Classes and Objects

Class Basics

```scala
class Rational1(n: Int, d: Int) {

  require(d != 0, "Denominator can't be negative")

  def numer: Int = n

  def denom: Int = d
}
```

- `n` and `d` are constructor parameters
- Think of the body of the class as the body of the primary constructor
  - The require is the first statement to execute in the constructor
- `n` and `d` are in scope in the bodies of methods `numer` and `denom` as local variables in the primary constructor.

Instance Basics

Given:

```
class Rational1(n: Int, d: Int) {
  require(d != 0, "Denominator can't be negative")
  def numer: Int = n
  def denom: Int = d
}
val r1 = new Rational1(1, 2)
```

`n` and `d` are not fields (instance variables), so this won't compile:

```
val r1 = new Rational1(1, 2)
```

`numer` and `denom` are methods, so this is the right way to access those values:

```
print(r1.numer + "/" + r1.denom)
```

`val` Fields and Overriding

```scala
class Rational2(n: Int, d: Int) {

  require(d != 0, "Denominator can't be neg")

  val numer: Int = n
  val denom: Int = d

  override def toString =
    s"$numer/$denom"
}

val r2 = new Rational2(3, 4)
```

- fields normally defined as vals
- `override` is keyword in Scala and required iff overriding

Self References

Like Java, using this keyword

```
class Rational3(n: Int, d: Int) {
  require(d != 0, "Denominator can't be negative")

  val numer: Int = n
  val denom: Int = d

  override def toString = s"$numer/$denom"

  def add(other: Rational3) =
    new Rational3(
      this.numer * other.denom + other.numer * this.denom,
      this.denom * other.denom
    )
}
```

Private Members

Default visibility is public. Here we compute the GCD with a private helper method:

```scala
class Rational4(n: Int, d: Int) {
  require(d != 0, "Denominator can't be negative")

  // Normalize fractions
  val numer: Int = n / gcd(n, d)
  val denom: Int = d / gcd(n, d)

  override def toString = s"$numer/$denom"

  def add(other: Rational4) =
    new Rational4(
      this.numer * other.denom + other.numer * this.denom,
      this.denom * other.denom
    )

  private def gcd(a: Int, b: Int): Int =
      if (b == 0) a else gcd(b, a % b)
}
```

Operators

In Scala, method names are quite flexible. In fact, operators are just methods on classes, like in this version of `Rational`:

```
class Rational5(n: Int, d: Int) {

  // ...

  def +(other: Rational5) =
    new Rational5(
      this.numer * other.denom + other.numer * this.denom,
      this.denom * other.denom
    )
}
```

Since single-paramter methods can be called using “operator” notation, we can do this:

```
val r5Half = new Rational5(1, 2)
val r5Quarter = new Rational5(1, 4)
val r5ThreeQuarters = r5Half + r5Quarter
```

Companion Objects

Scala doesn't have "static" members but use cases for static members can be done with a *companion object*, which:

- has the same name as its companion class
- must be defined in the same source file as its companion class
- has access to its companion class's private members (and vice-versa)

Companion objects are most often used for factory methods:

```
class Item(val description: String, val price: Double)

object Item {
  def apply(description: String, price: Double): Item =
    new Item(description, price)
}

val item = Item("Key Lime", 3.14) // Calls Item.apply
```

Exercise: add a companion object with a factory method to `Rationals`

Scala Applications

Singleton objects don’t have to be companion objects. A singleton object with a `main` method is a console application (similar to the `main` method in a Java application):

```scala
object Hello {
  def main(args: Array[String]) = {
    println("Hello, $args[0]")
  }
}
```

Scala’s library provides a shortcut trait called `App`:

```scala
object Hello extends App {
    println("Hello, $args[0]")
}
```