Is Java 8 a Scala killer?

Urs Peter
upeter@xebia.com
@urs_peter

Xebia

http://bit.ly/xc_is

Some Facts about Scala

Born in 2003 – 10 years old

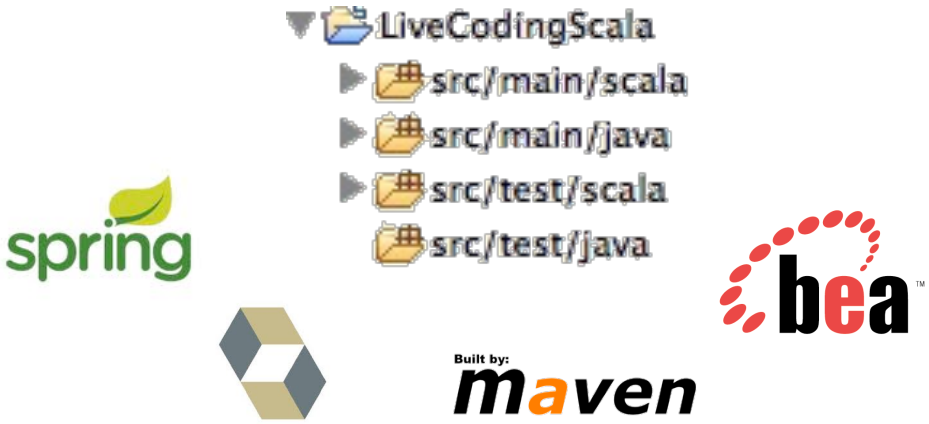
Sponsored by EPFL
(École Polytechnique Fédérale de Lausanne – Switzerland)

Some Facts about Scala

Creator: Martin Odersky

Also co-designed Generic Java,
and built the current generation of javac.

Some Facts about Scala



Scala is fully interoperable with Java

Some Facts about Scala

```
Welcome to Scala version 2.10.0 (Java
HotSpot(TM) 64-Bit Server VM, Java
1.6.0_35).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

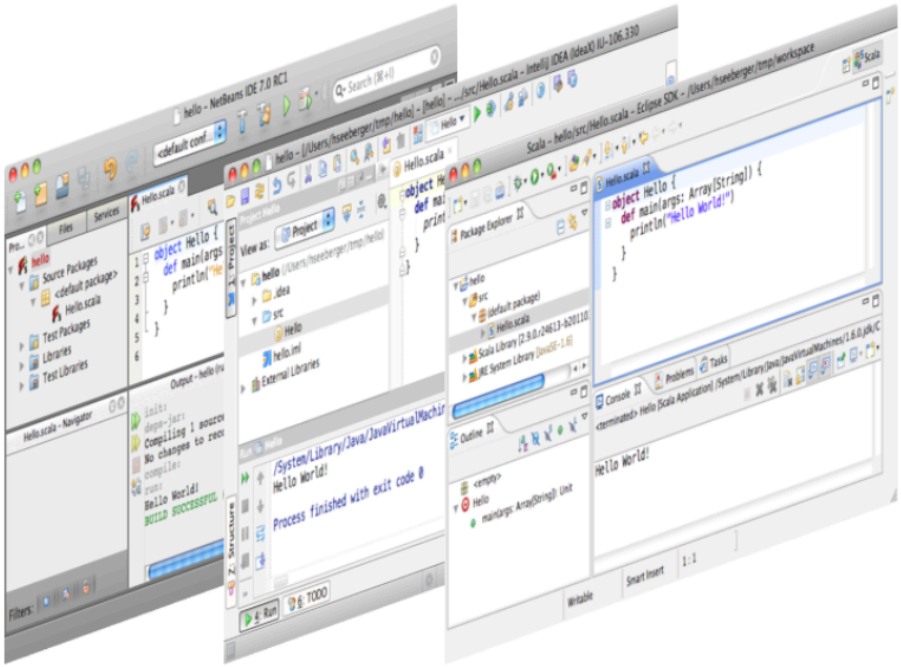
Scala has a REPL
(Command line tool for Read-Eval-Print Loop)

Some Facts about Scala

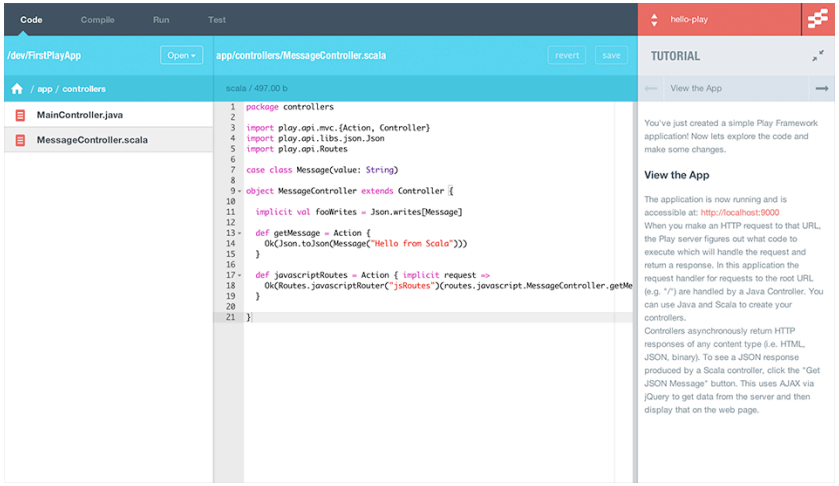
Scala is used in many worldclass companies & applications

Some Facts about Scala

IDE's are mature



…recently added:



web-IDE Typesafe Activator

Some Facts about Scala

There is a company behind Scala

Some Facts about Scala

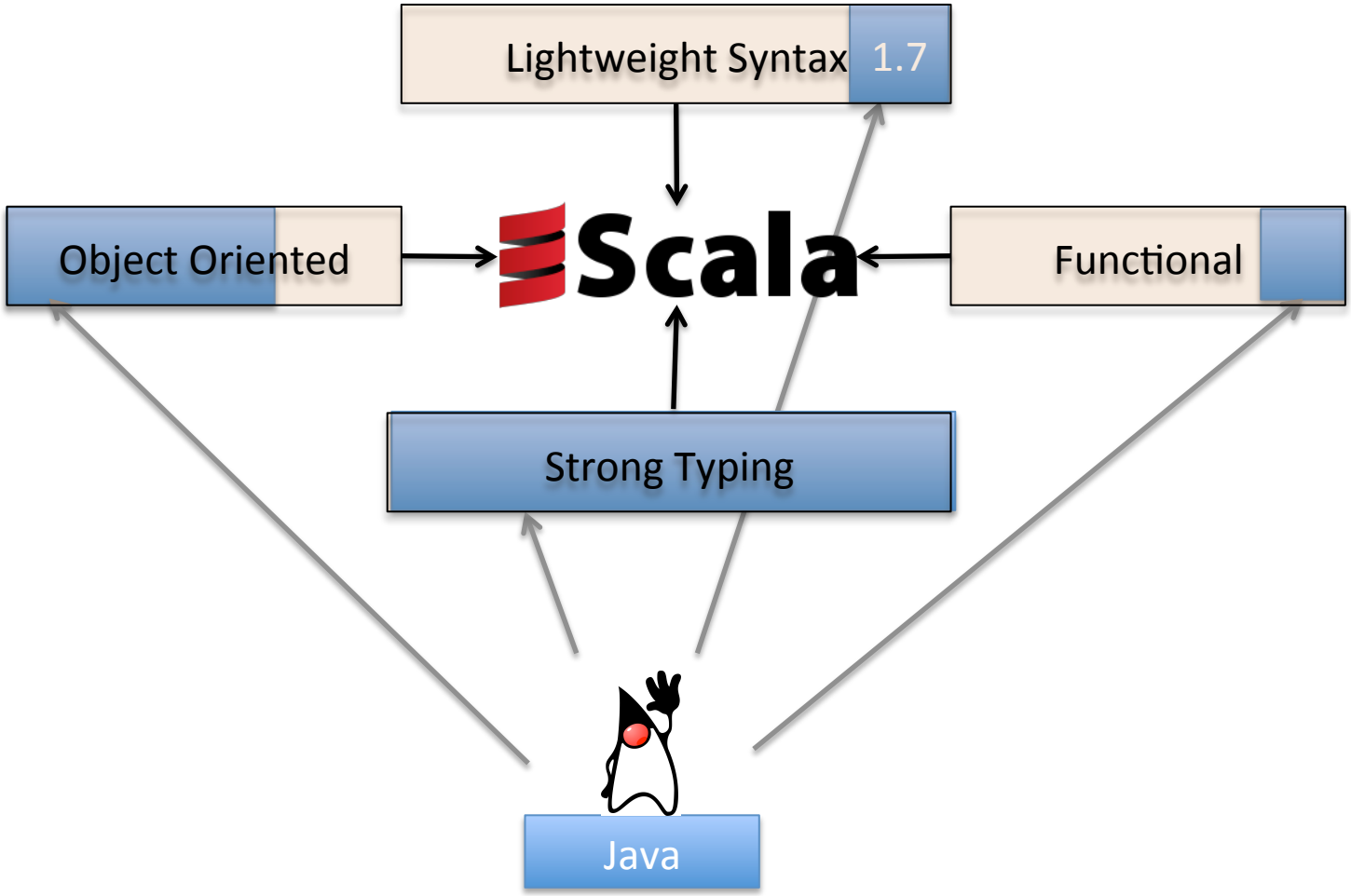
…that offers a supported Stack

Scala. Akka. Play.

Some Facts about Scala

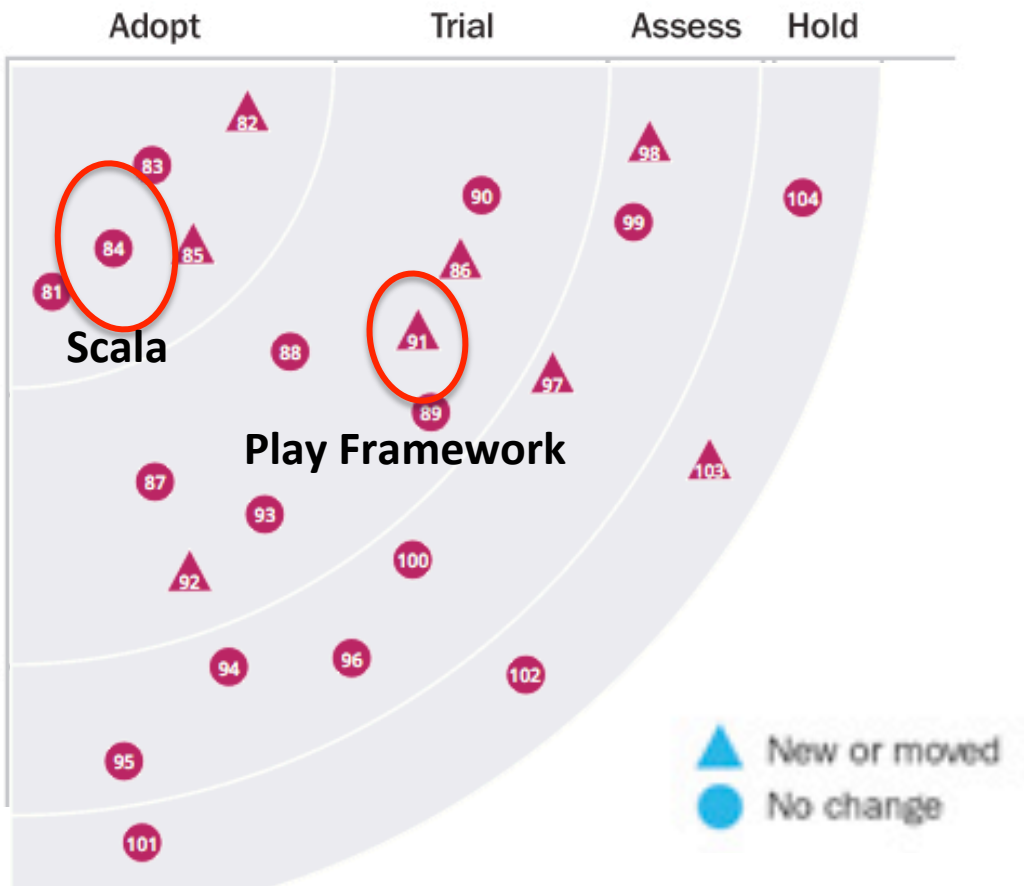
Recently joined Typesafe:

Some Facts about Scala



Some Facts about Scala

Thoughtworks Technology Radar **May 2013**



Java 8

- JSRs
  - JSR 335 Lambda Expressions and Virtual Extension Methods (in debate since 2008)
  - JSR 308 Annotations on Java Types
  - JSR 310 Date and Time API

Java 8

- JSRs
  - **JSR 335 Lambda Expressions and Virtual Extension Methods (in debate since 2008)**
  - JSR 308 Annotations on Java Types
  - JSR 310 Date and Time API

Java∞

- JSRs
  - **JSR 335 Lambda Expressions and Virtual Extension Methods (in debate since 2008)**
  - JSR 308 Annotations on Java Types
  - JSR 310 Date and Time API
- Schedule:
  - 2013/02/21 M7
  - ~~2013/07/05 M8 (Final Release Candidate)~~
  - ~~2013/09/09 GA (General Availability)~~
  - Spring 2014

Java 8 Lambda's versus Scala Functions

What's so fun about Functional Programming anyway?

From Imperative to Functional in Java

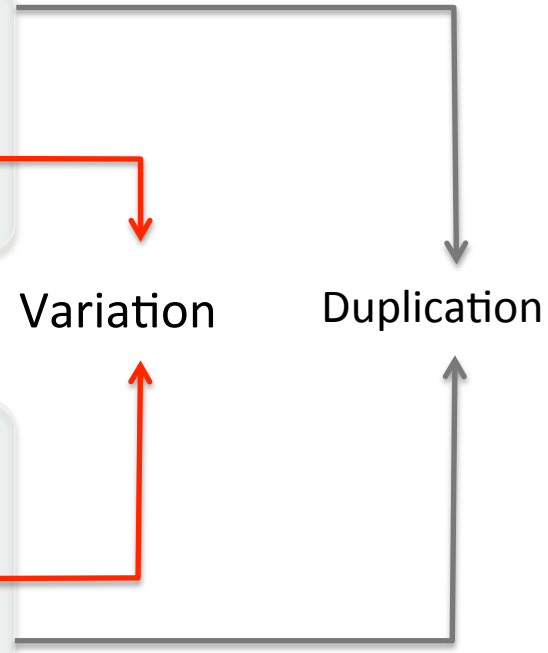
```
List<Integer> numbers = Arrays.asList(1,2,3);
```

Filter even numbers

```
List<Integer> res = new ArrayList<>();
for (Integer i : numbers) {
  if( i % 2 == 0 ) res.add(i);
}
return res;
```

Filter odd numbers

```
List<Integer> res = new ArrayList<>();
for (Integer i : numbers) {
  if( i % 2 != 0 ) res.add(i);
}
return res;
```



From Imperative to Functional in Java

```
List<Integer> res = new ArrayList<>();
for (Integer i : numbers) {
  if( i % 2 == 0 ) res.add(i);
}
return res;
```

```
interface Predicate<T> {
   public boolean op(T item);
}
```

The Function: Varying computation

```
public List<T> filter(List<T> items, Predicate<? super T> predicate) {
  List<T> res = new ArrayList<>();
  for(T item : items) {
     if(predicate.op(item)){
        res.add(item);
     }
  }
  return res;
}
```

The Loop: Generic Control Structure

Generic Control Structure
==
Higher Order Function

From Imperative to Functional in Java

```
List<Integer> numbers = Arrays.asList(1, 2, 3);
```

Filter even numbers

```
List<Integer> result = filter(numbers, new Predicate<Integer>() {
   public boolean op(Integer item) {
     return item % 2 == 0;
   }
});
```

Filter odd numbers

```
List<Integer> result = filter(numbers, new Predicate<Integer>() {
   public boolean op(Integer item) {
     return item % 2 != 0;
   }
});
```

Is the Functional Concept new in Java?

Take Spring:

```
jdbcTemplate.queryForObject("select * from student where id = ?",
  new Object[]{1212l},
  new RowMapper() {
    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new Student(rs.getString("name"), rs.getInt("age"));
    }
});
```

Take Google Guava:

Can you see it?

```
Iterables.filter(persons, new Predicate<Person>() {
    public boolean apply(Person p) {
        return p.getAge() > 18;
    }
});
```

Functional Programming with Lambda's

Before (< Java 8)

```
List<Integer> result = filter(numbers, new Predicate<Integer>() {
                                 public boolean op(Integer i) {
                                   return i % 2 == 0;
                                 }
                               });
```

Lambda Expression
==
Syntactic sugar

After (>= Java 8)

```
List<Integer> result = filter(numbers, (Integer i) -> i % 2 == 0 );
```

Lambda Expressions can be used for '**Functional Interfaces**' (here `Predicate`) that have a single method (here: `boolean op(T f)`).

Lambda Features at one Glance

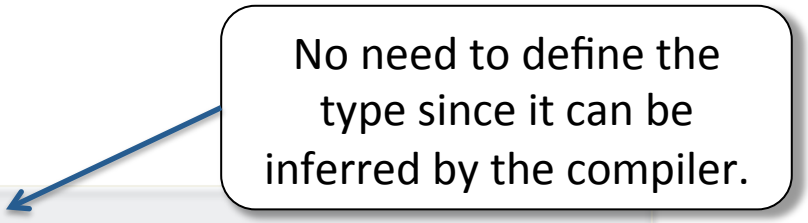
Java 8

Complete Lambda Expression

```
List<Integer> result = filter(numbers, (Integer i) -> i % 2 == 0);
```

Lambda Expression with Type Inference

```
List<Integer> result = filter(numbers, i -> i % 2 == 0);
```



Method Reference at one Glance

Java 8

Static Method References

```
public class NumUtils {
     public static boolean isEven(Integer i) {
          return i % 2 == 0;
      }
}
List<Integer> result = filter(numbers, NumUtils::isEven );
```

Turn an existing static method with the same signature as the functional interface into a closure, using ::

```
(Integer i) -> NumUtils.isEven(i)
```

Instance Method References

```
List<Boolean> trueOrFalse = Arrays.asList(true, false);
List<Boolean> trueOnly = filter(trueOrFalse, Boolean::booleanValue );
```

Boolean::booleanValue refers to the instance that the filter is iterating through.

```
(Boolean b) -> b.booleanValue()
```

Java 8 Lambda Expressions versus Scala Functions

Complete Lambda Expression

Java 8

```
List<Integer> result = filter(numbers, (Integer i) -> i % 2 == 0);
```

Scala

```
val result = filter(numbers, (i:Int) => i % 2 == 0)
```

Java 8 Lambda Expressions versus Scala Functions

Lambda with type inference

Java 8

```
List<Integer> result = filter(numbers, i -> i % 2 == 0);
```

Scala

```
val result = filter(numbers, i => i % 2 == 0)
```

Java 8 Lambda Expressions versus Scala Functions

Static Method References

Java 8

```
List<Integer> result = filter(numbers, NumUtils::isEven);
```

Scala

```
val result = filter(numbers, NumUtils.isEven)
```

Every method that is not called with its arguments is automatically turned into a Function.
Methods == Functions

Java 8 Lambda Expressions versus Scala Functions

Instance Method References

Java 8

```
List<Boolean> trueOnly = filter(trueOrFalse, Boolean::booleanValue);
```

Scala

Scala uses the _ (underscore) to refer to the current instance that is iterated through

```
val trueOnly = filter(trueOrFalse, _.booleanValue )
```

So, what's the difference?

Higher Order Functions reveal the difference:

Every Higher Order Function needs a **specific Functional Interface** in Java 8.

Java 8

```
interface Collection<T> {

  public Collection<T> filter(Predicate<T> p);
...
```

```
interface Predicate<T> {
  public boolean op(T t);
}
```

The compiler transforms this syntax to a **generic Function object**. Therefore, no Functional Interface needs to be defined.

Scala

```
trait Seq[T] {

  def filter(f:T => Boolean):Seq[T]
...
```

```
class Function1[I, R] {
   def apply(i:I): R
}
```

Scala has a function syntax.

So, what's the difference?

Using Lambda's/Functions as Objects reveal the difference too:

A Lambda always needs to be assigned to a **matching Functional Interface** in Java 8.

Java 8

```
Predicate<Integer> evenFilter = (Integer i) -> i % 2 == 0;
Predicate<Integer> evenFilter = NumUtils::isEven;

numbers.filter(evenFilter);
```

Due to Scala's Type Inference combined with Functions as 'First Class Citizens' approach this is not necessary:

Scala

```
val evenFilter = (i:Int) => i % 2 == 0

//evenFilter has type: Int => Boolean
numbers.filter(evenFilter)
```

Detailed Comparison

Java 8 Lambda Expressions

- Syntax is limited to *calling* Higher Order Functions
- Functions are *interfaces* with one method

Scala Functions

- Have a syntax for *defining* and *calling* Higher Order Functions
- Functions are *objects* with methods
- Rich Function Features:
  - Function Composition
  - Currying
  - Call-by-name arguments
  - PartialFunctions
  - Pattern matching

Lambda's are **Syntactic Sugar** for **Functional Interfaces**

Functions are **First Class Citizens** fully supporting the **Functional Programming Paradigm**

Taking off with Functional Programming

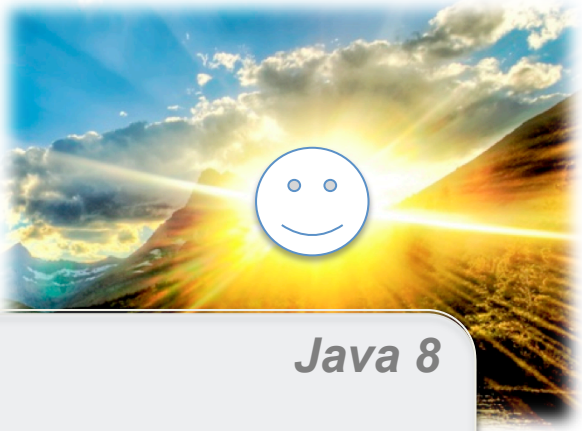
Iterations in Java are 'external' by default…

What is going on here?

Java

```
List<Student> students = ...;
List<Student> topGrades = new ArrayList<Student>();
List<String> result = new ArrayList<String>();
for(Student student : students){
    if(student.getGrade() >= 9) {
        topGrades.add(student);
     }
   }
}
Collections.sort(topGrades, new Comparator<Student>() {
  public int compare(Student student1, Student student2) {
    return student1.getGrade().compareTo(student2.getGrade());
  }
});
for(Student student : topGrades){
    result.add(student.getFirstName() + " " + student.getLastName());
}
```

Internal Iteration: When Lambda's/ Functions start to shine



Ah, now I get it:

```
List<Student> students = ...
List<String> topGrades =
    students.filter(s -> s.getScore() >= 9)
            .sortedBy(Student::getLastName)
            .map(s -> s.getFirstName() + " " + s.getLastName())
            .into(new ArrayList<>());
```

Java 8

Not yet available

Advantages:

- Re-use of generic, higher level control structures
- Less error prone
- More expressive and readable
- Chainable

Internal Iteration: When Lambda's/ Functions start to shine

Java 8

```
List<Student> students = ...
List<String> topGrades =
    students.filter(s -> s.getScore() >= 9)
            .sortedBy(Student::getLastName)
            .map(s -> s.getFirstName() + " " + s.getLastName())
            .into(new ArrayList<>());
```

Not yet available

No need to use the **into**() method. This happens in Scala 'automagically'

Scala

```
val students = ...
val topGrades =
    students.filter(_.score >= 9)
            .sortBy(_.lastName)
            .map(s => s.firstName + " " + s.lastName)
```

But wait: there is more

Find longest word in a huuuuge text file

```java
List<String> lines = //get many many lines

int lengthLongestWord = 0;
for(String line : lines) {
    for(String word : line.split(" ")) {
        if(word.length() > lengthLongestWord) {
            lengthLongestWord = word.length();
        }
     }
}
```

What's the problem with the **for** loops ?

But wait: there is more

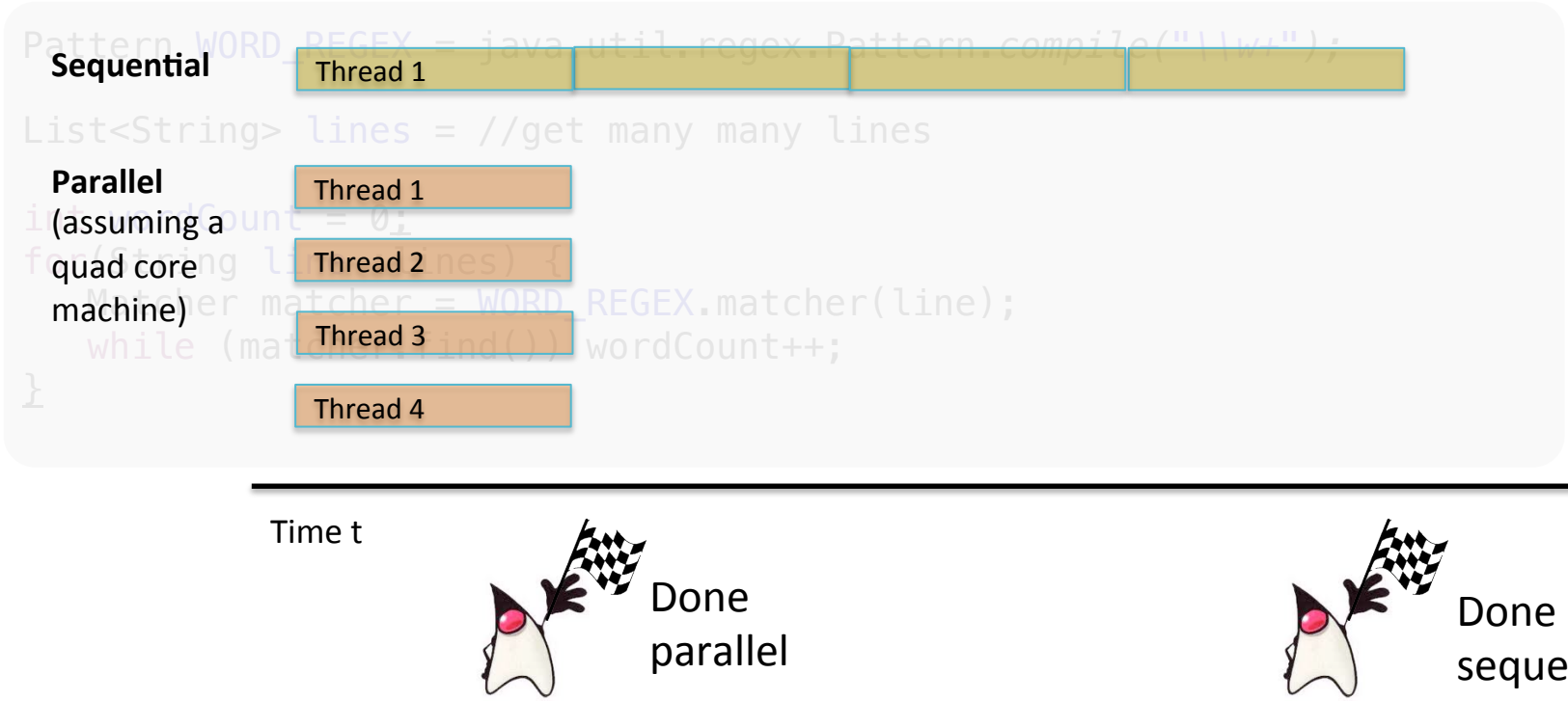
Find longest word in a huuuuge text file

```java
List<String> lines = //get many many lines

int lengthLongestWord = 0;
for(String line : lines) {
    for(String word : line.split(" ")) {
        if(word.length() > lengthLongestWord) {
            lengthLongestWord = word.length();
        }
     }
}
```

they cannot be executed in parallel!

But wait: there is more

Find longest word in a huuuuge text file



Parallel Collections to the rescue

Find longest word in a huuuuge text file

Java 8

```
List<String> lines = //get many many lines
int lengthLongestWord = lines.parallel()
                             .map(line -> line.split(" ")
                                              .map(String::length)
                                              .reduce(Math::max))
                             .reduce(Math::max);
```

Scala

```
val lines = //get many many lines
val lengthLongestWord = lines.par
                             .map(_.split(" ").map(_.length).max)
                             .max
```

Detailed Comparison: Collections & Lambda's/Functions

	Java 8 Collections	Scala Collections
Higher Order Functions		
Parallelism		
Automatic collection conversion		
Fast immutable data structures		

Other

Automatic conversion from Java to Scala collections

Back to Java 8: How to make it fit?

How is it possible to add Higher Order Functions (e.g. `filter map foreach` etc.) to the `java.util.Collection` interface without breaking backwards compatibility?

Java 8

```
interface Collection<T> {

  public Collection<T> filter(Predicate<T> p);

  public <R> Collection<R> map(Mapper<T, R> m);

 ...
```

'Old' Java code (prior Java 8) running in JRE 8 would have to implement all the new methods that offer 'internal iteration' with Lambda's.

Back to Java 8:
How to make it fit?

…with **default** implementations for interfaces
aka *Virtual Extension Methods*!

Java 8

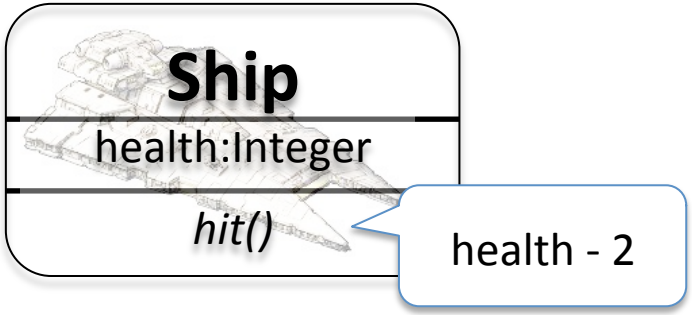
```
interface Collection<T> {

  public default Collection<T> filter(Predicate<? super T> p) {
     Collection<T> res = new ArrayList<>();
     for(T item : this) {
        if(p.test(item)) {
           res.add(item);
        }
     }
     return res;
  }

 ...
```

A closer look at Java 8' Virtual Extension Methods

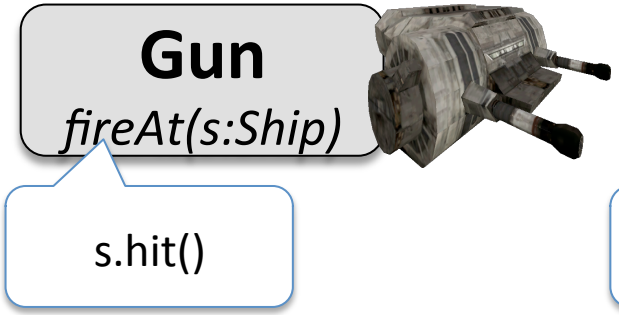
- Primary motivator is API evolution
  (Backwards compatibility for Lambda's in Collections)
- Useful mechanism by itself:
  Enables **Multiple Inheritance of *behavior***
- Inspired by Scala traits (among others)

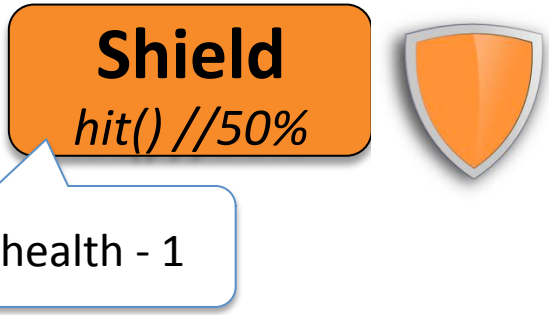
Do we need Multiple Inheritance?

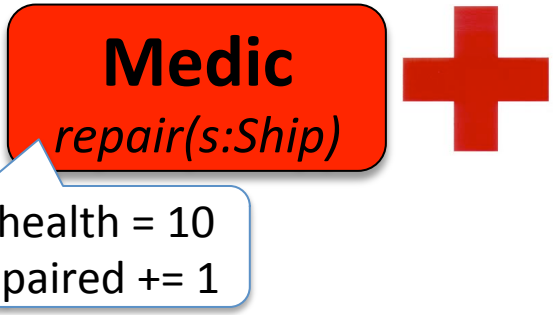
Let's model the domain for a 'futuristic' game



Possible characteristics of a Ship:

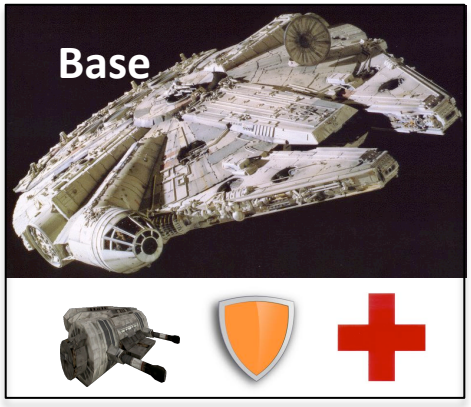


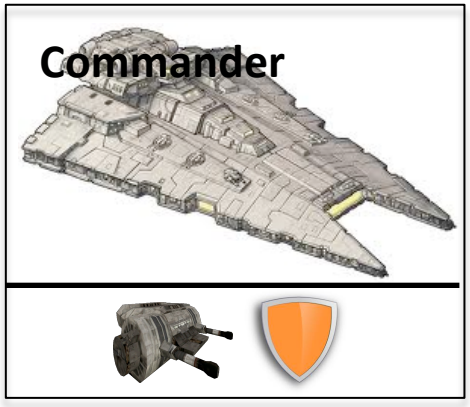


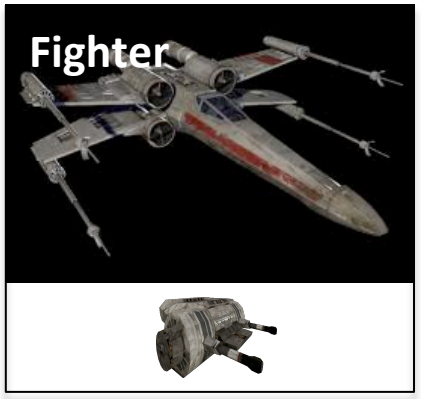


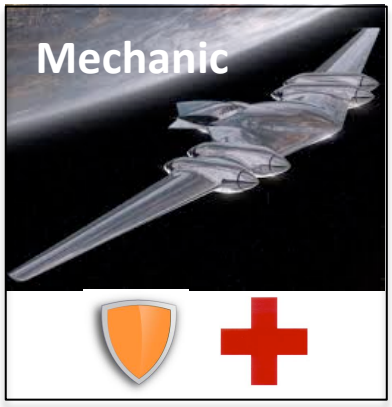
The Limits of Single inheritance

Possible Ship implementations:









The Limits of Single inheritance

Implementation Challenge: Single inheritance won't work

	Gun	Shield	Medic
Base			
Commander			
Fighter			
Mechanic			

Scala
Multiple Inheritance of Behavior

Adding Behavior to a Ship (Scala)

```
trait Gun {
  def fireAt(s:Ship) = s.hit
}
```

A trait is an interface with an implementation (behavior and/or state)

```
class Ship(var health:Int = 0) {
  def hit() = health -= 2
}
```

```
val goodFighter = new Ship(10) with Gun
val evilFighter = new Ship(10) with Gun

goodFighter.fireAt(evilFighter)

println(evilFighter.health)
> 8
```

It can be 'mixed-in' with a class using the keyword: `with`

Java 8
Multiple Inheritance of Behavior

Adding Behavior to a Ship (Java 8)

```
interface Gun {
  default void fireAt(Ship s) { s.hit(); }
}
```

```
class Ship {
  int health = 0;
  //constructor, getters, setters, hit method
}
```

Works with Virtual Extension Methods

```
class Fighter extends Ship implements Gun {…}
Fighter goodFighter = new Fighter(10);
Fighter evilFighter = new Fighter(10);

goodFighter.fireAt(evilFighter);

println(evilFighter.getHealth());
> 8
```

Scala
Multiple Inheritance of Behavior

Change Behavior of Ship (Scala)

```
trait Shield {
    self:Ship =>
    override def hit() = self.health -= 1
}
```

```
class Ship(var health:Int = 0) {
    def hit() = health -= 2
}
```

Trait's can have a self-type. The self-type tells with which class this trait can be 'mixed'.

The self-type provides access to the 'mixed-in' class

Traits can override methods of the 'mixed-in' class

```
val commander = new Ship(10) with Gun with Shield
val evilFighter = new Ship(10) with Gun

evilFighter.fireAt(commander)

println(commander.health)
> 9
```

Java 8
Multiple Inheritance of Behavior

Change Behavior of Ship (Java 8)

```
trait Shield {
    self:Ship =>
    override def hit(
}
```

```
class Ship(var health
    def hit() = healt
}
```

```
val commander = new Sh
val evilFighter = new

evilFighter.fireAt(commander)

println(commander.health)
> 9
```

Does not work with Virtual Extension Methods (VEM)

Why?

- VEM cannot override methods of a class (they are overridden)
- VEM have no self-types

Scala
Multiple Inheritance of State

Adding State and Behavior to Ship (Scala)

```
trait Medic {
    var repaired = 0
    def repair(s:Ship) = {
        s.health = 10
        repaired += 1
    }
}
```

A trait can have state.

```
val medic = new Ship(10) with Shield with Medic
val brokenShip = new Ship(1) with Gun

medic.repair(brokenShip)

println(brokenShip.health)
> 10
println(medic.repaired)
> 1
```

Java 8
Multiple Inheritance of State

Adding State and Behavior to Ship (Java 8)

```
trait Medic {
    var repaired = 0
    def repair(s:Ship
        s.health = 1
        repaired +=
    }
}
```

Does not work with Virtual Extension Methods (VEM)

Why?

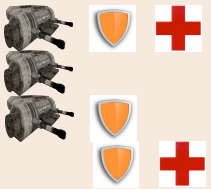
- VEM cannot have state

```
val medic = new Ship(10) with Shield with Medic
val brokenShip = new Ship(1) with Gun

medic.repair(brokenShip)

println(brokenShip.health)
> 10
println(medic.repaired)
> 1
```

Multiple Inheritance Done Right

Implementation in Scala with traits:

```
val base = new Ship(100) with Gun with Shield with Medic
val fighter = new Ship(10) with Gun
val commander = new Ship(50) with Gun with Shield
val mechanic = new Ship(20) with Shield with Medic
```



- No duplication
- Perfectly modular
- Types are preserved `assert(base.isInstanceOf[Medic])`
- Execution and initialization is predictable

Scala Traits vs Java 8 Virtual Extension Methods

	Java 8 Virtual Extension Methods	Scala Traits
Have methods		
Have fields		
Access implementing class in a typesafe way		
Override methods		
Stackable (call to super is linearized)		
Intent	Grow the language	Multiple inheritance ‘without the issues’

Is there no better way to "Grow the language"?

- What we really want is 'a mechanism' that adds 'new methods' to 'existing classes'
- Virtual Extension Methods solve this problem on the interface level:
  - When we extend the interface with `defaults`, all classes inheriting from this interface gain the functionality
  - E.g. `Arrays.asList(1,2,3).forEach(i -> println(i));`
- Drawback: *limited to interfaces*
- But how about:

I want to extend existing classes by myself too!

```
"You win %2.2f%n".format(333333.3333);
new java.util.Date().toString("yyyy-MM-dd");
```

Welcome to Scala Implicits

…or the ‘pimp my library pattern’

Implicits in Practice

Add a new method to an existing class (pimp my library):

```scala
implicit class RichDate(val date:Date) extends AnyVal {
  def toString(pat:String):String = new SimpleDateFormat(pat).format(date)
}

new java.util.Date().toString("yyyy-MM-dd")
```

Scala

```scala
//compiler turns this into:
RichDate.methods$.toString(new Date(), "yyyy-MM-dd")
```

Scala

Lambda's for Collections: Scala solved it before…

Import `JavaConversions` and you are done:

```scala
import collection.JavaConversions._

java.util.Arrays.asList(1,2,3).foreach(i => println(i))

java.util.Arrays.asList(1,2,3).filter(_ % 2 == 0)
```

Scala

The `JavaConversions` object contains implicits that convert a `java.util.Collection` into a Scala's counterpart.

Now we have all Higher Order Functions like `foreach`, `filter`, `map` etc. available

Implicits can do more...

Implicitly convert from one type to another
(here java.sql.Date to org.joda.time.DateTime):

```
implicit def fromSqlDate(sd:java.sql.Date):DateTime = new DateTime(sd)

val d:DateTime = resultSet.getDate("month") //would return java.sql.Date
```

```
//compiler turns this into:
val d:DateTime = fromSqlDate(resultSet.getDate("month"))
```

How about converting 'Functional Interfaces' into 'First Class Functions'?

Convert a Scala Function into a Google Guava Predicate

```
implicit def convertFun[T](fun:T => Boolean) = new Predicate[T]() {
    def apply(in:T):Boolean =  fun(in)
```

```
import com.google.common.collect._
Collections2.filter(Arrays.asList(1,2,3), (i:Int) => i % 2 == 0 )
```

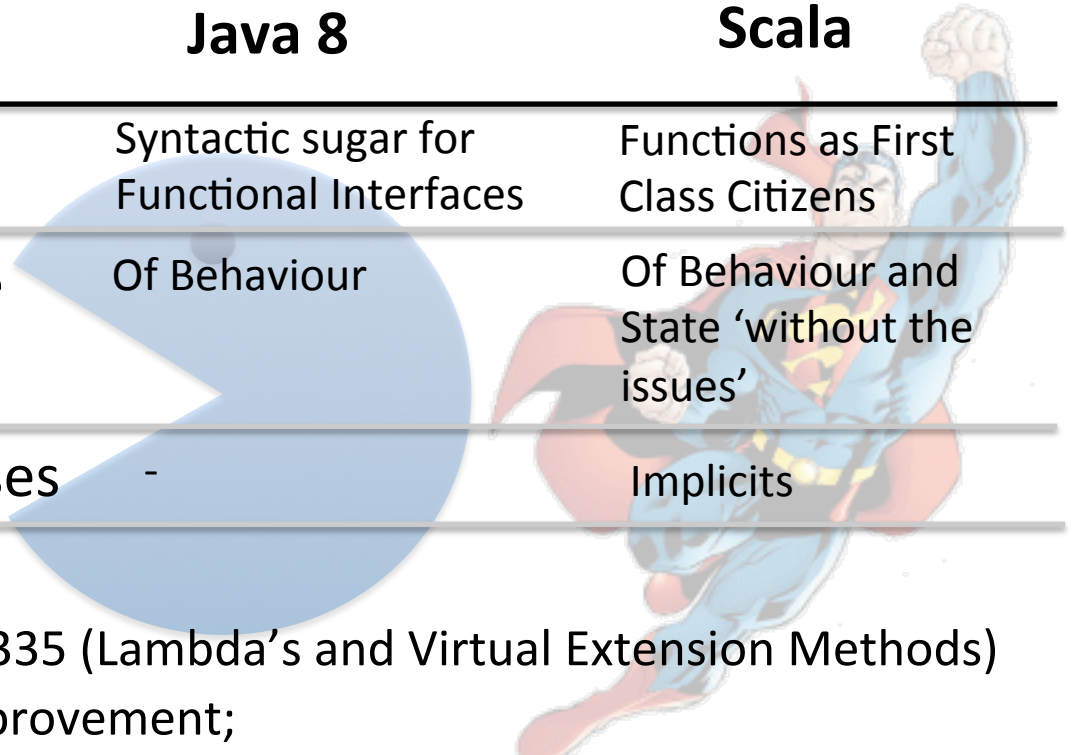
Now we can use a Scala Function in the Google API (instead of the Guava Predicate)

If Java 8 had chosen for implicits…

- A proven mechanism was added to add new methods to existing classes *without being limited to interfaces*
- *API designers* AND *API users* would be able to add new methods to existing classes
- Bridging of API's – like Google Guava to Java Collections – would be possible
- Last but not least: the implicit mechanism is not new in Java: How about: *Autoboxing, Lambda's*?

Is Java 8 a Scala killer?

	Java 8	Scala
Lambda's/Functions	Syntactic sugar for Functional Interfaces	Functions as First Class Citizens
Multiple Inheritance	Of Behaviour	Of Behaviour and State 'without the issues'
Extend existing classes	-	Implicits



- The features of JSR 335 (Lambda's and Virtual Extension Methods) are definitely an improvement;
- Choosing *Implicits* above *Virtual Extension Methods* for preserving backwards compatibility would have made Java 8 more powerful;
- Java 8' new language constructs are not as features rich and complete as in Scala.

Q & A

```
def ask(question: Any) = question match {
   case "?"   => "Another layer of abstraction"
   case "???" => "It depends"
   case _     => 42
 }
```

Evalutation

http://bit.ly/xc_is