Cut Your Boilerplate with Scala

Lex Spoon, Ph.D.
Google, Inc.

Why boilerplate matters

- It's unwieldy
- It has bugs
- Code is read more than written
- It's harder to learn good composition patterns
- Traditional folding editors aren't enough
- A theoretical inline folder would be equivalent to a new language

About the language

http://www.scala-lang.org

- Developed by Prof. Martin Odersky
- Parallels development of Java, especially generics, but develops much faster
- Blends functional and object-oriented features
- More orthogonal, thus has fewer odd corner cases
- Many, many ways to reduce your boilerplate

Cut Your Boilerplate with Scala

- Cleaner fields
- Case classes
- Scripty maps and sets
- Operator overloading
- Interfaces with implementations
- Definition-side variance
- Pattern matching instead of visitors

Cleaner Fields

- Java:
```
class Point {
  private final int x;
  private final int y;
  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  int getX() {
    return x;
  }
  int getY() {
    return y;
  }
}
```
- Scala:

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

Cleaner Fields

- Where are the getters?
  - It's myPoint.x, not myPoint.getX()
    (which is also less boilerplate....)
  - Concise final fields: so they aren't all mutable
  - Uniform access principle: you can replace it with a method and the caller still compiles

Cleaner Fields

- Concise final fields
  - Java: final int x;
  - Scala: val x
- (So, more of your fields are final)

Cleaner Fields

- Changing to a method:

  ```
  def x = r * Math.cos(theta)
  ```

- Caller still writes p.x

Cleaner Fields

- Where are the constructors?
  - Scala classes have parameters
  - They must be supplied at construction time
  - (Just like generic classes must have types supplied at construction time)
  - Their parameters are visible within the class
  - Prefixing “val” makes them visible as read-only fields
  - You can still have extra constructors if you want

Case classes

- Java:

```
class Point {
  ...
  public String toString() {
    return “Point(“ + x + “,” + y + “)”;
  }
  public boolean equals(Object that) {
    if (that instanceof Point) {
      Point thatPoint = (Point) that;
      return (x == thatPoint.x)
        && (y == thatPoint.y);
```

Case classes

- (Java, cont'd)

```
    }
  return false;
}
public int hashCode() {
  return (13 + 17 * x) * 17 + y;
}
```

- Scala:

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

Case classes

- Sometimes a class really is just its data
- It helps on the construction side, too:
- Java:
  new Line(new Point(0, 0), new Point(1, 3))
- Scala:
  Line(Point(0,0), Point(1,3))

Scripty Maps and Sets

- Java:
```
set.add(“foo”);
map.put(“foo”, “bar”);
map.put(“foo”, map.get(“bar”)+map.get(“baz”))
```

- Scala:
```
set += “foo”
map(“foo”) = “bar”
map(“foo”) = map(“bar”) + map(“baz”)
```

Scripty Maps and Sets

- Ruby:
```
h = {
'dog' => 'canine',
'cat' => 'feline',
'donkey' => 'asinine',
 12 => 'dodecine'}
puts h.length
puts h['dog']
puts h
puts h[12]

# from rubylearning.com
```

- Scala:
```
val h = Map(
  "dog" -> "canine",
  "cat" -> "feline",
  "donkey" -> "asinine",
  12 -> "dodecine")
println(h.size)
println(h("dog"))
println(h)
println(h(12))
```

Scripty Maps and Sets

- Groovy:

```
scores = [
 "Brett":100,
 "Pete":"Did not finish",
 "Andrew":86.87934 ]
println scores["Pete"]
println scores.Pete
scores["Pete"] = 3
# from docs.codehaus.org
```

- Scala:

```
import
collection.mutable.Map;

val scores = Map(
 “Brett” -> 100,
 “Pete” -> “Did not finish”,
 “Andrew” -> 86.87934)
println(scores(“Pete”))
println(scores(“Pete”))
scores(“Pete”) = 3
```

Operator Overloading

- //Java
```
BigInteger fact(BigInteger n) {
  if (n.equals(BigInteger.valueOf(0))) {
    return BigInteger.valueOf(1));
  } else {
    return n.multiply(
      n.subtract(BigInteger.valueOf(1)));
  }
```
- // Scala
```
def fact(n: BigInteger): BigInteger =
  if (n==0) 1 else n*fact(n-1)
```

Operator Overloading

- In Scala:
```
class BigInteger {
  // Define a * method just like any other
  def *(n: BigInteger) = ...
  ...
}
```
- Use it like this:
```
  n * fact(n - BigInteger.valueOf(1))
```

Operator Overloading

- For the language lawyers:
  - Precedence is fixed, e.g. * always happens before +
  - Syntactically, * is an identifier just like fact
  - Logically, you can use words as operators:
    node setPosition -1
- Not everything you can do makes for good code
  - n.*(fact(n.-(1))) // yeck!

Operator Overloading

- Implicit conversions remove the valueOf calls
- For the language lawyers:
  - Only implicit conversions in scope can apply
  - It's a compile error if two different conversions would work
  - They do not chain; only one is used at a time
- By the way, the same features let you use Java's BigInteger as is

Operater Overloading

- Given operator overloading and implicit conversions:

```
def fact(n: BigInteger): BigInteger =
  if (n == 0) 1 else n * fact(n-1)
```

Interfaces with Implementations

- String API has ~50 instance methods
  - indexOf, replace, toLowercase, charPointAt, ...
- CharSequence has 4 methods
- All the missing ones translate to boilerplate

Interfaces with Implementations

- Likewise for Sets, so you write things like:
  Sets.difference(Sets.union(set1, set2), set3)
- It sure would be nice to write:
  set1 ++ set2 -- set3

Interfaces with Implementations

- Scala has “traits” to replace Java interfaces
- They allow concrete methods
- There are non-obvious semantics for “super” calls
- However, everything else is straightforward

Interfaces with Implementations

- trait Set[+T] {

```
// abstract
def contains(x: Any): Boolean

// concrete
def subsetOf(that: Set[T]) =
  size == that.size && forall(that.contains(_))
```

}

Definition-side Variance

- Java:
  int countCalories(List<? extends Fruit> fruit)

- Scala:
  def countCalories(List[Fruit] fruit): Int

Definition-side Variance

- In tutorials, you see code like this:

```
int countCalories(List<Fruit> fruit) { ... }

List<Fruit> myFruit = new ArrayList<Fruit>();
countCalories(myFruit);
```

Definition-side Variance

- Larger code bases run into this situation:

```
int countCalories(List<Fruit> fruit) { ... }

List<Kumkwat> kumkwats =
  new ArrayList<Kumkwat>();

countCalories(kumkwats); // compile error!
```

Definition-side Variance

- Why the type error?
- List<Kumkwat> kumkwats = ...
  List<Fruit> fruit = kumkwats;
  fruit.add(0, new Orange());
  kumkwats.get(0); // *got an Orange!*

Definition-side Variance

- Java's answer is use-side variance:

```
int countCalories(List<? extends Fruit> fruit) {
  ...
}

countCalories(kumkwats); // OK now
```

- Problem solved?

Definition-side Variance

- It's like const poisoning from C:

```
int countCalories(List<? extends Fruit> f) { ... }

boolean chk(List<? extends Kumkwat> l) { ... }

<T> List<T> filter(
  List<? extends T> list,
  Predicate<? super T> pred) { ... }
```

Definition-side Variance

- What goes wrong:
  - The types take up more space than the code
  - The definer of countCalories had to get it right
  - To get it right requires serious chops:
    It's List<? extends Fruit>
    but Predicate<? super Fruit>

Definition-side Variance

- Scala has definition-side variance
- List[T] automatically means List[_ <: T]
- Predicate[T] means Predicate[_ >: T]
- The definer has to ask for it, like this:

  ```
  abstract class List[+T] // + means T is covariant
  ```
- Now, List[Kumkwat] is a subtype of List[Fruit]

Definition-side Variance

- def countCalories(List[Fruit] fruit): Int = ...

  ...

  var kumkwats: List[Kumkwat] = ...
  countCalories(kumkwats)

Definition-side Variance

- Wait, why is this safe?
- The default List type is immutable
- var kumkwats: List[Kumkwat] = ...
  var fruit: List[Fruit] = kumkawts
  fruit.add(0, new Orange()) // type error
  kumkwats.get(0)

Definition-side Variance

- The definition side is checked
- For +T, only the covariant uses are allowed

```
abstract class List[+T] {
  def apply(n: Int): T   // covariant
  def add(x: T)   // contravariant!
  // trickier cases:
  def from(n: Int): List[T] // still covariant
  def bar(n: Int): Predicate[T] // contravariant!
}
```

Definition-side Variance

- Type List: covariant
- Type ListBuffer: ***invariant!***

- Type immutable.Set: covariant
- Type mutable.Set: invariant

Pattern Matching

- Java:
```
if (foo() instanceof Point) {
  Point p = (Point) foo();
  if (p.x == 0 && p.y == 3) {
    // match!
  }
}
```
- Scala:
```
foo() match {
  case Point(0, 3) => // match!
}
```

Pattern Matching

- It's a bigger win when patterns nest
- Java:
```
if (foo() instanceof Line) {
  Line line = (Line) foo();
  if (line.getStart() instanceof Point) {
    Point p = (Point) line.getStart();
    if (p.getX() == 0 && p.getY() == 0) {
      // match!
```
- Scala:
```
foo() match {
  case Line(Point(0,0), p) => // match!
```

Questions?

http://www.scala-lang.org

- Cleaner fields
- Case classes
- Scripty maps and sets
- Operator overloading
- Interfaces with implementations
- Definition-side variance
- Pattern matching instead of visitors