Classes, Objects, Apps and More

Agenda

A Scala Class Definition

- On the JVM, all methods/fields must go inside classes (unlike the REPL)
- In Scala, this includes objects, traits, and package objects (more later)

```scala
class DemoWithFieldsAndMethods {
  val x: Int = 10
  val y: Int = x * 2

  def timesY(a: Int): Int = a * y
}
```

```scala
class DemoWithParams(name: String) {
  println(s"Constructing for $name")

  def sayHi(times: Int): Unit = {
    var time = 0

    while (time < times) {
      println(s"Hi, $name")
      time += 1
    }
  }
}
```

Constructor

```
class DemoWithParams(name: String) {
  println(s"Constructing for $name")

  // rest of class...
}
```

- Parameters on the class definition become *primary constructor* parameters
- Code in the class (not in defs) becomes the *primary constructor* code, runs when a new instance is constructed
- Can't access the constructor parameters from outside (private)

```
val demo = new DemoWithParams("Jill")
demo.name
// Error:(33, 83) value name is not a member of DemoWithParams
```

Parameters, Fields and Parametric Fields

- Constructor parameters are `private` (actually `private[this]`), also `val`s
- `private` and `protected` are keywords, there is no `public` keyword, that's the default for `val`s and `def`s (but not for constructor parameters)
- Adding a `val` keyword before the parameter definition makes it a `public` *parametric field*:

```scala
class DemoWithParams(val name: String) {
  println(s"Constructing for $name")
}

val demo = new DemoWithParams("Jill")
demo.name  // Jill
```

- Parametric fields are idiomatic Scala (remember they are `val`s)

```scala
// how Scala re-writes the above:
class DemoWithParams(_name: String) {  // parameter still private[this]
  val name: String = _name    // the public field definition
  println(s"constructing for $name")
}
```

A Rational Class

- As in, let's make something to represent a Rational number from what we know so far:

```
class Rational(val n: Int, val d: Int)  // look ma, no body!

val half = new Rational(1, 2)
// half: Rational = Rational@6c643605
```

- Every class has a `toString` method that can be overridden:

```
class Rational(val n: Int, val d: Int) {
  override def toString: String = s"R($n/$d)"
}

val half = new Rational(1, 2)
// half: Rational = R(1/2)

val divByZero = new Rational(1, 0)
// divByZero: Rational = R(1/0) -- probably should prevent this
```

Checking Preconditions in the Constructor

```scala
class Rational(val n: Int, val d: Int) {
  require(d != 0, "Zero denominator!") // precondition

  override def toString: String = s"R($n/$d)"
}

val half = new Rational(1, 2)
// half: Rational = R(1/2)

val divByZero = new Rational(1, 0)
// java.lang.IllegalArgumentException: requirement failed: Zero denominator!
```

- If you use `require` and the predicate fails, you will get an `IllegalArgumentException` thrown
- The String field in `require` is optional but recommended

Referencing Self

- Could also write `require(this.d != 0, "Zero denominator!")`
- this is a reference to the current instance. It is inferred by Scala when possible

```scala
class Rational(val n: Int, val d: Int) {
  require(d != 0, "Zero denominator!") // precondition

  override def toString: String = s"R($n/$d)"

  def min(other: Rational): Rational =
    if ((n.toDouble / d) < (other.n.toDouble / other.d))
      this else other   // have to use this to return
}

val half = new Rational(1, 2)
val fifth = new Rational(1, 5)

val smaller = fifth min half
// smaller: Rational = R(1/5)
```

- Could have used `this` for the n and d references in `min`
- Note also infix use of `min` method, equivalent to `fifth.min(half)`

Infix and Symbolic Methods

```
class Rational(val n: Int, val d: Int) {
  require(d != 0, "Zero denominator!")

  override def toString: String = s"R($n/$d)"

  // rational addition
  def add(other: Rational): Rational =
    new Rational(
      this.n * other.d + this.d * other.n,
      this.d * other.d
    )
}

val half = new Rational(1, 2)
val fifth = new Rational(1, 5)

val sum = half add fifth
// sum: Rational = R(7/10)
```

- Scala doesn't have operator overloading, per-se
- But it does have symbolic method names, (and operator precedence rules for first character)

  http://scala-lang.org/files/archive/spec/2.11/06-expressions.html#infix-operations

Infix and Symbolic Methods

```scala
class Rational(val n: Int, val d: Int) {
  require(d != 0, "Zero denominator!")

  override def toString: String = s"R($n/$d)"

  // symbolic rational addition
  def +(other: Rational): Rational =
    new Rational(
      this.n * other.d + this.d * other.n,
      this.d * other.d
    )
}

val half = new Rational(1, 2)
val fifth = new Rational(1, 5)

val sum = half + fifth
// sum: Rational = R(7/10)
```

- Change `add` to `+` and infix does the rest

Adding an Int to a Rational

- Strategy one: construct a `Rational` from an `Int`
- Can use an *auxiliary constructor* for this:

```
class Rational(val n: Int, val d: Int) {
  ...
  def this(i: Int) = this(i, 1)
  ...
}

val fifth = new Rational(1, 5)
val five = new Rational(5)

val sum = five + fifth
```

- Auxiliary constructors are quite limited, they can **only** call another constructor
- Better alternative is to use *factory methods*

Introducing: Companion Objects

- An object in the same source file with the same name as the class (or trait)
- Shares private state and behavior with the class (and vice versa)
- Scala's alternative to `static`
- Good place for a factory method (or two):

```scala
// in same source file as Rational class
object Rational {
  def apply(n: Int, d: Int): Rational =
    new Rational(n, d)

  def apply(i: Int): Rational =
    new Rational(i, 1)
}

val fifth = Rational(1, 5)  // can drop the new
val five = Rational(5)

val sum = five + fifth
```

Because It's a Companion

- It can access private behavior on the class
- We can make the constructor private and use the factory methods only:

```scala
class Rational private (val n: Int, val d: Int) {  // note position of private
  ...
}

object Rational {
  def apply(n: Int, d: Int): Rational =
    new Rational(n, d)  // companion can still call new

  def apply(i: Int): Rational =
    new Rational(i, 1)
}

val fifth = Rational(1, 5)  // R(1/5)
val five = Rational(5)      // R(5/1)
val half = new Rational(1, 2) // not allowed!
```

- Factory methods and private constructors are idiomatic

Adding an Int to a Rational

- Strategy two, overloading:

```
class Rational private (val n: Int, val d: Int) {
  require(d != 0, "Zero denominator!")

  override def toString: String = s"R($n/$d)"

  def +(other: Rational): Rational =
    new Rational(
      this.n * other.d + this.d * other.n,
      this.d * other.d
    )

  def +(i: Int): Rational =
    this + Rational(i)  // from companion
}

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

  def apply(i: Int): Rational =
    new Rational(i, 1)
}

Rational(1, 2) + 5  // R(11/2)
```

Adding a Rational to an Int

- Can do `half + 5` and `Rational(5) + half` but not `5 + half`... `implicit`!

```
class Rational private (val n: Int, val d: Int) {
  require(d != 0, "Zero denominator!")

  override def toString: String = s"R($n/$d)"

  def +(other: Rational): Rational =
    new Rational(
      this.n * other.d + this.d * other.n,
      this.d * other.d
    )
}

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

  implicit def apply(i: Int): Rational =
    new Rational(i, 1)
}

val half = Rational(1, 2)
half + 5
Rational(5) + half
5 + half
```

Implicits

- For implicit conversion, must `import` `language.implicitConversions` to avoid warning
- No longer need the overloaded + method for `Int`
- Implicits used by Scala to solve type problems
- Implicit conversion has single "in" type and single "out" type, e.g.

```
implicit def apply(i: Int): Rational = new Rational(i, 1)
```

- Companion objects **for types involved** in type problem are **one** of the places Scala looks for implicits
- Implicits in a companion object are hard to "un-invite"
- Must be marked with `implicit` keyword
- Name does not matter to Scala, only types matter
- Implicits can also be used for `val`, `object` and `class` (more on implicits later)

Exercises for Module 3

- Find the Module03 class and run it with ScalaTest
- This time it will be all green, but read the instructions and you will see that you need to uncomment the tests, write code, and make it compile and pass again
- As you get each section working, there may be a section following that also needs to be uncommented