CS 696 Intro to Big Data: Tools and Methods
Fall Semester, 2017
Doc 5 Scala Classes & Magic
Sep 11, 2017

Sample Class

```
class Fraction {
    var numerator = 0
    private var denominator = 0

    def set(x: Int) = {this.denominator = x}

    override def toString()= numerator + "/" + denominator
}
```

```
val test = new Fraction
test.numerator = 10
test.set(3)
println(test)
```

Protection Levels

- private
  - Same as Java/C++
  - Accessible only in the class
- protected
  - Accessible in class
  - Accessible in subclasses
  - Not accessible other places
- public
  - Accessible in any class or function
  - with reference to object
  - Default protection level

Any

Root of class hierarchy

!=
==
asInstanceOf
equals
hashCode
isInstanceOf
toString

Class Parameters

```
class Foo(var a: Int, val b: Int, c: Int) {
   override def toString() = "Foo: " + a + " " + b + " " + c
}
```

```
val x = new Foo(1,2,3)
x.a = 4
println(x.a)
println(x.b)
println(x.c) //Compile error
println(x)
x.b = 5//Compile error
x.c = 6//Compile error
```

Parameterless Methods

```
class Foo {
   var x = 0
   def a: Int = x
   def b(): Int = x
}
```

```
var result = 0
val test = new Foo
test.x = 1
result = test.x
result = test.a
result = test.b
result = test.b()
result = test.a() // Compile error
```

Explicit Setters & Getters

```
class Foo {
   var x = 0
   def a: Int = x
   def a_=(b:Int) {x = b}
   }
```

```
class Foo {
   private var privateA = 0
   def a: Int = privateA
   def a_=(b:Int) {privateA = b}
   }
```

```
var result = 0
val test = new Foo
test.x = 1
result = test.x
test.a = 2
test.x == 2
test.x = 12
12 == test.a
```

Constructors

```
class Fraction(n: Int, d: Int) {
    println("Start")
    private var numerator = n
    private var denominator = d

    def this(x: Int) = {this(x,1); println("auxiliary")}

    override def toString()= numerator + "/" + denominator

    println("End")
}
```

```
val test = new Fraction(1,2)
```

Output
Start
End

```
val two = new Fraction(2)
```

Output
Start
End
auxiliary

Operators & Overloading

```
class Fraction(n: Int, d: Int) {
    private var numerator = n
    private var denominator = d

    def this(x: Int) = {this(x,1); println("auxiliary")}

    def *(that: Int) = new Fraction(numerator*that, denominator)

    def *(that: Fraction) = new Fraction(numerator*that.numerator,
        denominator*that.denominator)

    override def toString()= numerator + "/" + denominator
}
```

Using the Operators

```
val halve = new Fraction(1,2)
var one = halve * 2
println(one)                    //prints 2/2

val two = new Fraction(2)
one = halve * two
println(one)                    //prints 2/2
```

But

```
val halve = new Fraction(1,2)
var one =  2 * halve                    //Compile Error
```

Implicit Conversions

```
implicit def intToFraction(x: Int) = new Fraction(x)

val test:Fraction = 2

val halve = new Fraction(1,2)
var one =  2 * halve
println(one)                                    //Prints 2/2
```

require

```
class Fraction(n: Int, d: Int) {
   require(d != 0)

   private var numerator = n
   private var denominator = d

   def this(x: Int) = {this(x,1); println("auxiliary")}
   def *(that: Int) = new Fraction(numerator*that, denominator)
   def *(that: Fraction) = new Fraction(numerator*that.numerator,
      denominator*that.denominator)
   override def toString()= numerator + "/" + denominator
}
```

```
new Fraction(1,0)
//Causes exception
```

No static fields or methods

Use singleton objects

Singleton Objects

```
object JustOne {
   private var x = 0
   def getX(): Int = x
   def setX(x: Int) = this.x = x

   override def toString() = "JustOne: " + x
}
```

```
JustOne.setX(10)
println(JustOne.getX())

println(JustOne)

new JustOne // compile error
```

Companion objects & classes

```
class Fraction(n: Int, d: Int) {
    private var numerator = n
    private var denominator = d

    override def toString()=
   numerator + "/" + denominator
}

object Fraction {
    def zero() = new Fraction(0,1)
    def unity() = new Fraction(1,1)
}
```

```
val a = Fraction.zero()
val b = new Fraction(1,2)
```

Scala Application

```
object StartHere {
    def main(args: Array[String]) {
        for (arg <- args)
            println(arg)
    }
}
```

```
StartHere.main(Array("this", "is", "a", "test"))
```

Output
this
is
a
test

Object & primary constructor

```
object Foo {
   private val x = 3

   println("Before main")

   def main(args: Array[String]) {
      println("In Main")
   }
   println("After main")
}
```

```
Foo.main(Array("test"))
```

Output

```
Before main
After main
In Main
```

Traits

```
trait Example {
    val a: String
    val b = "bb"
    def bar(x:Int) = x + 1
    def foo(x:String): String
}
```

```
class A extends Example {
    val a = "aa"
    def foo(x:String) = b + x
}

class Parent {
    override def toString = "Parent"
}

class Childs extends Parent with Example {
    val a = "aa"
    def foo(x:String) = b + x
}

object Test extends Example {
    val a = "aa"
    def foo(x:String) = b + x
}
```

Magic

Where is println defined?

What other functions can we call?

```
object Foo extends Application{
   private val x = 3
   println("Why use main")
   println("If you don use the args?")
}
```

Where is println defined?

object Predef

Part of standard Scala library

Is imported in all Scala files

defines many methods

Magic Trick 1

```
val result = 10!
```

```
def factorial(n: BigInt): BigInt = {
  def factorial(n: BigInt, accumulator: BigInt): BigInt = {
    if (n <= 1)
      accumulator
    else
      factorial(n - 1, n * accumulator)
  }
  factorial(n, 1)
}

class Factorial(n :BigInt) {
   def !():BigInt = factorial(n)
}

implicit def intToFactorial(x: Int) = new Factorial(x)
```

Magic Trick 2

```
def repeatWhile(condition: => Boolean)(code: => Unit) {
   while (condition) {
      code
   }
}
```

```
var x = 0
repeatWhile (x < 4) {
   println(x)
   x += 1
}
```

Output

```
0
1
2
3
```

Blocks as Arguments

```
var y = 1;
def bar(x: Int) = {
   println("In Bar");
   println(x)
}

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

Expression	Output
bar(y + 1)	In Bar 2
bar({y + 1})	In Bar 2
bar({ println("Call Bar"); y + 1 })	Call Bar In Bar 2
bar(println("Call Bar"); y + 1)	Compile Error
bar{ println("Call Bar"); y + 1}	Call Bar In Bar 2

Magic Trick 3

```
class Repeat(code: => Unit) {
  def until(condition: => Boolean) = {
    while (!condition) { code }
  }

  def when(condition: => Boolean) = {
    while (condition) { code }
  }
}

def repeat(code: => Unit) = new Repeat(code)
```

```
var x = 0
repeat {
  println(x)
  x += 1
} when (x < 5)

var y = 0
repeat {
  println(y)
  y += 1
} until (y == 3)
```