OOP in Scala

Scala Training, 2016

Object-oriented Scala

- Classes and traits
  - Fields keep state
  - Nothing `static`
  - Encapsulation
    - methods provide operations
    - access modifiers declare visibility

Object-oriented Scala

- Classes and traits
  - Fields keep state
  - Nothing `static`
  - Encapsulation
    - methods provide operations
    - access modifiers declare visibility
- Singleton objects are first-class

Object-oriented Scala

- Classes and traits
  - Fields keep state
  - Nothing `static`
  - Encapsulation
    - methods provide operations
    - access modifiers declare visibility
- Singleton objects are first-class
- Inheritance
  - Single inheritance (i.e. Java, extend exactly one superclass)

Object-oriented Scala

- Classes and traits
  - Fields keep state
  - Nothing `static`
  - Encapsulation
    - methods provide operations
    - access modifiers declare visibility
- Singleton objects are first-class
- Inheritance
  - Single inheritance (i.e. Java, extend exactly one superclass)
  - Mix-in as many traits as you want

Object-oriented Scala

- Classes and traits
  - Fields keep state
  - Nothing `static`
  - Encapsulation
    - methods provide operations
    - access modifiers declare visibility
- Singleton objects are first-class
- Inheritance
  - Single inheritance (i.e. Java, extend exactly one superclass)
  - Mix-in as many traits as you want
    - ... more on traits later

The simplest class

```
scala> class Rational
defined class Rational

scala> val rational = new Rational
rational: Rational = Rational@9d15d35

scala> rational.toString
res0: String = Rational@9d15d35
```

The almost simplest class

Java:

```
// Rational.java
public class Rational {
  public final int n;
  public final int d;
  public Rational(int n, int d) {
    this.n = n;
    this.d = d;
  }
}
```

Scala:

```
// Rational.scala
case class Rational(n: Int, d: Int)
```

Primary Constructors

- similar to Java's default no-args constructor
- spans the complete class definition

```
// class body = primary constructor
class Rational // no-arg constructor
{
  // implementation here

  // everything except for definitions
  // is run at construction time
  val x: Int = 0
  println(x)
}
```

Primary Constructors (2): class parameters

- class parameters are just constructor parameters
- they cannot be accessed from the outside

```
class A(x: Int)

val a = new A(2)
println(a.x)
```

```
error:  value x is not a member of A
  println(a.x)
            ^
```

Class parameters vs fields

Class parameters can be promoted to *fields*

- add `val` or `var` before them
- Q: `val` vs `var`?

```
class Rational(n: Int, d: Int) {
  def add(that: Rational) = {
    // error: value d is not a member of Rational
    new Rational(n * that.d, d * that.n, d * that.d)
  }
}

class Rational(val n: Int, val d: Int) {
  def add(that: Rational) = {
    // OK
    new Rational(n * that.d, d * that.n, d * that.d)
  }
}
```

Fields: mutable vs immutable

It's best to keep the API *immutable*

- no side effects $\rightarrow$ easier to reason about
- pure functions can be tested easily
- immutable objects means fewer concurrency issues

Rule of thumb

Make your variables immutable, unless there's a good reason not to.

Other constructors = auxiliary constructors

- use `def this` to define an auxiliary constructor
- invoke another constructor as its first action
- can't invoke a superclass constructor
  - only the primary may invoke a super-constructor

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

// ...
val r = new Rational(2)
```

Defining methods

- methods are class members
- last value computed = value returned by the method
  - Scala allows explicit `return`, but avoid using it
- method parameters are `vals`, they cannot be reassigned

```
def add(n: Int) = {
  val x = n + 1
  x // value of last expression = return value
}
```

```
def add(n: Int) = {
  n = n + 1 // error - reassignment
}
```

Hello sugar - infix notation

- methods with one parameter can be used in infix notation
- all operators are actually methods in infix notation
- any method can be an operator!

```
scala> "Hello World".split(" ")
res1: Array[String] = Array(Hello, World)

scala> "Hello World" split " "
res2: Array[String] = Array(Hello, World)
```

Hello sugar (2) - postfix notation

- methods without parameters can be used postfix
- use it when there are no side effects involved

```
scala> "Hello World" split " " size
warning: there were 1 feature warning(s); re-run with
    -feature for details
res0: Int = 2
```

Hello sugar (3) - prefix notation

Methods may be used in prefix position if

- names start with `unary_` followed by +, -, ! or

```
scala> !true
res0: Boolean = false

// exact same thing
scala> true.unary_!
res1: Boolean = false
```

Default arguments

```
def name(first: String = "", last: String = "Doe") =
 first + " " + last

scala> name("John")
res0: String = John Doe
```

- lets you omit *trailing* arguments
  - reduce boilerplate
- how do you omit *leading* arguments?

Default arguments(2)

How to omit leading arguments:

- name them when you call the method
- all arguments may be named, omitted or mixed

```
scala> name(last = "Popescu")
res0: String = Popescu

scala> name(first = "John", last = "Doe")
res0: String = John Doe
```

Other constructors

- all members are *public* by default
- use *private protected* to restrict access
  - same meaning as in Java
- use a qualifier to relax access up to a given entity
- use `this` to restrict the access to the instance

```
class Hello {
  private[this] val message = "Hello!"
  def messagesEqual(that: Hello) =
    this.message == that.message // Won't compile!
  }
```

Singleton objects

- Scala is more object-oriented than Java
  - Scala classes cannot have static members
- singletons are first-class citizens
- singletons replace static, e.g. holding constants
- use the `object` keyword to define a singleton object

```
object Rational {
  val denominator = 1
}

scala> Rational.denominator
res0: Int = 1
```

Singleton objects (2) - companions

Companion = a Scala object with the same *name, package and file* with a class or trait

- companions can access private members
- otherwise, there is no relation at all

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

// a companion object
object Rational {
  val denominator = 1
}
```

Scala applications

Q: entry point in a Java application?

Scala applications

Q: entry point in a Java application?

In Scala, an entry point has to

- be a *standalone* (= non-companion) object
- have a *main method*

```
object MyApp {
  // notice the signature
  def main(args: Array[String]) = {
    println("Hello world")
  }
}
```

Scala applications

Q: entry point in a Java application?

In Scala, an entry point has to

- be a *standalone* (= non-companion) object
- have a *main method*

```
object MyApp {
  // notice the signature
  def main(args: Array[String]) = {
    println("Hello world")
  }
}
```

Alternative: extend *App* and just write code:

```
object MyMain extends App {
  println("hello")
}
```

Q: why do you think the above code works?

Case classes

```
case class Rational(n: Int, d: Int = 1)

scala> val one = Rational(1)
one: Rational = Rational(1,1)
```

Features:

- instantiation without *new*
- nice implementations of *toString*, *equals*, *hashCode*
- class params are promoted to *immutable fields* automatically
- copy method is automatically implemented
- *pattern matching* (covered later)

Digression: what a function *actually is*

Explanation by example:

```
object MyFunction {
  def apply(x: Int): Int = x + 1
}

MyFunction(2) // 3
```

Digression: what a function *actually is*

Explanation by example:

```
object MyFunction {
  def apply(x: Int): Int = x + 1
}

MyFunction(2) // 3
```

Magic?

Digression: what a function *actually is*

Explanation by example:

```
object MyFunction {
  def apply(x: Int): Int = x + 1
}

MyFunction(2) // 3
```

Magic?

In Scala, all functions are instances of a `FunctionX` type:

```
// what you think you say
val lessThan = (a: Int, b: Int) => a < b

// what you actually say
val lessThan = new Function2[Int, Int, Boolean] {
  def apply(a: Int, b: Int) = a < b
}
```

Back: Case classes (2)

- all case classes have a companion with an *apply* method
- *apply* is a factory method

Back: Case classes (2)

- all case classes have a companion with an *apply* method
- *apply* is a factory method
  - **Q:** what's a factory method?

```
scala> Rational(1)
res0: Rational = Rational(1,1)

scala> Rational.apply(1)
res1: Rational = Rational(1,1)
```

Case classes (3)

Other features - operators, accessors and copy

```
scala> one == Rational(1)
res1: Boolean = true

scala> one == Rational(1, 2)
res2: Boolean = false

scala> one.nominator
res3: Int = 1

scala> one.copy(denominator = 2)
res4: Rational = Rational(1,2)
```

Imports

- use _ to import all members of a package / object
- import multiple members with {}
- rename imported objects
- scope imports from method

```
// import multiple items
import scala.concurrent.{Future, Promise}
// import all from package
import spray.json._
// import under a different name (alias)
import java.sql.{Date => SqlDate}
```

Recap

- classes & singletons
- companions
- class parameters & fields
- constructors
- methods
- infix & postfix notation
- default arguments
- case classes
- *what a function actually is*