Języki i środowiska przetwarzania danych rozproszonych

Scala 1

Szymon Grabowski
sgrabow@kis.p.lodz.pl

Based mostly on:
C. Horstmann, *Scala for the Impatient*, 2012
M. Odersky et al., *Programming in Scala, 2nd Ed.*, 2010
and http://www.cs.columbia.edu/~bauer/cs3101-2/

Łódź, 2016

Install

www.scala-lang.org/downloads
http://www.scala-sbt.org/download.html (SBT is a build tool)

scala -version // scala[.bat] – opens the REPL
compiler: scalac

```
scala> 12 * 11.9
res0: Double = 142.8
scala> :quit  // or :q
```

```
// ScriptDemo.scala
println("Hello, Scala!")
```

To execute a script:
scala ScriptDemo.scala

val and var

val (=value) is immutable
var (=variable) is mutable

If possible, prefer values over variables!

```
val x = 5
// x = 2  // ERROR!
val x1, x2 = 20
```

Type inference:
from the init with e.g. 5 Scala ‘knows’ that our value is of type Int.
Same with Double, String, Vector…

Or:

```
val x: Int = 5; val y: Double = -3.2;

val anotherDouble = -3.2
val text = "Hello"  // or: val text: String = "Hello"
var x = 3.5; x = -1
```

Some more types

Boolean: true | false

```
val keyKnown = false
val ok: Boolean = true
```

Vector – a popular container

scala.collection.immutable.Vector[…]

```
val v = Vector(3, 2, 10, 5)
var v2 = v.sorted
println(v2)
// v(0) = -1  // ERROR
v2 = v2.reverse  // it's a method call!
println(v2)
val v3 = v updated(1, 99)  // Vector(3, 99, 10, 5)
```

```
val r1 = Range(5, 8)  // 5, 6, 7
val r2 = Range(5, 8).inclusive
                      //  5, 6, 7, 8
```

Expressions

An expression produces a result.
E.g.:
scala> val i = 4;
i: Int = 4;
scala> i + 3
res1: Int = 7

```
// Expressions.scala

val c = {
  val i1 = 2
  val j1 = 4/i1
  i1 * j1
}
println(c)
/* Output:
4
*/
```

Watch this!

```
scala> val e = print(5)
5e: Unit = ()
```

Unit type: more or less equivalent to void in Java.

Brevity in Scala (some examples)

- return keyword may be omitted (the last expression in a compound expression is the result)
- semicolons may be omitted
- type inference
- parens in method calls may be omitted if zero or one argument follows – only in the so-called *operator notation*!
- dot after the object name in method call may be omitted in the same case as above

The last two listed rules in action:

```
3 * 4
// It's the same as: 3.*(4)

val ell = List(2, 3, 5, 7, 11)
print(ell head)  // 2
Console print ell.head  // 2
```

One of my most productive days was throwing away 1000 lines of code.
/ Ken Thompson,
co-designer of Unix /

Scala's best
(from Java to Scala)

```
public class Person
{
  private  String firstName;
  private  String lastName;
  String getFirstName() { return firstName; }
  void setFirstName(String firstName) { this.firstName = firstName; }
  String getLastName() { return  lastName; }
  void setLastName(String lastName) { this.lastName = lastName; }
  int hashCode() ....
  boolean equals(Object o) { ....  }
}
```

```
case class Person(var firstName: String, var lastName: String)
```

Methods with no parameters

As said, parens are then not needed.
Yet, it's good style to use () for a mutator method
and drop the () for an accessor method.

```
myCounter.increment() // Use () with mutator
println(myCounter.current) // Don't use () with accessor
```

Even more: we can enforce this convention:
class Counter { def current = value // no () }

Now the invocation myCounter.current() is forbidden
(only myCounter.current is correct).

On type inference

```
scala> val z : Boolean = if (20 > 10) true
<console>:7: error : type mismatch ;
found : Unit
required : Boolean
(...)
```

```
scala > val z = if (20 > 10) true
z: AnyVal = true
```

Are these two functions equivalent?

```
def check1(x: Double) = if (x < 1.0 || x > 10.0) false else true

def check2(x: Double) = if (x >= 1.0 && x <= 10.0) true else false
```

Is check1(x) == check2(x) for any x?

No.

```
val x = Double.NaN  // or: … = 0.0 / 0.0

print(check1(x), check2(x))
// (true,false)
```

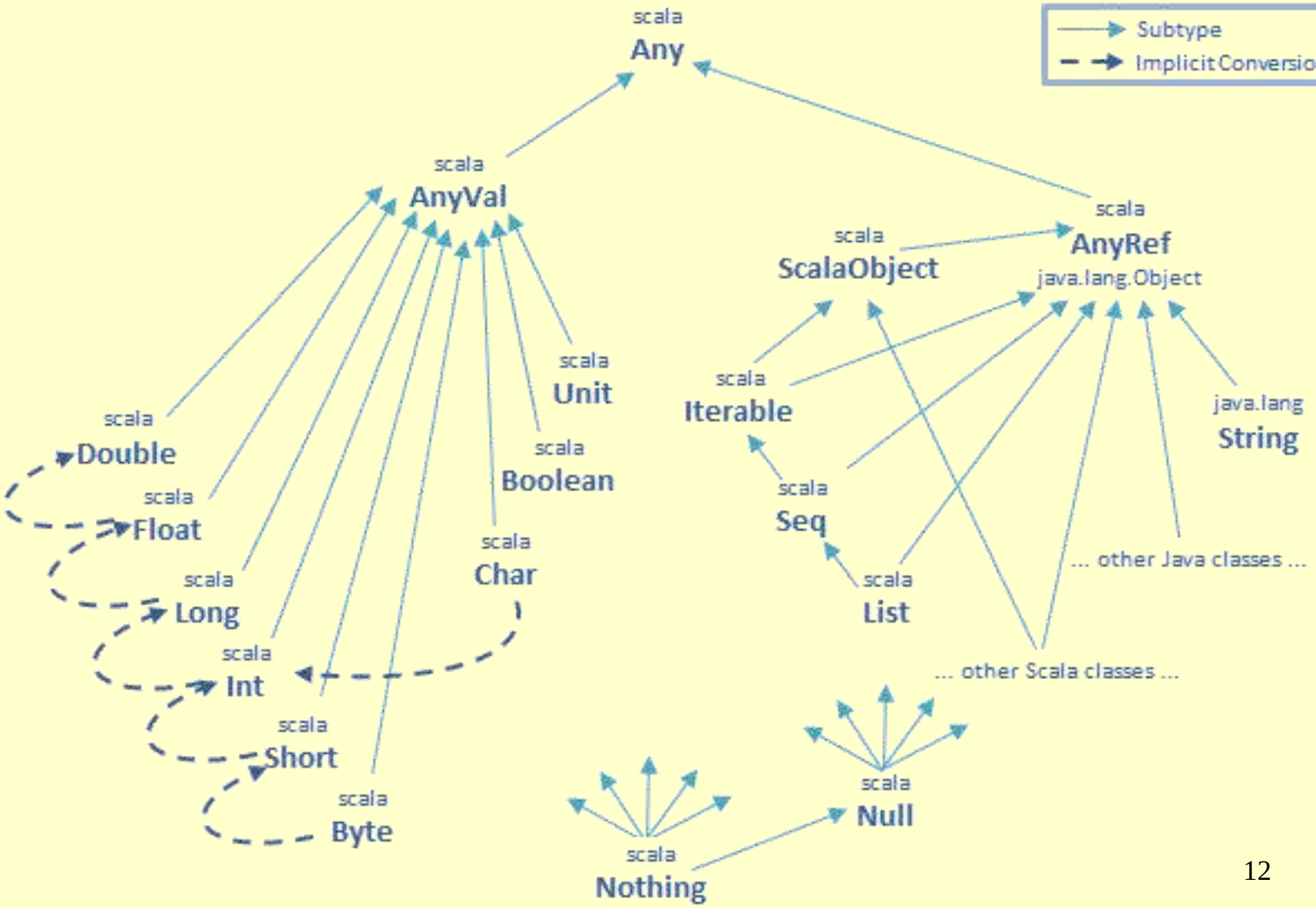
Functions in Scala are objects

Their type depends on how many arguments they have.
E.g. Function2[Int, String, Boolean] (which extends AnyRef)
means a 2-argument function (Int, String) => Boolean.

A call f(3, "abc") will be expanded to f.apply(3, "abc").

Methods can be used as functions:
if you write a method name
where a Function object is required,
the compiler will create a Function object for you.

Type hierarchy



A note on Scala types

The Scala classes Any, AnyRef and AnyVal
don't appear as classes in bytecode,
because of intrinsic limitations of the JVM.
(in Java not everything is an object!).

In Scala, on the other hand, everything *is* an object,
all objects belong to a class, and they interact through methods.
The JVM bytecode generated does not reflect this.

So, in Scala, all objects are descendant from Any,
and that includes both what
Java considers objects and what Java considers primitives.
Everything that is considered a primitive in Java
is descendant from AnyVal in Scala.
AnyRef in Scala is equivalent to java.lang.Object
(on the JVM; on .NET it was an alias for System.Object).

http://stackoverflow.com/questions/2335319/
what-are-the-relationships-between-any-anyval-anyref-object-and-how-do-they-m

The Java primitive types

You can't write: new Int(5) (nor Int(5), new Int etc.) in Scala.
error: class Int is abstract; cannot be instantiated

Int, Double etc. classes are abstract and final.
The instances of these classes are
all written as literals in Scala.
Internally, Scala stores e.g. Ints are 32-bit integers,
exactly like Java's int values.
Good for efficiency and interoperability with Java libraries.

Type Nothing inherits any other type

A `throw` expression has the special type `Nothing`. That is useful in `if/else` expressions. If one branch has type `Nothing`, the type of the `if/else` expression is the type of the other branch. For example, consider

```
if (x >= 0) { sqrt(x)
} else throw new IllegalArgumentException("x should not be negative")
```

The first branch has type `Double`, the second has type `Nothing`. Therefore, the `if/else` expression also has type `Double`.

Nothing != Unit. Nothing has no instances.

Lists

Lists (like strings) are immutable.
List has *O*(1) prepend and head/tail access. Most other operations (incl. index-based lookup, length, append) take *O*(*n*) time.

```
scala> val l0 = Nil  // the empty list
res1: scala.collection.immutable.Nil.type = List()

scala> val l = 1 :: 2 :: Nil  // :: is pronounced "cons"
l: List[Int] = List(1, 2)

scala> val m = List(3, 4, 5)
m: List[Int] = List(3, 4, 5)

scala> l ::: m
res2: List[Int] = List(1, 2, 3, 4, 5)
```

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

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

‘cons’ operator

```
scala> 1 :: List(2,3)
res18: List[Int] = List(1, 2, 3)

scala> List(2,3).::(1)
List[Int] = List(1, 2, 3)
```

Cons behavior is specific.

Lists are constructed right-to-left.

General rule: if an operator ends in :
it is translated into a method call on the right operand.

Some operations on lists

Indexing: list(0)

Slicing: list.slice(1, 3); list.slice(2, list.last)

Reversing: list.reverse

Sorting: list.sorted

Partitioning according to a predicate:
partition (p : (A) => Boolean) : (List[A], List[A])

span (p : (A) => Boolean) : (List[A], List[A])
Returns the longest prefix of the list whose elements all satisfy the given predicate, and the rest of the list.

=> called informally a rocket operator

Some operations on lists, cont'd

forall

```
val list = List("boo", "woo", "woo")
print(list forall(it => it endsWith "o"))  // true
print((list ::: "o!" :: Nil) forall(it => it endsWith "o"))  // false
```

exists

```
val list = List(5, -3, 2, 1)
print(list exists(x => x % 2 == 0))  // true
```

They work for many other collections (and strings) too.
(For maps, however, it's probably much more efficient to use *contains* than *exists*.)

```
val s = "STRING"; println(s forall Character.isUpperCase)  // true
```

What is Nil?

Nil is defined as:

```
package scala.collection.immutable
object Nil extends List[Nothing] with Product with
Serializable
```

List[A] is a covariant type (more on this later), which means that e.g. List[String] is a subtype of List[AnyRef], or List[Nothing] is a subtype of List[A] for any type A.

Therefore, we don’t need to write Nil[A] for all types A! One Nil object is enough.

Dean Wampler & Alex Payne, *Programming Scala*, 2nd Ed., 2015, Chap. 10.

Extract Method (common refactoring technique)

```
void scalarProduct(String[] params) {
  int[] x = prepareX(params);
  int[] y = prepareY(params);
  int[] product = computeXY(x, y);
  //
  for (i = 0; i < x.length; i++) {
    out.println("X = " + x[i]);
    out.println("Y = " + y[i]);
    out.println("X * Y = " + product[i]);
  }
```

```
void scalarProduct(String[] params) {
  int[] x = prepareX(params);
  int[] y = prepareY(params);
  int[] product = computeXY(x, y);
  // ...
  printScalarProduct(x, y, product);
```

```
void printScalarProduct(int[] x, int[]y,
int[] product) {
  for (i = 0; i < x.length; i++) {
    out.println("X = " + x[i]);
    out.println("Y = " + y[i]);
    out.println("X * Y = " + product[i]);
  }
```

Example from:
http://wazniak.mimuw.edu.pl/images/d/d8/Zpo-8-wyk.pdf

Can we always easily extract a method?
(And what does it have to do with Scala?)

If some variable from scalarProduct(…) changes in the supposed printScalarProduct(…), we are unlucky…

In Scala: we could have e.g.
val x: List[Int] = … (same with y, product)
and we're safe from possible side-effects.

for loop

for (y <- List(1, 2, 3)) { println(y) }

for as an expression:

```
scala> for (x <- List(1,2,3)) yield x*2
res8: List[Int] = List(2, 4, 6)

scala> for { x <- 1 to 7 // generator
             y = x % 2;     // definition
             if (y == 0)    // filter
           } yield {
             println(x)
             x
           }
2
4
6
res1: scala.collection.immutable.IndexedSeq[Int] =
    Vector(2, 4, 6)
```

```
scala> for {x <- List(1,2,3);
            y<-List(4,5))       } yield x * y
res10: List[Int] = List(4, 5, 8, 10, 12, 15)
```

for loop: multiple filters possible, variable binding…

```
var myArray : Array[Array[String]] = new Array[Array[String]](10);
...

for(anArray : Array[String] <- myArray;
    aString : String        <- anArray;
    aStringUC = aString.toUpperCase()
    if aStringUC.indexOf("VALUE") != -1;
    if aStringUC.indexOf("5") != -1
    ) {
    println(aString);
}
```

for / yield = for comprehension

The generated collection is compatible with the first generator.

```
for (c <- "Hello"; i <- 0 to 1) yield (c + i).toChar
  // Yields "HIeflmlmop"
for (i <- 0 to 1; c <- "Hello") yield (c + i).toChar
  // Yields Vector('H', 'e', 'l', 'l', 'o', 'I', 'f', 'm', 'm', 'p')
```


http://tutorials.jenkov.com/scala/for.html

Operators

Formally, there's no "operator overloading" in Scala;
e.g. + is "just a function/method".
Yet, there are some things to remember…

If a plain (method/function or var) identifier starts with a letter or _,
it cannot contain operator symbols (+-*/< etc.).
E.g. val xyz++= = 1 won't compile.

Operators have priorities and associativity.
2 + 3 * 2 → 8
but 2.+(3).*(2) → 10 !! (2.+(3) * 2 == 10 too)
2 + 3.*(2) == 8 (conclusion: the period binds earlier than e.g. + or *).

Priority depends on the first (leftmost) symbol of an operator,
while associativity on the last (rightmost) symbol.

Creating an operator function

```
def ~=(x: Double, y: Double, precision: Double) = {
  if ((x - y).abs < precision) true else false
}
```

```
scala> val a = 0.3
a: Double = 0.3

scala> val b = 0.1 + 0.2
b: Double = 0.30000000000000004

scala> ~=(a, b, 0.0001)
res0: Boolean = true
```


Alvin Alexander, *Scala Cookbook*, 2013. Chap. 2.5.

Infix operators

Any method with 1 param can be used as an infix operator.

```
case class MyBool(x: Boolean) {
  def and(that: MyBool): MyBool = if (x) that else this
  def or(that: MyBool): MyBool = if (x) this else that
  def negate: MyBool = MyBool(!x)
}
```

Now we can define xor like this:

```
def not(x: MyBool) = x.negate
def xor(x: MyBool, y: MyBool) = (x or y) and not(x and y)
```

More traditionally, it would be like:
def xor(x: MyBool, y: MyBool) = x.or(y).and(x.and(y).negate)

http://docs.scala-lang.org/tutorials/tour/operators.html

Arrays are mutable

```
scala> val a = Array(1,2,3,4)
a: Array[Int] = Array(1, 2, 3, 4)

scala> a(3) = 100

scala> a
res2: Array[Int] = Array(1, 2, 3, 100)
```

There are mutable and immutable versions of many reference types in Scala.

If possible, use immutable ones!

Hints on arrays

- Use an `Array` if the length is fixed, and an `ArrayBuffer` if the length can vary.
- Don't use `new` when supplying initial values.
- Use `()` to access elements.
- Use `for (elem <- arr)` to traverse the elements.
- Use `for (elem <- arr if ... ) ... yield ...` to transform into a new array.
- Scala and Java arrays are interoperable; with `ArrayBuffer`, use `scala.collection.JavaConversions`.

```
val nums = new Array[Int](10)
  // An array of ten integers, all initialized with zero
val a = new Array[String](10)
  // A string array with ten elements, all initialized with null
val s = Array("Hello", "World")
  // An Array[String] of length 2—the type is inferred
  // Note: No new when you supply initial values
```

Multi-dim array: val a = Array.ofDim[Int](3, 2)

ArrayBuffer is equivalent of java.util.ArrayList

```
import scala.collection.mutable.ArrayBuffer
val b = ArrayBuffer[Int]()
  // Or new ArrayBuffer[Int]
  // An empty array buffer, ready to hold integers
b += 1
  // ArrayBuffer(1)
  // Add an element at the end with +=
b += (1, 2, 3, 5)
  // ArrayBuffer(1, 1, 2, 3, 5)
  // Add multiple elements at the end by enclosing them in parentheses
b ++= Array(8, 13, 21)
  // ArrayBuffer(1, 1, 2, 3, 5, 8, 13, 21)
  // You can append any collection with the ++= operator
b.trimEnd(5)
  // ArrayBuffer(1, 1, 2)
  // Removes the last five elements
```

Conversions to Array/ArrayBuffer:
b.toArray, a.toBuffer

Traversing Arrays (and ArrayBuffers)

```
for (i <- 0 until a.length)
  println(i + ": " + a(i))

// or:
for (i <- 0 to a.length-1) ...

// or (just the element, no index):
for (elem <- a) println(elem)


0 until (a.length, 2)
// Range(0, 2, 4, ...)

(0 until a.length).reverse
// Range(..., 2, 1, 0)
```

Transforming arrays

arr is an array of integers,
we want to double the even elements (even values, not at even positions) and reject odd ones.

Possible (and equivalent) solutions:

```
for (elem <- arr if elem % 2 == 0) yield 2 * elem
arr.filter(_ % 2 == 0).map(2 * _)
arr filter { _ % 2 == 0 } map { 2 * _ }
```

Sort an array:

```
val b = ArrayBuffer(1, 7, 2, 9)  // could be Array(…) as well
val bSorted = b.sorted  // ArrayBuffer(1, 2, 7, 9)
val bSorted2 = b.sortWith(_ > _)  // ArrayBuffer(9, 7, 2, 1)
```

Some other functions on arrays

```
arr.mkString(" and ")
  // "1 and 2 and 7 and 9"
arr.mkString("<", ",", ">")
  // "<1,2,7,9>"
```

```
ArrayBuffer("Mary", "had", "a", "little", "lamb").max
// "little"
```

Sorting an Array (but not an ArrayBuffer) in place:

```
val a = Array(1, 7, 2, 9)
scala.util.Sorting.quickSort(a)
// a is now Array(1, 2, 7, 9)
// works with numbers, strings and generally all types
// with the Ordered trait (i.e. with defined comparison op)
```

Tuples

```
val things = (100, "Foo")
println(things._1)
println(things._2)
```

```
(1, 3.14, "Fred")
```

is a tuple of type

```
Tuple3[Int, Double, java.lang.String]
```

which is also written as

```
(Int, Double, java.lang.String)
```

NOTE: You can write `t._2` as `t _2` (with a space instead of a period), but not `t_2`.

Tuples, cont'd

Usually, it is better to use pattern matching to get at the components of a tuple, for example

```
val (first, second, third) = t // Sets first to 1, second to 3.14, third to "Fred"
```

You can use a _ if you don't need all components:

```
val (first, second, _) = t
```

Tuples are useful for functions that return more than one value. For example, the partition method of the StringOps class returns a pair of strings, containing the characters that fulfill a condition and those that don't:

```
"New York".partition(_.isUpper) // Yields the pair ("NY", "ew ork")
```

```
val (number, string) = (12, "hi")
```

Tuples and zipping

```
val symbols = Array("<", "-", ">")
val counts = Array(2, 10, 2)
val pairs = symbols.zip(counts)
```

yields an array of pairs

```
Array(("<", 2), ("-", 10), (">", 2))
```

The pairs can then be processed together:

```
for ((s, n) <- pairs) Console.print(s * n) // Prints <<---------->>
```

2-element tuples created with the method ->

```
"Poland" -> "Warsaw"
// (java.lang.String, java.lang.String) = (Poland,Warsaw)

3 -> "abc"
// (Int, java.lang.String) = (3,abc)

class X
new X -> "1" // still OK!  So, how does it work?
```

You can invoke the -> method on any object
and it works (as expected).
Thanks to the **implicit conversion** mechanism.

A tuple isn't a collection, but…

we can treat a tuple as a collection via an iterator:

```
val t = ("Poland" -> "Warsaw")
val it = t.productIterator
for (e <- it) print(e + " ")  // Poland Warsaw
```

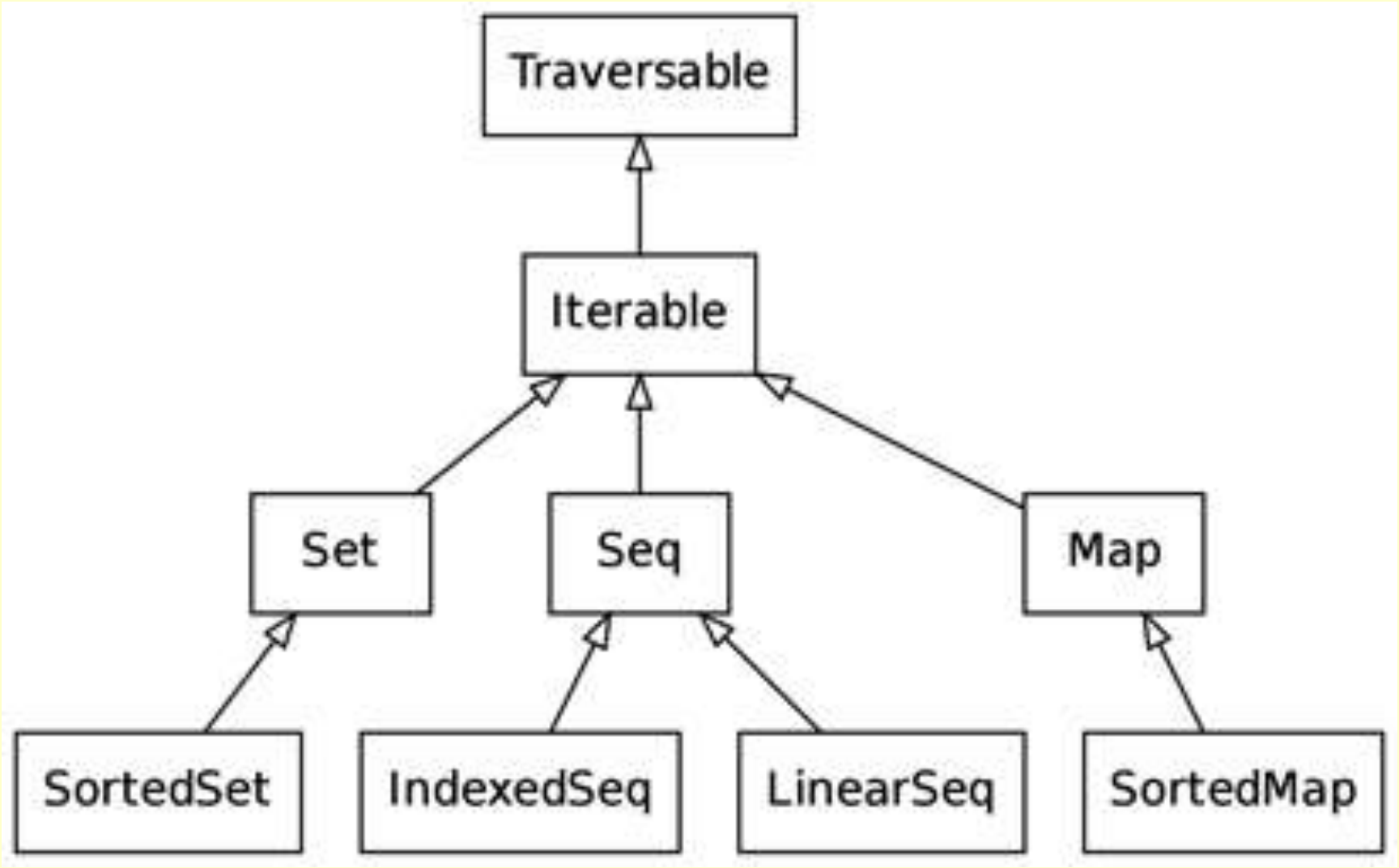
Converting a tuple to a collection:

```
t.productIterator.toArray
// Array[Any] = Array(Poland, Warsaw)
```

Collections in Scala
(more systematically)

- All collections extend the Iterable trait.
- The three major categories of collections are sequences, sets, and maps.
- Scala has mutable and immutable versions of most collections.
- Sets are unordered collections.
- Use a LinkedHashSet to retain the insertion order or a SortedSet to iterate in sorted order.
- + adds an element to an unordered collection; +: and :+ prepend or append to a sequence; ++ concatenates two collections; - and -- remove elements.

Key traits in the Scala collections library



http://www.wikiwand.com/nl/Scala_%28programmeertaal%29

Maps

Maps are iterables over (key, values) pairs.

```
scala> val capitals = Map("Japan"->"Tokyo",
                          "France"->"Paris")
capitals: scala.collection.immutable.Map[String,String] =
        Map(Japan -> Tokyo, France -> Paris)

scala> for ((country,city) <- capitals)
          println("Capital of " + country + ": " + city)
Capital of Japan: Tokyo
Capital of France: Paris
```

Map.get(key) returns an Option type.

Checking if a string contains any special chars

We want to check if a given string contains [a-zA-Z0-9] chars only, or something else.
Using a regex is one solution, but we can go without it.

```
val ordinary = (('a' to 'z') ++ ('A' to 'Z') ++ ('0' to '9')).toSet

def isOrdinary(s: String) = s forall(ordinary contains _)

print(isOrdinary("1Abc"), isOrdinary("1 Abc"))  // (true,false)
```

Implicit conversions

Implicit type conversions (also called views) are special functions that are automatically applied by the compiler if necessary.
Say, there is a type conversion from A to B, named AtoB. If some function is expecting an argument of type B, but is given an argument of type A, the compiler automatically inserts AtoB.

Example:
"Hello".toList
gets converted into
Predef.stringWrapper("Hello").toList

How to create implicit conversions

```
val button = new JButton
button.addActionListener(
  new ActionListener {
    def actionPerformed(event: ActionEvent) {
      println("pressed!")
    }
  }
)
```

```
implicit def function2ActionListener(f: ActionEvent => Unit) =
  new ActionListener {
    def actionPerformed(event: ActionEvent) = f(event)
  }
```

```
button.addActionListener(
  (_: ActionEvent) => println("pressed!")
)
```

Rules for implicits

Use the keyword implicit.

An inserted implicit conversion must be in scope.

Only one implicit is tried
(the compiler will never rewrite x + y to convert1(convert2(x)) + y).

If the code works without an implicit conversion, no implicits are attempted.

```
object MyConversions {
  implicit def stringWrapper(s: String): IndexedSeq[Char] = ...
  implicit def intToString(x: Int): String = ...
}

import MyConversions.stringWrapper
...
```

Implicit conversion examples

```
implicit def doubleToInt(x: Double) = x.toInt
val tup: (Int, Int) = (1.6, -2.8)   // tup: (Int, Int) = (1,-2)
// note: Double --> Int in Scala truncates towards zero
```

The scala.Predef object, implicitly imported into every Scala program, defines implicit conversions that convert "smaller" numeric types to "larger" ones.

For example, Predef includes:
implicit def int2double(x: Int): Double = x.toDouble
(that's why in Scala e.g. Int values can be stored in variables of type Double; no special rules, only implicit conversions in action).

Imagine:
class Complex(val re: Double, val im: Double) …
How to invoke c + 1, where c of type Complex?
Use an implicit conversion (an Int will be wrapped to a Complex).

Option

Instances of Option are either an instance of scala.Some or the object None.

```
scala> val capitals = Map("Japan"->"Tokyo",
                          "France"->"Paris")
scala> capitals get "Japan"
res0: Option[String] = Some(Tokyo)

scala> capitals get "Italy"
res1: Option[String] = None
```

```
val capitals = Map(...);  val countries = List(...)
for (c <- countries ) {
   val description = capitals.get(c) match {
      case Some(city) => c + ": " + city
      case None => "no entry for " + c
   }
   println(description)
}
```

Option use cases (1/2)

```
var x : Option[String] = None
x.get // java.util.NoSuchElementException: None.get at …
x.getOrElse("default")
x = Some("Now Initialized")
x.get  // String = Now Initialized
x.getOrElse("default") // String = Now Initialized
```

A great feature of Option is that you can treat it as a collection (of size 0 or 1) → can perform map, flatMap and foreach methods, and use inside a for expression.

```
def getTemporaryDirectory(tmpArg: Option[String]): java.io.File = {
   tmpArg.map(name => new java.io.File(name)).
     filter(_.isDirectory).
     getOrElse(new java.io.File(System.getProperty("java.io.tmpdir")))
}
```


Joshua D. Suereth, *Scala in Depth*, 2012. Chap. 2.4.

Option use cases (2/2)

```
val username: Option[String] = ...
for(uname <- username)
  println("User: " + uname)


def authenticateSession(session: HttpSession,
      username: Option[String],
      password: Option[Array[Char]]) = {
  for(u <- username;
       p <- password;
       if canAuthenticate(u, p))  {
      val privileges = privilegesFor(u)  // no need for Option
      injectPrivilegesIntoSession(session, privileges) // ditto
  }
}
```


Joshua D. Suereth, *Scala in Depth*, 2012. Chap. 2.4.

Classes: primary constructor, method(s) and fields

```
class Rational(n: Int, d: Int) {
    println("Created " + n + "/" + d)
}

scala> val r = new Rational(1,2)
Created 1/2
r: Rational = Rational@3394da56
```

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

    val numer = n
    val denom = d

    def +(other : Rational): Rational = {
        val new_n = numer * other.denom + other.numer * denom
        val new_d = denom * other.denom
        new Rational(new_n, new_d)
    }
}

scala> new Rational(1,2) + new Rational(3,4)
res4: Rational = Rational@2a491adf
```

Auxiliary constructors

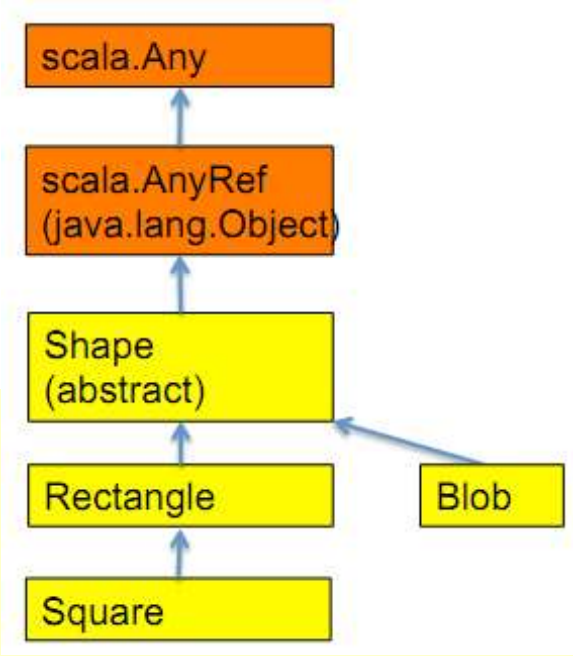
```
class Rational(n: Int, d: Int) {

    val numer = n
    val denom = d

    def this(n : Int) = this(n, 1) // auxiliary constructor

    override def toString = numer + "/" + denom

    def +(other : Rational): Rational = {
        val new_n = numer * other.denom + other.numer * denom
        val new_d = denom * other.denom
        new Rational(new_n, new_d)
    }
}

scala> new Rational(3)
Res0: Rational(3/1)
```

Private methods and fields

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

    // Private val to store greatest common
    // divisor of n and d
    private val g = gcd(n.abs, d.abs)

    val numer = n / g
    val denom = d / g

    def this(n : Int) = this(n, 1) // auxiliary constructor

    override def toString = numer + "/" + denom

    def +(other : Rational): Rational = {
        val new_n = numer * other.denom + other.numer * denom
        val new_d = denom * other.denom
        new Rational(new_n, new_d)
    }

    // Private method to compute the greatest common divisor
    private def gcd(a : Int, b : Int) : Int =
      if (b==0) a else gcd(b, a % b)

}

scala> new Rational(2/4)
Res0: Rational(1/2)
```

Overriding methods and fields

```
abstract class Shape {
    def area : Double
    val description : String
    override def toString = description + ", size: "+area
}

class Blob extends Shape {
    val area : Double = 12;
    val description = "Blob"
}

scala> val x = new Blob
x: Blob = Blob, size: 12.0
```



Getters and setters

In Java we avoid public fields.
So in Scala, but in a different (not obvious at the first look) way.

In Scala, fields in classes automatically come with getters and setters.

```
class Person {
  var age = 0
}
```

Scala generates a class for the JVM with a private field age, and a getter (here: age()) and a setter (here: age_=()) method. We say this class has an age property.

```
val joe = Person
joe.age = 25 // calls joe.age_=(25)
println(joe.age)  // calls joe.age()
```

Getters and setters, cont'd

Do we really want a getter/setter pair for each field?

Often not, and fortunately we have control over it.

If the field is private, the getter and setter are private.

If the field is val, only a getter is generated.

If we don't want any getter or setter,
we declare the field as private[this].

Getter / setter redefinition example

```
class Person {
  private var privateAge = 0 // Make private and rename

  def age = privateAge
  def age_=(newValue: Int) {
    if (newValue > privateAge) privateAge = newValue; // Can't get younger
  }
}
```

Lazy vals

A value is called lazy
when it is evaluated only at its first use.

```
def makeString =
  {println("in makeString"); "hello "+"world";}

def printString = {
  lazy val s = makeString
  print("in printString")
  println(s)
  println(s)
}

scala> printString
in printString
in makeString
hello world
hello world
```

Closures

```
object ClosureDemo1 {
  def main(args: Array[String]): Unit = {
    var total: Int = 100
    var calc = (num: Int) => num + total
    print(calc(5))
  }
}
```

```
object ClosureDemo2 {
  def main(args: Array[String]): Unit = {
    var total: Int = 100
    var calc = (num: Int) => num + total
    total = 2
    print(calc(5))
  }
}
```

Closures, cont'd

```
var (maxFirstLen, maxSecondLen) = (0,0)
list.foreach {
   x => maxFirstLen = max(maxFirstLen, x.firstName.length)
        maxSecondLen = max(maxSecondLen, x.secondName.length)
}
```

We cannot write an equivalent in Java 8, with a lambda, since it is impossible to modify the content the lambda expression has been called from.

http://kukuruku.co/hub/scala/java-8-vs-scala-the-difference-in-approaches-and-mutual-innovations

In Scala, one thing can be expressed in many ways

```
opt.foreach(s => {
  println(s)
})
opt.foreach{s => println(s)}
opt.foreach(s => println(s))
opt foreach(s => println)
opt.foreach(println(_))
opt.foreach(println)
```

```
val list1: List[Integer] = List(1,2,3)
val list2 = List(1,2,3);
val list3 = List.apply(1,2,3)
list1.contains(1)
list1 contains 1
list1 contains(1)
```


http://www.slideshare.net/Bozho/scala-the-good-the-bad-and-the-very-ugly?related=1

Inheritance

Recall that the primary constructor is intertwined with the class definition. The call to the superclass constructor is similarly intertwined. Here is an example:

```
class Employee(name: String, age: Int, val salary : Double) extends
  Person(name, age)
```

This defines a subclass

```
class Employee(name: String, age: Int, val salary : Double) extends
  Person(name, age)
```

and a primary constructor that calls the superclass constructor

```
class Employee(name: String, age: Int, val salary : Double) extends
  Person(name, age)
```

In Scala we never call super(params).

A Scala class can extend a Java class. Its primary constructor must invoke one of the constructors of the Java superclass. For example,

```
class Square(x: Int, y: Int, width: Int) extends
  java.awt.Rectangle(x, y, width, width)
```

Overriding methods

In Scala, you must use the override modifier when you override a method that isn't abstract.

Invoking a superclass method works just like in Java, with the keyword super.

Type checks and casts

Scala	Java
`obj.isInstanceOf[Cl]`	`obj instanceof Cl`
`obj.asInstanceOf[Cl]`	`(Cl) obj`
`classOf[Cl]`	`Cl.class`

If you want to test whether `p` refers to a `Employee` object, but not a subclass, use

```
if (p.getClass == classOf[Employee])
```

Overriding fields

- a def can only override another def
- a val can only override another val or a parameterless def
- a var can only override an abstract var

	with val	with def	with var
Override val	• Subclass has a private field (with the same name as the superclass field—that's OK). • Getter overrides the superclass getter.	Error	Error
Override def	• Subclass has a private field. • Getter overrides the superclass method.	Like in Java.	A var can override a getter/setter pair. Overriding just a getter is an error.
Override var	Error	Error	Only if the superclass var is abstract

Uniform access principle

[B. Meyer, *Object-Oriented Software Construction, 2nd Ed.*, 2000]

All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation.

In Scala, it is possible for a field to override a parameterless method. A client may not even know if he uses a field or calls a method (encapsulation!).

Named and default parameters

```
scala> def speed(distance: Double, time: Double): Double =
           distance / time
speed: (distance: Double, time: Double)Double

scala> speed(100, 20)
res0: Double = 5.0
scala> speed(time = 20, distance = 100)
res1: Double = 5.0
```

```
def printTime(out: java.io.PrintStream = Console.out) =
          out.println("time = " + System.currentTimeMillis())

scala> printTime()
time = 1415144428335

scala> printTime(Console.err)
time = 1415144437279
```

Function literals (= lambdas)

```
scala> (x: Int) => x + 1
res0: Int => Int = <function1>

scala> val numbers = List(-11,-10,5,0,5,10)
numbers: List[Int] = List(-11, -10, 5, 0, 5, 10)

scala> numbers.map((x: Int) => x + 1)
res1: List[Int] = List(-10, -9, 6, 1, 6, 11)
```

```
scala> var addfun = (x: Int) => x + 1
addfun: Int => Int = <function1>

scala> numbers.map(addfun)
res1: List[Int] = List(-10, -9, 6, 1, 6, 11)
```

Function literals, short form

```
scala> numbers.map(x => x+1)
res1: List[Int] = List(-22, -20, 10, 0, 10, 20)

scala> var addfun = x => x+1
<console>:7: error: missing parameter type
       var addfun = x => x+1
```

Strange?

Not really.
Target typing: If a function literal is used immediately, the compiler can do type inference.

Pattern matching (keywords: match, case)

```
expression match {
    case pattern1 => expression1
    case pattern2 => expression2
    ...
}
```

```
scala> val month : Int = 8
month: Int = 8

scala> val monthString: String = month match {
    case 1 => "January"
    case 2 => "February"
    case 3 => "March"
    case 4 => "April"
    case 5 => "May"
    case 6 => "June"
    case 7 => "July"
    case 8 => "August"
}
monthString: String = August
```

```
val cmd = "stop"
cmd match {
  case "start" | "go" => println("starting")
  case "stop" | "quit" | "exit" => println("stopping")
  case _ => println("doing nothing")
}
```

But it's much more than switch / case …

Cases with a guard (i.e., if)

```
i match {
  case a if 0 to 9 contains a => println("0-9 range: " + a)
  case b if 10 to 19 contains b => println("10-19 range: " + b)
  case c if 20 to 29 contains c => println("20-29 range: " + c)
  case _ => println("Hmmm...")
}
```

```
num match {
  case x if x == 1 => println("one, a lonely number")
  case x if (x == 2 || x == 3) => println(x)
  case _ => println("some other value")
}
```


https://www.safaribooksonline.com/library/view/scala-cookbook/9781449340292/ch03s14.html

Case classes and pattern matching

```
abstract class Publication
case class Novel(title: String, author: String) extends
    Publication
case class Anthology(title: String) extends
    Publication

val a = Anthology("Great Poems")
val b = Novel("The Castle", "F. Kafka")
val books: List[Publication] = List(a,b)

scala> for (book <- books) {
    val description = book match {
        case Anthology(title) => title
        case Novel(title, author) => title + " by " + author
     }
     println(description)
}
Great Poems
The Castle by F. Kafka
```

Pattern matching, more examples

(based on books: List[Publication] = List(…))

```
scala> for (book <- books) {
    val description = book match { // order matters!
        case Novel(title, "F. Kafka") => title + " by Kafka"
        case Novel(title, author) => title + " by " + author
        case other => other.title
    }
    println(description)
}
```

```
scala> for (book <- books) {
    val description = book match { // order matters!
        case Novel(title, _) => title
        case Anthology(title) => title
        case _ => "unknown publication type"
    }
    println(description)
}
```

Pattern matching, other examples

```scala
def matchPerson(person: Person): String = person match {
  // Then you specify the patterns:
  case Person("George", number) => "We found George! His number is " + number
  case Person("Kate", number)   => "We found Kate! Her number is " + number
  case Person(name, number)     => "We matched someone : " + name + ", phone : " + numbe
}
```

```scala
def matchEverything(obj: Any): String = obj match {
  // You can match values:
  case "Hello world" => "Got the string Hello world"

  // You can match by type:
  case x: Double => "Got a Double: " + x

  // You can specify conditions:
  case x: Int if x > 10000 => "Got a pretty big number!"

  // You can match case classes as before:
  case Person(name, number) => s"Got contact info for $name!"

  // You can match regular expressions:
  case email(name, domain) => s"Got email address $name@$domain"

  // You can match tuples:
  case (a: Int, b: Double, c: String) => s"Got a tuple: $a, $b, $c"

  // You can match data structures:
  case List(1, b, c) => s"Got a list with three elements and starts with 1: 1, $b, $c"

  // You can nest patterns:
  case List(List((1, 2, "YAY"))) => "Got a list of list of tuple"
}
```

Type checks with pattern matching

```
p match {
  case s: Employee => ... // Process s as a Employee
  case _ => ... // p wasn't a Employee
}
```

What does it print? Choose 1, 2, 3 or 4

```
def objFromJava: Object = "string"
def stringFromJava: String = null
def printLengthIfString(a: AnyRef) {
    a match {
        case s: String => println("String of length " + s.length)
        case _ => println("Not a string")
    }
}
printLengthIfString(objFromJava)
printLengthIfString(stringFromJava)
```

1. Prints:
   Not a string
   String of length 0
2. The first prints:
   Not a string and the second throws a NullPointerException
3. Prints:
   String of length 6
   Not a string
4. The first prints:
   String of length 6 and the second throws a NullPointerException

What does it print? Solution

The correct answer is 3:
String of length 6
Not a string

In Scala, matching by type is implemented via the isInstanceOf method, which performs checks based on runtime type.

Recall also that null is of type Null, and Null is at the bottom of the type hierarchy (just above Nothing).

A. Phillips, N. Šerifović, *Scala Puzzlers*, 2014 (Puzzler 27)

A pattern can be passed to any method that takes a single parameter function

```
list.filter(a => a match {
    case s: String => true
    case _ => false
})
```

```
list.filter {
    case s: String => true
    case _ => false
}
```

Sealed classes

A sealed class can't have any subclasses defined outside the same source file.

Note it's safe to use pattern matching on sealed classes: nobody can define additional subclasses later, creating unknown match cases.

```
sealed abstract class Publication
case class Novel(title: String, author: String) extends
    Publication
case class Anthology(title: String) extends
    Publication
```

Matching lists

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

scala> x match {
         case List(a,_*) => "head of list is "+a
       }
head of list is 1

scala> x match {
          case List(_,_,_) => "three element list"
      }
res40: String = three element list

scala> x match {
          case List(_,_) => "three element list"
      }
scala.MatchError: List(1, 2, 3) ...
```

Matching lists, cont'd

```
val items = List("apple", "orange", "pear", "nut")

val result = items match {
    case List("apple") => "Just apples"
    case List("apple", "orange", otherFruits @ _*) =>
          "apples, oranges, and " + otherFruits
    case _ => ()
}
```

result: Any = apples, oranges, and List(pear, nut)

Sorting by multiple fields

```
case class Person(first: String, middle: String, last: String)

val personList = List( Person("John", "A.", "Smith"),
                       Person("Steve", "J.", "Hathaway"),
                       Person("Bill", "W.", "Smith"),
                       Person("Philip", "A.", "Drake") )
```

We want to sort personList by:

- last name's length, descendingly,
- last name (lex),
- first name (lex).

```
val result = personList.sortBy{ case Person(f, m, l) => (-l.length, l, f) }

// List(Person(Steve,J.,Hathaway), Person(Philip,A.,Drake),
        Person(Bill,W.,Smith), Person(John,A.,Smith))
```

Sorting by multiple fields, cont'd

```
val r = new scala.util.Random(123456L)
val data = (1 to 5).map(x => (r.nextInt(2) + 1, r.nextDouble))
// data: scala.collection.immutable.IndexedSeq[(Int, Double)] =
// Vector((1,0.882072120762085 4), (1,0.252255440793930 5),
//        (2,0.306438700458979 8), (2,0.536064507919613 9),
(1,0.146892585205327))

val dataSorted = data.sortWith { case ((n1, m1), (n2, m2))  =>
    (n1 < n2) || (n1 == n2 && m1 < m2)
}
```


http://reactive.xploregroup.be/blog/8/Sorting-on-multiple-fields-and-Tuples