

# *Zaawansowane programowanie obiektowe*

## **Lecture 6 (Scala)**



Szymon Grabowski  
sgrabow@kis.p.lodz.pl  
<http://szgrabowski.kis.p.lodz.pl/zpo18/>

Based mostly on:

C. Horstmann, *Scala for the Impatient*, 2012, 2017

M. Odersky et al., *Programming in Scala*, 3rd Ed., 2016

and <http://www.cs.columbia.edu/~bauer/cs3101-2/>

Łódź, 2018



# Install

[www.scala-lang.org/downloads](http://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
```

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
```

Type inference:

from the init with e.g. 5 Scala 'knows' that our value is of type Int.

Same with Double, String, Vector...

## Some more types

Boolean: true | false

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

Vector – a container similar to `java.util.ArrayList`,  
but immutable

```
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
```

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

```
5e: Unit = ()
```

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

```
// Expressions.scala
```

```
val c = {  
    val i1 = 2  
    val j1 = 4/i1  
    i1 * j1  
}
```

Watch this!

```
}  
println(c)  
/* Output:  
4  
*/
```

## 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`?

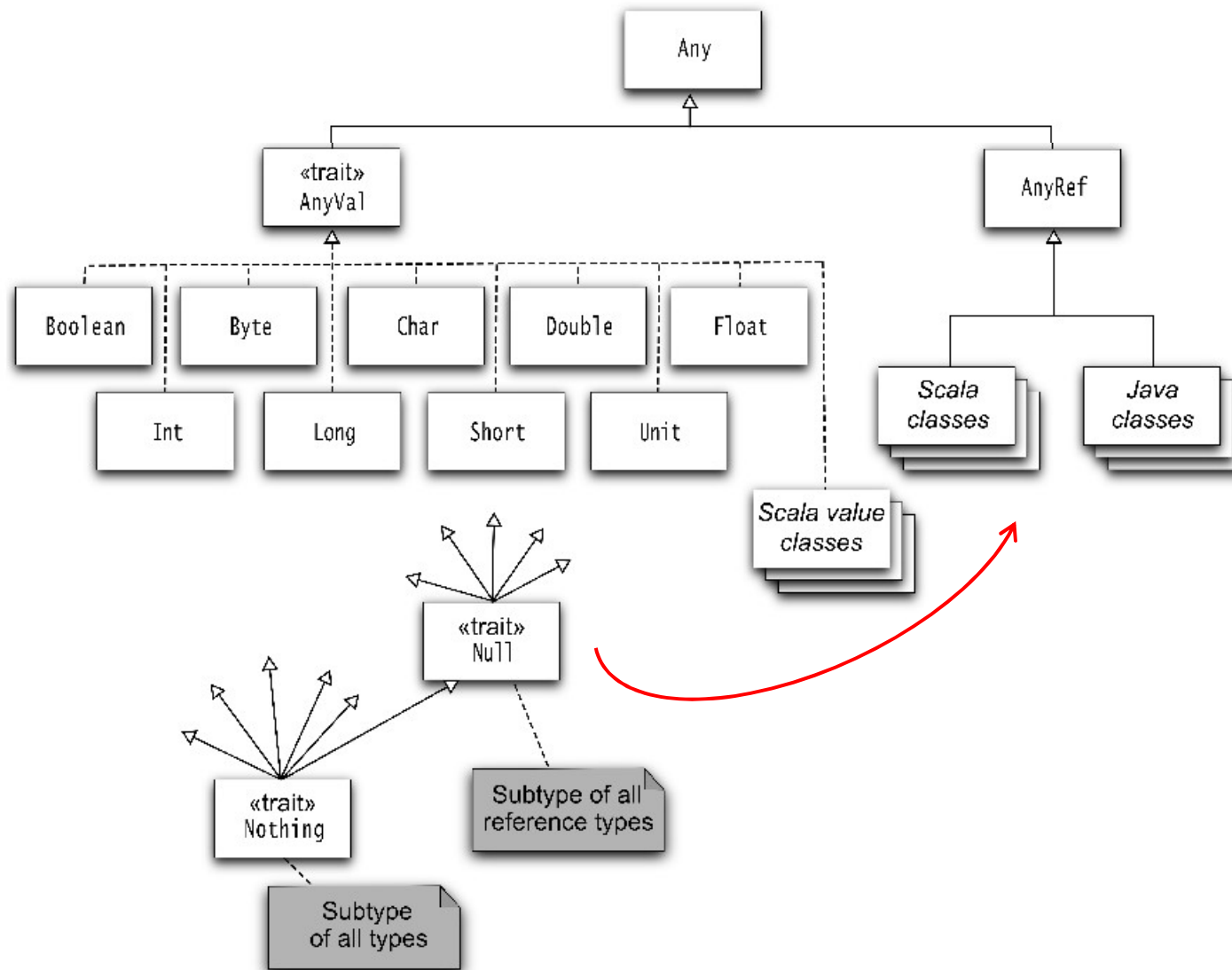
No.

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

```
print(check1(x), check2(x))
```

```
// (true,false)
```

# Type hierarchy



From: [C. Horstmann, *Scala for the Impatient*, 2nd Ed.]

## 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, integers etc. are objects, all objects belong to a class, and they interact through methods. The generated JVM bytecode 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** (at least on the JVM).

<http://stackoverflow.com/questions/2335319/what-are-the-relationships-between-any-anyval-anyref-object-and-how-do-they-m><sup>12</sup>

## 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



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

# 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`  $\rightarrow$  8

but `2.+(3).*(2)`  $\rightarrow$  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
```

## 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)
```

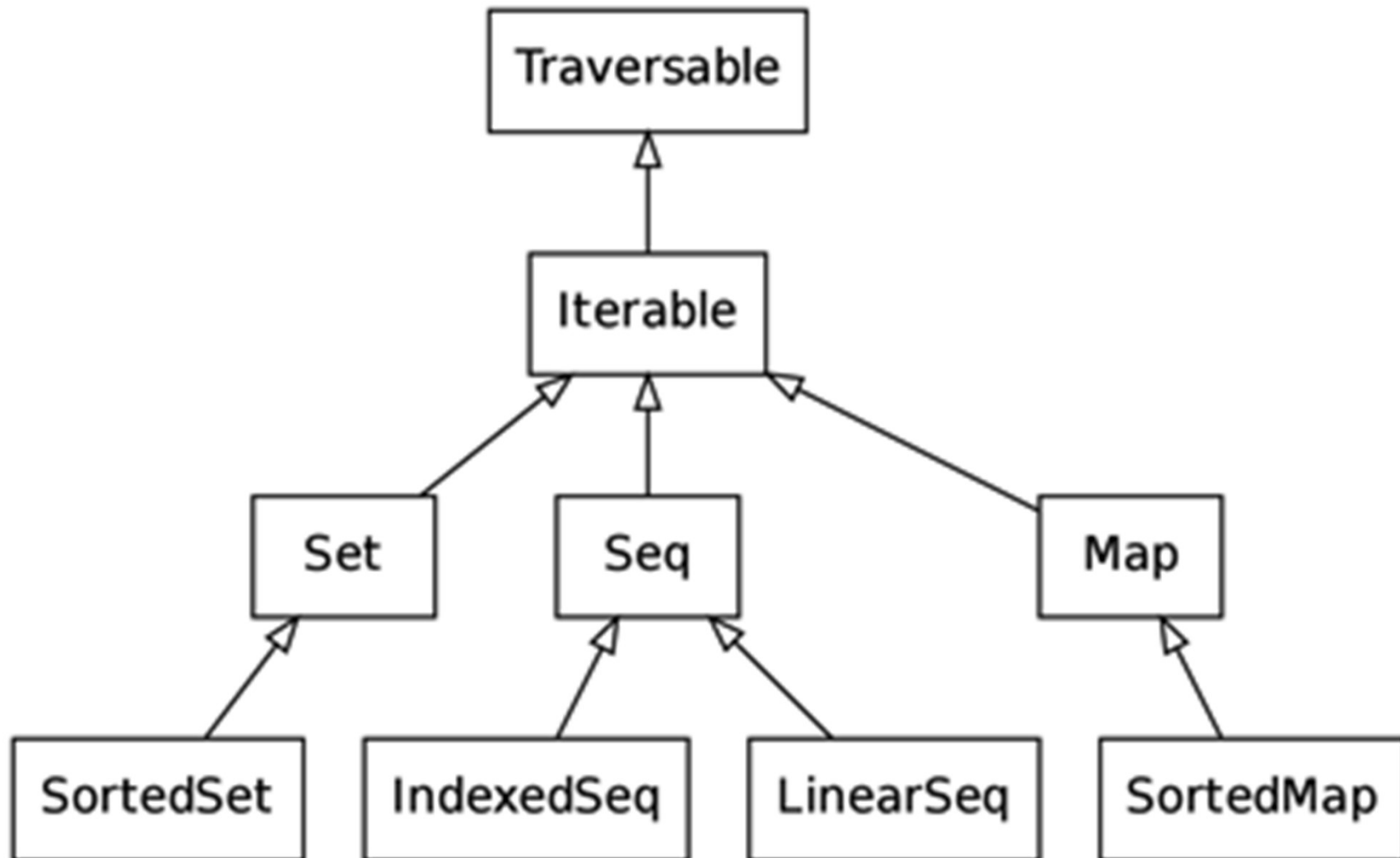
# Collections in Scala

(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



## Traversable / Iterable / Seq

**Traversable**: its main method is *foreach*, so it allows to do something for each element of the collection.

An **Iterable** can create an Iterator based on which *foreach* can be implemented. This defines some order of the elements, although that order might change for every Iterator.

**Seq**(uence) is an Iterable where the order of elements is fixed. Therefore, its elements have indexes.

**LinearSeq** and **IndexedSeq** do not add any new operations, but each offers different performance characteristics.

LinearSeq (e.g., `scala.collection.immutable.List`) has efficient *head / tail* operations, whereas an indexed sequence (e.g., `scala.Array`, `scala.collection.mutable.ArrayBuffer`) has efficient *apply*, *length*, and (if mutable) *update* operations.

# 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.

## import

Basically similar to Java, yet a different wildcard: \* → \_

e.g.

```
import scala.actors._
```

```
import java.io._
```

If the first segment is “scala”, we can omit it:

```
import actors._
```

(But don't use scala.actors, it's deprecated since v2.10. 😊)

Or: import x.y // say, this package contains class C

```
val c = y.C
```

The import is more flexible than in Java:

```
import scala.util.{Try, Success, Failure}
```

```
import java.util.{Map => JMap, List => JList} // import rename
```

Can use import anywhere in the code!

## Imported by default

```
import java.lang._  
import scala._  
import Predef._
```

Unlike all other imports, `import scala._` overrides the preceding import!

E.g. `scala.StringBuilder` overrides `java.lang.StringBuilder`.

Thx to the default imports, you can write  
e.g. `collection.immutable.SortedMap`  
rather than `scala.collection.immutable.SortedMap`.

## Interoperating with Java

If you import the `implicit` conversion methods from `scala.collection.JavaConversions`, then you can use e.g. Scala buffers in your code, and they automatically get wrapped into Java lists when calling a Java method.

```
import scala.collection.JavaConversions.bufferAsJavaList
import scala.collection.mutable.ArrayBuffer
val command = ArrayBuffer("ls", "-al", "/home/ray")
```

```
// calling java.util.ProcessBuilder's constructor
// which works with List<String>!
val pb = new ProcessBuilder(command)
```

```
import scala.collection.JavaConversions.asScalaBuffer
import scala.collection.mutable.Buffer
val cmd: Buffer[String] = pb.command() // Java to Scala
// can't use ArrayBuffer - the wrapped object is
// only guaranteed to be a Buffer
```

## More on 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`).<sup>42</sup>

# 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

```
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 nice 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")))
}
```

## 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.

## Find the length of the longest line in a file

```
import scala.io.Source
```

```
...
```

```
val lines = Source.fromFile(args(0)).getLines()
```

```
// getLines() returns an Iterator[String]
```

```
var maxWidth = 0
```

```
for (line <- lines) maxWidth = maxWidth.max(line.length)
```

```
// or:
```

```
val longestLine = lines.reduceLeft( (a, b) => if (a.length > b.length) a  
  else b )
```

```
val maxWidth = longestLine.length
```

```
// or:
```

```
val lineLengths = lines map { _.length }
```

```
val maxWidth = lineLengths.reduceLeft( (a, b) => if (a > b) a else b )
```



## zipWithIndex

In Python, `enumerate` is a useful generator:

```
li = ["a", "b", "xyz"]  
enumerate(li) # <enumerate object at 0x0000000002512E10>  
list(enumerate(li)) # [(0, 'a'), (1, 'b'), (2, 'xyz')]
```

Common app:

```
for i, line in enumerate(open(sys.argv[1])):  
    print i, line
```

In Scala, we can use `zipWithIndex` from `Iterable` trait:

```
for ((line, i) <- Source.fromFile(args(0)).getLines().zipWithIndex) {  
    println(i, line)  
}
```

## 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)  
  
}
```

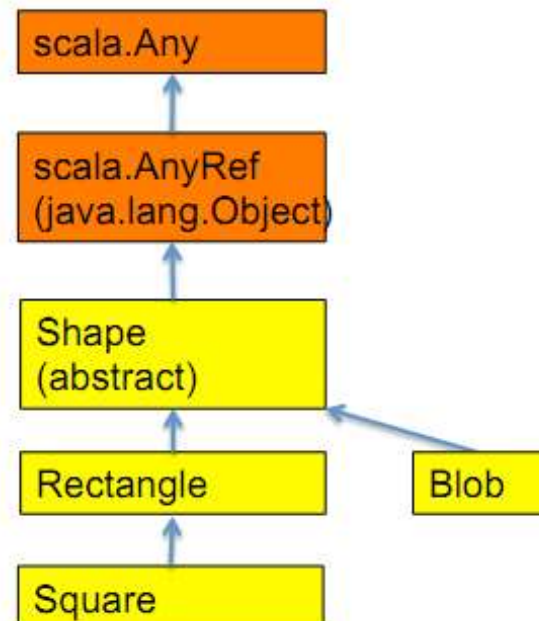
```
scala> new Rational(2/4)  
Res0: Rational(1/2)
```

# 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.

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)
```

# 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
<code>obj.isInstanceOf[C1]</code>	<code>obj instanceof C1</code>
<code>obj.asInstanceOf[C1]</code>	<code>(C1) obj</code>
<code>classOf[C1]</code>	<code>C1.class</code>

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	<ul style="list-style-type: none"><li>• Subclass has a private field (with the same name as the superclass field—that's OK).</li><li>• Getter overrides the superclass getter.</li></ul>	Error	Error
Override def	<ul style="list-style-type: none"><li>• Subclass has a private field.</li><li>• Getter overrides the superclass method.</li></ul>	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!).



## 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")  
}
```

## 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

```
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 : " + number  
}
```

```
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  
}
```

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)
```

## Traits are similar to Java 8 interfaces

That is, traits are interfaces with possible implementations (=concrete methods).

As opposed to Java 8, **traits also support state** (i.e., may have variables).

Traits can reference the implementing class and place restrictions on which type can mix-in them.

Traits can also override methods from Object, e.g.:

```
trait MyToString {  
  override def toString = s"[${super.toString}]"  
}
```

## Using a concrete (non-abstract) trait

```
trait ConsoleLogger {  
  def log(msg: String) { println(msg) }  
}
```

```
class SavingsAccount extends Account with ConsoleLogger {  
  def withdraw(amount: Double) {  
    if (amount > balance) log("Insufficient funds")  
    else balance -= amount  
  }  
  ...  
}
```

Note: when a trait changes,  
all classes that mix in that trait must be recompiled.

## Objects with traits

```
trait Logged { def log(msg: String) {} } // A dummy trait
```

```
class SavingsAccount extends Account with Logged {  
  def withdraw(amount: Double) {  
    if (amount > balance) log("Insufficient funds")  
    else ...  
  }  
  ...  
}
```

```
trait ConsoleLogger extends Logged {  
  override def log(msg: String) { println(msg) }  
}
```

```
val acct = new SavingsAccount with ConsoleLogger
```

When calling log on acct, log method of ConsoleLogger executes.

Of course, another object can add in a different trait:

```
val acct2 = new SavingsAccount with FileLogger
```

## Layered traits

With traits, `super.log` does NOT have the same meaning as it does with classes.

Instead, `super.log` calls the next trait in the trait hierarchy, which depends on the order in which the traits are added. Generally, **traits are processed starting with the last one.**

### Example:

```
val acct1 = new SavingsAccount with ConsoleLogger with  
    TimestampLogger with ShortLogger  
val acct2 = new SavingsAccount with ConsoleLogger with  
    ShortLogger with TimestampLogger  
// what is the result of calling log on acct1 and acct2?
```

```
Sun Feb 06 17:45:45 ICT 2011 Insufficient... // acct1's log  
Sun Feb 06 1... // acct2's log
```

## Trait construction order

Just like classes, traits can have constructors, made up of field initializations and other statements in the trait's body. For example,

```
trait FileLogger extends Logger {  
  val out = new PrintWriter("app.log") // Part of the trait's constructor  
  out.println("# " + new Date().toString) // Also part of the constructor  
  
  def log(msg: String) { out.println(msg); out.flush() }  
}
```

These statements are executed during construction of any object incorporating the trait. Constructors execute in the following order:

- The superclass constructor is called first.
- Trait constructors are executed after the superclass constructor but before the class constructor.
- Traits are constructed left-to-right.
- Within each trait, the parents get constructed first.
- If multiple traits share a common parent, and that parent has already been constructed, it is not constructed again.
- After all traits are constructed, the subclass is constructed.

## object (keyword)

Use objects (rather than classes) for singletons and utility methods.

A class can have a companion object with the same name.

Objects can extend classes or traits.

The **apply** method of an object is usually used for **constructing new instances** of the companion class.

To avoid the main method, use an object that **extends the App trait**.



## Companion object

```
class Account {  
  val id = Account.newUniqueNumber()  
  private var balance = 0.0  
  def deposit(amount: Double) { balance += amount }  
  ...  
}
```

```
object Account { // The companion object  
  private var lastNumber = 0  
  private def newUniqueNumber() =  
    { lastNumber += 1; lastNumber }  
}
```

The class and its companion object can access each other's private features.

They must be located **in the same source file.**

## apply method

Objects often have `apply()` method.

The `apply` method is called for expressions of the form `Object(arg1, ..., argN)`.

```
val arr = Array("a", "few", "strings") // no new!
```

Why? Simplified syntax:

```
Array(Array(1, 3), Array(2, 5))
```

vs

```
new Array(new Array(1, 3), new Array(2, 5))
```

## Beware!

Don't confuse `Array(10)` and `new Array(10)`.

The first expression calls `apply(10)`, yielding an `Array[Int]` with a single element, the integer 10.

The second one invokes the constructor `this(10)`.

The result is an `Array[Nothing]` with 10 null elements.

Type `Nothing` is used rarely, yet it has its use cases.

One example is the return type for methods which never return normally.

One example is method `error` in `scala.sys`, which always throws an exception.

## Case classes

Case classes can be pattern matched.

Case classes automatically define hashCode, equals, toString, copy.

Case classes automatically define getter (\*) methods for the constructor arguments.

(\*) and setter methods, if "var" is specified in the constructor argument

When constructing a case class, **a class as well as its companion object are created.**

The companion object implements the **apply** method that can be used as a factory method (no "new" then needed).

## Case classes & pattern matching, another example

```
import Shape._

trait Shape

case class Rectangle(base: Double, height: Double) extends Shape
case class Circle(radius: Double) extends Shape

object Shape {

  def area(shape: Shape): Double = shape match {
    case Rectangle(b, h) => b * h
    case Circle(r) => r * r * Math.PI
  }

  val rectangle: Shape = Rectangle(4, 5)
  val circle: Shape = Circle(4)

  val rectangleArea = area(rectangle)
  val circleArea = area(circle)
}
```

`equals` (or: `==`), `hashCode` (or: `##`), `eq`

In Scala, the `##` method is (almost) **equivalent to** the Java's `hashCode` method; it's null-safe (yields 0 for null instead of throwing an exception) and the `==` method is **equivalent to `equals`** in Java.

To check if two references point to the same object, use the method `eq` in Scala.

In Scala, when calling the `equals` or `hashCode` method it's better to use `##` and `==`. These methods provide additional support for value types. But the `equals` and `hashCode` method are used when overriding the behavior. This split provides better runtime consistency and still retains Java interoperability.

## On the == method

The == method is defined in AnyRef class.  
It first checks for null values, and then calls the equals method on the first object (i.e., this) to see if the two objects are equal.  
As a result, you don't have to check for null values when comparing strings.

```
scala> val s1 = null  
s1: Null = null
```

```
scala> s1 == s2  
res0: Boolean = true
```

```
scala> val s2: String = null  
s2: String = null
```

```
scala> s1 == s3  
res1: Boolean = false
```

```
scala> val s3 = "abc"  
s3: String = abc
```

```
scala> s2 == s3  
res2: Boolean = false
```

## More on == and equals

When you implement a class, you should consider overriding the equals method (if you do, don't forget to override hashCode as well).

Usually equals is defined as final (as it's very difficult to correctly extend equality – same problem with symmetry as in Java).

Be sure to **define equals with parameter type Any**.

Don't write:

```
final def equals(other: Item) = { ... } // Another method!
```

**Don't supply an == method** (you can't override == defined in AnyRef), but again the code below would define a different method!

```
final def ==(other: Item) = { ... } // Don't!
```

But: **use == (rather than equals)** for checking, it's safer and also works for value types.



???

(yes, it's a method)

```
package scala
```

```
object Predef {
```

```
  ...
```

```
  def ??? : Nothing = throw new NotImplementedError
```

```
  ...
```

```
}
```

As ??? returns Nothing, it can be called by **any** other function!

Typical example, a method declared but not yet defined:

```
/** @return (mean, standard_deviation) */
```

```
def mean_stdDev(data: Seq[Double]): (Double, Double) = ???
```

## What does it do?

```
object Main extends App {  
  val nums = """"(\d+) (\d+) (\d+)"""".r  
  io.Source.stdin.getLines().drop(1).map {  
    case nums(c, k, w) => c.toInt * w.toInt <= k.toInt  
  }.map(b => if (b) "yes" else "no").foreach(println)  
}
```

From the scala doc on regexes:

```
val date = """"(\d\d\d\d)-(\d\d)-(\d\d)"""".r  
"2004-01-20" match {  
  case date(year, month, day) => s"$year was a good year for PLs."  
  // or: case date(year, _) => ...  
}
```

## printf-style in print / println

```
scala> println(f"$name is $age years old, and weighs $weight%.2f  
pounds.")
```

Fred is 33 years old, and weighs 200.00 pounds.

**f string interpolator** allows to use printf style  
formatting specifiers inside strings

## Funny string methods (1/2)

**StringOps** — this class serves as a wrapper providing Strings with all the operations found in indexed sequences. Where needed, instances of String object are implicitly converted into this class.

### **distinct: String**

Builds a new sequence from this sequence without any duplicate elements. Returns a new sequence which contains the first occurrence of every element of this sequence.

```
assert("baca".distinct.sorted == "abc".distinct.sorted)
```

### **grouped(size: Int): Iterator[String]**

Partitions elements in fixed size iterable collections.

```
print("abcdefg".grouped(3).toList)  
// List(abc, def, g)
```

## Funny string methods (2/2)

**combinations(n: Int): Iterator[String]**

Iterates over unique combinations, with the elements taken in order.

```
for(s <- "take".combinations(2)) print(s + " ")
```

```
// ta tk te ak ae ke
```

```
for(s <- "cocoa".combinations(3)) print(s + " ")
```

```
// cco cca coo coa ooa
```

**permutations: Iterator[String]**

Iterates over distinct permutations.

**takeWhile(p: (Char) => Boolean): String**

Takes longest prefix of elements that satisfy a predicate.

```
for((s, i) <- "abcdefgh".sliding(4) zip (1 to s.length).toIterator)
  println(" " * i + s)
```

## try, catch, finally

```
try {  
    something  
} catch {  
    case ex: IOException => // handle  
    case ex: FileNotFoundException =>  
        // handle  
} finally { doStuff }
```

## 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)  
    }  
}
```

## Loan pattern in general

Ensures that a resource is deterministically disposed of once it goes out of scope.

```
def withResource[A](f : Resource => A) : A = {  
  val r = getResource() // Replace with the code to acquire the resource  
  try {  
    f(r)  
  } finally {  
    r.dispose()  
  }  
}
```

The client code:

```
withResource{ r =>  
  // do stuff with r....  
}
```



## Extract an integer from a string

```
var s = "15"; var n = s.toInt // it works, but...
```

```
s = "bug"; n = s.toInt; println(n)  
// java.lang.NumberFormatException
```

### Standard solution:

```
def toInt(s: String): Option[Int] = {  
  try {  
    Some(s.toInt)  
  } catch {  
    case e: Exception => None  
  }  
}
```

```
val x = toInt("bug") // x: Option[Int] = None  
print(x.getOrElse(0))
```

## Extract an integer from a string, simpler

```
import scala.util.Try // since Scala 2.10

val x = "bla"
println(Try(x.toInt).toOption.getOrElse(0))
```

There are **two types of Try**: If an instance of Try[A] represents a successful computation, it is an instance of **Success[A]**, simply wrapping a value of type A.

If, on the other hand, it represents a computation in which an error has occurred, it is an instance of **Failure[A]**, wrapping a Throwable, i.e. an exception.

<http://danielwestheide.com/blog/2012/12/26/the-neophytes-guide-to-scala-part-6-error-handling-with-try.html>

## flatten and flatMap

flatten collapses one level of nested structure.

```
scala> List(List(1, 2), List(3, 4)).flatten  
res0: List[Int] = List(1, 2, 3, 4)
```

flatMap takes a function that works on the nested lists  
and then concatenates the results back together

```
scala> val nestedNumbers = List(List(1, 2), List(3, 4))  
nestedNumbers: List[List[Int]] = List(List(1, 2), List(3, 4))  
  
scala> nestedNumbers.flatMap(x => x.map(_ * 2))  
res0: List[Int] = List(2, 4, 6, 8)
```

## File processing

- `Source.fromFile(...).getLines.toArray` yields all lines of a file.
- `Source.fromFile(...).mkString` yields the file contents as a string.
- To convert a string into a number, use the `toInt` or `toDouble` method.
- Use the Java `PrintWriter` to write text files.
- `"regex".r` is a `Regex` object.
- Use `""...""` if your regular expression contains backslashes or quotes.
- If a regex pattern has groups, you can extract their contents using the syntax `for (regex(var1, ..., varn) <- string).`

```
import scala.io.Source
val source = Source.fromFile("myfile.txt", "UTF-8")
// The first argument can be a string or a java.io.File
// You can omit the encoding if you know that the file uses
// the default platform encoding
val lineIterator = source.getLines
```

## Native XML support

```
scala> <ul>{(1 to 3).map(i => <li>{i}</li>)}</ul>
```

```
res0: scala.xml.Elem = <ul><li>1</li><li>2</li><li>3</li></ul>
```

```
def now = System.currentTimeMillis.toString
```

```
<b time={now}>Hello World</b>
```

```
res0: scala.xml.Elem = <b time="1448189090022">Hello  
World</b>
```

```
def proc(node: scala.xml.Node): String =  
  node match {  
    case <a>{contents}</a> => "It's an a: " + contents  
    case <b>{contents}</b> => "It's a b: " + contents  
    case _ => "It's something else."  
  }
```

```
scala> proc(<a>apple</a>)  
res16: String = It's an a: apple  
scala> proc(<b>banana</b>)  
res17: String = It's a b: banana  
scala> proc(<c>cherry</c>)  
res18: String = It's something else.
```

## Extracting XML nodes and attributes

```
val weather =
<rss>
  <channel>
    <title>Yahoo! Weather - Boulder, CO</title>
    <item>
      <title>Conditions for Boulder, CO at 2:54 pm MST</title>
      <forecast day="Thu" date="10 Nov 2011" low="37" high="58" text="Partly Cloudy"
        code="29" />
    </item>
  </channel>
</rss>
```

```
scala> val forecast = weather \ "channel" \ "item" \ "forecast"
forecast: scala.xml.NodeSeq = NodeSeq(<forecast day="Thu"
  date="10 Nov 2011" low="37" high="58" text="Partly Cloudy" code="29"/>)
```

```
val day = forecast \ "@day"      // Thu
val date = forecast \ "@date"    // 10 Nov 2011
val low = forecast \ "@low"      // 37
val high = forecast \ "@high"    // 58
val text = forecast \ "@text"    // Partly Cloudy
```



## Higher-order functions

Higher-order functions are functions that take other functions as parameters, or whose result is a function.

```
def use(f: Int => String, v: Int) = f(v)

class Decorator(left: String, right: String) {
  def layout[A](x: A) = left + x.toString() + right
}

val decorator = new Decorator("[", "]")
println(use(decorator.layout, 7))
```

Output: [7]

Note that method `decorator.layout` is a polymorphic method (i.e. it abstracts over some of its signature types) and the Scala compiler has to instantiate its method type first appropriately.

## Higher-order functions, another example

```
def my_map(lst : List[Int], fun : Int => Int) : List[Int] =  
  for (l <- lst) yield fun(l)  
  
val numbers = List(2,3,4,5)  
  
def addone(n : Int) = n + 1  
  
scala> my_map(numbers, addone)  
res0: List[Int] = List(3, 4, 5, 6)
```



## New control structures

Higher order functions allow to write new control structures.

```
def twice(op: Double => Double)(x: Double) = op(op(x))  
→ twice: (op: Double => Double)(x: Double)Double
```

```
twice ( _ + 2)(3)  
→ res16: Double = 7.0
```

```
twice {  
  x => x + 2  
} (3)  
→ res17: Double = 7.0
```

## Generic classes

```
class Stack[T] {  
  var elems: List[T] = Nil  
  def push(x: T) { elems = x :: elems }  
  def top: T = elems.head  
  def pop() { elems = elems.tail }  
}  
  
object GenericsTest extends App {  
  val stack = new Stack[Int]  
  stack.push(1)  
  stack.push('a')  
  println(stack.top)  
  stack.pop()  
  println(stack.top)  
}
```

## Covariance (1/2)

In Java, arrays are covariant, but not in Scala.  
It means that e.g. array of ints in Java is an array of Objects,  
but not in Scala (use Ints vs Any here).

```
scala> val a1 = Array("abc")  
a1: Array[String] = Array(abc)
```

```
scala> val a2: Array[Any] = a1  
<console>:12: error: type mismatch;  
found   : Array[String]  
required: Array[Any]
```

Note: **String** <: **Any**, but class Array is invariant in type T.

## Covariance (2/2)

Covariant arrays are unsafe. But if they are really needed in Scala (e.g. for using some Java classes/methods), we can use casting:

```
scala> val a2: Array[Object] = a1.asInstanceOf[Array[Object]]  
a2: Array[Object] = Array(abc)
```

The cast is always legal at compile-time, and it will always succeed at runtime, because the JVM's underlying run-time model treats arrays as covariant, just as Java the language does.

Create an own **covariant** generic class X:

```
class X[+T] { ... } // +T means that subtyping is covariant
```

E.g. Vector[T] is a covariant class.

Therefore, Vector[Dog] is a subtype of Vector[Animal] (if Dog <: Animal).

# Invariance

Arrays in Scala are invariant.


It means: no conversion wider to narrower,  
nor narrower to wider may be performed on the class.

Invariant generic class X: `class X[T] { ... }`

What about Set?

Scala Standard Library 2.12.0

Search



scala.collection.immutable  
**Set**

Companion object Set

```
trait Set[A] extends Iterable[A] with collection.Set[A] with  
GenericSetTemplate[A, Set] with SetLike[A, Set[A]] with Parallelizable[A,  
ParSet[A]]
```

A generic trait for immutable sets.

A set is a collection that contains no duplicate elements.

## Contravariance

Contravariance is the opposite of covariance.

**Contravariant** class X: **class X[-T]** { ... }

It means that if type A is a supertype of B, then X[B] will be a supertype of X[A] (strange??).

Use case: Function1[-T1, +R] (one-param function with argument of type T1 and return type R). Why does Function1 need to be contravariant on its the input parameter?

If we have Function1[Dog, Any] then this function should work for Dogs and its subtypes. But not necessarily for Animals.

The reverse conversion works however – if we have a function that works on Animals then this function by design should work on Dogs. Therefore, **Function1[Dog, Any]** **should be a supertype (!) of Function1[Animal, Any]**.

## Lists are covariant (why?)

Because List is immutable (as opposed to Array).

Assume that `scala.Array` is defined as  
`final class Array[+T] extends java.io.Serializable` with  
`java.lang.Cloneable` (in fact, it is `Array[T] !`).

Now smth like that would be possible (i.e., no compile-time error):

```
val arr1: Array[Int] = Array[Int](1, 2)
val arr2: Array[Any] = arr1
arr2(0) = 1.3
```

Yet, we cannot update a list (only create a new List, if needed).

## Currying

Methods may define **multiple parameter lists**.  
When a method is called with a fewer number of parameter lists, then this will yield a function taking the missing parameter lists as its arguments.

```
def modN(n: Int)(x: Int) = ((x % n) == 0)
val t = (modN(3)(10), modN(3)(12))
→ t: (Boolean, Boolean) = (false,true)
```

```
scala> arr.filter(modN(3))
res19: Array[Int] = Array(3, 21)
```

```
scala> arr.filter(modN(2))
res20: Array[Int] = Array(2, 10, 20)
```



## Currying, another example

```
def add(a: Int)(b: Int) = a + b
print(add(3)(4)) // 7
val f = add(5)_ // without _ it doesn't compile!
print(f(10))    // 15
```

## Partially applied functions

```
def f1(a:Int, b:Int) = a + b // this is a standard function  
val x = f1(2, _:Int) // x is a partially applied function  
print(x(3)) // 5
```

## Call-by-name function parameters

By default Scala is call-by-value (like Java): any expression is evaluated before it is passed as a function parameter.

We can force call-by-name by prefixing parameter types with `=>`. Expression passed to parameter is **evaluated every time** it is used.

```
def nano() = {  
    println("Getting nano")  
    System.nanoTime  
}
```

```
def delayed(t: => Long) = { // => indicates a by-name parameter  
    println("In delayed method")  
    println("Param: "+t)  
    t  
}
```

```
println(delayed(nano()))
```

```
In delayed method  
Getting nano  
Param: 4475258994017  
Getting nano  
4475259694720      // different value from Param
```

# Function composition

Example. Let's define a list and 3 lambdas:

```
scala> val foo = 1 to 5 toList
foo: List[Int] = List(1, 2, 3, 4, 5)

scala> val add1 = (x: Int) => x + 1
add1: (Int) => Int = <function1>

scala> val add100 = (x: Int) => x + 100
add100: (Int) => Int = <function1>

scala> val sq = (x: Int) => x * x
sq: (Int) => Int = <function1>
```

We want to apply first add1, then sq, then add100 to all elem. of foo:  
foo map add1 map sq map add100

Alternatively, we can use one “map” only (note the order of functions!):  
foo map (add100 **compose** sq **compose** add1)

## Function composition, example, cont'd

Yet, it's more general to use a list of functions:

```
val fns = List(add1, sq, add100)
```

and then apply them one by one:

```
foo map (fns.reverse reduce (_ compose _))
```

$$(f \text{ compose } g)(x) \Leftrightarrow f(g(x))$$
$$(f \text{ andThen } g)(x) \Leftrightarrow g(f(x))$$

We can thus express the above as:

```
foo map (fns reduce (_ andThen _))
```

Or even simpler:

```
val f = Function.chain(fns)
```

```
foo map f // List(104, 109, 116, 125, 136)
```

## JUnit4 vs ScalaTest

JUnit4 (Scala):

```
@Test(expected = StringIndexOutOfBoundsException.class)
public void charAtTest() {
    "hi".charAt(-1);
}
```

But if we want do smth with the exception, we need try..catch:

```
try {
    "hi".charAt(-1);
    fail();
}
catch (StringIndexOutOfBoundsException e) {
    assertEquals(e.getMessage(),
        "String index out of range: -1");
}
```

## JUnit4 vs ScalaTest, cont'd

In ScalaTest:

```
val caught = intercept[StringIndexOutOfBoundsException] {  
  "hi".charAt(-1)  
}  
assert(caught.getMessage ===  
  "String index out of range: -1")
```

"intercept" here does several things:

1. if there is no exception, fail()
2. if there is an exception, check that it is of the declared type, else fail()
3. return the exception

## What is the result?

```
List(1, 2, 3).map { i => print("* "); i + 1 }  
List(1, 2, 3).map { print("* "); _ + 1 }
```

In REPL:

```
scala> List(1, 2, 3).map { i => print("* "); i + 1 }  
* * * res5: List[Int] = List(2, 3, 4)
```

```
scala> List(1, 2, 3).map { print("* "); _ + 1 }  
* res6: List[Int] = List(2, 3, 4)
```

In the first statement the code block is one expression.

In the second statement – two expressions!

The block is executed and (only) the last expression is passed to the map.

That is, the scope of an anonymous function defined using placeholder syntax stretches only to the expression containing the underscore (\_).



## foldLeft, foldRight, fold

**foldLeft** – similar to reduce in Python

```
(1 to 5).foldLeft(0)(_ + _)
```

```
// or: (1 to 5).foldLeft(0)((res, curr) => res + curr)
```

```
val weightedGrades = Vector[(Double, Double)](  
  (4, 0.2), (4.5, 0.1), (5, 0.1), (4, 0.3), (3, 0.3))  
//or: val weightedGrades: Vector[(Double, Double)] = Vector(  
  (4, 0.2), (4.5, 0.1), (5, 0.1), (4, 0.3), (3, 0.3))  
assert((for(g <- weightedGrades) yield g._2).sum == 1.0)  
val weightedAvg = weightedGrades.foldLeft(0.0)((t, c) => t + c._1 * c._2)
```

**foldRight** – similarly, but goes from right to left

**fold** – arbitrary order (can use a tree structure), can be parallelized

Many examples:

<http://oldfashionedsoftware.com/2009/07/30/lots-and-lots-of-foldleft-examples/>

## foldLeft, foldRight, fold, cont'd

In most cases foldLeft and foldRight give the same results. But...

```
scala> val l = List(1, 2, 3)
```

```
scala> l.foldLeft(List.empty[Int]) { (acc, ele) => ele :: acc }  
res0: List[Int] = List(3, 2, 1)
```

```
scala> l.foldRight(List.empty[Int]) { (ele, acc) => ele :: acc }  
res1: List[Int] = List(1, 2, 3)
```

Note also that

foldLeft is implemented with a loop and local mutable variables.

foldRight is recursive, but not tail recursive, and thus consumes one stack frame per element in the list →  
might stack overflow for long lists.

## foldRight, example

Task: eliminate consecutive duplicates of list elements.

If a list contains repeated elements they should be replaced with a single copy of the element.

The order of the elements should not be changed.

```
compress(List('a', 'a', 'a', 'a', 'b', 'c', 'c', 'a', 'a', 'd', 'e', 'e', 'e'))  
--> List[Char] = List(a, b, c, a, d, e)
```

```
def compress[A](ls: List[A]): List[A] =  
  ls.foldRight(List[A]()) { (h, r) =>  
    if (r.isEmpty || r.head != h) h :: r  
    else r  
  }
```

# Lambda expressions and SAM (single abstract method) types in Scala 2.12 (like in Java 8)

## Old style:

```
scala> trait Increaser {  
    def increase(i: Int): Int  
}  
defined trait Increaser
```

---

```
scala> def increaseOne(increaser: Increaser): Int =  
    increaser.increase(1)  
increaseOne: (increaser: Increaser)Int
```

---

## New style:

```
scala> increaseOne(  
    new Increaser {  
        def increase(i: Int): Int = i + 7  
    }  
)  
res0: Int = 8
```

scala> increaseOne(i => i + 7) // Scala 2.12  
res1: Int = 8

