

# Scala in practice.

Eugene Dzhurinsky

August 15, 2014

# Why Scala?

## Better than Java

- ▶ Statically typed. Rich type system. Type inference.
- ▶ JVM-based.
- ▶ 100% compatible with Java libraries.
- ▶ Concise. Syntax sugar. Syntetic methods.
- ▶ High-order functions, closures.

# Why Scala?

## Better than Java

- ▶ Statically typed. Rich type system. Type inference.
- ▶ JVM-based.
- ▶ 100% compatible with Java libraries.
- ▶ Concise. Syntax sugar. Syntetic methods.
- ▶ High-order functions, closures.

# Why Scala?

## Better than Java

- ▶ Statically typed. Rich type system. Type inference.
- ▶ JVM-based.
- ▶ 100% compatible with Java libraries.
- ▶ Concise. Syntax sugar. Syntetic methods.
- ▶ High-order functions, closures.

# Why Scala?

## Better than Java

- ▶ Statically typed. Rich type system. Type inference.
- ▶ JVM-based.
- ▶ 100% compatible with Java libraries.
- ▶ Concise. Syntax sugar. Syntetic methods.
- ▶ High-order functions, closures.

# Why Scala?

## Better than Java

- ▶ Statically typed. Rich type system. Type inference.
- ▶ JVM-based.
- ▶ 100% compatible with Java libraries.
- ▶ Concise. Syntax sugar. Syntetic methods.
- ▶ High-order functions, closures.

# Java Engine Example

```
package sample;

public class Engine {
    protected final int power, maxRpm, minRpm;
    public Engine(int power, int maxRpm, int minRpm) {
        this.power = power; this.maxRpm = maxRpm; this.minRpm = minRpm;
    }
    public int getPower() { return power; }
    public int getMaxRpm() { return maxRpm; }
    public int getMinRpm() { return minRpm; }
    @Override public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Engine engine = (Engine) o;
        if (maxRpm != engine.maxRpm) return false;
        if (minRpm != engine.minRpm) return false;
        if (power != engine.power) return false;
        return true;
    }
    @Override public int hashCode() {
        int result = power;
        result = 31 * result + maxRpm;
        result = 31 * result + minRpm;
        return result;
    }
    @Override public String toString() {
        return "Engine{" +
            "power=" + power +
            ", maxRpm=" + maxRpm +
            ", minRpm=" + minRpm +
            '}';
    }
}
```

# Scala Engine Example

```
package sample

import scala.beans.BeanProperty

case class SEngine(@BeanProperty power: Int,
                   @BeanProperty maxRpm: Int,
                   @BeanProperty minRpm: Int)
```



# Val and Var. Type inference and string interpolation.

```
package sample

object ValVarExample extends App {

  val something = 10
  val somethingInt: Int = 20

  println(s"Something is $something, another something is $somethingInt")

  // something = 20 - doesn't compile

  var mutableSomething = 10
  println(s"Mutable? $mutableSomething")
  mutableSomething = 20
  println(s"Mutable! $mutableSomething")
}
```

# Val and Var in Java?

```
package sample;

public class JValVarExample {

    static int something;

    static int somethingInt;

    static int mutableSomething;

    public static void main(String[] args) throws Exception {
        something = 10;
        somethingInt = 20;
        System.out.println(
            String.format(
                "Something is %1$d, another something is %2$d",
                something,
                somethingInt)
        );
        something = 20; // you can't have a true "val" here — the closest would be "final"
                     // however you can't initialize a "static final" in "main"

        mutableSomething = 10;
        System.out.println(String.format("Mutable? %1$d", mutableSomething));
        mutableSomething = 20;
        System.out.println(String.format("Mutable! %1$d", mutableSomething));
    }
}
```

# Lazy variables, traits, “generics” and implicit conversions

```
package sample
object LazyTrait extends App {

  trait Worker[T] {
    val name: String
    println(s"$name I'm initialized!") // define working code in trait
    def work(aWork: T): Unit
  }

  // value class — no new instance generated
  case class CoolString(param: String) extends AnyVal
  implicit def lift2Cool(src: String) = CoolString(src) // "new" isn't necessary here

  lazy val lazyWorker = new {val name = "Laaazy"} with Worker[CoolString] {
    override def work(stringWork: CoolString) {
      println(s"Omg I've been called! ${stringWork.param}") // method call in braces
    }
  }

  val eagerWorker = new {val name = "Eager!"} with Worker[String] {
    override def work(stringWork: String) {
      println(s"Omg I've been called! $stringWork")
    }
  }
  eagerWorker.work("Work'o'holic")
  lazyWorker.work(CoolString("Laaaazy")) // simple constructor calls via .apply()
  lazyWorker.work("Implicit laaaazy") // implicit conversion in place
}
```



# More fun with implicits

```
package sample
object MoreImplicits extends App {
  trait mayAdd[T] {
    def +!(src: T): T
  }
  case class myString(param: String) extends mayAdd[myString] {
    override def +!(src: myString) = myString(param + ":" + src.param)
  }
  case class myInt(param: Int) extends mayAdd[myInt] {
    override def +!(src: myInt) = myInt(param * src.param)
  }

  implicit def pimpString(str: String) = myString(str)
  implicit def pimplyInt(num: Int) = myInt(num)

  println(1 +! 2 +! 3); println("aaa" +! "bbb")

  type strFunc = (String) => myString
  def doSomethingWith(str: String)(implicit f: strFunc) = f(str.reverse)
  {
    implicit def upperCase : strFunc = x => myString(x.toUpperCase)
    println(doSomethingWith("hello world").param)
  }
  {
    implicit def upperCase : strFunc = x => myString(x.capitalize)
    println(doSomethingWith("hello world").param)
  }
}
```