SIP: Value Classes

Martin Odersky
Jeff Olson
Paul Phillips
Joshua Suereth

7 February 2012

Change history

30 Jan 2012: Original inline classes proposal
7 Feb 2012: Changed inline classes to value classes, added Josh Suereth as author.

Introduction

This is a proposal to introduce syntax for classes in Scala that can get completely inlined, so operations on these classes have zero overhead compared to external methods. Some use cases for inlined classes are:

- *Inlined implicit wrappers*. Methods on those wrappers would be translated to extension methods.
- *New numeric classes*, such as unsigned ints. There would no longer need to be a boxing overhead for such classes. So this is similar to value types in .NET.
- Classes representing *units of measure.* Again, no boxing overhead would be incurred for these classes.

The proposal is currently in an early stage. It's not yet been implemented, and the proposed implementation strategy is too complicated to be able to predict with certainty that it will work as specified. Consequently, details of the proposal might change driven by implementation concerns.

Value Classes

The gist of the proposal is to allow user-defined classes to extend from AnyVal in situations like this:

```
class C (val u: U) extends AnyVal {
  def m1(ps1) = ...
```

```
  ...
  def mN(psN) = ...
}
```

Such classes are called *value classes*. A value class C must satisfy the following criteria:

1. C must have exactly one parameter, which is marked with **val** and which has public accessibility. The type of that parameter (e.g. `U` above) is called the *underlying type* of `C`.
2. C may not have @specialized type parameters.
3. The underlying type of C may not be a value class.
4. C may not have secondary constructors.
5. C may not define concrete `equals` or `hashCode` methods.
6. C must be either a toplevel class or a member of a statically accessible object.
7. C must be ephemeral.

A class or trait C is *ephemeral* if the following holds:

8. C may not declare fields (other than the parameter of a value class).
9. C may not contain object definitions.
10. C may not have initialization statements.

We say, a value class C *unboxes directly* to a class D if the underlying type of C is a type-instance of D. Indirect unboxing is the transitive closure of direct unboxing. A value class may not unbox directly or indirectly to itself.

The following implicit assumptions apply to value classes.

1. Value classes are implicitly treated as final, so they cannot be extended by other classes.
2. Value classes are implicitly assumed to have structural equality and hash codes. I.e. their `equals` and `hashCode` methods are taken to be defined as follows:

```
def equals(other: Any) = other match {
   case that: C => this.u == that.u
   case _ => false
}
def hashCode = u.hashCode
```

Universal traits

Scala's rule for inheritance do not permit value classes to extend traits that extend from `AnyRef.` To permit value classes to extend traits, we introduce *universal traits*, which extend

from `Any`. A universal trait `T` needs to explicitly extend class `Any`. In the example below, `Equals` is a universal trait with superclass `Any`, but `Ordered`'s superclass is still assumed to be `AnyRef`.

```
trait Equals[T] extends Any { … }
trait Ordered[T] extends Equals[T] { … }
```

To turn `Ordered` into a universal trait, add an explicit superclass `Any`:

```
trait Ordered[T] extends Any with Equals[T] { … }
```

Like value classes, universal traits need to be ephemeral.

Expansion of value classes.

Value classes are expanded as follows. For concreteness, we assume an value class `Meter` that is defined like this:

```
class Meter(val underlying: Double) extends AnyVal with Printable {
  def plus (other: Meter): Meter =
    new Meter(this.underlying + other.underlying)
  def divide (factor: Double): Meter = new Meter(this.underlying / factor)
  def less (other: Meter): Boolean = this.underlying < other.underlying
  override def toString: String = underlying.toString + "m"
}
trait Printable extends Any { def print: Unit = Console.print(this) }
```

For simplicity we assume that all expansion steps are done on erased types.

Step 1: Extracting methods.

Let the *extractable methods* of a value class be all methods that are directly declared in the class (as opposed to being inherited) and that do not contain a super call in their body. For each extractable method *m*, we create another method named `extension$`*m* in the companion object of that class (if no companion object exists, a fresh one is *created)*. The `extension$`*m* method takes an additional parameter in first position which is named `$this` and has the value class as type. Generally, in a value class

```
class C(val u: U) extends AnyVal
```

a method

```
def m(params): R = body
```

is expanded to the following method in the companion object of class C:

```
def extension$m($this: C, params): R = body’
```

`Here body’` is the same as `body` with each occurrence of `this` or `C.this` replaced by `$this`. The original method `m` in `C` will be changed to

```
def m(params): R = C.extension$m(this, params)
```

Overloaded methods may be augmented with an additional integer to distinguish them after types are erased (see the transformations of the `divide` method in the following steps).

Also in this step, synthetic `hashCode` and `equals` methods are added to the class.

In our example, the `Meter` class would be expanded as follows:

```
class Meter(val underlying: Double) extends AnyVal with Printable {
   def plus (other: Meter): Meter =
     Meter.extension$plus(this, other)
   def divide (other: Meter): Double =
     Meter.extension1$divide(this, other)
   def divide (factor: Double): Meter =
     Meter.extension2$divide(this, factor)
   def less (other: Meter): Boolean =
     Meter.extension$less(this, other)
   override def toString: String =
     Meter.extension$toString(this)
   override def equals(other: Any) =
     Meter.extension$equals(this)
   override def hashCode =
     Meter.extension$hashCode(this)
}
object Meter {
  def extension$plus($this: Meter, other: Meter) =
    new Meter($this.underlying + other.underlying)
  def extension1$divide($this: Meter, other: Meter): Double =
    $this.underlying / other.underlying
  def extension2$divide($this: Meter, factor: Double): Meter =
    new Meter($this.underlying / factor)
  def extension$less($this: Meter, other: Meter): Boolean =
    $this.underlying < other.underlying
```

```
  def extension$toString($this: Meter): String =
    $this.underlying.toString + “m”
  def extension$equals($this: Meter, other: Any) = other match {
    case that: Meter => $this.underlying == that.underlying
    case _ => false
  }
  def extension$hashCode($this: Meter) = $this.underlying
}
```

Step 2: Rerouting calls

In this step any call to a method that got extracted in step 1 into a companion object gets redirected to the newly created method in that companion object. Generally, a call

```
p.m(args)
```

where `m` is an extractable method declared in an value class `C` gets rewritten to

```
C.extension$m(p, args)
```

For instance the two calls in the following code fragment

```
val x, y: Meter
x.plus(y)
x.toString
```

would be rewritten to

```
Meter.extension$plus(x, y)
Meter.extension$toString(x)
```

Step 3: Erasure

Next, we introduce for each value class `C` a new type `C$unboxed` (this type will be eliminated again in step 4). The newly generated type is assumed to have no members and to be completely outside the normal Scala class hierarchy. That is, it is a subtype of no other type and is a supertype only of `scala.Nothing`.

We now replace every occurrence of the type `C` in a symbol’s type or in a tree’s type annotation by `C$unboxed`. There are however the following two exceptions to this rule:

1. Type tests are left unaffected. So, in the type test below, C is left as it is.

   ```
   e.isInstanceOf[C]
   ```

2. All occurrences of methods in class C are left unaffected.

We then re-typecheck the program, performing the following adaptations if types do not match up.

1. If `e` an expression of type `C$unboxed`, and the expected type is some other type `T`, `e` is converted to type `C` using

   ```
   new C(e.asInstanceOf[U])
   ```

where `U` is the underlying type of `C`. After that, further adaptations may be effected on `C`, employing the usual rules of erasure typing. Similarly, if a selection is performed on an expression of type `C$unboxed`, the expression is first converted to type `C` using the conversion above.

2. If the expected type of an expression `e` of type `T` is `C$unboxed`, then `e` is first adapted with expected type `C` giving `e'`, and `e'` then is converted to `C$unboxed` using

   ```
   e'.u.asInstanceOf[C$unboxed]
   ```

where `u` is the name of the value parameter of `C`. Similarly, if an expression e is explicitly converted using

   ```
   e.asInstanceOf[C$unboxed]
   ```

then `e` is first converted to type `C`, giving `e'`, and the cast is then replaced by

   ```
   e'.u.asInstanceOf[C$unboxed].
   ```

3. The rules for conversions from and to arrays over value classes are analogous to the rules for arrays over primitive value classes.

Value classes are rewritten at this stage to normal reference classes. That is, their parent changes from `AnyVal` to `java.lang.Object`. The `AnyVal` type itself is also rewritten during erasure to `java.lang.Object`, so the change breaks no subtype relationships.

We finally perform the following peephole optimizations:

   ```
   new C(e).u ⇒ e
   ```

```
new C(e).isInstanceOf[C] ⇒ true
new C(e) == new C(f) ⇒ e == f
new C(e) != new C(f) ⇒ e != f
```

Step 4: Cleanup

In the last step, all occurrences of type `C$unboxed` are replaced by the underlying type of `C`. Any redundant casts of the form

```
e.asInstanceOf[T]
```

where `e` is already of type `T` are removed are replaced by `e`.

Example 1:

The program statements on the left are converted using steps 1 to 3 to the statements on the right.

```
var m, n: Meter
var o: AnyRef
m = n
o = m
m.print
m less n
m.toString
m.isInstanceOf[Ordered]

m.asInstanceOf[Ordered]

o.isInstanceOf[Meter]
o.asInstanceOf[Meter]

m.isInstanceOf[Meter]

m.asInstanceOf[Meter]
```

```
var m, n: Meter$unboxed
var o: AnyRef
m = n
o = new Meter(m.asInstanceOf[Double])
new Meter(m.asInstanceOf[Double]).print
Meter.extension$less(m, n)
Meter.extension$toString(m)
new Meter(m.asInstanceOf[Double])
      .isInstanceOf[Ordered]
new Meter(m.asInstanceOf[Double])
      .asInstanceOf[Ordered]
o.isInstanceOf[Meter]
o.asInstanceOf[Meter].underlying
      .asInstanceOf[Meter$unboxed]
new Meter(m.asInstanceOf[Double])
      .isInstanceOf[Meter]
m.asInstanceOf[Meter$unboxed]
```

Including the cleanup step 4 the same program statements are converted as follows.

```
var m, n: Meter
```

```
var m, n: Double
```

```
var o: Any
m = n
o = m
m.print
m less n
m.toString
m.isInstanceOf[Ordered]
m.asInstanceOf[Ordered]
o.isInstanceOf[Meter]
o.asInstanceOf[Meter]
m.isInstanceOf[Meter]
m.asInstanceOf[Meter]
```

```
var o: Any
m = n
o = new Meter(m)
new Meter(m).print
Meter.extension$less(m, n)
Meter.extension$toString(m)
new Meter(m).isInstanceOf[Ordered]
new Meter(m).asInstanceOf[Ordered]
o.isInstanceOf[Meter]
o.asInstanceOf[Meter].underlying
new Meter(m).isInstanceOf[Meter]
m.asInstanceOf[Double]
```

Example 2:

After all 4 steps the Meter class is translated to the following code.

```
 class Meter(val underlying: Double) extends AnyVal with Printable {
   def plus (other: Meter): Meter =
     new Meter(Meter.extension$plus(this.underlying, other.underlying))
   def divide (other: Meter): Double =
     Meter.extension1$divide(this.underlying, other)
   def divide (factor: Double): Meter =
     new Meter(Meter.extension2$divide(this.underlying, factor))
   def less (other: Meter): Boolean =
     Meter.extension$less(this.underlying, other)
   override def toString: String =
     Meter.extension$toString(this.underlying)
   override def equals(other: Any) =
     Meter.extension$equals(this)
   override def hashCode =
     Meter.extension$hashCode(this)
}
object Meter {
  def extension$plus($this: Double, other: Double) =
    $this + other
  def extension1$divide($this: Double, other: Double): Double =
    $this / other
  def extension2$divide($this: Double, factor: Double): Double =
    $this / factor)
  def extension$less($this: Double, other: Double): Boolean =
    $this < other
  def extension$toString($this: Double): String =
```

```
    $this.toString + “m”
  def extension$equals($this: Double, other: Object) = other match {
    case that: Meter => $this == that.underlying
    case _ => false
  }
  def extension$hashCode($this: Double) = $this.hashCode
}
```

Note that the two `divide` methods end up with the same type in object `Meter`.
(The fact that they also have the same body is accidental). That’s why we needed to distinguish them by adding an integer number.

The same situation can arise in other circumstances as well: Two overloaded methods might end up with the same type after erasure. In the general case, Scala would treat this situation as an error, as it would for other types that get erased. So we propose to solve only the specific problem that multiple overloaded methods in a value class itself might clash after erasure.

Further Optimizations?

The proposal foresees that only methods defined directly in a value class get expanded in the companion object; methods inherited from universal traits are unaffected. For instance, in the example above

`m.print` would translate to `new Meter(m).print`

we might at some point want to investigate ways how inherited trait methods can also be inlined. For the moment this is outside the scope of the proposal.