CS 3101-2 - Programming Languages: Scala

Lecture 5: Exceptions, Generic Classes

Daniel Bauer (bauer@cs.columbia.edu)

November 19, 2014

Throwing Exceptions

- Exceptions work similar to Java
  `throw{...} catch{...} finally{...}`
- `throw` is an expression (but the value of the expression can never be used)

```
val half =
    if (n % 2 == 0)
        n / 2
    else
        throw new RuntimeException("n must be even")
}
```

- Exceptions are used less often than in Java or Python.

Catching Exceptions

- Exceptions are passed up the call hierarchy, until they reach a `catch` clause.

```
import scala.io.Source
import java.io.{FileNotFoundException, IOException}

val filename = "input.txt"
try {
    val input = Source.fromFile(filename)
    for (line <- input.getLines()) {
        println(line)
    }
} catch {
    case ex: FileNotFoundException =>
        println("File not found.")
    case ex: IOException =>
        println("Cannot read from file.")
}
```

finally clause

- A `finally` clause will be executed whether an exception occurs or not.

```
import scala.io.Source
import java.io.{FileNotFoundException, IOException}

val filename = "input.txt"

val input = Source.fromFile(filename)
try {
    for (line <- input.getLines()) {
        println(line)
    }
} catch {
    case ex: FileNotFoundException =>
        println("File not found.")
    case ex: IOException =>
        println("Cannot read from file.")
} finally {
    input.close()
}
```

The 'Loan' pattern

- Write a higher-order function that 'borrows' a resource and makes sure it is returned.

```
def withFileSource(filename: String)(op: Source => Unit) {
    val filesource = Source.fromFile(filename)
    try {
      op(filesource)
    } finally {
      filesource.close()
    }
}

withFileSource("input.txt") {
    input => {
        for (line <- input.getLines())
            println(line)
    }
}
```

Traits vs. Inheritance

- Inheritance means adding to the implementation of a single parent class (or overriding).
- Scala does not support multiple inheritance (unlike e.g. Python), but offers traits.
- Traits are a *'fundamental unit of code reuse'*.
  - Defines methods and attributes that can be re-used by various classes.
  - Classes can mix in any number of traits.
- Similar to Java interfaces.
- No parameters.

```
trait Philosophical {
    def philosophize() {
        println("I consume memory, therefore I am!")
    }
}
```

Defining and Using Traits

```
trait Philosophical {
  def philosophize() {
    println("I consume memory, therefore I am!")
  }
}
trait HasLegs { val legs : Int = 4 }

class Animal

class Frog extends Animal with Philosophical with HasLegs{
  override def toString = "green"
}
scala> val frog = new Frog
frog: Frog = green

scala> frog.philosophize
I consume memory, therefore I am!

scala> frog.legs
res0: Int = 4
```

Using Traits II

- A single Trait can be mixed in using `extends`.

```
trait Philosophical {
    def philosophize() {
        println("I consume memory, therefore I am!")
    }
}
// mix in Philosophical
class Philosopher extends Philosophical
```

```
scala> class Philosopher extends Philosophical
defined class Philosopher

scala> val p = new Philosopher
p: Philosopher = Philosopher@2dc4de05

scala> p.philosophize
I consume memory, therefore I am!
```

Traits are Types

```
trait Philosophical {
    def philosophize() {
        println("I consume memory, therefore I am!")
    }
}
class Animal
class Frog extends Animal with Philosophical {
    val color = "green"
}
```

```
scala> val phil : Philosophical = new Frog() // trait as type
f: Philosophical = Frog@16a15a6e

scala> phil.philosophize
I consume memory, therefore I am!
```

Traits are Types

```
trait Philosophical {
    def philosophize() {
        println("I consume memory, therefore I am!")
    }
}
class Animal
class Frog extends Animal with Philosophical {
    val color = "green"
}
```

```
scala> val phil : Philosophical = new Frog() // trait as type
f: Philosophical = Frog@16a15a6e

scala> phil.philosophize
I consume memory, therefore I am!

scala> phil.color // not accessible because defined on Frog
<console>:12: error: value color is not a member of
            Philosophical
```

Polymorphism with Traits

```
trait Philosophical {
    def philosophize() {
        println("I consume memory, therefore I am!")
    }
}

class Animal
class Frog extends Animal with Philosophical {
    override def toString = "green"
    override def philosophize() {
        println("It ain't easy being " + toString + "!")
    }
}
```

```
scala> val phrog : Philosophical = new Frog()
phrog: Philosophical = green

scala> phrog.philosophize
It ain't easy being green!
```

Thin vs. Rich Interfaces to Classes

Thin Interfaces:

- Minimal functionality, few methods.
- Easy for the developer of the interface.
- Larger burden on client using the class (needs to fill in the gaps or adapt general methods).

Rich Interfaces:

- Many specialized methods.
- Larger burden when implementing the class.
- Convenient for the client.

- Traits can be used to enrich thin interfaces, re-using existing methods.

Thin vs. Rich Interfaces - Example: Rectangular Objects

```
class Point(val x: Int, val y: Int)

class Rectangle(val topLeft: Point, val bottomRight: Point) {
    def left = topLeft.x
    def right = bottomRight.x
    def width = right - left
    // and many more geometric methods...
}
```

- Another class outside of the same type hierarchy with similar functionality:

```
abstract class Widget {
    def topLeft : Point
    def bottomRight : Point

    def left = topLeft.x
    def right = bottomRight.x
    def width = right - left
    // and many more geometric methods...
}
```

Thin vs. Rich Interfaces - Example: Rectangular Objects

```
def Rectangular {
    def topLeft : Point
    def bottomRight : Point

    def left = topLeft.x
    def right = bottomRight.x
    def width = right - left
    // and many more geometric methods...
}

 abstract class Widget extends Rectangular {
    // other methods...
  }

class Rectangle(val topLeft: Point,
               val bottomRight: Point) extends Rectangular {
        // other methods...
 }
```

Modifying Methods with Traits

```
import scala.collection.mutable.ArrayBuffer

abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}

class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}
```

Modifying Methods with Traits

```
import scala.collection.mutable.ArrayBuffer

abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}

class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}

scala> val queue = new BasicIntQueue
queue: BasicIntQueue = BasicIntQueue@24655f

scala> queue.put(10)
scala> queue.put(20)
```

Modifying Methods with Traits

```
import scala.collection.mutable.ArrayBuffer

abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}

class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}

scala> val queue = new BasicIntQueue
queue: BasicIntQueue = BasicIntQueue@24655f

scala> queue.put(10)
scala> queue.put(20)

scala> queue.get()
res0: Int = 10

scala> queue.get()
res1: Int = 20
```

Modifying Methods with Traits

- Traits can modify (override) methods of a base class.
- Add some functionality but then call method of the super class.

```
trait Incrementing extends IntQueue {
    abstract override def put(x: Int) { super.put(x + 1) }
}

scala> class MyQueue extends BasicIntQueue with Incrementing
defined class MyQueue
```

Modifying Methods with Traits

- Traits can modify (override) methods of a base class.
- Add some functionality but then call method of the super class.

```
trait Incrementing extends IntQueue {
    abstract override def put(x: Int) { super.put(x + 1) }
}

scala> class MyQueue extends BasicIntQueue with Incrementing
defined class MyQueue

scala> val queue = new MyQueue
scala> val queue = new BasicIntQueue with Incrementing
queue: BasicIntQueue with Incrementing = $anon$1@5fa12d
```

Modifying Methods with Traits

- Traits can modify (override) methods of a base class.
- Add some functionality but then call method of the super class.

```
trait Incrementing extends IntQueue {
    abstract override def put(x: Int) { super.put(x + 1) }
}

scala> class MyQueue extends BasicIntQueue with Incrementing
defined class MyQueue

scala> val queue = new MyQueue
scala> val queue = new BasicIntQueue with Incrementing
queue: BasicIntQueue with Incrementing = $anon$1@5fa12d

scala> queue.put(10)
scala> queue.get()
res: Int = 21
```

Modifying Methods with Traits

- Multiple traits can be mixed in to stack functionality.
- Methods on `super` are called according to linear order of `with` clauses (right to left).

```
trait Incrementing extends IntQueue {
    abstract override def put(x: Int) { super.put(x + 1) }
}

trait Filtering extends IntQueue {
    abstract override def put(x: Int) {
        if (x >= 0) super.put(x)
    }
}
```

Modifying Methods with Traits

- Multiple traits can be mixed in to stack functionality.
- Methods on `super` are called according to linear order of `with` clauses (right to left).

```
trait Incrementing extends IntQueue {
    abstract override def put(x: Int) { super.put(x + 1) }
}

trait Filtering extends IntQueue {
    abstract override def put(x: Int) {
        if (x >= 0) super.put(x)
    }
}

scala> val queue = new (BasicIntQueue
                        with Incrementing
                        with  Filtering)
queue: BasicIntQueue with Incrementing with Filtering...
```

Modifying Methods with Traits

- Multiple traits can be mixed in to stack functionality.
- Methods on super are called according to linear order of `with` clauses (right to left).

```
trait Incrementing extends IntQueue {
    abstract override def put(x: Int) { super.put(x + 1) }
}

trait Filtering extends IntQueue {
    abstract override def put(x: Int) {
        if (x >= 0) super.put(x)
    }
}

scala> val queue = new (BasicIntQueue
                        with Incrementing
                        with  Filtering)
queue: BasicIntQueue with Incrementing with Filtering...

scala> queue.put(-1); queue.put(0);
scala> queue.get()
res: Int = 1
```

Traits or Abstract Classes

Both traits and abstracts classes can have abstract and concrete members.

Traits:

- No constructor paramters or type parameters.
- Multiple traits can be mixed into class definitions.
- Semantics of `super` depends on order of mixins. Can call abstract methods.

Abstract Classes:

- Have constructor parameters and type parameters.
- Work better when mixing Scala with Java.
- `super` refers to unique parent. Can only call concrete methods.

Parametric Types

- Typically want to specify type of elements of a collection.
- Using generic classes.

```
scala> val x : List[Int] = 1 :: 2 :: 3 :: Nil
x: List[Int] = List(1, 2, 3)

scala> val y : List[Int] = 1 :: 2 :: "Hello" :: Nil
<console>:7: error: type mismatch;
 found    : List[Any]
 required: List[Int]
       val y : List[Int] = 1 :: 2 :: "Hello" :: Nil
```

Parametric Types

- Typically want to specify type of elements of a collection.
- Using generic classes.

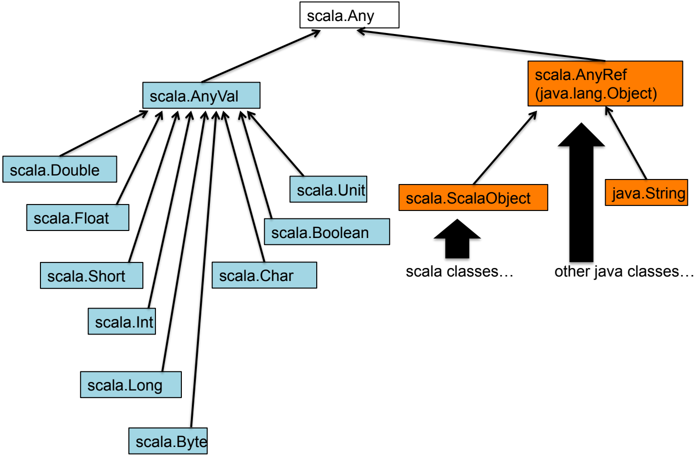
```
scala> val x : List[Int] = 1 :: 2 :: 3 :: Nil
x: List[Int] = List(1, 2, 3)

scala> val y : List[Int] = 1 :: 2 :: "Hello" :: Nil
<console>:7: error: type mismatch;
 found    : List[Any]
 required: List[Int]
       val y : List[Int] = 1 :: 2 :: "Hello" :: Nil

scala> val x = 1 :: 2 :: "Hello" :: Nil
x: List[Any] = List(1, 2, Hello)

scala> x(2)  // Don't know specific type of this element
res0: Any = Hello
```

Scala's Type Hierarchy



Type Parameters for Methods

- Methods can also have type parameters (for return value and parameters).

```
  def dup[T](x: T, n: Int): List[T] =
    if (n == 0)
      Nil
    else
      x :: dup(x, n - 1)

println(dup[Int](3, 4))
println(dup("three", 3))
```

By Defaul, Classes are Invariant

- Type parameters generate a family of types.
- Is `GenericClass[A]` a subtype of `GenericClass[B]` if `A` is a subtype of `b`?

```
scala> class Container[A](val content: A)
defined class Container

scala> val x = new Container("test")
x: Container[String] = Container@256f8274

scala> val y : Container[AnyRef] = x
<console>:9: error: type mismatch;
 found   : Container[String]
 required: Container[AnyRef]
Note: String <: AnyRef, but class Container is invariant in
       type A.
       You may wish to define A as +A instead. (SLS 4.5)
       val y : Container[AnyRef] = x
```

Covariance Annotations

- Prefixing a type parameter with + makes the class covariant in this parameter.

```
scala> class Container[+A](val content: A)
defined class Container
scala> val x = new Container("text")
x: Container[String] = Container@14f5da2c
scala> val y : Container[AnyRef] = x
y: Container[AnyRef] = Container@14f5da2c
```

`Container[String]` is now a subclass of any `Container[A]` if `String` is subtype of `A`.

Covariance can be tricky

- Prefixing a type parameter with + makes the class covariant in this parameter.

```
scala> class Container[+A](var content: A) // make it a var
```

Covariance can be tricky

- Prefixing a type parameter with + makes the class covariant in this parameter.

```
scala> class Container[+A](var content: A) // make it a var

<console>:9: error: covariant type A occurs in contravariant pc
       class Container[+A](var content: A)
```

What's wrong with this class definition?

Covariance can be tricky II

- Prefixing a type parameter with + makes the class covariant in this parameter.

```
scala> class Container[+A](val content: A) {
          def printExternal(x : A) {
              println(x)
          }
      }
```

Covariance can be tricky II

- Prefixing a type parameter with + makes the class covariant in this parameter.

```
scala> class Container[+A](val content: A) {
           def printExternal(x : A) {
               println(x)
           }
       }

<console>:10: error: covariant type A occurs in contravariant p
                 def printExternal(x : A) {
```

What's wrong with this class definition?

Covariance can be tricky II

- Prefixing a type parameter with + makes the class covariant in this parameter.

```
scala> class Container[+A](val content: A) {
            def printExternal(x : A) {
                println(x)
            }
       }

<console>:10: error: covariant type A occurs in contravariant p
                 def printExternal(x : A) {
```

What's wrong with this class definition?
General rules:

- Cannot use covariance if type parameter is used for a mutable field.
- Cannot use covariance if type paramater is used for a method paramter.

Lower Bounds

- Lower bounds can be used when defining methods of covariant classes.
- Reveal at least some information about the parameters.

```
scala> class Container[+A](val content: A) {
            def printExternal[B >: A](x : B)  {
                println(x)
            }
            def makeTuple[B >: A](other : B): (B, B) =
                (content, other)
     }

scala> val x = new Container("hi")
x: Container[String] = Container@6d2a209c

scala> x.makeTuple(3)
res1: (Any, Any) = (hi,3)
```