CSCI 334:
Principles of Programming Languages

Lecture 16: Intro to Scala

Instructor: Dan Barowy

Williams

Announcements

HW7 sent out as promised. See course webpage.

Announcements

No class on Tuesday, April 17.

Squeak demo

Scala!

The Programming World Today

"Tower of Babel"

OO vs Functional Tradeoff

Operation	Doctor	Nurse	Orderly
`Print`	`Print Doctor`	`Print Nurse`	`Print Orderly`
`Pay`	`Pay Doctor`	`Pay Nurse`	`Pay Orderly`

- Functional programming makes it easy to add operations.
- OO programming makes it easy to add data.
- Scala: Why not have both functional and OO?

REPL

```
$ scala
Welcome to Scala 2.12.5 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_144).
Type in expressions for evaluation. Or try :help.

scala>

scala> "hello world!"
res0: String = hello world!

scala> :quit
```

Semicolons are optional

```
scala> println("Hello world!")
Hello world!

scala> println("Hello world!");
Hello world!
```

Scala is object-oriented

```
scala> class Apple
defined class Apple

scala> val a = new Apple
a: Apple = Apple@31b7d869
```

Everything is an object!

Scala is functional

```
scala> val xs = List(1,2,3,4,5)

scala> xs.map(e => e + 1)
res0: List[Int] = List(2, 3, 4, 5, 6)

scala> xs.map(_ + 1)
res1: List[Int] = List(2, 3, 4, 5, 6)
```

Scala is functional

Supports many of your favorite HOFs (and then some!)

```
scala> xs.foldLeft (0)((acc,x) => acc + x)
res0: Int = 15

scala> xs.zip(xs)
res1: List[(Int, Int)] = List((1,1), (2,2), (3,3), (4,4), (5,5))

scala> val m = xs.groupBy(x => x > 3)
m: scala.collection.immutable.Map[Boolean,List[Int]] = Map(false ->
List(1, 2, 3), true -> List(4, 5))

scala> m(false)
res2: List[Int] = List(1, 2, 3)

scala> m(true)
res3: List[Int] = List(4, 5)

scala> m(true).head
res4: Int = 4
```

Scala is functional

Values are immutable

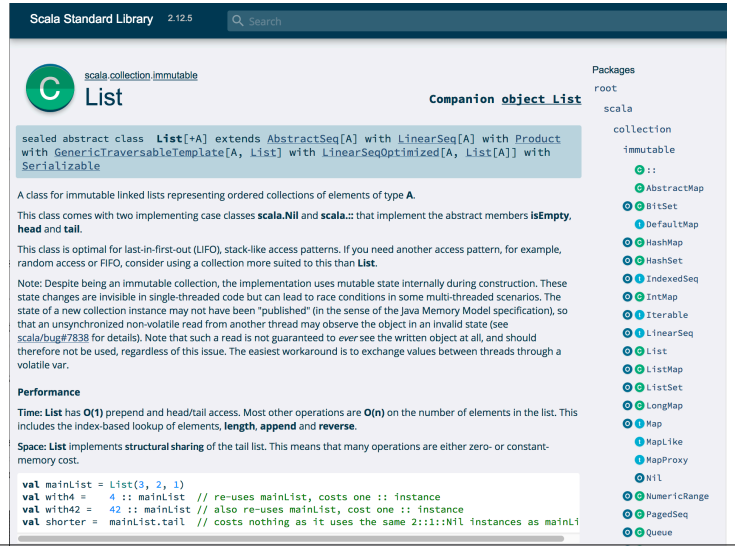
```
scala> class Thing {
     |   val i = 1
     |   def increment() { i += 1 }
     | }
<console>:13: error: value += is not a member of Int
  Expression does not convert to assignment because receiver is not
assignable.
         def increment() { i += 1 }
                           ^
```

But Scala is also pragmatic

You can also use mutable variables

```
scala> class Thing {
     |   var i = 1
     |   def increment() { i += 1 }
     | }
defined class Thing

scala> val t = new Thing
t: Thing = Thing@28d728f1

scala> t.increment

scala> t.i
res0: Int = 2
```

Scala has great documentation



Ordinary Functions

```
scala> def succ(x: Int) = x + 1;
succ: (x: Int)Int

scala> succ(12);
res0: Int = 13
```

Lambda (Anonymous) Functions

```
scala> val succ = (x : Int) => x + 1;
succ: Int => Int = $$Lambda$1514/3223023980@2fe12b04

scala> succ(3)
res0: Int = 4
```

Recursive Functions

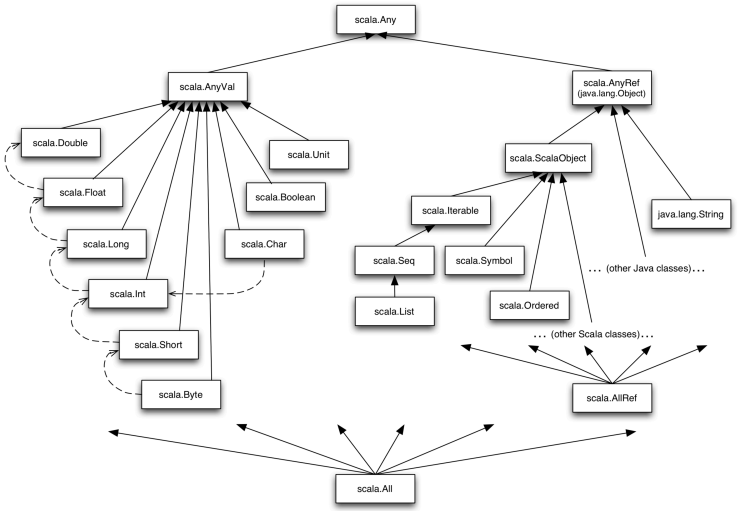
```
scala> def fact(n: Int) : Int =
     |   if (n == 0) 1 else n * fact(n-1)
fact: (n: Int)Int

scala> fact(4)
res0: Int = 24
```

Scala is built on top of Java

In general, Java classes and methods are available.

```
scala> val sb = new StringBuilder
sb: StringBuilder =

scala> sb.append("hello")
res0: StringBuilder = hello

scala> sb.append("world")
res1: StringBuilder = helloworld

scala> println(sb.toString)
helloworld
```

Scala has a rich set of built-in types



Scala has a rich set of built-in types

```
scala> true
res0: Boolean = true

scala> false
res1: Boolean = false

scala> 3
res2: Int = 3

scala> 43.3
res3: Double = 43.3
```

Most types fully compatible with Java

```
scala> "moo"
res8: java.lang.String = moo

scala> val str = "cow"
str: java.lang.String = cow

scala> str.length()
res9: Int = 3

scala> str.toUpperCase()
res10: java.lang.String = COW
```

Lightweight tuple syntax (like SML!)

```
scala> (1,"hello")
res0: (Int, String) = (1,hello)
```

You can abbrev. no-param calls

```
scala> str.length()
res9: Int = 3

scala> str.toUpperCase()
res10: java.lang.String = COW

scala> str.toUpperCase
res11: java.lang.String = COW

scala> str toUpperCase
res12: java.lang.String = COW
```

Scala has pattern matching

```
scala> val thing : Option[Int] = Some(3)
thing: Option[Int] = Some(3)

scala> thing match {
     |   case None => println("It was nothing")
     |   case Some(i) => println(i)
     | }
3
```

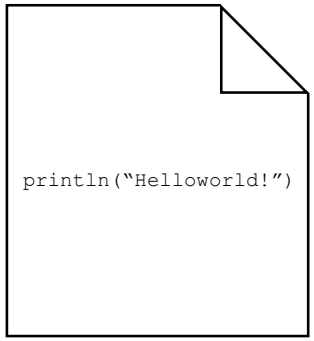
Scala has generics

```
scala> def foo[T](data: T) { println(data) }
foo: [T](data: T)Unit

scala> foo(1)
1

scala> foo("hello")
hello

scala> foo((1,"hello"))
(1,hello)
```

Scala has “lighter” syntax than Java

Scala programs can just be “scripts”

No need for “boilerplate”.



hello.scala

```
$ scala hello.scala
Helloworld!
```

Also supports traditional structure

Scala programs can also be compiled just like Java

```
object App {
  def main(args: Array[String]) {
    println("Helloworld!")
  }
}
```

```
$ scalac hello.scala
$ scala App
Helloworld!
```

Scala doesn't care where you put classes

Doesn't have Java's restrictive one class per file rule

```
class Apple {
  def whatami = "apple"
}

object App {
  def main(args: Array[String]){
    val apple = new Apple
    println(apple.whatami)
  }
}


$ scalac cool.scala
$ scala App
apple
```

Scala doesn't care where you put classes

You can even nest classes arbitrarily

```
class Apple {
  def whatami = "apple"
}

object App {
  class Orange {
    def whatami = "orange"
  }

  def main(args: Array[String]){
    val apple = new Apple
    val orange = new Orange
    println(apple.whatami + " " + orange.whatami)
  }
}

$ scalac hello.scala
$ scala App
apple orange
```

Scala has powerful facilities for abstraction

```
trait Fruit {
  def name: String
}

trait Box {
  def fruit: Fruit
  def contains(aFruit: Fruit) = fruit == aFruit
}

trait Color {
  def color: String
}

class Apple extends Fruit {
  def name = "Apple"
}

class AppleBox(apple: Apple) extends Box with Color {
  def fruit = apple
  def color = "brown"
}
```

Anonymous classes

```
scala> val apple = new Apple
apple: Apple = Apple@4f8659d0

scala> val ab = new Box { def fruit = apple }
ab: Box{def fruit: Apple} = $anon$1@1c011855
```

We can even “refine” types

F must be a subtype of Fruit

```
trait Box[F <: Fruit] {
  def fruit: F
  def contains(aFruit: Fruit) = fruit == aFruit
}
```

We can even “refine” types

F must be a subtype of Fruit

```
trait Box[F <: Fruit] {
  def fruit: F
  def contains(aFruit: Fruit) = fruit == aFruit
}
```

But now this doesn't work. Why?

```
val box: Box[Fruit] = new Box[Apple] { def fruit = apple }
```

Box is not “covariant”

What we want:

```
F <: Fruit

Box[F] <: Box[Fruit]
```

This is not true in Scala by default

(but the fix is simple)

```
trait Box[+F <: Fruit] {
  def fruit: F
  def contains(aFruit: Fruit) = fruit == aFruit
}

val box: Box[Fruit] = new Box[Apple] { def fruit = apple }
```