Języki i środowiska przetwarzania danych rozproszonych

Scala 2

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

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.

Traits, some extra notes

Traits can’t have auxiliary constructors
and their primary constructor can’t have parameters.

The primary constructor (=the body of the trait)
can’t pass values to a super type’s constructor.
As a result, traits can only extend classes with
a zero-param constructor (either primary or auxiliary).

The abstract keyword in front of a trait member
isn’t necessary as the compiler can figure it out
from its lack of initialization
(or lack of implementation for methods).
I.e.: def flyMessage: String
is OK.

http://joelabrahamsson.com/learning-scala-part-seven-traits/

Thin vs rich interfaces

Old Java interfaces are usually thin (=have few methods).
Otherwise, implementing them is burdensome for the clients.

But Java8 interfaces and Scala traits may contain
both non-empty (but possibly abstract!) methods
and method signatures.
If a method is implemented in the trait,
the client doesn't need to override it.

Example: Ordered[T] trait.
Define its compare method (for the class that uses this trait)
and the operators <, <=, >, >= will immediately be available.

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)
```


http://www.devx.com/enterprise/introduction-to-functional-programming-in-scala.html

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

In Scala, the ## method is equivalent to the Java's hashCode method
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.

Joshua D. Suereth, *Scala in Depth*, 2012. Chap. 2.5.

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> val s2: String = null
s2: String = null

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

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

scala> s1 == s3
res1: Boolean = false

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

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

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.

???
(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) = ???
```


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

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
```

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....
}
```


https://wiki.scala-lang.org/display/SYGN/Loan

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)
```


https://twitter.github.io/scala_school/collections.html

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(`var_1`, ...,`var_n`) <- `*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
```


http://alvinalexander.com/scala/xml-parsing-xpath-extract-xml-tag-attributes

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.

http://docs.scala-lang.org/tutorials/tour/higher-order-functions.html

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

```scala
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 }
}
```

```scala
object GenericsTest extends App {
  val stack = new Stack[Int]
  stack.push(1)
  stack.push('a')
  println(stack.top)
  stack.pop()
  println(stack.top)
}
```


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

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].

http://like-a-boss.net/2012/09/17/variance-in-scala.html

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, other examples

```
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


def member[T](x: Seq[T])(a: T) = x contains a
def isVowel(c: Char) = member[Char](
   List('a', 'e', 'i', 'o', 'u', 'y'))(c)
println(isVowel('p'), isVowel('e'))  // (false,true)
```

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)

https://bcomposes.wordpress.com/2011/08/20/fun-with-function-composition-in-scala/

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 compose g)(x)  ⇔  f(g(x))
(f andThen g)(x)  ⇔  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)
```

Scala preconditions

assert (not held → java.lang.AssertionError) –
a predicate which needs to be proven by a static code analyser

require (IllegalArgumentException) – used as a precondition;
blames the caller of the method for violating the condition

ensuring – similar to require, but a post-condition

```
def doublePositive(n: Int): Int = {
  require(n > 0)
  n * 2
} ensuring(n => n >= 0 && n % 2 == 0)
```

Disable assertions: cmd-line option to scalac : -Xdisable-assertions

http://maxondev.com/scala-preconditions-assert-assume-require-ensuring/

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");
}
```


http://www.artima.com/weblogs/viewpost.jsp?thread=252702

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 (_).

A.Phillips, N.Serifovic, *Scala Puzzlers*, 2014.

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
    }
```


http://aperiodic.net/phil/scala/s-99/p08.scala

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)
}
```

Keyword *type*

The most basic application of keyword type is aliasing a complicated type to a shorter name.

```
type ListInt = List[Int]  // not really meaningful…
val x: ListInt = List(2, -5, 1)
```

Or: type FunctorType = (LocalDate, HolidayCalendar, Int, Boolean) => LocalDate

def doSomeThing(f: FunctorType)
will be interpreted by the compiler as
def doSomeThing(f: (LocalDate, HolidayCalendar, Int, Boolean) => LocalDate)

Another example:
```
type Thing[A] = Map[String, Map[String, A]]
val t: Thing[Int] = Map.empty
```

Type members

In Scala, a class can have not only field and method members.
It can have type members.

```
trait Base {
  type T

  def method: T
}

class Implementation extends Base {
  type T = Int

  def method: T = 42
}
```

This looks like plain generics (in a weird form).
But we can do more…

http://stackoverflow.com/questions/19492542/understanding-what-type-keyword-does-in-scala

Lower bounds on types

```
class A {
  type B >: List[Int]  // B has a lower bound of List[Int]
  def f(a : B) = a
}

val x = new A { type B = Traversable[Int] }
// x: A{type B = Traversable[Int]} = $anon$1@650b5efb

x.f(Set(1))

val y = new A { type B = Set[Int] }
// error: overriding type B in class A with bounds >: List[Int];
// type B has incompatible type
```

Upper bounds on types

```
class A {
  type B <: Traversable[Int]
  def count(b : B) = b.sum
}

val x = new A { type B = List[Int] }
println(x.count(List(1, 2)))  // 3

/*
print(x.count(Set(1, 2)))
  //error: type mismatch;
  // found   : scala.collection.immutable.Set[Int]
  // required: this.x.B     (which expands to)  List[Int]
*/

val y = new A { type B = Set[Int] }
println(y.count(Set(1, 2)))  // 3
```

Streams

A stream is smth like a lazy list.

```
scala> val stream = 1 #:: 2 #:: 3 #:: Stream.empty
stream: scala.collection.immutable.Stream[Int] = Stream(1, ?)
```

```
val stream = (1 to 100000000).toStream
print(stream.head)  // 1, same as print(stream(0))
print(stream.take(5).sum) // 15
stream.filter(_ > 100) // scala.collection.immutable.Stream[Int] = Stream(101, ?)
stream.map(_ * 3) // scala.collection.immutable.Stream[Int] = Stream(3, ?)
```

```
def fibFrom(a: Int, b: Int): Stream[Int] = a #:: fibFrom(b, a + b)

print(fibFrom(1, 1).take(7).toList)  // List(1, 1, 2, 3, 5, 8, 11)
```

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
```

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

New style:

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


http://booksites.artima.com/programming_in_scala_3ed/examples/html/ch31.html