Scala magic

Alexander Podkhalyuzin

July 29, 2012

Syntactic sugar

For comprehensions

”For comprehensions” is syntactic sugar:

```
//map
for (i <- 1 to 5) yield i * i
//foreach
for (i <- 1 to 5) println(i)
//flatMap
for {
  i <- 1 to 5
  j <- 1 to
} yield i + j
```

```
//map
(1 to 5).map { i => i * i }
//foreach
(1 to 5).foreach { i =>
  println(i)
}
//flatMap
(1 to 5).flatMap { i =>
  (1 to 5).map { j => i + j }
}
```

For comprehensions

Irrefutable pattern means that matching can be checked by compiler, otherwise additional 'withFilter' will be generated:

```
val matrix: Seq[Seq[Int]] = ???
for (Seq(x, y) <- matrix)
  yield x + y
```

```
val matrix: Seq[Seq[Int]] = ???
matrix.withFilter {
  case Seq(x, y) => true
  case _ => false
} map {
  case Seq(x, y) => x + y
}
```

By-name parameters

By-name parameters is just shorthand:

```
def foo(x: => Int) {
  println(x)
}

foo(1)
```

```
def foo(x: () => Int) {
  println(x())
}

foo(() => 1)
```

So evaluation of call-side code is made on every parameter usage.

Non local return

Return statement inside of closure throws NonLocalReturnControl.

- It consumes additional CPU
- Don’t catch this Throwable
- Don’t miss that return is non-local inside of ”for statement”

String interpolation

New Scala 2.10 syntactic sugar example

```
val link = 1
ref"Obviously ${link} = 1"
```

```
val link = 1
StringContext("Obviously ",
  " = 1").ref(link)
```

Note: this is backward incompatible Scala syntax change

Generated ByteCode

Local functions

Let's look how it compiles

```
class LocalFunctions {
  def foo {
    var i = 1
    val j = 2
    def local {
      i += j
    }
    local
    local
  }
}
```

```
import scala.runtime.IntRef;
public class LocalFunctions {
    public void foo() {
        IntRef i$1 = new IntRef(1);
        int j$1 = 2;
        local$1(i$1, j$1);
        local$1(i$1, j$1);
    }
    private final void local$1(
      IntRef intref, int i) {
        intref.elem += i;
    }
    public LocalFunctions() {}
}
```

We can see that it's very similar to Java private members usage

Lazy values

This is also some shorthand for some usual Java usages

```
class Lazy {
  lazy val x = 1
}
```

```
import scala.runtime.BoxedUnit;
public class Lazy {
  public int x() {
    if ((bitmap$0 & 2) == 0)
      synchronized (this) {
        if ((bitmap$0 & 2) == 0) {
          x = 1;
          bitmap$0 = bitmap$0 | 2;
        }
        BoxedUnit _tmp =
          BoxedUnit.UNIT;
      }
      return x;
    }
  public Lazy() {}
  private int x;
  public volatile int bitmap$0;
}
```

Lazy vals can produce deadlocks

Classes and names

All Unicode symbols have equivalent

- : ➔ $colon
- + ➔ $plus
- © ➔ $u00A9

In Scala code you can use both identifier variants. Try NameTransfomer.encode/decode from scala-compiler.jar.

Objects

Object is actually two classes in bytecode

- Object$ contains static field MODULE$, and compiled code of all members
- Object contains all static members of object with body Object$.MODULE$.foo()

Objects

Static members added for compatibility with Java, however sometimes static members are not generated in

- Companion Trait
- Companion Class if it contains member with the same name
- Members of java.lang.Object

Traits

For every trait will be generated Trait$class with members implementation

```
trait A {
  val x = 1

  def foo(x: Int) = 2
}
```

```
public abstract class A$class {
    public static int foo(A $this,
                          int x) {
        return 2;
    }
    public static void $init$(
      A $this
    ) {
        $this.
          test$A$_setter_$x_$eq(1);
    }
}
```

Trait subclasses

Java class implementing trait can simply implement implemented trait members in the following way

```
int foo() { return Trait$class.foo(this); }
```

In Scala compiler does the same automatically

Linearization

Compiler needs this algorithm to define search order for members from base classes

```
trait A extends B
trait B
class C extends B
class D extends C with A
```

Linearization

Compiler needs this algorithm to define search order for members from base classes

```
trait A extends B
trait B
class C extends B
class D extends C with A
```

Linearization is D, A, C, B.

Linearization

If two members with the same signature inherited by some class and one of members has modifier ’override’ then overriding can happen implicitly according to linearization rules.

Initialization order

Initialization order is opposite to linearization order.

If you try to use val before its initialization, you can get NPE.

There are two ways to avoid possible NPE:

- Use lazy val
- Use early definitions

Binary compatibility

Methods

This is similar to Java

- You can add methods to trait, but only if old members haven't link to new methods, because implementation will not be added to inheritors.
- Do not add methods to Companion Object with name, which Class contains, because static implementations will be removed by compiler.

Methods

This is similar to Java

- You can add methods to trait, but only if old members haven't link to new methods, because implementation will not be added to inheritors.
- Do not add methods to Companion Object with name, which Class contains, because static implementations will be removed by compiler.

Values

Value in class and object compiled to getter (setter?) and field, which is initialized in constructor. So it’s like with methods.
Trait value compiled in the same way. Just field initialization in method Trait$class.$init$.

- Don’t add values to traits
- Don’t change def to val
- Don’t change val to def, because field won’t be initialized

Lazy values

Field bitmap is not generated if base class contains such field

- In such cases do not add lazy values to classes
- You can add lazy val to Trait, but this value shouldn’t be used in old code
- You can change lazy val to def and vice versa, however inheritors won’t see such changes

Questions?