Who are you?

Big Data & Java Technical Leader

Mentoring

Consulting

Lecturing

Writing courses

Writing code

A Brief Intro to **Scala**


I heard about Java 9 problems…


Use Scala, brother
I heard about Java 9 problems…

Dynamic vs. Static

Dynamic (Ruby / Python)

- Concise
- Scriptable
- **R**ead-**E**val-**P**rint **L**oop (irb)
- Higher Order Functions
- Extend existing classes
- Duck Typing

Static (Java)

- Better IDE Support
- Less tricks - Fewer Tests
- Documentation
- Open Source Libs (for Java)
- Performance
- JVM Tools (VisualVM)
- True Multi-threading

Don’t implement interface, just be it!

Interface sometimes is a bureaucracy

Dynamic vs. Static

Dynamic (Ruby / Python)

- Concise
- Scriptable
- **R**ead-**E**val-**P**rint **L**oop (irb)
- Higher Order Functions
- Extend existing classes
- Duck Typing
- method_missing

Static (Java)

- Better IDE Support
- Less tricks - Fewer Tests
- Performance
- JVM Tools (VisualVM)

Scala

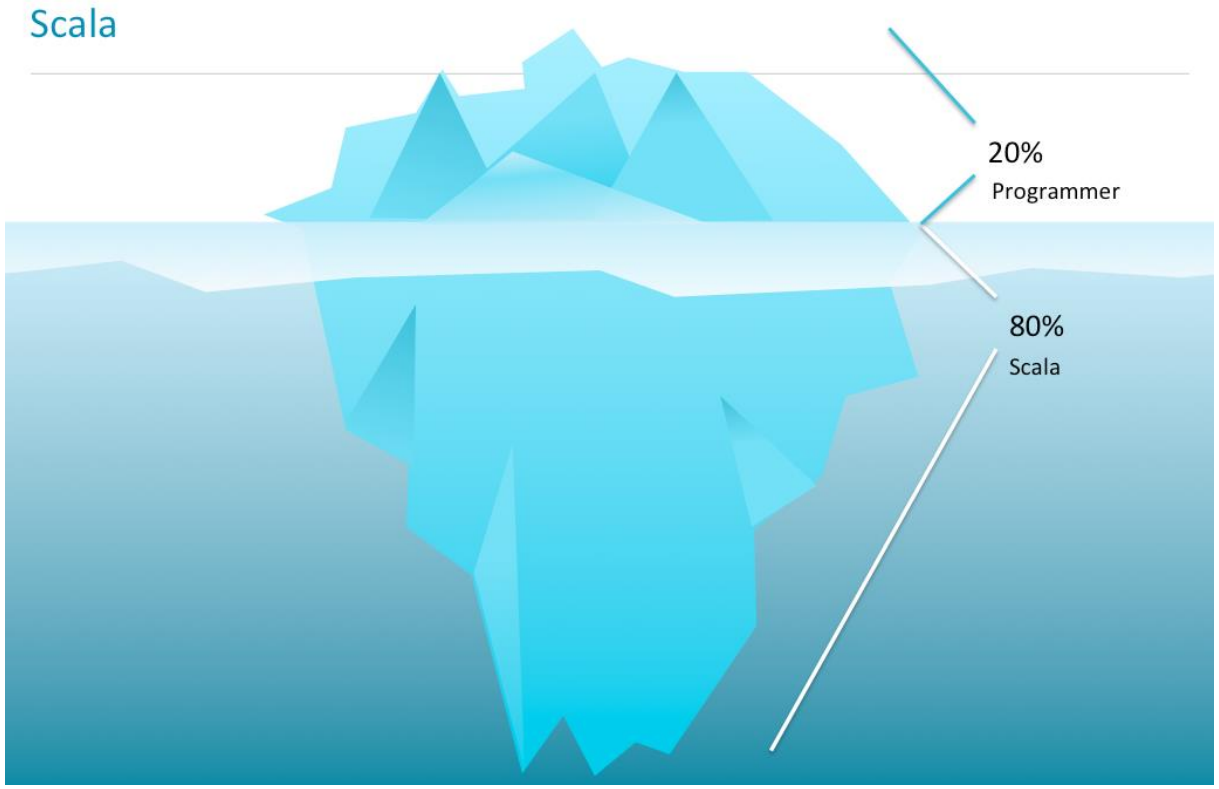
- ✓ Concise

Scriptable

- ✓ **R**ead-**E**val-**P**rint **L**oop
- ✓ Higher Order Functions
- ✓ Extend existing classes
- ✓ Duck Typing
- ✓ method_missing
- ✓ Better IDE Support
- ✓ Fewer Tests
- ✓ Performance
- ✓ JVM Tools (VisualVM)
- ✓ True Multi-threading

Scala & Java

- All Java types are available
  - But in most cases the developers will use Scala types analogs
- Scala is expensive



Scalable language

by Martin Odersky

Since 2001

Scala is a **modern multi-paradigm** programming language designed to express **common** programming **patterns** in a **concise**, **elegant**, and **type-safe** way.

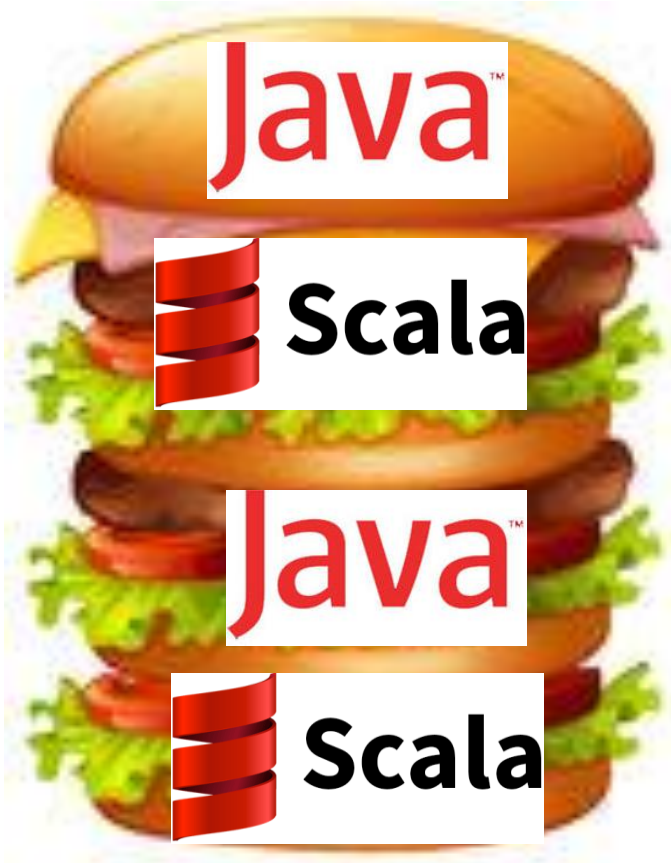
Scala

- Statically Typed, but types are not must
- Runs on JVM, full inter-op with Java
- Object Oriented
- Functional

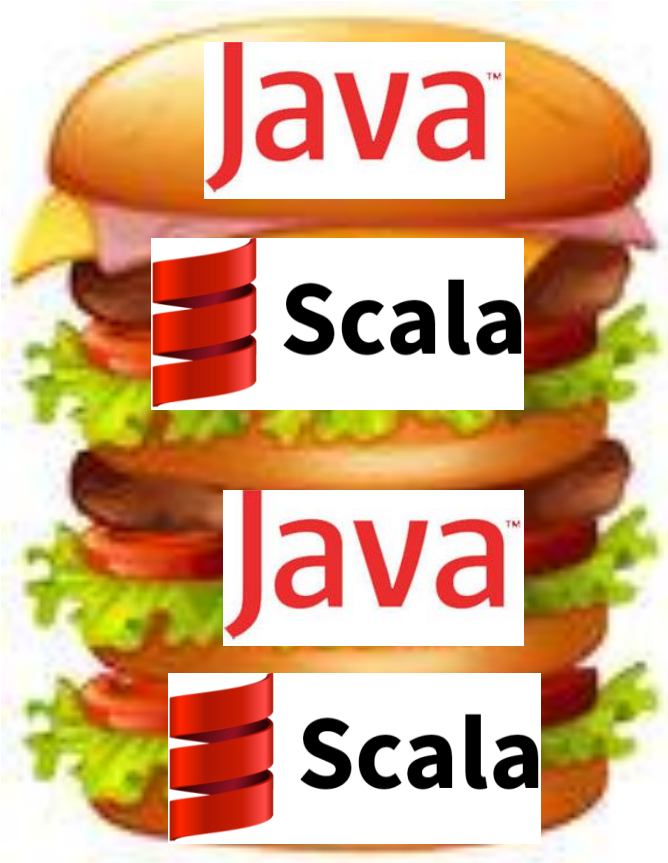
Scala is Practical

- Can be used as drop-in replacement for Java
  - Mixed Scala/Java projects
- Use existing Java libraries

You can mix Scala and Java



Not everything will work



Scala is Practical

- Can be used as drop-in replacement for Java
  - Mixed Scala/Java projects
- Use existing Java libraries
- Use existing Java tools and frameworks (Ant, Maven, JUnit, Spring etc…)

Not everything is working…

Scala is Practical

- Can be used as drop-in replacement for Java
  - Mixed Scala/Java projects
- Use existing Java libraries
- Use existing Java tools (Ant, Maven, JUnit, etc…)
- Decent IDE Support (NetBeans, **IntelliJ**, *Eclipse*)

Scala is Concise

Type Inference

```
val sum = 1 + 2 + 3

val nums = List(1, 2, 3)

val multiMmap = Map("ab" -> List(1,2))
```

Explicit Types

```
val sum: Int = 1 + 2 + 3

val nums: List[Int] = List(1, 2, 3)

val map: Map[String, List[Int]] = ...
```

Scala is not like Javascript

```
var a = 3
a = 12
a = "now I'm a String!!!"
```

Expression of type String doesn't conform to expected type Int

Higher Level

```
// Java – Check if string has uppercase character
boolean hasUpperCase = false;
for(int i = 0; i < name.length(); i++) {
    if(Character.isUpperCase(name.charAt(i))) {
        hasUpperCase = true;
        break;
    }
}
```

Higher Level

```
// Scala
val hasUpperCase = name.exists(_.isUpperCase)
```

Less Boilerplate

```
// Java
public class Person {
  private String name;
  private int age;
  public Person(String name, Int age) {  // constructor
    this.name = name;
   this.age = age;
  }
  public String getName() {              // name getter
    return name;
  }
  public int getAge() {                  // age getter
    return age;
  }
  public void setName(String name) {     // name setter
    this.name = name;
  }
  public void setAge(int age) {          // age setter
    this.age = age;
  }
}
```

Less Boilerplate

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

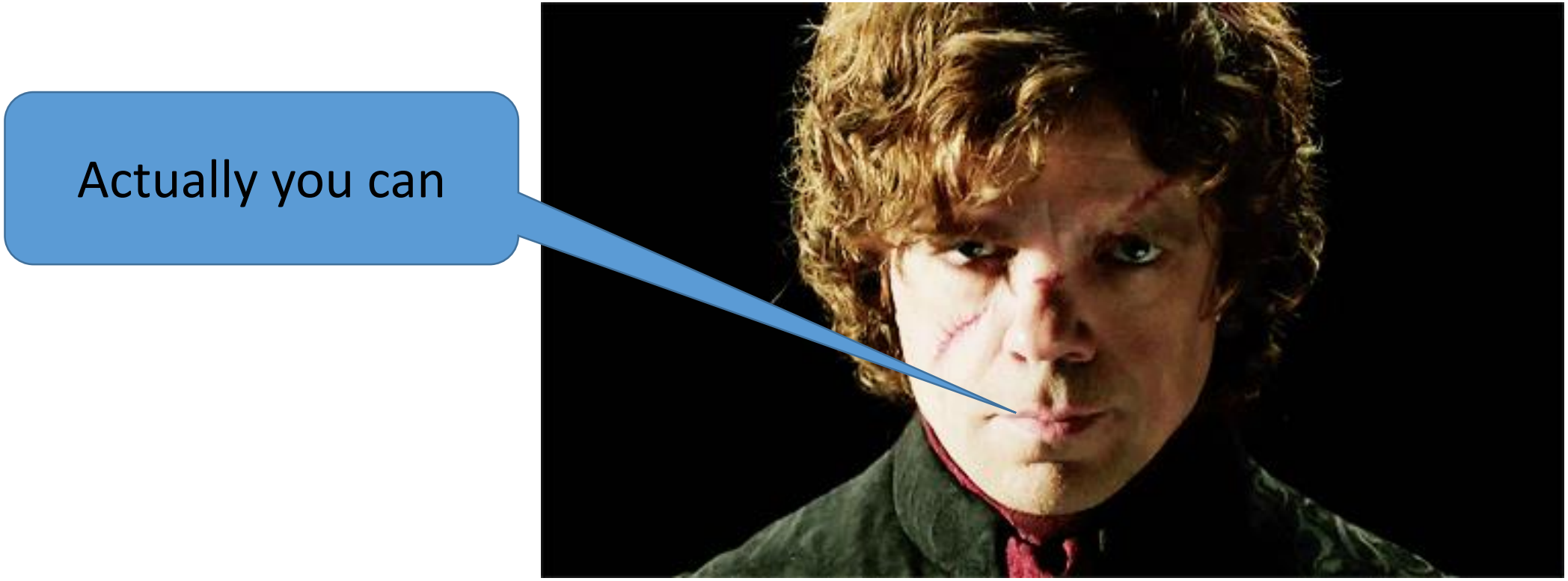

WHO NEEDS SETTERS?
BE IMMUTABLE !!!

You can’t fight the Lombok

```
@Data @AllArgumentConstructor
public class Person {
    private String name;
    private int age;
    private double income;
}
```

Three classes in 3 lines

```
case class Person(name:String,age:Int,income:Double)
case class Point(x:Int,y:Int)
case class Pub(beers:List[Beer])
```



Variables and **Val**ues

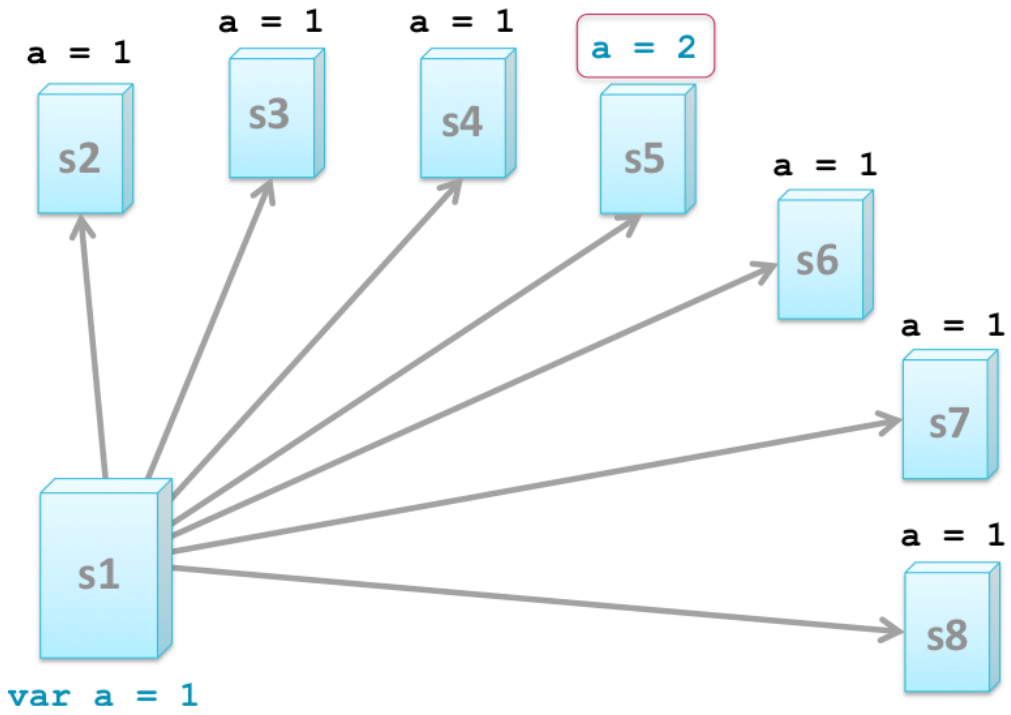
```
// variable
var foo = "foo"
foo = "bar"  // okay

// value
val bar = "bar"
bar = "foo"  // nope
```

Immutable vs Mutable

- When you declare variables:
  - val – immutable
  - var – mutable
- Collections
  - scala.collections.immutable
  - scala.collections.mutable

Why immutable is good?

- Because mutable is bad



Who is responsible to update the others?

Wait for more immutable later

Scala is **O**bject **O**riented

Pure O.O.

```
// Every value is an object
1.toString

// Every operation is a method call
1 + 2 + 3  →  (1).+(2).+(3)

// Can omit . and ( )
"abc" charAt 1  →  "abc".charAt(1)
```

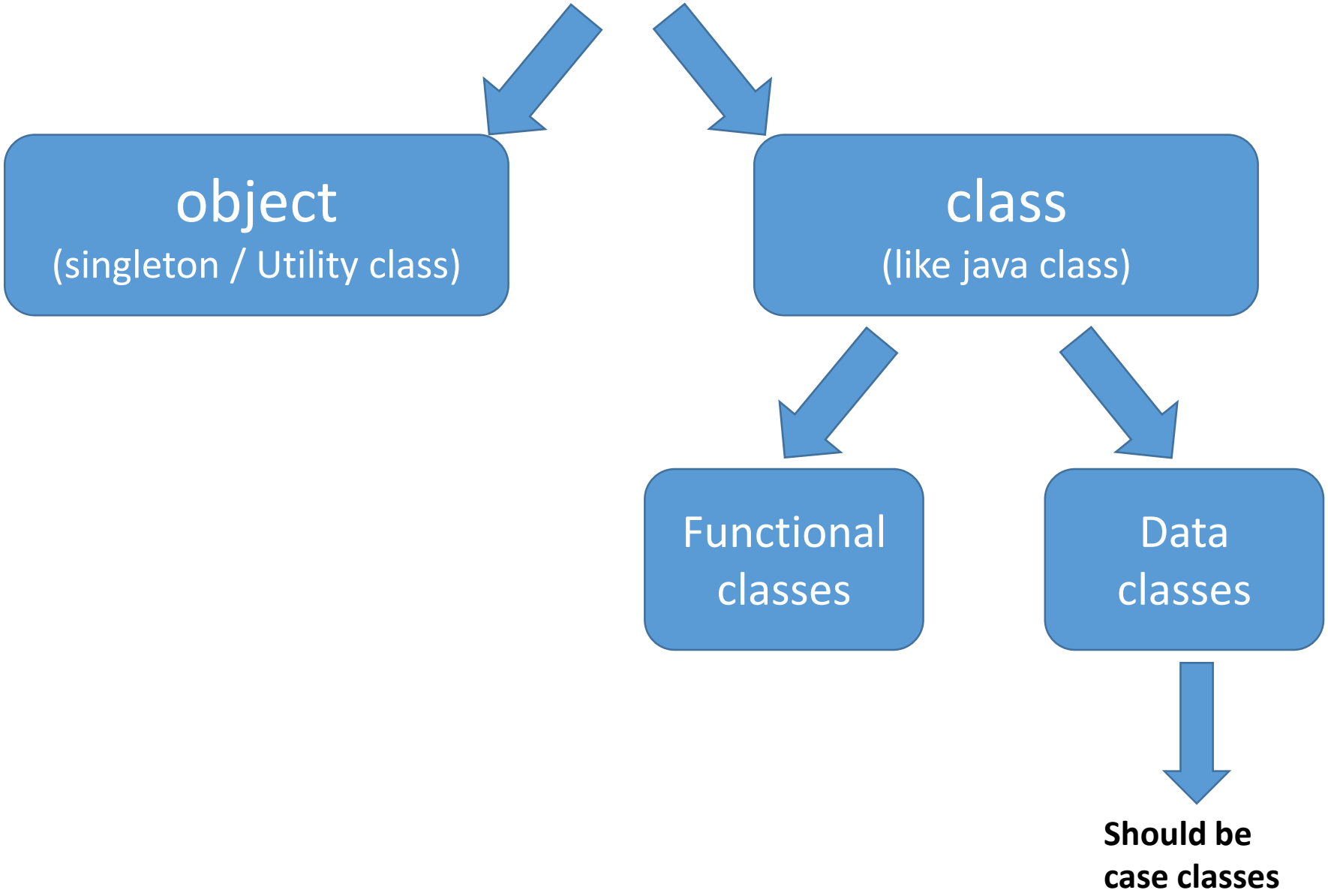
How can it be?
Java has primitive types!

Because Scala has its own types

Scala types – the important ones

Type	Description	Example
`Int`	4 byte integer	`3`
`Long`	8 byte integer	`32754L`
`Double`	8 byte floating point	`3.1415`
`Float`	4 byte floating point	`3.1415F`
`Char`	single character	`'c'` (single quotes)
`String`	sequence of characters	`"iFruit"` (double quotes)
`Boolean`	true or false	`true` (case sensitive)

Scala classes



Scala types

- trait
- object
- class
- case class
- enumaration
- AnyVal / AnyRef / Any


THERE CAN BE ONLY ONE


THERE CAN BE
ONLY ONE!

Scala object

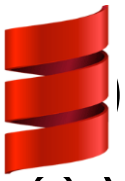
Scala object

- Looks like regular class, but called **object** instead of class
- An object of this class will be created when class will be loaded
- Scala doesn’t have static modifier, but an object class can be accessed from any context

Where the main method should be located?

1) object class
2) regular class
3) case class
4) Scala doesn’t have main method

Where the main method should be located?



1) object class
2) regular class
3) case class
4) Scala doesn't have main method

Hello World lab

- create scala object
- write main method (you can use Intelij life templates: main + tab)
- print “hello world from it”


SHOW US A CODE!!!

First Scala application

```
package solutions.lab0

/**
  * @author Evgeny Borisov
  */
object Main {
  def main(args: Array[String]): Unit = {
    println("hello world")  //one line comment
    val name = "Anatoliy"
    println("Hello "+name)
   /* block of
    comments*/
    println(s"Hello $name")
    println(s"Hello ${name+" !"}")
  }
}
```

Class names and packages

- Java class name
  - Classpath variable
    - Compile time
    - runtime
  - ClassLoaders
- Java package
  - Java meaning
  - File system conventions
- Scala class name
- Scala package

Classpath

- Java compiler should know the locations of the jars we use
- In Runtime we also needs to know the locations of the jars we use
- The same property: classpath is used to provide this location
  - javac Main.java –classpath=c:\libs
  - java Main –classpath=c:\libs

System Classloader

20 Facts about my self:
1. I'm lazy
2. …

Looks for the class by fully qualified name of the class (packages + file name) in all jars provided in runtime classpath ignoring the name of the Jars

java package

- Just a namespace
- Any Java class can declare by keyword package to which package he belongs.
- Physical location of the file is not requested
- By convention each package has his own directory name like a package
- NAME OF THE PACKAGE SHOULD BE WRITTEN IN lowercase

Scala class name

- Exactly like Java class name, because it is managed in runtime by the same class loaders rules

Scala package

- Kind of script file where you can declare and implement scala classes, objects, traits e.t.c
- Usually used to declare a lot of short case classes which part of the same model

Any scala object which extends App can be used as main method

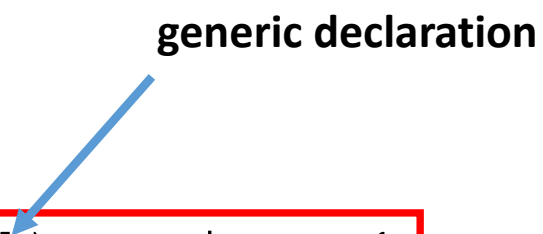
```
class Main extends App {
```


```
}
```

Go back to our code

First Scala application

```
package solutions.lab0

/**
  * @author Evgeny Borisov
  */
object Main {
  def main(args: Array[String]): Unit = {
    println("hello world")  //one line comment
    val name = "Alex"
    println("Hello "+name)
   /* block of
    comments*/
    println(s"Hello $name")
    println(s"Hello ${name+" !"}")
  }
}
```



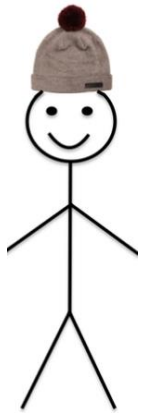
Special type - Unit

- When function returns nothing it returns the Unit

```
val myreturn = println("Hello, world")
> myreturn: Unit = ()
```

Just for information

This is Unit.
There is only one Unit in Scala
Unit can't be created
Unit – is built in singleton
Be like Unit



Special type - Any

```
val myreturn = if (true) "hi"
> res2: Any = hi

val mystring = myreturn.asInstanceOf[String]
> myreturn: String = hi
```

Just for information

This is Any.
Any – like Object, but wider
Any – solves problem with generic collections and primitives
Be like Any

Special type AnyRef

- Like any reference in java
- Kind of Object
- Null, when not initialized

Special type AnyVal

- Like record in some other languages
- Scala primitives

Scala types cont.

- You can’t declare private class member without providing to it default value

```
object Person{

  val name:String

}
```

Only classes can have declared but undefined members

```
18    class Person{

Class 'Person' must either be declared abstract or implement abstract member 'name: String' in 'solutions.lab0.Main.Person'

20      val name:String
```

```
class Person(name: String){

}
```

```
object Person{
  val name:String="Jeka"
}
```

```
class Person{
  val name:String="Jeka"
}
```

```
def main(args: Array[String]): Unit = {
 println(new Person().name)
 println(Person.name)
}
```

Scala object methods

```
object PersonService {
  def processPerson(person:Person):Unit={
    println(person)
  }
}
```

```
now in any context you can use it
PersonService.processPerson(new Person(name = "Jeka"))
```

Scala class methods

```
class PersonService {
  def processPerson(person:Person):Unit={
    println(person)
  }

  def main(args: Array[String]): Unit = {
    new PersonService().processPerson(new Person(name = "Jeka"))
//    val service:PersonService = new PersonService()
    val service = new PersonService()
    service.processPerson(new Person(name = "Jeka"))
  }
}
```

Constructor

- Both class and object can have only one constructor
- Constructor is just a code which written in the class body
- If you will think about it, actually every thing in the class body is a part of constructor
- Constructor is invoked when object is created
  - Once for object, when his bytecode will be loaded
  - For regular class every time when “new” keyword and class name after it new Person()

Constructor parameters

Supported only for class, not for object, because object is initialized by classloader

- Can be declared as following and will be class members in case it is declared with **`val`** or **`var`**

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

  - **Name and age are constants which will be available only in class constructor or methods of this class.**
  - **They are not members of object**
  - **They even not private, they just not available from outside the class**

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

  - **Name and age are parameters which will be available in constructor.**
  - **They are also members of object**

Puzzle

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

def main(args: Array[String]): Unit = {

//what line has compile error?
  val john = new Person(age = 12,name = "John")
  val jack = new Person("Jack",30)
  val mali = new Person(12,"Mali")
  val mike = new Person("Mike")
  val ivan = new Person("Jack",age = 30)
}
```

Constructor parameters

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


def main(args: Array[String]): Unit = {

//what line has compile error?
  val john = new Person(age = 12,name = "John")
  val jack = new Person("Jack",30)
  val mali = new Person(12,"Mali")
  val mike = new Person("Mike")
  val ivan = new Person("Jack",age = 30)
}
```

Default parameters

- Support both in methods and constructors

```
class Person(name:String="",var age:Int=18){}



// What is not valid?

val john = new Person(age = 12,name = "John")
val jack = new Person("Jack")
val mike = new Person(12)
val ivan = new Person(age = 30)
val mali = new Person(name = "Mali")
```

Default parameters

- Support both in methods and constructors

```
class Person(name:String="",var age:Int=18){}



// What is not valid?

val john = new Person(age = 12,name = "John")
val jack = new Person("Jack")
val mike = new Person(12)
val ivan = new Person(age = 30)
val mali = new Person(name = "Mali")
```

Default method

- In any class you can declare method called apply() which can take any parameters and return any type
- This method can be called without using it's name

```
class Point(val x: Int, val y: Int)
```

```
object PointCreator {
  def apply(x: Int, y: Int): Point = {
    new Point(x,y)
  }
}
```

```
def main(args: Array[String]): Unit = {
  val point: Point = PointCreator.apply(3,4)
  val point2: Point = PointCreator(3,4)
}
```

Default method “apply”

- This method make sense in object, not in class
- Usually this method used as kind of constructor
- You can theoretically declare this method in class, but in order to use it you first will be needed to create an object and then call this method from that object
- Scala objects has automatically methods apply and unapply

Case classes

```
case class Person(name:String,age:Int,income:Double)
```

- Scala will generate equals + hashcode + toString methods (like in Java lombok @Data)
- Scala will generate object for the case class with the same name, which will have apply method for simple instantation and unapply method which will be needed in pattern matching
- You don’t have to use **new** to create it
- Has **copy** method to help you with immutability
- You don’t need in case class to mark constructor variables with **val** to make the class members, but you still can use **var**

Case classes inheritance is prohibited

Case class example

```
case class Human(name:String, age:Int)


def main(args: Array[String]): Unit = {
  val human = Human("john", 12)
  val human2 = Human(name="john", age = 12)
  val human3= Human("john", age = 12)
  println(human.age)
  println(human2.name)
  println(human3)
}
```

Puzzler – what will be printed out

```
case class Human(name:String, age:Int=12)


val human = Human("john", 11)
println(human.copy(age = 4).copy("Jeka"))
```

Puzzler – what will be printed out

Human(Jeka,4)

Puzzler

```
case class Human(name:String="Nir", age:Int=37)
```

In some main we are trying to do this. What will happen?

```
val human = Human
println(human)
```

1. Compilation error
2. Runtime exception
3. Human will be printed
4. Human(“Nir”,37) will be printed
5. Human(null,0) will be printed

Puzzler

```
case class Human(name:String="Nir", age:Int=37)
```

In some main we are trying to do this. What will happen?

```
val human = Human
println(human)
```

1. Compilation error
2. Runtime exception
3. **Human will be printed**
4. Human(“Nir”,37) will be printed
5. Human(null,0) will be printed

Lab – case classes and Objects

- Declare case class NewLogin(username, password,confirmPassword)
- Write LoginValidator object with method validate(login:NewLogin)
- Method should print next errors:
  - Name can’t be “admin”
  - Password can’t start from “z”
  - Password can’t be equal to username
  - Password can’t contains only digits
  - ConfirmPassword should equal password
- Only one error will be printed, even if there more than one error
- The printed message should contain not only static text, but relevant object state
- In case login doesn’t have any problems it should be printed with “accepted”

Classes inheritance

```
// Classes (and abstract classes) like Java
abstract class Language(val name:String) {
  override def toString = name
}

// Example implementations
class Scala extends Language("Scala")

// Anonymous class
val scala = new Language("Scala") { /* empty */ }
```

What is not Valid?

```
class A(x:Int)
class B extends A
```

```
class A(val x:Int)
class B(val x:Int) extends A(x)
```

```
class A(x:Int=42)
class B(y:Int) extends A(y)
```

```
class A(x:Int)
class B(y:Double) extends A(y.toInt)
```

```
class A(x:Int=73)
class B(y:Int) extends A
```

```
class A(val x:Int)
class B() extends A(42)
```

```
class A(val x:Int)
class B(val y:Int) extends A(y)
```

```
class A(x:Int)
class B(val y:Int) extends A(y)
```

```
class A(val x:Int)
class B(var y:Int) extends A(y)
```

```
class A(val x:Int)
class B(y:Int) extends A(y)
```

Inheritance

- Class can extend from only one class (abstract or not)
- Object can extend from only one class (abstract or not)
- Nothing can extend from Object

Traits – almost like interface in Java

- Trait can declare abstract methods and methods with body
- Trait body is a constructor (exactly like in regular class) and it’s code always will be invoked for any class which extend from the trait in the moment when object will be created from this class, before his own constructor
- Trait can’t declare constructor parameters
- Class can extend from several Traits by using extend for the first trait or class and by using “with” for the rest traits

Traits

```
// Like interfaces in Java
trait Language {

  val name:String

  // But allow implementation
  override def toString = name
}
```

Traits

```
trait Language {
  val name:String
  override def toString = name
}
```

```
trait JVM {
  override def toString = super.toString+" runs on JVM" }
trait Static {
  override def toString = super.toString+" is Static" }

// Traits are stackable
class Scala extends Language with JVM with Static {
  val name = "Scala"
}

println(new Scala) → ???
```

Traits

```
trait JVM {
  override def toString = super.toString+" runs on JVM" }
trait Static {
  override def toString = super.toString+" is Static" }


// Traits are stackable
class Scala extends Language with JVM with Static {
   val name = "Scala"
}


println(new Scala) → "Scala runs on JVM is Static"
```

Lab for the Traits

- Write a trait Quoter with method printMessage, which doesn’t take any params and return Unit.
- Write three implementations
  - Scala object: RandomQuoter – his printMessage method will print random quote
  - Class: ShakespearQuoter - this class can take in constructor some message and print it, when printMessage invoked, the default message should be “to be or not to be”
  - Case class: MessageQuoter – this class should take mandatory parameter in constructor and print it every time when printMessage method will be invoked
- Bonus: for RandomQuoter use java dependency DataFactory
- Try to put all this 3 implementation in QuoterAggregator list (new class you should create)

Pattern matching lab

- Write method useQuoter(quoter:Quoter) depends on it’s type do different logic
- In case of RandomQuoter – invoke printMessage 10 times
- In case of MessageQuoter – take the message and print it uppercased
- In case of ShakespearQuoter – take it’s message and popup it

Lab – Write heroes game in text mode

Class Character int power, int hp void kick(Character c) boolean isAlive()

Hobbit power =0, hp = 3, kick(toCry();)

Elf hp = 10, power = 10, kick(kill everybody which weaker than him, otherwise decrease power of other character by 1)

King power 5-15, hp 5-15, kick(decrease number of hp of the enemy by random number which will be in range of his power

Knight power 2-12, hp 2-12, kick(like King)

CharacterFactory

Character createCharacter() – returns random instance of any existing character

GameManager

void fight(Character c1, Character c2){

to provide fight between to characters and explain via command line what happens during the fight, till both of the characters are alive

}

CharacterFactory problem

- Every time we will add new Game Character class we will need to update list of classes in CharacterFactory

```
val characterClasses = List(classOf[Troll],classOf[Knight],
                            classOf[Hobbit],classOf[Elf])
```

- It’s against open close principle
- In order to improve it we need scan particular packages or all classpath for classes which extends Game Character class
- We don’t have this API in standard reflection

Reflections.jar

```
<dependency>
    <groupId>org.reflections</groupId>
    <artifactId>reflections</artifactId>
    <version>0.9.11</version>
</dependency>
```

Implicits

- What if we want to add some methods to third party library classes in specific context?
- About what topic we going to talk now?

Implicit types

- Val implicit
- Def implicit
- Class implicit

Val implicit

```
implicit val x:Double = 10

def printMe(implicit d:Double):Unit=println(d)

printMe
```

Val implicit

```
import ConstantsHolder._

def main(args: Array[String]): Unit = {
  printPerson(Person("Olga"))
  printPerson
}

def printPerson(implicit p:Person): Unit =println(p)



object ConstantsHolder {
  implicit val defaultPerson:Person=Person(name = "Vsevolod")

}
```

Method implicit

```
implicit def createPerson(name:String):Person={
  Person(name)
}

def main(args: Array[String]): Unit = {
  val p:Person="Oleg" // if variable p will not be declared as Person,
                      it will be String

  "Svyatoslav".printMe()  //String doesn't have printMe method,
      so compiler will generate createPerson("Svyatoslav").printMe()
}
```

Method implicit

```
def main(args: Array[String]): Unit = {
  import Convertions._    //could be also Convertions.convert
  val p:Person="Ryrik"   // if variable p will not be declared as Person, it will be String
  "Svyatoslav".printMe()  //String doesn't have printMe method,
 so compiler will generate createPerson("Svyatoslav").printMe()
}


 object Convertions {
   implicit def convert(name:String):Person=Person(name)
 }
```

Implicit with class

1. Write class Extensions which will have all methods you want to add to class Current (usually the Extension class will take in constructor the Current instance, because in new methods you usually will use the methods of Current class
2. Write an object Registry with implicit method, which will take as a parameter Current instance and will return Extension instance (Usually this method will look like: new Extension(current)
3. In the context where you want this extensions method will be available for Current instance use import for all Registry methods (import labs.Registry._)
   It can be done in class, method, or some other code block
4. Ask me how does it work

Implicit lab

Add to String two methods

1. isBlank
2. isEmail

Long complicated lab

```
trait Costable {
  def cost: Int
}
```

```
trait ProductFactory {
  def generateProduct(): Costable
  def
generateProducts(amount:Int):List[Costable]
}
```

add getRandomItem to list
and use it in generateProduct

```
case class Chair(cost: Int) extends Costable

case class Table(cost: Int) extends Costable

case class Car(cost: Int) extends Costable

case class House(cost: Int) extends Costable
```

Write ProductFactoryImpl object

```
trait ProductFactory {
  def generateProduct(): Costable
  def generateProducts(amount:Int):List[Costable]
}
```

Try to use ClassResolver we already developed for generateRandomCharacter method, pay attention that his scanner configured to different package

Generate product should use random Item method, which should be added to List by implicit

Write MoneyCalculator object

def totalPrice(list:List[Pricable]):Int

```
trait Pricable {
  def price:Int
}
```

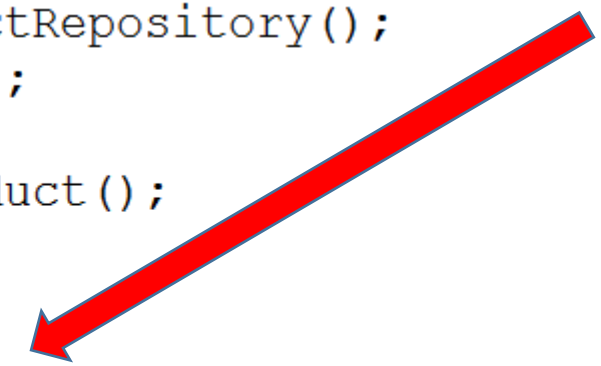
In your main method create call 10 times to generateProducd method, and put the result to List.

If you will try to use totalPrice method on this list, you will have compilation error.

Solve this problem without changing existing classes, only by adding new classes and changing main method

Your main will look like this, but in Scala

```
ProductRepository productRepository = new ProductRepository();
ArrayList<Costable> products = new ArrayList<>();
for (int i = 0; i < 5; i++) {
    Costable product = productRepository.getProduct();
    products.add(product);
}
int total = Calculator.calculateTotal(products);
System.out.println("total = " + total);
```



```
public class Calculator {
    public static int calculateTotal(List<Priceble> items) {
        int total = 0;
        for (Priceble item : items) {
            total += item.getPrice();
        }
        return total;
    }
}
```

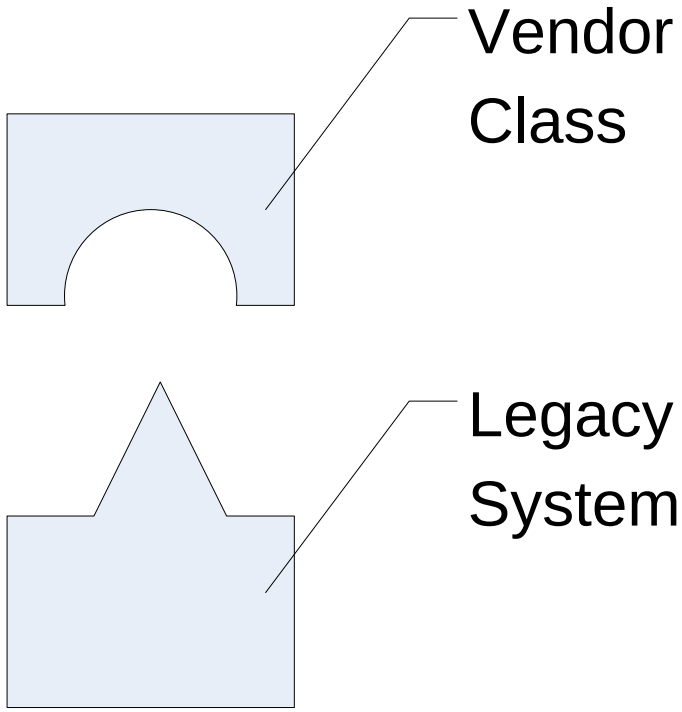
Adapter

Adapters all around

- Simple
  - Only change interface
- More complex
  - Have some processing inside

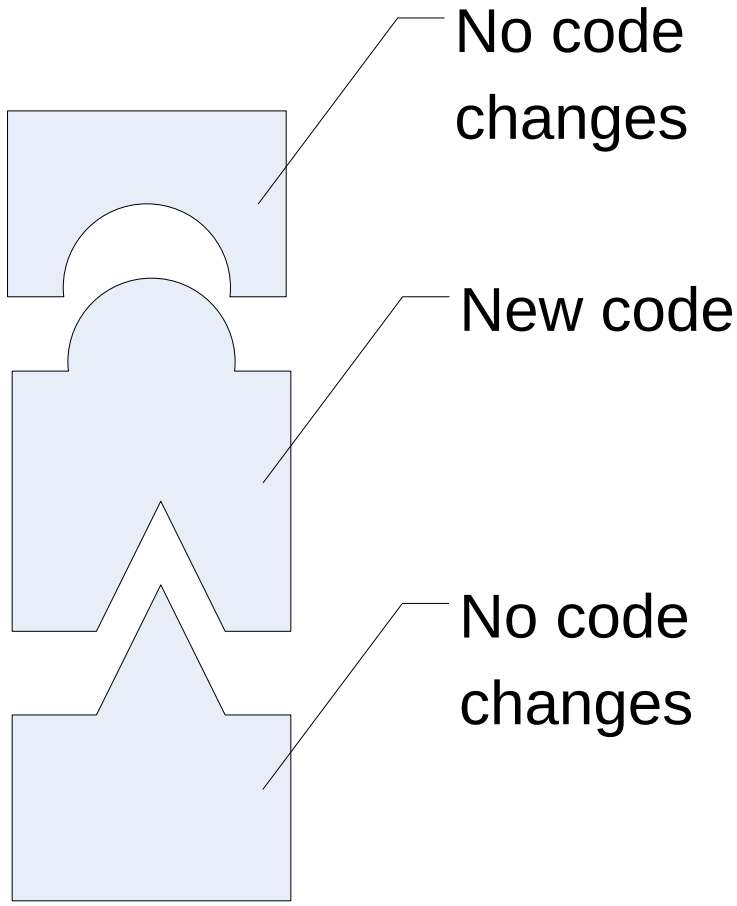
Adapter - The problem

- Sometimes there is a client with incompatible interface
- How can we use the interface without changing existent code?

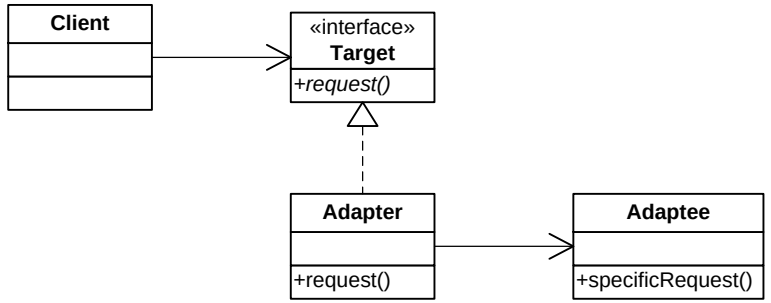


Adapter – The Solution

- As usual, another layer of indirection solves the problem
- The adapter implements interface of the legacy system
- And talks to the vendor interface to service the requests
- Players:
  - Adapter implements Target interface, which is known to the Client
  - Adapter passes requests to the Adaptee



Adapter – The Solution



- Adapter uses composition to pass the requests to from the Target to Adaptee
  - Another type of Adapter - Class Adapter subclasses both Target and Adaptee and overrides the requests
    - Multiple inheritance, not possible in Java

Lets complicate our task

- The cost of costable is in NIS
- The price of Pricable is in USD

Add implicit method to List[Costable]

```
val pricables = CostableFactory.generateProducts(10).asPricableList()
```

Do it once by using Adapter class, you already developed for previous lab
Do it by using Anonymous class

This is another anti-pattern: God Class

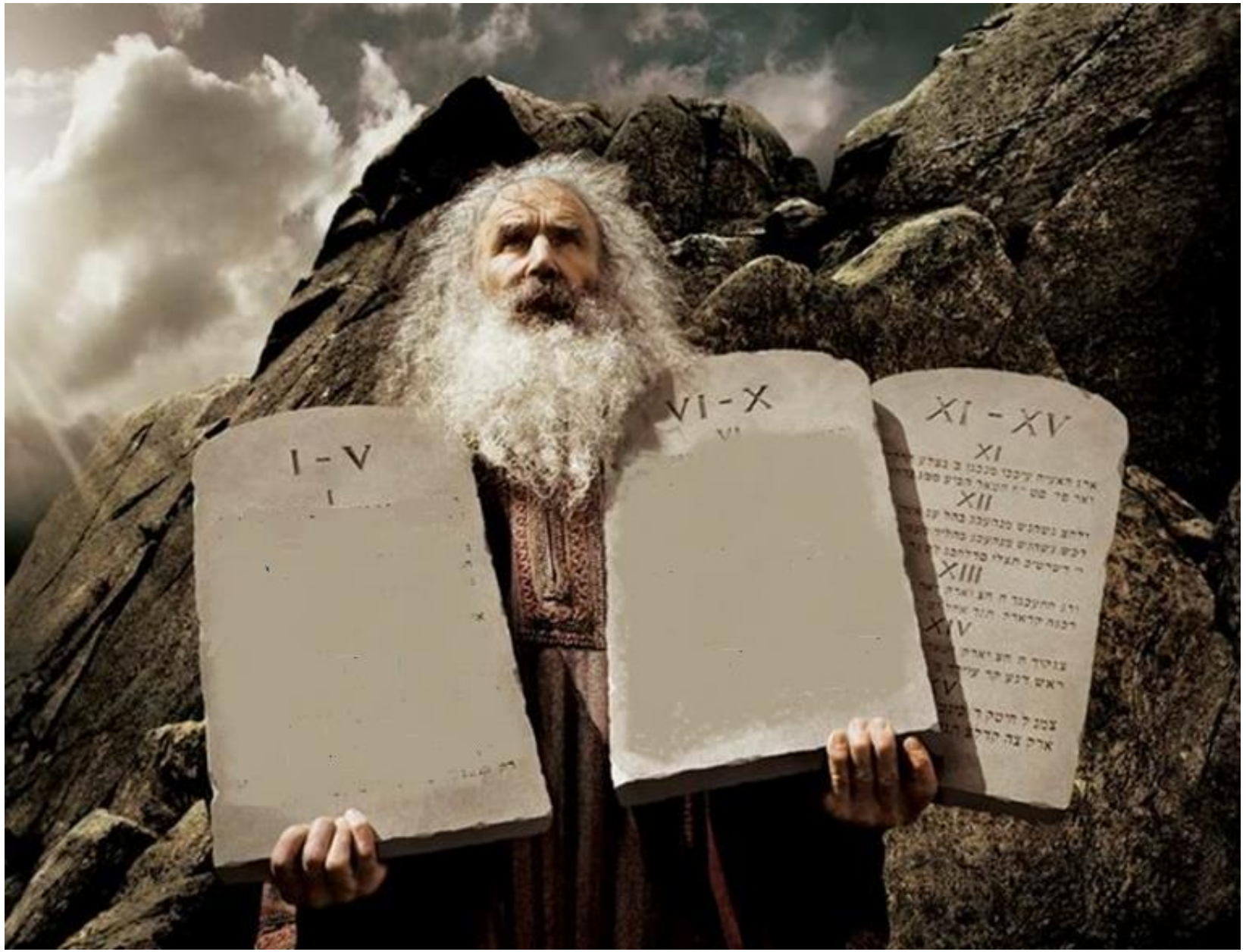
The reasons for it:

1. If / else
2. While
3. Switch / case
4. goto

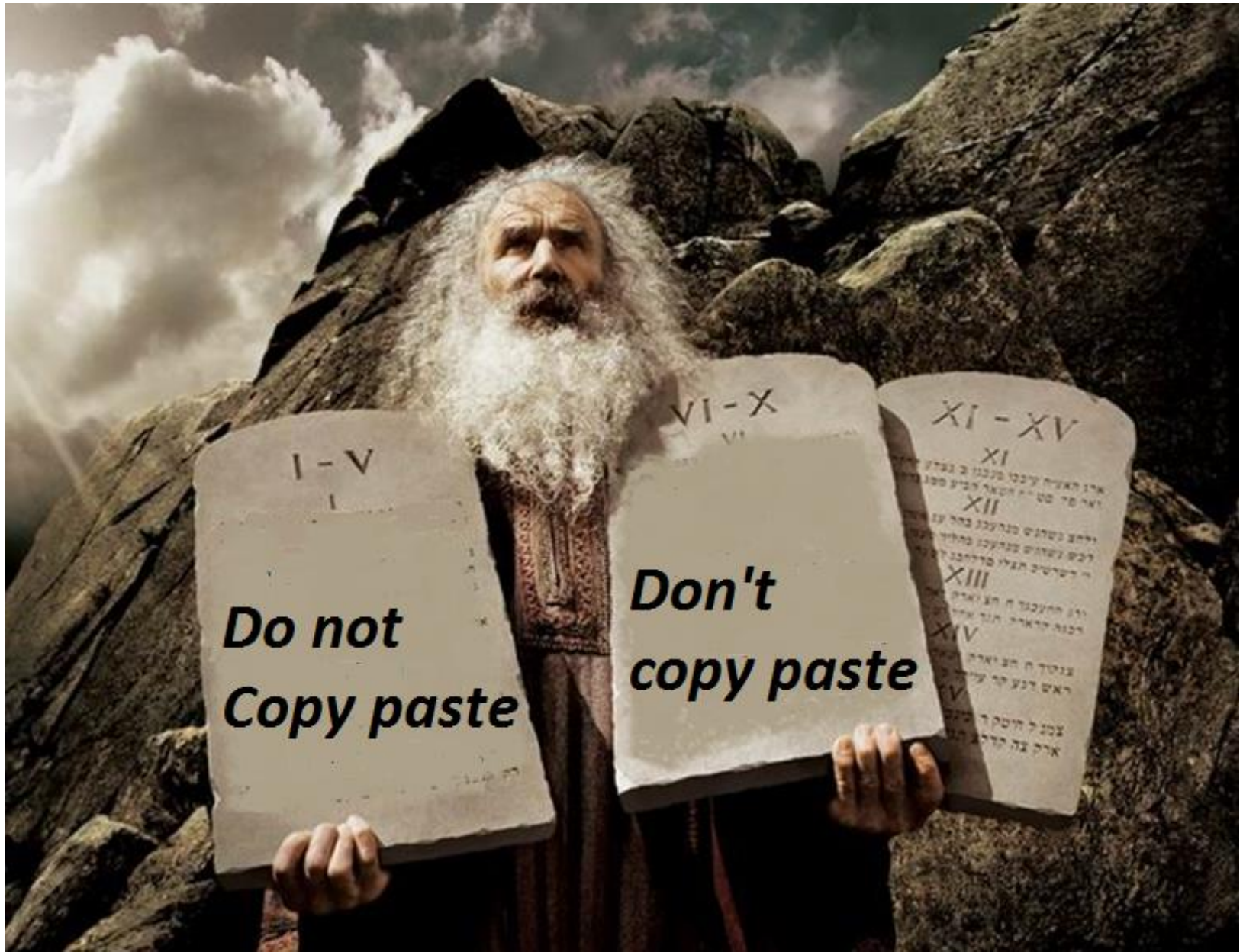
What is a bad design?

- Rigid – Hard to change a part of the system without affecting too many other parts
- Fragile – When making a change, unexpected parts of the system break
- Immobile – Hard to reuse it in another application because it cannot be disentangled from the current application

2 Main Commandments of developer



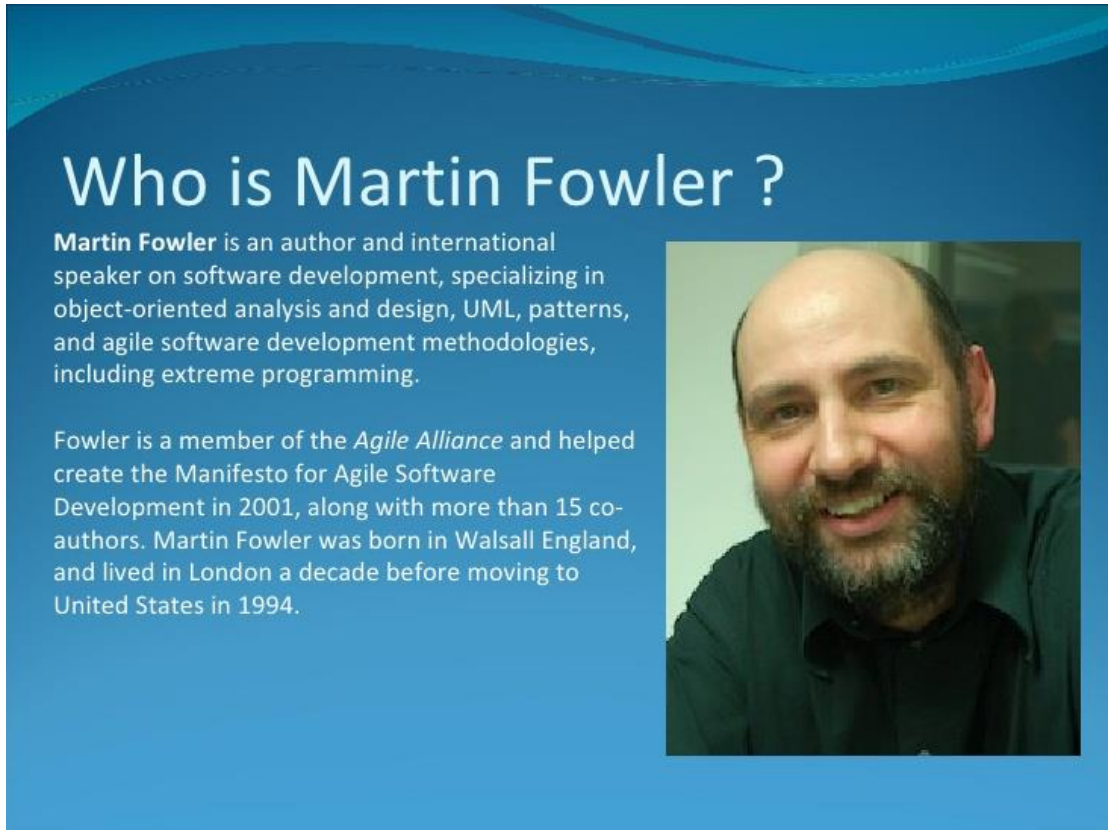
2 Main Commandments of developer



How can you know that code is ugly

Comments

“Comments often are used as a deodorant”



When comments are ok?

- Explain algorithm
- For public API
- Describe variables

When comments are ok?

- ~~Explain algorithm~~
- For public API
- ~~Describe variables~~

Removing the comments

- Rename
  - Fields
  - Constants
  - Methods
  - Input parameters
  - Classes
- Extract method

What's in a name?

"What's in a name?
That which we
call a rose
By any other name
would smell as sweet."

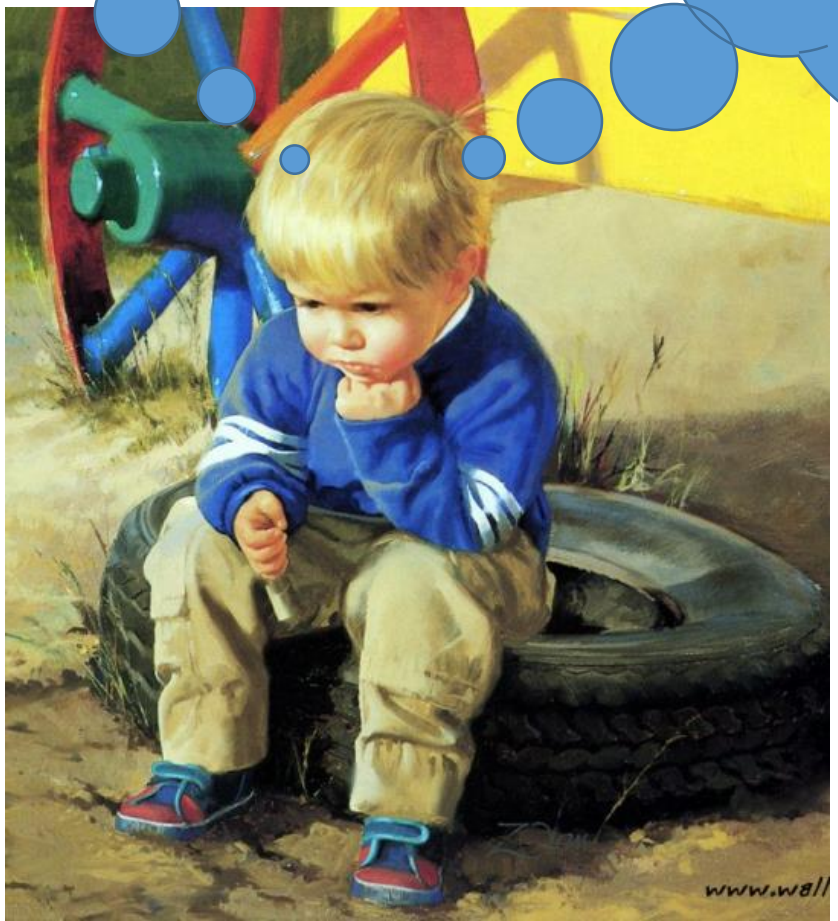
Java conventions in most cases are scala also

- Class starts from UpperCase
- THIS_IS_CONSTANT
- Only here you can use _ long package name(package – lowercased)
- Never use this: _methodVariable=3
- Only class (type) starts from uppercase.
- Use upperCaseToDevideBewtweenWords

Convention examples

- Interface name: PersonService
- Not IPersonService – this is not C#
- Class name: PersonServiceImpl
- Method name: printPersonDetails
- Variable names:
  String name, int age, Person person
- Constant: final int NUMBER_OF_LEGS = 2

Why use conventions?



How many mistakes can you find here?


new balance
NEW
NEW

How to give names

- Make the compiler happy
- Keep java naming convention
- Never use Ifc, use - Impl
- Don’t make spell errors
- Make it pronouncable (`dlgFnEvalItr`)
- Add logical names (Service, Listener, Model…)
- Don’t use abuse words (Class, Object…)
- Be consistent (calcPrice, calculateTime, getCalculatedResult)
- Use alt+ctrl+shift+N before creating new method
- Don’t economize in method or class name length

More code smells

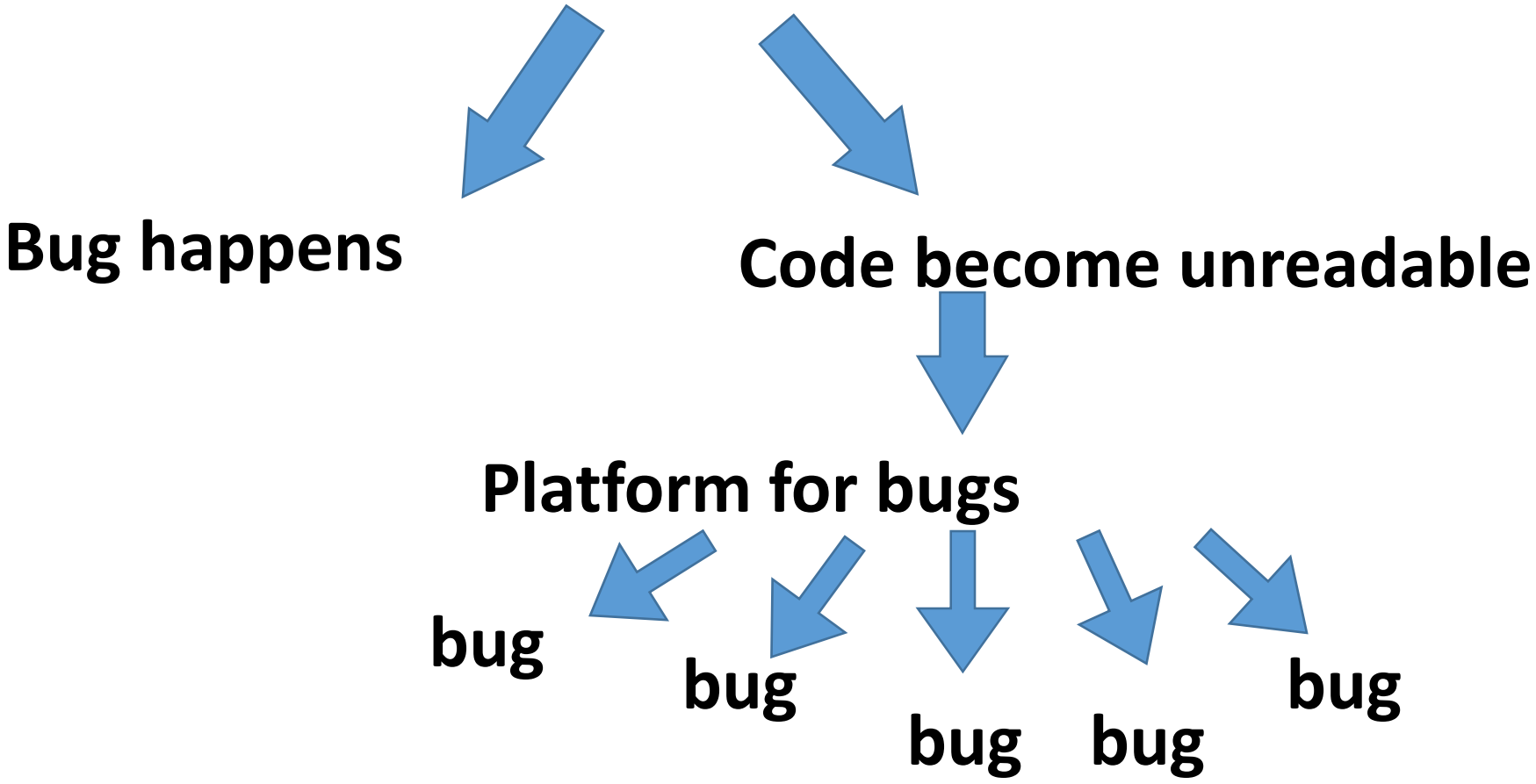
- Long methods

Why people write long methods

- Unrestrained feature growth
- Blind Conformance
- Ignorance
- Fear of “losing the thread”
- Performance Paranoia

About performance

There is a people which think too much about performance…
They try to write custom optimization for the code



http://habrahabr.ru/post/165729/

14 января в 14:03

Предельная производительность: C# из песочницы

Программирование*, Параллельное программирование*, Высокая производительность*

Я поделюсь 30 практиками для достижения максимальной производительности приложений, которые этого требуют. Затем, я расскажу, как применил их для коммерческого продукта и добился небывалых результатов!
Приложение было написано на C# для платформы Windows, работающее с Microsoft SQL Server. Никаких профайлеров – содержание основывается на понимании работы различных технологий, поэтому многие топики пригодятся для других платформ и языков программирования.

Предисловие

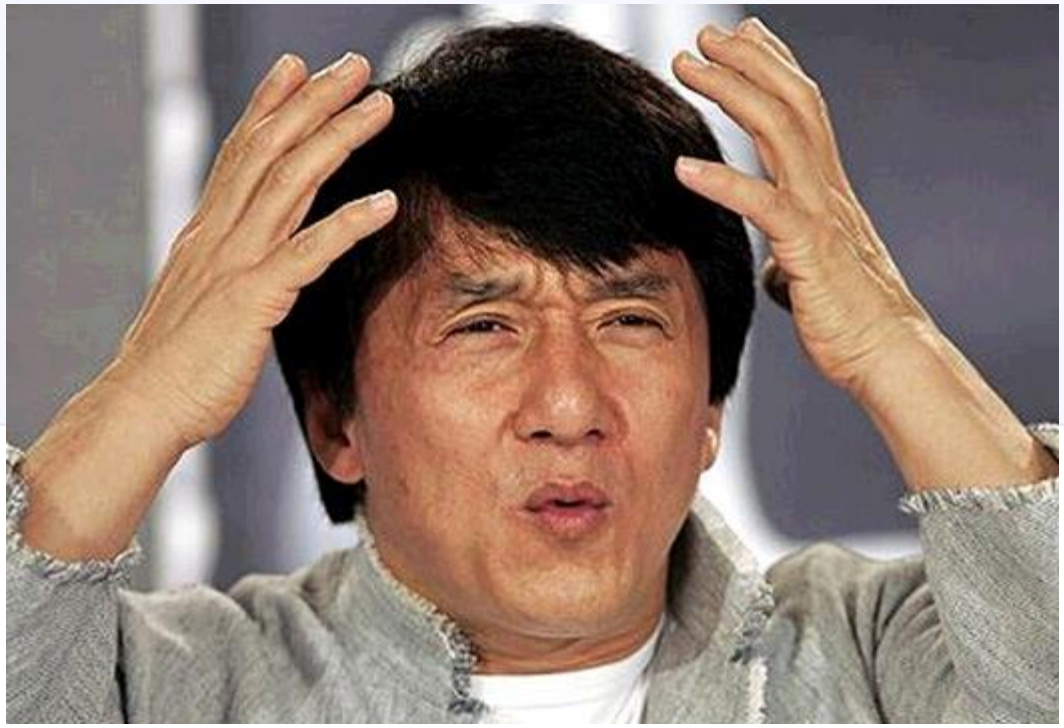
Don’t do that in Java

7. Разматывайте циклы.

Любой цикл – это та же конструкция «if». Если количество итераций небольшое, и их количество заранее известно, то иногда цикл лучше заменить на его тело, которое повторяется необходимое кол-во итераций (да, один раз Copy и N раз Paste).

▾ Пример – возведение числа в 4-ю степень

```
int power4(int v)
{
    int r = 1;
    r *= v;
    r *= v;
    r *= v;
    r *= v;
    return r;
}
```



About HotSpot

What does it mean….. HotSpot

Just-In-Time Compilation

Just-In-Time Compilation

- Everyone knows about JIT!
- Hot code is compiled to native
- What is “hot”?
  - Server VM – 10000 invocations
  - Client VM – 1500 invocations
  - Use -XX:CompileThreshold=# to change
    - More invocations – better optimizations
    - Less invocations – shorter warmup time

HotSpot optimizations

- JIT Compilation
  - Compiler Optimizations
  - Generates more performant code that you could write in native
- Adaptive Optimization
- Split Time Verification

Adaptive Optimization

- Allows HotSpot to uncompile previously compiled code
- Much more aggressive, even speculative optimizations may be performed
- And rolled back if something goes wrong or new data gathered
  - E.g. classloading might invalidate inlining

Two Types of Optimizations

- Java has two compilers:
  - javac bytecode compiler
  - HotSpot VM JIT compiler
- Both implement similar optimizations
- Bytecode compiler is limited
  - Dynamic linking
  - Can apply only static optimizations

Example – Bounds Check Elimination

```
 1 public class GameTest {
 2     @Test
 3     public void testScore() {
 4         Game game = new Game();
 5         GameMove[] gameMoves = {new GameMove("bob", 2), new GameMove("robert", 3)};
 6         int length = gameMoves.length;
 7         for (int i = 0; i < length; i++) {
 8             if (i < 0 || length <= i) throw new ArrayIndexOutOfBoundsException();
 9             game.score(gameMoves[i].getPlayer(), gameMoves[i].getPoints());
10         }
11     }
12 }
```

Loop unrolling

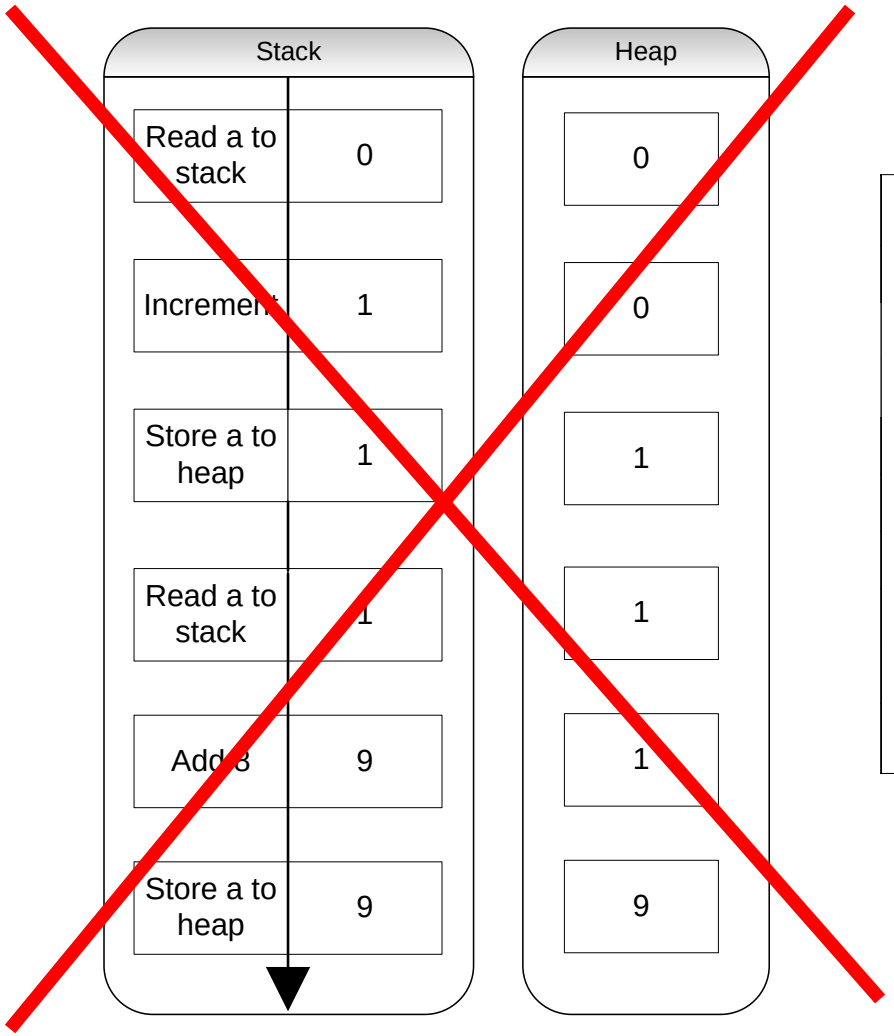
```
 1 public class GameTest {
 2     @Test
 3     public void testScore() {
 4         Game game = new Game();
 5         GameMove[] gameMoves = {new GameMove("bob", 2), new GameMove("robert", 3)};
 6         for (int i = 0; i < gameMoves.length; i++) {
 7             GameMove move = gameMoves[i];
 8             game.score(move.getPlayer(), move.getPoints());
 9         }
10     }
11 }
```

```
 1 public class GameTest {
 2     @Test
 3     public void testScore() {
 4         Game game = new Game();
 5         GameMove[] gameMoves = {new GameMove("bob", 2), new GameMove("robert", 3)};
 6         GameMove move = gameMoves[0];
 7         game.score(move.getPlayer(), move.getPoints());
 8         move = gameMoves[1];
 9         game.score(move.getPlayer(), move.getPoints());
10     }
11 }
```

OSR - On Stack Replacement

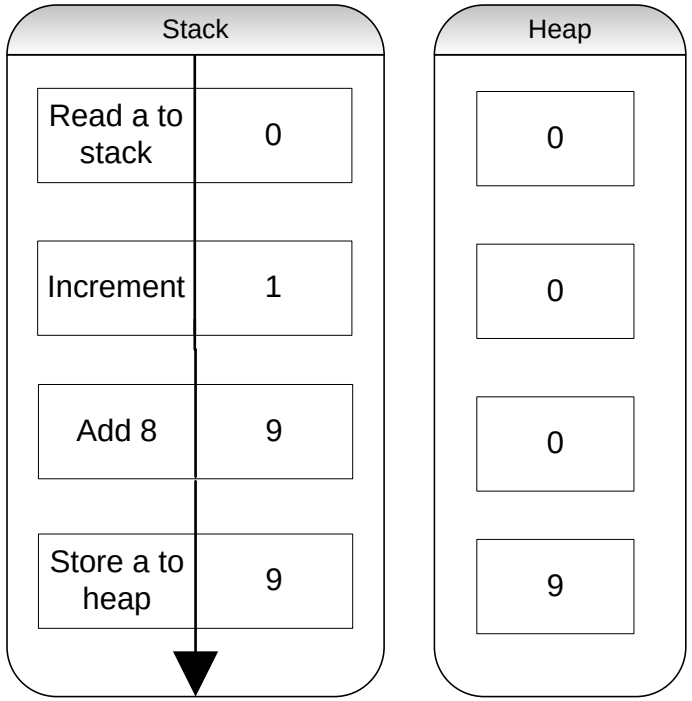
- Normally code is switched from interpretation to native in heap context
  - Before entering method
- OSR - switch from interpretation to compiled code in local context
  - In the middle of a method call
  - JVM tracks code block execution count
- Less optimizations
  - May prevent bound check elimination and loop unrolling

Out-Of-Order Execution




```
 1 public class OutOfOrder {
 2
 3     private int a;
 4
 5     public void foo(){
 6         a++;
 7         a+=8;
 8     }
 9
10 }
```

Out-Of-Order Execution




```
 1 public class OutOfOrder {
 2
 3     private int a;
 4
 5     public void foo(){
 6         a++;
 7         a+=8;
 8     }
 9
10 }
```

Inlining

- Love Encapsulation?
  - Getters and setters
- Love clean and simple code?
  - Small methods
- Use static code analysis?
  - Small methods
- No penalty for using those!
- JIT brings the implementation of these methods into a containing method
  - This optimization known as “Inlining”

Inlining

- Before inline optimization

```
Point point = new Point(3, 4);
PointService pointService = new PointService();
pointService.printPoint(point);
```

- Two additional object, will be created in the heap

Inlining

- After optimization

```
Point point = new Point(3, 4);
System.out.println(point.getX()+" "+point.getY());
```

But the object which was inlined can be used later…

Escape Analysis will solve it

Escape Analysis

- Escape analysis is not optimization
- It is check for object not escaping local scope
  - E.g. created in private method, assigned to local variable and not returned
- Escape analysis opens up possibilities for lots of optimizations

Example - Lock Elision

```
 1 public class GameTest {
 2     @Test
 3     public void testScore() {
 4         Game game = new Game();
 5         lock(game);
 6         game.logger.info("Bob" + " scores " + 5);
 7         game.totalScore += 5;
 8         game.logger.info("Jane" + " scores " + 7);
 9         game.totalScore += 7;
10         game.logger.info("Dwane" + " scores " + 1);
11         game.totalScore += 1;
12         unlock(game);
13     }
14 }
```

Scalar Replacement

```
Point point = new Point(3, 4);
System.out.println(point.getX()+" "+point.getY());
```

```
int x = 3;
int y = 4;
System.out.println(x+" "+y);
```

Constants Folding

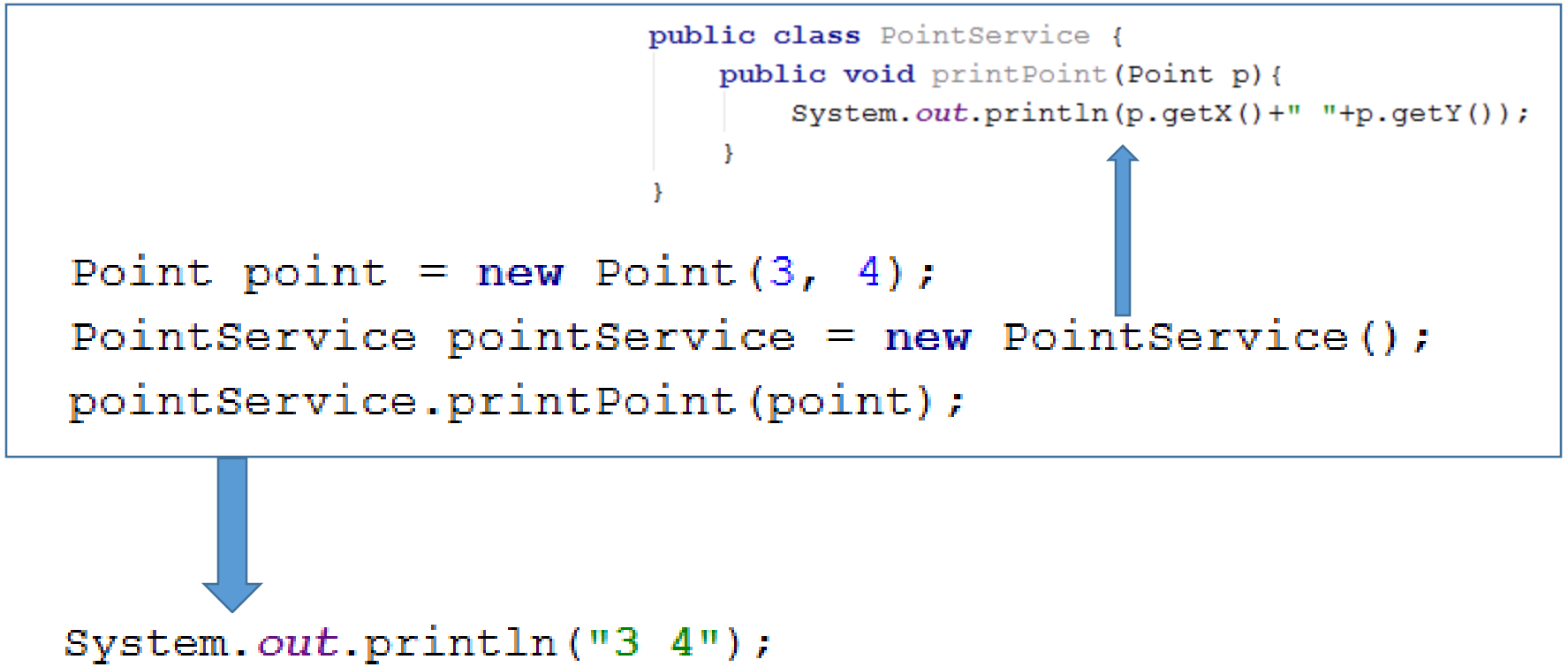
- Literals folding
  - Before: `int foo = 9*10;`
  - After: `int foo = 90;`
- String folding or StringBuilder-ing
  - Before: `String foo = "hi Joe " + (9*10);`
  - After:
```
String foo = new StringBuilder().
append("hi Joe ").
append(9 * 10).toString();
```
  - After: `String foo = "hi Joe 90";`

Constants Folding

```
int x = 3;
int y = 4;
System.out.println(x+" "+y);
```

```
System.out.println("3 4");
```

Finally


public class PointService {
public void printPoint(Point p){
System.out.println(p.getX()+" "+p.getY());
}
}
Point point = new Point(3, 4);
PointService pointService = new PointService();
pointService.printPoint(point);
System.out.println("3 4");

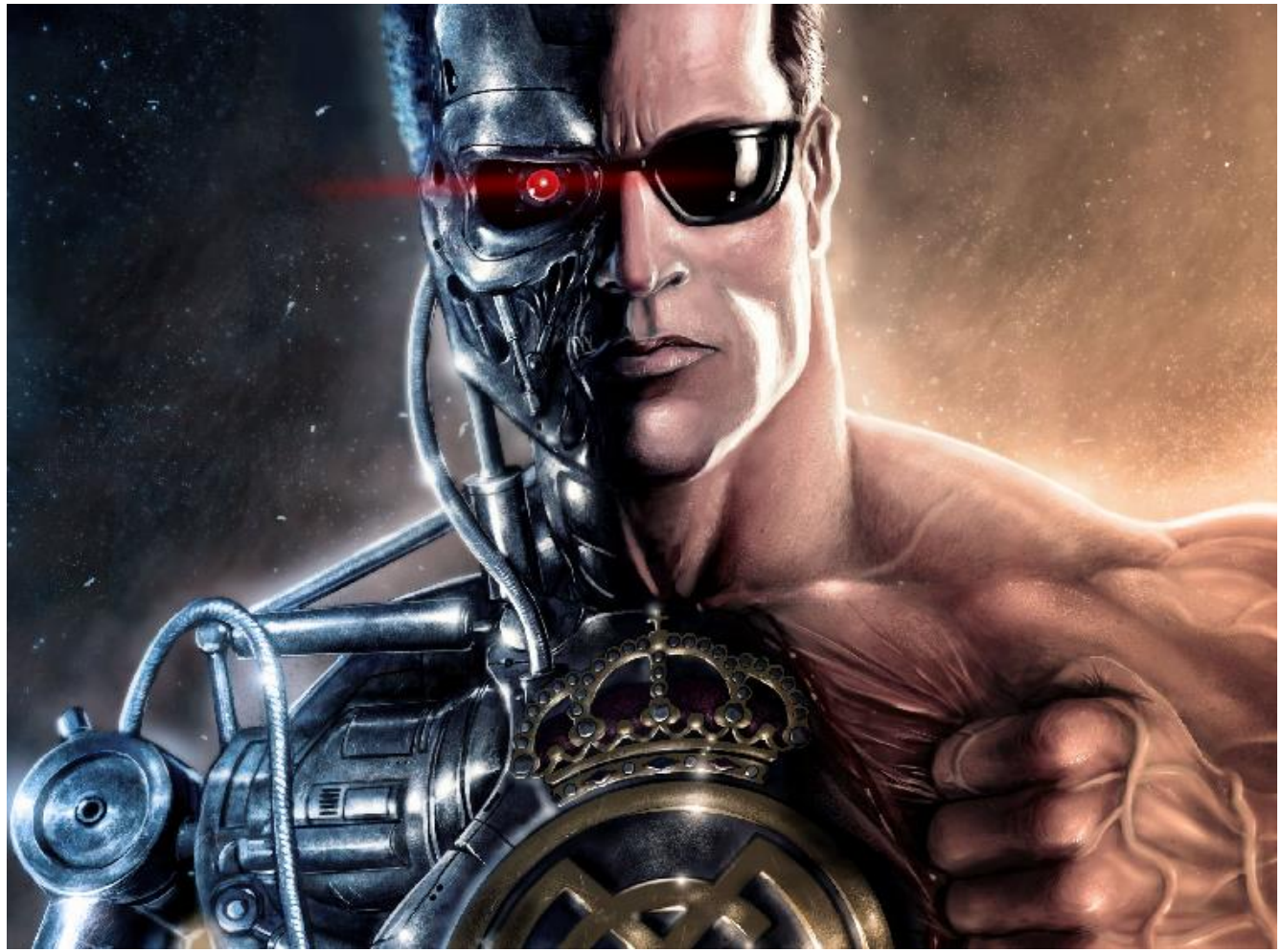
Optimizations is cool! But what can I do?

We need

- Just write good quality Java code
  - Object Orientation
  - Polymorphism
  - Abstraction
  - Encapsulation
  - DRY
  - KISS
  - **Short methods**
- **Let the HotSpot optimize**

So if long methods are bad, what can be done

- Refactor
- But don’t mix abstractions
- Let’s see the example



Method: womanPreparingToGoOut

- Take of the closes
- Enter the bath
- Clean yourself
- Get out from the bath
- Dress yourself
- Open lipstick
- Put lipstick 10 times
- Close lipstick
- Open mascara
- Make up left eye
- Make up right eye
- Look at the mirrow
- Clean up the mascara
- Change mascara color
- …

Method: womanPreparingToGoOut

- Take a shower
- Dress yoursekf
- Make-up
- …

SOLID

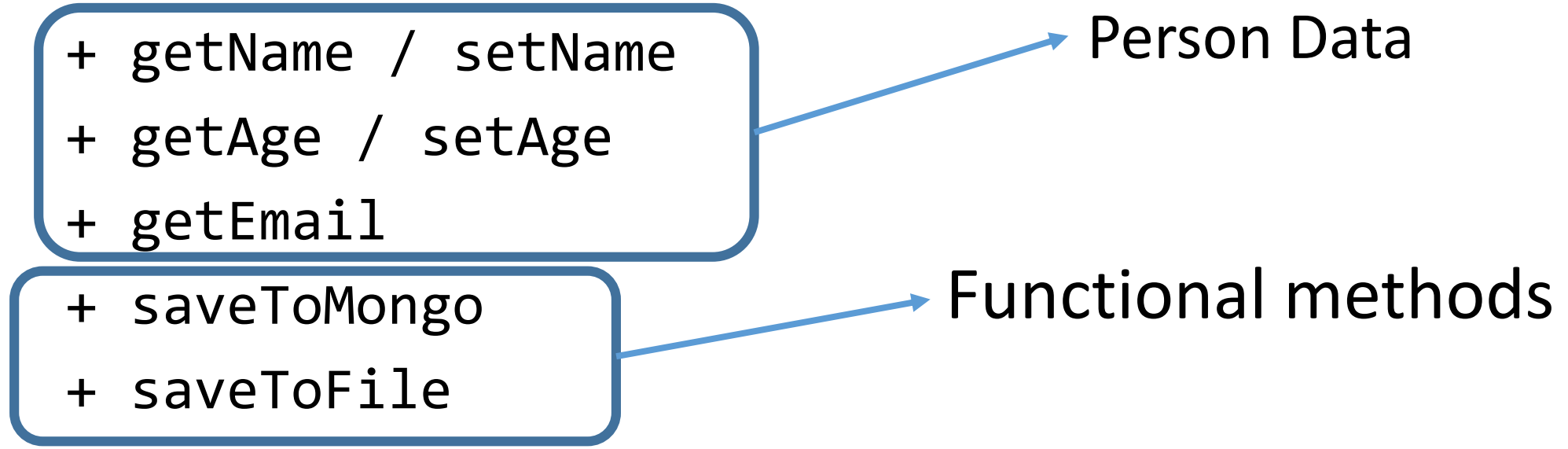
- Single Responsibility Principle
- Open/Close Principle
- Liskov’s Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Single Responsibility

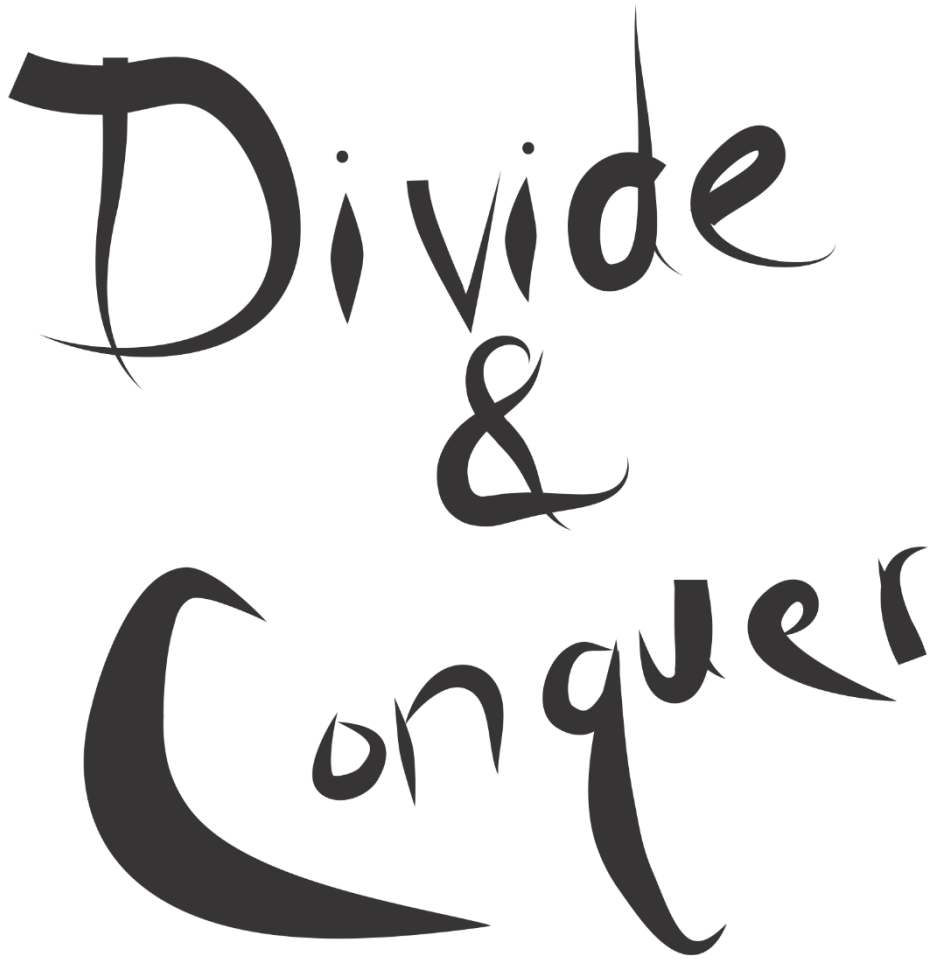
- A class should have only a single responsibility
- There should never be more than one reason for a class to change

Not good class

```
Class Person{
  + getName / setName
  + getAge / setAge
  + getEmail
  + saveToMongo
  + saveToFile
}
```



Single Responsibility



Open Closed Principle

- New features shouldn’t change existing code

Try to improve it

```
class Panel(g:Graphics){


      def drawPoint(p:Point):Unit{
            // draw point using by using g
      }
      drawCircle(circle: Circle):Unit{
            // draw circle using by using g
      }
}
```

Something like

```
class Panel(g:Graphics){

      public void drawShape(shape: Shape){
            shape.draw(g);
       }

}
```

Barbara Liskov

- Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program
- Corollary:
- Subtypes need to be **behaviorally** identical from the callers point of view

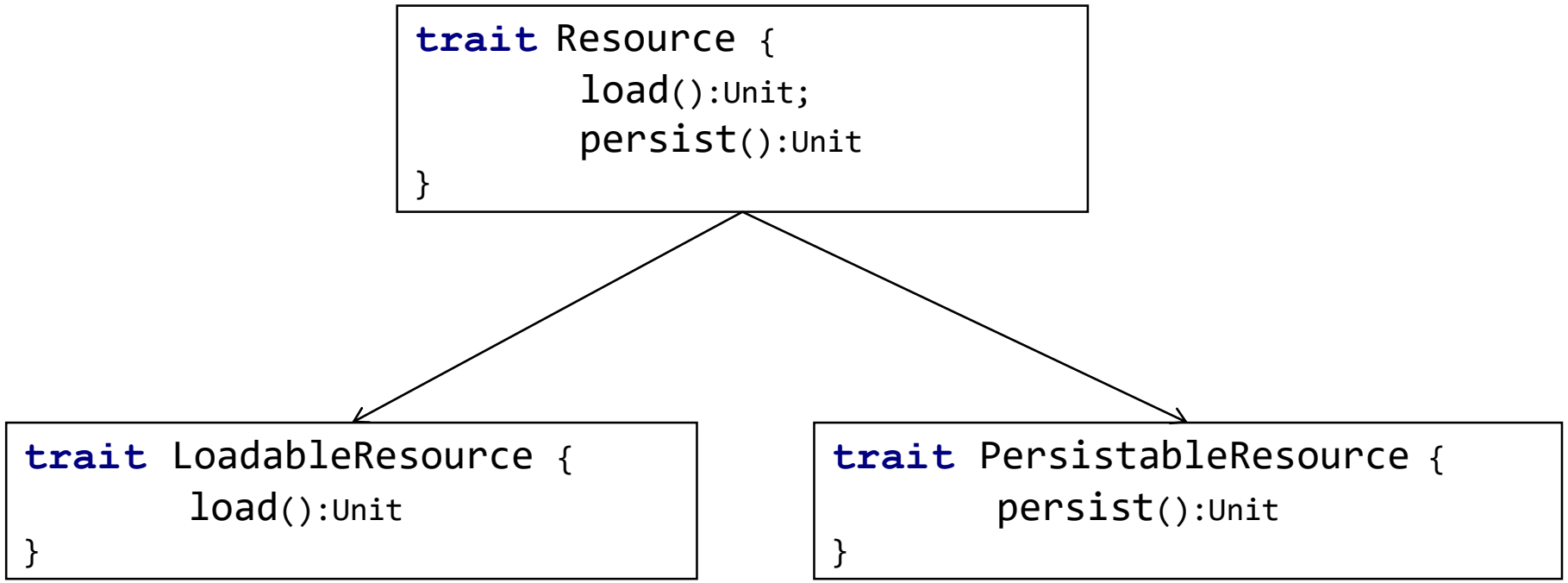
Interface segregation

- Clients should not be forced to depend upon interfaces that they do not use.
- Many client-specific interfaces are better than one general-purpose interface

Example of breaking this principle

```
public class MyMouseListener implements MouseListener {
    public void mouseClicked(MouseEvent e) {
        //To change body of implemented methods use File
    }

    public void mousePressed(MouseEvent e) {
        //To change body of implemented methods use File
    }

    public void mouseReleased(MouseEvent e) {
        //To change body of implemented methods use File
    }

    public void mouseEntered(MouseEvent e) {
        //To change body of implemented methods use File
    }

    public void mouseExited(MouseEvent e) {
        //To change body of implemented methods use File
    }
}
```

```
public class MyMouseListener implements MouseListener {
    public void mouseClicked(MouseEvent e) {
        //some logic
    }

    public void mousePressed(MouseEvent e) {
    }

    public void mouseReleased(MouseEvent e) {
    }

    public void mouseEntered(MouseEvent e) {
    }

    public void mouseExited(MouseEvent e) {
    }
}
```


trait Resource {
load():Unit;
persist():Unit
}
trait LoadableResource {
load():Unit
}
trait PersistableResource {
persist():Unit
}

Dependency inversion

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Corollary
- Modules should interact through abstraction, and have no knowledge of concrete implementation and life cycle

And what if oracle will be changed to Hdfs?

```
class OracleDao{
  def savePerson(person: Person): Unit ={
    // some logic here
  }
}

class PersonService(oracleDao: OracleDao){
  def processPerson(person: Person): Unit ={
    //some logic which process person
    oracleDao.savePerson(person)
  }
}
```

```scala
trait PersonDao {
  def savePerson(person: Person): Unit
}

class OracleDao extends PersonDao {
  def savePerson(person: Person): Unit = {
    // some logic here
  }
}

class HdfsDao extends PersonDao {
  def savePerson(person: Person): Unit = {
    // some logic here
  }
}

class PersonService(personDao: PersonDao) {
  def processPerson(person: Person): Unit = {
    //some logic which process person
    personDao.savePerson(person)
  }
}
```

Why do we need Design Patterns?

- Don’t reinvent the wheel

But don’t try to use them without a reason

Lets go...

Strategy


Аквилон
9459
21
BronzeAge
10
3140
138
0
0
Iggrock
#26
349
Defender
#27
284
Yukki
#28
271
Драугар
#29
264
kandellak
#30
243
DRAKON01
#31
233
Атака
Атака
Атака
Атака
Атака

Strategy – The Problem

- Sometimes parts of entities change frequently
- In those cases inheritance is limiting by being not dynamic enough
- More so, the changes can even break good inheritance principals
- How can we alter the changing parts of the interface rapidly without touching all the rest?

Strategy – Problem example

- Action adventure game development in progress:
  - Character interface has fight() method
    - Good for Knight, Wizard and Troll implementations
  - King impl is added
    - He fights like the Knight, but can’t be its subclass
    - What do you do?
  - Princess impl added
    - She doesn’t fight at all
    - What do you do?

Strategy – The Solution

- Encapsulate what's vary
- Define interface for the changing behavior
- Implement it in different ways
  - Inc. no-op implementation, if needed
- Use the encapsulated algorithms interchangeably

Strategy – Solution Example

```
1 public interface Character {
2
3     void fight();
4 }
```

```
1 public interface FightBehavior {
2
3     void fight();
4 }
```

```
1 public class SwordFightBehavior implements FightBehavior {
2
3     public void fight() {
4         …
5     }
6 }
```

```
1 public class NoFightBehavior implements FightBehavior {
2
3     public void fight() {
4
5     }
6 }
```

```
1 public class Knight implements Character {
2     private FightBehavior swordFightBehavior;
3
4     public Knight() {
5         swordFightBehavior = new SwordFightBehavior();
6     }
7
8     public void fight() {
9         swordFightBehavior.fight();
10
11    }
12 }
```

```
1 public class Princess implements Character {
2
3     FightBehavior noFightBehavior;
4
5     public Princess() {
6         noFightBehavior = new NoFightBehavior();
7     }
8
9     public void fight() {
10        noFightBehavior.fight();
11
12    }
13 }
```

Imports

`import pack1._`
- Imports all classes from **`pack1`**
- Equivalent to Java **`import *.pack1`**

`import pack1._, pack2._`
- Imports all classes from **`pack1`** and from **`pack2`**

`import pack1.Class1`
- Imports only **`Class1`** from **`pack1`**

`import pack1.(Class1, Class3}`
- Imports only **`Class1`** and **`Class3`** from **`pack1`**

More imports

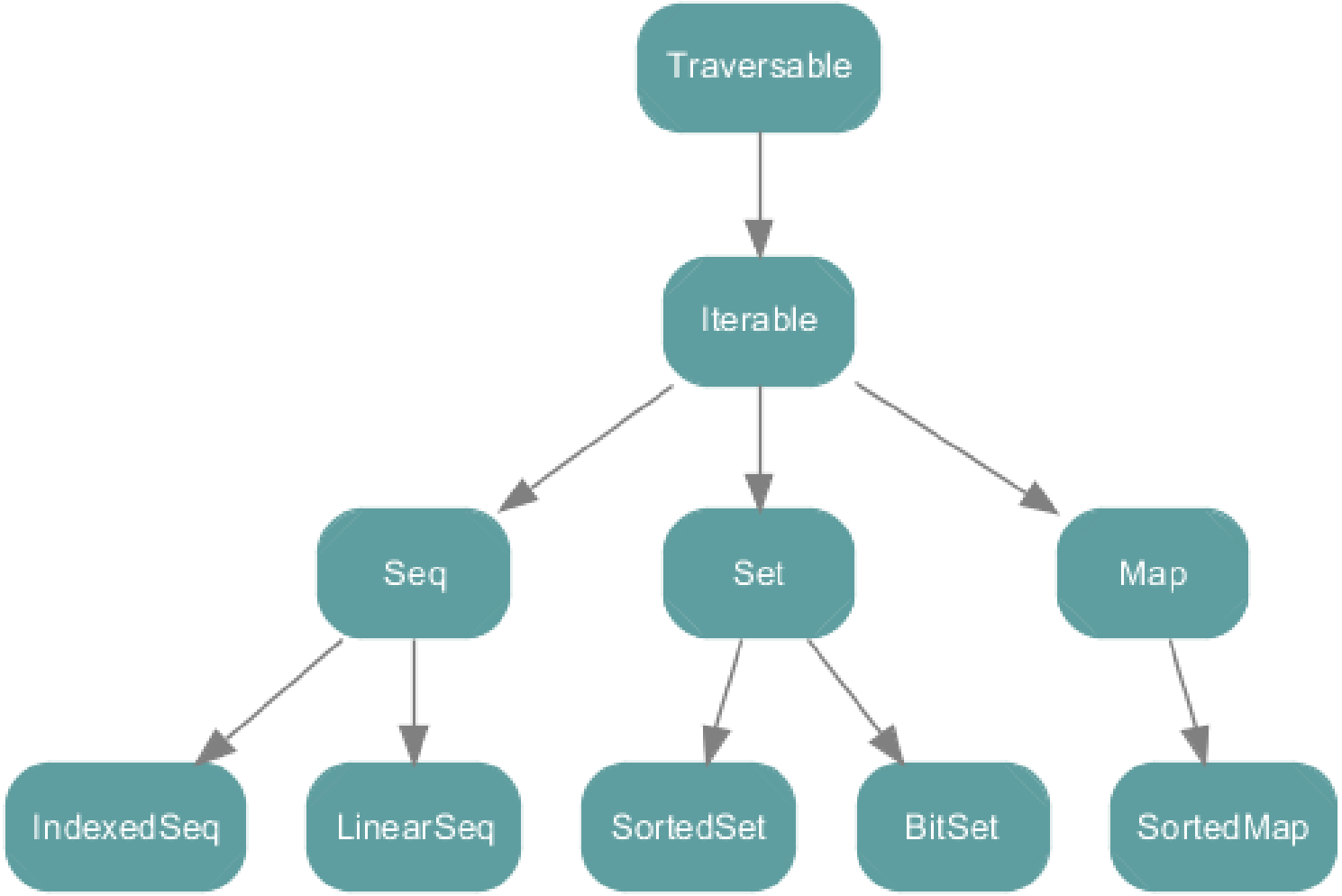
`import pack1.Class1._`

- Imports all members of **`Class1`** *including implicit definitions*
- No equivalent in Java

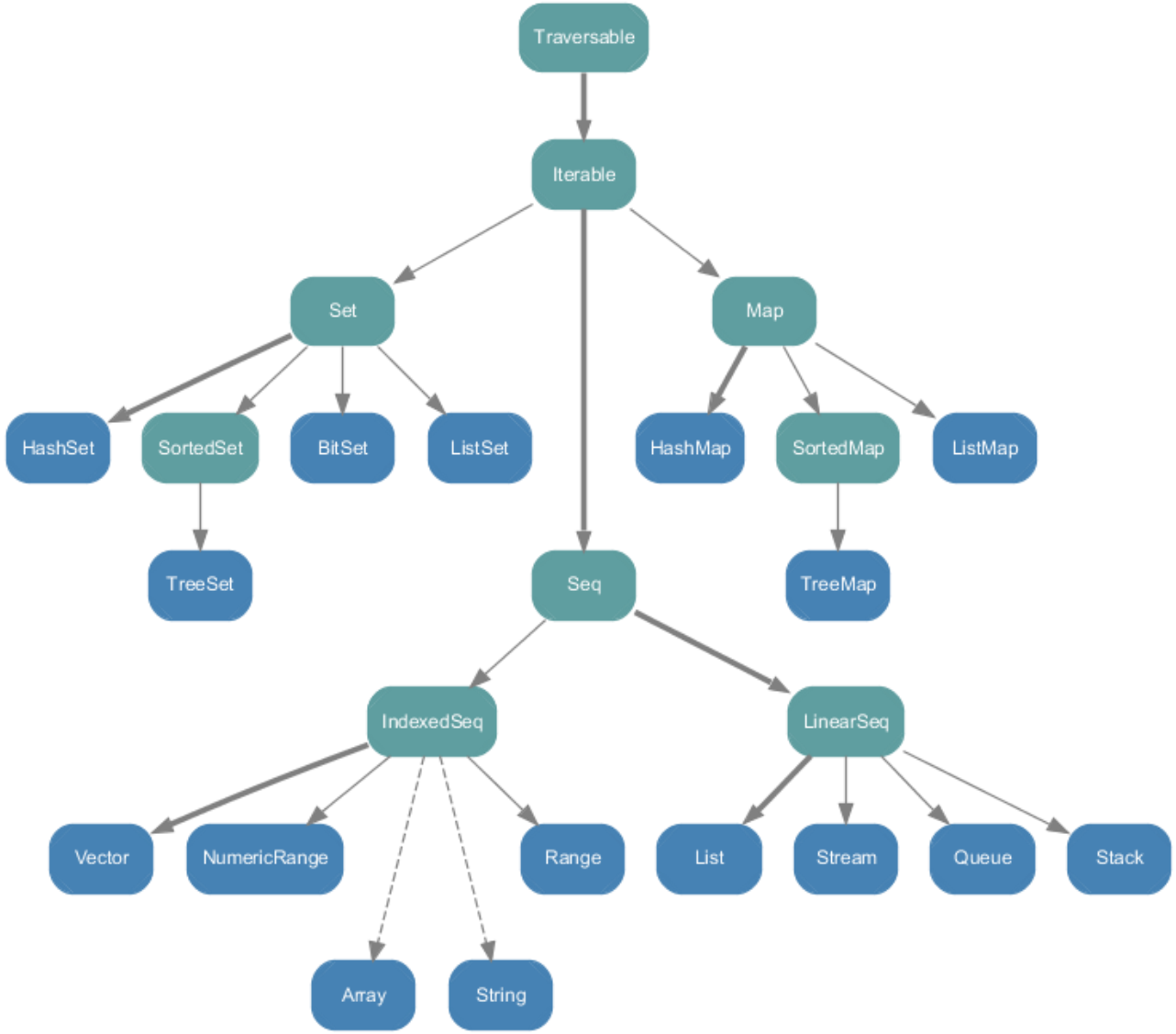
`import pack1.{Class1 => MyClass}`

- Imports **`Class1`** from pack1 and renames it **`MyClass`**
- This avoids collision with an existing function named **`Class1`**

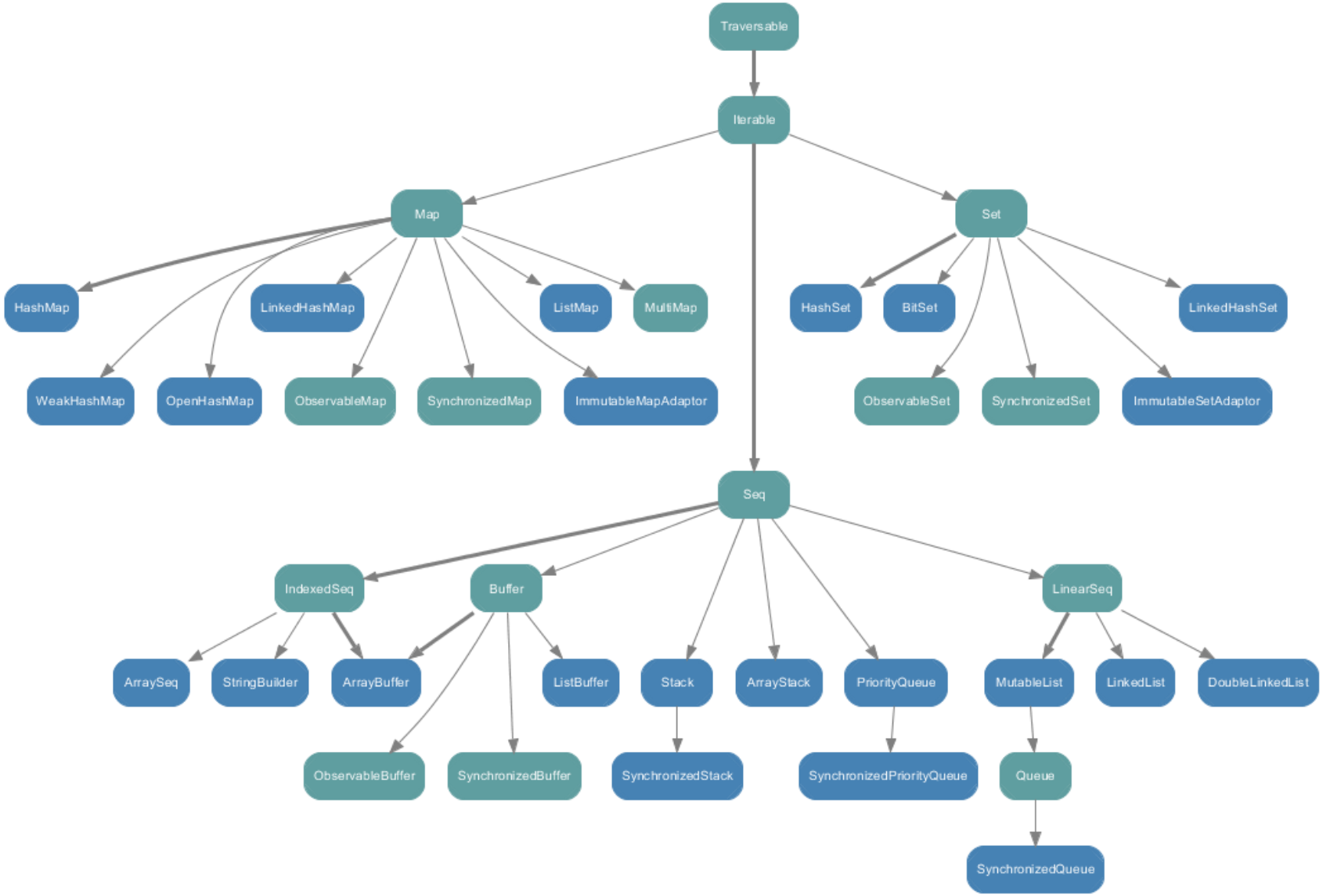
scala.collection



scala.collection.immutable



scala.collection.mutable



Or Use Existing Java Collections

- java.util
- Apache Commons Collections
- fastutil
- Trove
- Google Collections

- scala.collection.JavaConversion available to convert to and from java.util Interfaces

List methods

Simple methods

- head
- isEmpty
- nonEmpty
- size
- :: (append first)
- tail
- drop
- slice
- mkString
- reverse
- toArray / toSet / toSeq…

Functional methods

- filter
- forEach
- map
- flatMap
- reduce

Nil – empty list

Scala is Functional

First Class Functions

```
// Lightweight anonymous functions
(x:Int) => x + 1


// Calling the anonymous function
val plusOne = (x:Int) => x + 1
plusOne(5)  →  6
```

Closures

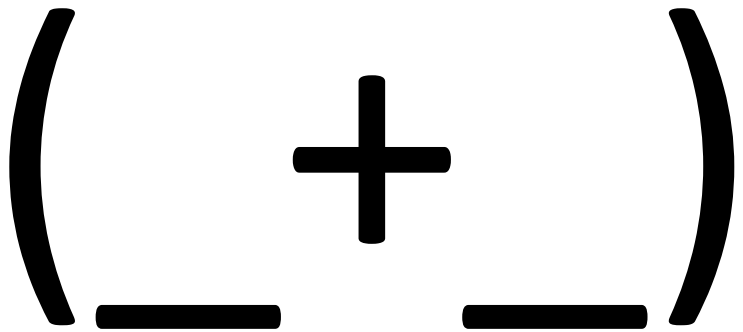
```
// plusFoo can reference any values/variables in
   scope
var foo = 1
val plusFoo = (x:Int) => x + foo

plusFoo(5)   → 6

// Changing foo changes the return value of
   plusFoo
foo = 5
plusFoo(5)   → 10
```

Higher Order Functions

```
val plusOne = (x:Int) => x + 1
var nums = List(1,2,3)

// map takes a function: Int => T
nums = nums.map(plusOne)→List(2,3,4)

// Inline Anonymous
nums.map(x => x + 1)   →   List(2,3,4)

// Short form
nums.map(_ + 1)        →   List(2,3,4)
```

The best Scala Spark operator

The best Scala Spark operator



Higher Order Functions

```
val nums = List(1,2,3,4)

// A few more examples for List class
nums.exists(_ == 2)          →  true
nums.find(_ == 2)            →  Some(2)
nums.indexWhere(_ == 2)      →  1
nums.reduceLeft(_ + _)       →  10
nums.foldLeft(100)(_ + _)    →  110

// Many more in collections library
```

Higher Order Functions

```
// functions as parameters
def call(f: Int => Int) = f(1)


call(plusOne)      → 2
call(x => x + 1)   → 2
call(_ + 1)        → 2
```

Higher Order Functions

```
// functions as parameters
def call(f: Int => Int) = f(1)


call(plusOne)       → 2
call(x => x + 1)    → 2
call(_ + 1)         → 2
```

Higher Order Functions

```
// functions as parameters
def each(xs: List[Int], fun: Int => Unit) {
  if(!xs.isEmpty) {
    fun(xs.head)
    each(xs.tail, fun)
  }
}

each(List(1,2,3), println)
    → 1
    → 2
    → 3
```

Higher Order Functions

```
// More complex example with generics & pattern matching

@tailrec
def each[T](xs: List[T], fun: T => Unit): Unit = xs match {
  case Nil =>
  case head :: tail => fun(head); each(tail, fun)
}

each(List(1,2), println)
    → 1
    → 2

each(List("foo", "bar"), println)
    → foo
    → bar
```

Pattern Matching

```
while (true) {
  val message: String = JOptionPane.showInputDialog(null, "what would you say")
  var answer: String = ""
  message match {
    case "hello" => answer = "hello";
    case "how are you" => answer = "better than you";
    case "hi" => answer = "hi!";
  }
  println(answer)
}
```

What if man asks: “what is your name?”


IF MAN ASKS A GIRL WHAT IS HER NAME


IF MAN ASKS A GIRL WHAT IS HER NAME
THE GIRL WILL THROW AN EXCEPTION

```
while (true) {
  val message: String = JOptionPane.showInputDialog(null, "what would you say")
  var answer: String = ""
  message match {
    case "hello" => answer = "hello";
    case "how are you" => answer = "better than you";
    case "hi" => answer = "hi!";
    case default => answer = "default answer!";
  }
  println(answer)
}
```

```
while (true) {
  val message: String = JOptionPane.showInputDialog(null, "what would you say")
  var answer: String = ""
  message match {
    case "hello" => answer = “hello";
    case "how are you" => answer = “better than you";
    case "hi" => answer = “hi!";
    case _ => answer = “default answer!";
  }
  println(answer)
}
```

Pattern Matching

```
def what(any:Any) = any match {
  case i:Int => "It's an Int"
  case s:String => "It's a String"
  case _=> "I don't know what it is:“ + _
}
```

```
what(123)      →  "It's an Int"
what("hello") →  "It's a String"
what(false)    →  "I don't know what it is"
```

Pattern Matching

```
val nums = List(1,2,3)

// Pattern matching to create 3 vals
val List(a,b,c) = nums
```

a → 1
b → 2
c → 3

Pattern Matching

```
val x = List(1,2,3,4,5) match {
  case a::2::4::Y =>a
  case Nil => 42
  case a::b::3::4:: _ => a + b
  case h::t=> h
  case _ => 101  // unreachable statement
}
```

x will be ?

Pattern Matching

```
val x = List(1,2,3,4,5) match {
  case a::2::4::Y =>a
  case Nil => 42
  case a::b::3::4:: _ => a + b
  case h::t=> h
  case _ => 101  // unreachable statement
}
```

x will be **3**

Immutable Types

```
// Immutable types by default
var nums = Set(1,2,3)
nums += 4   →   nums = nums.+(4)

// Mutable types available
import scala.collection.mutable._

val nums = Set(1,2,3)
nums += 4   →   nums.+=(4)
```

Scala is Dynamic

(Okay not really, but it has lots of features typically only found in Dynamic languages)

Scriptable

```
// HelloWorld.scala
println("Hello World")

bash$ scala HelloWorld.scala
Hello World

bash$ scala -e 'println("Hello World")'
Hello World
```

Read-**E**val-**P**rint **L**oop

```
bash$ scala
Welcome to Scala version 2.8.1.final (Java
   HotSpot(TM) 64-Bit Server VM, Java 1.6.0_22).
Type in expressions to have them evaluated.
Type :help for more information.

scala> class Foo { def bar = "baz" }
defined class Foo

scala> val f = new Foo
f: Foo = Foo@51707653
```

Structural Typing

```
// Type safe Duck Typing
def doTalk(any:{def talk:String}) {
  println(any.talk)
}

class Duck { def talk = "Quack" }
class Dog  { def talk = "Bark"  }

doTalk(new Duck)  →  "Quack"
doTalk(new Dog)   →  "Bark"
```

type => declaration of functional interfaces. Java 8 style

Implicit Conversions

```
// Extend existing classes in a type safe way

// Goal: Add isBlank method to String class
class RichString(s:String) {
  def isBlank = null == s || "" == s.trim
}

implicit def toRichString(s:String) = new
  RichString(s)


// Our isBlank method is now available on Strings
" ".isBlank    →  true
"foo".isBlank  →  false
```

method_missing

```
// Dynamic is a marker trait used by the compiler
class Foo extends Dynamic {
  def typed[T] = error("not implemented")

  def applyDynamic(name:String)(args:Any*) = {
    println("called:
  "+name+"("+args.mkString(",")+")")
  }

}

val f = new Foo
f.helloWorld       →  called: helloWorld()
f.hello("world")   →  called: hello(world)
f.bar(1,2,3)       →  called: bar(1,2,3)
```

Scala has tons of other cool stuff

Default Parameter Values

```
def hello(foo:Int = 0, bar:Int = 0) {
  println("foo: "+foo+"  bar: "+bar)
}

hello()          →  foo: 0  bar: 0
hello(1)         →  foo: 1  bar: 0
hello(1,2)       →  foo: 1  bar: 2
```

Named Parameters

```
def hello(foo:Int = 0, bar:Int = 0) {
  println("foo: "+foo+"  bar: "+bar)
}

hello(bar=6)          →  foo: 0  bar: 6
hello(foo=7)          →  foo: 7  bar: 0
hello(foo=8,bar=9)    →  foo: 8  bar: 9
```

Everything Returns a Value

```
val a = if(true) "yes" else "no"

val b = try{
  "foo"
} catch {
  case _ => "error"
}

val c = {
  println("hello")
  "foo"
}
```

Lazy Vals

```
// initialized on first access
lazy val foo = {
  println("init")
  "bar"
}
```

```
foo  →  init
foo  →
foo  →
```

Nested Functions

```
// Can nest multiple levels of functions
def outer() {
    var msg = "foo"
    def one() {
        def two() {
            def three() {
                println(msg)
            }
            three()
        }
        two()
    }
    one()
}
```

By-Name Parameters

```
// x parameter automatically wrapped in closure
def log(doLog:Boolean, x: => Int) {
  if(doLog) {
    print(x)  // evaluates x
  }
}
log(true,1/0)
```

This will fail!

By-Name Parameters

```
// x parameter automatically wrapped in closure
def log(doLog:Boolean, x: => Int) {
  if(doLog) {
    print(x)  // evaluates x
  }
}
log(false,1/0)
```

This will not fail!

By-Name Parameters

```
// x parameter automatically wrapped in closure
def log(doLog:Boolean, x:Int) {
  if(doLog) {
    print(x)  // evaluates x
  }
}
log(false,1/0)   /   log(false,1/0)
```

This will fail in both cases, because it is not lazy!

Many More Features

- **Actors**
- **Annotations** → `@foo def hello = "world"`
- **Case Classes** → `case class Foo(bar:String)`
- **Currying** → `def foo(a:Int,b:Boolean)(c:String)`
- **For Comprehensions**
  → `for(i <- 1.to(5) if i % 2 == 0) yield i`
- **Generics** → `class Foo[T](bar:T)`
- **Package Objects**
- **Partially Applied Functions**
- **Tuples** → `val t = (1,"foo","bar")`
- **Type Specialization**
- **XML Literals** → `val node = <hello>world</hello>`
- **etc…**

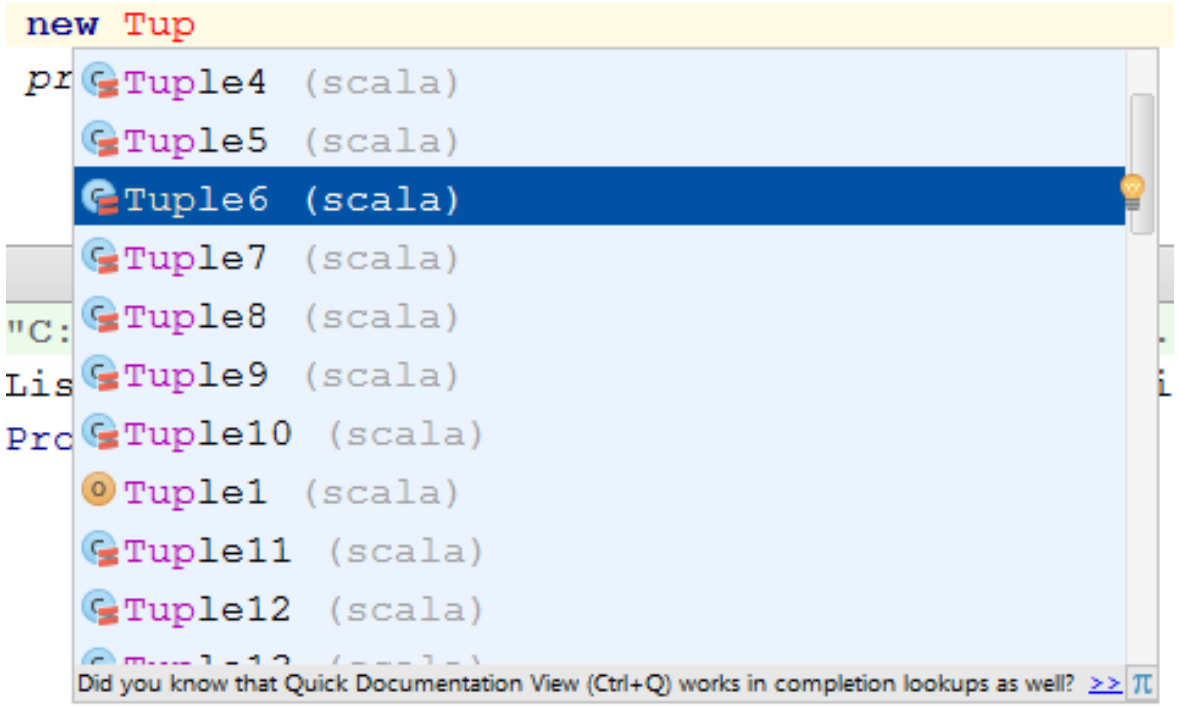
~~Pair~~ Tuples

```
val tuple1: (Int, String) = (1,"java")

val tuple2: (Int, String) = (40,"scala")
val tuple3: (Int, String) = (20,"groovy")
```

```
tuple1 = 1 -> "java"
```

```
val tuple: (LocalDateTime, String, Object) = (LocalDateTime.now(),"beanName",bean)

var tuples: List[(LocalDateTime, String, Object)] = List(tuple)
tuples = (LocalDateTime.now(), "beanName", bean) :: tuples
```

Tuple – till 22



Taking from Tupl-a

```
val dateTime: LocalDateTime = tuple._1
val beanName: String = tuple._2
val bean: Object = tuple._3
```

```
tuples.foreach(tupple=>println(tuple._1))
```

More about Tuple?

- swap

```
var tuple1: (Int, String) = (1, "java")
val swap: (String, Int) = tuple1.swap
```

- Only in case tuple2

You will never guess

```
val tuple: (LocalDateTime, String, Object) = (LocalDateTime.now(), "beanName", bean)
val prefix: String = tuple.productPrefix //prefix = Tuple3
val arity: Int = tuple.productArity  // arity = 3
```

Map – are constructed of Tuple2

```
var intToString1: Map[Int, String] = Map(1 -> "one", 2 -> "two")
var intToString2: Map[Int, String] = Map((3, "tree"), (4, "four"))
println(intToString1(1))
```

- keys, values, contains…

www.scala-lang.org

Scala & Spring - integrating together

- Spring works in runtime against bytecode, so logically it should work correctly

Important rules

- Use constructor injection
- When injecting to list or map, use Java type (java.util.List)
- If you need to switch from java to scala collection use:

```
import scala.collection.JavaConverters._
```

  than you can use asScala on any Java collection

- Injection to scala object will not work

Lookup for bean

```
context.getBean(classOf[MyService])
```

```
context.getBean("enricher").asInstanceOf[EnrichmentatorAggregator]
```

Scala spring labs

- Quoters
- Validators
- Enrichmentators