

# Enterprise Programming 2

## Lesson 01: Introduction

Prof. Andrea Arcuri

# Enterprise 2

- also known as the “*Next Generation*”
  - sorry, couldn't stop myself...



# Course Info

- 12 lessons, once a week
- Check TimeEdit for possible changes of time and rooms
- During the course, do **NOT** send me private messages, but rather use the discussion forum of the course

# Class Structure

- “Usually” 2+2
  - 2 hours of lecture: code (and very few slides...)
  - 2 hours in which you should do exercises and get help
- **IMPORTANT:** the 2 hours after lecture is not only for exercises. If you are falling behind, or you need some more revision, you can ask for my help on anything related to coding

# Necessary Tools

- JDK 8
- Git
- Maven
- YARN and NodeJS
- An IDE (I **strongly** recommend IntelliJ IDEA Ultimate Edition)
- **Docker**
- A Bash command-line terminal
  - Mac/Linux: use the built-in one
  - Windows: I recommend GitBash

# Git Repository

- [https://github.com/arcuri82/testing\\_security\\_development\\_enterprise\\_systems](https://github.com/arcuri82/testing_security_development_enterprise_systems)
- Same as PG5100, but now look at the “**advanced**” folder
- Note: pull often, as new material might be added and updated during the course

# Goals/Topics

- Full details of **RESTful** APIs and **HTTP**
- Knowledge of other kinds of Web Services: **SOAP** and **GraphQL**
- **Microservice** Architecture
  - gateways, load balancers, etc.
- **Security** in distributed systems
- Message Oriented Middleware (e.g., AMQP)

# If You Skip Class...

- Usually acceptable that a student skips 1-2 classes
- You are supposed to attend, although no strict checks
- If you skip too many classes, it is **YOUR** responsibility to catch up and find out what done in class



# Exam

- Written exam (40% of grade)
  - 3 hours
  - Theory questions
  - Not required to write code, but might be asked about what some code snippets do
- Project (60% of grade)
  - 72 hours
  - done individually, not in a group

Kotlin

# About Kotlin

- Recent language: 2011
  - Java is from 1995
- Compile to JVM bytecode (and JavaScript)
- High compatibility with Java
  - Can reuse all tools (eg Maven) and libraries (eg Spring)
- Made by JetBrains (same as IntelliJ)
- As of May'17, Kotlin became an official language for Android development

# Kotlin Island (St. Petersburg)



# Why?

- Java is a good, solid language, but is verbose and lacks many “modern” features, eg when compared to C#
  - Things got bit better with Java 8, but that’s 2014
  - Java still better than C# for enterprise development, but mainly due to its ecosystem (frameworks and libraries)
- Due to Google vs Oracle legal fight, Android development was stagnating in a Java 6 *wasteland*
  - Java 6 is from **2006**, eons in the software development world...
- Goal: provide a modern language that can be 100% interoperable with Java

# Main Features

- Null safety:
  - **Compiler** does check if a call to “foo.bar()” might have “foo” null
  - If a variable can contain null, it has to be marked so
- **No F\*KCING Checked Exceptions...**
- Removed a lot of *boilerplate*... code much shorter

```
public class JavaBase {  
  
    public boolean startWithFoo(String s) {  
        if (s == null) {  
            return false;  
        }  
  
        String foo = "foo";  
  
        return s.startsWith(foo);  
    }  
}
```

```
//"public" is default scope
```

```
class KotlinBase {
```

```
    // "public" is default scope
```

```
    fun startWithFoo(s: String)
```

```
        : Boolean    //return type is specified at the end after ":"
```

```
{
```

```
    val foo = "foo"    //no need for ";" at the end
```

```
    //type is implicit at compilation time, but you can specify it
```

```
    //if you want, eg:
```

```
    //    val foo: String = "foo"
```

```
    /*
```

```
        Do not need to worry of "s.startsWith" throwing a NPE,  
        because compiler checks that caller of "startWithFoo"  
        is not null
```

```
    */
```

```
    return s.startsWith(foo)
```

```
}
```

```
}
```



# Types

- Specified after with “:”
- Can be left unspecified if compiler can infer them
  - `val foo = “foo”`
- Note: Kotlin **IS** statically typed

# Var/Val

- “var” is for variables that can be modified
- “val” are values which are constant
  - equivalent to the use of “final” in Java

```
fun bar () {  
  
    var foo = "foo"  
    foo = "changed"  
    //foo = null // doesn't compile  
  
    //note the "?" after the type  
    var bar : String? = "foo"  
    bar = null  
  
}
```

*// note the ?*

```
fun startWithFoo(s: String?) : Boolean {
```

*//doesn't compile*

*//return s.startsWith("foo")*

*//elvis operator*

```
return s?.startsWith("foo") ?: false
```

```
}
```

```
fun fiveNextIsFoo(link : Link?) : Boolean{  
  
    return link?.next  
        ?.next  
        ?.next  
        ?.next  
        ?.next  
        ?.s?.equals("foo")  
    ?: false  
  
}
```

```
public boolean fiveNextIsFoo(Link link) {  
    if(link == null ||  
        link.next == null ||  
        link.next.next == null ||  
        link.next.next.next == null ||  
        link.next.next.next.next == null ||  
        link.next.next.next.next.next == null ||  
        link.next.next.next.next.next.s == null) {  
        return false;  
    }  
  
    //all checks above are necessary to guarantee this  
    //instruction does not throw a NPE  
    return link.next.next.next.next.next.s.equals("foo");  
}
```

```
public boolean fiveNextIsFooWithCatch(Link link) {  
  
    try {  
        return link.next.next.next.next.next.s.equals("foo");  
    } catch (NullPointerException e) {  
        //this is more expensive, as exceptions need  
        //to fill info from stacktrace  
        return false;  
    }  
}
```

```
fun aboutStrings () {  
  
    val multilineString = """  
        <foo>  
            Some message  
        </foo>  
    """  
  
    val x = 5  
    val s = "Use \$ to interpolate, eg x=$x"  
    print(s)  
    //The print(s) does output the following:  
    //Use $ to interpolate, eg x=5  
}
```



```
class KotlinConstructor(val s: String, var x: Int) {  
  
    fun foo() = s + x  
  
}
```

```
public class JavaConstructor {

    private final String s;
    private int x;

    public JavaConstructor(String s, int x) {
        this.s = s;
        this.x = x;
    }

    public String foo() {
        return s + x;
    }

    public String getS() {
        return s;
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }
}
```

# Properties

```
class KotlinProperty(  
    val x: Int  
) {  
    var y: Int = 0  
    private set  
  
    var z: Int  
        get() { return y }  
        set(value) { y = value }  
}
```

- Automatically create getters and setters for the fields
- Can override the default behaviors of those getters/setters
- In client code, do not need to write *foo.getY()*, but just *foo.y*

# Functional Programming

- Kotlin is not as good for FP as Scala, but provides more abstractions/utilities compared to Java
- All objects have the methods: **let**, **apply**, **run**, **also**
- Useful when using streams or trying to avoid creating local variables

# let, apply, run, also

- They are functions that take a lambda as input
  - Note: in Kotlin, when input is a single lambda, no need for “()”
- Return a value: caller itself, or result of the lambda expression
- The meaning of “*this*” and “*it*” inside the lambda will vary based on the function

## Return Caller

## Return Lambda Result

# Caller as “it”

**also**

let

## Caller as “this”

**apply**

run

```
fun createFoo() : Foo{  
  
    val foo = Foo()  
    foo.initialize()  
    foo.doSomething()  
    return foo  
}
```

```
fun createFooWithFP() : Foo {  
  
    return Foo().apply { initialize(); doSomething() }  
}
```

```
fun getHttpBodyBlockNoFP(message: String): String? {

    val tokens = message.split("\n")
    val emptyLineIndex = tokens.indexOfFirst({line ->
                                                line.isBlank() })

    if (emptyLineIndex < 0 ||
        emptyLineIndex == tokens.lastIndex) {
        return null
    } else {
        return tokens.subList(emptyLineIndex + 1, tokens.size)
            .joinToString("\n")
    }
}
```



```

fun getHttpBodyBlock(message: String): String? {

    return message.split("\n")
        .run {
            // this == message.split("\n")
            // indexOfFirst is called on List<String>
            indexOfFirst { it.isBlank() } // "it" represents element in list
            .let {
                // "this" has not changed, still pointing to List<String>
                // "it" here is the index returned by indexOfFirst
                // note that lastIndex is this.lastIndex, on List<String>
                if (it < 0 || it == lastIndex) return null
                // subList and size are called on "this", ie List<String>
                else return subList(it + 1, size).joinToString("\n")
            }
        }

    /*
    note the total lack of local variables...
    however it can become difficult to read...
    */
}

```

# More

- There is more related to Kotlin
- But you do not need to learn all details to be able to be productive in Kotlin
- Throughout the course, I might introduce some more concepts based on the code examples I wrote

# Kotlin Negative Sides

- Nothing is perfect, and you will always find different opinions
- Eg, *minor* things I do not like in Kotlin
  - No *ternary operator*, eg “return x==5 ? 0 : 1 ”, although in Kotlin “if” is an expression, eg “return if(x==5) 0 else 1”
  - Poor handling of *static methods*, but that might change in future releases
  - Still rough edges regarding typing and generics
- Lot of “magic” in Kotlin, so not recommended for total beginners (ie Java is a better introductory language)

# Kotlin Major Design Flaw

- In Kotlin, classes and methods are *final* by default
  - You need to use keyword *open* to specify they can be overridden
- *Final by default is a solution to a near non-existent problem*
- And unfortunately it creates a lot, a lot of problems
  - eg, when dealing with libraries like Spring and Hibernate
- Corollary: do not use Kotlin to write **libraries**. If a library is written in Kotlin, avoid using it if another equivalent library exists in Java

# Kotlin and Maven

- We will compile Kotlin to JDK bytecode
- We will compile with Maven
- Need special plugin to compile Kotlin code
- This plugin will need special settings to handle libraries like Spring and Hibernate

# Web Services

# Data/Operations Over Network

- Provide APIs over network
- Typically TCP connections
- HTTP most common protocol
- So, can see a Web Service as a process that opens a TCP port and responds to incoming requests

# Types of Web Services

- REST

- most common nowadays
- usually strongly tied to HTTP protocol
- *not a protocol, but set of architectural guidelines*
- typically serving data in JSON

- SOAP

- very common in the past, but disappearing nowadays
- actual protocol, usually over HTTP
- tied to XML

- GraphQL

- the new kid



# Why?

- When you want to provide programmable functionalities to your clients over the network
  - eg, see public list at <http://www.programmableweb.com/>
- Separation of *frontend* from *backend*
  - JavaScript doing client-side HTML rendering on browser, where backend is just a web service providing data
- *Microservice* Architecture
  - large systems split into several web services of more manageable size
  - extremely important for modern enterprise systems

# SpringBoot

- We will write web services in SpringBoot, using Kotlin
- We start with REST
  - This will take roughly half of the course
- To properly write REST, need to learn details of HTTP
- Recall: REST is just a set of architectural guidelines, and NOT a formal protocol
  - at the beginning, we will do some mistakes *on purpose*

# Data Formats

# Data in Web Services

- Web Services will provide data and functionalities over the network
- Servers and clients can be written in different languages
  - Java, C#, JavaScript, Kotlin, Python, Go, PHP, etc.
- *Data formats should be independent from the programming languages*

# XML

- Very popular in the *past*
- OKish for configuration files (eg, Maven *pom.xml*)
- Quite verbose for data over the network
- Not so much used any more (apart from SOAP services)

# JSON

- **JavaScript Object Notation (JSON)**
- Less verbose than XML
- Very poor for configuration files (e.g., no comments)
  - YAML and XML are better
- *Can be used directly by JavaScript running in the browser*
- Practically the most common data format for web services nowadays

# Git Repository Modules

- *NOTE: most of the explanations will be directly in the code as comments, and not here in the slides*
- **advanced/kotlin**
- **advanced/data-format**
- **advanced/calling-webservice**