWork() {
1    disableUI()
2    val exchangeRates = readExchangeRate()
3    val report = buildReport(exchangeRates)
4    saveReport(report)
5    enableUI()
return }
```

INSTIL

345

Solution 3 - Coroutines

```
suspend fun doWork() {
    disableUI()
suspend  val exchangeRates = readExchangeRate()
suspend  val report = buildReport(exchangeRates)
suspend  saveReport(report)
    enableUI()
return }

fun doWork() {
    disableUI()
    val exchangeRates = readExchangeRate()
    val report = buildReport(exchangeRates)
    saveReport(report)
    enableUI()
}
```

INSTIL

346

What is happening on the thread?

So, coroutines allow us to suspend a function

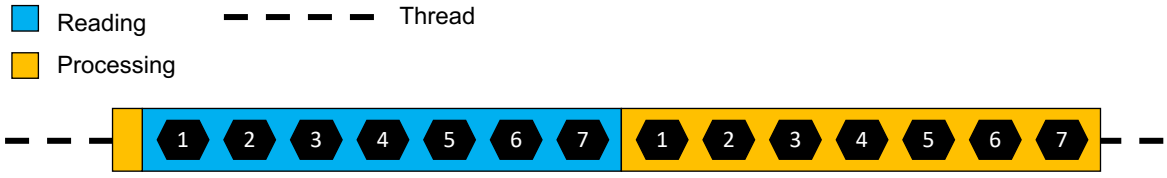
This frees the executing thread to go off and do other (useful) work

- Better resource (thread) utilisation
- Simpler code to alternatives methods

So what is actually happening on the thread?

To figure that out lets look at a non-asynchronous use case

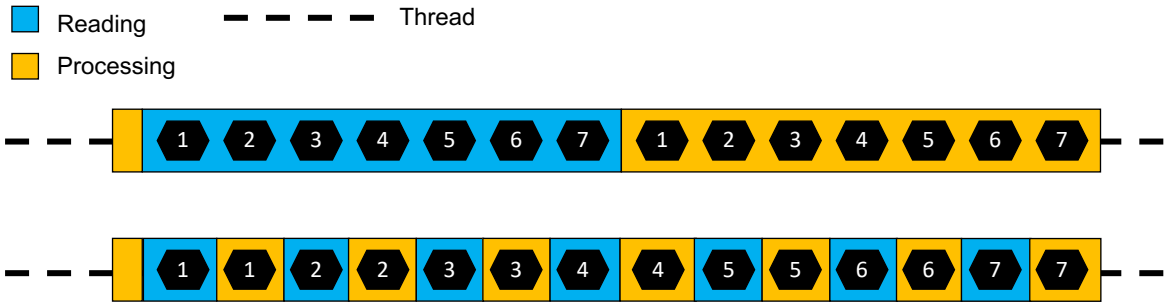
Generators - Another Use Case




```
fun processData(filename: String) {
    for (datum in readData(filename)) {
        log("Processing data $datum")
    }
}
```

```
fun readData(filename: String): List<Int> {
    val data = mutableListOf<Int>()
    File(filename).useLines {lines ->
        lines.forEach {
            val datum = it.toInt()
            log("Read $datum")
            data.add(datum)
        }
    }

    return data
}
```

Generator - Another Use Case



We may need to interleave reading and processing

How should we implement this?

- Perhaps hand code the interleaving?
- Perhaps use standard interfaces to simplify?
- Perhaps coroutines can help?

Synchronous vs Coroutine Generator

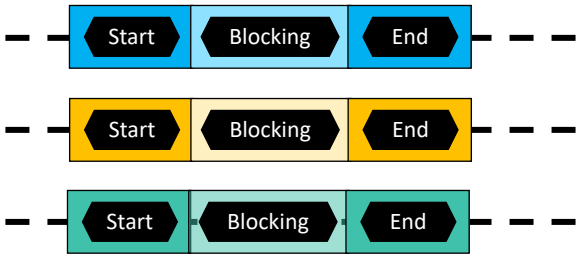
```
fun readData(filename: String): List<Int> {
    val data = mutableListOf<Int>()
    File(filename).useLines {lines ->
        lines.forEach {
            val datum = it.toInt()
            log("Read $datum")
            data.add(datum)
        }
    }
    return data
}

fun processData(filename: String) {
    for (datum in readData(filename)) {
        log("Processing data $datum")
    }
}
```

```
fun readData(filename: String): Sequence<Int> {
    return sequence {
        File(filename).useLines {lines ->
            lines.forEach {
                val datum = it.toInt()
                log("Read $datum")
                yield(datum)
            }
        }
    }
}

fun processData(filename: String) {
    for (datum in readData(filename)) {
        log("Processing data $datum")
    }
}
```

Event Loop with Asynchronous Blocks

Consider 3 independent jobs of work

If we want to run these in parallel we could create multiple threads

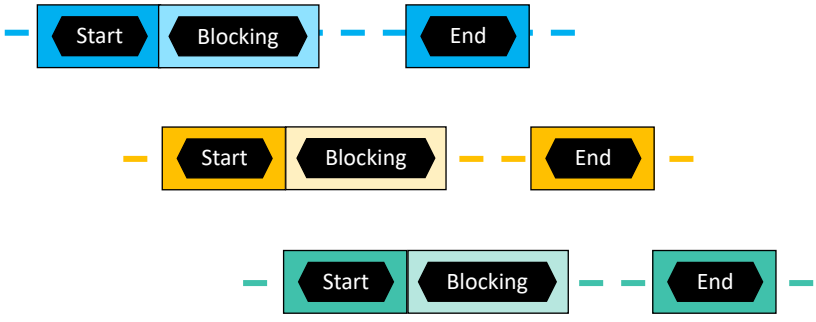


INSTIL

351

Or as Coroutines

Less Threads == Better Resource Utilisation

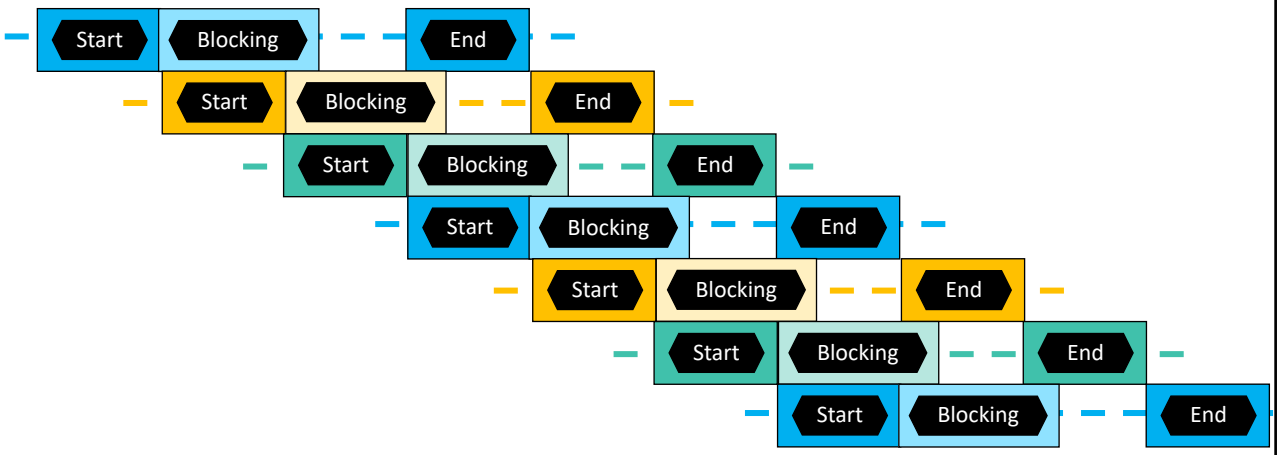


INSTIL

352

Coroutines Scale

Less Threads == Better Resource Utilisation



INSTIL

353

Benefits of Coroutines

They do not help us write faster code

- Although they can help us write more efficient systems
- This may make the whole system more responsive

They are **easier** to write, as they are very similar to standard code

We get to use all the standard coding elements we are used to

- if, for, while
- try-catch
- function calls
- Etc

INSTIL

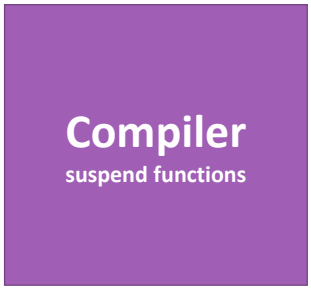
354

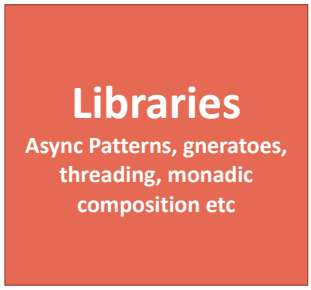
Separation of Responsibility

Some languages like C# & JavaScript have specific async/await keywords

Kotlin has went a different way a produced a much more general solution

The only keyword in the language is suspend – libraries handle the rest





355

Coroutine Builders

The kotlinx.coroutines library adds the API entitles for useful coroutines

There are a number of modules we may want to use:

- `kotlinx-coroutines-core` - Core primitives to work with coroutines
- `kotlinx-coroutines-test` - Test primitives for coroutines
- `kotlinx-coroutines-android` – Support for Android applications
- And more for Swing, JavaFX and others

356

Coroutine Builders

It introduces some primitives and terminology:

Scope – Encloses a coroutines and its child coroutines

Context – A key/value collection of values to control execution

Dispatcher – Controls coroutine execution e.g. which threads to use

Builders – functions to create a coroutine

Interceptor – Wraps/intercepts coroutine execution

Bridge – A builder that can be called from a non-suspend function

Coroutine Builders

These are functions that are supplied a suspend function/lambda and creates a coroutine

Builder	Description
`launch`	Returns a **`Job`** object representing the coroutine but is largely fire and forget. It is synchronised with another coroutine using join.
`async`	Returns a **`Deferred<T>`** object which is an extension of Job that includes a result. The result is acquired using await.
`runBlocking`	Runs a coroutine and blocks the current thread until it completes
`sequence`	Creates generators using a suspend function

Coroutine Dispatchers

Controls which thread is used to execute a continuation

- A scope with its context will define a dispatcher

These will be platform dependent but exists in `Dispatchers` object

Dispatcher	Description
`Default`	Executes on a default background thread pool
`Unconfined`	Executes using any available thread
`Main`	Executes on the Main (UI) thread
`IO`	Executes blocking I/O calls. Shares threads with Default but will create and shutdown additional threads

Suspending Functions

These suspending functions execute against a Coroutine Scope

- They are **extension methods** of **CoroutineScope**

Builder	Description
`delay`	Executes a non-blocking sleep
`yield`	Yields the thread in single thread disaptchers
`withContext`	Like runBlocking within a coroutine i.e. it suspends
`withTimeout`	Sets a time limit on execution of a function
`await, awaitAll`	Waits for one or multiple Deferred to complete and returns the result/s
`join, joinAll`	Waits for one or multiple jobs to complete

Using Coroutines with Dispatchers

```
suspend fun doWorkWithThreadControl() {
1       disableUI()
        withContext(Dispatchers.IO) {
2           val exchangeRates = readExchangeRate()
3           val report = buildReport(exchangeRates)
4           saveReport(report)
        }
5       enableUI()
return }
```

- The inner lines execute on a different thread
- The coroutine library maintains the order and handles marshalling

INSTIL

361