Topic 13: Packages and Imports

Presentation

Jesper Pedersen Notander

Department of Computer Science
Lund University Faculty of Engineering

Learning Scala
Seminar 3

Contents

- Packages - How to achieve scalability by modularizing your code
- Imports - The way to prevent name space cluttering
- Access Modifiers - The visibility rules of your declarations
- Package objects - A place for utility functions, types and implicit conversions

Packages

- Why? - Modularizing, Code Encapsulation, Maintainability
- How? - Java style, C# style, Scala style
- Why again? - Simple access, Encapsulate together

```
package companies.ecocar
// rest of the file
```

```
package companies.ecocar {
// your code
}
```

```
package companies {
  // top level code
  package ecocar {
     // your code
}}
```

```
package company
package ecocar
// your code
```

Example: Scoping Rules

```
package vehicles {
  class AnyCar(val name: String)
}
package company.dummycars {
  class DummyCars extends Company // not ok, qualified access needed
}
package company {
  class Company

  package vehicles {
    class CompanyCar
  }

  package ecocars {
    package vehicles {
      class EcoCar
    }
    class EcoCars extends Company { // ok, normal nested scoping rules
      val car1 = new vehicles.EcorCar()
      val car2 = new company.vehicles.CompanyCar()
      val car3 = new _root_.vehicles.AnyCar() // special _root_ package access
    }
  }
}
```

Imports

- Why? - Simple name access of packages and members
- How? - As Java but _ instead of *

```
import company.Company   // the  company class
import company._         // all package members
import company.Company._ // all class members
```

- Does the order matter?
- How again? - Everywhere, Packages, Values

```
import vehicles
def drive(car: vehicles.Car) {
  import car._
  println("The car " + name + " is driving")
}
```

The Import Selector Clause

- Explicit member import

```
import Cars.{EcoCar, DummyCar}
```

- Import renaming

```
import Cars.{EcoCar=>TheBestCar, DummyCar}
import vehicles.cars.{Cars => Automobiles}
import vehicles.{cars => c}
```

- Catch all and member hiding

```
import vehicles.cars.Cars.{_}
import Cars.{DummyCar=>_, _}
```

Access Modifiers

- Why? - Protect Data, Hide code
- How? - Like Java but more
  - Three levels - `private, protected` and public
  - With qualifiers - `private[x] protected[x]`, where x can be package, class or `this`
- What about companion objects?
  - Shared access rules, supposedly
  - No protected members

Example: Scope of Protection

```
package vehicles
abstract class Car() {
  class State[T](private var value: T)
  val driveState = new State(false)
  def start(): Unit = driveState.value = true // not ok

  protected def debug(msg: String) = println(msg)

  private val speed: Int = 0
  def relativeSpeed(other: Car) = speed - other.speed // ok
}

class EcoCar extends Car {
  debug("EcoCar created")
}

class Debugger(car: Car) {
  car.debug("Debugger registered") // not ok
}
```

Example: Qualifiers

```
package vehicles
abstract class Car() {
  class State[T](private[Car] var value: T)
  val driveState = new State(false)
  def start(): Unit = driveState.value = true // not ok

  protected[vehicles] def debug(msg: String) = println(msg)

  private[this] val speed: Int = 0
  def relativeSpeed(other: Car) = speed - other.speed // ok
}

class EcoCar extends Car {
  debug("EcoCar created")
}

class Debugger(car: Car) {
  car.debug("Debugger registered") // not ok
}
```

Package Objects

- Why? - A place for helper functions, type aliases and implicits
- How? - Each package can have one object
  - Similar to a package declaration

```
package object vehicles {
  def getUniqueSerialNumber(): Int {
    // ...
  }
}
```

  - Place source in package directory
  - Naming convention, package.scala

Summary

- Organize your code with packages.
  - Nested definitions, multiple definitions.
- Import code with `import`.
  - Rename or hide ambiguous declarations.
- Restrict access with `private` and `protected`.
  - Modify the access scope with qualifiers `modifier[qualifier]`.
- Put common definitions in package objects.

Chapter 14

Assertions and Unit Testing

CS Scala course 2012

Overview: Chapter 14

- Assertions
- Unit testing
  - ScalaTest
  - JUnit
- Behavior-Driven Development
- Property-based testing

+ Exercise

Assertions

Assertions

- assertion = predicate placed in a program to indicate that the developer *thinks* that the predicate is always true at that place
- Defensive programming
  - Pre-conditions or post-conditions
- Typically enabled in debug builds
- Fundamental element in test code
- Safety-critical development
  - odds of intercepting defects increase with assertion density

assert in Scala

- Toggle assertions using JVM flags
  - ea
  - da

- assert method defined in the Predef object
- throws AssertionError

assert (condition)

assert (condition, explanation)

ensuring

- asserts the result of a function
- ensuring method also in the Predef object
- can be used with any result type thanks to an implicit conversion
- if false: throws AssertionError

ensuring (condition)

ensuring (condition, explanation)

Example

```
def getAttendance(ScalaSeminar: Int): Int = {
      // calculate attendance at specific seminar
      assert(nbrStudents > 0)
      nbrStudents
}



def countNewStudents (): Int = {
      val attendanceSem2 = getAttendance(2)
      val attendanceSem4 = getAttendance(4)
      // calculate difference
      nbrNewStudents
} ensuring (_ >= 0, "We have dropouts!")
```

Unit testing

ScalaTest™

simply productive™

- Create classes that extend org.scalatest.Suite
- Prefix ”test”

```
import org.scalatest.Suite
class ExampleSuite extends Suite {

      def testAddition {
             val sum = 1 + 1
             assert(sum == 2)
      }

      def testSubtraction {
            val diff = 4 - 1
            assert(diff == 3)
      }
}
```

ScalaTest™
simply productive™

- Extend trait org.scalatest.FunSuite to define tests as functions values
- No prefix, call test, specifiy name

```
import org.scalatest.Suite
class ExampleSuite extends FunSuite {

         test("integer addition") {
                   val sum = 1 + 1
                   assert(sum == 2)
         }

         test("integer subtraction") {
                  val diff = 4 - 1
                  assert(diff == 3)
         }
}
```

ScalaTest™
simply productive™

- **===** can be used in the assert method to produce more information
- **expect** method to verify results
- **intercept** method to check exceptions

```
expect(5) {
      nbrStudents
}

Intercept[NoStudentsException] {
      checkStudentAttendance(0)
}
```

- Write JUnit tests in Scala
  - Compile and run them in JUnit
- Reuse your JUnit tests in ScalaTest
  - Use the JUnitWrapperSuite

BDD

- Behavior-Driven Development
  - Based on Test-Driven Development
- Readable specifications of code behavior
- Accompanied by tests that verify the code
- Extend one of the **Spec** traits

```
class ExampleSpec extends FunSpec {
  describe("A Stack") {
    it("should pop values in last-in-first-out order") {
      val stack = new Stack[Int]
      stack.push(1)
      stack.push(2)
      assert(stack.pop() === 2)
      assert(stack.pop() === 1)
    }
}
```

Property-based testing

- *A property* is a high-level specification of a behavior that should hold no matter what
- Traditional tests verify behavior based on specific data points
- ScalaCheck library generates test data looking for values for which the property does not hold
- https://github.com/rickynils/scalacheck

Summary

- For defensive Scala programming
  - Put assertions in the production code
- Use ScalaTest for unit testing
- Reuse JUnit tests using wrappers
- Scala supports recent test research

Topic 15
Case classes and Pattern Matching

Alfred Theorin

Automatic Control
LTH

Case Classes is Syntactic Sugar

```
abstract class Expr
case class Var(name: String) extends Expr
case class Num(num: Double) extends Expr
case class UnOp(op: String, expr: Expr) extends Expr
case class BinOp(op: String, l: Expr, r:Expr) extends Expr
```

For Case Classes You Get ...

```
// a factory method to construct instances
val x = Var("x")
val op = BinOp("+", Var("x"), Num(1))
val z = new BinOp("+", new Var("x"), new Num(1))

// Implicitly val on constructor parameters
scala> x.name
res0: String = x

// "natual" toString, hashCode, and equals
scala> op == z
res1: Boolean = true

// a copy method
scala> op.copy(l = Num(41))
res2: BinOp = BinOp("+", Num(41.0), Num(1.0))
```

What `match` does that `switch` doesn't

- Uses pattern matching
- Returns a value
- No fallthrough
- Throws `MatchError` on failure

Some ways to use `match`

```
def simplify(expr: Expr): Expr = expr match {
  // Deep match, variable pattern, constant pattern
  case BinOp("+", Num(0), e) => e
  // Guard
  case BinOp("+", x, y) if x == y => BinOp("*", x, Num(2))
  // Pattern overlap
  case BinOp(op, l, r) =>
    BinOp(op, simplify(l), simplify(r))
  // Variable binding
  case UnOp("abs", e @ UnOp("abs", _)) => e
  case _ => expr // Wildcard
}
```

First Letter Case Determines If Variable Patterns

```
val pi = math.Pi
val Pi = math.Pi
def isPi(x: Double): Boolean = {
  x match {
    case Pi => true
    case pi => true // Match! First letter is lower case
    case this.pi => true // Comparing to class member pi
    case `pi` => true // Comparing to variable pi
    case _ => false
  }
}

scala> isPi(math.E)
res0: Boolean = true
```

Pattern Matching for Sequences

```
def describe(x: List[Int]) { println(x match {
  case List(_, _, _) => "length = 3"
  case List(_, _*) => "length >= 1"
  case _ => "empty"
})}

scala> describe(List(1,2,3))
length = 3

scala> describe(List(0,1))
length >= 1

scala> describe(List())
empty
```

Pattern Matching for Tuples

```
def describe(x: Any) {
  println(x match {
    case (a, b) => "A tuple with two elements"
    case _ => "Something else"
  })
}

scala> describe((1,2))
A tuple with two elements

scala> describe((1,2,3))
Something else
```

Pattern Matching Instead of Type Checking

```
expr match {
  case op: BinOp => println(op.l)
  case _ =>
}
```

Pattern Matching Instead of `null` Checks

```
val map = Map("keyValid" -> "value")

def show(x: Option[String]): String = x match {
  case Some(s) => s
  case None => "not in map"
}

scala> show(map get "keyValid")
value

scala> show(map get "keyInvalid")
not in map
```

Compiler Feedback For Pattern Matching

- Errors for unreachable cases
- Warning if not exhaustive for `sealed` classes:
  `sealed abstract class Expr`

Pattern Matching Without `match`

```
scala> val (a, b) = (10, "asdf")
a: Int = 10
b: String = asdf

scala> val BinOp(oper, l, r) = op
oper: String = "+"
...

scala> for((key, value) <- map) ...

scala> for(Some(x) <- xs) ...
```

Partial Functions

```
scala> val second: List[Int] => Int = {
  case x :: y :: _ => y }
warning: match is not exhaustive!

val second: PartialFunction[List[Int],Int] = {
  case x :: y :: _ => y }
scala> second.isDefinedAt(List(1,2,3))
res0: Boolean = true

scala> second.isDefinedAt(List(1))
res1: Boolean = false
```

Lists in Scala

- Scala Lists are quite similar to arrays with two important differences
  - Immutable: elements of a list can not be changed by assignment
  - Recursive structure unlike arrays that are flat

- A list of Strings:

```
val str: List[String] = List(“a”, “b”,“c”)
```

- A list of Integers:

```
val nums: List[Int] = List(1, 2, 3, 4)
```

- An empty List:

```
val empty: List[Nothing] = List()
```

- Scala lists are *homogenous* like arrays

  val nums: List[Int] = List(1, 2, 3, 4)

- List type in Scala is *covariant*
  - If S is a subtype of T, then List[S] is a subtype of List[T].
  - List[String] is a subtype of List[Object]
  - List[Nothing] is a subtype of all lists in Scala

Constructing lists

- All lists are defined using two fundamental building blocks
  - **Nil** (Nil represents the empty list)
  - **::** (pronounced as **cons)**
- :: operator expresses list extension at the front

  X :: XS

  X is a first element of list XS

Contd..

- List of strings using :: operator

  val str = “a” :: (“b” :: (“c” :: Nil))

- List of Integers using :: operator

  val nums = 1 :: (2 :: (3 :: (4 :: Nil)))

Methods of class List

- head: it returns the first element of a list
- tail: it returns a list consisting of all elements except the first
- isEmpty: it returns true if the list is empty otherwise false.

  - val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))

```
println( "Head of fruit : " + fruit.head )
println( "Tail of fruit : " + fruit.tail )
println( "Check if fruit is empty : " +
                                  fruit.isEmpty )
```

Contd..

- concat: it returns a list after combining two lists, for this ::: operator is also used
- init: returns a list containing all elements except the last one (Opposite of tail method)
- last: returns the last element of a list
- reverse: returns a list with elements on reverse order

- take n: returns the first n elements of a list
- drop n: returns all elements of a list except the n ones
- splitAt: it splits the list at a given index and returns pair of two lists

- map: it takes a list XS and a function f as operands and returns a list resulting from applying the function f

Example:

```
val nums = 1 :: 2 :: 3 :: 4 :: Nil
println(nums.map(_+1))
```

Result:

```
List(2, 3, 4, 5)
```

- filter: it takes a list XS and a predicate function p as operands. It yields the list of all elements x in XS for which p(x) is true

Example:

```
val nums: List[Int] = List(1, 2, 3, 4)
println(nums.filter(_>2))
```

Result:

```
List(3, 4)
```

- find: it works like filter method but it returns the first element satisfying given predicate rather than all such elements

Example:

```
val nums: List[Int] = List(1, 2, 3, 4)
println(nums.find(_>2))
```

Result:

```
Some(3)
```

- range: creates a list consisting of a range of numbers

  List.range(from, until)

  It returns list of all elements starting at from and going up to until minus one

Example:

```
val nums: List[Int] = List(1, 33, 7, 56,67)
println(nums.range(1,4))
```

Result:

```
List(1, 33, 7)
```

- fill: it creates a list consisting of zero or more copies of the same element

Example:

```
println(List.fill (4) ('a') )
```

Result:

```
List(a, a, a, a)
```

Collections - overview

1. Sequences
2. Buffer classes
3. StringOps
4. Sets
5. Maps
6. Immutable vs mutable
7. Sorted sets and maps
8. Conversions
9. Tuples

Sequences (1/2)

List – linked list, fast addition and removal in the beginning of the list, accessing arbitrary indexes takes linear time. Built it backwards and call reverse. See method table on page 44.

Example

```
scala> var fruits = List("apple", "orange", "pear")
fruits: List[java.lang.String] = List(apple, orange, pear)

scala> fruits.reverse
res0: List[java.lang.String] = List(pear, orange, apple)
```

Sequences (2/2)

Array – Zero based, accessed with (). Has all usual java array methods.

Example

```
scala> val fiveInts = new Array[Int](5)
fiveInts: Array[Int] = Array(0, 0, 0, 0, 0)

scala> fiveInts(1) = 5
res0:Array[Int] = Array(5, 0, 0, 0, 0)
```

Buffer classes (1/2)

List buffer – Scala.collection.mutable.ListBuffer.

Constant time appending (+=) and prepending (+=:) .

Example

```
import scala.collection.mutable.ListBuffer
Val lBuf = new ListBuffer[Int]
lBuf += 2 // res1: lBuf.type = ListBuffer(2)
lBuf.toList // res2: List[Int] = List(2)
```

Buffer classes(2/2)

Array buffer –
Scala.collection.mutable.ArrayBuffer.

Append and prepend constant time on average.

Example

```
import scala.collection.mutable.ArrayBuffer
Val aBuf = new ArrayBuffer[Int]
aBuf += 2 // res2: aBuf.type =
ArrayBuffer(2)
aBuf.toArray // res3: Array[Int] = Array(2)
```

StringOps

Strings are implicitly converted to sequences using StringOps.

Example

```
def hasUpperCase(s: String) =
s.exists(_.isUpper)

scala> hasUpperCase("camelCase")
res0: Boolean = true
```

Sets (1/2)

Immutable (default) and mutable versions. See table 17.1 on common operations.

Example

```
val iSet = Set(1, 3, 2)
iSet + 5 // Set(1, 3, 2, 5)
iSet ++ List(5, 6) // Set(1, 3, 2, 5, 6)

scala> iSet & Set(2) // intersection
res8: scala.collection.immutable.Set[Int] =
Set(2)
```

Sets (2/2)

Example

```
import scala.collection.mutable.Set, 6)

scala> val mSet = Set.empty[String]
mSet: scala.collection.mutable.Set[String] =
Set()

scala> mSet += "Hi"
res9: mSet.type = Set(Hi)

iSet += 2 // ☹
// Declared as var += interpreted as iSet = iSet
+ 2
```

Maps

Immutable (default) and mutable versions. See table 17.2 on common operations.

Example

```
val iMap =  val iMap = Map("France" -> "Paris",
"Germany" -> "Berlin")
iMap + (“Norway” -> “Oslo”)
// returns a new map.

scala> iMap(”France”)
res14: java.lang.String = Paris

// Use var to use += on immutables.
```

Immutable vs mutable sets and maps

Immutable sets and maps with less than 5 elements are own classes.

# of elements	**Implementation** (package immutable)
0	EmptySet/EmptyMap
1	Set1/Map1
2	Set2/Map2
3	Set3/Map3
4	Set4/Map4
5 or more	HashSet/HashMap

When in doubt, go for immutable.

Sorted sets and maps

Immutable TreeSet and TreeMap. The elements must mix in the `Ordered` trait.

Example

```
import scala.collection.immutable.TreeSet
val tSet = TreeSet(2, 5, 8, 3)
// TreeSet(2, 3, 5, 8)

import scala.collection.immutable.TreeMap
val tMap = TreeMap(2 -> "two", 1 -> "one",
0 -> "zero")
// Map(0 -> zero, 1 -> one, 2 -> two)
```

Conversions

To List or Array - call toList or toArray on Set or Map.

Between mutable and immutable – add the elements to new empty instance.

Example

```
val mutaSet =
scala.collection.mutable.Set.empty[Int] ++=
tSet // mutable Set(2, 3, 8, 5)

val immutaSet =
scala.collection.immutable.Set.empty ++ mutaSet
// immutable Set(2, 3, 8, 5) scala.collection.
```

Tuples (1/2)

Associates up to 22 values to each other. To return multiple values from functions. The relationship has no meaning.

Example

```
def longestWord(words: Array[String]) = {
var word = words(0)
var index = 0
for (i <- 1 until words.length)
   if(words(i).length > word.length){
      word = words(i)
      index = i
   }
(word, index)
}
```

Tuples (2/2)

```
// example cont.

scala> val (word, index) =
longestWord(“The fast and the
furious”.split(“ ”)) // word = furious,
index = 4


scala> val word, index =
longestWord(“The fast and the
furious”.split(“ ”)) // word =(furious,
4), index = (furious, 4)
```

Questions?

Chapter 18
Stateful Objects

What makes an object stateful?

- **Pure functional objects** produces the same results each time a method is called
- **Pure functional objects** can still have internal states that do not externally expose values;
  example: internal cache of values
- **Stateful objects** changes their externally observable state over time
- What are pure functional objects good for?

  \+ Easier to reason about; predictable results
- What are stateful objects good for?

  \+ Good for modeling real-world objects that change their properties over time; example: bank account

Reassignable variables

- The definition of `var hour = 12` generates a setter and getter for a field marked private[this]
- These classes are exactly equivalent:

```
class Time {
  var hour = 12
  var minute = 0
}
```

... is expanded to:

```
class Time {

  private[this] var h = 12
  private[this] var m = 0

  def hour: Int = h
  def hour_=(x: Int) { h = x }

  def minute: Int = m
  def minute_=(x: Int) { m = x }
}
```

Properties
– redefining getters and setters

- Example:
  Ensuring that a certain property holds by checking it in a setter:

```
class Time {
  private[this] var h = 12
  private[this] var m = 0
  def hour: Int = h
  def hour_= (x: Int) {
    require(0 <= x && x < 24)
    h = x
  }
  def minute = m
  def minute_= (x: Int) {
    require(0 <= x && x < 60)
    m = x
  }
}
```

Type Parameterization

Gustav Cedersjö

Outline

- Type parameters
- Type bounds
- Variance

Type parameters

- Type parameters are between [and]
- In classes and traits:
  ```
  class Array[A]
  ```
- In methods:
  ```
  def contains[A](x: A, List[A]): Bool
  ```

```
class Box[A](private var value: A) {
  def get: A = value
  def set(x: A) {
    value = x
  }
}
```

```
scala> val answerBox = new Box(42)
answerBox: Box[Int] = Box@462c4b0b

scala> val lunchBox = new Box("kebab")
lunchBox: Box[String] = Box@36160b84

scala> lunchBox.set("nobelmiddag")

scala> val lunch = lunchBox.get
lunch: String = nobelmiddag
```

Type bounds

- A <: B means that A is a subtype of B
- B >: A means that B is a supertype of A
- Can be used to restrict type parameters

```
scala> def compareBox[A <: Comparable[A]](
     |   x: Box[A], y: Box[A]) =
     |     x.get.compareTo(y.get)
compareBox: [A <: Comparable[A]](x: Box[A],
y: Box[A])Int

scala> compareBox(new Box("a"), new Box("b"))
res0: Int = -1
```

Variance

- List[String] is a subtype of List[Any]
- Array[String] is *not* a subtype of Array[Any]
- In Java List<String> is *not* a subtype of List<Object>, but String[] is a subtype of Object[]

Variance

Covariance	class Box[+A]	Box[String] <: Box[Any]
Contravariance	class Box[-A]	Box[Any] <: Box[String]
Non-variance	class Box[A]	neither

```
trait SimpleList[+A] {
  def head: A
  def tail: SimpleList[A]
  def isEmpty: Bool
  def prepend[B >: A](b: B) = new Cons(b, this)
}

class Cons[+A](val head: A, val tail: SimpleList[A])
    extends SimpleList[A] {
  def isEmpty = false
}

object EmptyList extends SimpleList[Nothing] {
  def head = sys.error(”empty”)
  def tail = sys.error(”empty”)
  def isEmpty = true
}
```

Abstract members
Chapter 20

Niklas Fors

August 27, 2012

Abstract members

Different kinds of abstract members:

```
trait Abstract {
    type T
    def transform(x: T): T
    val initial: T
    var current: T
}
```

Implementation:

```
class Concrete extends Abstract {
    type T = Int
    def transform(x: Int) = x*2
    val initial = 1
    val current = initial
}
```

Abstract vals (1/3)

Problem: Initializing abstract `vals`:

```
trait T {
    val nArg: Int
    require(nArg != 0)
    val n = nArg*2
    override def toString = n.toString
}

scala> new T {
           val nArg = 1
       }
java.lang.IllegalArgumentException: requirement failed
...
```

Abstract vals (2/3)

Solution 1: Pre-initialized fields

```
scala> new {
           val nArg = 2
       } with T
res1: java.lang.Object with T = 4
```

Abstract vals (3/3)

Solution 2: Lazy `vals`

```
trait T {
    val nArg: Int
    lazy val n = {
        require(nArg != 0)
        nArg*2
    }
    override def toString = n.toString
}

scala> new T {
           val nArg = 1
       }
res1: java.lang.Object with T = 2
```

Abstract types (1/2)

```
class Food
abstract class Animal {
    def eat(food: Food)
}

class Grass extends Food
class Cow extends Animal {
    override def eat(food: Grass) { }   // Compile error
}
```

Abstract types (2/2)

```
class Food
class Grass extends Food
class Fish extends Food
abstract class Animal {
    type SuitableFood <: Food
    def eat(food: SuitableFood)
}
class Cow extends Animal {
    type SuitableFood = Grass
    override def eat(food: Grass) { }
}

val cow = new Cow
cow.eat(new Grass)           // OK
cow.eat(new Fish)            // Compile error
```

Path-dependent type (1/3)

Path-dependent type:

```
scala> val bessy: Animal = new Cow
scala> bessy.eat(new Fish)
<console>:13: error: type mismatch;
 found   : Fish
 required: bessy.SuitableFood
              bessy.eat(new Fish)
                        ^
```

Path-dependent type (2/3)

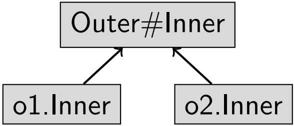
Using path-dependent type to create an object:

```
scala> val bessy = new Cow
scala> val moo = new Cow
scala> bessy.eat(new bessy.SuitableFood)
scala> bessy.eat(new moo.SuitableFood)
```

Path-dependent type (3/3)

```
class Outer {
    class Inner {
    }
}

> val o1 = new Outer
> val o2 = new Outer
> val i1 = new o1.Inner
i1: o1.Inner = ...
> val i2 = new o2.Inner
i2: o2.Inner = ...
```



Structural subtyping

Scala supports both *nominal subtyping* (as in Java) and *structural subtyping* (members). Structural subtyping:

```
def f[T <: { def close(): Unit } ](obj: T) {
    obj.close()
}
```

IMPLICIT CONVERSIONS AND PARAMETERS

Implicit Cheating

Outline

- Motivation
- Usage
- Rules
- Where
- Ambiguity

Motivation

- Someone else's code
- Class extension
- "Two bodies of software that were developed without each other in mind"
- Reduce boilerplate

Usage

- Generic scenario:
  - Encapsulate the class to be extended
  - Add the extending functionality to the wrapper class
  - Define an implicit conversion that converts the extended class to the wrapper class

```
class RichInt (i : Int) {
   def triple() = i*3 //
}

implicit def int2RichInt(i : Int) = new RichInt(i)

print(3.triple)
```

Rules

- Marking: `implicit`
- Scope
  - In scope
  - Single identifier
    - `someVariable.convert` – will not work
- One at a time
  - No implicit conversion inserted while already trying another one
    - `x+y => convert1(convert2(x)) + y` – does not happen
- Explicits-First
  - Working code will not change.

Where

- Conversion to expected type
- Converting the receiver
- Implicit parameters

Expected Type

- Assume types X and Y that have no subtype-supertype relation:

```
val y: Y = new Y(…)
val x: X = y
```

- Initially, this is compile-time error.
- Before giving up hope, compiler will try to find an implicit conversion from X to Y in the scope.
  - `implicit def xToY(x:X) = new Y(x)`

Receiver

- Applied on the method call receiver.
  - First example:

```
class RichInt (i : Int) {
   def triple() = i*3
}

implicit def int2RichInt(i : Int) = new RichInt(i)

print(3.triple)//conversion on the Int as the
         //reciever of triple()
```

Implicit Parameters

- Parameter lists can be implicit:
  - ```
    def foo(a:Int, b:Int)(implicit c: Int) = a+b+c
    implicit val impInt = 3
    foo(1,2)
    ```
  - Often used to provide information about a type that is explicit in an earlier parameter list
    - ```
      def maxListImpParm[T](elements: List[T])
            (implicit orderer: T => Ordered[T]): T
      ```
    - The implicit parameter `orderer` implies that `T` is a type that can be converted to `Ordered[T]`

Ambiguity

- What happens when multiple conversions apply?
- *Strictly more specific* conversion is applied
  - X is *more specific* than Y if:
    - Argument type of X is a subtype of Y's or,
    - X and Y are methods, and enclosing class of X extends the enclosing class of Y

`for` expressions revisited

Björn A. Johnsson

`for` expressions

Higher-order functions such as `map`, `flatMap` and `filterWith` provide powerful means for manipulating lists.

```
scala> persons withFilter (p => !p.isMale) flatMap (p
    => (p.children map (c => (p.name, c.name))))
res0: List[(String, String)] = List((Julie,Lara),
    (Julie,Bob))
```

Did you get all that?! No worries, it's not exactly trivial to either write nor read. The same thing can be expressed in a simpler way using for expressions:

```
scala> for (p <- persons; if !p.isMale; c <-
    p.children) yield (p.name, c.name)
```

`for` expressions (cont'd)

General form:

```
for ( seq ) yield expr
```

where *seq* is a sequence of *generators*, *definitions* and *filter*, separated by semicolons (or new line).

Example:

```
for (
  p <- persons
  n = p.name
  if (n startsWith "To")
) yield n
```

Generators

A *generator* has the following form:

```
pat <- expr
```

The expression `expr` typically returns a list.

The pattern `pat` is matched against the elements of `expr` one-by-one:

- On success, the variables of `pat` are bound to the corresponding parts of the current element (*pattern matching*).
- On failure, the current element is discarded (no errors thrown).

Example:

```
p <- persons
```

Generators (cont'd)

All `for` expressions start with a generator. If there are multiple generators, they are iterated over from most nested generator to least nested generator.

Example:

```
scala> for (x <- List(1, 2); y <- List("A", "B"))
     | yield (x, y)
res1: List[(Int, java.lang.String)] = List((1,A),
   (1,B), (2,A), (2,B))
```

Definitions

A *definition* has the following form:

```
pat = expr
```

This binds the variables of `pat` to the corresponding parts of `expr` (*pattern matching*). Same effect as a `val` definition:

```
val x = expr
```

Example:

```
n = p.name
```

Filters

A *filter* has the following form:

```
if expr
```

`expr` is of typ `Boolean`. All elements for which `expr` return `false` are discarded.

Example:

```
if (n startsWith "To")
```

Translation of `for` expressions

The Scala compiler translates:

- all `for` expressions that `yield` a result into combinations of the higher-order methods `map`, `flatMap` and `withFilter`.
- all `for` loops without `yield` into combinations of `withFilter` and `foreach`.

```
for (x <- expr1) yield expr2
↪ expr1 map (x => expr2)

for (x <- expr1 if expr2) yield expr3
↪  for (x <- expr1 withFilter (x => expr2)) yield expr3
   ↪  expr1 withFilter (x => expr2) map (x => expr3)

// and many more...
```

Generalizing `for`

Besides lists, arrays, etc., user created data types can also support `for` expressions.

For full support of `for` expressions, the data type must define `map`, `flatMap`, `withFilter` and `foreach`.

Partial `for` support follows these rules:

- `map` → `for` expression w/ single generator
- `map` & `flatMap` → `for` expression w/ several generators
- `foreach` → `for` loops w/ several generators
- `withFilter` → `for` expression w/ filters

Time for discussion?

Collections and partial functions

Christian.Soderberg@cs.lth.se

29 augusti 2012

Partial functions – recap

- If we begin a block with a `case`-statement (such as inside a `match`-statement), we get a *partial function*:

  ```
  ... {
    case a: A => ...
  }
  ```

  The function may be undefined for some input values, and if we call the function for one of them, we'll get a runtime error.
- We can have several cases after each other, and we can use guards:

  ```
  ... {
    case n if n > 0 => ...
    case n if n % 10 == 0 => ...
    case _ => ...
  }
  ```

More on partial functions

- It’s sometimes convenient to use partial functions, instead of writing

```
sentence.zipWithIndex.map {
  p => ”%2d: %s”.format(p._2, p._1)
}.foreach(println)
```

we can write:

```
sentence.zipWithIndex.map {
  case (word, index) => ”%2d: %s”.format(index, word)
}.foreach(println)
```

- To use this syntax, we have to use braces instead of parentheses – partial functions only resides in blocks.

More on partial functions

- A partial function can be saved in a `val`:

```
val lowSpeedDrag: PartialFunction[Double,Double] = {
  case v if v < 10 => alpha * v
}
```

- We can 'chain' partial functions using `orElse`:

```
val highSpeedDrag: PartialFunction[Double,Double] = {
  case v if v >= 10 => beta * v * v
}
val drag = lowSpeedDrag orElse highSpeedDrag
(5.0 to 15.0 by 2.0).map(drag).foreach(println)
```

Some collections are partial functions

- Indexed collections are partial functions from `Int` to some type `A`:

```
val sentence = Seq(”to”, ”be”, ”or”, ”not”)
val someWord = sentence(2)    // Int => String
val noWord = sentence(10)     // runtime error
```

- A `Map[K,V]` is a partial function from `K` to `V`:

```
val numbers = Map(”one” -> 1, ”two” -> 2, ”three” -> 3)
println(numbers(”two”))       // String => Int
println(numbers(”four”))      // runtime error
```

A special kind of partial function

- We can match using regular expressions:

```
val EmailAddress = """([\w\.]+)@([\w\.]+)""".r
val PhoneNumber = """(\d+)-(\d+)""".r
```

- If a string matches a regular expression, it will look as if we had a `case class` with the same name as the constant defining the regular expression:

```
string match {
  case EmailAddress(name, domain) =>
    "%s(at)%s".format(name, domain)
  case PhoneNumber(areaCode, number) =>
    "area: %s, number: %s".format(areaCode, number)
  case other =>
    "unknown string: " + other
}
```

- We will get a runtime error if we have no case which matches the input (the last case above catches all strings).

Matching with regular expressions

- We don’t need to use a `match`-statement when we map over a sequence, we can use a partial function directly:

```
val input = Seq(”karl.gustaf@royalcourt.se”,
                ”Kungliga slottet”, ”08-4026000”)
input.map {
  case EmailAddress(name, domain) =>
    ”%s(at)%s”.format(name, domain)
  case PhoneNumber(areaCode, number) =>
    ”area: %s, number: %s”.format(areaCode, number)
  case other =>
    ”unknown string: ” + other
}.foreach(println)
```

- We will still get a runtime error if we have no case which matches the input.

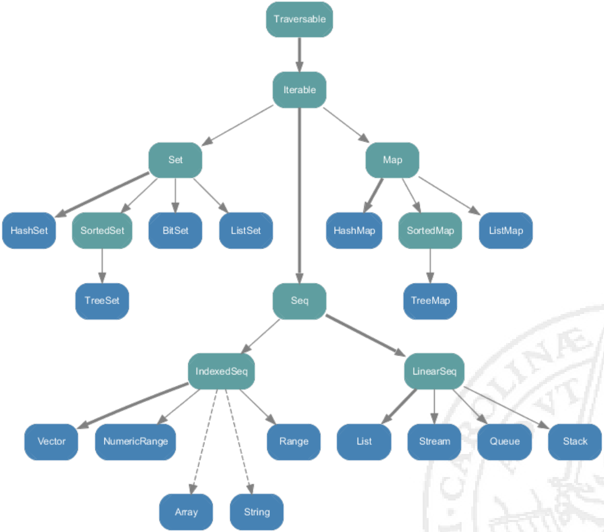
Collecting values

- We can also use `collect`, to collect all the values our partial function can compute:

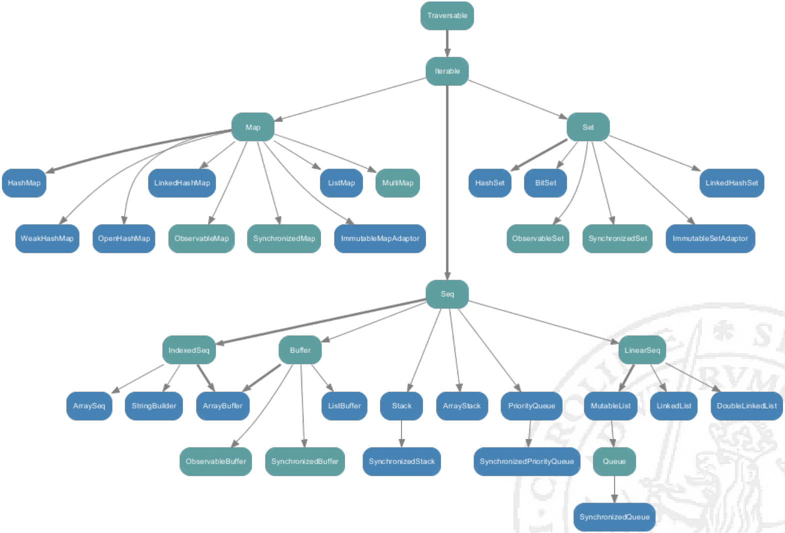
```
input.collect {
  case EmailAddress(name, domain) =>
    ”%s(at)%s”.format(name, domain)
  case PhoneNumber(areaCode, number) =>
    ”area: %s, number: %s”.format(areaCode, number)
}.foreach(println)
```

- In this case we don’t need to cover all cases – only those cases where our partial function is defined will give us values.
- The `flatMap` method works analogously, but collects optional values which are embedded in a Some-object.

Overview of classes



Overview of classes



Creating generic collections

```
// This works
def evenElems[T: ClassManifest](xs: Vector[T]): Array[T] = {
  val arr = new Array[T]((xs.length + 1) / 2)
  for (i <- 0 until xs.length by 2)
    arr(i / 2) = xs(i)
  arr
}
```