The Scala Programming Language

Troy Hut and Benjamin Killeen

Introduction

Scala is:

Introduction

Scala is:

- Object oriented

Introduction

Scala is:

- Object oriented
- Functional

Introduction

Scala is:

- Object oriented
- Functional
- Frustratingly well-typed

Scala is **Scalable**

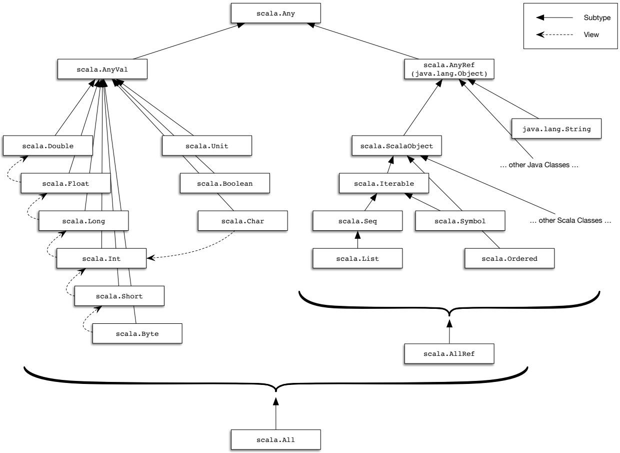


Figure: The Scala class hierarchy.

Traits, Objects, and Classes

```
trait Nat {
  def isZero: Boolean;
  def pred: Nat;
  def succ: Nat;
}
```

Traits, Objects, and Classes

```
trait Nat {
  def isZero: Boolean;
  def pred: Nat;
  def succ: Nat;
}

object Zero extends Nat {
  def isZero = true
  def succ = Succ(this)
  def pred = this
}
```

Traits, Objects, and Classes

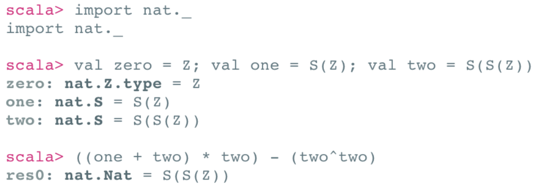
```scala
trait Nat {
  def isZero: Boolean;
  def pred: Nat;
  def succ: Nat;
}

object Zero extends Nat {
  def isZero = true
  def succ = Succ(this)
  def pred = this
}

case class Succ(v: Nat) extends Nat {
  def isZero = false
  def succ = Succ(this)
  def pred = v
}
```

Inferred Methods

```
package nat

trait Nat {
  def isZero: Boolean;
  def pred: Nat;
  def succ: Nat;
  def + (that: Nat): Nat = if (that.isZero) this else this.succ + that.pred
  def - (that: Nat): Nat = if (that.isZero) this else this.pred - that.pred
  def * (that: Nat): Nat = if (that.isZero) that else this + this * that.pred
  def ^ (that: Nat): Nat = if (that.isZero) that.succ else this * (this ^ that.pred)

}
```

Inferred Methods

```
package nat

trait Nat {
  def isZero: Boolean;
  def pred: Nat;
  def succ: Nat;
  def + (that: Nat): Nat = if (that.isZero) this else this.succ + that.pred
  def - (that: Nat): Nat = if (that.isZero) this else this.pred - that.pred
  def * (that: Nat): Nat = if (that.isZero) that else this + this * that.pred
  def ^ (that: Nat): Nat = if (that.isZero) that.succ else this * (this ^ that.pred)

}
```



```
scala> import nat._
import nat._

scala> val zero = Z; val one = S(Z); val two = S(S(Z))
zero: nat.Z.type = Z
one: nat.S = S(Z)
two: nat.S = S(S(Z))

scala> ((one + two) * two) - (two^two)
res0: nat.Nat = S(S(Z))
```

Type Parameters

```scala
package ord

trait Ordering[T] {
  def leq (x: T, y: T): Boolean;
  def geq (x: T, y: T): Boolean = leq(y, x)
  def gt (x: T, y: T): Boolean = !leq(x, y)
  def lt (x: T, y: T): Boolean = !leq(y, x)
  def eq (x: T, y: T): Boolean = leq(x, y) && leq(y, x)
  def neq (x: T, y: T): Boolean = !leq(x, y) || !leq(y, x)
}
```

The nat Package

```scala
package nat

import ord._

trait Nat {
  def isZero: Boolean;
  def pred: Nat;
  def succ: Nat;
  def + (that: Nat): Nat = if (that.isZero) this else this.succ + that.pred
  def - (that: Nat): Nat = if (that.isZero) this else this.pred - that.pred
  def * (that: Nat): Nat = if (that.isZero) that else this + this * that.pred
  def ^ (that: Nat): Nat = if (that.isZero) that.succ else this * (this ^ that.pred)
  def <= (that: Nat): Boolean = NatOrdering.leq(this, that)
  def >= (that: Nat): Boolean = NatOrdering.geq(this, that)
  def <  (that: Nat): Boolean = NatOrdering.lt(this, that)
  def >  (that: Nat): Boolean = NatOrdering.gt(this, that)
  def == (that: Nat): Boolean = NatOrdering.eq(this, that)
  def != (that: Nat): Boolean = NatOrdering.neq(this, that)
}

object NatOrdering extends Ordering[Nat] {
  def leq (x: Nat, y: Nat): Boolean =
    (x.isZero, y.isZero) match {
      case (true, _) => true
      case (false, true) => false
      case (false, false) => leq(x.pred, y.pred)
    }
}
```

The nat Package

```
package nat

import ord._

trait Nat {
  def isZero: Boolean;
  def pred: Nat;
  def succ: Nat;
  def + (that: Nat): Nat = if (that.isZero) this else this.succ + that.pred
  def - (that: Nat): Nat = if (that.isZero) this else this.pred - that.pred
  def * (that: Nat): Nat = if (that.isZero) that else this + this * that.pred
  def ^ (that: Nat): Nat = if (that.isZero) that.succ else this * (this ^ that.pred)
  def <= (that: Nat): Boolean = NatOrdering.leq(this, that)
  def >= (that: Nat): Boolean = NatOrdering.geq(this, that)
  def <  (that: Nat): Boolean = NatOrdering.lt(this, that)
  def >  (that: Nat): Boolean = NatOrdering.gt(this, that)
  def == (that: Nat): Boolean = NatOrdering.eq(this, that)
  def != (that: Nat): Boolean = NatOrdering.neq(this, that)
}

object NatOrdering extends Ordering[Nat] {
  def leq (x: Nat, y: Nat): Boolean =
    (x.isZero, y.isZero) match {
      case (true, _) => true
      case (false, true) => false
      case (false, false) => leq(x.pred, y.pred)
    }
}
```

```
scala> one < two
res5: Boolean = true

scala> one == two
res6: Boolean = false

scala> (one + two) == S(S(S(Z)))
res7: Boolean = true
```

Variance

Covariance

If `C` is a *covariant* type constructor and `S <: T`, then `C[S] <: C[T]`

Variance

Covariance

If `C` is a *covariant* type constructor and `S <: T`, then `C[S] <: C[T]`

Contravariance

If `C` is a *contravariant* type constructor and `S <: T`, then `C[S] >: C[T]`

Ordered List

```scala
package ordlist
import ord._

trait OrdList[+T] {
  def isEmpty: Boolean;
  def insert[U >: T](x: U, o: Ordering[U]): OrdList[U];
}

object Emp extends OrdList[Nothing] {
  override def toString = "Emp"
  def isEmpty = true
  def insert[U >: Nothing](x: U, o: Ordering[U]) =
    Cons(x, this.asInstanceOf[OrdList[U]])
}

case class Cons[+T] (head: T, tail: OrdList[T]) extends OrdList[T] {
  override def toString = s"$head :: $tail"
  def isEmpty = false
  def insert[U >: T](x: U, o: Ordering[U]) =
    if (o.leq(x, head)) Cons(x, this) else Cons(head, tail.insert(x, o))
}
```