CS 3101-2 - Programming Languages: Scala

Lecture 2: Classes and Objects / Inheritance / Imports

Daniel Bauer (bauer@cs.columbia.edu)

October 29, 2014

Contents

Some List Functions

Indexing and slicing

```
scala> val lst = 42 :: 23 :: 5 :: Nil;
lst: List[Int] = List(42, 23, 5)

scala> lst(0) // indexing
res2: Int = 42

scala> lst.slice(1,3)  // slicing
res4: List[Int] = List(23, 5)
```

More List Functions

Reversing and sorting

```
scala> val lst = List(1,3,2)
lst: List[Int] = List(1, 3, 2)

scala> lst.reverse  // Reverse the list
res1: List[Int] = List(2, 3, 1)

scala> lst.sorted
res2: List[Int] = List(1, 2, 3)
```

Parametric Types

- Typically want to specify type of elements of a collection.
- Using generic classes.

```
scala> val x : List[Int] = 1 :: 2 :: 3 :: Nil
x: List[Int] = List(1, 2, 3)

scala> val y : List[Int] = 1 :: 2 :: "Hello" :: Nil
<console>:7: error: type mismatch;
 found   : List[Any]
 required: List[Int]
       val y : List[Int] = 1 :: 2 :: "Hello" :: Nil
```

Parametric Types

- Typically want to specify type of elements of a collection.
- Using generic classes.

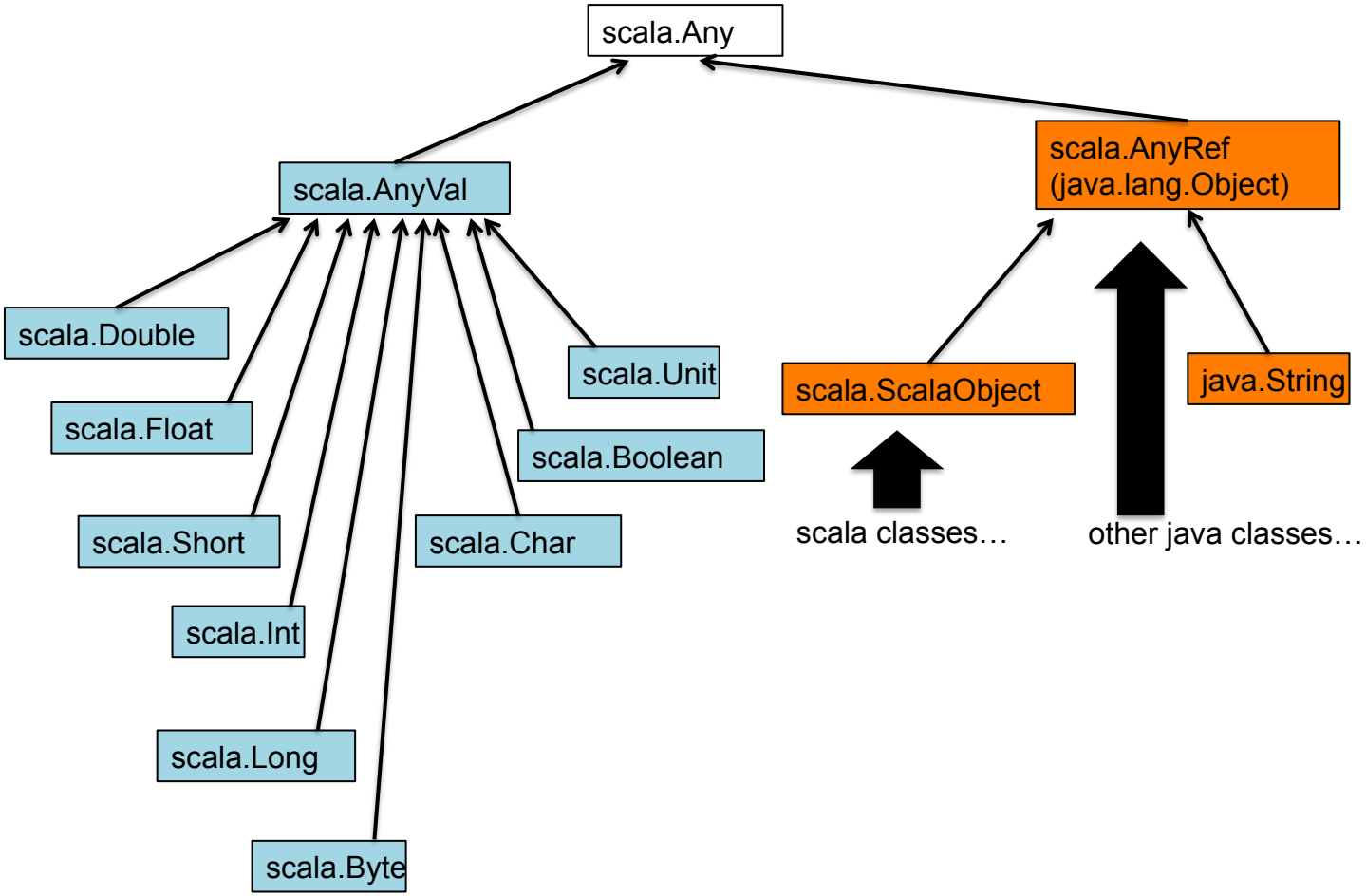
```
scala> val x : List[Int] = 1 :: 2 :: 3 :: Nil
x: List[Int] = List(1, 2, 3)

scala> val y : List[Int] = 1 :: 2 :: "Hello" :: Nil
<console>:7: error: type mismatch;
 found   : List[Any]
 required: List[Int]
       val y : List[Int] = 1 :: 2 :: "Hello" :: Nil

scala> val x = 1 :: 2 :: "Hello" :: Nil
x: List[Any] = List(1, 2, Hello)

scala> x(2)  // Don't know specific type of this element
res0: Any = Hello
```

Scala's Type Hierarchy



1. Object Oriented Programming
   - Classes, Methods and Instances
   - Singleton Objects
   - Inheritance

2. Packages

Object Oriented Programming

- Objects contain complex collections of data.
- Objects have some functionality to operate on the data.
  - Mutate some data in the object.
  - Perform some computation using data within the object and return the result.
- Classes define *blueprints* for Objects.

A Data Type for Rational Numbers

```
scala> val oneHalf = new Rational(1,2)
oneHalf: Rational = 1/2

scala> val twoThirds = new Rational(2,3)
oneHalf: Rational = 2/3

scala> (oneHalf / 7) + (1 - twoThirds)
res0: Rational = 17/42
```

Defining Classes

```
scala> class Rational(n: Int, d: Int)
defined class Rational

scala> val oneHalf = new Rational(1,2)
oneHalf: Rational = Rational@58c1a471
```

Classes: Primary Constructor

```
class Rational(n: Int, d: Int) {
    println("Created " + n + "/" + d)
}

scala> val r = new Rational(1,2)
Created 1/2
r: Rational = Rational@3394da56
```

Classes: Adding Methods

```
class Rational(n: Int, d: Int) {
    def +(other : Rational): Rational = {
        val new_n = n * other.d + other.n * d
        val new_d = d * other.d
        new Rational(n * other.d + other.n * d)
    }
}
```

Classes: Adding Methods

```
class Rational(n: Int, d: Int) {
    def +(other : Rational): Rational = {
        val new_n = n * other.d + other.n * d
        val new_d = d * other.d
        new Rational(n * other.d + other.n * d)
    }
}
```

```
$ scalac Rational.scala
Rational.scala:5: error: value d is not a member of
this.Rational
        val new_n = n * other.d + other.n * d
                            ^
...
```

Defining Classes: Adding fields

```
class Rational(n: Int, d: Int) {

    val numer = n
    val denom = d

    def +(other : Rational): Rational = {
        val new_n = numer * other.denom + other.numer * denom
        val new_d = denom * other.denom
        new Rational(new_n, new_d)
    }
}
```

Defining Classes: Adding fields

```
class Rational(n: Int, d: Int) {

    val numer = n
    val denom = d

    def +(other : Rational): Rational = {
        val new_n = numer * other.denom + other.numer * denom
        val new_d = denom * other.denom
        new Rational(new_n, new_d)
    }
}

scala> new Rational(1,2) + new Rational(3,4)
res4: Rational = Rational@2a491adf
```

Defining Classes: Overriding Methods

```
class Rational(n: Int, d: Int) {

    val numer = n
    val denom = d

    override def toString = numer + "/" + denom

    def +(other : Rational): Rational = {
        val new_n = numer * other.denom + other.numer * denom
        val new_d = denom * other.denom
        new Rational(new_n, new_d)
    }
}

scala> new Rational(1,2) + new Rational(3,4)
res4: Rational = 10/8
```

Classes: Auxiliary Constructors

```
class Rational(n: Int, d: Int) {

    val numer = n
    val denom = d

    def this(n : Int) = this(n, 1) // auxiliary constructor

    override def toString = numer + "/" + denom

    def +(other : Rational): Rational = {
        val new_n = numer * other.denom + other.numer * denom
        val new_d = denom * other.denom
        new Rational(new_n, new_d)
    }
}

scala> new Rational(3)
Res0: Rational(3/1)
```

Classes: Private Methods and Fields

```
class Rational(n: Int, d: Int) {

    // Private val to store greatest common
    // divisor of n and d
    private val g = gcd(n.abs, d.abs)

    val numer = n / g
    val denom = d / g

    def this(n : Int) = this(n, 1) // auxiliary constructor

    override def toString = numer + "/" + denom

    def +(other : Rational): Rational = {
        val new_n = numer * other.denom + other.numer * denom
        val new_d = denom * other.denom
        new Rational(new_n, new_d)
    }

    // Private method to compute the greatest common divisor
    private def gcd(a : Int, b : Int) : Int =
      if (b==0) a else gcd(b, a % b)

}

scala> new Rational(2/4)
Res0: Rational(1/2)
```

Singleton Objects

- There are no static methods in Scala.
- Instead Scala supports *singleton objects*.

Singleton Objects

- There are no static methods in Scala.
- Instead Scala supports *singleton objects*.
- Many use cases:
  - Single point access to a common resource (large data structures...)
  - Repository for utility methods.
  - Companion objects for classes (same name as Class) to define "static" methods and factories.
  - Writing Scala applications.

Singleton Objects

- There are no static methods in Scala.
- Instead Scala supports *singleton objects*.
- Many use cases:
  - Single point access to a common resource (large data structures...)
  - Repository for utility methods.
  - Companion objects for classes (same name as Class) to define "static" methods and factories.
  - Writing Scala applications.

```
object RationalSummer {
    var sum : Double = 0.0
    def add(r : Rational)  = { sum += r.numer.toDouble
                               / r.denom.toDouble; sum}
}
```

Companion Objects and apply

```
object Rational {  // Companion object for the class Rational

    def invertRational(r : Rational) =
                    new Rational(r.denom, r.numer)

    def apply(n: Int, d: Int) = new Rational(n,d)
    def apply(n: Int) = new Rational(n)
}

scala> val r =  Rational(1,2) + Rational(3)
r: Rational = 7/2
scala> Rational.invertRational(r)
res0: Rational = 2/7
```

- Scala converts `f(a)` into `f.apply(a)`.

Scala Applications

file `FractionApp.scala`

```
object FractionApp {
    def main(args: Array[String]) {
        println(Rational(1,2) + Rational(2,3))
    }
}
```

Scala Applications

file `FractionApp.scala`

```
object FractionApp {
    def main(args: Array[String]) {
        println(Rational(1,2) + Rational(2,3))
    }
}
```

```
$ scalac FractionApp.scala
$ ls
FractionApp$.class
FractionApp.class
FractionApp.scala
Rational$.class
Rational.class
Rational.scala
RationalSummer$.class
RationalSummer.class
$ scala FractionApp
7/6
```

Inheritance

```
class Rectangle(w: Double, h: Double) {
    def area = w * h
    val description = "Rectangle"
}
```

Inheritance

```
class Rectangle(w: Double, h: Double) {
    def area = w * h
    val description = "Rectangle"
}

class Square(w: Double) extends Rectangle(w, w) {
    override val description = "Square"
}
```

Abstract Classes

```
abstract class Shape {
    def area : Double
    val description : String
    override def toString = description + ", size: "+area
}

class Rectangle(w: Double, h: Double) extends Shape{
    def area = w * h
    val description = "Rectangle"
}

class Square(w: Double) extends Rectangle(w, w) {
    override val description = "Square"
}


scala> val x = new Square(3)
x: Square = Square, size: 9.0
```

Overriding Methods and Fields

```
abstract class Shape {
    def area : Double
    val description : String
    override def toString = description + ", size: "+area
}

class Blob extends Shape {
    val area : Double = 12;
    val description = "Blob"
}

scala> val x = new Blob
x: Blob = Blob, size: 12.0
```

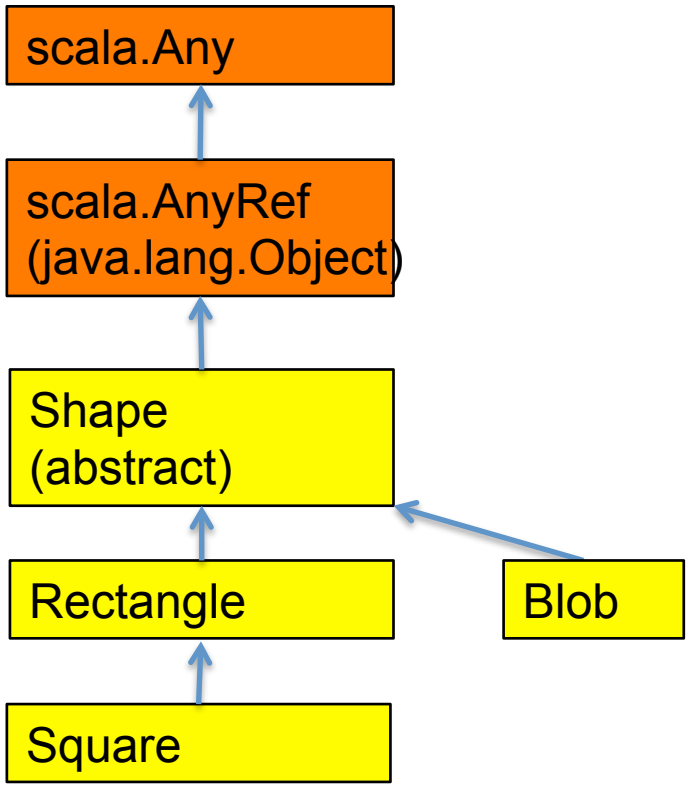
Making Members Final

```
abstract class Shape {
    def area : Double
    final val description : String = "Shape"
    override def toString = description + ", size: "+area
}

class Blob extends Shape {
    val area : Double = 12;
    val description = "Blob"
}

<console>:10: error: overriding value description in class
Shape of type String;
value description cannot override final member
           val description = "Blob"
                   ^
```

Class Diagram for Shapes



Polymorphism and Dynamic Binding

- Call a different method/value based on the object type.

```
scala> val x = new Rectangle(2,3)
x: Rectangle = Rectangle, size: 6.0

scala> val y = new Square(5)
y: Square = Square, size: 25.0

scala> val z = new Blob
z: Blob = Blob, size: 12.0
```

Polymorphism and Dynamic Binding

- Call a different method/value based on the object type.

```
scala> val x = new Rectangle(2,3)
x: Rectangle = Rectangle, size: 6.0

scala> val y = new Square(5)
y: Square = Square, size: 25.0

scala> val z = new Blob
z: Blob = Blob, size: 12.0

scala> val l : List[Shape] = List(x,y,z)
l: List[Shape] = List(Rectangle, size: 6.0,
                      Square, size: 25.0,
                      Blob, size: 12.0)
```

Polymorphism and Dynamic Binding

- Call a different method/value based on the object type.

```
scala> val x = new Rectangle(2,3)
x: Rectangle = Rectangle, size: 6.0

scala> val y = new Square(5)
y: Square = Square, size: 25.0

scala> val z = new Blob
z: Blob = Blob, size: 12.0

scala> val l : List[Shape] = List(x,y,z)
l: List[Shape] = List(Rectangle, size: 6.0,
                      Square, size: 25.0,
                      Blob, size: 12.0)

scala> for (x<-l) println(x.description+" "+x.area)
Rectangle 6.0
Square 25.0
Blob 12.0
```

Defining Generic Classes

```
class Stack[T] {
  var elems: List[T] = Nil
  def push(x: T) { elems = x :: elems }
  def top: T = elems.head
  def pop() { elems = elems.tail }
}
```

1. Object Oriented Programming
   - Classes, Methods and Instances
   - Singleton Objects
   - Inheritance

2. Packages

Creating Packages

- Goal: Modularize programs, so that parts of it can be re-used.
- Package are special objects that define a set of member classes, objects and other packages.

Creating Packages

- Goal: Modularize programs, so that parts of it can be re-used.
- Package are special objects that define a set of member classes, objects and other packages.
- Option 1: package statement at the beginning of file.

```
package bobsrockets.navigation

class Navigator
```

Creating Packages

- Goal: Modularize programs, so that parts of it can be re-used.
- Package are special objects that define a set of member classes, objects and other packages.
- Option 1: package statement at the beginning of file.

```
package bobsrockets.navigation

class Navigator
```

- Option 2: multiple packages in one file

```
package bobsrocket.navigation{
    class Navigator
}
```

Creating Packages - Nested Packages

```
package bobsrocket{
    package navigation{

        class Navigator

        package tests {
            class NavigatorSuite
        }
    }
}
```

Packages - Concise Access To Related Code

```
package bobsrocket{
    package navigation{
        class Navigator {
            val map = new StarMap
        }
        class StarMap
    }
    class Ship {
        val nav = new navigation.Navigator
    }
    package fleets {
        class Fleet {
            def addShip() = new Ship
        }
    }
```

Packages - Imports

- Can always use fully qualfied package name

```
val navigator = new bobsrocket.navigation.Navigator
```

Packages - Imports

- Can always use fully qualfied package name

```
val navigator = new bobsrocket.navigation.Navigator
```

- `import` allows you to access items by their name alone (without prefix)

```
import bobsrocket.navigation.Navigator
val navigator = new Navigator
```

Packages - Imports

- Can always use fully qualfied package name

```
val navigator = new bobsrocket.navigation.Navigator
```

- `import` allows you to access items by their name alone (without prefix)

```
import bobsrocket.navigation.Navigator
val navigator = new Navigator
```

- Or use wildcards

```
import bobsrocket.fleet
val navigator = new Ship
```

Packages - Imports

- Can always use fully qualfied package name

```
val navigator = new bobsrocket.navigation.Navigator
```

- `import` allows you to access items by their name alone (without prefix)

```
import bobsrocket.navigation.Navigator
val navigator = new Navigator
```

- Or use wildcards

```
import bobsrocket.fleet
val navigator = new Ship
```

- Or import the package itself import bobsrocket.navigator val navigator = navigator.Navigator

import Is Even More Flexible

- Can use import anywhere in code.
- Can import specific methods of singleton objects...

```
def printAndAddFraction(r : Rational) = {
    println(r)
    import RationalSummer.add
    val total = add(r)
}
```

import Is Even More Flexible

- Can use import anywhere in code.
- Can import specific methods of singleton objects...

```
def printAndAddFraction(r : Rational) = {
    println(r)
    import RationalSummer.add
    val total = add(r)
}
```

- ...or members of any other object.

```
def printDenominator(r : Rational) = {
    import r._
    println(denom);
}
```